Incremental Query Processing on Big Data Streams

Leonidas Fegaras

Abstract—This paper addresses online query processing for large-scale, incremental data analysis on a distributed stream processing engine (DSPE). Our goal is to convert any SQL-like query to an incremental DSPE program automatically. In contrast to other approaches, we derive incremental programs that return accurate results, not approximate answers, by retaining a minimal state during the query evaluation lifetime and by using a novel incremental evaluation technique, which, at each time interval, returns an accurate snapshot answer that depends on the current state and the latest batches of data. Our methods can handle many forms of queries on nested data collections, including iterative and nested queries, group-by with aggregation, and equi-joins. Finally, we report on a prototype implementation of our framework, called MRQL Streaming, running on top of Spark and we experimentally validate the effectiveness of our methods.

Index Terms—Incremental data processing, distributed stream processing, big data, MRQL, spark

1 Introduction

Big Data analysis has become increasingly important in recent years as large volumes of data are being generated, analyzed, and used at an unprecedented scale and rate. Data analysis tools that process these data are typically batch programs that need to work on the complete datasets, thus repeating the computation on existing data when new data are added to these datasets. Consequently, batch processing may be prohibitively expensive for Big Data that change frequently. For example, the Web is evolving at an enormous rate with new Web pages, content, and links added daily. Web graph analysis tools, such as PageRank, which are used extensively by search engines, need to recompute their Web graph measures very frequently since they become outdated very fast. There is a recent interest in incremental Big 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. Incremental data processing can generally achieve better performance and may require less memory than batch processing for many data analysis tasks. It can also be used for analyzing Big Data incrementally, in batches that can fit in memory. Consequently, incremental data processing can also be useful to stream-based applications that need to process continuous streams of data in real-time with low latency, which is not feasible with existing batch analysis tools. For example, the Map-Reduce framework [12], which was designed for batch processing, is ill-suited for certain Big Data workloads, such as real-time analytics, continuous queries, and iterative algorithms. New alternative frameworks have emerged that address the inherent limitations of the Map-Reduce model and perform better for a wider spectrum of workloads. Currently, among them, the most promising frameworks that seem to be good alternatives to Map-Reduce while addressing its drawbacks are Google's Pregel [24], Apache Spark [33], and Apache Flink [18], which are in-memory distributed computing systems. There are also quite a few emerging distributed stream processing engines (DSPEs) that realize online, low-latency data processing with a series of batch computations at small time intervals, using a continuous streaming system that processes data as they arrive and emits continuous results. To cope with blocking operations and unbounded memory requirements, some of these systems build on the well-established research on data streaming based on sliding windows and incremental operators [2], which includes systems such as Aurora [1] and Telegraph [9], often yielding approximate answers, rather than accurate results. Currently, among these DSPEs, the most popular platforms are Twitter's (now Apache) Storm [30], Spark's D-Streams [38], Flink Streaming [18], Apache S4 [35], and Apache Samza [32].

This paper addresses online processing for large-scale, incremental computations on a distributed processing platform. Our goal is to convert any batch data analysis program to an incremental distributed stream processing (DSP) program automatically, without requiring the user to modify this program. We are interested in deriving DSP programs that produce accurate results, rather than approximate answers. To accomplish this task, we need to carefully analyze a program to identify those parts that can be used to process the incremental batches of data and those parts that can be used to merge the current results with the new results of processing the incremental batches. Such analysis is hard to attain for programs written in an algorithmic programming language, but can become more tractable if it is performed on declarative queries. Fortunately, most programmers already prefer to use a higher-level query language, such as Apache Hive [21], to code their distributed data analysis applications, instead of coding them directly in an algorithmic language,

• *The author is with the University of Texas at Arlington, Arlington, TX 76019. E-mail: fegaras@cse.uta.edu.*

Manuscript received 6 Mar. 2016; revised 1 July 2016; accepted 8 Aug. 2016. Date of publication 17 Aug. 2016; date of current version 3 Oct. 2016.
Recommended for acceptance by C. Chan.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TKDE.2016.2601103

1041-4347 © 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

such as Java. For instance, Hive is used for over 90 percent of Facebook Map-Reduce jobs. There are many reasons why programmers prefer query languages. First, it is hard to develop, optimize, and maintain non-trivial applications coded in a general-purpose programming language. Second, given the multitude of the new emerging distributed processing frameworks, such as Spark and Flink, it is hard to tell which one of them will prevail in the near future. Data intensive applications that have been coded in one of these paradigms may have to be rewritten as technologies evolve. Hence, it is desirable to express these applications in a declarative query language that is independent of the underlying distributed platform. Furthermore, the execution of such queries can benefit from cost-based query optimization and automatic parallelism, thus relieving the application developers from the intricacies of Big Data analytics and distributed computing. Therefore, our goal is to convert batch SQL-like queries to incremental DSP programs. Since we are interested in deriving incremental programs that return accurate results, not approximate answers, our focus is in retaining a minimal state during the query evaluation lifetime and across iterations, and in deriving an accurate snapshot answer that depends on the current state and the latest batches of data.

We have developed general, sound methods to transform batch queries to incremental queries. 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 technique is known as lineage tracking ([3], [4], [11]). That way, the values in the query output are grouped by a key combination, which corresponds the join and 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 current data can be combined with the computations on the new data by joining the data on these keys. This approach requires that we can combine computations on data that have the same lineage to derive incremental results. In our framework, this task is accomplished by transforming a query to a 'monoid homomorphism' by extracting the non-homomorphic parts of the query outwards, using algebraic transformation rules, and combining them to form an answer function, which is detached from the rest of the query. We have implemented our incremental processing framework using Apache MRQL [28] on top of Apache Spark Streaming [38], 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 SQL, XQuery, Jaql, and Hive.

The contribution of this work can be summarized as follows:

- We present a general automated method to convert most distributed data-analysis queries to incremental stream processing programs.
- Our methods can handle many forms of queries, including iterative and nested queries, group-by with aggregation, and joins on one-to-many relationships.
- We report on a prototype implementation of our framework using Apache MRQL running on top of Apache Spark Streaming. We show the effectiveness of our method through experiments on four queries: groupBy, join-groupBy, k-means clustering, and PageRank.

The rest of this paper is organized as follows. Section 2 illustrates our approach through examples. Section 3 compares our work with related work. Section 4 describes our earlier work on MRQL query processing. Section 5 defines the algebraic operators used in the MRQL algebra. Section 6 presents transformation rules for converting algebraic terms to a normal form that is easier to analyze. Section 7 describes an inference algorithm that statically infers the merge function that merges the current results with the results of processing the new data. Section 8 describes our method for transforming algebraic terms to propagate all keys used in joins and group-bys to the query output. Section 9 describes an algorithm that pulls the non-homomorphic parts of a query outwards, deriving a homomorphism. Section 10 gives some implementation details. Section 11 presents experiments that evaluate the performance of our incremental query processing techniques for four queries.

2 Highlights of Our Approach

A dataset in our framework is a bag (multiset) that consists of arbitrarily complex values, which may contain nested bags and hierarchical data, such as XML and JSON fragments. We are considering continuous queries over a number of streaming data sources, S_i, for $0 < i \leq n$. A data stream S_i in our framework consists of an initial dataset, followed by a continuous stream of incremental batches ΔS_i that arrive at regular time intervals Δt. In addition to streaming data, there may be other input data sources that remain invariant through time. A streaming query in our framework can be expressed as $q(\overline{S})$, where an $S_i \in \overline{S}$ is a streaming data source. Incremental stream processing is feasible when we can derive the query results at time $t + \Delta t$ by simply combining the query results at time t (i.e., the current results) with the results of processing the incremental batches ΔS_i only, rather than the entire streams $S_i \uplus \Delta S_i$, where $\uplus$ is the additive (bag) union. This derivation is possible if $q(\overline{S \uplus \Delta S})$ can be expressed in terms of $q(\overline{S})$ (the current result) and $q(\overline{\Delta S})$ (the incremental query result). In abstract algebra, a function f over a bag is a monoid homomorphism if $f(X \uplus Y) = f(X) \otimes f(Y)$ for some monoid $\otimes$ (an associative function with a zero element $\otimes_z$). Consequently, in our framework, a query $q(\overline{S})$ is incremental if it is a homomorphism[1] over $\overline{S}$.

Unfortunately, not all queries are homomorphisms. Consider, for example, the following query (all queries are expressed in MRQL):

$$q(S) = \textbf{select}(\text{x.A}, \textbf{avg}(\text{x.B}))\ \textbf{from}\ \text{x}\ \textbf{in}\ S\ \textbf{group by}\ \text{x.A}.$$

1. We use the term *homomorphism* throughout the paper as an abbreviation of *monoid homomorphism*.

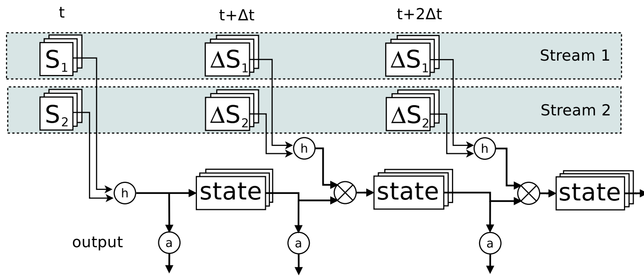

Fig. 1. Incremental query processing.

For this query to be a homomorphism, there must exist a merge function $\otimes$ such that $q(S_1 \uplus S_2) = q(S_1) \otimes q(S_2)$. This function $\otimes$ should be able to combine the average values $\mathbf{avg}(x.B)$ from $q(S_1)$ and $q(S_2)$ that correspond to the same group-by key x.A, which is impossible without knowing the sizes of S_1 and S_2. Another example is counting the number of distinct elements in a stream. This query too is not a homomorphism because when we count the distinct elements in a bag S we loose the information about which elements are contained in S, and therefore, counting the distinct elements of $S_1 \uplus S_2$ cannot be derived by combining the counts of the distinct elements in S_1 and S_2 separately, since S_1 and S_2 may have common elements. For such non-homomorphic queries, our approach is to break q into two functions a and h, so that $q(\overline{S}) = a(h(\overline{S}))$ and h is a homomorphism. For example, the avg(x.B) in the first query $q(S)$ is decomposed into the pair (sum(x.B), count(x.B)), since both sum and count are homomorphisms. Then, $q(S) = a(h(S))$, where

$$\begin{aligned} h(S) = &\ \textbf{select}\ (\text{x.A}, (\textbf{sum}(\text{x.B}), \textbf{count}(\text{x.B}))) \\ &\ \textbf{from}\ \text{x}\ \textbf{in}\ S\ \textbf{group by}\ \text{x.A} \\ a(X) = &\ \textbf{select}\ (\text{k}, \text{s/c})\ \textbf{from}\ (\text{k}, (\text{s}, \text{c}))\ \textbf{in}\ X. \end{aligned}$$

The other query that counts the number of distinct elements, can be broken into the query h that returns the list of distinct elements (a homomorphism), followed by the answer query a that counts these elements. Ideally, we would like most of the computation in q to be done in h, leaving only some computationally inexpensive data mappings to the answer function a. Note that, the obvious solution in which a is equal to q and h is the union of data sources, is also the worst-case scenario that we try to avoid, since it basically requires computing the new result from the entire input, $\overline{S \uplus \Delta S}$. On the other hand, in the special case when a is the identity function and $\otimes$ is equal to $\uplus$, the output at each time interval can be simply taken to be only $h(\overline{\Delta S})$, which is the output we would expect to get from a fixed window system (i.e., new batches of output from new batches of input).

If we split q into a homomorphism h and an answer function a, then we can calculate h incrementally by storing its results into a state and then using the current state to calculate the next h result. Initially, $\text{state} = \otimes_z$ or, if there are initial stream data, $\text{state} = h(\overline{S})$. At each time interval Δt, the new state is $h(\overline{S \uplus \Delta S}) = h(\overline{S}) \otimes h(\overline{\Delta S})$, where $h(\overline{S})$ is the current state, and the query answer is $a(\text{state})$. Hence, the incremental program is

$$\begin{aligned} &\text{state} \leftarrow \text{state} \otimes h(\overline{\Delta S}) \\ &\textbf{return}\ a(\text{state}). \end{aligned}$$

In our system implementation on Spark, the state and the invariant data sources are stored in memory as Distributed DataSets (Spark's RDDs [37]) and are distributed across the worker nodes. However, the streaming data sources are implemented as Discretized Streams (Spark's D-Streams [38]), which are also distributed.

Our framework works best for queries whose output is considerably smaller than their input, such as for data analysis queries that aggregate data. Such queries require a smaller state and impose less processing overhead on $\otimes$.

Fig. 1 shows the evaluation of an incremental query $q(S_1, S_2) = a(h(S_1, S_2))$ over two streaming data sources, where $h(S_1 \uplus \Delta S_1, S_2 \uplus \Delta S_2) = h(S_1, S_2) \otimes h(\Delta S_1, \Delta S_2)$.

Our query processing system performs the following tasks:

1) It pulls all non-homomorphic parts of a query q out from the query, using algebraic transformations.
2) It collects these non-homomorphic parts into an answer function a, leaving an algebraic homomorphic term h, such that $q(\overline{S}) = a(h(\overline{S}))$.
3) From the homomorphic algebraic term $h(\overline{S})$, our system derives a merge function $\otimes$, such that $h(\overline{S \uplus \Delta S}) = h(\overline{S}) \otimes h(\overline{\Delta S})$.

These tasks are, in general, hard to attain for a program expressed in an algorithmic programming language, but become more tractable if they are performed on higher-order operations, such as the MRQL query algebra ([15], [16]). 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 iterative queries, complex nested queries with any form and any number of nesting levels, general group-bys with aggregations, and general one-to-one and one-to-many equi-joins. Our methods though cannot handle non-equi-joins and many-to-many equi-joins.

Example. For example, consider the following query:

$$\begin{aligned} q(S_1, S_2) = &\ \textbf{select}\ (\text{x.A}, \textbf{avg}(\text{y.D})) \\ &\ \textbf{from}\ \text{x}\ \textbf{in}\ S_1, \text{y}\ \textbf{in}\ S_2 \\ &\ \textbf{where}\ \text{x.B} = \text{y.C} \\ &\ \textbf{group by}\ \text{x.A}. \end{aligned}$$

where S_1 and S_2 in $q(S_1, S_2)$ are streaming data sources. Unfortunately, $q(S_1, S_2)$ is not a homomorphism over 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 $q(\Delta S_1, \Delta S_2)$ exclusively. The intrinsic reason is that there is no lineage in the query output that links a pair in the query result to the join key (x.B or y.C) that contributed to this pair. Consequently, there is no way to tell how the new data batches ΔS_1 and ΔS_2 will contribute to the previous query results if we do not know how these results are related to the inputs S_1 and S_2. To compensate, we need to establish links between the query results and the data sources that were used to form their values. This process is called *lineage tracking* and has been used for consistent representation

of uncertain data [3] and for propagating annotations in relational queries [4]. In our case, this lineage tracking can be accomplished by propagating all keys used in joins and group-bys along with the values associated with the keys, so that, for each combination of keys, we have one group of result values. For our query, the lineage tracking is done by including the join key $\mathsf{x.B}$ as a group-by key. That is, in the following transformed query:

$$\begin{aligned} h(S_1, S_2) = &\ \textbf{select}\ ((\mathsf{x.A}, \mathsf{x.B}), (\mathbf{sum}(\mathsf{y.D}), \mathbf{count}(\mathsf{y.D}))) \\ &\ \textbf{from}\ \mathsf{x}\ \textbf{in}\ S_1, \mathsf{y}\ \textbf{in}\ S_2 \\ &\ \textbf{where}\ \mathsf{x.B} = \mathsf{y.C} \\ &\ \textbf{group by}\ \mathsf{x.A}, \mathsf{x.B}. \end{aligned}$$

the join key is propagated to the output values so that the avg components, sum and count, are aggregations over groups that correspond to unique combinations of $\mathsf{x.A}$ and $\mathsf{x.B}$. 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, and order-by operations will be transformed to a query that injects the join, group-by, and order-by keys to the output so that each output value is annotated with a combination of N keys. The answer query a that gives the final result for $q(S_1, S_2) = a(h(S_1, S_2))$ is

$$\begin{aligned} a(X) = &\ \textbf{select}(\mathsf{k}, \mathbf{sum}(\mathsf{s})/\mathbf{sum}(\mathsf{c})) \\ &\ \textbf{from}((\mathsf{k}, \mathsf{j}), (\mathsf{s}, \mathsf{c}))\ \textbf{in}\ X \\ &\ \textbf{group by}\ \mathsf{k}, \end{aligned}$$

that is, $a(X)$ removes the lineage j from X but also groups the result by the group-by key again and calculates the final avg values by adding the partial sums s and the partial counts c. The merge function $\otimes$ for the homomorphism h is a full outer join on the lineage key θ that aggregates the matches. It is specified as

$$\begin{aligned} X \otimes Y = &\ \textbf{select}\ (\theta, (\mathsf{sx} + \mathsf{sy}, \mathsf{cx} + \mathsf{cy})) \\ &\ \textbf{from}\ (\theta, (\mathsf{sx}, \mathsf{cx}))\ \textbf{in}\ X, \\ &\ \qquad (\theta, (\mathsf{sy}, \mathsf{cy}))\ \textbf{in}\ Y \\ &\ \textbf{union select}(\theta, (\mathsf{sy}, \mathsf{cy}))\ \textbf{from}(\theta, (\mathsf{sy}, \mathsf{cy}))\ \textbf{in}\ Y \\ &\ \qquad \textbf{where}\ \theta\ \textbf{not in}\ \mathsf{keys}(X) \\ &\ \textbf{union select}(\theta, (\mathsf{sx}, \mathsf{cx}))\ \textbf{from}(\theta, (\mathsf{sx}, \mathsf{cx}))\ \textbf{in}\ X \\ &\ \qquad \textbf{where}\ \theta\ \textbf{not in}\ \mathsf{keys}(Y), \end{aligned}$$

where θ is the lineage (k, j) and $\mathsf{keys}(X)$, which is equal to **select** θ **from** $(\theta, (\mathsf{sx}, \mathsf{cx}))$ **in** X, returns the lineage keys of X. The first part of this query is an equi-join over θ, while the other two give the part of Y not joined with X and the part of X not joined with Y. The incremental execution of this query is shown in Fig. 1, which replaces the current state with $\text{state} \otimes h(\Delta S_1, \Delta S_2)$ and returns the snapshot query answer $a(\text{state})$ at each time interval Δt.

Merging States. The effectiveness of our incremental processing depends on the efficient implementation of the state transformation that merges the previous state with the results of processing the new data, $\text{state} \otimes h(\overline{\Delta S})$. The merge operation $X \otimes Y$ in the previous example can be implemented efficiently as a partitioned join. On Spark, for example, both the state and the new results are kept in the distributed memory of the worker nodes (as RDDs), while the full outer join can be implemented as a coGroup operation, which shuffles the join input data across the worker nodes using hash partitioning. However, when the new state is created by coGroup, it is already partitioned by the join key and is ready to be used for the next call to coGroup to handle the next batch of data. Consequently, only the results of processing the new data, which are typically smaller than the state, have to be shuffled among the worker nodes before coGroup. Although other queries may require different merge functions, the correlation between the previous state and the results of processing the new data is always based on the lineage keys. Therefore, regardless of the query, we can keep the state partitioned on the lineage keys by simply leaving the partitions of the new state at the place they were generated. However, the new results have to be partitioned and shuffled across the working nodes to be combined with the current state.

Our approach is based on the assumption that, since the state is kept in the distributed memory, there will be very little overhead in replacing the current state with a new state, as long as it is not repartitioned. But this assumption may not be valid if the input data or the current state is larger than the available distributed memory. Indeed, one of our goals is to be able to process data larger than the available memory, by processing these data incrementally, in batches that can fit in memory. However, if the state is stored on disk, replacing it with a new state may become prohibitively expensive. Fortunately, there is no need for using a distributed key-value store to update the state in-place. There is, in fact, a simpler solution: since the state is kept partitioned on the lineage keys, every worker node may store its assigned state partition on its local disk, as a binary file (such as, an HDFS Sequence file) sorted by the lineage key. Then, after the results of processing the new data are distributed to worker nodes (using uniform hashing based on the lineage key), each worker node will sort its new data partition by the lineage key and will merge it with its old state (stored on its local disk) to create a new state.

Iteration. Given that many important data analysis and mining algorithms, such as PageRank and k-means clustering, require iteration, we have extended our methods to include iteration, so that these algorithms too can become incremental. An iteration can take the following general form:

$$\begin{aligned} &X \leftarrow g(\overline{S}) \\ &\textbf{for}\ i \leftarrow 1 \ldots n \\ &\qquad X \ \leftarrow \ f(X, \overline{S}), \end{aligned}$$

where X is the fixpoint of the iteration that has initial value $g(\overline{S})$. Note that X is not necessarily a bag. For example, non-negative Matrix Factorization splits a matrix S into two non-negative matrices W and H. In that case, the fixpoint X is the pair (W, H), which is refined at every loop step.

An exact incremental solution of the above iteration is only possible if f is a homomorphism; a strict requirement that excludes many important iterative algorithms, such as

PageRank and k-means clustering. Instead, our approach is to use an approximate solution that works well for iterative queries that improve a solution at each iteration step. As before, we split f into a homomorphism h, for some monoid $\otimes$, and an answer function a, so that $f(X, \overline{S}) = a(h(X, \overline{S}))$ and $h(X, \overline{S \uplus \Delta S}) = h(X, \overline{S}) \otimes h(X, \overline{\Delta S})$. Let T be the current state on input $\overline{S}$ and let T' be the new state on input $\overline{S \uplus \Delta S}$. An ideal solution would be to derive the new state T' incrementally, as $T \otimes \Delta T$, so that the state increment ΔT depends on ΔS but not on T. But this independence from T is not always possible for many iterative algorithms. In PageRank, for example, we cannot just calculate the Page-Ranks of the new data and merge them with the PageRank of the existing data because the new PageRank contributions may have to propagate to the rest of the graph. On the other hand, it would be too expensive to correlate the entire state T with ΔT at each iteration step. Our compromise is to partially correlate T with ΔT at each iteration step, and then fully merge ΔT with T after the iteration. This partial correlation is accomplished with the operation $\widehat{\otimes}$, called *diffusion*, which is directly derived from the merge function $\otimes$. That is, while $T \otimes \Delta T$ returns a new complete state by merging T with ΔT, the operation $T \widehat{\otimes} \Delta T$ returns a new ΔT that contains only the part of $T \otimes \Delta T$ that is different from T. Although $T \widehat{\otimes} \Delta T$ is larger than ΔT, it is often far smaller than $T \otimes \Delta T$. Based on this analysis, we are using the following approximation algorithm to compute the iteration incrementally:

Algorithm 1. Incremental Computation of Iteration:

$$
\begin{aligned}
&\Delta X \leftarrow g(\overline{\Delta S}) \\
&\textbf{for } i \leftarrow 1 \dots n \\
&\quad \Delta T \leftarrow T \ \widehat{\otimes}\ h(\Delta X, \overline{\Delta S}) \\
&\quad \Delta X \leftarrow a(\Delta T) \\
&T \leftarrow T \otimes \Delta T \\
&\textbf{return } a(T).
\end{aligned}
$$

The diffusion operator $\widehat{\otimes}$ must satisfy the property

$$T \otimes (T \widehat{\otimes} \Delta T) = T \otimes (T \otimes \Delta T),$$

that is, when $T \widehat{\otimes} \Delta T$ and $T \otimes \Delta T$ are merged with T, they give the same answer, because $T \widehat{\otimes} \Delta T$ discards the parts of T that are not joined with ΔT, but these parts are embedded back in the answer when they are merged with T.

For example, PageRank is an iterative algorithm that calculates the PageRank of a graph node as the sum of the incoming PageRank contributions from its neighbors, while its own PageRank is equally distributed to its outgoing neighbors. The PageRank query expressed in MRQL is a simple self-join on the graph, which is optimized into a single group-by operation [16]. For PageRank, the state-merging operator $\otimes$ is a full outer join that incorporates the new PageRank contributions to the existing PageRanks. The diffusion operation $T \ \widehat{\otimes} \Delta T$, on the other hand, propagates the new PageRank contributions from ΔT to T, that is, from the nodes in ΔT to their immediate outgoing neighbors in both T and ΔT. Thus, at each iteration step, ΔT is expanded to $T \ \widehat{\otimes} \Delta T$, growing one level at a time, in a way similar to breadth-first-search. Consequently, our approximation algorithm propagates PageRanks up to depth n, starting from the ΔT nodes, and only the affected nodes will be part of the new ΔT. The $\widehat{\otimes}$ operation is similar to $\otimes$, but with a right-outer join instead of a full outer join. That way, the data from T that are not joined with ΔT will not appear in the new ΔT.

To exemplify our incremental approach for iterative queries more concretely, we consider the following MRQL query that implements the k-means clustering algorithm by repeatedly deriving k new centroids from the old ones

```
repeat centroids = ...
step select < X : avg(s.X), Y : avg(s.Y) >
    from s in S1
    group by k : (select c from c in centroids
                  order by distance (c, s))[0],
```

where S_1 is the input stream of points on the X-Y plane, centroids is the current set of centroids (k cluster centers), and distance is a function that calculates the distance between two points. The initial value of centroids (the ... value) can be a bag of k random points. The inner select-query in the group-by part assigns the closest centroid to a point s (where [0]returns the first tuple of an ordered list). The outer select-query in the repeat step clusters the data points by their closest centroid, and, for each cluster, a new centroid is calculated from the average values of its points.

Based on our analysis of iterative queries, the repeat step (the outer select query) of the k-means clustering query is split into a homomorphism h and an answer function a. The homomorphism $h(X, S_1)$ is

```
select (k, ((sum(s.X), count(s.X)),
            (sum(s.Y), count(s.Y))))
from s in S1
group by k : (select c from c in X
              order by distance(c, s))[0].
```

That is, in addition to the lineage k (a previous centroid), it returns a new centroid $< X : \textbf{avg}(s.X), Y : \textbf{avg}(s.Y) >$, decomposed into a sum-count pair. The merge function $\otimes$ of h is a full outer join, similar to the one used by the join-groupBy example. The diffusion operation $X \widehat{\otimes} Y$ though is a right-outer join that discards those state data that do not join with the new data. That is, it is equal to the $X \otimes Y$ query without the last union

```
X ⊗̂ Y = select(k, (sx + sy, cx + cy))
        from(k, (sx, cx)) in X,
            (k, (sy, cy)) in Y
      union select(k, (sy, cy)) from(k, (sy, cy)) in Y
            where k not in keys(X).
```

The answer query that returns the result (the centroids) is

$$a(\textbf{state}) = \textsf{select} < \mathrm{X}:\mathrm{sx}/\mathrm{cx},\ \mathrm{Y}:\mathrm{sy}/\mathrm{cy} > \textsf{from}(\mathsf{k},((\mathrm{sx},\mathrm{cx}),(\mathrm{sy},\mathrm{cy})))\ \textbf{in state}.$$

It does not need a group-by since k is the only lineage key.

3 Related Work

New frameworks in distributed Big Data analytics have become essential tools to large-scale machine learning and scientific discoveries. Among these frameworks, the Map-Reduce programming model [12] has emerged as a generic, scalable, and cost effective solution for Big Data processing on clusters of commodity hardware. One of the major drawbacks of the Map-Reduce model is that, to simplify reliability and fault tolerance, it does not preserve data in memory across consecutive Map-Reduce jobs, which imposes a high overhead on complex workflows and graph algorithms, such as Page-Rank, which require repetitive Map-Reduce jobs. Recent systems for Big Data analysis use distributed memory for inter-node communication, such as the Main Memory Map-Reduce (M3R [31]), Apache Spark [33], Apache Flink [18], and distributed GraphLab [23].

Although the Map-Reduce framework was originally designed for batch processing, there are several recent systems that have extended Map-Reduce with online processing capabilities. Some of these systems build on the well-established research on data streaming based on sliding windows and incremental operators [2], which includes systems such as Aurora [1] and Telegraph [9]. MapReduce Online [10] maintains state in memory for a chain of MapReduce jobs and reacts efficiently to additional input records. It also provides a memoization-aware scheduler to reduce communication across a cluster. Incoop [5] 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 [39] 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 [30] 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 [27] 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 [22] is a continuous bulk processing system on Hadoop that provides a stateful groupwise 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 [26] handles iterative computations in which changes in the form of deltas are propagated across iterations and a 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 [8] is a high throughput, low latency streaming query processor for temporal relational data, developed at Microsoft Research. The Reactive Aggregator [36], developed at IBM Research, is a new sliding-window streaming engine that performs many forms of sliding-window aggregation incrementally. Although these systems are general enough to capture most incremental programs, none of them can convert batch programs to incremental ones automatically. In addition to these general data analysis engines, there are many data analysis algorithms that have been implemented incrementally, such as incremental PageRank [13].

4 Earlier Work: MRQL

Apache MRQL [28] is a query processing and optimization system for large-scale, distributed data analysis. MRQL was originally developed by the author's research group at UTA ([15], [16]), but is now an Apache incubating project with many developers and users worldwide. The MRQL language is an SQL-like query language for large-scale data analysis on computer clusters. The MRQL query processing system can evaluate MRQL queries in four modes: In Map-Reduce mode using Apache Hadoop [19], in BSP mode (Bulk Synchronous Parallel model) using Apache Hama [20], in Spark mode using Apache Spark [33], and in Flink mode using Apache Flink [18]. The MRQL query language is powerful enough to express most common data analysis tasks over many forms of raw in-situ data, such as XML and JSON documents, binary files, and CSV documents. The design of MRQL has been influenced by XQuery and ODMG OQL, although it uses SQL-like syntax. In fact, when restricted to XML, MRQL is as powerful as XQuery. MRQL is more powerful than other current high-level Map-Reduce languages, such as Hive [21] and PigLatin [29], 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. For example, the PageRank query on raw DBLP XML data, which ranks authors based on the number of citations they have received from other authors, is 16 lines long [15] and can be executed on all the supported platforms as is, without changing the query.

A recent extension to MRQL, called *MRQL Streaming*, supports the processing of continuous MRQL queries over streams of batch data (that is, data that come in continuous large batches). Before the incremental MRQL work presented in this paper, MRQL Streaming supported traditional window-based streaming based on a fixed window during a specified time interval. Any batch MRQL query can be converted to a window-based streaming query by replacing at least one 'source' call in the query, which accesses a data source, to a 'stream' call (with exactly the same arguments). If the stream data source is a directory of files, the MRQL Streaming engine will first process all the existing files in the directory and then will check this directory periodically for new files. When new files are inserted in the directory, it will process the new batch of data using distributed query processing. A query may work on multiple stream sources and multiple batch sources. Multiple stream sources are not implicitly synchronized in a query; instead, if such synchronization is desired, it can be done by joining the streams on their

timestamps. If there is at least one stream source, the query becomes continuous (it never stops). The output of a continuous query is stored in a file directory, where each file contains the results of processing each batch of streaming data. Currently, MRQL Streaming works on Spark Streaming only but there are current efforts to add support for Storm and Flink Streaming in the near future.

The work reported here, called *Incremental MRQL*, extends the current MRQL Streaming engine with incremental stream processing. Some preliminary ideas on incremental query processing have appeared in our earlier work [14]. The research paper [14] describes the basic incremental processing method using examples, but it does not provide a detail algorithm, it does not address iterative algorithms, and it does not prove the soundness of the approach. Finally, the experiments reported in [14] were performed on a virtual cluster, while our experiments in this paper were done on a private cluster.

5 The MRQL Algebra

The MRQL compiler translates queries to algebraic terms and then uses rewrite rules to put these algebraic terms into a homomorphic form, which is then used to compute the query results incrementally by combining the previous results with the results of processing the incremental batches.

The MRQL algebra described in this section is a variation of the algebra presented in our previous work [16], but is more suitable for describing our incremental methods. The relational algebra, the nested relational algebra, as well as many other database algebras can be easily translated to our algebra. Our algebra consists of a small number of higher-order homomorphic operators [16] that are defined using structural recursion based on the union representation of bags [17].

The first operator, cMap (also known as concat-map or flatten-map in functional programming languages), 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 $\alpha \rightarrow \{\beta\}$ 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, it is expressed as

$$\mathrm{cMap}(f, X) \triangleq \{\, z \,|\, x \in X,\, z \in f(x)\,\}, \tag{1}$$

or, alternatively, using structural recursion

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

Given an arbitrary type κ that supports value equality (=), an arbitrary type 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,"A"), (2,"B"), (1,"C") }) returns {(1, {"A","C"}), (2,{"B"})}. The groupBy operation cannot be defined using a set former notation, but can be defined using structural recursion

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

where the parametric monoid $\Updownarrow_{\oplus}$ is a full outer join that merges groups associated with the same key using the monoid $\oplus$ (equal to $\uplus$ for groupBy)

$$\begin{aligned}
X \Updownarrow_{\oplus} Y \triangleq &\{\, (k, a \oplus b) \,|\, (k, a) \in X,\, (k, b) \in Y \,\}\\
&\uplus \{\, (k, a) \,|\, (k, a) \in X,\, k \notin \pi_1(Y) \,\}\\
&\uplus \{\, (k, b) \,|\, (k, b) \in Y,\, k \notin \pi_1(X) \,\},
\end{aligned} \tag{2}$$

where $\pi_1(X) = \{\, k \,|\, (k, x) \in X \,\}$. In other words, the monoid $\Updownarrow_{\uplus}$ constructs a set of pairs whose unique key is the first pair element. In fact, any bag X can be converted to a set using $\pi_1(\mathrm{groupBy}(\mathrm{cMap}(\lambda x.\, \{(x, x)\},\, X)))$. Note also that $\mathrm{groupBy}(X)$ is not the same as the nesting $\{\, (k, \{\, y \,|\, (k', y) \in X,\, k = k' \,\}) \,|\, (k, x) \in X \,\}$, as the latter contains duplicate entries for the key k. Unlike nesting, unnesting a groupBy returns the input bag (proven in Appendix B, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TKDE.2016.2601103, in the Supplemental Material file)

$$\{\, (k, v) \,|\, (k, s) \in \mathrm{groupBy}(X),\, v \in s \,\} \;=\; X. \tag{3}$$

Although a join $X \bowtie_p Y$ can be expressed as a nested cMap

$$\mathrm{cMap}(\lambda x.\, \mathrm{cMap}(\lambda y.\, \mathrm{if}\, p(x, y)\, \mathrm{then}\, \{(x, y)\}\, \mathrm{else}\, \{\,\},\, Y),\, X),$$

this term is not always a homomorphism on both inputs. Instead, MRQL provides a special homomorphic operation for equi-joins and outer joins, $\mathrm{coGroup}(X, Y)$, between a bag X of type $\{(\kappa, \alpha)\}$ and a bag Y of type $\{(\kappa, \beta)\}$ over their first component of a type κ, which returns a bag of type $\{(\kappa, (\{\alpha\}, \{\beta\}))\}$

$$\begin{aligned}
&\mathrm{coGroup}(X_1 \uplus X_2, Y_1 \uplus Y_2)\\
&\quad = \mathrm{coGroup}(X_1, Y_1) \Updownarrow_{\uplus\times\uplus} \mathrm{coGroup}(X_2, Y_2)\\
&\mathrm{coGroup}(\{(k, a)\}, \{(k, b)\}) = \{(k, (\{a\}, \{b\}))\}\\
&\mathrm{coGroup}(\{(k, a)\}, \{(k', b)\})\\
&\quad = \{(k, (\{a\}, \{\,\})),\, (k', (\{\,\}, \{b\}))\} \quad (\text{for } k' \neq k)\\
&\mathrm{coGroup}(\{(k, a)\}, \{\,\}) = \{(k, (\{a\}, \{\,\}))\}\\
&\mathrm{coGroup}(\{\,\}, \{(k, b)\}) = \{(k, (\{\,\}, \{b\}))\}\\
&\mathrm{coGroup}(\{\,\}, \{\,\}) = \{\,\},
\end{aligned}$$

where the product of two monoids, $\oplus \times \otimes$ is a monoid that, when applied to 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 $\Updownarrow_{\uplus\times\uplus}$ 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,"A"), (2,"B"), (1,"C") }, { (1,"D"), (2,"E"), (3,"F") }) returns {(1,({"A","C"},{"D"})), (2,({"B"},{"E"})), (3,({ },{"F"}))}. It can be proven (with a proof similar to that of Equation (3)) that both coGroup inputs can be derived from the coGroup result

$$\begin{aligned}
\{\, (k, x) \,|\, (k, (s_1, s_2)) \in \mathrm{coGroup}(X, Y),\, x \in s_1 \,\} &= X\\
\{\, (k, y) \,|\, (k, (s_1, s_2)) \in \mathrm{coGroup}(X, Y),\, y \in s_2 \,\} &= Y.
\end{aligned}$$

Aggregations are captured by the operation $\text{reduce}(\oplus, X)$, which aggregates a bag X using a commutative monoid $\oplus$. For example, $\text{reduce}(+, \{1, 2, 3\}) = 6$. Finally, the iteration $\text{repeat}(F, n, X)$ over the bag X of type $\{\alpha\}$ applies the function F of type $\{\alpha\} \to \{\alpha\}$ to X n times, $F(F(\ldots F(X)))$, yielding a bag of type $\{\alpha\}$.

Definition 1 (MRQL Algebra). *The MRQL algebra consists of terms that take the following form:*

$$\begin{array}{llll} e & ::= & \text{cMap}(f, e) & \text{flatten-map} \\ & | & \text{groupBy}(e) & \text{group-by} \\ & | & \text{coGroup}(e, e) & \text{join} \\ & | & \text{reduce}(\oplus, e) & \text{aggregation} \\ & | & S_i & \text{stream source,} \end{array}$$

where $\oplus$ is a monoid on basic types, such as $+$, $*$, $\wedge$, and $\vee$. Function f is an anonymous function that may contain such algebraic terms but is not permitted to contain any reference to a stream source, S_i.

The restriction on f in Definition 1 excludes non-equi-joins, such as cross products, which require a nested cMap in which the inner cMap is over a data source. For brevity, this algebra does not include terms for non-streaming input sources, general tuple and record construction and projection, iteration, bag union, singleton and empty bag, arithmetic operations, if-then-else expressions, boolean operations, etc.

For example, the join-groupBy query $q(S_1, S_2)$ used as the first example in Section 2 (the join-groupBy query) is translated to the following algebraic term $Q(S_1, S_2)$:

$$\begin{array}{l} \text{cMap}(\lambda(k, s).\{(k, avg(s))\}, \\ \quad \text{groupBy}(\text{cMap}(\lambda(j, (xs, ys)).g(xs, ys), \\ \qquad \text{coGroup}(\text{cMap}(\lambda x.\{(x.B, x)\}, S_1), \\ \qquad\quad \text{cMap}(\lambda y.\{(y.C, y)\}, S_2))))), \end{array}$$

where $avg(s) = \text{reduce}(+, s)/\text{reduce}(+, \text{cMap}(\lambda v.\, \{1\}, s))$and $g(xs, ys) = \text{cMap}(\lambda x.\, \text{cMap}(\lambda y.\, \{(x.A, y.D)\},\, ys),\, xs)$.

Although all algebraic operators used in MRQL are homomorphisms, their composition may not be. For instance, $\text{cMap}(f, \text{groupBy}(X))$ is not a homomorphism for certain functions f, because, in general, cMap does not distribute over $\Updownarrow_{\uplus}$. One of our goals is to transform any composition of algebraic operations into a homomorphism.

6 Query Normalization

In earlier work, we presented general algorithms for unnesting nested queries [16]. For example, consider the following nested query:

$$\begin{array}{l} \textbf{select}\ \mathsf{x}\ \textbf{from}\ \mathsf{x}\ \textbf{in}\ X \\ \textbf{where}\ \mathsf{x.D} > \textbf{sum}(\textbf{select}\ \mathsf{y.C}\ \textbf{from}\ \mathsf{y}\ \textbf{in}\ Y\ \textbf{where}\ \mathsf{x.A} = \mathsf{y.B}). \end{array}$$

A typical method for evaluating this query in a relational system is to first group Y by y.B, yielding pairs of y.B and sum(y.C), and then to join the result with X on x.A = y.B using a left-outer join, removing all those matches whose x.D is below the sum. But, in our framework, this query is translated into

$$\begin{array}{l} \text{cMap}(\lambda(\mathsf{k}, (\text{xs}, \text{ys})).\text{cMap}(\lambda \mathsf{x}\ \textbf{if}\ \mathsf{x.D} > \text{reduce}(+, \text{ys}) \\ \qquad\qquad \textbf{then}\{\mathsf{x}\}\ \textbf{else}\{\,\}, \text{xs}) \\ \quad \text{coGroup}(\text{cMap}(\lambda \mathsf{x}.\{(\mathsf{x.A}, \mathsf{x})\}, \text{X}), \\ \qquad \text{cMap}(\lambda \mathsf{y}.\{(\mathsf{y.B}, \mathsf{y.C})\}, \text{Y}))) \end{array}$$

That is, the query unnesting is done with a left-outer join, which is captured concisely by the coGroup operation without the need for using an additional group-by operation or handling null values. This unnesting technique was generalized to handle arbitrary nested queries, at any place, number, and nesting level. (The reader is referred to our earlier work [16] for details.)

After query unnesting, the algebraic terms derived from MRQL queries can be normalized using the following rule:

$$\text{cMap}(f, \text{cMap}(g, S)) \to \text{cMap}(\lambda x.\, \text{cMap}(f, g(x)), S), \quad (4)$$

which fuses two cascaded cMaps into a nested cMap, thus avoiding the construction of the intermediate bag. This rule can be proven directly from the cMap definition in Equation (1)

$$\begin{array}{l} \text{cMap}(f, \text{cMap}(g, S)) \\ = \{\, z \,|\, w \in \{\, y \,|\, x \in S,\, y \in g(x) \,\},\, z \in f(w) \,\} \\ = \{\, z \,|\, x \in S,\, y \in g(x),\, z \in f(y) \,\} \\ = \{\, z \,|\, x \in S,\, z \in \{\, w \,|\, y \in g(x),\, w \in f(y) \,\} \,\} \\ = \text{cMap}(\lambda x.\, \text{cMap}(f, g(x)), S). \end{array}$$

If we apply the transformation (4) repeatedly, and given that we can always use the identity $\text{cMap}(\lambda x.\{x\}, X) = X$ in places where there is no cMap between groupBy and coGroup operations, any algebraic terms in Definition 1 can be normalized into the following form:

Definition 2 (Normalized MRQL Algebra). *The normalized MRQL algebra consists of terms q that take the following form:*

$$\begin{array}{llll} q & ::= & \text{reduce}(\oplus, c) & \textit{the query header} \\ & | & c & \\ c & ::= & \text{cMap}(f, e) & \\ e & ::= & \text{groupBy}(c) & \textit{the query body} \\ & | & \text{coGroup}(c, c) & \\ & | & S_i, & \end{array}$$

where function f is an anonymous function that does not contain any reference to a stream source, S_i.

That is, the query body is a tree of groupBy and coGroup operations connected via cMaps.

7 Monoid Inference

One of our tasks is, given an algebraic term $f(\overline{S})$, where an $S_i \in \overline{S}$ is a streaming data source, to prove that f is a homomorphism by deriving a monoid $\otimes$ such that

$$f(S_1 \uplus S_1', \ldots, S_n \uplus S_n') = f(S_1, \ldots, S_n) \otimes f(S_1', \ldots, S_n'). \quad (6)$$

We have developed a monoid inference system, inspired by type inference systems used in programming languages. We use the judgment $\rho \vdash e : \oplus$ to indicate that e is a monoid

$$\rho \vdash v : \rho(v) \tag{5a}$$

$$\frac{\forall v \in e : \rho \vdash v : \boxdot}{\rho \vdash e : \boxdot} \tag{5b}$$

$$\frac{\rho \vdash X : \uplus}{\rho \vdash \text{reduce}(\oplus, X) : \oplus} \tag{5c}$$

$$\frac{\rho \vdash X : \uplus,\ \rho \vdash Y : \uplus}{\rho \vdash (X \uplus Y) : \uplus} \tag{5d}$$

$$\frac{\rho \vdash X : \uplus}{\rho \vdash \text{cMap}(f, X) : \uplus} \tag{5e}$$

$$\frac{\rho \vdash X : \uplus}{\rho \vdash \text{groupBy}(X) : \Updownarrow_{\uplus}} \tag{5f}$$

$$\frac{\rho \vdash X : \uplus,\ \rho \vdash Y : \uplus}{\rho \vdash \text{coGroup}(X, Y) : \Updownarrow_{\uplus \times \uplus}} \tag{5g}$$

$$\frac{\rho \vdash X : \Updownarrow_{\oplus},\ \rho[k : \boxdot, s : \oplus] \vdash f(k, s) : \Updownarrow_{\otimes}}{\rho \vdash \text{cMap}(f, X) : \Updownarrow_{\otimes}} \tag{5h}$$

Fig. 2. Monoid inference rules.

homomorphism with a merge function $\oplus$ under the environment ρ, which binds variables to monoids. The notation $\rho(v)$ extracts the binding of the variable v, while $\rho[v : \oplus]$ extends the environment with a new binding from v to $\oplus$. Equation ((6)) can now be expressed as the judgment

$$[S_1 : \uplus, \ldots, S_n : \uplus] \vdash f(\overline{S}) : \otimes.$$

If a term is invariant under change, such as an invariant data source, it is associated with the special monoid $\boxdot$

$$X \boxdot Y \triangleq \begin{cases} X & \text{if } X = Y \\ \text{error} & \text{otherwise.} \end{cases}$$

Our monoid inference algorithm is a heuristic algorithm that annotates terms with monoids (when possible). It is very similar to type inference. Most of our inference rules are expressed as fractions: The numerator (above the line) contains the premises (separated by comma) and the denominator (below the line) is the conclusion. For example, the rule $\frac{\rho \vdash x : \oplus}{\rho \vdash f(x) : \otimes}$ indicates that $f(x_1 \oplus x_2) = f(x_1) \otimes f(x_2)$. Fig. 2 gives some of the inference rules. More rules will be given in Lemma 1. Rules (5c) through (5f) are derived directly from the algebraic definition of the operators. Rule (5a) retrieves the associated monoid of a variable v from the environment ρ. Rule (5b) indicates that if all the variables in a term e are invariant, then so is e. Rule (5h) indicates that a cMap over a groupBy is a homomorphism as long as its functional argument is a homomorphism. It can be proven as follows:

$$\begin{aligned}
&\text{cMap}(f, X \Updownarrow_{\oplus} Y) \\
&= \{(\theta, z) | (k, s) \in (X \Updownarrow_{\oplus} Y), (\theta, z) \in f(k, s)\} \\
&= \{(\theta, z) | (k, x) \in X, (k, y) \in Y, (\theta, z) \in f(k, x \oplus y)\} \\
&\quad \uplus \cdots \\
&= \{(\theta, z) | (k, x) \in X, (k, y) \in Y, \\
&\qquad\qquad (\theta, z) \in (f(k, x) \Updownarrow_{\otimes} f(k, y))\} \uplus \cdots \\
&= \{(\theta, z) | (k, x) \in X, (k, y) \in Y, \\
&\qquad\qquad (\theta, z) \in \{(\theta, a \otimes b) | (\theta, a) \in f(k, x), \\
&\qquad\qquad\qquad\qquad (\theta, b) \in f(k, y)\}\} \uplus \cdots \\
&= \{(\theta, a \otimes b) | (k, x) \in X, (k, y) \in Y, (\theta, a) \in f(k, x), \\
&\qquad\qquad (\theta, b) \in f(k, y)\} \uplus \cdots \\
&= \{(\theta, a \otimes b) | (\theta, a) \in \text{cMap}(f, X), (\theta, b) \in \text{cMap}(f, Y)\} \\
&\quad \uplus \cdots \qquad\qquad \text{(given that } k \text{ and } \theta \text{ are unique keys)} \\
&= \text{cMap}(f, X) \Updownarrow_{\otimes} \text{cMap}(f, Y),
\end{aligned}$$

where the ... are terms that have been omitted for brevity.

8 INJECTING LINEAGE TRACKING

In this section, we transform the algebraic terms given in Definition 2 in such a way that they propagate the join and group-by keys. More specifically, each value v returned by these terms is annotated with a lineage θ, as a pair (θ, v), where θ takes the following form:

$$\begin{array}{lll}
\theta \ ::= & (\kappa, \theta) & \text{extended with a groupBy key } \kappa \\
\quad | & (\kappa, (\theta, \theta)) & \text{extended with a coGroup key } \kappa \\
\quad | & () & \text{empty lineage.}
\end{array}$$

That is, the lineage θ of the query result v is the tree of the groupBy and coGroup keys that are used in deriving the result v (one key for each groupBy and coGroup operation). The lineage tree θ has the same shape as the tree of groupBy and coGroup operations derived from the query. We transform a query q in such a way that, if the output of the query is $\{t\}$ for some type t, then the transformed query will have output $\{(\theta, t)\}$. Furthermore, if the the output is a non-collection type t, then the transformed query will also have output $\{(\theta, t)\}$, which separates the contributions to t associated with each combination of group-by and join keys.

A query q in our framework is transformed in such a way that it propagates the lineage from the data stream sources to the query output, starting with the empty lineage $()$ at the sources and extending the lineage with the join and group-by keys. This transformation is accomplished with the help of the following functions:

$$\begin{aligned}\text{sMap1}(f, X) \triangleq \{((\theta, k), b) | ((\theta, k), a) \in X, \\ b \in f(k, a)\}\end{aligned} \tag{7a}$$

$$\begin{aligned}\text{sMap2}(f, X) \triangleq \{(k', ((k, \theta), b)) | ((k, \theta), a) \in X, \\ (k', b) \in f(k, a)\}\end{aligned} \tag{7b}$$

$$\text{sMap3}(f, X) \triangleq \{(k, ((), b)) | a \in X, (k, b) \in f(a)\} \tag{7c}$$

$$\text{swap}(X) \triangleq \{((k, \theta), v) | (k, (\theta, v)) \in X\} \tag{7d}$$

$$\begin{aligned}\text{mix}(X) \triangleq \{&((k, (\theta_x, \theta_y)), (xs, ys)) \\ &| (k, (s_1, s_2)) \in X, \\ &(\theta_x, xs) \in \text{groupBy}(s_1), \\ &(\theta_y, ys) \in \text{groupBy}(s_2)\}.\end{aligned} \tag{7e}$$

The sMap1 operation is a cMap that propagates the input lineage θ to the output as is. The sMap2 operation is a cMap that extends the input lineage θ with a groupBy or coGroup key k. The sMap3 operation embeds the empty lineage $()$ to

every value. The swap operation moves the previous lineage from the value to the key to prepare the values for a group-by on both the group-by key and the previous lineage. Finally, the mix operation extracts the lineage keys θ_x and θ_y from the coGroup result, X, which are the propagated lineages from the coGroup inputs, and combine them into a pair of lineages.

Algorithm 2. Lineage Annotation

Input: a normalized query term q defined in Definition 2.
Output: a term $\mathcal{T}_q[\![q]\!]$, which is q annotated with a lineage θ.

$$\mathcal{T}_q[\![\text{reduce}(\oplus,\, \text{cMap}(f,\, S_i))]\!] = \{((),\, \text{reduce}(\oplus,\, \text{cMap}(f,\, S_i)))\} \tag{8a}$$

$$\mathcal{T}_q[\![\text{reduce}(\oplus,\, \text{cMap}(f,\, e))]\!] = \text{reduce}(\Updownarrow_\oplus,\, \text{sMap1}(f,\, \mathcal{T}_e[\![e]\!])) \qquad e \neq S_i \tag{8b}$$

$$\mathcal{T}_q[\![\text{cMap}(f,\, S_i)]\!] = \{\, ((), b) \,|\, a \in S_i,\, b \in f(a) \,\} \tag{8c}$$

$$\mathcal{T}_q[\![\text{cMap}(f,\, e)]\!] = \text{sMap1}(f,\, \mathcal{T}_e[\![e]\!]) \qquad e \neq S_i \tag{8d}$$

$$\mathcal{T}_e[\![\text{groupBy}(c)]\!] = \text{groupBy}(\text{swap}(\mathcal{T}_c[\![c]\!])) \tag{8e}$$

$$\mathcal{T}_e[\![\text{coGroup}(c_1, c_2)]\!] = \text{mix}(\text{coGroup}(\mathcal{T}_c[\![c_1]\!], \mathcal{T}_c[\![c_2]\!])) \tag{8f}$$

$$\mathcal{T}_c[\![\text{cMap}(f,\, S_i)]\!] = \text{sMap3}(f,\, S_i) \tag{8g}$$

$$\mathcal{T}_c[\![\text{cMap}(f,\, e)]\!] = \text{sMap2}(f,\, \mathcal{T}_e[\![e]\!]) \qquad e \neq S_i \tag{8h}$$

The lineage propagation is done by the cMap Rules (8d) and (8h). Rule (8d) applies to the outer query cMap that produces the query output. It simply propagates the lineage from the cMap input to the output. Rule (8h) applies to a cMap that returns the input of a groupBy or coGroup. The output of this cMap must be a bag of key-value pairs, as is expected by a groupBy or a coGroup. Thus, Rule (8h) extends the lineage with a new key and prepares the cMap output for the enclosing groupBy or coGroup, which is done by translating cMap to sMap2. Rule (8b) indicates that a total aggregation becomes a group-by aggregation by aggregating the values of each group associated with a different lineage θ. Rule (8e) translates a groupBy on a key to a groupBy on the entire lineage (which includes the groupBy key). Rule (8f) translates a coGroup on a key to a coGroup on the entire lineage, but it is done using the function mix (defined in (7e) because the left input lineage θ_x is not necessarily compatible with the right input lineage θ_y). Rule (8f) generates a coGroup on the join key first, and then, for each join key k, it groups the left and right join matches by θ_x and θ_y respectively, so that the output contains unique lineage key combinations, $(k, (\theta_x, \theta_y))$. Finally, Rule (8g) annotates each value of the input stream S_i with the empty lineage $()$.

For example, consider again the algebraic term $Q(S_1, S_2)$ for the join-groupBy query, presented at the end of Section 5. If we apply the transformations in Equations (8a) through (8h), we derive the following term $\mathcal{T}_q[\![Q(S_1, S_2)]\!]$, which propagates the join and group-by keys to the query output:

$$\begin{aligned}&\text{sMap1}(\lambda(k, s).\{(k, avg(s))\},\\ &\quad \text{groupBy}(\text{swap}(\text{sMap2}(\lambda(j, (xs, ys)).g(xs, ys),\\ &\qquad \text{mix}(\text{coGroup}(\text{sMap3}(\lambda x.\{(x.B, x)\}, S_1),\\ &\qquad\qquad \text{sMap3}(\lambda y.\{(y.C, y)\}, S_2))))))),\end{aligned} \tag{9}$$

where $avg(s) = \text{reduce}(+, s)/\text{reduce}(+, \text{cMap}(\lambda v.\, \{1\}, s))$ and $g(xs, ys) = \text{cMap}(\lambda x.\, \text{cMap}(\lambda y.\, \{(x.A, y.D)\},\, ys),\, xs)$. If we expand sMap1, sMap2, swap, and mix, then the transformed query is

$$\begin{aligned}&\{(\theta, (k, avg(s)))\\ &|(\theta, (k, s)) \in \text{groupBy}(\{((k, (j, ((), ()))), v)\\ &\qquad |(j, (s_1, s_2)) \in join,\\ &\qquad ((), xs) \in \text{groupBy}(s_1),\\ &\qquad ((), ys) \in \text{groupBy}(s_2),\\ &\qquad (k, v) \in g(xs, ys)\})\},\end{aligned}$$

where $join$ is

$$\text{coGroup}(\{\, (x.B, ((), x)) \,|\, x \in S_1 \,\},\, \{\, (y.C, ((), y)) \,|\, y \in S_2 \,\}).$$

But, $\text{groupBy}(s_1) = \{((), \pi_2(s_1))\}$ and $\text{groupBy}(s_2) = \{((), \pi_2(s_2))\}$ since the key is $()$. Consequently, the transformed query becomes

$$\begin{aligned}&\{(\theta, (k, avg(s)))\\ &|(\theta, (k, s)) \in \text{groupBy}(\{((k, (j, ((), ()))), v)\\ &\qquad |(j, (s_1, s_2)) \in join,\\ &\qquad (k, v) \in g(\pi_2(s_1), \pi_2(s_2))\})\}.\end{aligned}$$

The following lemma indicates that the generated sMap1 and sMap2 in $\mathcal{T}_q[\![q]\!]$ are homomorphisms, provided that their functional arguments are homomorphisms

Lemma 1 (Transformed Term Judgments).

$$\frac{\rho \vdash X : \Updownarrow_\otimes,\, \rho[k : \boxdot, v :, \oplus] \vdash f(k, v) : \Updownarrow_\otimes}{\rho \vdash \text{sMap1}(f, X) : \Updownarrow_\otimes} \tag{10a}$$

$$\frac{\rho \vdash X : \Updownarrow_\oplus,\, \rho[k : \boxdot, v : \oplus] \vdash f(k, v) : \uplus}{\rho \vdash \text{sMap2}(f, X) : \uplus} \tag{10b}$$

$$\frac{\rho \vdash X : \uplus}{\rho \vdash \text{sMap3}(f, X) : \uplus} \tag{10c}$$

$$\frac{\rho \vdash X : \uplus}{\rho \vdash \text{swap}(X) : \uplus} \tag{10d}$$

$$\frac{\rho \vdash X : \Updownarrow_{\uplus \times \uplus}}{\rho \vdash \text{mix}(X) : \Updownarrow_{\uplus \times \uplus}}. \tag{10e}$$

The proof of this lemma is given in Appendix B, available online.

Definition 3 (Query Merger Monoid). *The query merger monoid $\mathcal{M}[\![q]\!]$ of a normalized query term q is defined as follows:*

$$\begin{aligned}\mathcal{M}[\![\text{reduce}(\oplus,\, \text{cMap}(f,\, e))]\!] &= \Updownarrow_\oplus\\ \mathcal{M}[\![\text{cMap}(f,\, S_i)]\!] &= \uplus\\ \mathcal{M}[\![\text{cMap}(f,\, e)]\!] &= \Updownarrow_\otimes \quad \text{if } e \neq S_i,\end{aligned}$$

where the monoid $\Updownarrow_\otimes$ in the last equation comes from $\text{sMap1}(f,\, X) : \Updownarrow_\otimes$ *in Equation (10a).*

Based on the judgments in Lemma 1, the following theorem indicates that the transformed query is a homomorphism

Theorem 1 (Homomorphism). *Any transformed term $\mathcal{T}_q[\![q]\!]$, where q is defined in Definition 2, is a homomorphism over the input streams, provided that each generated sMap1 and sMap2 term in $\mathcal{T}_q[\![q]\!]$ satisfies the premises in Judgment (10a) and (10b)*

$$\rho[S_1 : \uplus, \ldots, S_n : \uplus] \vdash \mathcal{T}_q[\![q]\!] :\ \mathcal{M}[\![q]\!]. \tag{11}$$

The proof is given in Appendix B, available online.

Definition 4 (Query Answer): *The query answer $\mathcal{A}[\![q]\!]_x$ of the query q, defined in Definition 2, is a function over the current state $x = \mathcal{T}_q[\![q]\!]$ and is derived as follows:*

$$\begin{aligned}
&\mathcal{A}[\![\text{cMap}(f,\, S_i)]\!]_x = \pi_2(x)\\
&\mathcal{A}[\![\text{reduce}(\oplus,\, \text{cMap}(f,\, e))]\!]_x = \text{reduce}(\oplus,\, \pi_2(x))\\
&\mathcal{A}[\![\text{cMap}(f,\, e)]\!]_x = \pi_2(\text{reduce}(\Updownarrow_\otimes, T(x))) \quad \text{if } e \neq S_i\\
&\mathcal{A}[\![q]\!]_x = \pi_2(\text{elem}(\text{reduce}(\Updownarrow_\oplus, x))) \qquad \textit{otherwise},
\end{aligned}$$

where the monoid $\oplus$ in the last equation is $\mathcal{M}[\![q]\!]$, the monoid $\otimes$ in the second equation comes from $\text{sMap1}(f,\, X) : \Updownarrow_\otimes$ *in Equation (10a), and*

$$\text{elem}(X) = \begin{cases} v & \text{if } X = \{v\}\\ \text{error} & \textit{otherwise} \end{cases}$$

$$T(X) = \{(k, a) | ((k, \theta), a) \in X\}.$$

The following theorem indicates that $\mathcal{A}[\![q]\!]_x$ returns the same answer as q:

Theorem 2 (Correctness). *The $\mathcal{A}[\![q]\!]_x$ over the state $x = \mathcal{T}_q[\![q]\!]$ returns the same result as the original query q, where q is defined in Definition 2:*

$$\mathcal{A}[\![q]\!]_x = q \qquad \textit{for } x = \mathcal{T}_q[\![q]\!]. \tag{12}$$

The proof of this theorem is given in Appendix B, available online.

8.1 Restricting Joins

Theorem 1 indicates that a query transformed by Algorithm 1 is a homomorphism if the cMap functional arguments in the query satisfy the premises in Judgments (10a) and (10b). But, consider the following join:

$$\begin{aligned}&\text{cMap}(\lambda(k, (xs, ys)).\text{cMap}(\lambda x.\text{cMap}(\lambda y.\{(x, y)\}, xs), ys),\\ &\qquad \text{coGroup}(X, Y)).\end{aligned}$$

Our monoid inference system cannot prove that the functional argument of the outer cMap is a homomorphism on both xs and ys. This limitation is expected because this join could be a many-to-many join, which we know cannot be a homomorphism. But, as we will show, if we give up on handling many-to-many joins, we can extend the monoid inference algorithm to always assume that every join is a one-to-one or one-to-many join and draw inferences based on this assumption.

We have already seen in Section 7 that if a term is invariant under change, such as an invariant data source, it is associated with the special monoid $\boxdot$. We can also use the annotation $\boxdot$ to denote certain functional dependencies, such as $\kappa \rightarrow \alpha$ on a bag X of type $\{(\kappa, \alpha)\}$, which indicates that the second component of a pair in X depends on the first. This dependency is captured by annotating $\text{groupBy}(X)$ with the monoid $\Updownarrow_\boxdot$, which indicates that each group remains invariant under change, implying that the group-by key is also a unique key of X. That is, if $(k, s_1) \in X_1$ and $(k, s_2) \in X_2$, then $\{\, (k, s_1 \boxdot s_2) \,|\, (k, s_1) \in X_1,\ (k, s_2) \in X_2 \,\}$ in Definition (2) must have $s_1 = s_2$ so that $s_1 \boxdot s_2 = s_1$, otherwise it will be an error. This assertion implies that we cannot have two different groups associated with the same key k. Given that a groupBy over a singleton gives a singleton group, each group returned from a groupBy is a singleton that remains invariant. A similar functional dependency can also apply to a join between X of type $\{(\kappa, \alpha)\}$ and Y of type $\{(\kappa, \beta)\}$, which is a $\text{coGroup}(X, Y)$ of type $\{(\kappa, (\{\alpha\}, \{\beta\}))\}$. To indicate that this join is one-to-one or one-to-many, we annotate coGroup (X, Y) with the monoid $\Updownarrow_{\boxdot \times \uplus}$, which enforces the constraint that the bag $\{\alpha\}$ in $\{(\kappa, (\{\alpha\}, \{\beta\}))\}$ be either empty or singleton, that is, at most one X element can be joined with Y over a key κ. This constraint is imposed by merging the X and Y values that are joined over the same key with $\boxdot$ and $\uplus$, respectively, as indicated by $\Updownarrow_{\boxdot \times \uplus}$. Therefore, to incorporate the assumption that all joins are one-to-one or one-to-many in the monoid inference system, we have to replace Judgment (5g) with the following judgment:

$$\frac{\rho \vdash X : \uplus,\ \rho \vdash Y : \uplus}{\rho \vdash \text{coGroup}(X, Y) : \Updownarrow_{\boxdot \times \uplus}}. \tag{13}$$

Given this judgment, the cMap input of our join example will be annotated with $\Updownarrow_{\boxdot \times \uplus}$, which means that xs and ys will be annotated with $\boxdot$ and $\uplus$, respectively (i.e., xs is invariant). Based on these annotations, the functional argument $\text{cMap}(\lambda x.\, \text{cMap}(\lambda y.\, \{(x, y)\},\, xs),\, ys)$ satisfies the premise in Judgment (10b), since it is annotated with $\uplus$.

8.2 Handling Iterative Queries

Algorithm 2 presented in Section 2, which processes iterations incrementally, requires an additional operation, called the diffusion operator $\widehat{\otimes}$, that satisfies the following property:

$$T \otimes (T \,\widehat{\otimes}\, \Delta T) = T \otimes (T \otimes \Delta T).$$

The diffusion operation $\widehat{\otimes}$ can be easily defined in terms of $\otimes$ in such a way that $T \,\widehat{\otimes}\, \Delta T$ contains only those parts of $T \otimes \Delta T$ that have changed from T

Definition 5 (Diffusion). *The diffusion $\widehat{\otimes}$ of a monoid $\otimes$ is defined as follows:*

$$\begin{aligned}
\widehat{\Updownarrow_\oplus} &= \Downarrow_{\widehat{\oplus}} \qquad \widehat{\otimes \times \oplus} = \widehat{\otimes} \times \widehat{\oplus}\\
\widehat{\oplus} &= \oplus \qquad \textit{for any other monoid},
\end{aligned}$$

where $\Downarrow_{\otimes}$ is a right-outer join defined as follows:

$$X \Downarrow_{\otimes} Y \quad \triangleq \quad \{ (k, a \otimes b) \,|\, (k,a) \in X,\, (k,b) \in Y \} \uplus \{ (k,b) \,|\, (k,b) \in Y,\, k \notin \pi_1(X) \}.$$

The following theorem indicates that the diffusion operation $\widehat{\oplus}$ satisfies the desired property:

Theorem 3 (Diffusion Property).

$$T \otimes (T \,\widehat{\otimes}\, \Delta T) = T \otimes (T \otimes \Delta T).$$

This theorem is proven in Appendix B, available online. This appendix also contains the proof of the following theorem:

Theorem 4 (Correctness of Algorithm 2). *Given the iteration*

$$X \leftarrow g(\overline{S})$$
$$\textbf{for}\ \ i \leftarrow 1 \ldots n: \quad X \leftarrow f(X, \overline{S}).$$

The query answer X calculated by Algorithm 2 is approximately equal to the result X of the above iteration.

The main reason that Algorithm 2 calculates an approximate solution, rather than an accurate one, is that the state at time $t + \Delta t$ is formed by combining the state on streams $\overline{S}$ with the state on streams $\overline{\Delta S}$, but the state on $\overline{S}$ is taken to be constant (equal to the state at time t) across the iteration in Algorithm 2, which is an approximation.

9 Non-Homomorphic Terms

Theorem 2 indicates that the terms generated by the transformations (8a) through (8h) are homomorphisms as long as the premises of the judgements in Lemma 1 are true. These premises indicate that the functional arguments must be homomorphisms too. We want the operations that cannot be annotated with a monoid to be transformed so that the non-homomorphic parts of the operation are pulled outwards from the query using rewrite rules. We achieve this transformation with the help of kMap

$$\text{kMap}(f, X) \triangleq \text{cMap}(\lambda(\theta, v).\, \{(\theta, f(v))\},\, X),$$

which is a cMap that propagates the lineage θ as is. In our framework, all non-homomorphic parts take the form of a kMap and are accumulated into one kMap using

$$\text{kMap}(f, \text{kMap}(g, X)) \;\rightarrow\; \text{kMap}(\lambda x.\, f(g(x)),\, X).$$

More specifically, we first split each non-homomorphic cMap to a composition of a kMap and a cMap so that the latter cMap is a homomorphism, and then we pull and merge kMaps. Consider the term $\text{cMap}(\lambda(\theta, v).\, \{(\theta', e)\},\, X)$, which creates a new lineage θ' from the old θ. In our framework, we find the largest subterms in the algebraic term e, namely $e_1, \ldots, e_n$, that are homomorphisms. This task is accomplished by traversing the tree that represents the term e, starting from the root, and by checking if a 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

$$\begin{aligned}
&\text{cMap}(\lambda(\theta, v).\{(\theta', e)\}, X)\\
&\quad \rightarrow \text{cMap}(\lambda(\theta, v).\{(\theta', f(e_1, \ldots, e_n))\}, X)\\
&\quad \rightarrow \text{kMap}(\lambda(x_1, \ldots, x_n).f(x_1, \ldots, x_n),\\
&\qquad\quad \text{cMap}(\lambda(\theta, v).\{(\theta', (e_1, \ldots, e_n))\}, X)).
\end{aligned}$$

The kMaps are combined and are pulled outwards from the query using the following rewrite rules:

$$\begin{aligned}
&\text{sMap2}(g, \text{kMap}(f, X)) \rightarrow \text{sMap2}(\lambda(k, v).g(k, f(v)), X)\\
&\text{sMap1}(g, \text{kMap}(f, X)) \rightarrow \text{sMap1}(\lambda(k, v).g(k, f(v)), X)\\
&\text{groupBy}(\text{kMap}(f, X))\\
&\quad \rightarrow \text{kMap}(\lambda s.\text{map}(f, s), \text{groupBy}(X))\\
&\text{coGroup}(\text{kMap}(f_x, X), \text{kMap}(f_y, Y))\\
&\quad \rightarrow \text{kMap}(\lambda(k, (xs, ys)).(k, (\text{map}(f_x, xs), \text{map}(f_y, ys))),\\
&\qquad\quad \text{coGroup}(X, Y)),
\end{aligned}$$

where $\text{map}(f, X) = \text{cMap}(\lambda x.\, \{f(x)\},\, X)$. Rewrite rules as these, when applied repeatedly, can pull out and combine the non-homomorphic parts of a query, leaving a homomorphism whose merge function can be derived from our annotation rules.

For example, consider again the algebraic term $Q(S_1, S_2)$, presented at the end of Section 5, which was transformed to the query (9) in Section 8. We now check if the transformed query (9) is a homomorphism. Both coGroup inputs in (9) are sMap2 terms, annotated with $\uplus$, which means that, based on Equation (5g), the coGroup is annotated with $\Updownarrow_{\boxdot \times \uplus}$. From Equation (10b), the sMap2 operation is annotated with $\uplus$ as long as $g(xs, ys)$ is annotated with $\uplus$. Hence, the groupBy operation is annotated with $\Updownarrow_{\uplus}$, based on Equation (5f). Unfortunately, based on Equation (7a), the sMap1 term is not a homomorphism as is, because $avg(s)$ is not a homomorphism over s. Based on the methods described in Section 9 that factor out non-homomorhic parts from terms, the sMap1 term is broken into two terms $a(h(S_1, S_2))$, where $h(S_1, S_2)$ is

$$\begin{aligned}
&\text{sMap1}(\lambda(k, s).\{(k, (\text{reduce}(+, s),\\
&\qquad \text{reduce}(+, \text{cMap}(\lambda v.\, 1, s))))\}, \ldots).
\end{aligned}$$

This term is equivalent to the homomorphism $h(S_1, S_2)$ given in Section 2.

In addition, from Definition 4, the answer function $a(x)$ is $\pi_2(\text{reduce}(\Updownarrow_{\otimes}, T(x)))$. Therefore, the answer function $a(x)$, combined with the non-homomorphic part of the query, is

$$\begin{aligned}
&\text{cMap}(\lambda(k, (s, c)).\{(k, s/c)\},\\
&\qquad \text{reduce}(\Updownarrow_{\boxdot \times ((+) \times (+))}, T(x)))
\end{aligned}$$

This term is equivalent to the answer query given in Section 2. Finally, $h(S_1, S_2)$ is a homomorphism annotated with $\Updownarrow_{\boxdot \times ((+) \times (+))}$, because the reduce terms are annotated with the monoid $(+)$ for numerical addition (from Equation (5c)) and we have a pair of homomorphisms. This term is equivalent to the merge function $X \otimes Y$ given in

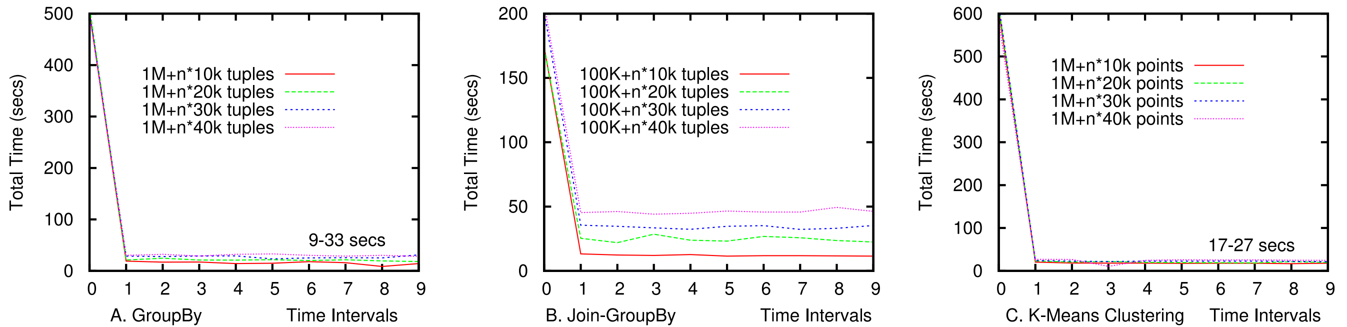

Fig. 3. Incremental query evaluation of GroupBy, Join-GroupBy, and K-Means clustering.

Section 2, implemented as a partitioned join combined with aggregation.

10 Implementation

We have implemented our incremental processing framework using Apache MRQL [28] on top of Apache Spark Streaming [38]. 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.

We have introduced a new physical operator, called Incr, which is a stateful operator that updates a state. More specifically, every instance of this operation is annotated with a state number i and is associated with a state, state_i, of type α_i. The operation $\mathrm{Incr}(i, s_0, F)$, where F is a state transition function of type $\alpha_i \rightarrow \alpha_i$ and s_0 is the initial state of type α_i, has the following semantics:

$$\begin{aligned}&\mathrm{state}_i \;\leftarrow\; F(\mathrm{state}_i)\\&\mathbf{return}\ \mathrm{state}_i,\end{aligned}$$

with $\mathrm{state}_i = s_0$, initially.

Recall that, in our framework, we break a query q into a homomorphism h and an answer function a, and we evaluate the query incrementally using

$$\begin{aligned}&\mathrm{state}_1 \;\leftarrow\; \mathrm{state}_1 \otimes h(\overline{\Delta S})\\&\mathbf{return}\ a(\mathrm{state}_1),\end{aligned}$$

at every time interval, where, initially, $\mathrm{state}_1 = \otimes_z$. This operation is implemented using the following physical plan

$$a(\mathrm{Incr}(1, \otimes_z, \lambda T.\, T \otimes h(\overline{\Delta S}))).$$

For an iteration $\mathrm{repeat}(\lambda X.\, f(X, \overline{\Delta S}), n, g(\overline{S}))$, we use the approximate solution, described in Section 2: We split f into a homomorphism h, for some monoid $\otimes$, and a function a, and we derive a diffusion operator $\widehat{\otimes}$

$$\begin{aligned}a(\mathrm{Incr}(1, \otimes_z,\\&\lambda T.T \otimes \mathrm{repeat}(\lambda \Delta T.T\ \widehat{\oplus}\ h(a(\Delta T), \overline{\Delta S}),\\&\qquad n-1,\\&\qquad T\ \widehat{\oplus}\ h(g(\overline{\Delta S}), \overline{\Delta S})))).\end{aligned}$$

11 Performance Evaluation

The system described in this paper is available as part of the latest official MRQL release (MRQL 0.9.6). We have experimentally validated the effectiveness of our methods using four queries: groupBy, join-groupBy, k-means clustering, and PageRank. The platform used for our evaluations is a small cluster of nine Linux servers, connected through a Gigabit Ethernet switch. Each server has four Xeon cores at 3.2 GHz with 4 GB memory. For our experiments, we used Hadoop 2.2.0 (Yarn) and Spark 1.6.0. The cluster frontend was used exclusively as a NameNode and ResourceManager, while the rest eight compute nodes were used as DataNodes and NodeManagers. For our experiments, we used all the available 32 cores of the compute nodes for Spark tasks.

The data streams used by the first three queries (groupBy, join-groupBy, and k-means clustering) consist of a large set of initial data, which is used to initialize the state, followed by a sequence of nine equal-size batches of data (the increments). The experiments were repeated for increments of size 10, 20, 30, and 40 K tuples, always starting with a fresh state (constructed from the initial data only). The performance results are shown in Fig. 3. The x-axis represents the time points Δt when we get new batches of data in the stream. At time $0\Delta t$, we have the processing of the initial data and the construction of the initial state. Then, the nine increments arrive at the time points $1\Delta t$ through $9\Delta t$. The y-axis is the query execution time, and there are four plots, one for each increment size.

The join-groupBy and the k-means clustering queries aregiven in Section 2. The groupBy query is 'select(x, avg(y)) from (x, y) in S group by x'. The datasets used for both the groupBy and join-groupBy queries consist of pairs of random integers between 0 and 10,000. The groupBy initial dataset has size 1 M tuples, while the two join-groupBy inputs have size 100 K tuples. The datasets used for the k-means query consist of random (X, Y) points in four squares that have X in $[2, 4]$ or $[6, 8]$ and Y in $[2, 4]$ or $[6, 8]$. Thus, the four centroids are expected to be $(3, 3)$, $(3, 7)$, $(7, 3)$, and $(7, 7)$, which also means that the state contains four centroids only. The initial dataset for k-means contains 1 M points and the k-means query uses 10 iteration steps. The k-means incremental program uses the approximate solution described in Section 2.

From Fig. 3, we can see that processing incremental batches of data can give an order of magnitude speed-up compared to processing all the data each time. Furthermore,

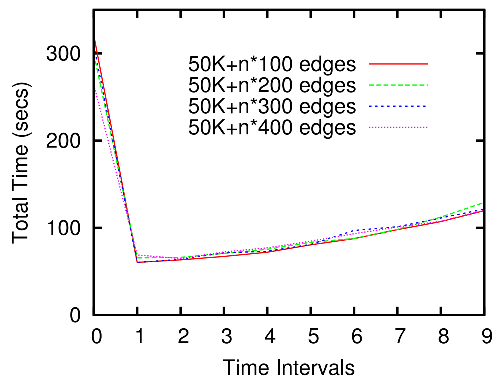

Fig. 4. Incremental query evaluation of PageRank.

the time to process each increment does not substantially increase through time, despite that the state grows with new data each time (in the case of the groupBy and join-groupBy queries). This invariance in the time overhead happens because merging states is done with a partitioned join (implemented as a coGroup in Spark) so that the new state created by coGroup is already partitioned by the join key and is ready to be used for the next coGroup to handle the next increment. Consequently, only the results of processing the new data, which are typically smaller than the state, are shuffled across the worker nodes before coGroup. This approach makes the incremental processing time largely independent of the state size in most cases since data shuffling is the most prevalent factor in main-memory distributed processing systems.

We have also evaluated our system on a PageRank query on random graphs generated by the R-MAT algorithm [7] using the Kronecker graph generator parameters: a=0.30, b=0.25, c=0.25, and d=0.20. The PageRank query expressed in MRQL is a simple self-join on the graph, which is optimized into a single group-by operation. (The MRQL PageRank query is given in [16].) The PageRanks were calculated in 10 iteration steps. This time, the initial graph used in our evaluations had size 5 K nodes with 50 K edges and the four different increments had sizes 100 nodes with 100, 200, 300, and 400 edges, respectively. The performance results are shown in Fig. 4. We believe that the reason that we did not get a flat line for our incremental PageRank implementation was due to the lack of sufficient memory tuning. In Spark, cached RDDs and D-Streams are not automatically garbage-collected; instead, Spark leaves it to the programmer to cache the queued operations in memory if their results are used multiple times, and to dispose the cache later. Removing all memory leaks is very crucial to continuous stream processing, and, if not done properly, it will eventually cause the system to run out of memory. We are planning to fine-tune our streaming engine to remove all these memory leaks.

12 Conclusion

We have presented a general framework for translating batch MRQL queries to incremental DSPE programs. In contrast to other systems, our methods are completely automated and are formally proven to be correct. In addition to incremental query processing on a DSPE platform, our framework can also be used on a batch distributed system to process data larger than the available distributed memory, by processing these data incrementally, in batches that can fit in memory. Although our methods are described using the unconventional MRQL algebra, instead of the nested relational algebra, we believe that many other similar query systems can use our framework by simply translating their algebraic operators to the MRQL operators, then, using our framework as is, and, finally, translating the resulting operations back to their own algebra.

Acknowledgments

This work is supported in part by the US National Science Foundation under the grant CCF-1117369. Some of our performance evaluations were performed at the Chameleon cloud computing infrastructure, supported by US National Science Foundation, https://www.chameleoncloud.org/.

References

[1] D. J. Abadi, et al., “Aurora: A new model and architecture for data stream management,” *VLDB J.*, vol. 12, no. 2, pp. 120–139, 2003.
[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, “Models and issues in data stream systems,” in *Proc. 21st ACM SIGMOD-SIGACT-SIGART Symp. Principles Database Syst.*, 2002, pp. 1–16.
[3] O. Benjelloun, A. D. Sarma, A. Halevy, and J. Widom, “ULDBs: Databases with uncertainty and lineage,” in *Proc. 32nd Int. Conf. Very Large Data Bases*, 2006, pp. 953–964.
[4] D. Bhagwat, L. Chiticariu, W. C. Tan, and G. Vijayvargiya. “An annotation management system for relational databases,” in *Proc. 30th Int. Conf. Very Large Data Bases*, 2004, pp. 900–911.
[5] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin, “Incoop: MapReduce for incremental computations,” in *Proc. 2nd ACM Symp. Cloud Comput.*, 2011, Art. no. 7.
[6] O. Boykin, S. Ritchie, I. O’Connell, and J. Lin, “Summingbird: A framework for integrating batch and online MapReduce computations,” *Proc. VLDB Endowment*, vol. 7, pp. 1441–1451, 2014.
[7] D. Chakrabarti, Y. Zhan, and C. Faloutsos, “R-MAT: A recursive model for graph mining,” in *Proc. 4th SIAM Int. Conf. Data Mining*, 2004, pp. 442–446.
[8] B. Chandramouli, et al., “Trill: A high-performance incremental query processor for diverse analytics,” in *Proc. VLDB Endowment*, vol. 8, pp. 401–412, 2014.
[9] S. Chandrasekaran, et al., “TelegraphCQ: Continuous data flow processing for an uncertain world,” in *Proc. 1st Biennial Conf. Innovative Data Syst. Res., 2003.*
[10] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, “MapReduce online,” in *Proc. 7th USENIX Symp. Netw. Syst. Des. Implementation*, 2010, pp. 21–21.
[11] Y. Cui and J. Widom, “Lineage tracing for general data warehouse transformations,” in *Proc. 27th Int. Conf. Very Large Data Bases*, 2001, pp. 471–480.
[12] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” in *Proc. 6th Conf. Symp. Operating Syst. Des. Implementation*, 2004, pp. 1–10.
[13] P. Desikan, N. Pathak, J. Srivastava, and V. Kumar, “Incremental PageRank computation on evolving graphs,” in *Special Interest Tracks Posters 14th Int. Conf. World Wide Web*, pp. 1094–1095, 2005.
[14] L. Fegaras, “Incremental stream processing of nested-relational queries,” in *Proc. 27th Int. Conf. Database Expert Syst. Appl.*, Sep. 2016, pp. 305–320.
[15] L. Fegaras, C. Li, U. Gupta, and J. J. Philip, “XML query optimization in Map-Reduce,” in *Proc. 14th Int. Workshop Web Databases*, 2011.
[16] L. Fegaras, C. Li, and U. Gupta, “An optimization framework for Map-Reduce queries,” in *Proc. 15th Int. Conf. Extending Database Technol.*, 2012, pp. 26–37.
[17] L. Fegaras and D. Maier, “Optimizing object queries using an effective calculus,” *ACM Trans. Database Syst.*, vol. 25, no. 4, pp. 457–516, 2000.
[18] Apache Flink. (2016). Online. Available: http://flink.apache.org/

[19] Apache Hadoop. (2016). Online. Available: http://hadoop.apache.org/
[20] Apache Hama. (2016). Online. Available: http://hama.apache.org/
[21] Apache Hive. (2016). Online. Available: http://hive.apache.org/
[22] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum, "Stateful bulk processing for incremental analytics," in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 51–62.
[23] 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," *Proc. VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.
[24] G. Malewicz, et al., "Pregel: A system for large-scale graph processing," in *Proc. 28th ACM Symp. Principles Distrib. Comput.*, 2009, pp. 6–6.
[25] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard, "Differential dataflow," in *Proc. Conf. Innovative Data Syst. Res.*, 2013.
[26] S. R. Mihaylov, Z. G. Ives, and S. Guha, "REX: Recursive, delta-based data-centric computation," *Proc. VLDB Endowment*, vol. 5, no. 11, pp. 1280–1291, 2012.
[27] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in *Proc. 24th ACM Symp. Operating Syst. Principles*, 2013, pp. 439–455.
[28] Apache MRQL (incubating). (2016). Online. Available: http://mrql.incubator.apache.org/
[29] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: A not-so-foreign language for data processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2008, pp. 1099–1110.
[30] D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications," in *Proc. 9th USENIX Conf. Operating Syst. Des. Implementation*, 2010, pp. 251–264.
[31] A. Shinnar, D. Cunningham, B. Herta, and V. Saraswat, "M3R: Increased performance for in-memory Hadoop jobs," *Proc. VLDB Endowment*, vol. 5, no. 12, pp. 1736–1747, 2012.
[32] Apache Samza. (2016). Online. Available: http://samza.apache.org/
[33] Apache Spark. (2016). Online. Available: http://spark.apache.org/
[34] Apache Storm: *A System for Processing Streaming Data in Real Time*. (2016). Online. Available: http://hortonworks.com/hadoop/storm/
[35] Apache S4 (incubating): A Distributed Stream Computing Platform. (2016). Online. Available: http://incubator.apache.org/s4/
[36] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu, "General incremental sliding-window aggregation," in *Proc. VLDB Endowment*, vol. 8, no. 7, pp. 702–713, 2015.
[37] M. Zaharia, et al., "Resilient distributed satasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Symp. Netw. Syst. Des. Implementation*, 2012, pp. 2–2.
[38] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proc. 24th ACM Symp. Operating Syst. Principles*, 2013, pp. 423–428.
[39] Y. Zhang, S. Chen, Q. Wang, and G. Yu, "i2MapReduce: Incremental MapReduce for mining evolving big data," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 7, pp. 1906–1919, Jul. 2015.

Leonidas Fegaras received the BTech degree in Electrical Engineering from the National Technical University of Athens, Greece, in 1985, the MS degree in Electrical and Computer Engineering from the University of Massachusetts-Amherst, in 1987, and the PhD degree in computer science from the University of Massachusetts-Amherst, in 1992. He is an associate professor in the Computer Science & Engineering Department, University of Texas at Arlington (UTA). Prior to joining UTA, he was a senior research scientist with the OGI School of Science and Engineering. His research interests include big data management, distributed computing, web data management, data stream processing, and query processing and optimization.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.