Tutorial: An Outlook to Declarative Languages for Big Streaming Data

Riccardo Tommasini
Politecnico di Milano
riccardo.tommasini@polimi.it

Sherif Sakr
Unversity of Tartu
sherif.sakr@ut.ee

Marco Balduini
Politecnico di Milano
marco.balduini@polimi.it

Emanuele Della Valle
Politecnico di Milano
emanuele.dellavalle@polimi.it

ABSTRACT

In the Big Data context, data streaming systems have been introduced to tame velocity and enable *reactive* decision making. However, approaching such systems is still too complex due to the paradigm shift they require, i.e., moving from scalable batch processing to continuous analysis and detection. Initially, modern big stream processing systems (e.g., `Flink`, `Spark`, `Storm`) have been lacking the support of declarative languages to express the streaming-based data processing tasks and have been mainly relying on providing low-level APIs for the end-users to implement their tasks. However, recently, this fact has been changing and most of them started to provide SQL-like languages for their end-users.

In general, declarative Languages are playing a crucial role in fostering the adoption of Stream Processing. This tutorial focuses on introducing various approaches for declarative querying of the state-of-the-art big data streaming frameworks. In addition, we provide guidelines and practical examples on developing and deploying Stream Processing applications using a variety of SQL-like languages, such as `Flink-SQL`, `KSQL` and `Spark Streaming SQL`.

CCS CONCEPTS

• **Information systems → Data management systems**.

KEYWORDS

stream processing, complex event processing, streaming sql

ACM Reference Format:
Riccardo Tommasini, Sherif Sakr, Marco Balduini, and Emanuele Della Valle. 2019. Tutorial: An Outlook to Declarative Languages for Big Streaming Data. In *DEBS '19: The 13th ACM International Conference on Distributed and Event-based Systems (DEBS '19), June 24–28, 2019, Darmstadt, Germany*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3328905.3332462

1 INTRODUCTION

In every second of every day, we are generating massive amounts of data. Initially, Big data processing systems (e.g., `MapReduce` [8]) focused on scalable batch processing for static data. Recently, the challenges related to Big Data processing become were clarified. It becomes clear the need of taming data processing along multiple dimensions also known as the four Vs, i.e., *Volume*, *Variety*, *Veracity*, and with the growing interest of having (near) real-time processing, *Velocity* [12].

In general, stream and event processing techniques were around for decades [11, 14]. Nevertheless, in the the Big Data context Data Velocity cannot be entirely separated from the other dimensions. Therefore, solutions like `Storm`, `Flink`, `Spark Structured Streaming`, and `Kafka Streams` emerged among the others to tame data Velocity. In practice, a lot of work was done on handling the complexity of writing applications by designing systems' APIs that allow systems' users to implement a large class of applications. Users' requirements comprise consuming data generated from multiple sources; pushing data asynchronously to servers responsible to transform, manipulate, and enrich these data joining them with other resources [9]. Most of big stream processing systems initially provided functional APIs to write programs. Later on, SQL-inspired languages started spreading, acknowledging the need need for declarative languages [10, 15].

The goal of this tutorial is to provide an overview of the state-of-the-art of the ongoing research and development efforts in the domain of declarative languages for big streaming data. In particular, the focus of the tutorial are on the following aspects:

- To provide an overview of the fundamental notions for processing streams with declarative languages;
- To present an outlook and comparison of prominent Big Streaming Data frameworks that offer declarative domain-specific languages;
- To provide a critical discussion on challenges and opportunities for declarative streaming query languages.

In the following sections, we will present examples comprising three prominent Big Streaming Data frameworks that started to develop their declarative SQL-like domain-specific languages to tame data Velocity, i.e., `Flink` [5] with `Flink SQL`[1], `Spark Structed Streaming` [3] with Spark SQL[2], and `Kafka Streams`[3] with `KSQL`[4]. In particular, we will present relevant examples for each of the selected systems and languages, highlighting similarities and differences alongside the following fundamental features for stream

[1]https://ci.apache.org/projects/flink/flink-docs-stable/dev/table/sql.html
[2]https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html
[3]https://kafka.apache.org/documentation/streams/
[4]https://www.confluent.io/product/ksql/

processing languages: (i) Windowing and Aggregates (ii) Stream Enrichment (Stream-Table Joins) (iii) Stream-to-Stream Joins

2 INTENDED ANALYSIS PRINCIPLES

In general, the main aim of declarative languages (DL) is to facilitate the development of applications by achieving data independence [6]. In practice, DLs become particularly useful in those contexts where the complexity of writing a program is as high as the problem the program is made to solve. In fact, DLs simplify common coding tasks making code more readable and software maintainable. Moreover, DLs decouple program design and optimization.

Figure 1 illustrates the general steps for processing declarative queries. DLs typically distinguish the following levels of abstractions, i.e., the *syntactic* level, which correspond the to the set of programs a user can write; the *logical* level, which corresponds to the set of valid programs for which it is possible to provide a logical plan of execution; and the *operational* level, which corresponds to the set of programs that can be actually executed. These levels map respectively to different states in the program life-cycle, i.e., parsing, planning, and evaluating.

In practice, declarative languages allow to treat usability, optimization and, thus, efficiency in a separate manner. Nevertheless, it is possible to catch up with behavioral inconsistencies across systems implementing the same language when they are not backed-up with a solid theoretical background [7]. Language design principles like Codd's ones also have played a critical role on making languages portable. In particular, the main design principles of declarative languages include: *Minimality*, i.e., a language should provide only a small set of needed language constructs so that the same meaning cannot be expressed by different language constructs; *Symmetry*, i.e., a language should ensure that the same language construct always expresses the same semantics regardless of the context it is used in; and *Orthogonality*, i.e., a language should guarantee that every meaningful combination its constructs is applicable.

So far, in the context of Big Data Streams, declarative languages suffer from the absence of a shared formal framework that clarify execution models and time-management approaches. Therefore, existing systems have developed their declarative languages with a main focus on meeting specific industrial and application needs. In particular, they try to be as syntactically-close as possible to SQL consequently neglecting Codd's criteria for language design. Nevertheless, SQL was not intended to be used with unbounded streams of data nor with the continuous semantics required to process them. Formal models to extend traditional relational algebra into a backwards-compatible streaming algebra exist, e.g., CQL [1], but they were only partially considered [13]. In fact, existing systems privilege APIs that only resemble the SQL syntax but often lack a clear separation of concerns like Arasu et al.'s [2].

3 STREAMING LANGUAGES: EXAMPLES

In this section, we provide details about the declarative domain-specific languages provided by prominent Big Data framework for writing programs that consume and produce streams. Our selection include Apache Kafka's KSQL[5], Flink's SQL [5], and Spark Structured Streaming [3]. These frameworks have different goals and design principles, but they all offer declarative solution to process streaming data.

KSQL is a recent effort by the Kafka Stream community that aims at reducing the offer of writing Stream Processing applications[3,4].
Flink SQL[1,4] makes use of Apache Calcite [4] for parsing and planning. Flink SQL offers a unified solution for static and stream processing that, following Calcite intent, tries to be as close as possible to SQL standard syntax.
Spark Structured Streaming [2,4]is a library for Apache Spark that enables stream processing on top of Spark's DataFrame APIs.

In the following, we present canonical examples of queries, written using the DSLs of the aforementioned systems. We kept the examples as simple as possible, in order to focus on specific features of the languages.

3.1 Time Window Operators & Aggregates

```
SELECT nation , count(*) FROM pageviews GROUP BY nation
```

Listing 1: Generic Aggregation implicitly referring to a landmark window opened when the query was started.

Aggregations over time windows is one of the most demanded use case in stream processing. Let's consider, for instance, a Web Analytics scenario where Web servers stream each visited page (e.g., DEBS2019 homepage) together with information about the visitors such as the nation of their IP addresses (e.g., Italy and Estonia). In this context, it is useful to count the number of users per nation: 1) since the opening of the website (i.e., using a *landmark* window), 2) in the last hour (i.e., using a *tumbling* window), 3) in the last hour updating the result every minute (i.e., using a *hopping* window) or for every new visit (i.e., using a *sliding* window), or 4) within a *session* window defined as the time span in which visits are dense (e.g., two consecutive visits are separated by less then a minute).

All the considered languages allow (implicitly) landmark windows and continuously update the aggregate. All basic SQL aggregates are supported.

Hopping windows (as well as the special case of the tumbling window in which the overlap between consecutive windows is zero) are available in all languages.

```
CREATE TABLE analysis AS SELECT nation , COUNT(*)
FROM pageviews
WINDOW HOPPING (SIZE 1 HOUR, ADVANCE BY 1 MINUTE)
GROUP BY nation;
```

Listing 2: KSQL Hopping Window. Note the presence of the DDL statement CREATE TABLE ... AS as well as the window clause in the FROM.

```
SELECT nation , COUNT(*)
FROM pageviews
GROUP BY HOP(rowtime ,INTERVAL 1H,INTERVAL 1M) , nation
```

Listing 3: Flink SQL Hopping Window. Note the window specified as a GROUP BY criteria.

[5]https://www.confluent.io/product/ksql/

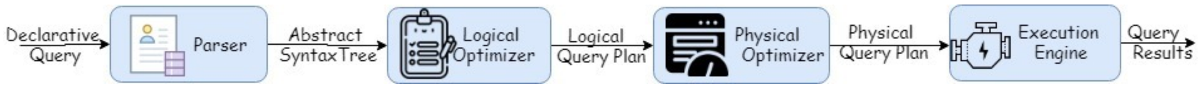

Figure 1: General Processing Steps of Declarative Queries

```
val df = pageviews.groupBy(
  window($"timestamp", "1 hour", "1 minute"),$"nation"
).count()
```

Listing 4: Spark Hopping Windowing. Note the window specified as a GROUP BY criteria.

Notably, the window clause is specified in the FROM in KSQL while it is a group by criteria in Flink and Spark. KSQL follows the CQL style, while Flink and Spark adhere to SQL.

Tumbling windows are specified with the clause `WINDOW TUMBLING` in KSQL and `TUMBLE(...)` in Flink, while Spark simply uses the same method `window(...)` without specifing the hopping interval.

Sliding windows (i.e., those that update the result as soon as a new data arrive) as well as session windows are available in KSQL and FLink but not in Spark. This is due to the nature of discretized streams in Spark and its micro batching processing model. The next two listing present the case of session windows in KSQL and Flink. The case of sliding window is not illustrated.

```
CREATE TABLE analysis AS
SELECT nation, count (*),
 TIMESTAMPTOSTRING(windowstart(), 'yyyy-MM-dd HH:mm:ss')
     AS window_start_ts,
 TIMESTAMPTOSTRING(windowend(),  'yyyy-MM-dd HH:mm:ss')
     AS window_end_ts
FROM pageviews WINDOW SESSION (1 MINUTE)
GROUP BY nation;
```

Listing 5: KSQL Session Windows.

```
SELECT nation, count(*), SESSION_START(...), SESSION
     _ROWTIME(...)
FROM pageviews
GROUP BY SESSION(rowtime, INTERVAL 1M), nation
```

Listing 6: Flink SQL Session Window.

Note that both languages project the opening and closing time of the window since they are not known a-priori as for all the other windows illustrated in this session; they depend on the data. In the example, a window clause as soon as the temporal distance between two consecutive visits is longer than a minute.

3.2 Stream-Table Joins

In Industry 4.0, it is common to instrument production lines with IoT sensors whose readings refer to the items passing the station at the time the reading is taken. While this independent deployment minimizes the initial costs, it poses the challenge to join sensor reading and items at analysis time.

	Landmark	Tumble	Hop	Session	Aggregates
KSQL	implicit	✓	✓	✓	Standard SQL
Flink SQL	implicit	✓	✓	✓	Standard SQL
SSS	implicit	✓	X	X	Standard SQL

Table 1: Time-Based Window Operators and aggregates across different systems.

	Inner	Left Outer	Right Outer	Full Outer
KSQL	S	S	NS	NS
Flink SQL	S	S*	S*	S*
SSS	S, Stateless	S, Stateless	NS	NS

Table 2: Stream-Static Joins. [S]upported, [N]ot[S]upported. S*, Flink memory usage might grow indefinitely, Temporal Tables can be used to avoid it.

Let's, for instance, assume that the sensor readings flow on a stream named `SENSOR_READINGS` and that their schema is `LINE_ID`, `SENSOR_ID`, and `READING_VALUE`. The items in production are recorded in a table named `ITEMS_IN_PRODUCTION` whose schema is `ITEM_ID`, `LINE_ID`, and many other attributes that we can ignore. Notably, in this simplified example we assume that there is only one product in production per line, but some line may have no products. We want to preserve also readings that do not match a product. Therefore we need to perform a `LEFT JOIN`. The following listings presents the queries for this example for the three languages.

```
CREATE STREAM SENSOR_ENRICHED AS
SELECT S.SENSOR_ID, S.READING_VALUE, I.ITEM_ID
FROM SENSOR_READINGS S LEFT JOIN ITEMS_IN_PRODUCTION I
    ON S.LINE_ID=I.LINE_ID;
```

Listing 7: LEFT JOIN of a stream with a table in KSQL.

```
SELECT S.SENSOR_ID, S.READING_VALUE, I.ITEM_ID
FROM SENSOR_READINGS S LEFT JOIN ITEMS_IN_PRODUCTION I
    ON S.LINE_ID=I.LINE_ID;
```

Listing 8: LEFT JOIN of a stream with a table in Flink.

```
val itemsInProduction = spark.read. ...
val sensorReadings = spark.readStream. ...
val enrichedSensorReadings = sensorReadings.join(
    itemsInProduction, "LINE_ID", "left-join")
```

Listing 9: LEFT JOIN of a stream with a table in Spark.

Notably, all languages chose a SQL like syntax, where they hide the different processing semantics. Indeed, each time a reading appears in the stream this is instantaneously joined with the table. Therefore, there is an implicit count-based window of one element opened on the stream.

While the choice may appear to simplify the life of the user for basic use case (such as this one), the user may be surprised to know that not all the join forms are supported or that in spark some output modes are not compatible with this join between streams and tables. Such limitation would be clear if the window would have been explicit. For instance, it is clear that a RIGHT join between a stream and a table is meaningless because any data may possibly appear on a stream, but we cannot wait until it appears to emit the result. No operations that can block indefinitely the processing are admitted in the streaming computational model.

3.3 Stream-to-Stream Join

The last use case that we want to highlight in this tutorial is the stream-to-stream join. Consider for example a case of Web Analytics where we need to correlate advertisements' impressions and clicks over time. Let's imagine that we can listen to an impression stream that tells when an advertisement is displayed and to a click stream that tells when an advertisement is clicked. Indeed a user can stay for a long time on a page, so an advertisement displayed now can be clicked after an hour. As in the case of stream-to-table join, all three languages decided to hide the window that CQL would have made explicit, but they cannot do so completely since they need to specify a period of time after which elements can be evicted from the stream. KSQL does it using the clause `WITHIN` in the FROM. Flink and Spark simply assume that the timestamps of the elements in the stream are accessible and they can be used in specifying the logic of the join. Spark also relies on the notion of a Watermark to determine the maximum delay allowed for late arrivals.

```
CREATE STREAM IMPRESSION_CLICKS_JOIN AS  \
SELECT * FROM IMPRESSIONS JOIN CLICKS
WITHIN 60 SECONDS ON (IMPRESSION_TIME=CLICK_TIME);
```

Listing 10: KSQL Stream-Stream Join

KSQL can perform any kind of join (even full-outer ones) for windows stream-to-stream joins. Differently from the stream-to-table join this is possible because there is a maximum time to wait before becoming sure of the presence of a given element.

```
SELECT * FROM IMPRESSIONS, CLICKS
WHERE IMPRESSION_ID = CLICK_ID AND
CLICK_TIME BETWEEN IMPRESSION_TIME - INTERVAL '1' HOUR
    AND IMPRESSION_TIME
```

Listing 11: Flink SQL Stream-Stream time-windowed Join

Flink natively treats streams as infinite tables. Therefore, aforementioned stream-table joins are possible for stream-to-stream too. However, this implies infinitely growing resource usage as the state must grows as the stream progresses. Temporal Table (TT) are intended to solve. A TT is a parameterized view that represents the changelog of an append-only table. A Temporal table function provides access to the state of a TT at a specific point in time.

```
val impressions = spark.readStream. ...
val clicks = spark.readStream. ...
// Apply watermarks on event-time columns
val imprWithWtmrk = impressions.withWatermark("
    impressionTime", "2 hours")
val clicksWithWatermark = clicks.withWatermark("clickTime
    ", "3 hours")
imprWithWtmrk.join(
  clicksWithWatermark,
  expr("""  clickAdId = impressionAdId AND
            clickTime >= impressionTime AND
            clickTime <= impressionTime + interval 1 hour
    """))
```

Listing 12: Spark Stream-Stream Inner-Joins

Spark addresses such an unbounded state problem by defining additional join conditions such that indefinitely old inputs cannot match with future inputs and therefore can be cleared from the state. It does so by defining: 1) watermark delays on both input streams such that the engine knows how delayed the input can be; and, 2) a constraint on event-time across the two input streams such that the engine can figure out when old rows of one input is not going to be required (i.e. will not satisfy the time constraint) for matches with the other input.

	Inner	Left Outer	Right Outer	Full Outer
KSQL	S, win	S, win	S, win	S, win
Flink SQL	S	S*	S*	S*
SSS	S	S + w on left	S + w on right	NS

Table 3: Stream-Stream Joins. [S]upported, [N]ot[S]upported, [W]atermark, [Win]dow. S*, Flink memory usage might grow indefinitely.

4 CONCLUSION

This paper provides an overview of modern declarative stream processing languages. As all of these languages have been just recently introduced, it is most likely that more features and capabilities for these languages will continue to be introduced and (re)-defined. That said, this paper aims of spreading the knowledge of the state-of-the-art of the available choices and the current capabilities of these languages. Of course, from the perspective of researchers, several research challenges are still open. For instance, defining a *standard* syntax and semantics for a *portable* SQL-based streaming language over the different Big Data Streaming framework is expected to gain a lot of momentum soon.

ACKNOWLEDGMENT

This work of Sherif Sakr is funded by the European Regional Development Funds via the Mobilitas Plus programme (grant MOBTT75).

REFERENCES

[1] Arvind Arasu et al. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2), 2006.
[2] Arvind Arasu et al. STREAM: the stanford data stream management system. In *Data Stream Management - Processing High-Speed Data Streams*. 2016.
[3] Michael Armbrust et al. Structured streaming: A declarative api for real-time applications in apache spark. In *SIGMOD*, 2018.
[4] Edmon Begoli et al. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *SIGMOD*, 2018.
[5] Paris Carbone et al. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.
[6] Donald D Chamberlin. Early history of sql. *IEEE Annals of the History of Computing*, 34(4):78–82, 2012.
[7] Edgar F Codd. *Relational completeness of data base sublanguages*. Citeseer, 1972.
[8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1), 2008.
[9] Martin Hirzel, Guillaume Baudart, Angela Bonifati, Emanuele Della Valle, Sherif Sakr, and Akrivi Vlachou. Stream processing languages in the big data era. *SIGMOD Record*, 47(2), 2018.
[10] Volker Markl. Breaking the chains: On declarative data analysis and data independence in the big data era. *Proceedings of the VLDB Endowment*, 7(13), 2014.
[11] Shanmugavelayutham Muthukrishnan et al. Data streams: Algorithms and applications. *Foundations and Trends® in Theoretical Computer Science*, 1(2), 2005.
[12] Sherif Sakr. *Big Data 2.0 Processing Systems - A Survey*. Springer Briefs in Computer Science. Springer, 2016.
[13] Matthias J. Sax et al. Streams and tables: Two sides of the same coin. In *BIRTE Workshop*, 2018.
[14] Michael Stonebraker and Ugur Çetintemel. Stream processing. In *Encyclopedia of Database Systems, Second Edition*. 2018.
[15] Michael Stonebraker, UÇğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *ACM Sigmod Record*, 34(4), 2005.