FlinkSQL

The SQL dialect for Flink

Khuong Duy Vu

Department of Informatics

ELTE University

This thesis is submitted for the degree of

Master Thesis

May 2015

Chapter 1

Introduction(3 pages)

- BUILDING ALGEBRA SEMANTIC FOR FLINK STREAMING - IMPLEMENTATION

Big Data

Data Streaming

CQL and related work

Features of stream processing languages(Fundamental of Stream Processing book page 110)

Flink and FlinkQL

Structure

[2014] Fundamentals of Stream Processing- Application Design, Systems, and Analytics.pdf [2]

Sliding Window Query Processing over Data Stream by Lukasz Golab

Chapter 2

Data Stream Model(10 pages)

Order

Ordering is rather important in Distributed system, specially Stream Processing System in particular.

In traditional model, there are a single program, one process, one memory space running on one CPU. Programs are written to be executed in an ordered fashion in queue: staring from the beginning, and then going toward the last. In distributed system, programs run in many machines across the network but try to reserve the order of the result. One of easiest way to define "correctness" is to say "it works like it would on a single machine". And that usually means that a) we run the same operations and b) that we run them in the same order - even if there are multiple machines. [Takada]

In theory, they are defined 2 types of orders: total order and partial order.

Definition 2.1 *Paritial order is a binary relation $\leq$ over a set P which is reflexive, anti-symmetric and transitive, i.e., which satisfies for all a, b and c in S (wiki):*

- $a \leq a$ *(reflexivity)*
- *if* $a \leq b$ *and* $b \leq a$ *then* $a = b$ *(antisymmetry)*
- *if* $a \leq b$ *and* $b \leq c$ *then* $a \leq c$ *(transitivity)*

Definition 2.2 *Total order is a binary relation "$\leq$" over a set S which is anti-symmetric and transitive and total. Therefore, total order is a partial order with totality*

- $a \leq b$ *or* $b \leq a$ *(totality)*

Total order "<" is strict on a set S if and only if $(S, <)$ has no non-comparable pairs:

$$\forall x, y \in S \Rightarrow x < y \cup y < x \tag{2.1}$$

In a totally ordered set, every two elements are comparable whereas in a partial ordered set, some pairs of elements are incomparable and hence we do not have the exact order of every item.

The natural state in a distributed system is partial order. Neither the network nor independent nodes make any guarantees about relative order; but at each node, you can observe a local order.

In streaming processing, one may discretize a stream into a discrete stream of windows. Therefore, we need to specify the order of elements to define which elements should be inserted into a given window. Furthermore, data stream involve the temporal order and the positional order [13] in stream.

The **temporal order** is induced by the timestamp of items in a stream. Using the value of timestamp, one can determine whether something happen chronologically before something else. Nevertheless, if some items happen simultaneously, they will have the same timestamp, then it may not be strict. For instance, there are two items with the same timestamp but system is required to take one only, the chosen is non-deterministic between two. Therefore, we may require a strict order.

The **positional order** is a strict order induced by the position of items in stream. Two items may have the same timestamp but one may arrive before the other so that they have different positional orders. The positional order can be defined by arrival order or id of item regardless of explicit timestamp.

//TODO:The temporal order: Order. When I say that time is a source of order, what I mean is that:

we can attach timestamps to unordered events to order them we can use timestamps to enforce a specific ordering of operations or the delivery of messages (for example, by delaying an operation if it arrives out of order) we can use the value of a timestamp to determine whether something happened chronologically before something else.

Time

Time Domain The time domain $\mathbb{T}$ is a discrete, total ordered, countably infinite set of time instants $t \in \mathbb{T}$. We assume that $\mathbb{T}$ is bounded in the past , but not necessarily in the future. For the sake of simplicity, we will assume that the time domain is the domain of non-negative long integer ($\mathbb{T} = \mathbb{N}$) 0,1,2,3,..[6] and totally ordered.

2 type of times: system time t^{sys} (implicit) and application time t^{app} (explicit). These both take values from the time domain $\mathbb{T}$, but carry two different meaning.

Application Timestamps In many case, each item in stream contains an explicit source-assigned timestamp itself. In other words, the timestamp attribute will be a part of the stream schema. Consider a common log format for web application which contains a timestamp specifying when the action is taken place. A log line record user named *pablo* to get an image

```
216.58.209.174 user-identifier pablo [10/Oct/2000:13:55:36 -0700] \\
"GET /image.gif HTTP/1.0" 200 1234
```

Sequence number attribute could represent a timestamp instead.

System Timestamps Even if the item arrives at the system are not equipped with a timestamp, the system assigns a timestamp to each tuple, by default using the system's local clock. While this process generates a raw stream with system timestamps that can be processed like a regular raw stream with application timestamps, the user should be aware that application time and system time are not necessarily synchronized[10]. Since system timestamp is assigned implicitly by system , one may not notice it presence on schema. As we mentions above, time domain is total ordered in local machine , but partial ordered across the system because of possible postpone on processing or asynchronous timestamp at different node.

Both application and system timestamp captures time information but they cary two different meanings. The former is related to the occurrence of the application event(when the event happens), whereas the latter is related to the occurrence of related system(when the corresponding event data arrive at system). Multiple items may have the same application timestamps but they will not arrive in the same order. Therefore, system will assign the different unique system timestamp based on their arrival. Sytem then can believe in the system timestamp as a strict total ordered basis for reasoning about arrival items to perform processing. For example, another log from different users arrive at system:

```
219.53.210.143 user-identifier fabio [10/Oct/2000:13:55:36 -0700] \\
"GET /image.gif HTTP/1.0" 200 1432
```

However, it might arrive after the first log for user *pablo* then system would response *pablo*'s first, instead of *fabio*'s request.

Note: more on Timestamp in Streams [4] page 13

Tuple

A tuple is a finite sequence of atomic values. Each tuple can be defined by a Schema corresponding to a composite type. Tuple can represend a relational tuple , a event or a

record of sensor data and so on. [3]. For instance, the log format in previous example follow a schema:

```
<SourceIP, IdentityType, user, timestamp, action, response, packageSize>
```

A data tuple is the fundamental, or atomic data item, embedded in a data stream and processed by an application. A tuple is similar to a database row in that it has a set of named and typed attributes. Each instance of an attribute is associated with a value[2]. Furthermore, one can consider a tuple as a partial order mapping a finite subset of attribute names to atomic values[13]. A tuple consits of a set of (Atribute x Value) pairs such as $(SourceIP, 219.53.210.143)$

2.1 Stream Model

Based on time and tuple domain, basically, CQL [3] defines a data stream as

Definition 2.3 *A stream $\mathbb{S}$ is a countably continuous and infinite set of elements $< s,t > \in \mathbb{S}$, where s is a tuple belonging to the schema of $\mathbb{S}$ and $t \in \mathbb{T}$ is the timestamp of the element.*

There are several stream definitions varying based on the execution model of systems. On the previous definition, a timestamp attribute can be an non-strict total ordered application timestamp so that system may not rely on it to order the coming tuples to react on them. For this reason, stream can contain an extra physical identifier φ [12] such as increment tuple id to specify its order. The tuple with smaller id mean that they arrive and should be processed before the tuples with bigger id. Another way to identify the order of a tuple item is to separate the concept of application and system timestamp. In SECRET model [Botan et al.], each stream item is composed of a tuple for event contents , an application timestamp, a system timestamp, and a batch-id value. The idea of batch-id is critical to SECRET system we do not mention in the thesis. In short, in practice, we assume that elements of a stream are totally ordered by the system timestamp and physical identifier.

In Apache Flink, for the flexibility, system accepts a user-defined timestamp function $f : \mathbb{TP} \rightarrow \mathbb{T}$ to map a tuple to its application timestamp value. One of the most common scenario is that the function f extract one attribute of Schema and consider it as timestamp value.

Consider the example of temperature sensors, the sensors feed a stream $S(TIME : long, TEMP : int)$ indicating that at the *TIME*, the temperature is *TEMP*. Since $TIME$ attribute has the *long* long, we are able to consider it as a timestamp due to function

$$f : (TIME, TEMP) \rightarrow SEC \tag{2.2}$$

However, we may receive the stream from a different timezone. Thus, the application timestamp must be converted to the current timezone for the sake of data integration. For instance, one acquires the timestamp value (in milliseconds) in next timeszone.

$$f : (TIME, TEMP) \rightarrow TIME + 3600 * 1000 \tag{2.3}$$

I propose the definition of Stream $s :< v >$ extends to $s :< v, t_{app}, t_{sys} >$, with $t_{app} = f(v)$, $t_{app} \in \mathbb{T}$, $t_{sys} \in \mathbb{T}$

For further analysis on execution model in Apache Flink, I propose a extend definition of a data stream as:

Definition 2.4 *A stream $\mathbb{S}$ is a countably infinite set of elements $s \in \mathbb{S}$. Each stream element $s :< v, t_{app}, t_{sys} >$, consists of a relational tuple v conforming to a schema TP, with an optional application time value $t_{app} = f(v) \in \mathbb{T}$, and a timestamp t_{sys} generated automatically by system, due to the event arrival.*

With the partial function f to extract application timestamp from tuple: $f : \mathbb{TP} \rightarrow \mathbb{T}$

Data Stream Properties

In the data stream model, some of all the input data that are to be operated are not available for random access from disk or memory, but rather arrive as one or more continuous data streams. Data streams differ from the conventional stored relation model in several ways[4]

Data Stream may have the following properties [11]:

- They are considered as sequences of records, ordered by arrival time or by another ordered attributed such as generation time which is explicitly specified in schema, that arrive for processing over time instead of being available a priori. Totally order by time.
- They are emitted by a variety of external sources. Therefore, the system has no control over the arrival order or data rate, either within a stream or across multiple streams
- They are produced continually and, therefore, have unbounded, or at least unknown, length. Thus, a DSMS may not know if or when the stream "ends". We may set a time-out waiting for new event. Exceeding the time-out, the stream considerably ends.
- Typically, big volume of data arrives at very high speed so that data need to process on the fly. Once an element from a data stream model has been processed it is discarded or archived. Stream elements cannot be retrieved, unless it is explicitly stored in storage or memory, which typically is small relative to the size of the data stream. [4]

Stream Representations

Base Stream vs. Derived Stream

We distinguish 2 kinds of streams: *base stream*(source stream) and *derived stream*. Base stream stream is produced by the sources whereas derived stream is produced by continuous queries and their operators[3]. For example, [11] page 17 From now on, we give example queries on input stream named *StockTick*[Str] and the schema associated with the incoming tuples includes 4 fields:

- **Symbol**, a string field of maximum length 25 characters that contains the symbol for a stock being traded (e.g. IBM);
- **SourceTimestamp**, a timestamp field containing the time at which the tuple was generated by the source application(timestamp is represented with date time format or long integer);
- **Price**, a double field containing transaction price
- **Quantity**, a integer fields that contains the transaction volume
- **Exchange**, a string field of maximum length 4 that contains the name of Exchange the trade occurred on (e.g. NYSE)

A sample tuple represents the price of IBM stock unit is 81.37 at “1 May 2015 10:18:23" from NYSE market

```
<IBM,1430468303,81.37,NYSE>
```

The *StockTick* is emitted directly from source so that it is a base stream. However, a below *HighStockTick* stream is a derived stream originated from *StockTick*. *HighStockTick* contains only transaction of stock with price of more than $100 each unit.

```
CREATE STREAM HighStockTick AS
SELECT * FROM StockTick
WHERE price > 100
```

In practice, base stream are almost always append-only , mean that previously arrived stream elements are never modified. However, derived stream may or may not be append-only[11]. A derived stream that present the average transaction volumes between interval time $[t_1, t_2]$. The query produce an element s_1 to the stream immediately after t_2. However, an element of *StockTick* stream with timestamp $t_{12} \in [t_1, t_2]$ arrive late at $t_2 + \phi$. If the system

takes into account of the late arrival element, it will update the previous average volumes at $[t_1, t_2]$. In this case, this derived stream is not append-only. Unfortunately, Flink has not supported Delay function, so that all stream is apparently append-only.

Note: another examples from **epl-guide Note**: stock example : http://www.codeproject.com/Articles/553 Introduction-to-Real-Time-Stock-Market-Data-Pro

Logical Stream vs. Physical Stream

// TODO : Logical vs. physical stream ? [10]

Logical stream is a conceptual and abstract data stream which is processed linearly through a series of chaining operators. According to logical data flow graph, one is able to observe the order of operators that data is processed and what are the input and output of process.

Physical stream flow graph indicates how system really process the data in data parallelism environment. Physical operator is replicated and the internal operator state is partitioned and segmented. Figure 2.1 depict the different between logical and physical data flow. A logical stream from source go straight through an aggregate and a filter operator before written to Sink, whereas physical streams are segmented and go through different internal replica of the same logical operator. Eventually, all physical streams will be merged and written to one Sink.

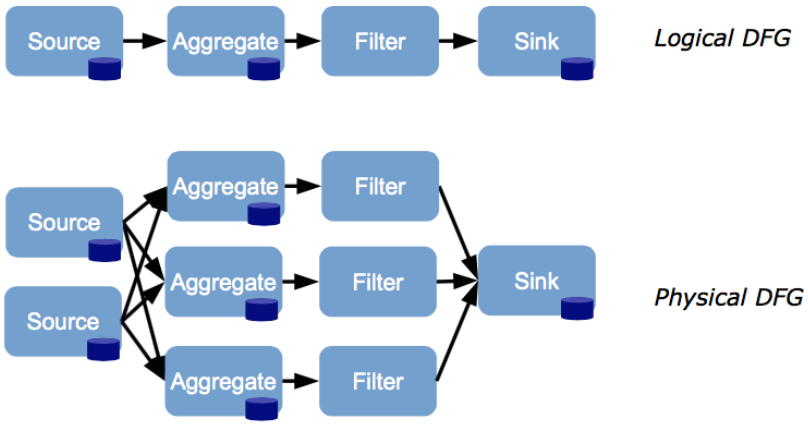

Fig. 2.1 Extract data parallelism from a logical DFG by replicating operators and segmenting their internal state in the corresponding physical DFG. The disk icon represent an operator's internal state [8]

Signals[Option] [11]

2.2 Stream Windows

Why do we need window?

From the system's point of view, it is often infeasible to maintain the entire history of the input data stream. Because data stream is running infinitely, we do not know when it ends. It nearly impossible to query over the entire stream with some operators such as sum, average. Accumulating a tuple attribute for entire stream may results to a very big value causing buffer overflow. From the user's point of view, recently data may be insight and more useful to make a data-driven decision. Those reasons motivated the user of windows to restrict to the scope of continuous queries.

Many stateful stream processing operators are designed to work on windows of tuples, making it a fundamental concept in stream processing. Therefore, a stream processing language must have rich windowing semantics to support the large diversity in how stream processing engine can consume data on a continuous basis.

Definition 2.5 *A **Window** W over a stream $\mathbb{S}$ is a finite subset of stream $\mathbb{S}$ [6]*

A window over streaming data can be created, buffering a continuous sequence of individual tuples. However, the size of window is a finite number so that system must decide what and how to buffer data based on the window specification.

The specification consists of several parameters :

1. an optional partitioning clause, which partitions data in window into several groups. Query on window will be taken place regard for each group, instead of the whole window

2. a window size, that may be expressed either as the number of tuples included in it or as the temporal interval spanning its contents

3. an window slide, the distance between the starts of 2 consecutive window (i.e., waiting 2 seconds or 5 data elements before starting a new window). This crucial property determine whether and in what way a window change state over time. If the window slide parameter is missing, system can assign implicitly that the slide size is equal to the window size. In this case, we have a stream of disjoint windows so called batch windows or tumbling windows.

4. an optional filtering predicate, keeping only elements that satisfy the predicate.

For example, a window specification

```
[
SIZE 2 hours EVERY 1 hours
  PARTITIONED BY Exchange
  WHERE Quantity > 100.000
 ]
```

means that window cover all transactions with $Quantity > 100.000$ over last 2 hours once 1 hour. Transaction tuples inside the window are partitioned into group specified by Exchange keyword Figure 2.2

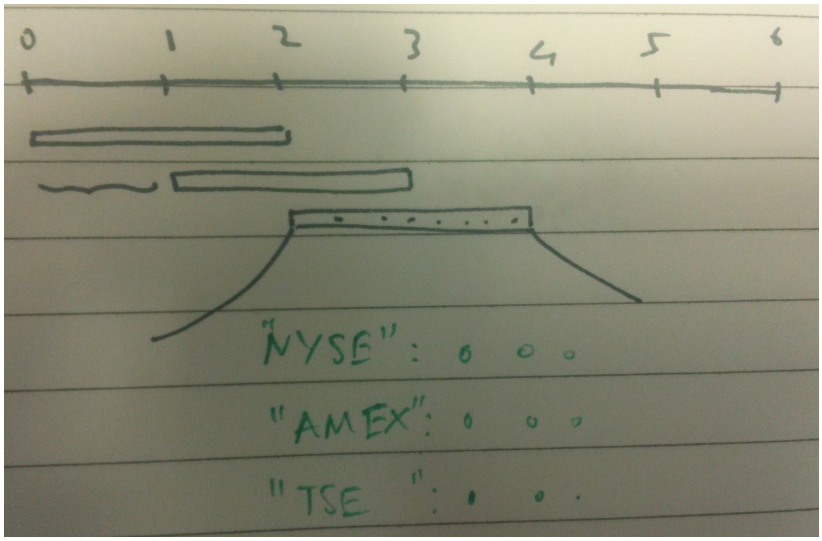

Fig. 2.2 Windowed Stream

Windows can be construct according to (window specification) that defines what to buffer, resulting in many window variations. These variations differ in their policies with respect to evicting old data that should no longer be buffered, as well as in when to process the data that is already in the window.[2]

Window may be classified according the following criteria:

2.2.1 Direction of movement

Window can fixed or sliding along the stream.

- Fixed Window: has both upper-bound and lower-bound fixed. Therefore the window is evaluated only once and capture a constant portion information of stream. For instance, window stores the transactions generated in 2 hours from ”2015/01/01 12:00:00" to ”2015/01/01 14:00:00"

- Landmark Window: One of the bounds remains anchored at a specific system timestamp. The other edge of the window is allowed to move freely. Usually, the lower-bound is fixed , and the upper-bound shifted forward in pace with time progression. For example, windows capture all transaction from 1a.m once every hour. Figure 2.3

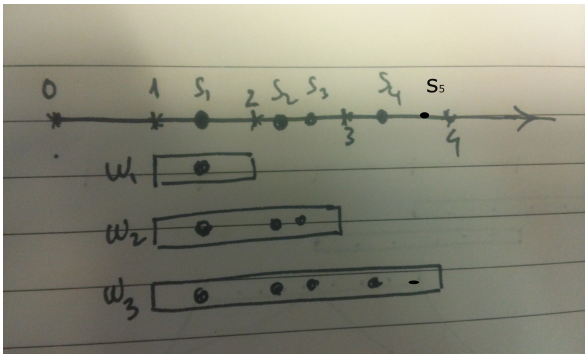

Fig. 2.3 Landmark Window

Up to 4am, the stream contains 3 window

  - $W_1(1,2) : \{s_1\}$
  - $W_2(1,3) : s_1, s_2, s_3$
  - $W_3(1,4) : s_1, s_2, s_3, s_4, s_5$

- Sliding window: the width of the window may be fixed in term of logical unit (i.e., time interval units) or physical unit (i.e., tuple count in window). However, the boundaries of windows change overtime along the stream.

  For example, window contains last 3 transactions once every 1 transaction passed. Up to 4am, the stream contains 5 windows. Figure 2.4

  - $W_1 : \{s_1\}$: at the beginning there is only tuple s_1 on stream
  - $W_2 : s_1, s_2$ there are only tuple s_1 and s_2 on stream. Window may take up to 3 tuples so that both s_1 and s_2 are included
  - $W_3 : s_1, s_2, s_3$
  - $W_4 : s_2, s_3, s_4$ there are 4 tuples on stream but window can take last 3 tuples only. Therefore , window buffer will insert s_4 and drop s_1
  - $W_5 : s_3, s_4, s_5$ window buffer insert s_5 and drop s_2

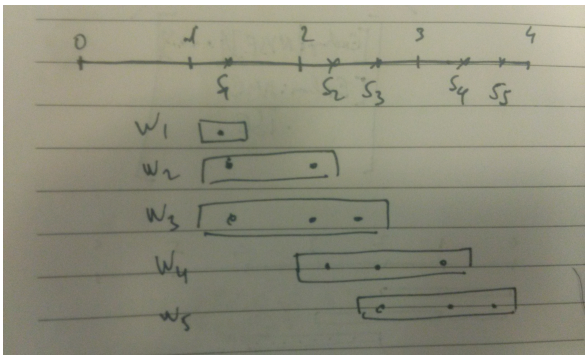

Fig. 2.4 Sliding Window

- Tumbling window: a particular sliding window where the boundaries move is equal to the window's width. Windows are disjoint or non-overlapped each other. Windowed stream will cover all elements on based stream.

  Example: For example, window contains last 2 transactions once every 2 transaction passed. Up to 5am, the stream contains 3 windows. Figure 2.5

  - $W_1 : \{s_1, s_2\}$
  - $W_2 : \{s_3, s_4\}$
  - $W_3 : \{s_5, s_6\}$

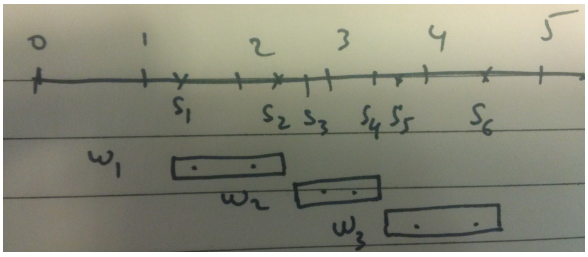

Fig. 2.5 Tumbling Window

- Jumping window: a particular sliding window where the boundaries move is larger than the window's width. Windows are disjoint or non-overlapped each other but some of tuples may be discarded. For example, window contains last 2 transactions once every 4 transaction passed. Up to 5am, the stream contains 2 windows. Figure 2.6

  - $W_1 : \{s_3, s_4\}$ When s_4 has arrived, window buffer contains 4 tuples $\{s_1, s_2, s_3, s_4\}$ but window's width is 2 so that $\{s_1, s_2\}$ will be evicted from window. Window buffer keeps $\{s_3, s_4\}$ then emits W_1
  - $W_2 : \{s_7, s_8\}$ Window buffer evicts $\{s_5, s_6\}$, keeps $\{s_7, s_8\}$ then emits W_2

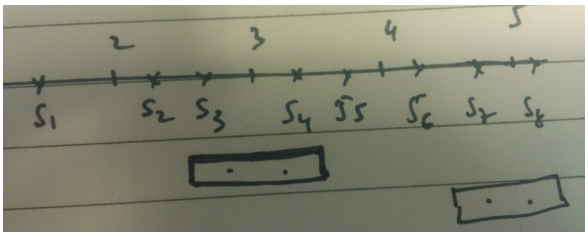

Fig. 2.6 Jumping Window

2.2.2 Definition of contents

- Logical or time-based windows are defined in terms of time interval, e.g., a time-base sliding window may maintain the the last one minutes of data.

- Physical(also known as count-based or tuple-based) windows are defined in terms of the number of tuples, e.g., a count-based window may store the last arrived 100 tuples.

- Delta-based windows are defined in terms of a delta function and a threshold value. The function calculates a delta between 2 elements such as absolute distance or Euclidean distance between 2 data elements. In delta-based windows, the delta between the first element and any of the rest must not be larger than the threshold, respectively. Currently new arrival data point will join the window if the delta between it and the first elements of window is equal or less than threshold. Otherwise, the window is closed and emitted; the currently arrival data point trigger a new window.

  Formally, a delta windows W contains n interval ordered elements $s_1, s_2, ..., s_n$ continuously so that every elements a_k with $k \in [1, n]$ must satisfies

  $$\Delta(s_1, s_k) \leq \phi \tag{2.4}$$

  There is a new arrival tuple s_{n+1}.

  - if $\Delta(s_1, s_{n+1}) \leq \phi$, s_{n+1} will join window W
  - Otherwise, window W is closed and emitted for further computation. s_{n+1} will trigger new window $W' : \{s_{n+1}\}$

  For example, assuming that stream S contains 4 tuples so far (Figure 2.7). Element in window W must satisfy the condition that the absolute distance between it and the first elements is not higher than 10.

  Up to the moment s_4 has processed, Stream S is discretized into window streams of

- $W_1 : \{s_1, s_2, s_3\}$ satisfies $\Delta(s_k, s_1) = |s_k - s_1| < 10$ where $k \in [1,3]$
- $W_2 : \{s_4\}$ because $\Delta(s_4, s_1) = |s_4 - s_1| = 12 > 10$, s_4 trigger a new window W_2

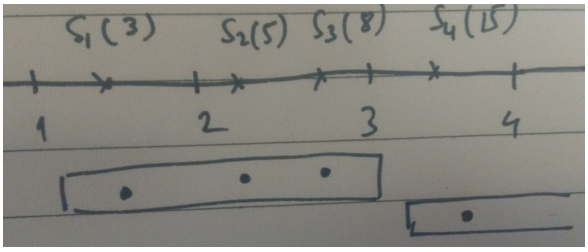

Fig. 2.7 Delta-based Window

- Partitioned Windows contain only the elements in the same group which differentiates itself from the other groups by the value of a grouping attributes (subset of its schema), e.g., a partitioned window store last 100 elements with the same value of (*StockSymbol*, *Exchange*)(Figure 2.8). Thus, several substreams are derived logically from the base stream, each one is represented by an existing combinations of value$< a_1, a_2, ..., a_k > \in Dom(S)$ on the grouping attributes $< A_1, A_2, ..., A_k > \subset S$ (S is schema of tuples, $Dom(S)$ is domain of S). Each group maintains a separate window buffer to capture arrival events and emit when satisfies window specification.

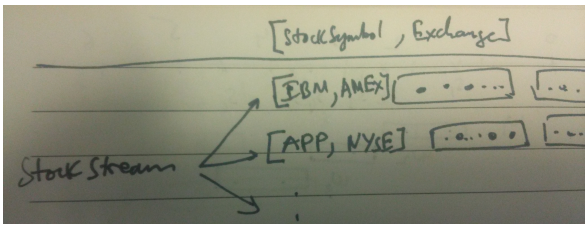

Fig. 2.8 Partitioned Window

- Predicate window [7] , in which an arbitrary logical predicate specifies the contents. Only tuples that satisfies the predicate will join the window, otherwise it is discarded. e.g.,predicate window maintain last 100 transactions which have more than 100.000 units in terms of *volume*, respectively. Every transaction with fewer quantity will be discarded.

(check Sliding Window Query Processing over Data Stream pages 25)

The notion of sliding windows requires at least an ordering on data stream elements. In many cases, the arrival orders of the elements suffices as an ïmplicit timestampättached to each data element. However, sometimes it is preferable to user ëxplicit timestamppṗrovided as

part of data stream. Formally, we say that a data stream consists of a set of (tuple, timestamp) pairs. Timestamp attribute could be a traditional timestamp or it could be a sequence number - all that is required is that it come from a totally ordered domain with a distance metric. The ordering induced by the timestamp is used when selecting the data elements making up a sliding window.

Notes: in CQL tuple-based sliding window may be non-deterministic - and therefore may not be appropriate - when timestamp are not unique

Notes: Streambase: tuple-at-a-time

Notes: Tuple relational calculus by Edgar F. Codd

Notes: http://en.wikipedia.org/wiki/Relational_algebra

Chapter 3

The execution semantic of Stream Processing in Flink (10 pages)

3.1 Heterogeneity

Since the first commercial project of Complex Event Processing launched by Bell Labs in 1998 with its "Sunrise Project", we have seen the fast growing of many stream processing frameworks. However, there is a huge degree of heterogeneity across these frameworks in various forms[6]

1. Syntax: Although the ISO/IEC 9075 is published standard to defines the complete syntax and operations in SQL language as a whole, there is no standard language for stream processing. Different stream processing engines use different syntax to depict the same function. For example,every 5 seconds, a window captures all event last 10 seconds.

   ```
   CQL:  [RANGE 10 seconds SLIDE 5 second ]
   Flink:  [SIZE 10 sec EVERY 5 sec]
   ```

2. Capability heterogeneity: Those engines also provide different set of query types and operations based on which functions they are capable of. For examples, *Streambase* support pattern matching on stream, whereas STREAM does not.

3. Execution Model: Below the language level, hidden from application layers, each stream processing engine has its own underlying execution model. With the same data stream but different model produce different output which varies based on the differences on tuple ordering, window construction, evaluation and so on. We are going to focus on the differences between several existing execution models below.

We have learned that there are at least three different execution models:

- **Time-driven** execution model, followed by CQL, Oracle CEP. In the model, each tuple have a timestamp. Timestamp induces the total order of tuples on stream, but not a strict total order. Or more specifically, there is no ordering between tuples with identical timestamps. These tuples are consider as simultaneous tuples. It is problematic when we select a window of last 10 tuples but more than 10 simultaneous tuples arrived at a given time instant. In this case, there is no different between those tuples, the system will select only 10 out of all in a non-deterministic way.

  Assuming that we has stream $\mathbb{S}$

$$\mathbb{S}(value, t_{app}) = (1,1),(10,2),(20,2),(100,3) \tag{3.1}$$

  Consider a query which continuously recal the last arrival tuple i.e., we select tuple-based window with size of 1 tuple. In the time-based execution model, the state of a window changes as timestamp progress. Window get re-evaluated only when timestamp change. At $t = 1$ or $t = 3$, there is only 1 arrival tuple, new 1-tuple-size window will open , pick the tuple then close. Thus the stream derives window $W_1 : \{(1,1)\}$ and $W_3 : \{(100,3)\}$. On other hand, at $t = 2$, there are 3 new arrival tuples. New window W_2 opens and be able to pick 1 tuple only. Since these 3 tuples arrive simultaneously, they will have the same timestamp and thus no any temporal differences between them. Engine randomly pick one of them for window W_2. In short, window W_2 contains one of following options: $(10,2)$ or $(20,2)$. The derived stream will be one of the streams:

```
(1,1),(10,2), (100,3) or (1,1),(20,2), (100,3)
```

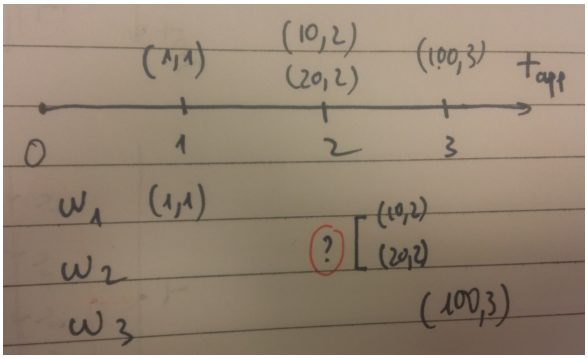

Fig. 3.1 time-driven

- **Tuple-driven** execution model, followed by StreamBase, Apache Flink. In this model, tuples may have an application timestamp attribute on its schema. Some of timestamp values might be identical but tuples themselves are completely distinguished in stream. There exists a strict total order in stream based on their arrival order.

  There are several ways to represent tuple order in stream. StreamBase system assigns an incremental internal rank to tuples to arriving tuples. It ensures that the tuple with lower rank with be processed before tuples with higher rank. In Apache Flink, we implicitly use system timestamp t_{sys} at which system receives the tuple. Since system timestamp are strictly totally ordered, it is simply suitable for Flink execution model. Several other works propose to use tuple Id [6] or a physical identifier[12] instead.

  In tuple-drive execution model, each tuple arrival cause a system to react, instead of each application timestamp progress. In previous example, tuple $(10,2)$ and $(20,2)$ has the same application timestampe but system will open a new 1-tuple-size window for each of them. Therefore,assuming that tuple $(10,2)$ arrives before tuple $(20,2)$, the derived stream will be exact as

  ```
  (1,1),(10,2),(20,2), (100,3)
  ```

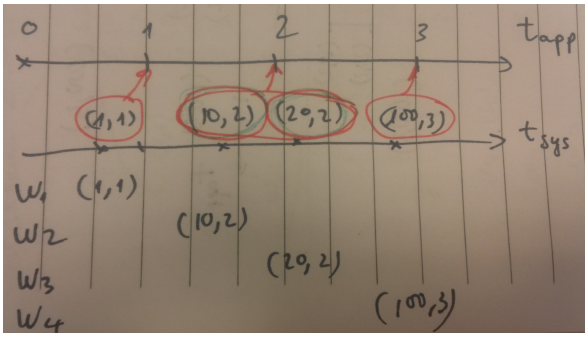

Fig. 3.2 tuple-driven

- **Batch-driven** execution model, followed by Coral8, mentioned in SECRET[Botan et al.] descriptive model

. In this model, each tuple is assigned an batch-id. Every tuple which belong to a batch must have the same timestamp, but 2 separate tuple with the identical timestamp may belong to two different batches. As we can see, batch-driven model is in between of tuple-driven and time-driven model (Figure 3.3). Assuming that at a given application timestamp τ, the system receives 5 tuples $< v_1, \tau >, < v_2, \tau >, < v_3, \tau >, < v_4, \tau >, < v_5, \tau >$. Time-driven model treats them as simultaneous tuples with no difference. Tuple-driven considers them as

5 concrete tuples in strict order. And batch-driven model may divide them into 2 batches $\{<v_1,\tau>,<v_2,\tau>,<v_3,\tau>\}$, $\{<v_4,\tau>,<v_5,\tau>\}$ depending on window specification.

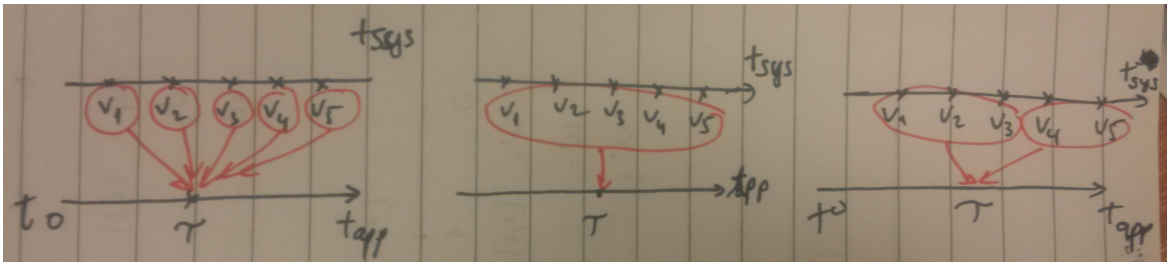

Fig. 3.3 executionModel

We extends the examples [6] with Flink implementation in order to demonstrate that system with different execution model may produce different output , even with the same input and query.

1. Example 1: differences in window constructions Given *Instream* stream with schema $S(time, value)$. Consider a query which continuously computes the average value of tuples in a time-based tumbling window of size 3.

$$\begin{aligned} Instream(time, value) = \{&(10,10),(11,20),(12,30),(13,40),(14,50),\\ &(15,60),(16,70),...\} \end{aligned}$$
$$OracleCEP\,(avg) = \{(20),(50),...\}$$
$$Flink\,(avg) = \{(15),(40),...\}$$

   Obviously, Oracle CEP constructs the first window with first 3 tuples whereas Flink picks first 2 tuples only. The second window , they both take next 3 tuples. We implement the test on Flink with default configuration, however we are able to customize the upper bound of the first window ($startTime = 12$)so that it produces the same result as Oracle CEP.

2. Example 2: differences in window evaluations

   Consider a query which continuously computer the average value of tuples over last 5 second once every 1 second (time-based window of size 5s that slides by 1s)

$$
\begin{aligned}
Instream(time, value) &= \{(30,10),(31,20),(36,30),...\} \\
OracleCEP\,(avg) &= \{(10),(15),(20),...\} \\
Flink\,(avg) &= \{(10),(15),(15),(15),(15),(20),...\} \\
Coral8\,(avg) &= \{(10),(15),(20),...\}
\end{aligned}
$$

Flink produced a different result than Oracle and Coral8. In Oracle and Coral8, a new window is emitted for invoking the average operator only when the window content's change; whereas in Flink, it emits a new window every second as the sliding progress , even if the content does not change. The state of window is depicted on Figure 3.4

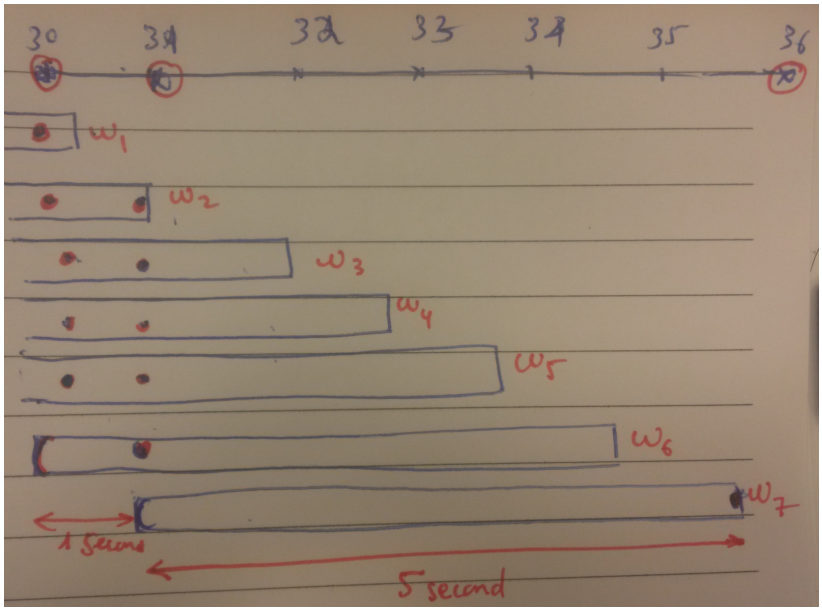

Fig. 3.4 Window State

Remember that window closes at upper boundary and opens at lower boundary so that in window W_6 cover from $t = 35$ to right after $t = 30$ will exclude tuple $(30,10)$. Similarly, W_7 excludes tuple $(31,20)$

3. Example 3: differences in processing granularity

Consider a query which computes the average value of tuples over a tuple-based tumbling window of size 1 tuple.

$$\begin{aligned} Instream(time, value) = \{&(10,10),(10,20),\\ &(11,30),\\ &(12,40),(12,50),(12,60),(12,70),\\ &(13,80),...\} \end{aligned}$$
$$OracleCEP(avg) = \{(20),(30),(70),(80)...\}$$
$$Flink(avg) = \{(10),(20),(30),(40),(50),(60),(70),(80)...\}$$
$$Coral8(avg) = \{(10),(20),(30),(40),(50),(60),(70),(80)...\}$$

Oracle CEP implements time-based execution model so that it reacts to each application timestamp. If there are multiple simultaneously tuple arrive, it will pick one of them non-deterministically to construct the window , since window size is 1. In other hand, Flink and Coral8 react to every tuple arrival so that they emit new 1-tuple-size tumbling window for every tuple.

tuple: each tuple arrival cause a system to react time: the progress of tapp cause a system to react batch: where either a new batch arrival or the progress of tapp cause a system to react

3.2 Policy-based Window Semantics

Flink constructs windows based on parameters in specification. Currently Flink does not support Predicate Window so that two of most critical parameters are to notify when system should trigger new windows (indicating the lower bound of window) and when system must end the window (indicating the upper bound of window) and emit it to window stream. For that purpose, Flink implements a mechanism called *"Policy-based windowing"*. It is a highly flexible way to specify stream descretization. It has two independent policies corresponding to open and close window: Trigger and Eviction Policy. To demonstrate the concepts of two policies, let's consider the scenario with *StockTick* stream: check every 10 minutes the total transaction volume of all transaction last 30 minutes. In other words, in every 10 minutes create a new window to cover all the transactions in last 30 minutes. The syntax in Fink:

```
StockTick.window(Time.of(30, MINUTES))
 .every(Time.of(10, MINUTES)).sum(Quantity)
```

1. Eviction Policy: define the length of a window. The length is passed in to *window(...)* function. It could be the time interval, number of tuples and delta function with threshold (in case of delta window). We formalize the concept of window due to its size

   Definition 3.1 *A time-based window $W_t = (l, u, \omega)$ over a stream S is a finite subset of $\mathbb{S}$ containing all data elements $s \in \mathbb{S}$ where $l, u, \omega \in \mathbb{T}$ andl $< s.t \leq u$. The length of window in time unit is $\omega = u - l$*

   Notice that in a time-based window, *s.t* can be tuple's application or system timestamp depending on query. There are maybe many simultaneous tuple with identical t_{app}. However, there is at least one arriving tuple at a given t_{sys}. The second point is that W_t open at $t = l$, it does not include tuple at this time instant.

   Definition 3.2 *A count-based window $W_c = (l, u, n)$ over a stream S is also a finite subset of $\mathbb{S}$ containing all data elements $s \in \mathbb{S}$ where $l, u \in \mathbb{T}$, $\omega \in \mathbb{N}$ and $l \leq s.t_{sys} \leq u$. The length of window is the number of tuples in interval time $(l, u]$, i.e., $|ns \in \mathbb{S}(t_{sys}) : l < s.t_{sys} \leq u|$*

   The count-based window W_c is independent from application timestamp t_{app}. It is related to system timstamp t_{sys} which indicate tuple's order in stream.

2. Trigger Policy: In general, it defines window slide or the distance between 2 consecutive windows. On above example , trigger policy states that from beginning, system must trigger a new window every 10 minutes. No other window would be triggered within this 10 minutes.

   Suppose that we have 2 consecutive windows $W^1 = (l_1, u_1, \_)$ and $W^2 = (l_2, u_2, \_)$ where $l_1 < l_2$. There is no window $W^1 = (l_3, u_3, \_)$ such that $l_1 < l_3 < l_2$.

   - If W_1 and W_2 are time-based windows, a slide $\beta_t = u_2 - u_1$.
   - If W_1 and W_2 are count-based windows, a slide $\beta_c = |s \in \mathbb{S}(t_{sys}) : u_1 < s.t_{sys} \leq u_2|$

If the correlation between window size and slide size conforms to the movement type of windows.

- Sliding window: $\omega > \beta$ or $n > e$
- Tumbling window: $\omega = \beta$ or $n = e$

- Jumping window: $\omega < \beta$ or $n < e$

However, Flink allow to mix between time-based trigger policy with count-based eviction policy and vice versa. For example, calculate sum of quantities of last 100 transactions every 1 hour.

```
StockTick.window(Count.of(100))
 .every(Time.of(1, HOURS)).sum(Quantity)
```

3.3 The execution semantic

Window Buffer *wb* is a linking list contain tuple with composite type *IN*

$$t = \begin{cases} t_{sys} & if\ t_{app} = None \\ \\ t_{app} & otherwise \end{cases} \tag{3.2}$$

$size_t(wb) = wb.last.t - wb.first.t$
$size_c(wb) = wb.length()$

Algorithm 1 Process new arrived tuple

The first step, before processing a new arrived tuple, check if a current window buffer should be emit. If yes, copy the current window buffer to a window object and put to Windowed Stream. The second step, calculating which tuples should be evicted if the current window buffer appends new arrived tuple. There are two separate case for time-based and tuple-based window. The third step, evicting those tuples and appending new arrived tuple.

Require: *wb*: the current window buffer
Require: ω_t: size of window in time interval
Require: ω_c: size of window in tuple count

```
 1: method PROCESSNEWTUPLE(newTuple : IN)
 2:     if NOTIFYTRIGGER(newTuple) then
 3:         window ← wb
 4:         EMIT(window)
 5:     end if
 6:     if window is time-based then
 7:         evict_t ← size_t(window.append(newTuple)) − ω_t
 8:         if evict_t > 0 then
 9:             lastEvictedTimestamp ← wb.first.t + evict_t − 1
10:             for all element e in window buffer wb do
11:                 if e.t ≤ lastEvictedTimestamp then
12:                     wb.remove(e)
13:                 end if
14:             end for
15:         end if
16:     end if
17:     H ← new Associative Array
18:     userid ← c.userId
19:     message ← c.message
20:     for all emotional regular expression p[i] do
21:         if message matches expression p[i] then
22:             H[i] ← 1
23:         else
24:             H[i] ← 0
25:         end if
26:     end for
27:     EMIT(term userId, array H)
28: end method
    class REDUCER
 2:     method REDUCE(term userId, counts [H_1, H_2, ...])
            sums ← new Array
 4:         for all array H ∈ lists of arrays [H_1, H_2, ...] do
                sums[i] ← sums[i] + H[i]
 6:         end for
            EMIT(term userId, array sums)
 8:     end method
    end class
```

Chapter 4

Continuous Query Semantics and Operators (10 pages)

In the streaming literature, query language model a stream as a representation for an infinite append-only relation. The append-only stream model effects the following limitations[7]

- It limits the applicability of the language since the append-only models cannot represent streams from the various domain(e.g., the update streams or streams that represent concatenation of the states of a fixed size relation).
- The append-only stream model limits the types of queries that the language can express since only non-blocking queries can produce append-only streams as output
- The semantics of query composition in the append-only stream model is complex and the meaning of the composed queries is difficult to understand.

Query [**?**]
Window Specification over Data Streams.pdf [Stream] Evaluate CQL

4.1 Continuous Query Semantics and Operators

4.2 Continuous Query Language

3. Query Formulation 4. Logical Operator Algebra
Semantic and Implementation of Continuous Sliding window queries over data streams

Chapter 5

Implementations

References

[Str] Streamsql tutorial. http://www.streambase.com/developers/docs/latest/streamsql/usingstreamsql.html/. Accessed May 10, 2015.

[2] Andrade, H. C. M., Gedik, B., and Turaga, D. S. (2014). *Fundamentals of Stream Processing*. Cambridge University Press. Cambridge Books Online.

[3] Arasu, A., Babu, S., and Widom, J. (2006). The cql continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142.

[4] Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J. (2002). Models and issues in data stream systems. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 1–16, New York, NY, USA. ACM.

[Botan et al.] Botan, I., Derakhshan, R., Dindar, N., Haas, L., Miller, R. J., and Tatbul, N. Secret: A model for analysis of the execution semantics of stream processing systems. *Proc. VLDB Endow.*, 3(1-2).

[6] Dindar, N., Tatbul, N., Miller, R. J., Haas, L. M., and Botan, I. (2013). Modeling the execution semantics of stream processing engines with secret. *The VLDB Journal*, 22(4):421–446.

[7] Ghanem, T. M., Elmagarmid, A. K., Larson, P.-A., and Aref, W. G. (2008). Supporting views in data stream management systems. *ACM Trans. Database Syst.*, 35(1):1:1–1:47.

[8] Henrique Andrade, B. G. and Turaga, D. (2013). Stream processing in action.

[9] Jain, N., Mishra, S., Srinivasan, A., Gehrke, J., Widom, J., Balakrishnan, H., Çetintemel, U., Cherniack, M., Tibbetts, R., and Zdonik, S. (2008). Towards a streaming sql standard. *Proc. VLDB Endow.*, 1(2):1379–1390.

[10] Krämer, J. and Seeger, B. (2009). Semantics and implementation of continuous sliding window queries over data streams. *ACM Trans. Database Syst.*, 34(1):4:1–4:49.

[11] Lukasz Golab, M. T. O. (2010). Data stream management.

[12] Petit, L., Labbé, C., and Roncancio, C. L. (2010). An algebric window model for data stream management. In *Proceedings of the Ninth ACM International Workshop on Data Engineering for Wireless and Mobile Access*, MobiDE '10, pages 17–24, New York, NY, USA. ACM.

[13] Petit, L., Labbé, C., and Roncancio, C. L. (2012). Revisiting formal ordering in data stream querying. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 813–818, New York, NY, USA. ACM.

[Takada] Takada, M. *Distributed Systems For Fun and Profit*.