Table of contents

List of figures

List of tables

Chapter 1

Introduction

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.

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 [20]. Since that time, people have started generating a lot of different buzzwords around it 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 [31], 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 [2] is one of these frameworks.

Apache Flink

Apache Flink is an open source platform for scalable batch and stream data processing. Several innovative features make Flink standout from the Big Data world:

Fast: Flink exploits in-memory data streaming and integrates iterative processing deeply into the system runtime. This makes the system extremely fast for data-intensive and iterative jobs. Besides Flink also provides many more complex operations like joins, group-by, or reduce-by operations so that users can model quite complex data flows at ease.

Reliable and Scalable: Flink is designed to perform well when memory runs out. Flink contains its own memory management component, serialization framework, and type inference engine.

Easy to Use: Flink requires few configuration parameters. And the system's bult-in optimizer takes care of finding the best way to execute the program in any enviroment.

Compatible with Hadoop: Flink supports all Hadoop input and output formats and data types. You can run your legacy MapReduce operators unmodified and faster on Flink. Flink can read data from HDFS and HBase, and runs on top of YARN

Flink contains APIs in Java and Scala for analyzing data from batch and streaming data sources, as well as its own optimizer and distributed runtime with custom memory management (Figure 1.1)

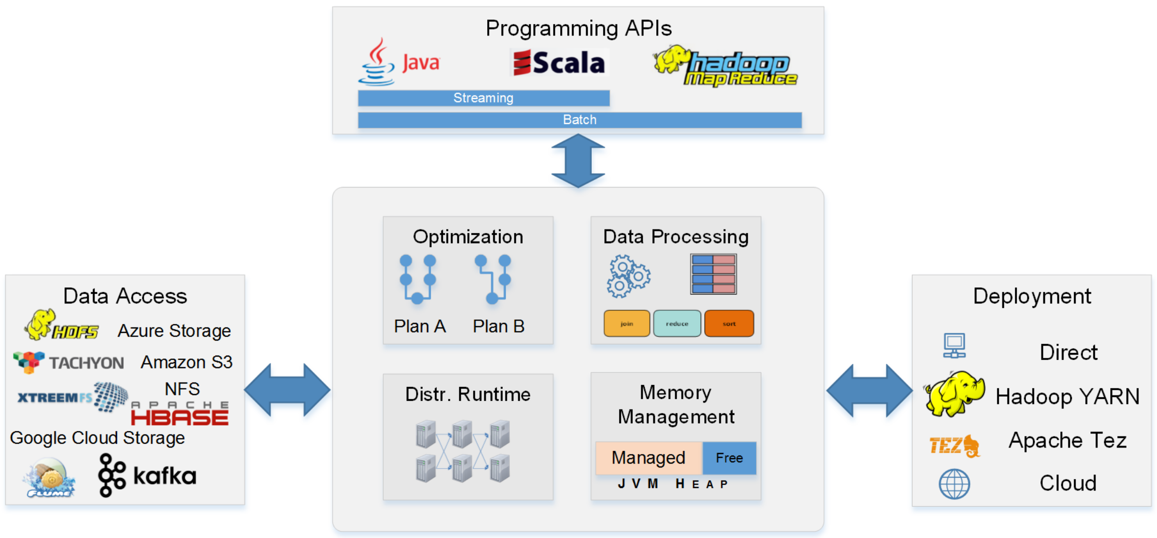


Fig. 1.1 Apache Flink

However, similar to most of big data frameworks providing such rich APIs for imperative programing only, Flink is still struggling to gain traction from enterprise in relation of interaction UX. 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. We tackle solving the problem by building a extended version of ubiquitous SQL language on application layer of Flink. Since SQL is so popular, compact, well-design and easy to use, it is the most suitable choice to rely on for our extension. In the first step, this thesis aims to analyze streaming execution model in order to design and implement an SQL-extension (FlinkCQL - Flink Continuous Query Language) for Flink Stream Processing Engine.

Structure

We present our works in 5 following chapters:

- **Chapter 2:** *Data Stream Model*. The chapter describes the concept of data stream and what are the differences between data stream and traditional database.
- **Chapter 3:** *The execution semantic of Flink Stream Processing*. This part helps to understand how the streaming processing engine works under the hood.
- **Chapter 4:** *FlinkCQL - Queries over Data Stream*. We fully describe the specification of FlinkCQL syntax , as well as its semantics.

- **Chapter 5:** *Implementations.* An tree-based query interpreter is built to translate a input query into a Flink program written in Scala. The program then operates input data streams to produce desired outputs.
- **Chapter 6:** *Conclusions*

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 [28].

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

Definition 2.1 *Paritial order [27] 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 guarantee 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 [27] 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 101^{th} 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 [25].

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 timestamps but system is required to take one element 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 in order to avoid unfortunate random choices which mislead users about data stream's insight.

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 $\mathbb{T}$ 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 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 [12].

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 logs sharing a t_{app} timestamp 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 [18]. 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 postponements 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.

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 [7]. For instance, the line of log in the previous example follows 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 [6]. Furthermore, one can consider a tuple as a partial order mapping a finite subset of attribute names to atomic values [25]. 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 language in STREAM engine [7] 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 φ [24] 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 [12], 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 strict-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 extracts 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 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

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

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 seconds) in next timezone.

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

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: $f : \mathbb{TP} \rightarrow \mathbb{T}^*$ with tuple domain $\mathbb{TP}$, time domain $\mathbb{T}^*$

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

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 Stream may have the following properties [21]:

- 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 priorie.
- 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 terminated.
- 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 [9].

Stream Representations

Base Stream vs. Derived Stream

They distinguish 2 kinds of streams [18] *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 [7]. From now on, we give example queries on input stream named *StockTick* [5] 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. And 30.000 units is sold in the transaction.

```
<IBM,1430468303,81.37,30000,NYSE>
```

The *StockTick* is emitted directly from a 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 stocks with price of more than $100 per 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[21]. 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 by this time, so that all streams are append-only.

Logical Stream vs. Physical Stream

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 parallelism environment. Physical operator is replicated and the internal operator state is partitioned and segmented. Figure 2.1 depicts 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.

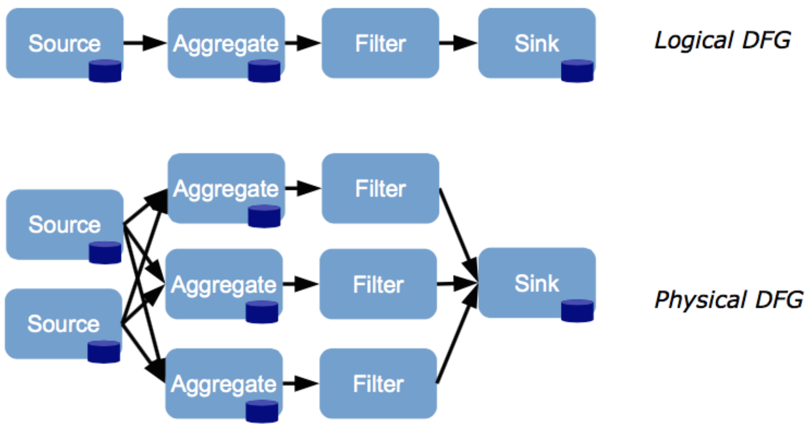


Fig. 2.1 Extract data parallelism from a logical Data Flow Graph(DFG) by replicating operators and segmenting their internal state in the corresponding physical DFG. The disk icon represent an operator's internal state [15]

2.2 Stream Windows

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 may not know exactly 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. When a bounded amount of memory is limited, hence it is barely capable of producing exact answers for data stream queries. Nevertheless, high-quality approximate answers are often acceptable instead.

We have seen many techniques to tackle the problem such as sketches, random sampling, histogram and so on. However, the most preferred is *windowing technique* which continually runs queries over recent portion of data stream in lieu of then entire of its past history. For examples, every 10 minutes, asking for total numbers of transaction happened last hour only. The semantic of window is clear and well-defined that makes it easy to operated by system and understood by users. The size of window is relatively smaller than size of stream so that system can keep latency of computation low. More importantly, from the user's point of view, recent data may be more insightful and informative to make a data-driven decision immediately. Those reasons motivated the use of windows to restrict to the scope of continuous queries. Indeed, 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 [12] over a stream $\mathbb{S}$ is a finite subset of stream $\mathbb{S}$*

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. A 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)

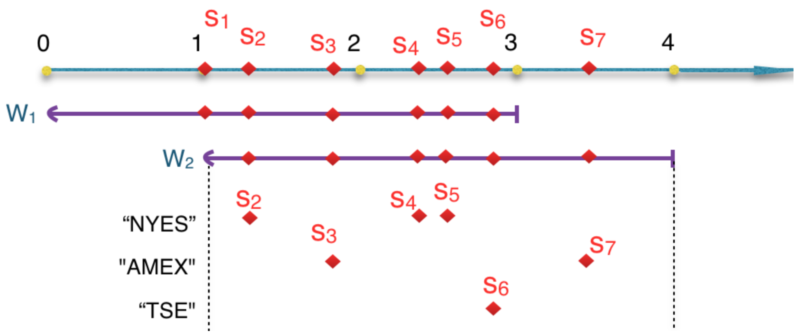


Fig. 2.2 Windowed Stream (i.e., Stream of Windows)

In the example, we obtain 2 windows based on window size, window slide and predicate condition:

- Window 1 : $W_1 : \{S_1, S_2, S_3, S_4, S_5, S_6\}$
- Window 2 : $W_1 : \{S_2, S_3, S_4, S_5, S_6, S_7\}$

However, inside each window, system divides tuples into groups by Exchange keywords. For examples, in window W_2, there are 3 groups of tuples: "NYSE":$\{S_2, S_3, S_4\}$, "AMEX":$\{S_3, S_7\}$, "NYSE":$\{S_6\}$. Aggregation operators such as count, sum, average will be applied to each group. In case the partitioning clause is undefined, those operators will be executed on the whole window W_1, W_2 instead.

Windows can be constructed 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 apply query operators on window buffer. Window may be classified according the following criteria:

2.2.1 Direction of movements

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 captures 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 1 a.m once every hour (Figure 2.3).

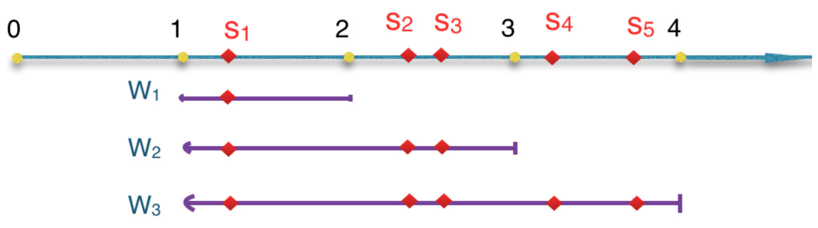


Fig. 2.3 Landmark Window

  Up to 4 a.m, the stream contains 3 windows:

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

- **Sliding window**: the width of the window may be fixed in term of logical unit (i.e., time interval unit) 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. However, windowed stream will still cover all elements on based stream.

  For example, window contains last 2 transactions once every 2 transactions passed. Up to 5 a.m, the stream contains 3 windows (Figure 2.5)

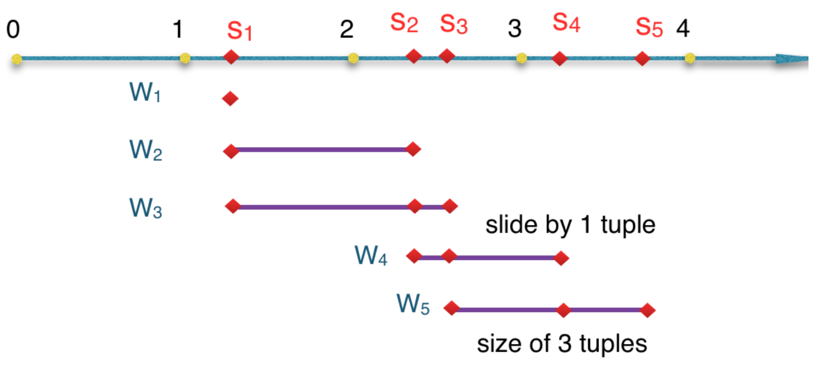


Fig. 2.4 Sliding Window

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

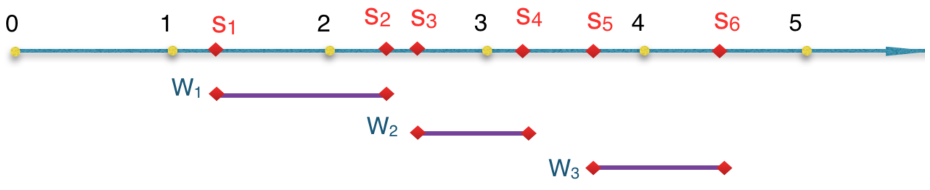


Fig. 2.5 Tumbling Window

- **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 transactions 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 buffer as window W_1.
  - $W_2 : \{s_7, s_8\}$ window buffer evicts $\{s_5, s_6\}$, keeps $\{s_7, s_8\}$ then emits buffer as window W_2.

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 minute of data.

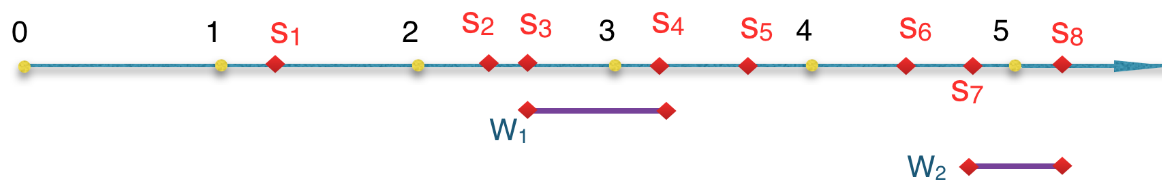


Fig. 2.6 Jumping Window

- **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 them. 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

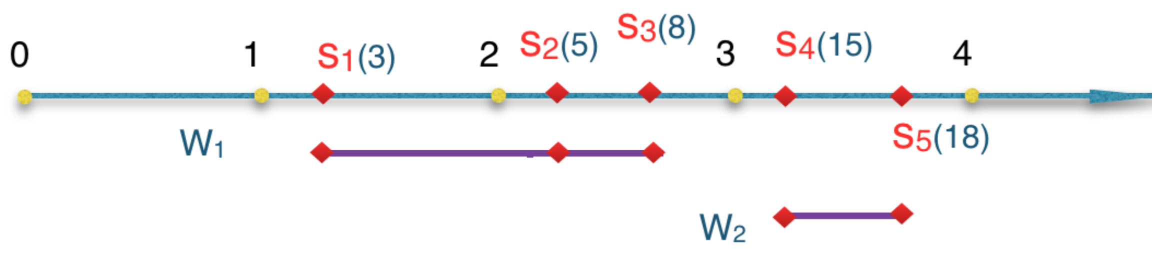


Fig. 2.7 Delta-based Window

- **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 or apply Opertors on the buffer when appropriate.

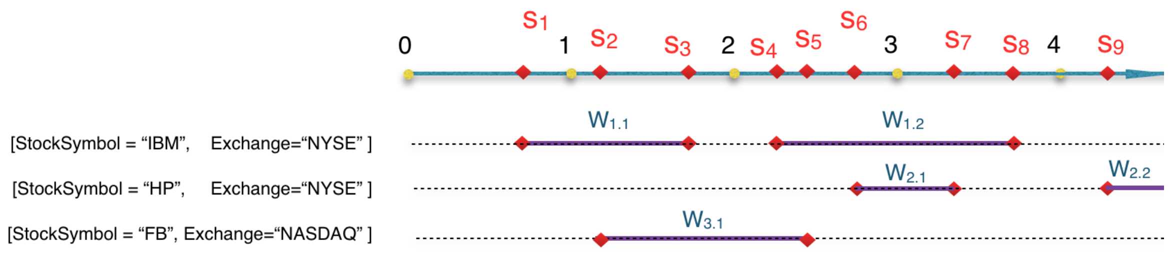

S1
S2
S3
S4
S5
S6
S7
S8
S9
0
1
2
3
4
[StockSymbol = "IBM", Exchange="NYSE"]
[StockSymbol = "HP", Exchange="NYSE"]
[StockSymbol = "FB", Exchange="NASDAQ"]
W1.1
W1.2
W2.1
W2.2
W3.1

Fig. 2.8 Partitioned Window

- **Predicate windows** [14], 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 maintains last 100 transactions which have more than 100.000 units in terms of *volume*, respectively. Every transaction with fewer quantity will be discarded.

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 real-time billing project named "Sunrise" [13], 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 [12]:

1. **Syntax**: Although the ISO/IEC 9075 documentation 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, to refer to a window which captures all event last 10 seconds and advanced by 5 seconds, CQL and FlinkCQL accept different query syntax:

   ```
   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* [5] support pattern matching on stream, whereas STREAM does not.

3. **Execution Model**: Under 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 produces different output which varies based on the

differences on policies of 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 STREAM, Oracle Complex Event Processing (Oracle CPE). In the model, each tuple has 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. We have no control on which one the system will pick.

  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 recalls 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 changes. 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:

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

- **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.

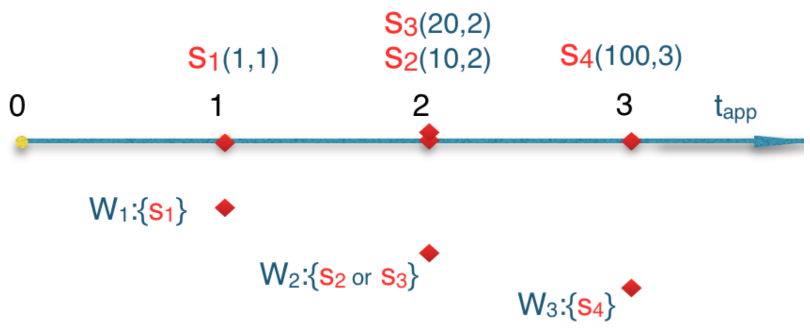


Fig. 3.1 Time-driven Execution model. Window size of 1 tuple

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 start processing the tuples. Executing this *tuple-at-a-time* model, logically Flink places new arriving tuple to a queue and processes it one by one. Therefor, there is at most one tuple considered arriving at a given 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 [12] or a physical identifier [24] instead.

In tuple-driven execution model, each tuple arrival causes 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.

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

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)

$$W_1\{s_1\}, W_2\{s_2\}, W_3\{s_3\}\, W_4\{s_4\} \qquad (3.4)$$

- **Batch-driven** execution model, followed by Coral8 [4], SECRET [11] descriptive models

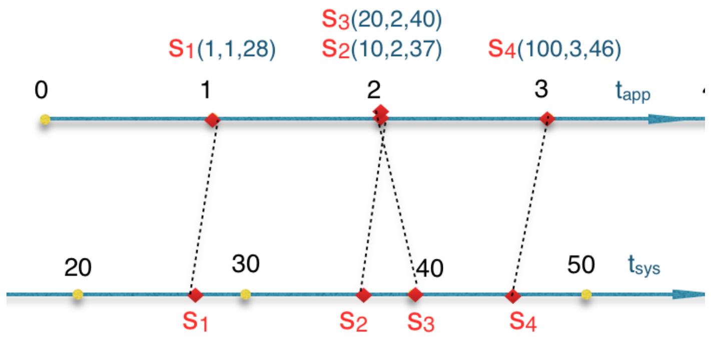
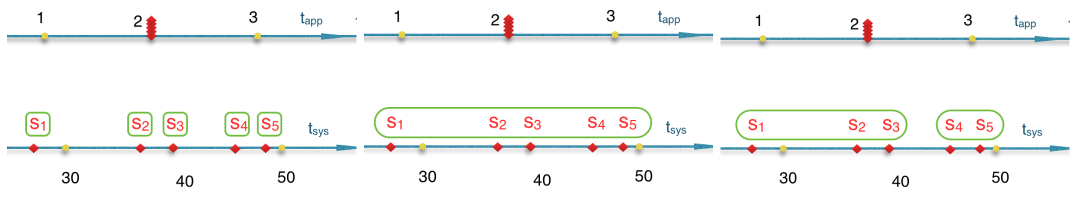


Fig. 3.2 Tuple-driven Execution model

In this model, every tuple is assigned an batch-id. Tuples which belong to a batch must possess a same timestamps, 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 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 [12] with experiments on Flink in order to demonstrate that a system with a different execution model may produce a different output, even with the same input and queries.

1. **Example 1:** *differences in window constructions.*

   Given an *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.

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

Obviously, Oracle CEP constructs the first window with first 3 tuples whereas Flink picks first 2 tuples only. The reason is that Oracle CEP starts constructing a first window when the first tuple has arrived so that the first window always contains 3 first tuples. However, in default, Flink assumes that the first possible window starts from $t = 0$ and Flink emits a window only when it is not empty. In the example, the first non-empty window buffers the tuples arrived at $t_1 = 9$ to $t_3 = 11$. Since there is no tuple at $t_1 = 9$, the first window contains only 2 tuples $\{(10,10),(11,20)\}$.

From the second window, they both take next 3 tuples according to window specifications. We implemented the test on Flink with default configuration, however we are able to customize the upper bound of the first window ($startTime = 12$)so that Flink 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 5 seconds that slides by 1 second)

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

Flink produced a different result than Oracle CEP and Coral8. In Oracle CEP 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

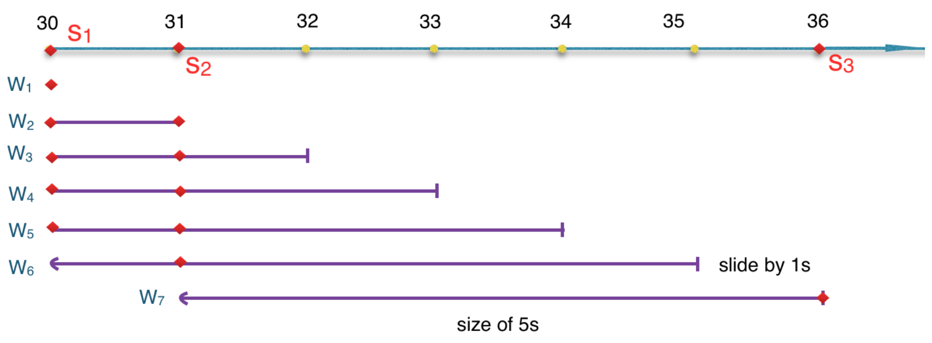


Fig. 3.4 Window Evaluation

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.

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

   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.

Recently, Apache Spark [3] and Apache Storm Trident [1] have introduced a technique called *micro-batching* [32]. In which, operators treat a stream as a continuous series of small batches or chunks of data covered within a tiny time internals. The basic abstraction of Spark streaming , a discretized stream, is represented by a infinite sequence of RDDs (Resilient Distributed Datasets) which is immutable and distributed storage abstraction of Spark. Each RDD contains data from a certain interval. In short, the granular level of Spark stream is a micro-batch of tuples, rather than a individual tuple which is presented in previous execution model. It seems that limiting abstract level of stream to a batch, it makes less sense of a true record-by-record processing [26].

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 ω_c is the number of tuples in interval time $[l, u]$, i.e., $\omega_c = |s \in \mathbb{S} : l \leq s.t_{sys} \leq u|$*

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 previous 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 $W_1 = (l_1, u_1, \omega)$ and $W_2 = (l_2, u_2, \omega)$ where $l_1 < l_2$. There is no window $W' = (l_3, u_3, \omega)$ 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 $\beta_c = |s \in \mathbb{S} : u_1 < s.t_{sys} \leq u_2|$*

There is a correlation between window size ω and slide size β conforming to the movement type of windows.

- Sliding window: $\omega > \beta$
- Tumbling window: $\omega = \beta$
- Jumping window: $\omega < \beta$

However, Flink provide flexibility in mixing up 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 constructions([18], [8], [14], [19], [12],[23], [16]). However, none of them give a full description in the case that we combine time-based and count-based specification for window size and slide.

In this section, we will fully describe Flink execution model in details.

As we mentioned, basically, Flink engine borrows a window buffer to construct a window. Window Buffer *wb* is a linking list contain tuples with composite type *IN*. Flink is able to add a new arriving tuple to the end of window buffer, remove obsolete tuples at the beginning of window buffer or emit a whole buffer to derived stream of windows.

We define a logical size of window buffer according to the range of timestamp. The logical size is used when window is specified by timestamp such as window of last 5 seconds e.g., logical size of window is 5 seconds:

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

In other hand, we also state that physical size of window buffer is the number of tuples stored in buffer *wb*:

$size_c(wb) = wb.length$

3.3.1 Tick Model

As we mentioned in 3.1, there are three common Tick execution models[12] 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 arriving 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* (in Algorithm 1). System takes action on new tuples one by one due to its moment of arrival. Again, Flink employs 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:

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:         $evict_t \leftarrow size_t(\text{wb.append(newTuple)}) - \omega_t$
 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 $e.t \leq$ lastEvictedTimestamp then
12:                     $wb.remove(e)$
13:                 end if
14:             end for
15:         end if
16:     else                                                      ▷ window is count-based
17:         $evict_c \leftarrow size_c(\text{wb.append(newTuple)}) - \omega_c$
18:         if $evict_c > 0$ then                                    ▷ remove old tuples
19:             for $i \leftarrow 1, evict_c$ do
20:                 $wb.removeFirst()$
21:             end for
22:         end if
23:     end if
24:     $wb \leftarrow wb.append(newTuple)$              ▷ 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 $(\text{newTuple.t} - lastUpperBound) > \beta_t$ then
 4:             lastUpperBound $\leftarrow$ lastUpperBound $+ \beta_t$
 5:             return true
 6:         else
 7:             return false
 8:         end if
 9:     else                                          ▷ window is count-based
10:         if counter $\geqslant \beta_c$ then
11:             counter $\leftarrow 1$
12:             return true
13:         else
14:             counter $\leftarrow$ counter $+ 1$
15:             return false
16:         end if
17:     end if
18: end method
```

- **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 $(wb \bigcup \{s\})$ starts exceeding slide size β (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 *lastUpperBound*
  - If the slide is count-based, reset counter to 1

  Notice that system has not inserted new tuple to window buffer yet, but at **Step 3**.

- **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 ω. This eviction policy is activated whenever a new tuple has arrived to ensure that window buffer size never exceed ω

  - if the window is time-based, time interval cover the window is not bigger than ω_t
  - if the window is tuple-based, number of tuples in the window does not exceed ω_c

- **Step 3:** officially inserts 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 arriving tuple.

- The eviction policy is to keep size of window buffer shrunk to pre-defined window size ω. Flink executes its eviction policy whenever a new tuple has arrived but executes its trigger policy only when *notifyTrigger* function returns a true.

- One is able to mix different type of windows and slide. 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 one of 3 factors: application timestamp, system timestamp or number of tuples. Therefore, we have 9 ways to construct a basic window with combination of window and slide size. Naturally, each window will have a upper bound an a lower bound. This session will show how they are related to window size and slide.

Upper boundary

In time-based slide, supposed that u_1 is the upper boundary of the first window and β_t is the slide size in either system or application timestamp value. According to the definition of window slide (3.3),we deduce that the upper boundary of the k^{th} window is $u_k = u_1 + k.\beta_t$

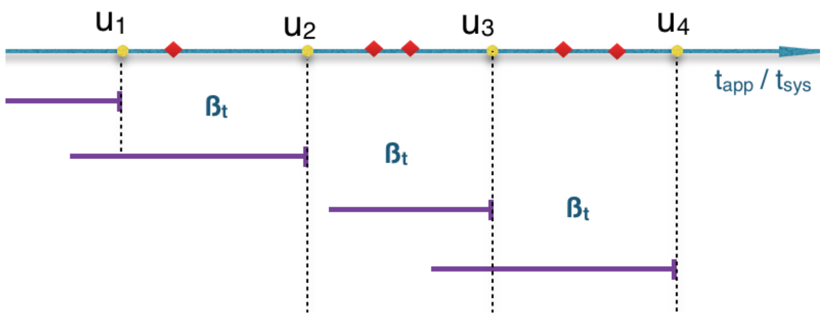


Fig. 3.5 Time-based slide

Similarly, 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 $|s \in \mathbb{S}(t_{sys}) : u_1 < s.t_{sys} \leq u_2| = k.\omega_c$

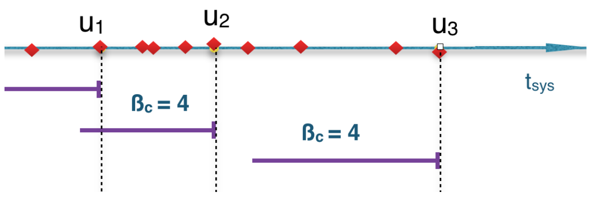


Fig. 3.6 Count-based slide

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 $l_k = max(0, u_k - \omega_t)$ with ω_t is the logical size of windows in timestamp.

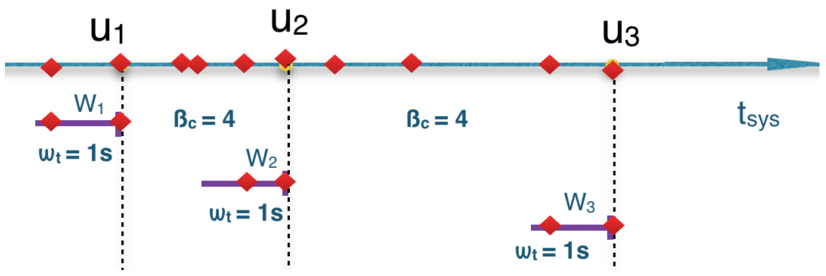


Fig. 3.7 Time-based window

- 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 and $\omega_{t_{app}}$ is the size of window in application timestamp unit
- 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 and $\omega_{t_{sys}}$ is the size 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 $W = \{s :< v, \_, t_{sys} > \in \mathbb{S} \wedge t_{sys} \leq u_k \wedge |\{< v, \_, t'_{sys} > \in \mathbb{S} : t_{sys} \leq t'_{sys} \leq u_k\}| \leq \omega_c\}$

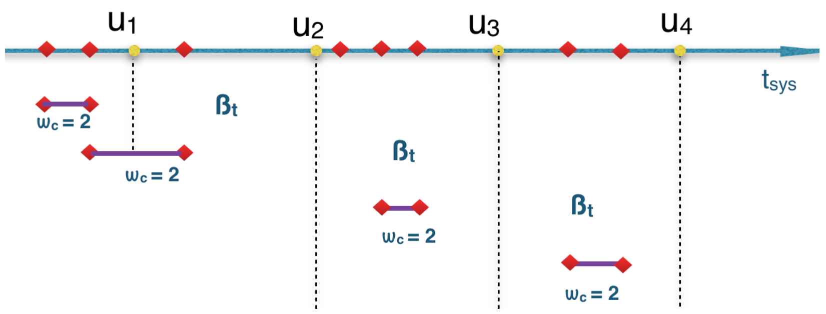


Fig. 3.8 Count-based window

Profoundly understand the scope of windows, one is able to alter the window specification to match the needs. Otherwise, one will not understand which data they are working on.

Chapter 4

FlinkCQL- Queries over Data Stream

4.1 Fundamental of query languages

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 [29] [9].

- **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* [30] 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 languages. They 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 *assignment*, *for-loop*, *if-else* statements and so on to construct a 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 [17]:

- 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 [10], Java program is typically 50 times less compact than SQL query for the same purpose. Generally, that is 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 syntax 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 for Flink Streaming Processing

Recently, there are 2 common forms 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 is 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 quality-guaranteed to use. Moreover, parsers, visualizers and composers for SQL are readily available to extend.

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

- **Language closure**: In Relational algebra, closure property states that each operation takes one or more input relations and return an output relation. Thanks to this property, we are able to write nested relational expression. For example, a nested *Select* query in *From* clause or in *Insert* statement. Therefore, to give more room for user to create a complex query at once, FlinkCQL should support nested queries and pay attention to the closure property.

- **Windowing** FlinkCQL should allow to define window specifications and implement related operators. There is no doubt that Windowing is a essential technique in Stream Processing, not only because it is a mean to find a approximate query answering but also because it is strongly inspired from users' requirements.

- **Blocking and Non-blocking operators** Naturally, blocking operation is not applicable to data stream. According to [9]:

  Definition 4.1 *A blocking query operator is a query operator that unable to produce the first tuple of its output until it has seen its entire input.*

  Sorting, joining or aggregation such as SUM, COUNT, MIN, MAX, AVG are examples of blocking operators. Obviously, the end point of a data stream is nearly unpredictable so that these blocking operators are not very suitable to the data stream model.

  In contrast, non-blocking operators, which can produce some tuples of output before it has detected the end of input, can be applied on data stream. Some of non-blocking operators such as projection, selection, partition, merge, split stream, can process new tuples on-the-fly without storing any temporary results (stateless).

  However, it would be a huge drawback if blocking operators are missing in query language since they are very common for simple analytics. Fortunately, since a window contains a finite set of tuples, we are able to implement a blocking operators on each window in the lieu of the whole stream.

  In short, blocking operators are possible to query on Windows only while non-blocking operators can be applied on both concrete data streams or windows.

I propose FlinkCQL and describe its syntax on session 4.2 then its semantic on session 4.3

4.2 Continuous Query Language

4.2.1 Data Type

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

4.2.2 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.

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'	

- | : alternation (or)

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

Create Schema

$\langle$*schema statement*$\rangle$ ::= CREATE SCHEMA $\langle$*schema ident*$\rangle$
($\langle$*named schema ident*$\rangle$|$\langle$*anonymous schema*$\rangle$)
[EXTENDS $\langle$*parent schema ident*$\rangle$]

$\langle$*anonymous schema*$\rangle$ ::= '(' $\langle$*typedAttribute*$\rangle$ {',' $\langle$*typedAttribute*$\rangle$}')'

$\langle$*typedAttribute*$\rangle$ ::= $\langle$*attribute ident*$\rangle$ $\langle$*data type*$\rangle$

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 attributes (the combination of attribute 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*⟩	::=	⟨*subSelect*⟩ | ⟨*merge*⟩
⟨*raw source*⟩	::=	SOCKET '(' ⟨*host*⟩, ⟨*port*⟩ [, ⟨*delimiter*⟩]')'
	|	FILE '(' ⟨*file path*⟩ [, ⟨*delimiter*⟩]')'
	|	STREAM '(' ⟨*stream name*⟩ ')'

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 SOCKET ("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
```

4.2.3 Data Manipulation Language (DML)

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
```

Insert

⟨*insert statement*⟩ ::= INSERT INTO ⟨*stream ident*⟩ [AS] (⟨*merge*⟩ | ⟨*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
MERGE stockTickFromNYSE, stockTickFromAMEX;
```

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 ⟨*attribute ident*⟩ {,⟨*attribute 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 ⟨*attribute ident*⟩)

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

⟨*spec*⟩ ::= ⟨*int*⟩ ON ⟨*attribute 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, window specification refer to 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 functions 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.*HAVING* clause is also not applicable in this case. However, we may obtain a stream of partitioned windows (Figure 2.8) when discretizing those sub-streams using window operators. We present here the query to compute continuously average of transaction quantities in partitioned windows at (Figure 2.8)

```
SELECT ave(quantity)
FROM StockTick [SIZE 2]
GROUP BY symbol, exchange
```

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 Function
    - Conversion Function
    - Data and Time Function
    - String Function

4.3 Continuous Query Semantics and Operators

Streams and Relations

As we mentioned, it is nearly impossible to perform non-blocking operators on data streams due to its infinity or velocity. However, those operators usually bring up many useful and concise insights about the recent state of the stream such as joining, aggregation and so on. Stream processing engine offers windowing techniques to restrict the scope of the operators to a finite set of elements from underlying stream. The set of tuples within a window is really closed to concept of relations in traditional database system in terms of

- the size of data set is finite so that system can perform all relation-to-relation operators on it.
- Each element of data set belongs to a predefined Schema.
- The order of elements is not a matter, since most of non-blocking operators treat every element equally.

For that reason, the concept of relation in data stream was first proposed in CQL [7] for further analysis.

Definition 4.2 *A relation R is a partial function mapping from each time instant $t \in \mathbb{T}$ to a finite but unbound set of tuples belonging to the schema of R.*

In our model, we make a clear different between application and system timestamp. $R(t_{app} = \tau)$ denotes an unordered set of tuples at any application time instant τ in a condition that $R(t_{app} = -1) = \emptyset$ since function f is not defined. $R(t_{sys} = \tau)$ also implies the tuple at the system time instant if existing because there is at most one tuples at that given time instant.

Querying operators on FlinkCQL falls into 4 classes (Figure 4.1):

- **Stream-to-Relation** operators: are based on windowing technique. It takes a stream as an input then produces a relation. System continuously buffers and reflects the interested part of the stream as a relation for further process. Relation of a window can be thought of as the set of all tuples within the window.
- **Relation-to-Relation** operators: produce a output relation from one or more input relations. The operators are directly inherited from the standard SQL operators as a whole.
- **Relation-to-Stream** operator is responsible for producing a output stream from a input relation. Flink provides a flatten function to unwrap a windows and convert a windowed stream to normal data stream. Thanks to this function, FlinkCQL perform a single Relation-To-Stream right after finishing *projection* operator (specified on SELECT clause), and produce a new output stream.
- **Stream-to-Stream** operators: Three of previous operator classes is fully support in CQL. However, it does not support direct operators between stream. To do so, system transforms a stream into relation, perform a chain of relational operators then convert back to stream format. In contrast, Flink allows to transform directly from input streams to output streams on all non-blocking operators and preserves the order of elements in stream.

Stream-to-Stream

1. Projection $\mathscr{M}_{S2S}[\![Select(A_1,A_2,..,A_k)]\!]$
   $= \lambda_S.\{< (v.A_1, v.A_2, ..., v.A_k), \tau_{app}, \tau_{sys} >: \forall s < v, \tau_{app}, \tau_{sys} > \in S$
   $\wedge (A_1,A_2,..,A_k) \subset Schema\}$
2. Selection

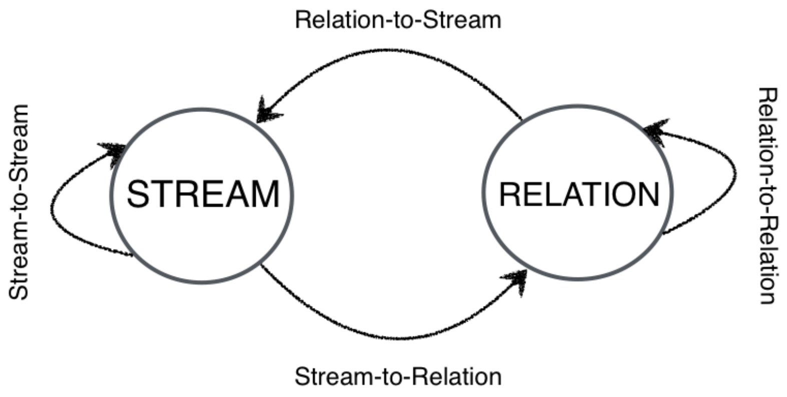


Fig. 4.1 FlinkCQL operators

$$\mathscr{M}_{S2S}[\![Where(p)]\!] \\ = \lambda_S.\{s : \forall s \in S \wedge p(s.v) = true\}$$

3. Merge/Union

$$\mathscr{M}_{S2S}[\![Merge]\!] \\ = \lambda_{S_1}\lambda_{S_1}...\lambda_{S_k}.\{s : s \in S_1 \vee s \in S_1 \vee ... \vee s \in S_k\}$$

4. Split

$$\mathscr{M}_{S2S}[\![Split]\!] \\ = \lambda_S.\lambda_f.\{s : s \in S \wedge f(s.v) = true\}$$

5. Grouping

$$\mathscr{M}_{S2S}[\![GroupBy(A_1, A_2, .., A_k)]\!] \\ = \lambda_S.\lambda_{a_1}.\lambda_{a_2}...\lambda_{a_k}.\{s : s \in S \wedge s.v.A_i = a_i for \forall i \in [1,k] \\ \wedge (A_1, A_2, .., A_k) \subset Schema\}$$

Stream-to-Relation

1. Application-time based

$$\mathscr{M}_{S2R}[\![Size\, T\, on A]\!] \\ = \lambda_S.\lambda_{t_{app}}\{v : s < v, t'_{app}, t_{sys} > \in S \wedge t'_{app} = v.A \\ max(T - t_{app}, 0) < t'_{app} \leq t_{app}\}$$

2. System-time based

$$\mathscr{M}_{S2R}[\![Size\, T\, milliseconds]\!] \\ = \lambda_S.\lambda_{t_{sys}}.\{v : s < v, t_{app}, t'_{sys} > \in S \wedge max(T - t_{sys}, 0) < t'_{sys} \leq t_{sys}\}$$

3. Count-based

$$\mathscr{M}_{S2R}[\![Size\,N]\!]$$
$$= \lambda_S.\lambda_{t_{sys}}\{v : s < v, t_{app}, t'_{sys} > \in S \wedge (t'_{sys} \leq t_{sys})$$
$$\wedge (N \geqslant |\{< e'', t''_{app}, t''_{sys} > \in S : t'_{sys} \leq t''_{sys} \leq t_{sys}\}|)\}$$

Relation-to-Relation

1. Cross Join

$$\mathscr{M}_{R2R}[\![CrossJoin]\!]$$
$$= \lambda_{E_1}.\lambda_{E_2}.\{(e_1, e_2) : e_1 \in E_1 \wedge e_2 \in E_2\}$$

2. Inner Join

$$\mathscr{M}_{R2R}[\![InnerJoin(a, b)]\!]$$
$$= \lambda_{E_1}.\lambda_{E_2}.\{(e_1, e_2) : e_1 \in E_1 \wedge e_2 \in E_2 \wedge e_1.a = e_2.b \wedge a \in Schema(e_1)$$
$$= \wedge b \in Schema(e_2)\}$$

3. Grouping

$$\mathscr{M}_{R2R}[\![GroupBy(A_1, A_2, .., A_k)Selection(AggFunc)]\!]$$
$$= \lambda_E.\lambda_{a_1}.\lambda_{a_2}...\lambda_{a_k}.\{AggFunc(E) \wedge \forall e \in E, \forall i \in [1, k]e.A_i = a_i$$
$$\wedge (A_1, A_2, .., A_k) \subset Schema(e)\}$$

4. Projection

$$\mathscr{M}_{R2R}[\![Select(A_1, A_2, .., A_k)]\!]$$
$$= \lambda_E.\{(e.A_1, e.A_2, ..., e.A_k) : e \in E$$
$$\wedge (A_1, A_2, .., A_k) \subset Schema(e)\}$$

Chapter 5

Implementations

5.1 Architecture

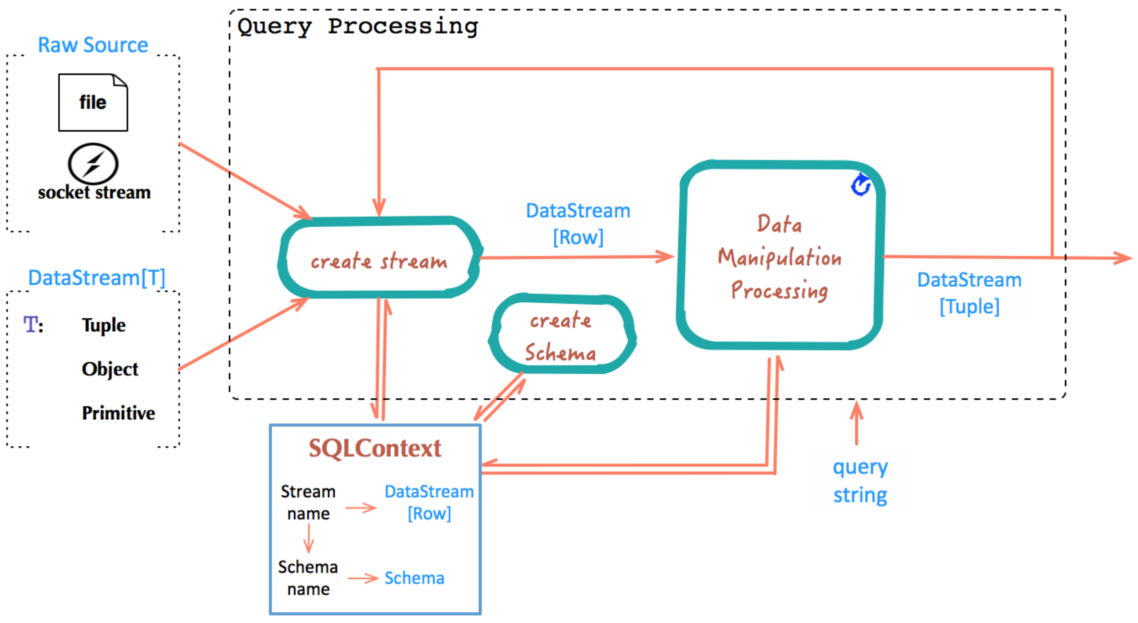


Fig. 5.1 Architecture

Figure 5.1 depicts the overall architecture of FlinkCQL query processing layer. It accepts data streams, a query string and a SQLContext as inputs and return one or more output data streams.

5.1.1 Input Sources

The query processing layer can connect to and process data streams from different data sources like file sources, web sockets, message queues (Apache Kafka, RabbitMQ, Twitter Streaming API . . .), and also from any user-defined data sources. We classify the sources into 2 categories: *Raw source* and predefined *DataStream[T]* with *T* is the type of stream elements.

Raw Source

Text file stream: the source stream contains the lines of the files created (or modified) in a given directory. The system continuously monitors the given path, and processes any new files or modifications. The file will be read with the system's default character set. FlinkCQL expresses the text file stream source via "SOURCE FILE (*filePath, delimiter*)", the default delimiter is a comma. The delimiter is used to tokenize string and convert its result into tuples according to its schema

Socket text stream: the source stream contains the strings received from the given socket. Strings are decoded by the system's default character set. Socket text stream is specified in FlinkCQL as "SOURCE SOCKET (*filePath, delimiter*)". The user can optionally set a delimiter for the same purpose as in Text file stream.

Message queue connectors: There are pre-implemented connectors for a number of popular message queue services. Connectors provide an interface for accessing data from various third party sources (message queues). Currently three connectors are natively supported, namely Apache Kafka, RabbitMQ and the Twitter Streaming API. In the recent prototype of FlinkCQL, we have not designed any syntax for those connectors due to lacking of testing facilities. However, in the next prototype, we could extend its syntax at ease.

DataStream[T]

The *DataStream[T]* is the basic data abstraction provided by Flink Streaming. It represents a continuous, parallel, immutable stream of data of a certain type *T*. Type *T* could be :

- Primitive data type: integer, double, ...
- Tuple type: which is a composite of multiple primitive data types such as *(integer, string, integer)*
- Class object: such as the instances of class *CarEvent(id: Int, speed: Int)*

5.1.2 SQL Context

At the beginning, we encounter a problem of diverse data types of stream elements. A source stream can contains elements of string type (as in raw source), other primitive types, tuple or class instance. It is cumbersome and impractical to provide query processing for all kinds of source streams respectively. Thus, when a data stream is entitled to a schema (via *Create Stream* or *Insert* statement), we first transform the origin source stream to a Data Stream of a universal type *Row* . *Row* is simply a tuple containing *an array of any data* including *Null*. We do not specify the type of elements inside the Row. Those types will be derived from Schema.

Recall the example in 4.2.2 to create schema and stream of "StockTick".

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

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

"StockTick" is the Stream used inside FlinkCQL query. It is entitled to "StockTickSchema" and mapped to a real *DataStream[Row]* which is derived from the socket text stream. The correlation between Stream and Schema is expressed in Figure 5.2. Be aware that the concept of *Stream* in FlinkSQL and *DataStream* in Flink is not identical. In fact, a *Stream* in FlinkSQL is a name to represent a *DataStream[T]* in Flink.

All Stream-Schema-DataStream relationships are stored in 3 dictionaries of *SQLContext* (Figure 5.1). Stream and Schema are unique but many Stream can share a Schema. And a Stream is mapped to a DataStream[T]. When users send out some further queries, query processing framework will lookup *SQLContext* to decide which Stream-Schema is specified to generate an precise snippet of code.

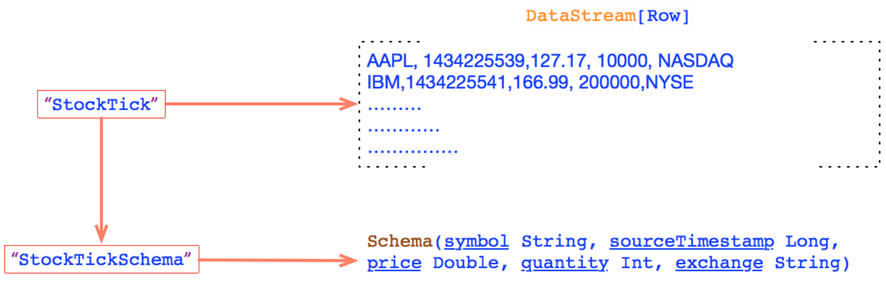

DataStream[Row]
AAPL, 1434225539,127.17, 10000, NASDAQ
IBM,1434225541,166.99, 200000,NYSE
"StockTick"
"StockTickSchema"
Schema(symbol String, sourceTimestamp Long,
price Double, quantity Int, exchange String)

Fig. 5.2 *Stream-Schema* mapping in *SQLContext*

5.1.3 Data Manipulation Processing

Data Manipulation Processing is used to process Data Manipulation Queries such as *SELECT, MERGE, SPLIT, INSERT*. These queries specify which operators to process on given Streams and its Fields. First, system will first look up its SQLContext to retrieve the actual input *DataStream[Row]* according to the *Streams*. Second, Operators and *Schemas* are together to decide how to transform input *DataStreams* to a new one. The output of processing is either one or more Streams which are updated into *SQLContext* or returned as a DataStream of tuple. This DataStream of tuple can be consumed to create other Stream via *CREATE STREAM* query. For examples, the output of *SELECT* query is a DataStream of tuple but the outputs of *SPLIT* query are more than one *Stream* (along with its *Schema* and *DataStream[Row]*).

5.2 FlinkCQL Query Interpreter

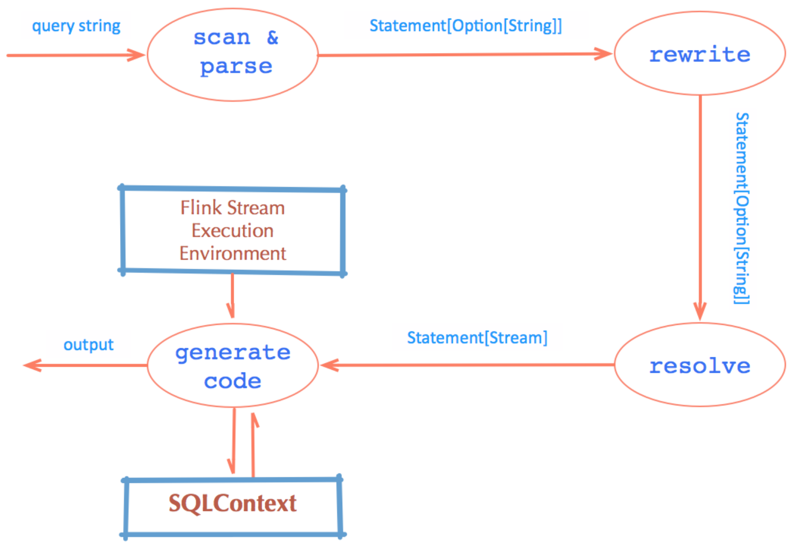

query string
scan & parse
Statement[Option[String]]
rewrite
Statement[Option[String]]
Flink Stream Execution Environment
output
generate code
Statement[Stream]
resolve
SQLContext

Fig. 5.3 Query Processing

Flink Stream Processing is offering a rich set of API to deal with one or multiple input *DataStream[T]*. Our goal is to build an extension which accepts FlinkCQL query strings as commands and translate the commands to corresponding programs constructed with Flink's APIs. Thus we are simply required to build a interpreter instead of a compiler from scratch. For this purpose, I have implemented a tree-based interpreter [22] to process FlinkCQL queries. The idea is that the query interpreter executes program by constructing an abstract syntax tree (AST) from the input query string and walking through the tree to generate a executable Flink program. More specifically, AST is a form of intermediate representation of the hierarchical syntactical structure of the source program written in query.

The interpretation proceed through a series of well-defined phrases (Figure 5.3). Each phrase transforms or extracts information of use from the output of previous phrase and return a result which is a input of next phrase. In first proof-of-concept prototype, we build a pipeline consisting of 4 steps:

- **Scanning and parsing:** serve to recognize the structure of the program, without regard to its meaning. This steps produce an AST with its root is highest semantic level of query : *Statement[Option[String]]*

- **Rewriting** is to modify a sub-tree of AST to a more compact, optimal AST without changing the semantic of program.

- **Resolving:** It is possible that two or more streams, which are specified in query, shares the same Schema. Thus, there is a potential ambiguity between two fields which have a identical name but belong to different streams. This phrase is to figure out to which fields refers. It is clear that we cannot generate code from an ambiguous AST. The output of this step is a *Statement[Stream]* once we resolve all ambiguous Field-Stream relationships.

- **Code generation**: In addition to a compact and unambiguous AST, the interpreter also lookup Stream-Schema information from SQLContext to generate a Flink program based on pre-defined translation rules. Last but not at least, we need the Flink stream execution environment to execute our generated code.

  I describe all phrases processing a simple query:

  ```
  SELECT s.Quantity * Price, StockTick.Symbol
  FROM StockTick AS s
  ```

5.2.1 Scanning and Parsing

This phrase is composed of 2 smaller steps : Scanning (lexical analysis) and parsing (syntax analysis).

Scanner reads characters from the query input and use regex patterns to groups them into *lexemes*, which are the lexically meaningful units of the program, and produces as output tokens representing these lexemes. A token consists of two components: a token class and value. For example, in token <*keyword, "select"*>, its class is *keyword* and its value is "select".

In FlinkCQL, we utilize several token classes which described in Table 5.1

Table 5.1 Token classes

Token class	Description
Identifier	strings of letters or digits, starting with a letter. Regex:
Keyword	reserved keywords which cannot be declared as an Identifier. Example: select, from , as, insert, create, stream, schema,...
Integer	a non-empty string of digits with sign (optional)
DecimalNumber	An integer followed by a decimal point and fractional part. One of the parts can be missing except for the decimal point. Example: 2.2, 3., .1
floatingPointNumber	represented approximately to a fixed number of significant digits (the significand) and scaled using an exponent. For example, $3E-5$
stringLiteral	Single quotes enclosing a sequence of string. For example: 'IBM', 'AAPL'
Whitespace	a non-empty sequence of blanks, newlines, and tabs.
Other	addition strings to make up the query. For example: "(", ")", "." , ","

The regex pattern of these tokens are supported in *JavaTokenParsers* class in Scala language. The output of this steps is a sequence of tokens. For examples, the list of tokens derived from the previous query sample (ignoring all Whitespace tokens):

```
<keyword, "SELECT">, <identifier, "s">, <other, ".">,
<identifier, "Quantity">, <identifier, "*">,
<identifier, "Price">, <other, ",">,
<identifier, "StockTick">, <other, ".">,
<identifier, "Symbol">, <keywork, "FROM">,
<identifier, "StockTick">, <keywork, "AS">,
<identifier, "s">
```

Oviously, scanner helps reduce the size of the input (there are many more characters than tokens) and remove extraneous characters like white space. Thus , scanning is considered a pre-processing step to simplify the task of the parser.

Parser helps organize the input tokens into an AST. We build up an AST for each query using Packrat Parser which is available in Scala Parser Combinators. In short, Packrat Parsing is a technique for implementing backtracking, recursive-descent parsers, with the advantage that it guarantees unlimited lookahead and a *linear* parse time. Using this technique, left recursive grammars can also be accepted.

To produce a complete meaningful AST using Scala parser combinators, we goes through a procedure of 4 substeps:

- **Step 1:** Executing the grammar

  Based on the EBNF grammars of FlinkCQL which are described mostly in session 4.2, we write equivalent executable grammars in Scala syntax with helps of combinators:

  - $p_1 \sim p_2$: sequencing combinator: must match p_1 followed by p_2. For example, *Projection* clause is followed by *From* clause
  - $p_1 | p_2$: alternation combinator: must match either p_1 or p_2, with preference given to p_1. For example, a window policy can be either time-based or count-based.
  - $opt(p_1)$: optionality combinator: may match p_1 or not. For example, *Where* clause is optional in *Select* statement.
  - $rep(p_1)$: repetition combinator: matches any number of repetitions of p_1. For instance: there is one or more entries in *Projection* clause.

  In our processor, we define a grammar for *Select* statement as :

  ```
  lazy val selectStmtSyntax = projectionClause ~
     fromClause ~ opt(whereClause) ~ opt(groupByClause)
  ```

  It means that *projectionClause* is followed by a *fromClause*, an optional *whereClause* and an optional *groupByClause* in order.

  The topmost abstraction of the FlinkCQL grammar is *Statement*. It can express either *Select*, *Create schema*, *Create stream*, *Merge*, *Insert*, *Split* query syntax.

  ```
  lazy val stmt = (selectStmtSyntax|
     createSchemaStmtSyntax | createStreamStmtSyntax |
     MergeStmtSyntax | insertStmtSyntax| splitStmtSyntax)
  ```

The parsing process of the grammar will generate a parse tree but the ouput tree is represented by a string. Let us consider the parse tree of the sample query:

```
((List(((((s~.)~Quantity),*,Price)~None),(((StockTick
   ~.)~Symbol)~None))~(((StockTick~None)~Some(s))~None)
   )~None)~None
```

The List denote that the query require 2 entries in projection clause. The last 2 *None* values express that *Where* clause and *GroupBy* clause is missing.

However, this output format is useless for next steps since it is also a string in a different way. Thus, we need to design a more powerful abstract semantic model and integrate it to the parser tree so that the parser will return a **Statement** object , instead of a string. Once we obtain a **Statement** object, we are able to exploit its components recursively for further traversal. As long as we build a complete semantic model for FlinkCQL (in Step 2), Scala offers a bunch of function application combinators to plug directly these semantic abstractions to the parse rules (in Step 3)

- **Step 2:** Building the semantic model for the FlinkCQL

As we mentioned above, the topmost abstraction of FlinkCQL is *Statement* which we can defined in Scalas as a *trait* (similar to Interface).

```
sealed trait Statement[T]
case class CreateSchema[T](...) extends Statement[T]
case class CreateStream[T](...) extends Statement[T]
case class Select[T](...) extends Statement[T]
case class Insert[T](...) extends Statement[T]
case class Merge[T](...) extends Statement[T]
case class Split[T](...) extends Statement[T]
```

Each inherited class of *Statement* contain its attributes as child nodes in the whole AST model. Each of these attribute may contains several child notes as well. For example, each *Entry* in Select's projection clause is composed of its name, alias name and an expression:

```
case class Entry[T](name: String, alias: Option[
   String], expr: Expr[T])
```

We are then able to access to this *Entry*'s expression *expr*. Hence, we are able to build up a complete semantic model for FlinkCQL using a top-down approach.

- **Step 3:** Generating the AST using function application combinators.

  Up to now, we have a set of parser rules which produce a parser tree, and also design a underlying domain-level abstractions of FlinkCQL. The last step to generate a AST is to plug suitable abstractions into grammar rules using Scala function application combinators. Consider the following examples of *Select* query:

  ```
   lazy val selectStmtSyntax =
   selectClause ~ fromClause ~ opt(whereClause) ~opt(
      groupBy) ^^ {
   case s ~ f ~ w ~ g => Select(s, f,w, g)
  }
  ```

  We do use the combinator "∧∧" to deconstruct the tuple returned by the parser and to thread it into the anonymous pattern-matching function. The last result is a Select object as above. Similar integration rules are applied for other parsing components across the process. If the whole parsing process is successful, the processor effectively built the whole semantic model as the AST used by next phrase.

  For the sample query, this step return a AST with topmost abstraction is a Select object which is visualized in Figure 5.4

  ```
   Select(List(Entry(<constant>,None,ArithExpr(ArithExpr
      (Field(Quantity,Some(s)),*,Field(Price,None)),+,
      Constant((Double,Double),10.2,double))), Entry(
      Symbol,None,Field(Symbol,Some(StockTick)))),
      ConcreteStream(Stream(StockTick,Some(s),false),
      None,None),None,None)
  ```

5.2.2 Query Rewriting

The AST, which is produced from previous phrase, is analyzed directly from FlinkCQL query string. For some reason, the query may not be optimal. Consider a very simple example, one entry in projection clause: $Quantity + Quantity$, which could be replaced by $Quantity * 2$. Such query rewriting rules optimize the AST but optional.

We do implement a AST rewriting rule that helps to reduce the numbers of translation rules in Code Generation phrase. This rewriting rule is apply to a subquery clause which produce an output *Windowed Stream*. If we able to rewrite it to a new subquery that returns

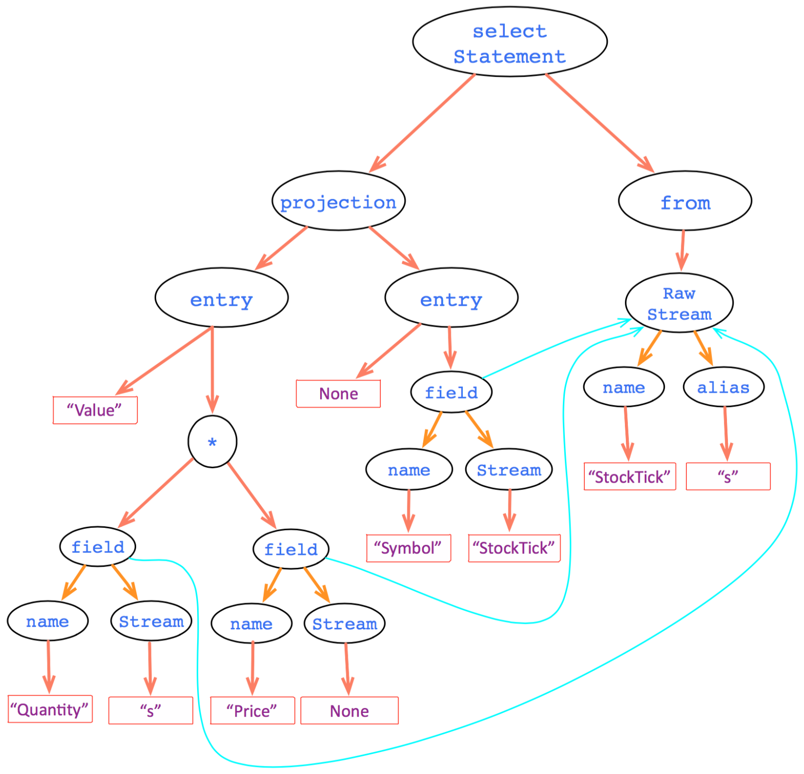


Fig. 5.4 Parse

a concrete DataStream instead, we need not to define any translation rule for Windowed Stream anymore.

Consider two queries which have identical meaning:

```
From (select Symbol,Price From StockTick [Size 3]) as s
=
From (select Symbol,Price From StockTick) [Size 3] as s
```

In the first query , its subquery will return a Windowed Stream which is not supported. Therefore the query is invalid. In the second query, its subquery return a concrete DataStream which is acceptable.

The change we made on AST structure of the first query is to move the window of the subquery to its upper-level (From clause). Thus, in the new structure (query 2), the subquery will produce a *DataStream* of *(Symbol, Price)* whereas the upper-level will process on a windowed stream with size of 3.

5.2.3 Resolving

Looked at the Figure 5.3 again, the output of last 2 phrases is a *Statement* object with type parameter *T* is *Optional[String].* It indicates that there are some potential fields/columns whose streams are not precisely specified in AST. In other words, the value of *stream* is None or a stream alias.

```
case class Field[T](name: String, stream : T)
```

In the example (Figure 5.4), there are three fields: “StockTick.Symbol”, “s.Quantity”, “Price”. Their *stream* value is “StockTick", “s" , and *None* respectively. The interpreter is not sure if those identifiers of streams do exist or not. Moreover, it is impossible to operate on a field whose stream is ambiguous. Thus, we need to figure out which streams that “Symbol”, “Quantity”, “Price” fields refer to.

This step helps to resolve the problem so that we can generate the exact Flink program in next phrase.

To solve the problem, each Statement maintains a list of its Streams so that FlinkCQL processor can look up to resolve which streams the fields refer to. Recall the example, processor acknowledges that there is one Stream exists whose name and alia are “StockTick” and “s”. Since the prefix of the fields match either those values or *None*, the stream of fields are updated to “StockTick” (Figure 5.5)

The output of this phrase is a rewritten AST as *Statement[Stream]* since stream values of fields are totally resolved.

```
Select(List(Entry(<constant>,None,ArithExpr(Field(
    Quantity,Stream(StockTick,Some(s),false)),*,Field(
    Price,Stream(StockTick,Some(s),false)))), Entry(Symbol
    ,None,Field(Symbol,Stream(StockTick,Some(s),false)))),
    ConcreteStream(Stream(StockTick,Some(s),false),None,
    None),None,None)
```

5.2.4 Code Generation

The last phrase of interpretation is to generate an executable Flink program in Scala from the output AST of previous phrase. Scala language provides compile-time and run-time reflection to build or manipulate a Scala Abstract Syntax Tree which used to represent Scala programs.

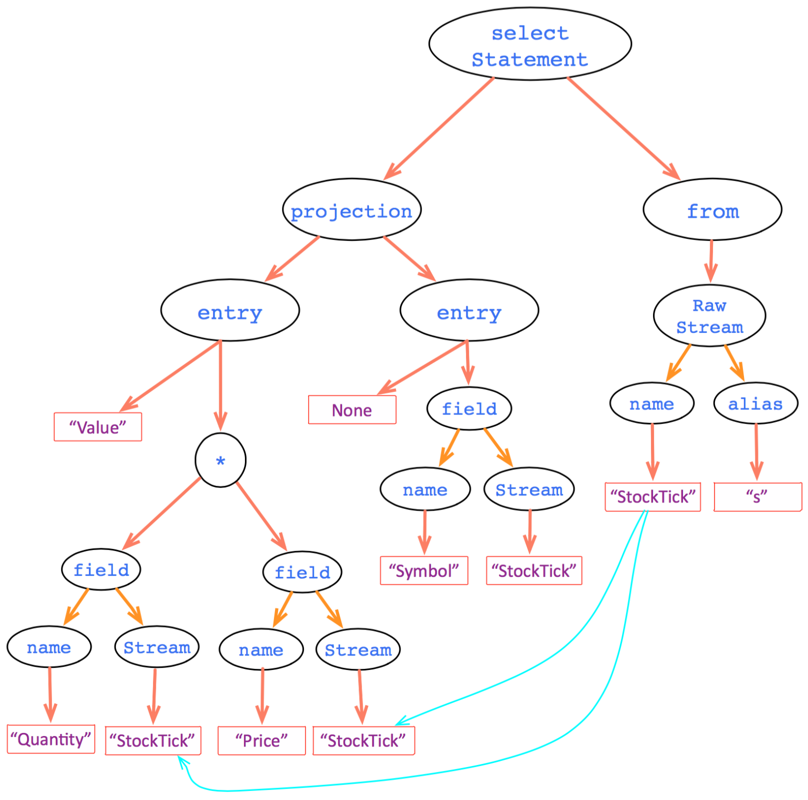


Fig. 5.5 Resolve

The compile-time reflection is realized in the form of macros, which provide the ability to execute methods that manipulate Scala AST at compile-time. However, Scala also provide ToolBox api one can generate, compile Scala AST and run Scala code at runtime.

I design a different set of translation rules to derive a Scala AST from different Statement object which serve as FlinkCQL AST. The Scala AST could be compiled to Flink executatble program in either compile-time or runtime.

Supposed that

$$\overline{X} = Generate(X) \tag{5.1}$$

with X is object abstraction denoting a subtree of a query AST and $\overline{X}$ is an equivalent block of Scala AST code generated from X.

Select statement Generation

Select object has 4 attributes: Projection *proj*, StreamReference *streamRef*, Where *w* and GroupBy *by*. *Where* and *Groupby* attribute object are optional.

Supposed that Select object s :

```
s = Select (proj , streamRef , w, by)
```

StreamReference object has up to 3 attributes : Stream *stream*, window Specification *ws* and Join *j*.

```
streamRef = StreamReference(stream , ws , j)
```

However, we can generate a StreamReference object from Join object in condition that WindowSpecification object w_1 and w_2 specify a same time-based window. Otherwise, processor is not able to generate code of Join object due to Flink's functionality constraint.

$$\text{j} = \text{Join}(s_2, w_2, \text{joinCondition}) \tag{5.2}$$
$$\text{jointStreamRef} = \text{StreamRef}(s_1, w_1, \text{Join}(s_2, w_2, \text{joinCondition})) \tag{5.3}$$

$$\begin{aligned} StreamReference(stream, ws) &= \overline{jointStreamRef} \\ &= \overline{s_1}.\text{join}(\overline{s_2}).\text{onWindow}(\overline{s_2}).\text{where}(\overline{joinCondition}) \end{aligned} \tag{5.4}$$

in which, “join”, “onWindow”, “where” are Flink's functions applied to *DataStream[Row]* object generated from s_1

Therefore, we could assume that

```
streamRef = StreamReference(stream , ws)
```

There are 2 possible scenarios for optional WindowSpecification object *ws*:

1. ***ws* is undefined**. It means that processor does query on a concrete *DataStream[Row]*. The generated block of Flink code for Select object *s* is defined as

$$\overline{s} = \overline{stream}.filter(\overline{w}).map(\overline{proj}) \tag{5.5}$$

   Be aware that *GroupBy* operator is not applicable for a concrete Stream because it partitions the original data stream into multiple undefined streams which processor can not feed as input streams for other queries.

2. ***ws* is defined** as:

```
ws = WindowSpecification(size , every , partitioned)
```

   It means that processor does query on a *DataStream[Row]* discretized by windowing technique. The generated block of Flink code for *Select* object *s* is defined as

$$\begin{aligned}\overline{s} = \overline{stream}.filter(\overline{w}).groupBy(\overline{by}).window(\overline{size}).every(\overline{every})\\ .groupBy(\overline{partitioned}).mapWindow(\overline{proj}).flatten\end{aligned} \quad (5.6)$$

Merge statement Generation

Merge object has only one member attribute which is the list of input *Streams*

$$\text{m = Merge(List(}s_1, s_2, ...s_k\text{))} \quad (5.7)$$

The generated block of Flink code for *Merge* object *m* is:

$$\overline{m} = \overline{s_1}\text{.merge(}\overline{s_2}, ...\overline{s_k}\text{))} \quad (5.8)$$

Split statement Generation

Split object *sp* has 2 attributes: source stream *s* and list *insertList* of *Insert* objects. Process first extracts targeted Streams and split conditions in *Where* clause of *Insert* object. It then collects those information to a tuple list *tupleList* of *(Stream, Where)*. Finally, a Flink program is generated based on *tupleList* list , stream *s* and *split* function in Flink API.

$$sp = Split(s, insertList)) \quad (5.9)$$

$$tupleList : List(Stream, Where) = extract(insertList) \quad (5.10)$$

$$\overline{sp} = \overline{s}\text{.split(}\overline{tupleList}) \quad (5.11)$$

For other statements such as *CreateSchema*, *CreateStream* and *Insert*, processor simply creates new objects, establish mapping between Stream and Schema. At the end, all of mapping relationship will be updated to *SQLContext* for later use.

5.3 Evaluations

Experimental environment

The proof-of-concept prototype of FlinkCQL query interpreter is built and tested on a machine equipped Intel Core i7 2,6 GHz, memory of 16Gb RAM, and free space of 10Gb in harddisk, and running MacOS 10.10.3.

The source code is committed to Flink repository at Github (https://goo.gl/V6Y5dK)[1]

5.3.1 Two testing phrases

Recall that processor goes through 4 phrases to complete the interpretation process. The first 3 phrases is to build up a semantic AST from a input query string. The last phrase is to generate an executable program from the AST, *SQLContext* and Flink streaming execution environment. We recognize the first 3 phrases is nearly independent from input data streams. Therefore, we decide to evaluate the results in 2 phrases:

- **AST construction** testing phrase: to evaluate the output AST of the first 3 steps: scanning - parsing, rewriting and resolving.
- **Executable Program** testing phrase: to evaluate the complete interpretation process.

Phrase 1: AST Construction

Each Stream Processing engine may define its own continuous query language. Up to now, there is no benchmark such as *TPC-H* to evaluate and measure the correctness and performance of query-based stream processing engines in general. Hence, we decide to test the AST construction on our own set of queries (Appendix A). Queries are broken into 6 groups corresponding to its types:

- Group 1: Create Schema queries
- Group 2: Create Stream queries
- Group 3: Select queries
- Group 4: Merge queries
- Group 5: Insert queries
- Group 6: Split queries

Figure 5.6 depicts the *maximum* runtime (in milliseconds) of each phrases in each group. We observe that Select and Split query usually take longer time to process since they have most complicated syntax. However, the maximum of total process is less than **70 milliseconds** only. In additional, a large portion of time is to parsing the input query string to the first AST.

[1]https://github.com/mbalassi/flink/tree/linq/flink-staging/flink-streaming/flink-streaming-DSL

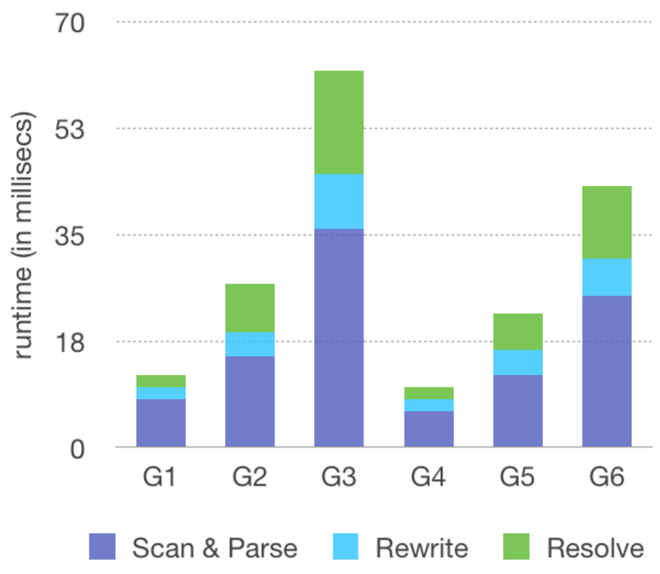


Fig. 5.6 AST Construction Testing

Phrase 2: Executable Program

We finally implement a simple but complete application consisting of 4 queries in order:

Table 5.2 Code Generation Test

No	Query
1	create schema carSchema (carID Int, price Int)
2	create stream CarStream carSchema source stream ('CarSales')
3	select carID, c.price * 0.05 as tax from CarStream as c
4	select c.pr * 1.05 as fullTax from (select pr from (select carID , price as pr from CarStream) as d) as c

We are able to register a *CarStream* stream which is derived from user-defined *DataStream[Car]* named “CarSales”

```
case class Car (carID: Int, price: Int)
```

The third query is to observe the tax which owners need to pay for their cars. The last query is just to demonstrate that we are able to write n-nested subquery with FlinkCQL. Even with this complex query, the whole interpretation process takes less than 60 milliseconds to generate an executable program since the query was received. Consider that the stream is possibly unbound, the latency of 60 milliseconds to translate the query is not considerable.

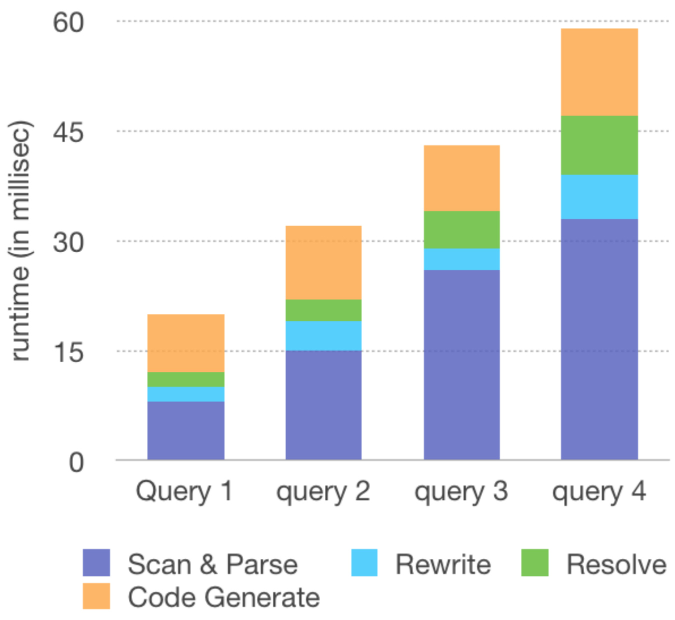


Fig. 5.7 Executable Program Testing

5.4 Future Work

Though we ran experiments successfully on our set of queries, the number of queries is till very restricted and objective. We are not able to test all possible queries with limited user-generated queries. Our future work is to build a Random Query Generator (RQG) for coverage and functional testing. RQG generates thousands of random queries based on SQL templates the queries must conform to. The idea is that we build a Finite State Machine model from the templates and decide how to walk around, expand an AST and generate numerous valid queries probabilistically.

For the last phrases of Code Generation, we have employed Scala *Macros* technique to gradually construct the Scala AST code which is executed at *compile-time*. However, there is a disadvantage when applying Macro to generate Flink program. Some Flink functions required informations extracted from SQLContext at runtime. For example, “mapWindow” function is applied on a *WindowedStream* to transforms a window to other type of window. We able to use “mapWindow” to reduce the window to a tuple (using aggregation function) which may fulfill the projection clause of *Select* query. However, the parameter of this function is a anonymous function which required to specify its input type and output type. Although the output type could be deduced by AST and Schema in SQLContext, we are unable to extract a specific SQLContext information at compile-time. Thus, we could not generate code for “mapWindow” parameters. One of potential solution is to generate and execute code at *runtime* using *ToolBox Api*. One can deduce the type of the output, plug it into Scala AST of mentioned anonymous function and generate this function at *runtime*. "mapWindow" may work in this case.

In conclusion, we may use runtime compilation approach to generate the Scala AST. In addition, a Random Query Generator is needed in the purpose of testing.

Chapter 6

Conclusion

The work described on the thesis has been concerned with designing a complete language syntax and implementing its interpreter for FlinkCQL - a continuous query language serving as an application layer of Flink Stream Processing Engine.

The first of its contributions is to study the execution model of Flink Stream Processing engine and to describe its semantics formally. Based on its tuple-driven model, we propose an extended definition of Data Stream in Flink whose elements logically consist of tuple value , optional application timestamp and system timestamp.

By understanding its execution model and available features, the complete set of FlinkCQL syntax is designed based on standard native SQL. Users are able to send their essential commands to Flink engine through 6 types of queries: create schema, create stream, select, insert, merge and split. Besides complex subquery syntax, FlinkCQL also supports window operators to observe and investigate the interested recent data.

The thesis also proposes and implements an architecture of FlinkCQL query processing. The heart of the processor is a tree-based query interpreter which accepts a query string and source Data streams as inputs. The input data streams are processed and transformed based on a chain of operators which is specified in query string. The experiments show that the proof-of-concept prototype of FlinkCQL interpreter is able to completely translate a common query string in *less than 60 milliseconds* which is really efficient in the sense that the Flink program keeps querying and producing outputs as long as the source data streams are till emitting data.

References

[1] (2010). Apache storm. https://storm.apache.org Accessed May 1, 2015.

[2] (2012). Apache flink. https://flink.apache.org Accessed May 1, 2015.

[3] (2012). Apache spark. https://spark.apache.org Accessed May 1, 2015.

[4] (2013). Sap event stream processor. http://www.sap.com/pc/tech/database/software/sybase-complex-event-processing/index.html Accessed May 1, 2015.

[5] (2013). Streamsql tutorial. http://www.streambase.com/developers/docs/latest/streamsql/usingstreamsql.html Accessed May 10, 2015.

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

[7] 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.

[8] Arasu, A. and Widom, J. (2004). A denotational semantics for continuous queries over streams and relations. *SIGMOD Rec.*, 33(3):6–11.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] Gehani, N. (2003). Bell labs: life in the crown jewel. *Professional Communication, IEEE Transactions on*, page 78.

[14] 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.

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

[16] 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.

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

[18] 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.

[19] Law, Y.-N., Wang, H., and Zaniolo, C. (2011). Relational languages and data models for continuous queries on sequences and data streams. *ACM Trans. Database Syst.*, 36(2):8:1–8:32.

[20] Luckham, D. C. (2001). *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

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

[22] Parr, T. (2009). *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Pragmatic Bookshelf, 1st edition.

[23] Patroumpas, K. and Sellis, T. (2006). Window specification over data streams. In *Proceedings of the 2006 International Conference on Current Trends in Database Technology*, EDBT'06, pages 445–464, Berlin, Heidelberg. Springer-Verlag.

[24] 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.

[25] 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.

[26] Shahrivari, S. (2014). Beyond batch processing: Towards real-time and streaming big data. *CoRR*, abs/1403.3375.

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

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

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

[30] 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.

[31] Vincent, P. (2014). Cep tooling market survey 2014. http://www.complexevents.com/2014/12/03/cep-tooling-market-survey-2014/ Accessed May 1, 2015.

[32] Zaharia, M., Das, T., Li, H., Shenker, S., and Stoica, I. (2012). Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 10–10, Berkeley, CA, USA. USENIX Association.

Appendix A

Query Interpreter Testing

Table A.1 Group 01: Create Schema

No	Query
1	create schema mySchema0 (speed int, time long)
2	create schema mySchema1 mySchema0
3	create schema mySchema2 (id int) extends mySchema0
4	CREATE SCHEMA StockTickSchema (symbol String, sourceTimestamp Long, price Double, quantity Int, exchange String)
5	CREATE SCHEMA StockTickSchema2 StockTickSchema
6	CREATE SCHEMA StockTickSchema3 (id Int) EXTENDS StockTickSchema

Table A.2 Group 02: Create Stream

No	Query
1	create stream CarStream (speed int) source stream ('cars')
2	create stream CarStream carSchema source stream ('cars')
3	create stream myStream(time long) as select p.id from oldStream as p
4	create stream myStream(time long) as merge x1, x2, x3
5	CREATE STREAM StockTick StockTickSchema
6	CREATE STREAM StockTick (symbol String, price Double, quantity Int)
7	CREATE STREAM StockTick StockTickSchema source SOCKET ('98.138.253.109', 2000,';')
8	CREATE STREAM StockTick StockTickSchema source Stream ('stockDataStream')
9	CREATE STREAM StockTick (symbol String, price Double, quantity Int) source file ('//server/file.txt')
10	CREATE STREAM StockPrice (symbol String, price Double) AS SELECT symbol, price FROM StockTick

Table A.3 Group 03: Select

No	Query
1	select s.Quantity * Price + 10.2, StockTick.Symbol from StockTick as s
2	select s.Quantity * Price + 10.2, s.Symbol from StockTick as s where s.price > 90.5 and Quantity <= 100000
3	select c.price * 1000 from (select plate , price from CarStream) as c
4	select max(s1.price), avg(s2.quantity) from stream1 [size 3 sec] as s1 cross join stream2 [size 3 sec] as s2
5	select time, min(price)+1 from stream1 [size 3] as s1 left join stream2 [size 3] as s2 on s1.time=s2.time
6	select id from (select p.id as id from oldStream2 as p) [size 3 partitioned on s] as q
7	select from stream1 [size 3 on s] as s1 left join stream2 [size 3 on s] as s2 on s1.time=s2.thoigian
8	select id from stream1 [size 3 min] as s1 left join stream2 [size 3 min] as s2 using time
9	Select Count(*) From Bid[Size 1 partitioned on field1] Where item_id <= 200 or item_id >= 100 group by item_id
10	select count(price) from (select plate , price from CarStream [Size 1]) as c
11	select count(price) from (select plate , price from CarStream) [Size 1] as c
12	select * from (select plate , price from (select plate , price from (select plate , price from CarStream [Size 1]) as e) as d) as c
13	select c.plate + 1000/2.0 from (select plate as pr from (select plate , price + 1 as pr from CarStream) as d) as c
14	Select count(*), max(item_id) From (Select * From Bid Where item_id >= 100 and item_id <= 200) [Size 10] p

Table A.4 Group 04: Merge

No	Query
1	MERGE stockTickFromNYSE, stockTickFromAMEX, stockTickFromNASDAQ
2	merge x1, x2, x3

Table A.5 Group 05: Insert

No	Query
1	INSERT INTO StockTick AS MERGE stockTickFromNYSE, stockTickFromAMEX
2	INSERT INTO StockTick AS SELECT symbol, price FROM StockTick where quantity > 100000

Table A.6 Group 06: Split

No	Query
1	on x insert into x1 as select f1 from x where f1 > 3 or f1 < 1 ; insert into x1 as select f1 from x where f1 <=3 and f >=1