Incremental Stream Processing of Nested-Relational Queries

Leonidas Fegaras

University of Texas at Arlington
fegaras@cse.uta.edu

Abstract. Current work on stream processing is focused on approximation techniques that calculate approximate answers to simple queries by focusing on a fixed or sliding window that contains the most recent tuples from an input stream and by using condensed synopses to summarize the state. It is widely believed that without using approximation techniques, most interesting queries would be blocking (i.e., they would have to wait for the end of stream to release their results) or unbounded (i.e., their memory requirements would grow proportionally to the stream size, which may be infinite). The goal of this paper is to convert nested-relational queries to incremental stream processing programs automatically. In contrast to most current stream processing systems that calculate approximate answers, our system derives incremental programs that return accurate results. This is accomplished by retaining a state during the query evaluation lifetime and by using incremental evaluation techniques to return an accurate snapshot answer at each time interval that depends on the current state and the data in the current fixed window. Our methods can handle most forms of declarative queries on nested data collections, including arbitrarily nested queries, group-by with aggregation, and equi-joins. We report on a prototype system implementation and we show some preliminary results on evaluating queries on a small computer cluster running Spark.

1 Introduction

New frameworks in Big Data analytics have become indispensable tools for large-scale data mining and scientific data analysis. Currently, the most popular framework for Big Data processing is Map-Reduce [12], which has emerged as a powerful, generic, and scalable solution for a wide range of data analysis applications. Unfortunately, to simplify fault tolerance and recovery, the Map-Reduce model does not preserve data in memory between consecutive jobs, which inflicts a high overhead on complex workflows and repetitive algorithms, such as PageRank and data clustering. Although the Map-Reduce framework was originally designed for batch processing, there are several recent systems that have extended Map-Reduce with on-line processing capabilities, such as MapReduce Online [11], Incoop [7], and i^2MapReduce [37]. In addition, many distributed stream processing engines (DSPEs) have emerged recently, such as Apache Storm [28], Spark's D-Streams [36], and Flink Streaming [16]. Most of these systems use data stream processing techniques based on fixed or sliding windows and incremental operators [4], which have already been used successfully in relational stream processing systems, such as Aurora [1] and Telegraph [10].

Furthermore, there is a recent interest in incremental data analysis, where data are analyzed in incremental fashion, so that existing results on current data are reused and merged with the results of processing the new data. In many cases, incremental data processing can achieve better performance and may require less memory than batch processing for many common data analysis tasks. Incremental processing can also be used for analyzing large amounts of data incrementally, in small batches that can fit in memory, thus enabling to process more data with less hardware. It can also be very valuable to stream-based applications that need to process continuous streams of data in real-time with low latency, which is not possible with existing batch analysis tools.

We are presenting a novel incremental stream processing framework for large-scale data analysis queries that run on a distributed stream processing engine (DSPE). Our design objective is to be able to convert any batch data analysis query to an incremental distributed stream processing program automatically, without requiring the user to rewrite the query. Furthermore, in contrast to most current stream processing systems that calculate approximate answers, we want our system to derive incremental programs that return accurate results. Such a task requires a query analysis to separate the query parts that can be used to process the incremental batches of data from the query parts that merge the current results with the new results of processing the incremental batches of data. Such analysis is more tractable if it is performed on declarative queries than on algorithmic programming languages, such as Java. We have developed our framework on Apache MRQL [26], because it is the only Big Data query language powerful enough to express complex data analysis tasks, such as PageRank, data clustering, and matrix factorization. We have developed general methods to transform batch MRQL queries to incremental queries. The derived incremental queries retain a state during their evaluation lifetime and use incremental evaluation techniques to return an accurate snapshot answer at each time interval that depends on the current state and the latest batches of data.

The first step in our approach is to transform a query so that it propagates the join and group-by keys to the query output. This is known as lineage tracking ([5,6]). That way, the values in the query output are grouped by a key combination that corresponds the join/group-by keys used in deriving these values during query evaluation. If we also group the new data in the same way, then computations on existing data can be combined with the computations on the new data by joining the data on these keys. Our approach requires that we can combine computations on data that have the same lineage to derive incremental results. In our framework, this is accomplished by transforming a query to a *monoid homomorphism* by removing the non-homomorphic parts of the query, using algebraic transformation rules, and by combining them to form an answer function. The remaining query, which is now a monoid homomorphism, is used to derive the state transformation that merges the existing state with the new results to form a new state.

We have implemented our incremental processing framework on MRQL [26] on top of Spark Streaming [36], which is an in-memory distributed stream processing platform. Our system is called *Incremental MRQL*. MRQL is currently the best choice for implementing our framework because other query languages for data-intensive, distributed computations provide limited syntax for operating on data collections, in the

form of simple relational joins and group-bys, and cannot express complex data analysis tasks, such as PageRank, data clustering, and matrix factorization, using SQL-like syntax exclusively. Our framework, though, can be easily adapted to apply to other query languages, such as Hive, PigLatin, SQL, XQuery, and Jaql.

The contribution of this work can be summarized as follows:

- We present a general framework for translating nested-relational data analysis queries to incremental stream processing programs that can run on a distributed stream processing platform.
- This framework can handle many forms of queries over nested data, including deeply nested queries, group-by with aggregation, and equi-joins.
- We report on a prototype system implementation and we show some preliminary results on evaluating three queries, groupBy, join-groupBy, and PageRank step, on a small computer cluster running Spark.

2 MRQL Overview

Apache MRQL [26] is a query processing and optimization system for large-scale, distributed data analysis. MRQL was originally developed by the author [14,13], but is now an Apache incubating project with many users and developers. The MRQL language is an SQL-like query language for large-scale data analysis on clusters of commodity hardware. The MRQL query processing system can evaluate MRQL queries in four modes: in Map-Reduce mode using Apache Hadoop, in BSP mode (Bulk Synchronous Parallel model) using Apache Hama, in Spark mode using Apache Spark [31], and in Flink mode using Apache Flink. The MRQL query language is sufficiently powerful to express many data analysis tasks over many different kinds of raw data, such as XML documents, JSON documents, binary files, and text documents in CSV format. The design of MRQL has been influenced by XQuery and OQL. In fact, when restricted to XML data, MRQL is as powerful as XQuery. MRQL is more powerful than other current high-level Map-Reduce languages, such as Hive and PigLatin, since it can operate on more complex data and supports more powerful query constructs, thus eliminating the need for using explicit procedural code. With MRQL, users are able to express complex data analysis tasks, such as PageRank, k-means clustering, matrix factorization, etc, using SQL-like queries exclusively, while the MRQL query processing system is able to compile these queries to efficient Java code that can run on various distributed processing platforms. The MRQL system stack is shown in Figure 1.

A recent extension to MRQL, called *MRQL Streaming*, supports the processing of continuous MRQL queries on streams of batch data (that is, data that come in continuous large batches). Before our incremental MRQL work presented in this paper, MRQL Streaming supported traditional window-based streaming based on a sliding window during a specified time interval. Currently, MRQL Streaming works on Spark Streaming only but we are currently adding support for Storm and Flink Streaming. The work reported here, called *Incremental MRQL*, extends the current MRQL Streaming engine with incremental stream processing. Incremental MRQL is now available in the latest official MRQL release (MRQL-0.9.6).

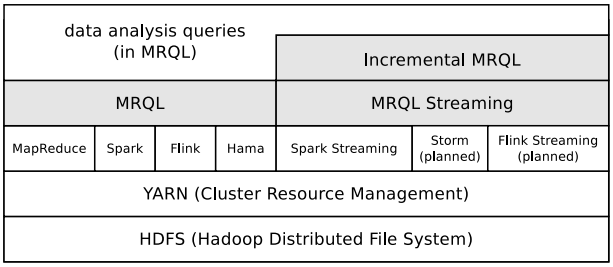

Fig. 1. The MRQL System Stack

3 Incremental Query Processing

The MRQL data model consists of collections types, such as lists (sequences), bags (multisets), and key-value maps. The difference between a list and a bag is that a list supports order-based operations, such as indexing and subsequence. In addition, MRQL supports records, tuples, algebraic data types (union types), and basic types, such as integers and booleans. These types can be freely nested, thus supporting nested relations and hierarchical data. For example, XML data can be represented as a recursive algebraic data type with two value constructors, Node and CData:

```
data XML = Node: < tag: String, attributes: bag( (String,String) ),
                   children: list (XML) >
         | CData: String
```

Non-streaming MRQL queries work on datasets, which are stored in the distributed file system (HDFS). A dataset is a bag (multiset) that consists of arbitrarily complex values. Datasets are stored as text files, such as XML, JSON, and CSV, or sequence (binary) files. Streaming MRQL queries work on 'batch' data streams, where stream data arrive in batches (bags) of new data. For example, the query:

```
select (x,avg(y)) from (x,y) in stream(binary,"data/points") group by x
```

groups a stream of points (x, y) by x and returns the average y values in each group. The MRQL Streaming engine will first process all the existing sequence files in the directory data/points and then will check this directory periodically for new files. When a new file is inserted in the directory, MRQL will process the new batch of data using distributed processing. In addition to directory of files, MRQL Streaming supports a special socket input format for listening to TCP sockets for text input, based on one of the current supported MRQL Parsed Input Formats (XML, JSON, CSV). A query may work on multiple stream sources and multiple batch dataset sources. If there is at least one stream source in the query, the query becomes continuous, that is, it never stops. The output of a continuous query is dumped into a directory HDFS as a sequence of files, so that each file in the sequence contains the results of processing a single batch of streaming data.

Our incremental query processing framework can handle continuous queries over a number of streaming data sources, $S_1, \ldots, S_n$, denoted by $\overline{S}$. A data stream S_i in

our framework consists of an initial dataset, followed by a continuous stream of incremental batches ΔS_i, which arrive at regular time intervals Δt. In MRQL Streaming, these are batch data streams, where stream data arrive in batches of new data in the form of new files created inside some pre-specified directories. Then, a streaming query can be expressed as $q(\overline{S})$, where an $S_i \in \overline{S}$ is a streaming data source. Incremental stream processing is attainable if we can derive the query results at time $t + \Delta t$ by combining the query results at time t with the results of processing ΔS_i, rather than processing the query over the entire streams $S_i \uplus \Delta S_i$, where $\uplus$ is bag union. This is possible if $q(\overline{S \uplus \Delta S})$ can be expressed in terms of the current query result, $q(\overline{S})$, and the incremental query result, $q(\overline{\Delta S})$, that is, when $q(\overline{S})$ is a monoid homomorphism over $\overline{S}$. In abstract algebra, a monoid $\otimes$ is an associative function with a zero element $\otimes_z$, such that $x \otimes \otimes_z = \otimes_z \otimes x = x$. In addition, h is a monoid homomorphism if $h(x \oplus y) = h(x) \otimes h(y)$, for two monoids $\oplus$ and $\otimes$.

Unfortunately, some queries, such as counting the number of distinct elements in a stream or calculating average values after a group-by, are not (monoid) homomorphisms. The first query is not a homomorphism because when we count the distinct elements in a bag X we loose the information about which elements are contained in X, and therefore, counting the distinct elements of $X \uplus Y$ cannot be derived by combining the counts of the distinct elements in X and Y separately, since X and Y may have common elements. The average in the second query prevents the query from becoming a homomorphism because the average value is the sum divided by the count of values, and both these values are lost after we derive the query result. To handle a non-homomorphic query $q(\overline{S})$, we break q into two functions a and h, so that h is a homomorphism with $q(\overline{S}) = a(h(\overline{S}))$. Recall that function h is a homomorphism if $h(\overline{S \uplus \Delta S}) = h(\overline{S}) \otimes h(\overline{\Delta S})$ for some monoid $\otimes$. For example, the first query that counts the number of distinct elements can be broken into the query h that returns the list of distinct elements, which is a homomorphism, and the answer query a that counts the derived distinct elements. For this approach to be effective, most of the computations in q must be done in h, possibly leaving some computationally inexpensive data mappings to the answer function a. After we split the query q into an answer function a and a homomorphism h, we can calculate the results of h incrementally by storing its results into a state, which is maintained across the stream processing, and then combine the current state with the new data to calculate the next state instance. More specifically, at each time interval Δt, the query answer $h(\overline{S \uplus \Delta S})$ is calculated from the new state, $\text{state} \leftarrow \text{state} \otimes h(\overline{\Delta S})$, and is equal to $a(\text{state})$, which is the snapshot of the query answer at time $t + \Delta t$.

Our framework translates incremental MRQL queries into incremental query plans as follows. First, it pulls all non-homomorphic parts of a query q out from the query using algebraic transformations, leaving an algebraic homomorphic term h, such that $q(\overline{S}) = a(h(\overline{S}))$. Then, it combines these non-homomorphic parts into an answer function a. Finally, from the homomorphic algebraic term $h(\overline{S})$, our framework synthesizes a merge function $\otimes$, such that $h(\overline{S \uplus \Delta S}) = h(\overline{S}) \otimes h(\overline{\Delta S})$. All these tasks are performed using algebraic transformations on the MRQL query algebraic terms [14,13]. Although all algebraic operations used in MRQL are homomorphic, their composition may not be. We have developed transformation rules to derive homomorphisms from

compositions of homomorphisms, and for pulling non-homomorphic parts outside a query. Our methods can handle most forms of queries on nested data sets, including complex nested queries with any form of nesting and any number of nesting levels, complex group-bys with aggregations, and general one-to-one and one-to-many equi-joins. Our methods cannot handle non-equi-joins and many-to-many equi-joins, as they are very difficult to implement efficiently in a streaming or an incremental computing environment.

For example, consider the following MRQL query $q(S_1, S_2)$:

```
select (x, avg(z))
  from (x,y) in S1, (y,z) in S2
 group by x
```

where S_1 and S_2 are stream data sources. Here, the streams S_1 and S_2 are joined so that the second column of S_1 is equal to the first column of S_2, then the join result is grouped by the first column of S_1, and finally the average value of all z values in the group is returned. This query is not a homomorphism over both S_1 and S_2, that is $q(S_1 \uplus \Delta S_1, S_2 \uplus \Delta S_2)$ cannot be expressed in terms of $q(S_1, S_2)$ and $q(\Delta S_1, \Delta S_2)$, because the query result does not contain any information on how the avg(z) value is related to the x value. That is, there is no lineage in the query output that links a pair in the query result to the join key that contributed to this pair. Hence, it is impossible to tell how the new data ΔS_1 and ΔS_2 will contribute to the previous query results if we do not know how these results are related to the previous inputs S_1 and S_2. Our approach is to establish links between the query results and the parts of the data sources that were used to form their values. This is called *lineage tracking* and has been used for propagating annotations in relational queries [6]. In our case, this lineage tracking can be done by propagating all keys used in joins and group-bys along with the values associated with these keys, so that, for each combination of keys, we return one group of result values. For our query, this is done by including the join key y in the group-by keys. That is, the query is transformed to $h(S_1, S_2)$:

```
select ((x,y), (sum(z), count(z)))
  from (x,y) in S1, (y,z) in S2
 group by (x,y)
```

Hence, the join key y is propagated to the output values so that the avg components, sum and count, are aggregations over groups associated with unique combinations of x and y. This query is a homomorphism over S_1 and S_2, provided that the join is not on a many-to-many relationship. In general, a query with N join/group-by/order-by steps is transformed to a query that injects the join/group-by/order-by keys to the output so that each output value is associated with a unique combination of N keys. The answer query a that returns the final result in $q(S_1, S_2) = a(h(S_1, S_2))$ is:

```
select (x, sum(s)/sum(c))
  from ((x,y),(s,c)) in State
 group by x
```

where State is the current state, equal to $h(S_1, S_2)$. This query removes the lineage y (the join key) from the State but also groups the result by the group-by key again, since there may be duplicate x values, and returns the final average value. (Note that

sum(s) adds the partial sums while sum(c) adds the partial counts.) The merge function H1 $\otimes$ H2 of the homomorphism h, which combines tuples with the same lineage key, is a full outer-join on the lineage key that aggregates the matches:

```
select (k,(s1+s2,c1+c2))
  from (k,(s1,c1)) in H1,
       (k,(s2,c2)) in H2
union select (k,(s2,c2)) from (k,(s2,c2)) in H2 where k not in π1(H1)
union select (k,(s1,c1)) from (k,(s1,c1)) in H1 where k not in π1(H2)
```

where π_1 is a bag projection that retrieves the keys. The first select is an equi-join between H1 and H2 over the lineage key k, equal to the pair (x,y) that contains the group-by key x and the join key y. The first union returns all H2 pair that are not joined with H1, while the second union returns all H1 pair that are not joined with H2.

MRQL Streaming has been implemented on Apache Spark [31] running on an Apache Yarn cluster. More specifically, the incremental state is cached in memory as a Distributed DataSet (an RDD [35]), which is distributed across the worker nodes, while the streaming data sources are implemented as Discretized Streams (D-Streams [36]), which are also distributed across the worker nodes. The full outer-join used in the homomorphism h, which merges the query result on the new data with the current state, is implemented efficiently as a distributed hash-partitioned join (a Spark's coGroup operation), by keeping the state partitioned on the lineage keys and shuffling only the new results to worker nodes to be combined locally with the state using merging. That is, the results of processing the new data, which are typically substantially smaller than the state, are shuffled across the worker nodes before coGroup. Our approach is to keep the state partitioned on the lineage keys by simply leaving the partitions of the newly created state by coGroup at the place they were generated, since hash-partitioning generates data partitioned by the same join key. That way, only the results from the new data would have to be partitioned and shuffled across the working nodes to be combined with the current state.

4 The Translation Framework

The MRQL algebra consists of a small number of higher-order homomorphic operators [14], which are defined using structural recursion based on the union representation of bags [15]. The most important operation is cMap (also known as flatten-map in functional programming), which generalizes the select, project, join, and unnest operators of the nested relational algebra. Given two arbitrary types α and β, the operation $\mathrm{cMap}(f, X)$ maps a bag X of type $\{\alpha\}$ to a bag of type $\{\beta\}$ by applying the function f of type $\alpha \to \{\beta\}$ to each element of X, yielding one bag for each element, and then by merging these bags to form a single bag of type $\{\beta\}$. Using a set former notation on bags, $\mathrm{cMap}(f, X)$ can be expressed as $\{\, z \,|\, x \in X,\, z \in f(x) \,\}$. Using structural recursion, it can also be defined as a homomorphism:

$$\begin{aligned} \mathrm{cMap}(f, X \uplus Y) &= \mathrm{cMap}(f, X) \uplus \mathrm{cMap}(f, Y) \\ \mathrm{cMap}(f, \{a\}) &= f(a) \\ \mathrm{cMap}(f, \{\,\}) &= \{\,\} \end{aligned}$$

The second in importance operator is groupBy, which groups a bag of pairs by their first value. Given the arbitrary types κ and α, and a bag X of type $\{(\kappa, \alpha)\}$, the operation $\mathrm{groupBy}(X)$ groups the elements of the bag X by their first component and returns a bag of type $\{(\kappa, \{\alpha\})\}$. For example, groupBy({(1,10),(2,20),(1,30),(1,40)}) returns {(1,{10,30,40}),(2,{20})}. The groupBy operation can be defined using structural recursion:

$$\begin{aligned} \mathrm{groupBy}(X \uplus Y) &= \mathrm{groupBy}(X) \Uparrow_{\uplus} \mathrm{groupBy}(Y) \\ \mathrm{groupBy}(\{(k,a)\}) &= \{(k,\{a\})\} \\ \mathrm{groupBy}(\{\}) &= \{\} \end{aligned}$$

where the parametric monoid $\Uparrow_{\oplus}$ is a full outer-join that merges groups associated with the same key using the monoid $\oplus$ (equal to $\uplus$ for groupBy). It is expressed using a set-former notation for bags:

$$\begin{aligned} X \Uparrow_{\oplus} Y = &\{\, (k, a \oplus b) \,|\, (k,a) \in X,\ (k',b) \in Y,\ k = k' \,\} && \text{(join between } X \text{ and } Y\text{)} \\ \uplus\ &\{\, (k,a) \,|\, (k,a) \in X,\ \forall (k',b) \in Y : k' \neq k \,\} && (\subseteq X \text{ not joined with } Y) \\ \uplus\ &\{\, (k,b) \,|\, (k,b) \in Y,\ \forall (k',b) \in X : k' \neq k \,\} && (\subseteq Y \text{ not joined with } X) \end{aligned}$$

In other words, the monoid $\Uparrow_{\oplus}$ constructs a set of pairs whose unique key is the first pair element. In fact, any bag X can be converted to a set:

$$\text{distinct}(X) = \text{cMap}(\lambda(k,s).\{k\},\ \text{groupBy}(\text{cMap}(\lambda \text{x\_}.\ \{(x,x)\},\ X)))$$

Equi-joins and outer-joins between a bag X of type $\{(\kappa, \alpha)\}$ and a bag Y of type $\{(\kappa, \beta)\}$ over their first component of a type κ, are captured by $\mathrm{coGroup}(X, Y)$, which returns a bag of type $\{(\kappa, (\{\alpha\}, \{\beta\}))\}$:

$$\mathrm{coGroup}(X_1 \uplus X_2, Y_1 \uplus Y_2) = \mathrm{coGroup}(X_1, Y_1) \Uparrow_{\uplus \times \uplus} \mathrm{coGroup}(X_2, Y_2)$$

(plus more equations for cases with singleton and empty bags). Here, the product of two monoids, $\oplus \times \otimes$ is a monoid that, when applied to the two pairs (x_1, x_2) and (y_1, y_2), returns $(x_1 \oplus y_1, x_2 \otimes y_2)$. That is, the monoid $\Uparrow_{\uplus \times \uplus}$ merges two bags of type $\{(\kappa, (\{\alpha\}, \{\beta\}))\}$ by unioning together their $\{\alpha\}$ and $\{\beta\}$ values that correspond to the same key κ. For example, coGroup({(1,10),(2,20),(1,30)},{(1,100),(2,200),(3,300)}) returns {(1,({10,30},{100})),(2,({20},{200})),(3,({},{300}))}. Finally, aggregations are captured by $\mathrm{reduce}(\oplus, X)$, which aggregates a bag X using a commutative monoid $\oplus$. For example, $\mathrm{reduce}(+, \{2, 1, 1\}) = 4$.

Algebraic terms can be normalized using the following rule:

$$\mathrm{cMap}(f, \mathrm{cMap}(g, S)) \rightarrow \mathrm{cMap}(\lambda x.\, \mathrm{cMap}(f, g(x)), S)$$

which fuses two cascaded cMaps into a nested cMap, thus avoiding the construction of the intermediate bag. If we apply this transformation repeatedly, any algebraic term can be normalized to a tree of groupBy/coGroup operations connected via cMaps, while the root of the tree may be either a cMap or a reduce operation, if the query is a total aggregation.

A query q in our framework is transformed in such a way that it propagates the lineage to the query output, starting with the empty lineage $()$ at the sources and extended

with the join and group-by keys. Each value v returned by the transformed query is annotated with a lineage θ, in the form of a pair (θ, v). The lineage θ of the query result v is a tree of groupBy and coGroup keys that are used in deriving the result v. The lineage tree θ has the same shape as the groupBy/coGroup tree of the algebraic term of the query.

The transformation of an algebraic term to a term that propagates the join and group-by keys is done using rewrite rules, which make use of the fact that normalized algebraic terms are trees of groupBy/coGroup operations connected via cMaps. The first rule is:

$$\begin{array}{l}\mathrm{cMap}(f,\ \mathrm{groupBy}(X)) \\ \to\ \{\,(k',((k,\theta),v)) \,|\, ((k,\theta),s) \in \mathrm{groupBy}(\{\,((k,\theta),x) \,|\, (k,(\theta,x)) \in X\,\}), \\ \qquad\qquad\qquad\qquad (k',v) \in f(k,s)\,\}\end{array}$$

Here, X at the left-hand side is a bag of type $\{(k,x)\}$ that contains the groupBy key k, while the cMap result is a bag of type $\{(k',v)\}$ so that the new key k' is used for the subsequent groupBy or coGroup (the parent operation) in the algebraic tree. In the transformed term, X is lifted to a type $\{(k,(\theta,x))\}$, which includes the incoming lineage θ, while the cMap result is lifted to $\{(k',((k,\theta),v))\}$, which extends the lineage with the groupBy key k. The second rule handles joins:

$$\begin{array}{l}\mathrm{cMap}(f,\ \mathrm{coGroup}(X,Y)) \\ \to\ \{\,(k',((k,(\theta_x,\theta_y)),v)) \,|\, (k,(s_1,s_2)) \in \mathrm{coGroup}(X,Y), \\ \qquad\qquad\qquad\qquad (\theta_x,xs) \in \mathrm{groupBy}(s_1),\ (\theta_y,ys) \in \mathrm{groupBy}(s_2), \\ \qquad\qquad\qquad\qquad (k',v) \in f(k,(xs,ys))\,\}\end{array}$$

Here, both inputs X and Y have been lifted to $\{(k,(\theta_x,x))\}$ and $\{(k,(\theta_y,y))\}$, respectively, with possibly different lineages. Hence, these two lineages θ_x and θ_y must be paired with k to form the new lineage. Finally, if there is a $\mathrm{reduce}(\oplus, X)$ at the tree root, it is lifted to $\mathrm{reduce}(\Uparrow_\oplus, X)$, which aggregates data with the same lineage.

Transforming an algebraic term to propagate the group-by and join keys alone does not guarantee that the resulting term would be a homomorphism, but it is a required step. Although all algebraic operators used in MRQL are homomorphisms, their composition may not be. For instance, $\mathrm{cMap}(f, \mathrm{groupBy}(X))$ is not a homomorphism for certain functions f, because, in general, cMap does not distribute over $\Uparrow_\uplus$. Since both groupBy and coGroup are $\Uparrow_\oplus$ homomorphisms, our goal is to make sure that cMap is a $\Uparrow_\oplus$ homomorphism too, so that the entire query would be a $\Uparrow_\oplus$ homomorphism. More specifically, it can be proved by induction that $\mathrm{cMap}(f, X)$ is a $\Uparrow_\otimes$ homomorphism if X is a $\Uparrow_\oplus$ homomorphism and $f(k,s)$ is a $\Uparrow_\otimes$ homomorphism, if s is a $\oplus$ homomorphism. That is, a cMap is a homomorphism if its functional argument is a homomorphism too. This rule is part of a monoid inference system that we have developed, inspired by type inference systems used in programming languages, which infers the monoid of any algebraic term, if exists.

If a cMap term is not a $\Uparrow_\otimes$ homomorphism, our approach is to split cMap into two cMaps, one homomorphic and one not, and pull out and fuse all non-homomorphic cMaps at the root of the algebraic tree, thus splitting the query into two parts: the answer query and a homomorphism. Consider the term $\mathrm{cMap}(\lambda v.\, e,\ X)$. In our framework, we

find the largest subterms in the algebraic term e, namely $e_1, \ldots, e_n$, that are homomorphisms. This is accomplished by traversing the tree that represents the term e, starting from the root, and by checking if the node can be inferred to be a homomorphism. If it is, the node is replaced with a new variable. Thus, e is mapped to a term $f(e_1, \ldots, e_n)$, for some term f, and the terms $e_1, \ldots, e_n$ are replaced with variables when f is pulled outwards. That is, $\mathrm{cMap}(\lambda v.\, e,\, X)$ is split into two cMaps:

$$\mathrm{cMap}(\lambda(v, v_1, \ldots, v_n).\, f(v_1, \ldots, v_n),\, \mathrm{cMap}(\lambda v.\, \{(v, e_1, \ldots, e_n)\},\, X))$$

5 Performance Evaluation

We have implemented our incremental processing framework using Apache MRQL [26] on top of Apache Spark Streaming [36]. The Spark streaming engine monitors the file directories used as stream sources in an MRQL query, and when a new file is inserted in one of these directories or the modification time of a file changes, it triggers the MRQL query processor to process the new files, based on the state derived from the previous step, and creates a new state. The platform used for our evaluations is a small cluster of 9 nodes, built on the Chameleon cloud computing infrastructure, `www.chameleoncloud.org`. This cluster consists of nine m1.medium instances running Linux, each one with 4GB RAM and 2 VCPUs at 2.3GHz. For our experiments, we used Hadoop 2.6.0 (Yarn) and MRQL 0.9.6. The cluster front-end was used exclusively as a NameNode/ResourceManager, while the rest 8 compute nodes were used as DataNodes/NodeManagers. For our experiments, we used all the available 16 VCPUs of the compute nodes for Map-Reduce tasks.

We have experimentally validated the effectiveness of our methods using three queries: groupBy, join-groupBy, and a PageRank step. The groupBy and join-groupBy queries are expressed in MRQL as follows:

```
select (x,avg(y))
  from (x,y) in stream(binary,"S")
 group by x;
```

```
select (x,avg(z))
  from (x,y) in stream(binary,"S1"),
       (y,z) in stream(binary,"S2")
 group by x;
```

The PageRank algorithm computes the importance of the web pages in a web graph based exclusively on the topology of the graph. For a graph with vertices V and edges E, the PageRank P_i of a vertex $v_i \in V$ is calculated from the PageRank P_j of its incoming neighbors $v_j \in V$ with $(v_j, v_i) \in E$ using the rule $P_i = \sum_{(v_j,v_i)\in E} \frac{P_j}{|\{\, v_k \mid (v_j,v_k)\in E \,\}|}$. We represent the web graph as a bag of edges (i, j) from node i to node j. The following MRQL query computes one step of the PageRank algorithm:

```
select < id: a, rank: (1−factor)/graph_size+factor∗sum(in_rank) >
  from n in ( select < id: src, rank: 0.1, adjacent: dst >
                from (src,dst) in stream(binary,"S")
              group by src )
       a in n.adjacent,
       in_rank = n.rank/count(n.adjacent)
 group by a;
```

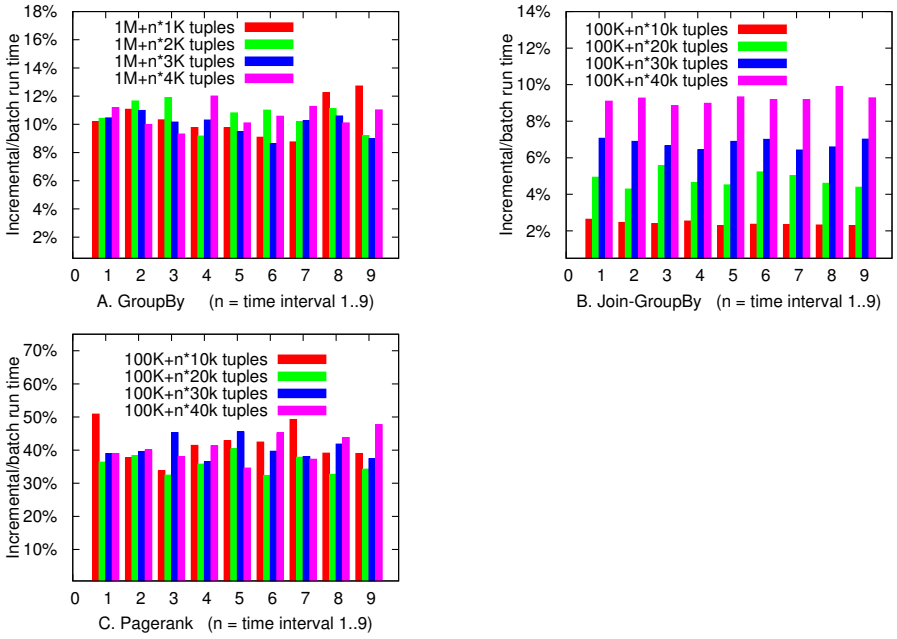

Fig. 2. Incremental Query Evaluation of GroupBy, Join-GroupBy, and PageRank step

where factor=0.85 is the dumping factor and graph_size is the number of graph nodes. The inner select-query converts the bag of edges from the input stream to a nested bag of type {<id: int, rank: double, adjacent: {int} >}, that is, to a bag of nodes where each node is associated with a unique id, a current PageRank value rank, and a bag of its outgoing neighbors adjacent. Then, the outer-select query binds the in_rank variable to one incoming PageRank contribution from node n to node a (= in MRQL is for binding a variable to a value, not ranging through a bag), and all these contributions are added in the sum to form the new rank of a. This query is over nested relations (it uses the node adjacent list) and is also a nested query.

The data streams used by the first two queries (groupBy and join-groupBy) consist of a large set of initial data, which is used to initialize the state, followed by a sequence of 9 equal-size batches of data (the increments). The groupBy initial dataset had size 1M tuples, while the two join-groupBy inputs had sizes 100K tuples. The experiments were repeated for increments of size 1K-4K tuples for groupBy and 10K-40K tuples for join-groupBy, always starting with a fresh state (constructed from the initial data only). For the PageRank step query, we generated random graphs using the R-MAT algorithm using the Kronecker graph generator parameters: a=0.30, b=0.25, c=0.25, and d=0.20. The initial graph used in our evaluations had size 10K nodes with 10K edges and the four different increments have sizes 100, 200, 300, and 400 edges, respectively. The performance results are shown in Figure 2. The x-axis represents the time points Δt when we get new batches of data in the stream. The 9 increments arrive at the time

points $1\Delta t$ through $9\Delta t$. The y-axis is the incremental execution time for each batch of data divided by the total processing time of the initial data (percentage).

We can see from Figure 2 that incremental processing can give an order of magnitude speed-up compared to batch processing that processes all the data every time the data changes. More importantly, the time to process each new batch of data remains nearly constant through time, even though the state grows with new data each time. The reason that the incremental processing time does not substantially increase through time is that we have carefully implemented state merging using Spark's coGroup operation (a hash-partitioned join), which is configured to partitioned the new batch of data only, but not the existing state, since the state has already been partitioned during the previous incremental step. That is, the new state created by coGroup is cached as a Spark RDD and is already partitioned by the join key, so that next time, when the state is merged with the new batch of data, it does not need to be partitioned again. Re-partitioning data is very expensive because it requires to suffle the data across the worker nodes. That way, the state merging is done very fast, because the new batch of data is expected to be substantially smaller than the state.

6 Related Work

Recently, there is a large number of Big Data stream processing systems, also known as distributed stream processing engines (DSPEs), that have emerged. The most popular one is Twitter's Storm [28], which is now part of the Apache ecosystem for Big Data analytics. It provides primitives for transforming streams based on a user-defined topology, consisting of spouts (stream sources) and bolts (which consume input streams and may emit new streams). Other popular DSPE platforms include Spark's D-Streams [36], Flink Streaming [16], Apache S4, and Apache Samza. Many of these systems build on the well-established research on incremental stream processing of relational data, based on sliding windows and incremental operators [4], which includes systems such as Aurora [1] and Telegraph [10]. In addition, there are several recent systems that have extended Map-Reduce with online processing capabilities, since Map-Reduce was originally designed for batch processing only. MapReduce Online [11] maintains state in memory for a chain of Map-Reduce jobs and reacts efficiently to additional input records. It also provides a memoization-aware scheduler to reduce communication across a cluster. Incoop [7] is a Hadoop-based incremental processing system with an incremental storage system that identifies the similarities between the input data of consecutive job runs and splits the input based on the similarity and file content. i^2MapReduce [37] implements incremental iterative Map-Reduce jobs using a store, MRB-Store, that maps input values to the reduce output values. This store is used for detecting delta changes and propagating these changes to the output. Google's Percolator [28] is a system based on BigTable for incrementally processing updates to a large data set. It updates an index incrementally as new documents are crawled. Microsoft Naiad [25] is a distributed framework for cyclic dataflow programs that facilitates iterative and incremental computations. It is based on differential dataflow computations, which allow incremental computations to have nested iterations. CBP [20] is a continuous bulk processing system on Hadoop that provides a stateful group-wise operator that

allows users to easily store and retrieve state during the reduce stage as new data inputs arrive. Their incremental computing PageRank implementation is able to cut running time in half. REX [24] handles iterative computations in which changes in the form of deltas are propagated across iterations and state is updated efficiently. In contrast to our automated approach, REX requires the programmer to explicitly specify how to process deltas, which are handled as first class objects. Trill [9] is a high throughput, low latency streaming query processor for temporal relational data, developed at Microsoft Research. The Reactive Aggregator [33], developed at IBM Research, is a new sliding-window streaming engine that performs many forms of sliding-window aggregation incrementally. Furthermore, the incremental query processing is related to the problem of incremental view maintenance, which has been extensively studied in the context of relational views (see [17] for a literature survey).

In programming languages, self-adjusting computation [2] refers to a technique for compiling batch programs into programs that can automatically respond to changes to their data. It requires the construction of a dependence graph at run-time so that when the computation data changes, the output can be updated by re-evaluating only the affected parts of the computation. In contrast to our work, which requires only the state to reside in memory, self-adjusting computation expects both the input and the output of a computation to reside in memory, which makes it inappropriate for unbounded data in a continuous stream. Furthermore, such dynamic methods impose a run-time storage and computation overhead by maintaining the dependence graph. The main idea in [2] is to manually annotate the parts of the input type that is changeable, and the system will derive an incremental program automatically based on these annotations. Each changeable value is wrapped by a mutator that includes a list of reader closures that need to be evaluated when the value changes. A read operation on a mutator inserts a new closure, while the write operation triggers the evaluation of the closures, which may cause writes to other mutators, etc, resulting to a cascade of closure execution triggered by changed data only. This technique has been extended to handle incremental list insertions (like our work), but it requires the rewriting of all list operations to work on incremental lists. Recently, there is a proof-of-concept implementation of this technique on map-reduce [3], but it was tested on a serial machine. It is doubtful that such dynamic techniques can be efficiently applied to a distributed environment, where a write in one compute node may cause a read in another node. Finally, there is recent work on static incrementalization based on derivatives [8]. In contrast to our work, it assumes that the merge function that combines the previous result with the result on the delta changes uses exactly the same delta changes, a restriction that excludes aggregations and group-bys.

7 Conclusion

We have presented a general framework for incremental distributed stream processing that translates SQL-like data analysis queries to incremental distributed stream processing programs. In contrast to other stream processing approaches, our framework derives incremental programs that return accurate results, rather than approximate answers. Our framework can also be used on a batch distributed system to process data larger than

the total available memory in the cluster, by processing these data incrementally, in batches that can fit in memory. As a future work, we are planning to improve our state transformations by using a map join, where the results of processing the new batch of data are broadcast to all workers and joined with their state partitions locally. We are also planning to store each state partition as a local key-value map at each worker to implement state updates more efficiently.

Acknowledgments: This work is supported in part by the National Science Foundation under the grant CCF-1117369. Our performance evaluations were performed at the Chameleon cloud computing infrastructure, `www.chameleoncloud.org`, supported by NSF.

References

1. D. J. Abadi, D. Carney, U. Cetintemel, et al. Aurora: A New Model and Architecture for Data Stream Management. In *VLDB Journal*, 12(2):120–139, 2003.
2. U. A. Acar, G. E. Blelloch, M. Blume, R. Harper, and K. Tangwongsan. An Experimental Analysis of Self-Adjusting Computation. In *ACM Trans. Prog. Lang. Sys., 32(1):3:153, 2009.*
3. U. A. Acar and Y. Chen. Streaming Big Data with Self-Adjusting Computation. In *Workshop on Data Driven Functional Programming (DDFP)*, 2013.
4. B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *Symposium on Principles of Database Systems (PODS)*, pages 1–16, 2002.
5. O. Benjelloun, A. D. Sarma, A. Halevy, and J. Widom. ULDBs: Databases with Uncertainty and Lineage. In *International Conference on Very Large Data Bases (VLDB)*, pages 953–964, 2006.
6. D. Bhagwat, L. Chiticariu, W. C. Tan, and G. Vijayvargiya. An Annotation Management System for Relational Databases. In *International Conference on Very Large Data Bases (VLDB)*, pages 900–911, 2004.
7. P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: Mapreduce for Incremental Computations. In *ACM Symposium on Cloud Computing (SoCC)*, 2011.
8. Y. Cai, P. G. Giarrusso, T. Rendel, and K. Ostermann. A Theory of Changes for Higher-Order Languages. Incrementalizing λ-Calculi by Static Differentiation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 145-155, 2014.
9. B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, J. Wernsing. Trill: A High-Performance Incremental Query Processor for Diverse Analytics. In *International Conference on Very Large Data Bases (VLDB)*, pages 401–412, 2014.
10. S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous Data flow Processing for an Uncertain World. In *Conference on Innovative Data System Research (CIDR)*, 2003.
11. T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce Online. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 10(4), 2010.
12. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Symposium on Operating System Design and Implementation (OSDI)*, 2004.
13. L. Fegaras, C. Li, U. Gupta, and J. J. Philip. XML Query Optimization in Map-Reduce. In *International Workshop on the Web and Databases (WebDB)*, 2011.

14. L. Fegaras, C. Li, and U. Gupta. An Optimization Framework for Map-Reduce Queries. In *International Conference on Extending Database Technology (EDBT)*, pages 26–37, 2012.
15. L. Fegaras and D. Maier. Optimizing Object Queries Using an Effective Calculus. In *ACM Transactions on Database Systems (TODS)*, 25(4):457–516, 2000.
16. Apache Flink. `http://flink.apache.org/`.
17. A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. In *IEEE Bulletin on Data Engineering*, 18(2):145–157, 1995.
18. Apache Hadoop. `http://hadoop.apache.org/`.
19. Apache Hive. `http://hive.apache.org/`.
20. D. Logothetis, C. Olston, B. Reed, K.C. Webb, and K. Yocum. Stateful Bulk Processing for Incremental Analytics. In *ACM Symposium on Cloud Computing (SoCC)*, 2010.
21. Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. In *Proceedings of the VLDB Endowment*, 5(8):716-727, 2012.
22. G. Malewicz, M. H. Austern, A. J.C Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a System for Large-Scale Graph Processing. In *ACM symposium on Principles of Distributed Computing (PODC)*, 2009.
23. F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential Dataflow. In *Conference on Innovative Data System Research (CIDR)*, 2013.
24. S. R. Mihaylov, Z. G. Ives, and S. Guha. REX: Recursive, Delta-Based Data-Centric Computation. In *Proceedings of the VLDB Endowment*, 5(11):1280-1291, 2012.
25. D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a Timely Dataflow System. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2013.
26. Apache MRQL (incubating). `http://mrql.incubator.apache.org/`.
27. C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-Foreign Language for Data Processing. In *ACM SIGMOD International Conference on Management of Data*, pages 1099-1110, 2008.
28. D. Peng and F. Dabek. Large-scale Incremental Processing using Distributed Transactions and Notifications. In *Symposium on Operating System Design and Implementation (OSDI)*, 2010.
29. R. Power and J. Li. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *Symposium on Operating System Design and Implementation (OSDI)*, 2010.
30. A. Shinnar, D. Cunningham, B. Herta, and V. Saraswat. M3R: Increased Performance for In-Memory Hadoop Jobs. In *Proceedings of the VLDB Endowment*, 5(12):1736-1747, 2012.
31. Apache Spark. `http://spark.apache.org/`.
32. Apache Storm: A System for Processing Streaming Data in Real Time. `http://hortonworks.com/hadoop/storm/`.
33. K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu. General Incremental Sliding-Window Aggregation. In *Proceedings of the VLDB Endowment*, 8(7):702-713, 2015.
34. L. G. Valiant. A Bridging Model for Parallel Computation. In *Communications of the ACM (CACM)*, 33(8):103-111, August 1990.
35. M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.
36. M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Symposium on Operating Systems Principles (SOSP)*, 2013.
37. Y. Zhang, S. Chen, Q. Wang, and G. Yu. i^2 MapReduce: Incremental MapReduce for Mining Evolving Big Data. In *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 27(7):1906-1919, 2015.