# FlinkSQL

The SQL dialect for Flink


**Khuong Duy Vu**

Department of Informatics

ELTE University

This thesis is submitted for the degree of

*Master Thesis*

June 2015

# Table of contents

# Chapter 1

# Introduction

- BUILDING ALGEBRA SEMANTIC FOR FLINK STREAMING - IMPLEMENTATION

## Big Data

In the last few years Big Data generated a lot of buzz along with the launch of several successful big data products. Thanks to contribution from open source community and several giant Internet companies, the big data ecosystem has now approached a tipping point, where the basic infrastructure capabilities of supporting big data challenges are easily available. Entering the next generation of big data, so-called Big Data 2.0, two of its concentrated areas are Velocity and Applications, besides Data Quality. The cause for the former is that data is growing at an exponential rate and the ability to analyse it faster is more important than ever. For instance, sensors can generate data on millions of events per second and store all of those data and response in real-time is non trivial. The latter is helping to overcome the technical challenges of existing frameworks by making them easy to use and understand for everyone to benefit from big data.

As a result, the demand for streaming processing is increasing a lot these days. Processing big volume of data is not sufficient in the cases that infinite streaming data is arriving at high speed and users require a system to process fast and react to any incident immediately. In addition, although hardware price has plunged year over year, it's still expensive to equip a storage which is growing terabytes every day for batch analysis. Streaming processing engines are designed to operate high volume in real time with a scalable, high available and fault tolerant architecture.

One of the disadvantages to users is that many big data frameworks provide a rich imperative API code only to process data stream. First, users must spend time learning API documentation properly since those APIs are fairly new to them. Therefore, the cycle

time to develop products taking longer. Second, given that most big data applications are fairly simple application-wise, a block of API codes might be less optimal to use for most the popular queries[14]. Third, the only way to share your work is to pack it as a library or service that need to be deployed again. Fourth, the results of streaming queries keep changing upon the time. Thus, it demands a visual way to observe the streaming instead of a sequence of boring numbers. Those above drawbacks apparently provide an unfriendly UX that inhibit both productivity and system performance. It is really against what we expect from "Application" aspect of Big Data 2.0.

# Data Streaming

Streaming Processing is not a new concept. Indeed the similar concept, Complex Event Processing (CPE) had been proposed from the 1990s by Event Simulation Research at Stanford [1]. Since that time, people have started generating a lot of different buzzwords around it [2] and often reinventing ideas borrowing from other fields, but using a different vocabularies to describe the same concepts. Basically, the idea is to analyse one or multiple data streams to identify meaningful phenomena and respond to them as quickly as possible.

According to CEP Tooling Market Survey 2014 [3], since 1996, there has existed more than 30 companies providing Streaming Processing solutions. All the major software vendors (IBM, Oracle, Microsoft, SAP) also have good to excellent offerings in the CEP space for customers.

However, since a massive amount of data is growing rapidly every second, Hadoop is emerging distributed processing ecosystem today. Thanks to Hadoop, people can build a large scalable distributed system on Cloud. Even though Hadoop is designed to scale system up to thousands of machines with very high degree of fault tolerance, it is optimised to run batch jobs with a huge load of computation. Because of time factor, Hadoop has limited value in online environment where fast processing is crucial. Therefore, existing CEP solutions are barely compatible with Hadoop ecosystem. We demand a new sort of streaming framework which is able to integrate on top of Hadoop system. Apache Flink is one of these frameworks.

# CQL and related work

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

# Flink and FlinkQL

# Structure

Challenge: so flexible on execution model : $t_{app}$ $t_{sys}$

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

Sliding Window Query Processing over Data Stream by Lukasz Golab

Processing data streams is a a different paradigm, and moreover, Java is typicaly 50X less compact than say SQL – significantly more code required. Java and Scala require significant garbage collection which is particularly inefficient and troublesome for in-memory processing.

# Chapter 2

# Data Stream Model

## Order

The concept of *Order* 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 like a queue: starting from the beginning, and then going towards the end.

In distributed system, programs are designed to solve the same problems which one can solve on a single machine using multiple interconnected machines. Although these machines are physically located across the network with possible delays or failures, the system tries to reserve the order of the result as if running on a single machine only. In other words, the ideal is that a) we run the same operations and b) that we run them in the same order - even if there are multiple machines[17].

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

**Definition 2.1** *Paritial order [16] 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:*

- $a \leq a$ *(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)*

A set of elements, which is partially ordered, does not always ensure the order of 2 arbitrary elements. The natural state in a distributed system is partial order. Neither in the network nor between independent nodes the system is able to make any guarantees

about relative order of two elements, probably due to many factors such as network latency, performance and so on; but at each node, one can observe a local total order.

**Definition 2.2** *Total order [16] 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 element.

In streaming processing, one may not ask for entire stream but rather a portion of data stream (i.e., window) periodically for further computation. For example, every 5 minutes, they would like to know the average volumes of last 100 transactions to detect any abnormal transaction. Any older element from 101th will be discarded. Since the number of transactions is bounded within 100, the order of elements is crucially needed here to decide which one should fall into the window but others do not. This property also make stream processing model is different from rational data base system in which order of elements might not necessary. Traditional DBMS already knows the bounded data set involved in queries whereas stream processing engine has to decide its window based on the order.

Depend on the execution model of different systems, they may design different strategies of order such as temporal or positional order [15].

The **temporal order** is induced by the timestamp of elements in a stream. Using the value of timestamp, one can determine whether something happen chronologically before something else. In practice, they usually use time as a source of order. System can attach timestamps to unordered events to maintain an order between events. Nevertheless, if some elements happen simultaneously, they will have the identical timestamp, then the order is total but non-strict. For instance, there are two elements with the same timestamp but system is required to take one only, the chosen is non-deterministic between two because there is no difference between them in term of order. Therefore, we may require a strict order.

The **positional order** is a strict order induced by the position of elements in stream. Two elements 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 element regardless of an explicit timestamp.

## 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.

Time instant can be signified by either human-readable formatted string such as ”Wed Aug 21 2013 00:00:00 GMT-0700 (PDT)" or a *Long* number as a milliseconds time value. In practice, in many high-level languages, the ”zero epoch" moment $t = 0$ is usually set to the midnight of Jan 1 1970 and time unit is millisecond. Obviously, it is exchangeable between 2 representing formats. For the sake of simplicity, we will assume that the time domain is the domain of non-negative long number ($\mathbb{T} = \mathbb{N}$) 0,1,2,3,.. and totally ordered[8].

When considering the order, each event may be attached with either or both of : system time $t^{sys}$ (implicit) and application time $t^{app}$ (explicit).

**Application Timestamps** $t_{app}$ In many case, each element in stream contains an explicit source-assigned timestamp itself. In other words, the timestamp attribute may be a part of the stream schema. To consider a common log format for a web application which contains a timestamp specifying when the action is taken place. A log line records an action of user *pablo* to get an image on Oct 10 2000:

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

Since web server may handle thousands of concurrent requests per second, it is possible to have many line of log sharing a $t_{app}$ value. Therefore, application timestamp can be used as a source of total but non-strict ordering.

**System Timestamps** $t_{sys}$ Even if the element 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, one should be aware that application time and system time are not necessarily synchronized[12]. Since system timestamp is assigned implicitly by system, one may not notice its presence on schema.

Both application and system timestamp captures time information but they carry 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 elements 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. System then can believe in the system timestamp as a strict total ordered basis for reasoning about arrival elements 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.

As we mentions above, in general, 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 nodes. From now on, we are going to analyze the execution model of stream processing on logical layer which means that it work like it would on a single machine. Thus, we could assume that time domain is the source for total ordering.

**TODO**: 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 represent a relational tuple, a event or a record of sensor data and so on [3]. For instance, the line of log in previous example follow a schema:

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

A data tuple is the fundamental, or atomic data element, 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[15]. A tuple consists of a set of *(Attribute* $\times$ *Value)* pairs such as $(SourceIP, 219.53.210.143)$

## 2.1 Stream Model

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

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

There are several definitions of data stream varying based on the execution model of systems. On the previous definition, a timestamp attribute can be a non-strict total ordered application timestamp so that system may not rely on it to select tuples on some operations requiring proper order between any pair of tuples. For this reason, stream can contain an extra physical identifier $\varphi$ [14] 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 element is to separate the concept of application and system timestamp. In SECRET model [8], each stream element 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, we learn 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 in milliseconds.

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

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

However, we may receive the stream signal 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 timezone.

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

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 S, with an optional application time value $t_{app} \in \mathbb{T}^*$ with $\mathbb{T}^* = \{-1\} \bigcup \mathbb{T}$ and a timestamp $t_{sys} \in \mathbb{T}^*$ generated automatically by system, due to the event arrival.*

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

$$t_{app} = \begin{cases} f(v) & if\ funtion\ f\ is\ defined \\ \\ -1 & otherwise \end{cases} \tag{2.4}$$

## 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 [13]:

- 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, [13] page 17 From now on, we give example queries on input stream named *StockTick*[1] 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[13]. 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 ? [12]

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.

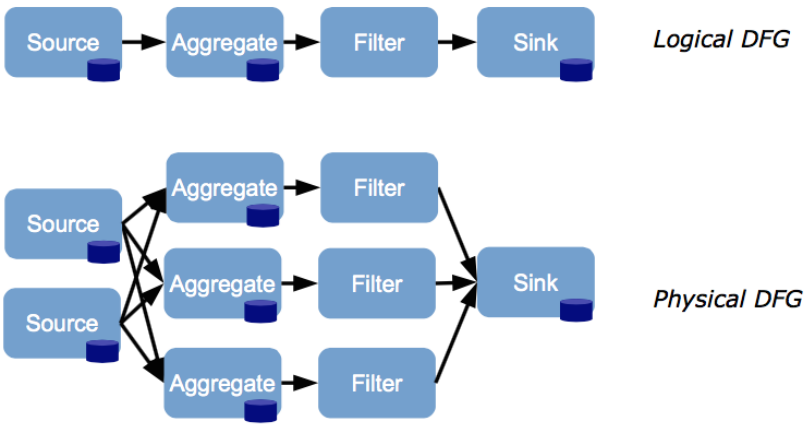



Fig. 2.1 Extract data parallelism from a logical DFG by replicating operators and segmenting their internal state in the corresponding physical DFG. The disk icon represent an operator's internal state [10]

## 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.

// TODO: Approximate Query Answering: (Models and issues in data stream systems)

**Definition 2.5** *A **Window** $W$ over a stream $\mathbb{S}$ is a finite subset of stream $\mathbb{S}$ [8]*

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 3 hours EVERY 1 hours
  PARTITIONED BY Exchange
  WHERE Quantity > 100.000
 ]
```

means that window cover all transactions with $Quantity > 100.000$ over last 3 hours once every 1 hour. Transaction tuples inside the window are partitioned into groups specified by Exchange keyword Figure 2.2

```
[
  SIZE 3 hours EVERY 1 hours
  PARTITIONED BY Exchange
  WHERE Quantity > 100.000
 ]
```

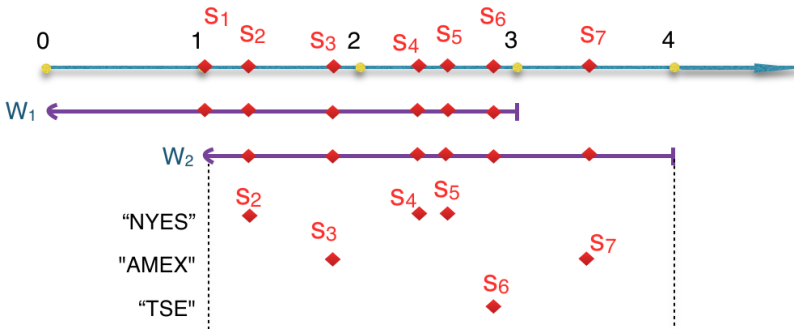



Fig. 2.2 Windowed Stream

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

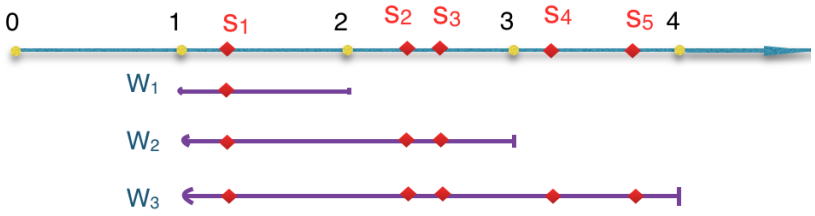



Fig. 2.3 Landmark Window

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 $s_2$

- 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

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

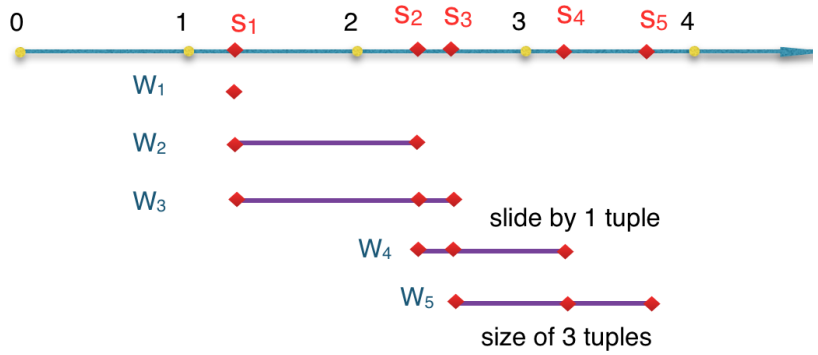



Fig. 2.4 Sliding Window

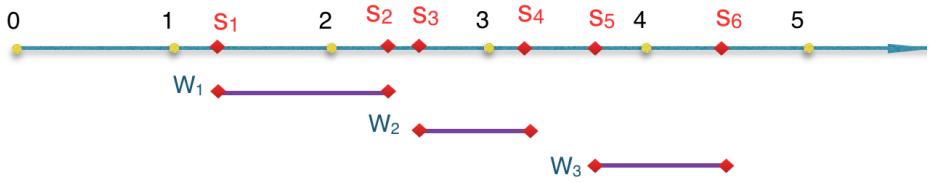



Fig. 2.5 Tumbling Window

– $W_3 : \{s_5, s_6\}$

- 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 $W_2$

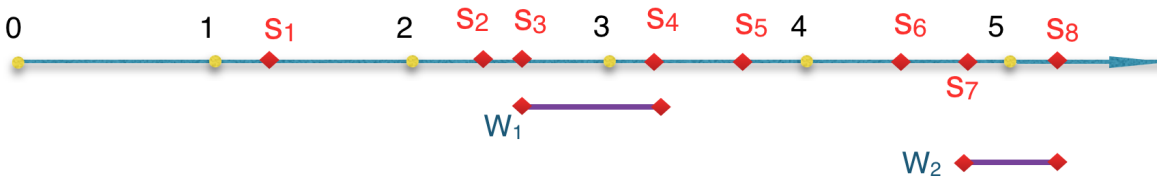



Fig. 2.6 Jumping Window

### 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

  $$\Delta(s_1, s_k) \leq \phi \tag{2.5}$$

  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 $W_2$

- 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

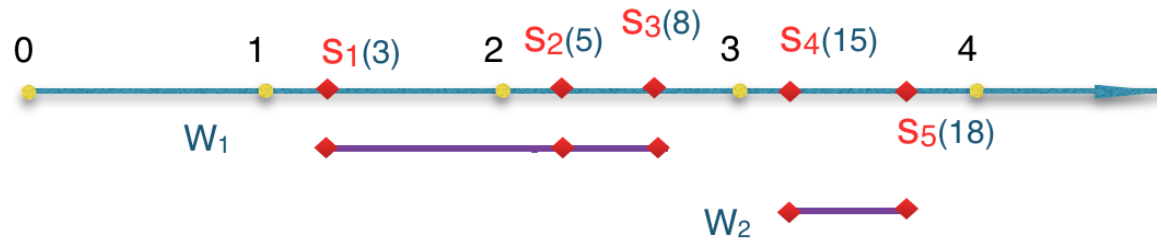



Fig. 2.7 Delta-based Window

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.

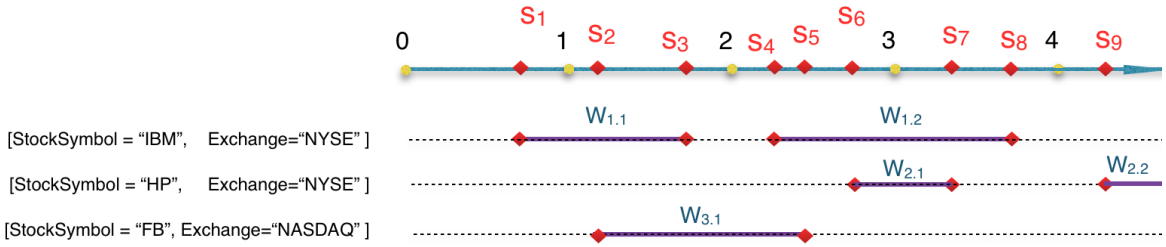



Fig. 2.8 Partitioned Window

- Predicate window [9], 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 timestamppïrovided 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 Flink Stream Processing

## 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 [8]:

1. **Syntax**: Although the ISO/IEC 9075 is published to standardize 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 semantic meaning. 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 considered 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 $\mathbb{S}$ (regardless of system timestamps):

$$\mathbb{S}(value, t_{app}) = s_1(1,1),\ s_2(10,2), s_3(20,2), s_4(100,3) \tag{3.1}$$

  Consider a query which continuously recall 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 gets re-evaluated only when timestamp change. At $t = 1$ or $t = 3$, there is only 1 tuple arrived, new 1-tuple-size window will open , pick the tuple then close. Thus the stream derives window $W_1 : \{s_1(1,1)\}$ and $W_3 : \{s_4(100,3)\}$. On the other hand, at $t = 2$, there are 2 new tuple arrivals simultaneously. New window $W_2$ opens and accepts 1 tuple only. Since these 2 tuples arrive simultaneously, they will have the same timestamp and thus no any temporal differences between them. System simply picks one of them randomly for window $W_2$. In short, window $W_2$ contains one of following options: $s_2(10,2)$ or $s_3(20,2)$. The derived stream will be one of the streams:

$$W_1\{s_1\}, W_2\{s_2\}, W_3\{s_4\}\ or\ W_1\{s_1\}, W_2\{s_3\}, W_3\{s_4\} \tag{3.2}$$

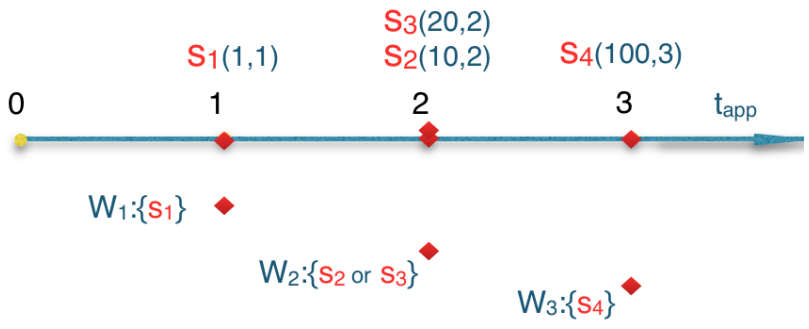



Fig. 3.1 Time-driven Execution model. Window size of 1 tuple

- **Tuple-driven** execution model, followed by StreamBase, Apache Flink. In this model, tuples may have an application timestamp attribute on its schema. Some of application 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 ranks. In Apache Flink, we implicitly use system timestamp $t_{sys}$ at which system receives the tuple. Executing this *tuple-at-a-time* model, logically Flink places new arrived tuple to a queue and process it one by one. Therefor, there is at most one tuple considered arriving at a give time. Since tuples with attached system timestamps are strictly totally ordered, it is perfectly suitable for Flink's execution model. Several other works propose to use tuple Id [8] or a physical identifier[14] instead.

  In tuple-driven execution model, each tuple arrival cause a system to react, instead of each application timestamp progress.

  Extending previous examples, every tuple is entitled to a system timestamp. As we mentioned in previous chapter, application and system timestamp are not necessarily synchronized.

$$\mathbb{S}(value, t_{app}, t_{sys}) = s_1(1,1,28),\ s_2(10,2,37), s_3(20,2,40), s_4(100,3,46) \tag{3.3}$$

  Tuple $s_2$ and $s_3$ has the same application timestamp $t_{app} = 2$ but system will open a separate 1-tuple-size window for each of them upon their arrivals. Therefore, since $s_2$ arrived before $s_3$, the derived stream will be exact as (Figure 3.2)

$$W_1\{s_1\}, W_2\{s_2\}, W_3\{s_3\}W_4\{s_4\} \tag{3.4}$$

- **Batch-driven** execution model, followed by Coral8, mentioned in SECRET[6] descriptive model

  In this model, every tuple is assigned an batch-id. Tuples which belong to a batch must have the same timestamp, but two separate tuples 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 $t_{app} = 2$, the system receives 5 tuples $\{s_1, s_2, s_3, s_4, s_5\}$, but they arrive at different $t_{sys}$ respectively. Time-driven model treats them as simultaneous tuples with

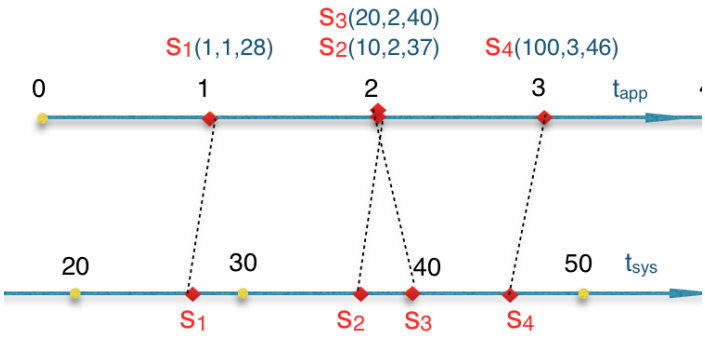
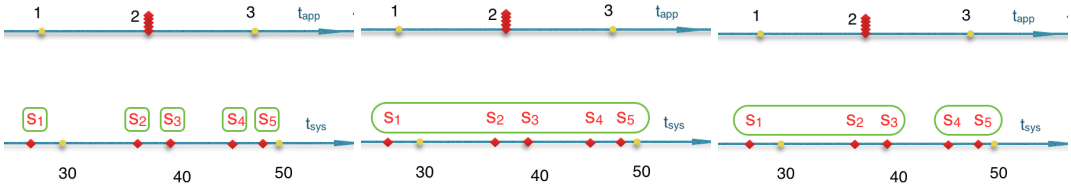



Fig. 3.2 Tuple-driven Execution model

no difference. Tuple-driven considers them as 5 concrete tuples in strict order. And batch-driven model may divide them into 2 batches $\{s_1, s_2, s_3\}$, $\{s_4, s_5\}$ depending on window specification.




(a) Tuple-driven (b) Time-driven (c) Batch-driven

Fig. 3.3 Execution models

We extends the examples in [8] 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)

$$\begin{aligned} Instream(time, value) &= \{s_1(30,10), s_2(31,20), s_3(36,30), ...\} \\ OracleCEP(avg) &= \{(10),(15),(20),...\} \\ Flink(avg) &= \{(10),(15),(15),(15),(15),(20),...\} \\ Coral8(avg) &= \{(10),(15),(20),...\} \end{aligned}$$

   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

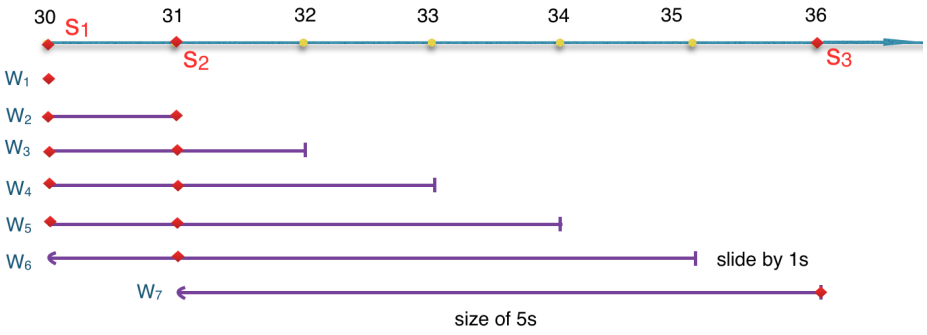



Fig. 3.4 Window Evaluation

   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 $s_2(30,10)$. Similarly, $W_7$ excludes tuple $s_3(31,20)$

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.

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

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

**TODO** http://blog.mikiobraun.de/2014/01/apache-spark.html

## 3.2 Policy-based Window Semantics in Flink

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 re-evaluate a 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 $W_t = (l, u, \omega_t)$ over a stream S is a finite subset of $\mathbb{S}$ containing all data elements $s \in \mathbb{S}$ where $l, u, \omega_t \in \mathbb{T}$ and $l < s.t \leq u$. The length of window in time unit is $\omega_t = u - l$*

   Notice that in a time-based window, $s.t$ can be tuple's application or system timestamp depending on query. Again, there are maybe many simultaneous tuples with an 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 $W_c = (l, u, \omega_c)$ over a stream S is also a finite subset of $\mathbb{S}$ containing all data elements $s \in \mathbb{S}$ where $l, u \in \mathbb{T}$, $\omega_c \in \mathbb{N}$ and $l \leq s.t_{sys} \leq u$. The length of window $\omega_c$ is the number of tuples in interval time $[l, u]$, i.e., $\omega_c = |s \in \mathbb{S}(t_{sys}) : l \leq s.t_{sys} \leq u|$*

   The count-based window $W_c$ is independent from application timestamp $t_{app}$. It is only related to system timstamp $t_{sys}$ which indicates 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 in between.

   Supposed 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 $W' = (l_3, u_3, \_)$ such that $l_1 < l_3 < l_2$.

   **Definition 3.3** *The distance or slide between 2 windows $W_1$ and $W_2$ is the distance between two of their upper-bound tuple as*

   - *A time-based slide $\beta_t = u_2 - u_1$.*
   - *A count-based slide $\beta_c = |s \in \mathbb{S}(t_{sys}) : u_1 < s.t_{sys} \leq u_2|$*

There is a correlation between window size and slide size conforming 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 allows 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 models in Flink

We present here the execution models of Flink Stream Processing engine. Tick model specifies how the engine reacts to new tuple arrival while Window Construction show the way Flink constructs and emits a complete window based on window specifications. There are pretty many reviews on window construction (
**TODO**: which?) ... However, none of them give a full description in the case that we mix up time-based and count-based specification for window size and slide.

Window Buffer *wb* is a linking list contain tuples with composite type *IN*
$size_t(wb) = wb.last.t - wb.first.t$
$size_c(wb) = wb.length()$

### 3.3.1 Tick Model

As we mentioned in 3.1, there are three common Tick execution models[8] implemented in various systems. STREAM and Oracle CEP implements time-driven model which reacts once to all tuples with an identical application timestamp. Coral 8 with batch-driven model takes action on an atomic batch which may contain multiple tuples with the same batch-id. Flink and StreamBase with tuple-driven model actively trigger actions on every new arrived tuple.

In Flink, tuples are strictly totally ordered based on its system-assigned timestamp. We describe a procedure taken place on a recent arrived tuple in method *processNewTuple* (of Algorithm1). System takes action on new tuples one by one due to its moment of arrival.

**Algorithm 1** Process new arrived tuple

The first step, before processing a new arrived tuple, check if a current window buffer should be emitted. 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
$\omega_t$: size of window in time interval
$\omega_c$: size of window in tuple count

```
1: method PROCESSNEWTUPLE(newTuple : IN)
2:     if NOTIFYTRIGGER(newTuple) & size_c(wb) > 0 then        ▷ trigger new window
3:         window ← wb
4:         EMIT(window)                                        ▷ emit to Windowed stream
5:     end if
6:     if window is time-based then
7:         evict_t ← size_t(wb.append(newTuple)) − ω_t
8:         if evict_t > 0 then                                 ▷ remove old tuples
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:    else                                                    ▷ window is count-based
17:        evict_c ← size_c(wb.append(newTuple)) − ω_c
18:        if evict_c > 0 then                                 ▷ remove old tuples
19:            for i ← 1, evict_c do
20:                wb.removeFirst()
21:            end for
22:        end if
23:    end if
24:    wb ← wb.append(newTuple)          ▷ add new tuple to current window buffer
25: end method
```

**Algorithm 2** Whether system should trigger a new window

When a new tuple has arrive, system check whether the current window buffer reached the point where distance of 2 windows is equal to slide size or not. The new tuple is not really appended to buffer, use it for qualifying purpose only.

**Require:** $\beta_c$: count-based slide size
$\beta_t$: time-based slide size
$lastUpperBound$: timestamp of upper boundaries of previous window

```
1:  method NOTIFYTRIGGER(newTuple : IN)
2:      if slide is time-based then
3:          if (newTuple.t − lastUpperBound) > β_t then
4:              lastUpperBound ← lastUpperBound + β_t
5:              return true
6:          else
7:              return false
8:          end if
9:      else                                          ▷ window is count-based
10:         if counter ⩾ β_c then
11:             counter ← 1
12:             return true
13:         else
14:             counter ← counter + 1
15:             return false
16:         end if
17:     end if
18: end method
```

Basically, Flink employs a linked list as a window buffer to temporarily store a queue of arrived tuples. New tuple will be added to window buffer whereas old tuples will be removed from window buffer according to eviction policy.

Whenever a new tuple has arrived, the procedure is as following:

- **Step 1:** The system checks whether the stream reached the point where it should emit the current window buffer to windowed stream and trigger a new window. The condition to trigger a new window is represented in method *notifyTrigger* (in algorithm 2). If the condition is satisfied and window buffer is not empty, system will emit the current window buffer to discretized Windowed Stream for later computation.

  In method *notifyTrigger*, system decides to trigger a new window according to trigger policy defined in window specification. Supposed that the previous emitted window is $W^1$, the current window buffer is *wb*, and new arrived tuple *s*. If the distance between $W^1$ and wb$\bigcup\{s\}$ start exceeding slide size $\beta$ (defined in window specification), system confirms the trigger point and reset the slide measurement:

  - If the slide is time-based, set timestamp of new upper boundaries as *lastUpper-Bound*
  - If the slide is count-based, reset counter to 1

  Notice that system has not inserted new tuple to window buffer yet, but at last step.

- **Step 2:** evict old tuples in window buffer. Supposed that system adds new tuple to window buffer, system evicts a number of oldest tuples so that size of the buffer does not exceed pre-defined window size $\omega$. This eviction policy is activated whenever a new tuple has arrived to ensure that window buffer size never exceed $\omega$

  - if the window is time-based, time interval cover the window is not bigger than $\omega_t$
  - if the window is tuple-based, number of tuples in the window does not exceed $\omega_c$

- **Step 3:** officially insert new tuple to window buffer.

In short, we figure out some crucial properties of Tick model in Flink

- Flink implements tuple-driven model reacting to every new arrived tuple.
- The eviction policy is to keep size of window buffer shrink shrunk to pre-defined window size $\omega$
- One is able to mix up different type of window and slide size. For instance, tuple-based window slide by time interval.

### 3.3.2 Window Constructions

We are able to define window and slide size based on application timestamp, system timestamp or number of tuple. Therefore, we have 9 ways to construct a basic window with combination of window and slide size.

#### Upper boundary

In time-based slide, supposed that $u_1$ is the upper boundary of the first window. The upper boundary of the $k$th window is $u_k = u_1 + k.\beta_t$

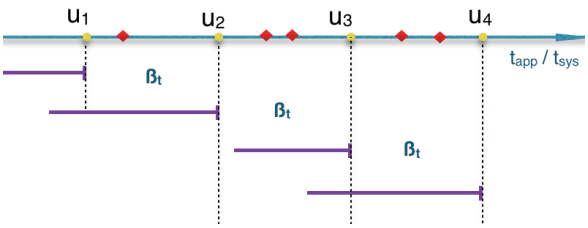



Fig. 3.5 time-based slide

In count-based slide, supposed that $u_1$ is the system timestamp of last element of the first window, the last element $s < v,, t_k >$ of the $k$th window must satisfies $|s \in \mathbb{S}(t_{sys}) : u_1 < s.t_{sys} \leq u_2| = n.\omega_c$

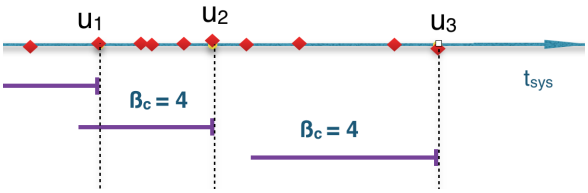



Fig. 3.6 count-based slide

#### Lower boundary

In time-based windows, windows are defined in terms of of timestamp. Supposed that we know the upper bound $u_k$ of the $k$ window. The lower bound $l_k = max(0, u_k - \beta_t)$

- Using application timestamp. The window content is $W = \{s :< v, t_{app}, \_ >\in \mathbb{S} \wedge max(0, u_k - \omega_{t_{app}}) < t_{app} \leq u_k\}$ where $u_k$ is upper boundary of window in application timestamp unit

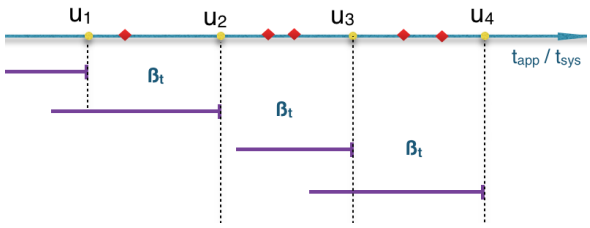



Fig. 3.7 time-based slide

- using system timestamp. The window content is $W = \{s :< v, \_, t_{sys} >\in \mathbb{S} \wedge max(0, u_k - \omega_{t_{sys}}) < t_{sys} \leq u_k\}$ where $u_k$ is upper boundary of window in system timestamp unit

In count-based windows, the scope of window is defined in terms of number of tuples. Thus, given the last tuple $s :< v, \_, u_k >$ of the window. The window content is $W = \{s :< v, \_, t_{sys} >\in \mathbb{S} \wedge t_{sys} \leq u_k \wedge |\{< v, \_, t'_{sys} >\in \mathbb{S} : t_{sys} \leq t'_{sys} \leq u_k\}| \leq \omega_c\}$

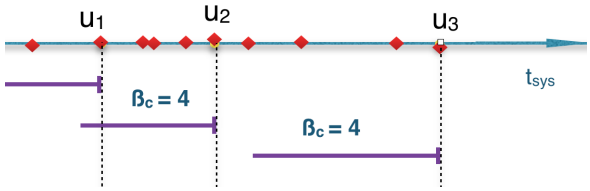



Fig. 3.8 count-based slide

**In case of partitioned group within window**

http://www.sqlstream.com/blog/2015/03/5-reasons-why-spark-streamings-batch-processing-of-data-streams-is-not-stream-processing/

# Chapter 4

# FlinkCQL- Queries over Data Stream

## Query

A query is a request telling system what to do in order to retrieve or alter desired information stored or processed by system. For instances, asking "How many products are sold today? ". Queries over data stream and in a traditional DBMS have a lot in common; however, due to characteristic of continuousness in data stream, we may classify a query as one-time query or continuous query [18] [4].

- **One-time queries** are evaluated once over data set at a given time instant, and terminated after returning its result. This is also called *passive query* [19] since system require queries and passively waits for users to issue these queries before executing.

- **Continuous queries**, in contrast, get evaluated continually as new data arrives to the observed stream. System continuously delivery new results over time according to the snapshot or state of data stream seen so far. Thus the output of queries is not a single result, but rather new streams of results for further operators if desired. Obviously, continuous queries really fit to user's requirements to observed data streams till its end.

## Query Language

Queries are expressed in terms of some query language. Queries provide for users and programmers a very general way to specify data selections, projections, combination, computation and so on over data set/stream. In the meanwhile, users can send the queries using either imperative or declarative language.

**Imperative vs. Declarative language**

- **Imperative language** requires users to define explicitly step by step **how** code should be executed to get **what** they want. They need to break the program into sequences of commands in particular order for the system to perform. Actually, many existing programming languages are imperative supporting *assignemnt*, *for-loop*, *if-else* statements and so on to construct the complete program. For example, assuming that *StockTick* is a list of stock transactions, to filter through *StockTick* to get all stock transactions from 'NYSE' only:

```
function getTransFromNYSE {
  var fromNYSE = []
  for (var i = 0; i < StockTick.length; i++) {
    if (StockTick[i].Exchange == "NYSE")
      fromNYSE.add(StockTick[i])
  }
}
```

- **Declarative language**, like SQL or LINQ, users just specify **what** they want - which sort of data, which transformation they want it to be afterwards, but **not how** to achieve the result. The corresponding declarative program for previous examples is written in SQL language as below:

```
SELECT * from StockTick where Exchange = "NYSE"
```

There are several advantages of declarative over imperative language [11]:

- Declarative language is typically more concise , more friendly and easier to work with. For instance, comparing 2 previous programs, the former has 7 lines of code while the latter goes with 1 line. According to [5], Java program is typically 50 times less compact than SQL query for the same purpose. Generally, that because they have different level of abstraction. Declarative language already hides complicated implementation details. It makes the program much more simpler but possible for system to introduce underneath improvement without any impact on queries.

- Declarative languages, for example SQL, HTML, are usually followed a set of standard syntaxs which make it more limited in functionality, give the system more room for automatic optimization.

- In terms of parallel execution environment, declarative language like SQL has a better chance to be executed faster. Imperative code instructs its sequence of operators to be performed in a certain order so that it is really hard to parallelize programs across distributed system. In the other hand, declarative queries are more atomic to be implemented in parallel if appropriate.

## FlinkCQL - a SQL-like dialect

Recently, there are 2 common form of declarative language applying to data query: SQL and LINQ. However, we decide to extend SQL syntax for our continuous query language due to several advantages:

- Traditional database model and data stream model are distinguished but till share many common features and operators. Since,SQL was a well-designed for handle data in batch mode, extending it to handle data-in-motion in data stream model is a possible incremental approach to quickly define language with less effort.
- SQL is so common and recognized by most of developers. Therefore, extended SQL languare would not challenge them to learn and master it.
- SQL is a standard adopted by most of relational database systems. As a result, their syntax is well-designed and refactored. Moreover, parsers, visualizers and composers for SQL are readily available to extend.

http://www.sqlstream.com/blog/2015/03/why-we-need-sql-for-real-time-streaming-analytics/

There are several properties of data stream which we take into account when extending SQL to FlinkCQL:

- **Language closure**: it ensures that the output of any one query can be input to another
- **Windowing** FlinkCQL allow to define window specification and implement related operators
- **Non-blocking operators** Naturally, blocking operation is not applicable to data stream

We are hereby going to present FlinkCQL syntax on 4.1 and its semantic on 4.2

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[9]

- 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 [4]
Query : Smart Votex , page 2
Window Specification over Data Streams.pdf
[Stream] Evaluate CQL
Designing_Data_Intensive_Applications.pdf

## 4.1 Continuous Query Language

### 4.1.1 Data Type

FlinkCQL supports numbers of data types including numeric types (*Byte*, *Short*, *Int*, *Long*, *Float*, *Double*), *Boolean* type, string types (*Char*, *String*), deta type (*Datetime*) with detailed descriptions in (Table 4.1)

### 4.1.2 Data Definition Language (DDL)

We utilize Extended Backus–Naur Form (EBNF) to make a formal descriptions of FlinkCQL. To understand the syntax, there are some EBNF notations to know

- **[...]** : Expression inside squared brackets is *optional*
- **{...}** : Expression, which is wrapped by curly braces, is omitted or repeated.
- | : alternation (or)

Moreover, be aware that *ident* stands for *Identifier* which is recognized as name of schema, stream, data field and so on.

Table 4.1 Data Type

| FlinkCQL Type | Description | Convertable to |
|---|---|---|
| **String** | A sequence of Chars | |
| **Boolean** | Either the literal true or the literal false | |
| **Char** | 16 bit unsigned Unicode character. Range from U+0000 to U+FFFF | Byte, Short, Integer, Long, Double |
| **Byte** | 8 bit signed value. Range from -128 to 127 | Short, Integer, Long,Float, Double |
| **Short** | 16 bit signed value. Range -32768 to 32767 | Integer, Long, Float, Double |
| **Int** | 32 bit signed value. Range -2147483648 to 2147483647 | Long, Float, Double |
| **Long** | 64 bit signed value. -9223372036854775808 to 9223372036854775807 | Float, Double |
| **Float** | 32 bit IEEE 754 single-precision float | Double |
| **Double** | 64 bit IEEE 754 double-precision float | |
| **Datetime** | 'YYYY-MM-DD HH:MM:SS' format. The supported range is '1000-01-01 00:00:00' to '9999-12-31 23:59:59' | |

**Create Schema**

⟨*schema statement*⟩ ::= CREATE SCHEMA ⟨*schema ident*⟩
(⟨*named schema ident*⟩|⟨*anonymous schema*⟩)
[EXTENDS ⟨*parent schema ident*⟩]

⟨*anonymous schema*⟩ ::= '(' ⟨*typedField*⟩ {',' ⟨*typedField*⟩}')'

⟨*typedField*⟩ ::= ⟨*field ident*⟩ ⟨*data type*⟩

Similar to CREATE TABLE statement in SQL, we are able to identify a schema of stream tuples. Each schema consist of name followed by list of data fields (the combination of field identifier and its data type). For example, we create schema for *StockTick* stream:

```
CREATE SCHEMA StockTickSchema (symbol String, sourceTimestamp Long,
price Double, quantity Int, exchange String)
```

We extends the grammar so that a schema can be referenced or extended to another schema. For examples, in below examples, *StockTickSchema2* is referencing to previous *StockTickSchema* so that they own a similar set of attributes. Meanwhile, *StockTickSchema3* extends from it to have one more attribute (*"id"*)

```
CREATE SCHEMA StockTickSchema2 StockTickSchema
CREATE SCHEMA StockTickSchema3 (id Int) EXTENDS StockTickSchema
```

**Create Stream**

⟨*Stream statement*⟩ ::= CREATE STREAM ⟨*schema ident*⟩
(⟨*named schema ident*⟩|⟨*anonymous schema*⟩ )
[⟨*source*⟩]

⟨*source*⟩ ::= (AS ⟨*derived source*⟩) | ( SOURCE ⟨*raw source*⟩)

⟨*derived source*⟩ ::= ⟨*stream ident*⟩| ⟨*subSelect*⟩

⟨*raw source*⟩ ::= HOST '('⟨*host*⟩, ⟨*port*⟩')'
| FILE '('⟨*file path*⟩, ⟨*delimiter*⟩')'

A stream cannot be queried unless it is registered with a schema, simply because system require users to specify name of attributes for expression in most of cases. For this reason, we allow to entitle a stream to its schema using CREATE STREAM statement. Except for the reserved keywords, the statement consists of 3 parts. Stream name declaration is followed by schema definition and source of stream. Its schema can be recalled from a previously-defined named schema such as *StockTickSchema*.

```
CREATE STREAM StockTick StockTickSchema;
```

One also has abilities to define new schema with a set of attribute name and type. In this case, the stream and its schema share the same name.

```
CREATE STREAM StockTick (symbol String, price Double, quantity Int)
```

The last part is optional source of stream. Recall that we have two kind of stream representations: base stream and derived stream. <*raw source*> clause indicates that this is a base stream obtained through a network connection (<*host*>, <*post*>) or from a text file. For instance, *StockTick* stream originates from host *98.138.253.109* via port *2000*

```
CREATE STREAM StockTick StockTickSchema
SOURCE HOST ("98.138.253.109", 2000)
```

Derived stream may come from another existing stream or output of a query and its operators. <*derived source*> clause indicates these two possibilities. For example, we register a new stream *StockPrice* which is derived from *StockTick* but pay attention to stock symbol and its price only

```
CREATE STREAM StockPrice (symbol String, price Double) AS
SELECT symbol, price
FROM StockTick
```

### 4.1.3 Data Manipulation Language (DML)

#### Insert

⟨*insert statement*⟩ ::= INSERT INTO ⟨*stream ident*⟩ [AS] (⟨*stream ident*⟩ | ⟨*subSelect*⟩)

In CREATE STREAM statement, stream identifier and its schema definition are required but its source is optional. It means that registered stream could attached its source later when it is available. In this case, we take the advantage of INSERT statement to complete stream registration procedure. However, we support INSERT statement for derived stream only. It naturally makes sense because a base stream is concrete and should be permanently registered from beginning. We then can insert it into other stream if possible. Keep in mind that INSERT statement is a complementary to CREATE STREAM statement in case <*source*> is missing. It will not work for a stream identifier which already refer to a real stream source.

```
CREATE STREAM StockTick StockTickSchema;
INSERT INTO StockTick AS stockStream;
```

#### Merge

⟨*merge statement*⟩ ::= MERGE ⟨*stream ident*⟩ ',' ⟨*stream ident*⟩ ',' ⟨*stream ident*⟩

One of data stream properties is that data are emitted by a variety of external sources. It is cumbersome to write a similar query for each of substream. To eliminate this duplication, Flink allows to merge various registered stream with same Schema into one. For example, we integrate all StockTick from several Exchange into one:

```
MERGE stockTickFromNYSE, stockTickFromAMEX, stockTickFromNASDAQ
```

**Split**

⟨*split statement*⟩ ::= ON ⟨*stream ident*⟩
⟨*insert clause*⟩ {, ⟨*insert clause*⟩}

⟨*insert clause*⟩ ::= INSERT INTO ⟨*stream ident*⟩
SELECT ⟨*target entry list*⟩ WHERE ⟨*predicate*⟩

Sometimes, user would like to divide an original stream into several sub-streams according to given criteria. And hence, it is more convenient to observe changes on different sub-streams or sends further queries. Consider the following examples, we classify the original *StockTick* stream and divide into 3 sub-streams based on quantity of transaction.

```
on StockTick
insert into LargeTicks select symbol, price where quantity >= 100000
insert into MediumTicks select symbol, price where quantity between 20000 and 100000
insert into SmallTicks select symbol, price where quantity > 0
```

**Select**

⟨*select statement*⟩ ::= SELECT ⟨*target entry*⟩ {, ⟨*target entry*⟩}
FROM ⟨*stream references*⟩
WHERE ⟨*predicate*⟩
GROUP BY ⟨*field ident*⟩ {,⟨*field ident*⟩}
INTO ⟨*stream ident*⟩

⟨*stream references*⟩ ::= ⟨*stream refererence*⟩ [⟨*join clause*⟩]

⟨*stream reference*⟩ ::= (⟨*stream ident*⟩| ⟨*subSelect*⟩) [‘[’Window specification‘]’]

⟨*join clause*⟩ ::= CROSS JOIN ⟨*stream reference*⟩
| [INNER] JOIN ⟨*stream reference*⟩ (ON ⟨*predicate*⟩ | USING ⟨*field ident*⟩)

⟨*window specification*⟩ ::= SIZE ⟨*spec*⟩
[EVERY ⟨*spec*⟩]
[PARTITIONED BY ⟨*field ident*⟩ {,⟨*field ident*⟩}]

⟨*spec*⟩ ::= ⟨*int*⟩ ON ⟨*field ident*⟩
| ⟨*int*⟩ ⟨*time unit*⟩
| ⟨*int*⟩

The specification of a SELECT query in FlinkCQL resembles the formulation of one-time queries in standard SQL with common SELECT, FROM, WHERE and GROUP BY clause. However, we did enhance the FROM clause with a <*window specification*> to cope with window operators. The semantic of GROUP BY also differ from one in native SQL as well. All changes will be mentioned in details in the following:

**FROM clause**

First, windowing constructs play an crucial role in stream processing and it really makes FlinkCQL distinctive. Our <*window specification*> fully supports all kind of window semantics. It is made up of 3 parts: *SIZE* denotes the window size, *EVERY* indicates the slide size, and *PARTITION BY* is to classify tuples of window into disjoint group. Recall the example in Figure 2.2 which query continuously on windows grouped by Exchange value, over last 3 hours once every 1 hour

```
[
  SIZE 3 hours
  EVERY 1 hours
  PARTITIONED BY Exchange
 ]
```

Furthermore, we assign different formulations of <*spec*> for different measurement unit. Let us consider different windows:

- System-timestamp-based windows: *SIZE 1 min*: means that window captures all tuples last minute of system clock.
- Application-timestamp-based windows: *SIZE 60 on sourceTimestamp*: means window size is 60 seconds based on *sourceTimestamp* attribute
- Count-based windows: *SIZE 100*: means that only last 100 tuples can be buffered in window.

Be aware that, if *EVERY* clause is missing, the window is a tumbling window by default.

Second, window specification has to follow a stream identifier or a sub query which producing a data stream. Consider a query which continuously computes the average value of transaction quantities in *StockTick* stream using tumbling window spanning last 100 transaction:

```
SELECT avg(quantity) FROM StockTick [SIZE 100]
```

If window specification is unavailable, the query is taken place on the scope of stream which can support non-blocking operators only. For examples, the following query is invalid:

```
SELECT avg(quantity) FROM StockTick
```

Third, currently *JOIN* and *CROSS JOIN* operators are supported only on time-based windows. Temporal operators take the current windows of both streams and apply the join/cross logic on these window pairs.

The Join transformation produces a new tuple data stream with two fields. Each tuple holds a joined element of the first input data stream in the first tuple field and a matching element of the second input data stream in the second field for the current window.

The Cross transformation combines two data streams into one stream. It builds all pairwise combinations of the elements of both input data streams in the current window, i.e., it builds a temporal Cartesian product.

#### GROUP BY clause

*GROUP BY* clause in FlinkCQL have a slight different meaning agains one in SQL. This clause in native SQL is used to collect data across data set and group the results by one or more columns. System then can compute some aggregation functios in each group. However, applying *GROUP BY* operator on a data stream results in several sub-streams distinguished by their "group keys". Since the output are streams, we are not able to apply any blocking operators on them. However, we may obtain a stream of partitioned windows (Figure 2.8)when discretizing those sub-streams using window operators.

#### SELECT clause

### 4.1.4 Operators

1. Scala-based Operators
    - Arithmetic
    - Logical
    - Comparison / Relational
    - Bitwise
2. List and Range Operators
    - In / Not in

- Between
- Null

3. String Operators
   - Like
   - Regex
4. Function
   - Aggregate Func
   - Coversion Func
   - Data and Time Func
   - String func

Semantic and Implementation of Continuous Sliding window queries over data streams
[thesis] CQL over data stream [BNF] alias

## 4.2 Continuous Query Semantics and Operators

Semantics of Data Streams and Operators
http://en.wikipedia.org/wiki/Relational_algebra

### Denotation Language

### Lamda Calculus

### 4.2.1 Abstract Syntax

### 4.2.2 Domain

### 4.2.3 Denotation Semantics

### 4.2.4 Standard Operator

- filter $\sigma$
- map $\mu$
- grouping $\gamma$

- union $\bigcup$

### 4.2.5 Window Operators

- Time-based Window
- Count-based Window
- Partitioned Window

# Chapter 5

# Implementations

http://www.sqlstream.com/blog/2015/03/5-reasons-why-spark-streamings-batch-processing-of-data-streams-is-not-stream-processing/

Data Input http://flink.apache.org/news/2015/05/11/Juggling-with-Bits-and-Bytes.html
http://ci.apache.org/projects/flink/flink-docs-master/internals/fig/stack.svg
http://ci.apache.org/projects/flink/flink-docs-master/internals/general_arch.html

# References


[1] (2013). 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.

[5] Beggs, R. (2015). 5 reasons why spark streaming's batch processing of data streams is not stream processing. http://www.sqlstream.com/blog/2015/03/5-reasons-why-spark-streamings-batch-processing-of-data-streams-is-not-stream-processing/ Accessed May 10, 2015.

[6] Botan, I., Derakhshan, R., Dindar, N., Haas, L., Miller, R. J., and Tatbul, N. (2010). Secret: A model for analysis of the execution semantics of stream processing systems. *Proc. VLDB Endow.*, 3(1-2):232–243.

[7] Cherniack, M. and Zdonik, S. (2009). Stream-oriented query languages and operators. In LIU, L. and ÖZSU, M., editors, *Encyclopedia of Database Systems*, pages 2848–2854. Springer US.

[8] 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.

[9] 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.

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

[11] Kleppmann, M. (2014). *Designing Data-Intensive Applications*. O'Reilly Media, Sebastopol, CA, USA.

[12] 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.

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

[14] 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.

[15] 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.

[16] Simovici, D. A. and Djeraba, C. (2008). *Mathematical Tools for Data Mining: Set Theory, Partial Orders, Combinatorics*. Springer Publishing Company, Incorporated, 1 edition.

[17] Takada, M. (2013). *Time and order*.

[18] Terry, D., Goldberg, D., Nichols, D., and Oki, B. (1992). Continuous queries over append-only databases. *SIGMOD Rec.*, 21(2):321–330.

[19] Tore Risch, Robert Kajic, E. Z. J. L. M. J. H.-U. H. (2011). Query language survey and selection criteria. Large Scale Integrating Project, Grant Agreement no.: 257899, Seventh Framework Programme.