Online View Maintenance for Continuous Query Evaluation

Soheila Dehghanzadeh
INSIGHT, NUI Galway

Daniele Dell'Aglio
DEIB, Politecnico di Milano

Shen Gao
IFI, University of Zurich

Emanuele Della Valle
DEIB, Politecnico di Milano

Alessandra Mileo
INSIGHT, NUI Galway

Abraham Bernstein
IFI, University of Zurich

ABSTRACT

In Web stream processing, there are queries that integrate Web data of various velocity, categorized broadly as *streaming* (i.e., fast changing) and *background* (i.e., slow changing) data. The introduction of local views on the background data speeds up the query answering process, but requires maintenance processes to keep the replicated data up-to-date. In this work, we study the problem of maintaining local views in a Web setting, where background data are usually stored remotely, are exposed through services with constraints on the data access (e.g., invocation rate limits and data access patterns) and, contrary to the database setting, do not provide streams with changes over their content. Then, we propose an initial solution: WBM, a method to maintain the content of the view with regards to query and user-defined constraints on accuracy and responsiveness.

Categories and Subject Descriptors

H.3.5 [**Information Storage and Retrieval**]: On-line Information Services

1. INTRODUCTION

In the Web, there is a growing presence of data streams, i.e., data characterized by a high velocity, such as data from social networks (e.g., Twitter and Instagram) and smart city services (e.g., real-time position of buses and trains). The presence of data streams on the Web poses new challenges to stakeholders interested in analyzing Web data. One of the most common challenge is responsiveness, i.e., providing the outputs within a time constraint. State of the art solutions rely on research made in Data Stream Management Systems and Complex Event Processing, where several techniques are proposed to process and analyze data streams online [1].

Complex analyses often require to combine dynamic (data streams) and quasi-static (background data) data. For example, consider an advertisement setting[1], where the cloth brand ACME wants to hire influential Twitter users to post commercial endorsements. Those influential users are identified through a query based on their two main characteristics. First, users must be "trend setters", i.e., there are more than 1000 tweets mentioning them in the past 60 minutes. Second, users must be famous, i.e., they have more than 10000 followers. To take precedence over the rival companies, ACME wants to identify new influential users as soon as possible, with updates every 15 minutes, and can accept approximate results with at least 75% accuracy.

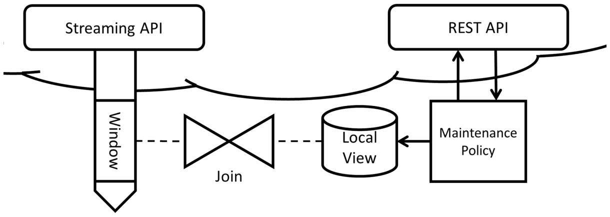

Figure 1: Query Processor Overview

To perform queries like this one, it is necessary to combine data streams (tweets) with background data (the number of followers), as shown in Figure 1. While the former are usually pushed into the query processor through push-based APIs (e.g., the Twitter Streaming API[2]) and processed through windows (time-varying subsets of the streams), the quasi-static data have to be accessed on demand through the invocation of pull-based APIs (e.g., the Twitter REST API[3]). However, the access cost to the remote services is high, and affects the responsiveness of the system. Moreover, the remote background data providers can become temporarily unavailable or they can set rate limits on the number of invocations (e.g., Twitter has different rate limits for each exposed service[4]), Furthermore, they can have predefined data access patterns constraints (e.g., by using the Twitter REST API it is possible to get the number of followers given a user, but it is not possible to ask for a list of users with more than 10000 followers).

Query processors can use *local views* to store the pulled data, in order to improve the performance, availability and scalability of the query processing [2]. However, the quality of the local view degrades over time due to the fact that the remote data can be modified and the local view is not updated. An efficient maintenance policy has to be introduced to keep the local view consistent with the remote data.

[1] Inspired by Chris Testa's SemTech 2011 talk: `http://goo.gl/kLSqGo`.

[2] `https://dev.twitter.com/streaming/overview`

[3] `https://dev.twitter.com/rest/public`.

[4] `https://dev.twitter.com/rest/public/rate-limits`.

WWW 2015 Companion, May 18–22, 2015, Florence, Italy.
ACM 978-1-4503-3473-0/15/05.
http://dx.doi.org/10.1145/2740908.2742761 .

2. ANALYSIS OF THE PROBLEM

A number of requirements lead to the design of a *Maintenance Policy* (MP), which has to take into account not only features already studied in the database literature (e.g., join selectivity), but also new ones introduced by the Web setting and the presence of Web streaming data.

R1: Data Access Constraints. The MP retrieves the background data through Web APIs. As explained above, the MP has to take into account the service constraints such as: the **data access patterns (R1.1)** have to be conform the API definitions ; the **access rate (R1.2)** of the pulling requests has to be below a limit.

R2: Quality of Service (QoS) Constraints. Each query has constraints over its **response time** and the **accuracy** of its answer. The MP must deal with the trade-off between them: frequent maintenance of local views leads to higher accuracy in the response, however, more time is required. The MP should monitor the quality of the answering process and fulfill the **QoS constraints** (R2).

R3: (Dynamic) Change Rate Distribution. The MP should consider the fact that different elements in background data can change over time at **different rates (R3.1)**. E.g., the number of followers of famous Twitter users changes often than the number of followers of others. Moreover, the change rate of elements can vary over time. E.g., the number of followers of a user changes more often during Twitter activity peeks and less often otherwise. Therefore, the MP may consider the **dynamic change rates (R3.2)**.

R4: Query Features. Joins may pair streaming and background data. The MP may exploit this unique feature to optimize the maintenance. Specifically, the MP may exploit the **sliding window definition (R4.1)**: at each evaluation, part of the window content does not change (as the window slides), and it can be used to select the local view elements to be maintained. The MP may also consider the **join selectivity (R4.2)** to find the most influential data elements on response accuracy.

3. PROPOSED SOLUTION

In this section, we propose the *Window-Based Maintenance policy* (WBM), which exploits the different change rates of the data elements (R3.1) and the sliding window definition (R4.1) of the query.

Consider the example in Figure 2. At each iteration, the query processor processes the elements in the current window and finds their join counterparts in the local view. E.g., at time 8, it uses the content of the W_1 window and selects $\{a, b, c, d\}$; at time 9, it uses W_2 and selects $\{b, c, d\}$. WBM is limited by an *update budget*, i.e., the maximum number of updates, defined w.r.t. the data access constraints (R1) and the responsiveness requirements (R2). In the example, the update budget is set to 1.

Among window elements, WBM only maintains their join counterparts in the local view. WBM works as follow: (1) it identifies the expired elements by estimating their change rate information (R3.1); in the example, it estimates that at time 8, a, c and d are expired in the local view. (2) Next, it computes in how many future windows they are going to stay, e.g., for a is 0 (it is not in W_2), c and d are 2 (they exit in W_4). (3) Based on the change rate information, WBM also estimates the next expiration time of a data element, *if updated now*, e.g., a will expire again at 12, c at 11, and d at 9. (4) Finally, WBM chooses the selected candidates that will save it most number of future updates. In the example, a is the one that would stay more time up-to-date, but it is going to exit the window in W_2. c and d, in contrast, will be also in W_3 and W_4, so WBM gives them priority. Between c and d, WBM chooses c, because it would stay up-to-date for a longer time than d, and, consequently, does not need to be updated anymore in the (near) future.

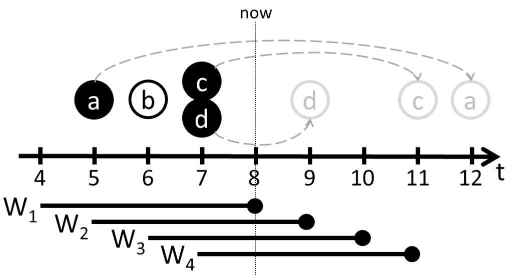

Figure 2: Selection process in the WBM policy

4. RESULTS AND CONCLUSION

We implemented WBM and we compared it with two query-driven baselines policies inspired by the Least Recently Used (LRU) and the Random (Rand) page replacement algorithms. We evaluated the policy against a synthetic data set and a real data set collected from Twitter. In the synthetic data, each element is assigned a different change rate sampled from a normal distribution.

	Rand	LRU	WBM
Synthetic	0.72	0.73	**0.83**
Twitter	0.60	0.66	**0.76**

Table 1: Accuracy of results in the two data sets

Table 1 shows the results of the experiments: it reports the accuracy of the policies. In both experiments, Rand shows the worst performance. Comparing WBM and LRU, WBM outperforms LRU by up to 13.5%, as WBM is more precise in the identification of expiring data by estimating the change rates. In the real data, the change rate distribution is highly skewed, i.e., many elements are changing slowly, and only a few have high change rates. Here, WBM shows even better improvements, as it updates only the expired data and LRU wastes more budgets on still valid data.

By considering the change rate (R3.1), WBM has improved the accuracy of results significantly. As an initial study, WBM does not consider the dynamic change rates of data (R3.2) and the join selectivity (R4.2) . In future, we will extend WBM in those directions.

Acknowledgments

This work has been partially funded by Science Foundation Ireland grant No. SFI/12/RC/2289.

5. REFERENCES

[1] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15, 2012.

[2] H. Guo, P.-Å. Larson, and R. Ramakrishnan. Caching with good enough currency, consistency, and completeness. In *VLDB 2005*, pages 457–468, 2005.