A Preference-based Stream Analyzer

Lena Rudenko, Markus Endres, Patrick Roocks, and Werner Kießling

University of Augsburg
86135 Augsburg, Germany
firstname.lastname@informatik.uni-augsburg.de

Abstract. Stream query processing is becoming increasingly important as more time-oriented data is produced and analyzed nowadays. Examples for stream-based applications include sensor networks and infrastructure monitoring, electronic trading on Wall Street, or social networks. In the last decade several technologies have emerged to address the challenges of processing such high-volume and real-time data streams, which do not take the form of persistent database relations, but rather arrive in continuous, rapid and time-varying data objects. In this paper we focus on the new problem of preference-based query processing to analyze data streams. Preferences allow us to filter out only relevant information of the continuous data flow. In addition we provide a database approach that stores only the most important information in the database w.r.t. the user's preference. For this we introduce novel preference-based integrity constraints to keep the data right and consistent.

Keywords: Stream processing, Preference SQL, Twitter

1 Introduction

Many modern applications such as network monitoring, financial analysis, infrastructure manufacturing, sensor networks, meteorological observations, or social networks require query processing over data streams, e.g., [1,2,3]. A stream is a continuous unbounded flow of data objects made available over time, hence, it is not feasible to locally store a stream in its entirety. Likewise, queries over streams run continuously over a period of time and incrementally return new results as new data arrive.

Such stream data can be simple numbers or complex data, depending on the kind of stream provider. Through collecting and analyzing this data new ways of doing business emerge, since having such comprehensive data allows to act faster and better on certain situations. However, several aspects of data management need to be considered in the presence of data streams, offering a new research direction for the database community, cp., [4,5]. The data model and query semantics must allow order-based and time-based operations (e.g., queries over a 60sec moving window). Due to performance and storage constraints, backtracking over a data stream is not feasible, hence online stream algorithms are restricted to making only one pass over the data.

In this paper, which is work in progress, we present an approach to analyze data streams with the support of user preferences. The goal is to find the most interesting information w.r.t. to one's preference among the incredible number of streamed objects.

Preferences help us to separate the relevant data from the pointless information and make it accessible to the user either as human readable data or by storing it in a database. Another advantage of preference queries is that they never lead to an empty result. In addition, we provide a language for connecting and querying streams in Preference SQL [6], a SQL extension which supports preference queries.

As running example we use the online social networking service Twitter as a real-world stream application. Twitter is a very fast medium of information dissemination. The amount of tweets, that are posted every day is very large. That is 500 million short messages daily according to data from the year 2015 [7]. It is an endless data flow, that includes important and interesting records, but for the most part it consists of spam and information trash. According to a study from 2009 [8] the news take only 4% of the whole data. Nevertheless, in the tweets one finds latest news about accidents, terroristic acts or natural disasters which are posted live by eyewitnesses before any journalist is in place. The same is true for various sports events, where the most important news are posted on Twitter by journalists or fans. Furthermore, Twitter provides easy access by the public Twitter API, a huge amount of diverse attributes and short 140 character messages (tweets) which can be analyzed.

The remainder of this paper is organized as follows: Section 2 highlights related work. Section 3 recapitulates essential concepts of the used preference model. In Section 4 we present the framework for analyzing data streams with the help of preferences, and in Section 5 we introduce a language for preference-based stream analysis. Finally, we conclude in Section 6.

2 Related Work

Data streams can be found in every area of life: stock market, social networks, network monitoring, or meteorological observations, just to name a few. Hence, it is not amazing that many scientists all over the world try to process and to analyze these streams to extract important information from such continuous data flows.

Babu and Widom [9] for example, focus primarily on the problem of query processing, specifically on how to define and evaluate continuous queries over data streams. They address semantic issues as well as efficiency concerns. In [10] the authors motivate the need for and research issues arising from stream data processing. The paper gives an overview on past work and projects in the area of data streams, and explores topics in query languages, requirements and challenges in stream query processing. Faria et. al [11] describe various applications of novelty detection in data streams, and Krempl et. al [12] discuss challenges for data stream mining such as: protecting data privacy, handling incomplete and delayed information, or analysis of complex data. In [13] the authors examine the characteristics of important preference queries (skyline, top-k and top-k dominating) and review algorithms proposed for the evaluation of continuous preference queries under the sliding window streaming model. However, they do not present any framework for preference based stream evaluation. An important research direction associated with stream processing is data stream clustering. This issue is discussed e.g., in [14,15,16]. Cherniack et. al [17] describe a large-scale distributed stream processing system, and discusses approaches for addressing load management, high availability,

and federated operation issues in such an environment. In [18] the authors investigate queries over continuous unbounded streams for applications like network monitoring, financial analysis, and sensor networks. For this they present the Stanford Data Stream Management System (STREAM) for rapid streams and long-running queries, where the system resources may be limited.

Since we refer to Twitter in our prototype implementation, we also present some current work on this social network. In [19] the authors describe an event notification system that monitors and delivers semantically relevant tweets if these meet the user's information needs. As an example they construct an earthquake prediction system targeting Japanese tweets. For their system they use keywords, the number of words, and the context of an event. Railean and Moraru [20] address the problem of determining the popularity of social events (concerts, festivals, sport events, conferences, etc.) based on their presence in Twitter. For this they compute an association coefficient for an event-tweet pair and use it to determine the popularity.

All previous work rely on analyzing the content of a stream or describe stream processing systems. In this paper we filter data by a preference based approach, which never leads to an empty result and only extracts the most important information w.r.t. the user's preference.

3 Preference Background

Preference queries in database systems have been in focus for some time, leading to diverse approaches, e.g., [21,22]. We follow the preference model from [22], where a preference $P = (A, <_P)$ is a strict partial order on the domain of A. Thus $<_P$ is irreflexive and transitive. The term $\mathbf{x} <_\mathbf{P} \mathbf{y}$ is interpreted as "*I like y more than x*". The *maximal objects* of a preference $P = (A, <_P)$ on an input database relation R are all tuples that are not dominated by any other tuple w.r.t. the preference. These objects are computed by the preference selection operator $\sigma[P](R)$. It finds all best matching tuples t w.r.t. the preference P, where $t.A$ is the projection to the attribute set A.

$$\sigma[P](R) := \{t \in R \mid \neg\exists t' \in R : t.A <_P t'.A\} \tag{1}$$

The retrieval of best-matching results follows a Best-Matches-Only (BMO) query model that retrieves exact matches if such objects exist and best alternatives else.

3.1 Preference Constructors

To specify a preference, the framework follows an inductive approach of base preference constructors that can be combined to complex preference constructors. Subsequently, we present some selected constructors which are often applied by users.

Preferences on single attributes are called *base preferences*. There are base preference constructors for *continuous* (*numerical*), *discrete* (*categorical*), *temporal*, and *spatial* domains. For example, the *categorical preference* POS$(A,$ POS-set$)$ expresses that a user has a set of preferred values, the POS-set. The BETWEEN$(A, [low, up])$ preference expresses the wish for a value between a *lower* and an *upper* bound. The LOWEST(A) constructor and the HIGHEST(A) constructor prefer the minimum and maximum, resp.

Note that the dominance criterion of the base preferences is based on a SCORE function $f : dom(A) \rightarrow \mathbb{R}_0^+$, where $f(v)$ describes how far the domain value v is away from the optimal value. Dominated tuples have higher function values, i.e., $\mathbf{x} <_\mathbf{P} \mathbf{y} \Longleftrightarrow f(x) > f(y)$. For more details we refer to [6].

Complex preferences determine the relative importance of preferences and combine base or again complex preference constructors. Intuitively, people speak of "this preference is more important to me than that one" or "these preferences are all equally important". In the *Pareto preference* $P := P_1 \otimes P_2 = (A_1 \times A_2, <_P)$ all preferences are of equal importance, while in a *Prioritization preference* $P := P_1 \ \& \ P_2$ the preference $P_1 = (A_1, <_{P_1})$ is more important than $P_2 = (A_2, <_{P_2})$. For a more formal definition and more detailed information we refer to [22,6].

Example 1. Consider the UEFA European Championship which takes place in France in 2016. The tournaments take place in various cities (`location`). Since many people post interesting information about the tournaments on Twitter, we want to get all tweets which are posted by an author who stays in Lyon or Marseilles. This preference can be expressed by a POS constructor: $P := \text{POS}(\text{location}, \{\text{Lyon, Marseilles}\})$. If there is no tweet posted in Lyon or Marseilles, the preference selection operator would find all tweets from other cities.

3.2 Preference SQL

The Preference SQL query language is a declarative extension of SQL by strict partial order preferences, behaving like soft constraints under the BMO query model as described above. Syntactically, Preference SQL extends the **`SELECT`** statement of SQL by an optional **`PREFERRING`** clause, cp. Figure 1.

```
SELECT      ... <projection, aggregation>
FROM        ... <table_reference>
WHERE       ... <hard_conditions>
 PREFERRING ... <soft_conditions>
 TOP        ... <number>
```

Fig. 1. Preference SQL query block.

The keywords **`SELECT`**, **`FROM`** and **`WHERE`** are treated as the standard SQL keywords. The **`PREFERRING`** clause specifies a preference which is evaluated after the **`WHERE`** condition. If **`TOP`**-k is specified, only the k best tuples according to the preference order are returned [6,23].

Example 2. The POS preference in Example 1 can be expressed in Preference SQL as follows, where `Tournament` is the database which contains information about the football competitions in France.

```
SELECT * FROM Tournament
PREFERRING location IN ('Lyon', 'Marseilles');
```

In the above example **IN** expresses a POS preference. The keyword **AND** would state a Pareto preference, and **PRIOR TO** leads to a Prioritization (not shown in this example).

The current University prototype of the Preference SQL database system adopts a (Java 1.7-based) middleware approach as depicted in Figure 2. For a seamless application integration we have extended standard JDBC technology towards a Preference SQL / JDBC package. This enables the client application to submit Preference SQL queries through familiar SQL clients. The Preference SQL middleware parses the query and performs preference query optimization. The preference selection operator implemented by several evaluation algorithms computes the final result. This approach has proven its flexibility and performance in various prototype applications conducted so far, see [6] for example.

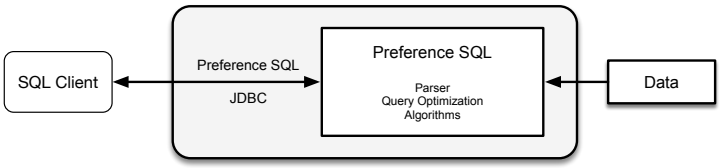

Fig. 2. System architecture of Preference SQL.

4 Analyzing Streams with Preference SQL

Our goal is to analyze data streams with Preference SQL, which needs well structured data. Hence, we need a framework that transforms the data from a stream into a processable format. For this goal we use *Apache Flink*, an open source platform for distributed stream and batch data processing[1]. The core of Apache Flink is a data flow engine created in Java and Scala. Flink includes several APIs[2], e.g., for Data-Streams, DataSets, Tables, etc. These data streams can be created from various sources (e.g., message queues, socket streams, files) and converted to other (structured) formats.

For our prototype we decided to use Twitter as an example data source, because it has an open API and free accessible data (tweets). In addition, Apache Flink provides a Twitter Streaming API connector for direct connection to this social network.

4.1 Stream Processing Framework

When processing data streams with Preference SQL, two fundamental tasks must be carried out:

1. The data must have a Preference SQL processable format, because Preference SQL works on attribute based data, but streams are often encoded in various formats, e.g., as JSON[3]-objects as in Twitter.

[1] `https://flink.apache.org/`
[2] Application Programming Interface
[3] `http://www.json.org/`

2. The result computation must be adapted to stream properties, since the dataflow is continuous. There is no "final" result after some data of the stream is processed. Hence, the result must be calculated and adjusted as soon as new data arrive.

Before we discuss these tasks in detail, we provide a brief overview of the whole system (see Figure 3). For more details we refer to the following sections. The incoming data stream is processed by Apache Flink, where our `StreamProcessor` transforms the data into a Preference SQL readable format. Our `DataAccumulator` builds chunks of objects, which can be processed by Preference SQL. Preference SQL evaluates the user preference on these data chunks and presents the result to the user. For this we suggest two possibilities: (1) we present the result to the user on his mobile phone, tablet or PC directly after the computation, or (2) we store it in a database for later usage. The first possibility is intended for users which are in front of their internet compatible device and want to find specific information rapidly. On the other hand we propose a method to store the result in a database for further processing to which we refer in Section 5.3.

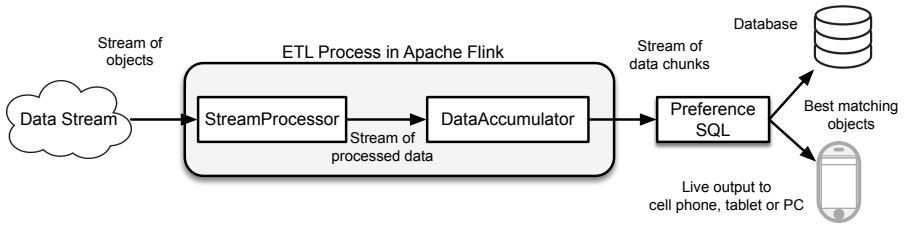

Fig. 3. ETL process in Apache Flink and analysis of data streams with Preference SQL.

4.2 Object Representation in a Preference SQL Processible Format

The individual objects delivered by a data stream are encoded and not compatible with Preference SQL, which works on attribute-based data. That means the data must be structured and needs an attribute based format like `attributeName` = *attributeValue*.

The transformation of the stream objects to a list of single attribute values occurs in the `StreamProcessor`, cp. Figure 3. For this one has to implement the mapping of the object in the stream to a table structured format. The data types of the attributes can be extracted from the stream objects, e.g., in the case of JSON-encoded tweets with the help of the Twitter API.

Example 3. The twitter messages (tweets) are JSON-encoded as shown in Figure 4. For example, this tweet was posted on June 22, 2016 at 12:49:31 UTC (***created_at***:*"Wed Jun 22 12:49:31 +0000 2016"*) and has a specific identifier (***id***:*745599599151353856*). There are several different fields which describe a tweet in detail, e.g., *name*, *description* (the user-defined string describing the account), *created_at* (date of account creation), *followers_count*, *status_count* (number of tweets of the user), *lang* (language the user speaks) and many more. Our `StreamProcessor` converts such objects into more structured data, e.g., into `created_at = ''Wed Jun 22 ...''` and defines the data type "VARCHAR".

{"created_at":"Wed Jun 22 12:49:31 +0000 2016","id":745599599151353856,"id_str":
"745599599151353856","text":"#SVK and #NIR have already qualified but who will join
them?\n\nFind out how the last 16 is shaping up: https://t.co/Wu1kXiTt7L #EURO2016",
"source":"Twitter Web Client",
"truncated":false,"in_reply_to_status_id":745581100681019392,
"in_reply_to_status_id_str":"745581100681019392","in_reply_to_user_id":1469402426,
"in_reply_to_user_id_str":"1469402426","in_reply_to_screen_name":"UEFAEURO","user":
{"id":1469402426,"id_str":"1469402426","name":"UEFA EURO 2016","screen_name":
"UEFAEURO","location":null,"url":"http://euro2016.tickets.uefa.com","description":
"The official home of #EURO2016 on Twitter.", ... }

Fig. 4. JSON object of a tweet.

After the object preparation the endless stream must be split into finite parts to be processed in Preference SQL. This is done by grouping objects into chunks. This concept is similar to moving window, described in [24]. The grouping occurs in the `DataAccumulator`, see Figure 3. It takes a stream of processed objects as input and provides a stream of chunks as output. The grouping can be *size* (how many objects are in one chunk) or *time* (the number of objects per chunk is determined by the time) based. For more details we refer to Section 5.2. The result of such a grouping is still a stream, but each chunk itself is finite and can be processed with the Preference SQL system.

In the last step we analyze the data chunks within Preference SQL. The input is a stream of chunks, the output contains the most preferred objects w.r.t. the preference specified by a user.

4.3 Finding the BMO-set of a Stream

Preference SQL computes the result based on the current chunk provided by the ETL processor (Figure 3). But this temporary BMO-set is not the final result, because when new data arrives it may occur that the next chunk contains better objects w.r.t. the user preference. Hence, the current temporary BMO-set must be compared to the objects from the next chunks in the ETL process, and so on.

More detailed: Let $c_1, c_2, ...$ be the chunks provided by the ETL processor. The Preference SQL system evaluates the user preference P on the first chunk, i.e., $\sigma[P](c_1)$. Since c_2 could contain better objects we also have to compare the new objects from c_2 with the current BMO-set, i.e., we compute $\sigma[P](\sigma[P](c_1) \cup c_2)$, and so on. However, this leads to a computational overhead if c_2 is large. Therefore we apply a pre-filter to c_2, i.e., we first compute $\sigma[P](c_2)$ and afterwards apply the preference selection to the union of $\sigma[P](c_1)$ and $\sigma[P](c_2)$, since the following holds [25]:

$$\sigma[P](c_1 \cup c_2) = \sigma[P](\sigma[P](c_1) \cup \sigma[P](c_2)) \tag{2}$$

This leads to a correct result, but is more efficient than the first approach.

5 A Language for Preference-based Stream Analysis

In this section we suggest a language for preference-based stream analysis. For this we first show how to connect streams within Preference SQL and afterwards we present a language for analyzing streams based on preference constructors. In addition, we present preference integrity constraints for table statements.

5.1 Connecting Streams in Preference SQL

We want do develop an approach, that allows to analyze any data stream with Preference SQL. For this, we have to connect the data stream to the Preference SQL system, in particular to the Apache Flink engine as depicted in Figure 3. Note that there are also other approaches to stream analysis and querying, e.g., [26,27,28], each of them has SQL-like syntax and enhanced support for windows and ordering, but they cannot be used together with preferences.

Since we want to connect different streams, we present a general approach to create a stream connection within the Preference SQL system. For this we provide an interface which must be implemented for each individual stream. In our current University prototype we provide a TwitterStream connector and we plan to implement connections for stock market and other well-known stream provider. After implementing the stream connector, a user can connect to a stream by a **`CREATE STREAM`** statement. Figure 5 presents our syntax of our proposed language for stream creation.

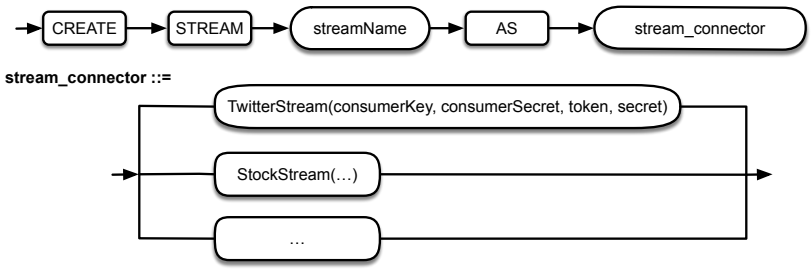

Fig. 5. Syntax diagram for create stream statement.

The keywords **`CREATE STREAM`** introduce a stream connection with the user defined name `streamName`. `TwitterStream` corresponds to the name of the pre-imple-mented Twitter connector which gets the user-specific login credentials.

Example 4. Figure 6 shows an example for the connection statement for Twitter. The cryptic character sequences represent the Twitter account login data.

```
CREATE STREAM MyTwitterStream AS TwitterStream
('NQEk0KbszVbaAcjCWLksbodkN',
 'HDKxlp2REOHvuq59oKrZZsdFovItwG6upOGJuSN4btr6npp2c3',
 '2400192752-DQSedtepr68SerQVyjHLzpHhMitcwJQfbvwxnLi',
 'BAk0krCYq77W4p45UwyuAuNnpR3nrv9WofO9PNL46YFch');
```

Fig. 6. Stream connection for Twitter.

5.2 Querying Streams in Preference SQL

After the connection to a stream the data can be queried with Preference SQL. For this we provide a preference-based stream language, whose syntax is shown in Figure 7.

After **`SELECT STREAM`** one can specify a list of columns `column_list`, where each column corresponds to a field in a stream object. *streamName* corresponds to the user defined name given in the **`CREATE STREAM`** statement (cp. Section 5.1). The kind and the quantity of the chunks, that are build from the endless stream (cp. Section 4.2), can be defined by the user. Thereby, the chunks can be size or time based. When specifying a time (in seconds), the DataAccumulator in the ETL process (cp. Figure 3) builds chunks based on the data retrieved within the given time slot. By specifying a size the ETL processor groups *size* chunks together. **`PREFERRING`** introduces the preference condition which will be evaluated on the data stream.

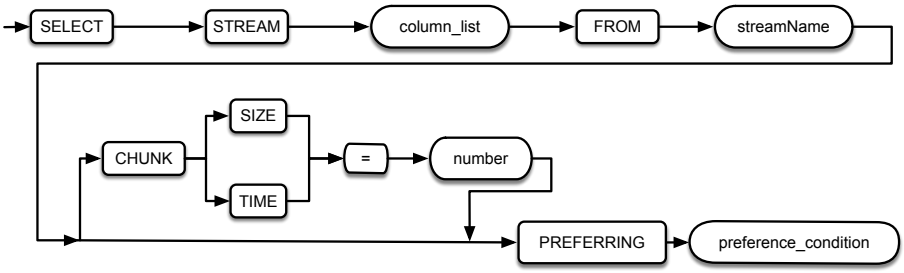

Fig. 7. Syntax diagram for select statement.

Example 5. Let us imagine that a user wants to read the tweets about the semifinals in the European Football Championship. The desired hashtag is *#EURO2016* or *#em2016* and the location should be the same as in Example 2. In addition, equally important (Pareto preference) to the previous preferences is the author of a tweet, in this example *UEFA Euro 2016* or *EM 2016 Frankreich*. The chunks should be constructed time based, here 60 seconds. The corresponding preference stream query is shown in Figure 8.

```
SELECT STREAM *
FROM MyTwitterStream
CHUNK TIME = 60
PREFERRING hashtag  IN ('#EURO2016', '#em2016')
       AND location IN ('Lyon','Marseilles')
       AND author   IN ('UEFA␣EURO␣2016','EM␣2016␣Frankreich');
```

Fig. 8. Querying Twitter data with preferences.

5.3 Preference Based Integrity Constraints

In this section we suggest an extension of SQL constraints [29] to specify preference conditions within a table definition. SQL constraints are used to specify rules for the data in a table which now should be preference based. If there is any violation between the preference constraint and the data action, the action is aborted by the constraint.

In this sense, the idea is to define preference based integrity constraints which apply whenever a tuple should be inserted into the table. If the new tuple passes the preference condition it will be stored in the table. Otherwise, if it is dominated by any other tuple already in the table, the new tuple can be discarded or moved to another table which holds dominated objects. Note that when considering streams, the current content of a database table does not necessarily represent the perfect matches, because they can be dominated by better candidates arriving later in the stream. Therefore it is important to specify what happens with these dominated tuples. Figure 9 shows the syntax diagram which describes our preference integrity constraints.

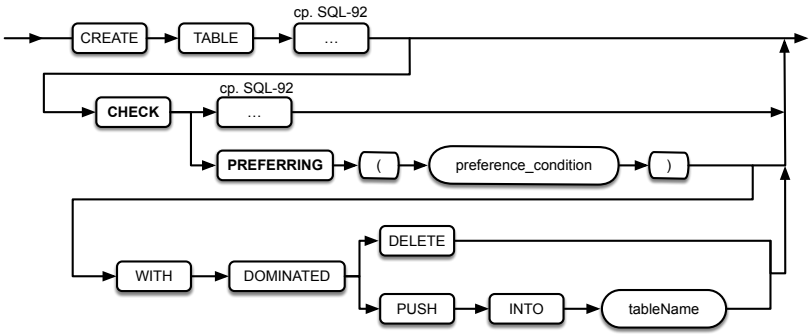

Fig. 9. Syntax diagram for preference-based integrity constraints.

The preference constraint follows the standard **`CREATE TABLE`** syntax [29]. After the keyword **`CHECK PREFERRING`** one can specify the preference condition a new tuple must fulfill. Since preferences apply to tuples, the new tuple must be compared to the objects already in the table. If the new one is dominated it can be discarded (**`WITH DOMINATED DELETE`**); this is also the default value. On the other hand one may decide to store dominated objects in another table specified by the *tableName* (**`WITH DOMINATED PUSH INTO`** ...). The preference constraints show a similar behaviour when a new tuple dominates objects already in the table. Then the dominated tuples can be deleted immediately or moved to another table containing worse objects.

Holding already dominating tuples could be useful, e.g., if one wants to get the Top-k elements of a preference query and the result w.r.t. the preference does not provide enough objects (cp. [23]). Hence, the result must be filled-up with the next better objects until k objects are retrieved. In this sense we also can specify a cascade of tables holding "worse tuples".

Example 6. Reconsider Example 5, but now we want to specify a table constraint instead of analyzing the streams directly within Preference SQL. Figure 10 shows the **`CREATE`**

TABLE statement, where attributes like author or location are stored and dominated objects should be moved to a table called *worseTweetsTable*.

```
CREATE TABLE em2016_tweets(
created_at VARCHAR(30),
author VARCHAR(20),
location VARCHAR(30),
text VARCHAR(100),
CHECK PREFERRING (hashtag  IN ('#EURO2016', '#em2016')
             AND  location IN ('Lyon','Marseilles')
             AND  author   IN ('UEFA EURO 2016',
                               'EM 2016 Frankreich'));
WITH DOMINATED PUSH INTO worseTweetsTable;
```

Fig. 10. Creating relation with preference based integrity constraints

Note that constructing a cascade of worse objects means to specify preference based integrity constraints for the table *worseTweetsTable* and move the dominated objects to a third relation, and so on.

6 Conclusion

In this paper we proposed a preference based approach for stream analysis. We suggested a language for connecting and querying streams in Preference SQL. In addition, we presented the possibility to specify integrity constraints based on preference constructors. We have implemented our approach based on Twitter data, because of its open API. However, the same approach can be used for other data streams, such as stock exchanges or Facebook posts.

Our work is in progress and thus, we plan to implement further stream connectors and to extend the preference stream query language. A comprehensive experimental evaluation as well as a comparison to other existing stream analysis frameworks is also future work.

References

1. J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings of SIGMOD '00*, pages 379–390, New York, NY, USA, 2000. ACM.
2. P. Bonnet, J. Gehrke, and P. Seshadri. Towards Sensor Database Systems. In *Proceedings of MDM '01*, pages 3–14, London, UK, UK, 2001. Springer-Verlag.
3. J. Sankaranarayanan, H. Samet, B. E. Teitler, M. D. Lieberman, and J. Sperling. Twitterstand: News in Tweets. In *Proceedings of ACM '09*, pages 42–51, 2009.
4. L. Liu, C. Pu, and W. Tang. Continual Queries for Internet Scale Event-Driven Information Delivery. *IEEE Trans. on Knowl. and Data Eng.*, 11(4):610–628, July 1999.
5. L. Golab and M. T. Özsu. Issues in Data Stream Management. *SIGMOD 03*, 32(2):5–14.
6. W. Kießling, Endres M., and Wenzel F. The Preference SQL System - An Overview. *Bulletin of the Technical Commitee on Data Engineering, IEEE CS*, 34(2):11–18, 2011.

7. C. Smith. DMR Twitter Statistic Report. http://expandedramblings.com/index.php/downloads/twitter-statistic-report/, 2015. Accessed: 2016-04-22.
8. R. Kelly. Twitter Study (Pearanalytics). `http://pearanalytics.com/wp-content/uploads/2009/08/Twitter-Study-August-2009.pdf`, 2009. Accessed: 2016-04-19.
9. S. Babu and J. Widom. Continuous Queries over Data Streams. *SIGMOD Rec.*, 30(3):109–120, September 2001.
10. B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *Proceedings of PODS '02*, pages 1–16, New York, USA, 2002. ACM.
11. E. R. Faria, I. J. C. R. Gonçalves, A. C. P. L. F. de Carvalho, and J. Gama. Novelty detection in data streams. *Artificial Intelligence Review*, 45(2):235–269, 2016.
12. G. Krempl, I. Žliobaite, D. Brzeziński, E. Hüllermeier, M. Last, V. Lemaire, T. Noack, A. Shaker, S. Sievi, M. Spiliopoulou, and J. Stefanowski. Open Challenges for Data Stream Mining Research. *SIGKDD Explor. Newsl.*, 16(1):1–10, September 2014.
13. M. Kontaki, A. N. Papadopoulos, and Y. Manolopoulos. *Continuous Processing of Preference Queries in Data Streams*, pages 47–60. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
14. J. A. Silva, E. R. Faria, R. C. Barros, E. R. Hruschka, A. C. P. L. F. de Carvalho, and J. Gama. Data Stream Clustering: A Survey. *ACM Comput. Surv.*, 46(1):13:1–13:31, July 2013.
15. R. D. Baruah, Angelov P., and Baruah D. Dynamically Evolving Clustering for Data Streams. In *Evolving and Adaptive Intelligent Systems (EAIS), 2014 IEEE Conference on*, pages 1–6.
16. X. Gu and P. P. Angelov. Autonomous Data-Driven Clustering for Live Data Stream. 2016.
17. M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, Y. Cetintemel, U.and Xing, and S. Zdonik. Scalable Distributed Stream Processing. In *CIDR 2003 - First Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2003.
18. A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. STREAM: The Stanford Stream Data Manager (Demonstration Description). In *Proceedings of SIGMOD '03*, pages 665–665, New York, NY, USA, 2003. ACM.
19. M. Okazaki and Y. Matsuo. Semantic Twitter: Analyzing Tweets for Real-Time Event Notification. In *BlogTalk*, volume 6045 of *Lecture Notes in Computer Science*, pages 63–74. Springer, 2009.
20. C. Railean and A. Moraru. Discovering Popular Events From Tweets. *Conference on Data Mining and Data Warehouses (SiKDD)*, October 2013.
21. J. Chomicki. Preference Formulas in Relational Queries. In *TODS '03: ACM Transactions on Database Systems*, volume 28, pages 427–466, 2003.
22. W. Kießling. Foundations of Preferences in Database Systems. In *Proceedings of VLDB '02*, pages 311–322. VLDB Endowment, 2002.
23. M. Endres and T. Preisinger. Behind the Skyline. In *Proceedings of DBKDA*. IARIA, 2015.
24. B. Babcock, M. Datar, and R. Motwani. Sampling from a Moving Window over Streaming Data. In *Proceedings of SODA '02*, pages 633–634, Philadelphia, PA, USA, 2002.
25. B. Hafenrichter and W. Kießling. Optimization of Relational Preference Queries. In *Proceedings ADC*, pages 175–184, Darlinghurst, Australia, 2005. Australian Computer Society.
26. A. Arasu, S. Babu, and J. Widom. *CQL: A Language for Continuous Queries over Streams and Relations*, pages 1–19. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.
27. A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal*, 15(2):121–142, June 2006.
28. N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Cetintemel, M. Cherniack, R. Tibbetts, and S. B. Zdonik. Towards a Streaming SQL Standard. *PVLDB*, 1(2):1379–1390, 2008.
29. SQL-92. Database Language SQL. Document ISO/IEC 9075:1992. ANSI Document X3.135-1992 (SQL92 Standard), 1992.