Data Stream Query Processing on Mobile Devices

Mitra Kazemzadeh
Department of Mathematics and Computer Science
University of Lethbridge
Lethbridge, Alberta, Canada
mitra.kazemzadeh@uleth.ca

Wendy Osborn*
Department of Mathematics and Computer Science
University of Lethbridge
Lethbridge, Alberta, Canada
wendy.osborn@uleth.ca

***Abstract*—Nowadays, data management and processing systems exist that handle streaming data. Streaming data is data that is very large and cannot all be stored, and also may only be valid for a certain period of time. In existing stream data managements systems, data streams are transmitted to a server where the management and processing of queries take place, before the results are transmitted on an outbound stream - possibly to another mobile device. However, no strategies exist that receive, manage and process stream data directly on a mobile device, without having to rely on the services of a server. This paper proposes an architecture and framework for stream data management and query processing on a mobile device. The architecture only keeps a minimal number of features. All database operations are handled with minimal data storage requirements. It also presents details on an implementation of this framework for a currency conversion application, which demonstrates the utility of our framework. This work will lead to several new directions of research in the area of stream processing on mobile devices.**

***Index Terms*—data streams, mobile computing**

I. Introduction

Nowadays, data management and processing systems exist that handle streaming data. A data stream (also known as streaming data) [1] is a data set where data items for it are generated continuously from some source, and may be considered obsolete after a certain period of time.

As mobile device usage increases, so do the opportunities for information processing applications for these devices. One such type of application is the processing of queries that involve streaming data. For example, a currency conversion application that continuously receives cash amounts in various currency and converts each to another currency using information in a currency table is an ideal application that can process this information locally on a mobile device.

In existing stream data managements systems, data streams are transmitted to a server where the management and processing of queries take place, before the results are transmitted on an outbound stream - possibly to a mobile device for display. However, to the best of our knowledge, no frameworks exist that receive, manage and process stream data directly on a mobile device, without having to rely on the services of a server. Having stream processing directly on a mobile device allows for the processing of stream queries to take place anywhere and at any time. In addition, the need for processing on an intermediate server before obtaining and displaying results can be eliminated, which in turn reduces the amount of network transmission that must take place. The trade-off here, however, is the limited storage available on mobile devices, which must be taken into account when supporting stream query processing applications.

This paper proposes a proof-of-concept architecture and functionality for stream data management and query processing on a mobile device. Both consider the properties of mobile devices - in particular, limited storage availability - for supporting stream queries. This paper also presents an implementation of this framework for a currency conversion application in order to demonstrate the utility of our framework. This work will lead to several new directions of research in the area of stream processing on mobile devices.

The remainder of this paper proceeds as follows. Section II summarizes existing research in the area of spatial data streams. Section III presents our proposed framework for query processing on mobile devices, including preliminaries, architecture and functionality details. Section IV presents an example application of the proposed framework. Finally, Section V concludes the paper and proposes directions for future work.

II. Related Work

In this section, we summarize some relevant related work in the area of stream query processing, with a focus on benchmarks and some more recent extensions. Babu and Widom [1] propose a server-based architecture for database query processing where the data being processed is in stream form, rather than stored form. They also address how existing database query operations can be processed in the architecture. Arasu, Babu and Widom [3] propose a SQL-based query language for specifying queries in a stream-based database system. They also present a related query execution plan, and processing strategies for this language assuming both streaming data and standard relations. Their approach also considers the sharing of results among multiple continuous queries. Maier *et al.* [5] address the issue of the actual definition of a data stream, and propose functions to address this. Jain *et al.* [6] proposal a unified stream querying standard that incorporates both time-based and tuple-based execution of continuous stream queries.

*Corresponding Author

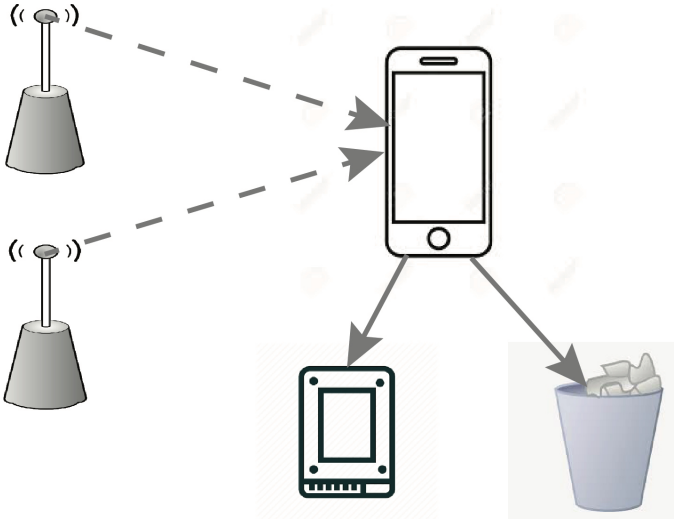

Fig. 1. Mobile Data Stream Processing Architecture

Stream processing strategies have been proposed for other types of data, such as spatial data streams. Kwon and Li [2] proposed a progressive spatial join strategy that joins the same pair of objects multiple times, beginning with a simplified representation (i.e. a minimum bounding box) and increasing the complexity of the representation for subsequent joins, in order to reduce the cost of the spatial join as simpler representations are computationally less expensive to join [2]. Osborn [4] proposed several strategies that utilize both a conventional spatial join and a bit-array join (i.e. similar to a Bloom join [7]). They employ the idea of a common region between the two spatial data streams to identify objects that would participate in a join.

One limitation of the above proposed architectures and strategies are that all process stream queries on a server, in which the result must be transmitted to the mobile device before it would be displayed. Eliminating the additional transmission step will result in more real-time processing and display of query results for the user.

III. Proposed Framework

In this section, we present our framework for data stream query processing on mobile devices. After summarizing some preliminaries, we present both the architecture and functionality of our framework.

A. Preliminaries

Our approach begins with adapting the architecture proposed by Babu and Widom [1] to a mobile environment. Their proposed architecture consists of the following components [1]:

- Input Stream 1 ... Stream n - the components that transmit data (e.g. tuples) to the data stream query processor.
- Query Processor - the component that processes queries on both the incoming tuples from the Input Streams, as well as tuples residing in the Scratch component (described below).
- Output Stream - the component where the tuples that are permanently part of the result are sent, to be received by others.
- Store - the component where result tuples that may not be permanently part of the result are kept. Along with the Output Stream, this make up the other part of the current result.
- Scratch - the component where tuples that may be required for future processing are kept. For example, if tuples are needed for joining with one that has not arrived yet, then it is stored here until it is discarded (i.e. either due to not longer being needed, or due to lack of space).
- Throw - the component where tuples that are no longer needed are sent. As mentioned by Babu and Widom, this component is generally not implemented in an active system.

B. Architecture

Figure 1 presents our overall architecture for mobile data stream processing architecture. The input for our architecture consists of two data streams, which transmit data directly to the mobile device. It also consists of a scratch storage, which is used for storing the "current" data that has arrived from the data streams. Although this is depicted externally in the diagram, this storage is actually within the mobile device. Finally, it consists of a throw, which is for tuples that will not be kept anymore.

Our architecture differs from [1] in the following ways:

- In theory our architecture could be extended to support multiple input streams. However, when taking storage limitations on a mobile device into consideration, we concluded that it makes sense to only support a limited number of streams. Therefore, our current architecture only supports up to two input streams.
- Our proposed architecture does not have an Output Stream. Instead, the result of a query at a given time is displayed on the screen of the mobile device.
- In addition, our proposed architecture does not maintain a store of results that may or may not remain in the result "forever". Therefore the only result consists of the tuples on the display of the mobile device at a given moment in time.

C. Functionality

In our framework, a query is only submitted once to the stream query processor on the mobile device. We assume tuple-based execution of a continuous query - specifically, a query is executed continuously for each new tuple that arrives from the input streams.

We have several table types available for use, including an append-only table, and a temporal table. In an append-only table, new tuples are simply appended to the end of the table, with tuples at the beginning of the table being removed if the capacity of the table has been reached. The temporal table consists of a list of "current" tuples, with outdated tuples being

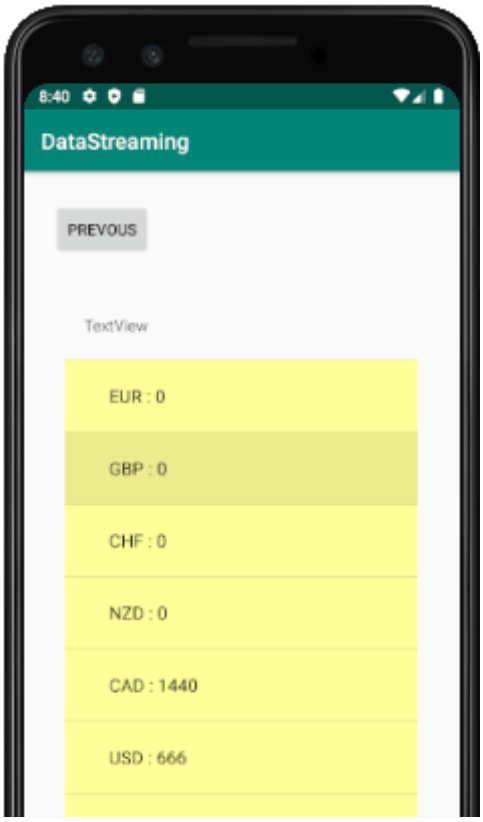

Fig. 2. Initial List of Currencies

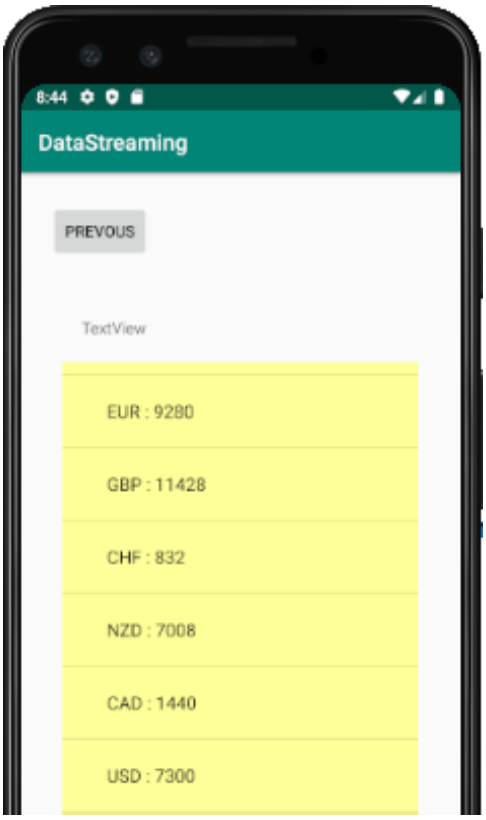

Fig. 3. After Processing of More Orders

archived in a separate list. The current list is the list that is utilized for query processing operations.

As mobile devices have very limited storage, the advantages of using both of the above table types are the following:

- The append-only table can be easily adjusted to accommodate a specific number of tuples - from none (i.e. not storing tuples at all after processing them on arrival) to some number depending on the availability of memory on the device.
- If outdated data is not required to be kept, then the temporal relation can be reduced to a standard, updatable table. However, the flexibility exists to store this information when a temporal table is used.

Our proposed framework supports select, aggregation, projection, and join operations that have minimal storage requirements. For the select and project operations, if both are being performed on an append-only stream we do not need to utilize the scratch storage. In both cases, tuples that meet either a selection and/or projection condition can be displayed to the user immediately, and do not need to be stored for future use.

Similar reasoning applies to aggregation operations. An incoming tuple on the append-only stream can e processed for whichever aggregate calculations are required. For example, if the value of one of the attributes is part of a running sum, this can be updated before it is stored. All running aggregate calculations will need to be stored in the scratch storage. But this is at most one value per aggregate formula, or possibly one per formula per group-by attribute value. However, the actual incoming tuples t do not need to be stored for future use.

A join operation require more consideration, but under certain conditions can be supported with minimal scratch storage requirements. For a join operation, we utilize a nested loop join strategy, with the assumption that one stream will be stored in an append-only relation, and the other stored in a temporal relation. The latter stores “current” information that may be updated, while the former stores information that - on a tuple basis - cannot be updated. If the submitted query only requires results to be current the moment they are generated, then only the temporal relation needs to be stored, while storage of the append-only relation is not necessary. If updates to tuples in the temporal relation require new joins with the tuples in the append-only relation, then the append-only tuples need to be stored in scratch storage, so they can be processed again if need be. However, for some applications, once the tuple is processed, any changes to tuple that it joined with does not affect the tuple that was generated previously.

We demonstrate the following situation below - no storage in the append-only relation, no storage of outdated data in the temporal relation - for an application that does not require the incoming tuples of one stream to be kept after processing, and only the current tuples of the other stream to be maintained.

IV. Application Example

To demonstrate the use of our proposed framework, we implemented an Android application that tracks ongoing sales of some of the top world currencies. This application was implemented using Apache Flink.[1]

This application obtained its input from two continuous data streams. The first is ExchangeRates, which keeps track of changes to the exchange rate of different currencies to the Japanese Yen. Each tuple in ExchangeRates consists of:

```
(time, currency, rate)
```

The second input stream is Orders, which keeps track of the amount of money paid in a particular currency for a particular product. Each tuple in the Orders stream consists of:

```
(time, productID, currency, amount)
```

For demonstration purposes, the data for both streams are randomly generated. The query that we are interested in performing on these data streams is:

```
select ExchangeRate.currency,
    sum(Orders.amount * ExchangeRate.rate)
from Orders, RateHistory
where Orders.currency=ExchangeRate.currency
group by ExchangeRate.currency;
```

The result of this query is a list of currencies, and the current total converted amount in Japanese Yen.

With respect to our proposed architecture above, the only components that need to be stored are: 1) the current exchange rates, and 2) the running totals of each currency converted into Yen, which is stored alongside each exchange rate. As we are only working with six exchange rates, we require very little storage on our mobile device. We do not need to store the incoming orders, since its converted amount depends only on the current exchange rate. Therefore, once each Order is processed it can be discarded.

Figures 2 and 3 depict the display of the current results, initially and after some time. In Figure 2, we can see that so far, only orders for conversion from Canadian (CAD) and United States (USD) dollars have been processed, with no orders having been processed for the British Pound (GBP). In Figure 3 we can see that the total of each currency (after conversion to Yen) increases as more orders arrive, and are updated on the display. For example, for GBP, we see orders that now total and convert to 11,428 Yen is displayed (Figure 3), while the total converted amount for CAD has not changed, since no further orders have arrived for this conversion. We also observe similar trends with the other currencies.

V. Conclusion

In this paper, we propose an architecture and functionality for stream data management and query processing framework on a mobile device. Both consider the properties of mobile devices in supporting stream queries. In particular, our proposed architecture does not maintain a separate result store, nor does it generate an output stream, as the the result is displayed immediately and continuously to the user. With respect to functionality, many application can benefit from database operations that only require minimal storage in order to generate results.

An implementation of this framework for a currency conversion application is also provided to demonstrate accurate results that require minimal storage, which is ideal for data stream processing on a mobile device.

This proof of concept has lead to several research directions, including the following. The first deals with handling database operations that work on updatable stream data - that is, a tuple that arrives later may be a replacement for an earlier tuple. In addition, having two streams that are not considered temporal relations will affect the storage requirements of the system, and would need consideration. Second, our system currently assumed that the data arrives in the order it is generated. If tuples arrive out of order, this can affect the accuracy of the result (e.g. from our example above, the Order tuple arriving much later than the exchange rate it was intended for). Third, our framework assumes tuple-based execution of a continuous query. Time-based execution is also a consideration. However such an approach will require different storage considerations, as tuples would need to be stored. Finally, our framework currently assumes that the number of tuples in the temporal relation is static, even if they are updatable during the execution of the application (e.g. from our example, if a new exchange rate arrives after Orders show up for it, and the Orders have been deleted). These and other considerations lead to exciting research in data stream query processing on mobile devices.

[1] URL: flink.apache.org

References

[1] S. Babu and J. Widom, "Continous queries over data streams," *SIGMOD Record*, vol. 30, no. 3, pp. 109–120, 2001.

[2] O. Kwon and K.-J. Li, "Progressive spatial join for polygon data stream," in *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 2011.

[3] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: semantic foundations and query execution," *VLDB Journal*, vol. 15, no. 2, 2006.

[4] W. Osborn, "Exploring bit arrays for join processing in spatial data streams," in *Proceedings of the 22nd International Conference on Network-Based information Systems*, 2019, pp. 73–85.

[5] D. Maier, J. Li, P. A. Tucker, K. Tufte, and V. Papadimos, "Semantics of data streams and operators," in *Proceedings of the 10th International Conference on Database Theory (ICDT 2005), Edinburgh, UK, January 5-7, 2005, Proceedings*, ser. Lecture Notes in Computer Science, T. Eiter and L. Libkin, Eds., vol. 3363. Springer, 2005, pp. 37–52.

[6] N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Çetintemel, M. Cherniack, R. Tibbetts, and S. B. Zdonik, "Towards a streaming SQL standard," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1379–1390, 2008.

[7] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.