

# Incremental Query Processing on Big Data Streams

Leonidas Fegaras

University of Texas at Arlington

fegaras@cse.uta.edu

## ABSTRACT

This paper addresses online processing for large-scale, incremental computations on a distributed stream processing engine (DSPE). Our goal is to convert any distributed batch 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 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. Our methods can handle many forms of queries, including iterative and nested queries, group-by with aggregation, and joins on one-to-many relationships. Finally, we report on a prototype implementation of our framework using MRQL running on top of Spark and we experimentally validate the effectiveness of our methods.

## 1. INTRODUCTION

In recent years, large volumes of data are generated 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. 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are used extensively by search engines, need to recompute their Web graph measures very frequently since they become outdated very fast.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 are reused and merged with the results of processing the new data. Incremental data processing can generally achieve better performance than batch processing. In addition, recent emerging stream applications on Big Data need to handle real-time incremental computations on continuous streams of data with low latency, which is not feasible

with existing batch analysis tools. The Map-Reduce framework [9], 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 [34], and Apache Flink [15], which are in-memory distributed computing systems. There are also many emerging distributed stream processing engines (DSPEs) that process the data continuously, as they become available. Many of these systems 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 emit continuous results. To cope with blocking operations and unbounded memory requirements, some of these systems process continuous streams using sliding windows and incremental operators [2], 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 [40], Flink Streaming [15], Apache S4 [36], and Apache Samza [33].

This paper addresses online processing for large-scale, incremental computations on a distributed processing platform. Our goal is to convert any distributed batch data analysis program to an incremental distributed stream processing (DSP) program automatically, without requiring the user to rewrite this program. We are interested in deriving DSP programs that produce accurate results, rather than approximate answers. To accomplish this task, we would need to carefully analyze the 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 previous results with the new results of processing the incremental batches. Such analysis is hard to attain for programs written in an algorithmic programming language. Instead, this analysis 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], for their distributed data analysis applications, instead of coding them directly in an algorithmic language, such as Java. For instance, Hive is used for over 90% 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 would be highly 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 queries to incremental DSP programs. We have developed our framework on Apache MRQL [28], because it is both platform-independent and powerful enough to express complex data analysis tasks, such as PageRank, data clustering, and matrix factorization, using SQL-like syntax exclusively. 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 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.

**Our Approach.** 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 data set, followed by a continuous stream of incremental batches  $\Delta S_i$  that arrive at regular time intervals  $\Delta t$ . 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  $q(\bar{S})$ , where an  $S_i \in \bar{S}$  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 previous results) with the results of processing the incremental batches  $\Delta S_i$  only, rather than the entire stream  $S_i \uplus \Delta S_i$ , where  $\uplus$  is additive (bag) union. This is possible if  $q(\bar{S} \uplus \Delta \bar{S})$  can be expressed in terms of  $q(\bar{S})$  (the previous query result) and  $q(\Delta \bar{S})$  (the incremental query result), that is, when  $q(\bar{S})$  is a homomorphism<sup>1</sup> over  $\bar{S}$ . But some queries, such as counting the number of distinct elements in a stream or calculating average values after a group-by, are not homomorphisms. For such queries, we break  $q$  into two functions  $a$  and  $h$ , so that  $q(\bar{S}) = a(h(\bar{S}))$  and  $h$  is a homomorphism. The function  $h$  is a homomorphism if  $h(\bar{S} \uplus \Delta \bar{S}) = h(\bar{S}) \otimes h(\Delta \bar{S})$  for some monoid  $\otimes$  (an associative function with a zero element  $\otimes_z$ ). For example, the query that counts the number of distinct elements (a homomorphism), followed by the answer query  $a$  the 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 where  $a$  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 to compute the new result from the entire input,  $\bar{S} \uplus \Delta \bar{S}$ . On the other

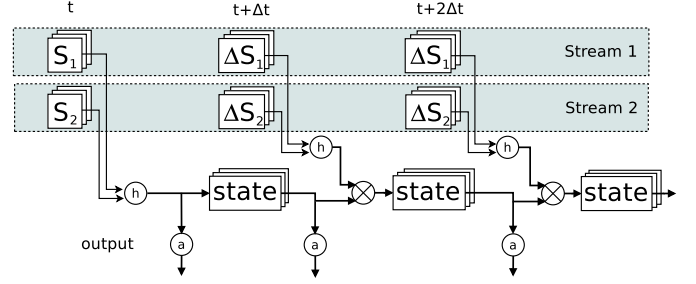


Figure 1: Incremental Query Processing

hand, in the special case where  $h$  is the identity function and  $\otimes$  is equal to  $\uplus$ , the output at each time interval can be simply taken to be only  $h(\Delta \bar{S})$ , which is the output we would expect to get from a sliding 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,  $\text{state} = h(\bar{S})$ . Then, at each time interval  $\Delta t$ , the query answer is calculated from the state, which becomes equal to  $h(\bar{S} \uplus \Delta \bar{S})$ :

```
state ← state ⊗ h(ΔS)
return a(state)
```

In Spark, for example, the state and the invariant data sources are stored in memory as Distributed DataSets (Spark's RDDs [39]) and are distributed across the worker nodes. However, the streaming data sources are implemented as Discretized Streams (Spark's D-Streams [40]), 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 would require a smaller state and impose less processing overhead to  $\otimes$ .

Figure 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:

**Task 1:** given a function  $f(\bar{S})$ , our system attempts to derive a monoid  $\otimes$ , such that  $f(\bar{S} \uplus \Delta \bar{S}) = f(\bar{S}) \otimes f(\Delta \bar{S})$ ;

**Task 2:** if Task 1 is unattainable, our system pulls out the non-homomorphic parts of function  $f$  outwards, and derives a new function  $h$ , such that  $h$  satisfies the conditions in Task 1 and  $f(\bar{S}) = a(h(\bar{S}))$ , for some answer function  $a$ .

Task 1 and Task 2 are, in general, hard to attain for a function  $f$  expressed in an algorithmic programming language. Instead, these tasks can become more tractable if they are performed on higher-order operations, such as the MRQL query algebra [13, 12]. 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 (Task 1), and for pulling non-homomorphic parts outside

<sup>1</sup>We use the term *homomorphism* throughout the paper as an abbreviation of *monoid homomorphism*.

a query (Task 2). Our methods can handle many forms of queries, including nested queries with any form and any number of nesting levels, group-by with aggregation, and joins on one-to-many/one-to-one relationships. They cannot handle non-equi-joins and joins on many-to-many relationships, as they are notoriously difficult to implement efficiently in a streaming or an incremental computing environment.

*Example.* For example, consider the following query (all queries are expressed in MRQL):

```
q(S1, S2) = select (x.A, avg(y.D))
from x in S1, y in S2
where x.B = y.C
group by x.A
```

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 behind this 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  $\Delta S_1$  and  $\Delta S_2$  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 is called *lineage tracking*, which has been used for consistent representation of uncertain data [3] and for propagating annotations in relational queries [4]. This lineage tracking in our case 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 result value. For our query, this is done by including the join key  $x.B$  as a group-by key. That is, in the following transformed query:

```
h(S1, S2) = select ((x.A, x.B), (sum(y.D), count(y.D)))
from x in S1, y in S2
where x.B = y.C
group by x.A, x.B
```

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  $x.A$  and  $x.B$ . It can be shown that 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  $N$  joins/group-bys/order-bys will be transformed to a query that injects the join/group-by/order-by keys to the output so that each output value is annotated with a chain of  $N$  keys. Note that the output of the query  $h$  is larger than that of the original query  $q$  because it creates more groups and each group is assigned two values (sum and count), instead of one (avg). This is expected since we extended  $q$  with lineage tracking. The answer query  $a$  that gives the final result in  $q(S_1, S_2) = a(h(S_1, S_2))$  is:

```
a(X) = select (k, sum(s)/sum(c))
from ((k,j),(s,c)) in X
group by k
```

that is, it removes the lineage  $j$  from  $X$  but also groups the result by the group-by key again and calculates the final avg

values. The merge function for the homomorphism  $h$  is an outer-join on the chain of keys that aggregates the matches. It is specified as:

```
X ⊗ Y = select (θ, (sx+sy, cx+cy))
from (θ, (sx, cx)) in X,
      (θ, (sy, cy)) in Y
union select x from x in X
where π1(x) not in π1(Y)
union select y from y in Y
where π1(y) not in π1(X)
```

where  $\theta$  matches the lineage  $(k, j)$  and  $\pi_1$  is pair and bag projection. This merge operation can be implemented efficiently as a partitioned join. Recall that this operation is used for merging the previous state with the results of processing the new data:

$$\text{state} \leftarrow \text{state} \otimes h(\overline{\Delta S})$$

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 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 coGroup to handle the next batch of data. Consequently, only the results of processing the new data, which are typically smaller than the state, would have to be shuffled across 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 new state partitions at the place they were generated. However, the new results would have to be partitioned and shuffled across the working nodes to be combined with the current state. This 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.

*Iteration.* Since repetitive queries, such as PageRank and k-means clustering, are very important in data analysis and mining, we have extended our methods to include repetition, so that these algorithms too can become incremental. A repetition can take the following general form:

```
X ← g( $\overline{S}$ )
for (i = 0; i < n; i++)
  X ← f(X,  $\overline{S}$ )
```

where  $X$  is the fixpoint of the repetition with initial value  $g(\overline{S})$ . That is,  $X = f^n(g(\overline{S}), \overline{S})$ , where  $f^n$  is  $n$  applications of  $f$ . Note that  $X$  is not necessarily a bag. For example, Gaussian Non-Negative Matrix Factorization [17], used in machine learning applications, such as for recommender systems, 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 for the previous repetition can only be derived if  $f$  is a homomorphism, a strict requirement that excludes many important iterative algorithms, such as PageRank and k-means clustering. Instead, we use

an approximate solution that works well for iterative queries that use monotonic aggregations. As before, we split  $f$  into a homomorphism  $h$  for some monoid  $\otimes$  and an answer function  $a$ , and we split  $g$  into a homomorphism  $h'$  for some monoid  $\oplus$  and an answer function  $a'$ . Our approximate solution maintains two states, one accurate state for the initial fixpoint value and one approximate state for the repetition result:

```

state' ← state'  $\oplus$  h'( $\overline{\Delta S}$ )
X ← a'(state')
for (i = 0; i < n; i++)
    ST ← state  $\otimes$  h(X,  $\overline{\Delta S}$ )
    X ← a(ST)
state ← ST

```

where, initially,  $\text{state} = \otimes_z$  and  $\text{state}' = \oplus_z$ . We can see that state remains unchanged during the loop, and is equal to the final state at the previous time interval. That is,  $\text{state} = h(X_{prev}, \overline{S})$ , where  $X_{prev}$  is the previous snapshot of  $X$ . This state is approximately equal to  $h(X, \overline{S})$ , because  $X$  is intended to be an improvement of  $X_{prev}$ . Given this approximation,  $ST \approx h(X, \overline{S} \oplus \Delta \overline{S})$ , which indicates that this solution will produce a similar answer as the original query over the entire stream.

For example, the following query implements the k-means clustering algorithm by repeatedly deriving  $k$  new centroids from the old:

```

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. As in the previous example, the average value of a bag of values is decomposed into a pair that contains the sum and the count of values. That is, the state is a bag of  $\{(k, ((sx, cx), (sy, cy)))\}$  so that the centroids are the bag of points  $\{(sx/cx, sy/cy)\}$  and  $k$  is the lineage (the group-by key), which is the X-Y coordinates of a centroid. Consequently, the answer query that returns the final result (the centroids) is:

```

a(state) = select < X: sx/cx, Y: sy/cy >
    from (k, ((sx, cx), (sy, cy))) in state

```

(it does not require a group-by since  $k$  is the only lineage key), while  $h(X, S_1)$  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]

```

The merge function  $\otimes$  is an outer-join, similar to the one of the previous example.

But, suppose that one of the group-by or join keys in a query is a floating point number. This is the case with the previous k-means clustering query, because it groups the points by their closest centroids, which contain floating point numbers. The group-by operation itself is not a problem because the centroids are fixed during the group-by. The problem arises when merging the current state with the results of processing the new data. For the k-means query example, the merge function  $\otimes$  is an equi-join whose join attribute is a centroid, so that the sum and count values associated with the same centroid are brought together from the join inputs and are accumulated. Since the join condition is over attributes with floating point numbers, the join condition will fail in most cases. This problem becomes even worse for the approximate solution, because it uses different sets of centroids  $X$  and  $X_{prev}$  when states are merged. Most iterative queries do not have this problem. The lineage in PageRank, for example, is the node ID, which remains invariant across iterations. For these uncommon queries, such as k-means, that have floating point numbers in their join/group-by/order-by attributes, we use yet another approximation: the join is done based on an “approximate equality” where two floating point numbers are taken to be equal if their difference is below some given threshold. This works well for our approximate solution for iteration because it is based on the assumption that the new solution  $X$  is approximately equal to the previous one,  $X_{prev}$ .

**Implementation and Contribution.** We have implemented our incremental processing framework on Apache MRQL [28] on top of Apache Spark Streaming [40], which is an in-memory distributed stream processing platform. 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.

The contribution of this work can be summarized as follows:

- We present a general automated method to convert any distributed data analysis query to an incremental stream processing program.
- 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 have extended our framework to handle deletions by using state increments to diminish the state in such a way that the query results would be the same as those we could have gotten on previous data after deletion.
- 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 three queries: groupBy, join-groupBy, and k-means clustering.

The rest of this paper is organized as follows. Section 2 compares our work with related work. Section 3 describes our earlier work on MRQL query processing. Section 4 defines our homomorphic algebraic operators used in the

MRQL algebra. Section 5 presents transformation rules for converting algebraic terms to a normal form that is easier to analyze. Section 6 describes an inference algorithm that statically infers the monoid of an algebraic term, if the term is a homomorphism. Section 7 describes our method for transforming algebraic terms to propagate all keys used in joins and group-bys to the query output. Section 8 describes an algorithm that pulls out the non-homomorphic parts of function outwards, deriving a homomorphism. Section 9, gives the detailed steps to derive the examples used in the Introduction. Section 10 extends our framework to handle data deletion. Section 11 gives some implementation details. Section 12 presents experiments that evaluate the performance of our incremental query processing techniques for three queries.

## 2. 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 [9] 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 between the map and reduce tasks of a Map-Reduce job or across consecutive jobs, which imposes a high overhead to complex workflows and graph algorithms, such as PageRank, which require repetitive Map-Reduce jobs. Recent systems for cloud computing use distributed memory for inter-node communication, such as the main memory Map-Reduce (M3R [32]), Apache Spark [34], Apache Flink [15], Piccolo [31], and distributed GraphLab [23]. Another alternative framework to the Map-Reduce model is the Bulk Synchronous Parallelism (BSP) programming model [38]. The best known implementations of the BSP model for data analysis on the cloud are Google’s Pregel [24], Apache Giraph [16], and Apache Hama [20].

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 [7]. MapReduce Online [8] 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. 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 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 [26] 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 [6] is a high throughput, low latency streaming query processor for temporal relational data, developed at Microsoft Research. The Reactive Aggregator [37], developed at IBM Research, is a new sliding-window streaming engine that performs many forms of sliding-window aggregation incrementally. In addition to these general data analysis engines, there are many data analysis algorithms that have been implemented incrementally, such as incremental pagerank [10]. Finally, 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 [18] for a literature survey).

Many novel Big Data stream processing systems, also known as distributed stream processing engines (DSPEs), have emerged recently. The most popular one is Twitter’s Storm [30], 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 [40], Flink Streaming [15], Apache S4 [36], and Apache Samza [33].

## 3. EARLIER WORK: MRQL

Apache MRQL [28] is a query processing and optimization system for large-scale, distributed data analysis [13, 12, 11]. It is currently an Apache incubating project. 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 [34], and in Flink mode using Apache Flink [15]. 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. 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.

A recent extension to MRQL, called MRQL streaming, supports the batch processing of continuous MRQL queries over streams of data. Currently, it works on Spark Streaming only but there are plans to add support for Flink Streaming and Storm in the future. Currently, MRQL supports window-based streaming based on a sliding window during a certain time interval. A stream source in MRQL takes the form `stream(...)`, which has the same parameters as the `source(...)` form. For example, the query:

```

select (k, avg(p.Y))
from p in stream(binary, "points")
group by k: p.X

```

processes all sequence files in the directory `points` and then checks this directory periodically for new files. When a new file is inserted in the directory (or if the modification time of an existing file changes), it processes the new files. One may work on multiple files and the query may contain both stream and regular data sources. If there is at least one stream source, the query becomes continuous (it never stops). The output of a continuous query is a sequence of files, one for each input batch, dumped into a directory.

## 4. THE MRQL ALGEBRA

Our 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 consists of a small number of higher-order homomorphic operators [13], which are defined using structural recursion based on the union representation of bags [14]. Monoid homomorphisms capture the essence of many divide-and-conquer algorithms and can be used as the basis for data parallelism [14].

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  $\alpha$  and  $\beta$ , the operation `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, this is expressed as:

$$\text{cMap}(f, X) \triangleq \{z \mid x \in X, z \in f(x)\} \quad (1)$$

or, alternatively, using structural recursion:

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

Given an arbitrary type  $\kappa$  that supports value equality ( $=$ ), an arbitrary type  $\alpha$ , and a bag  $X$  of type  $\{\kappa, \alpha\}$ , the operation `groupBy`( $X$ ) groups the elements of the bag  $X$  by their first component and returns a bag of type  $\{\kappa, \{\alpha\}\}$ . For example, `groupBy`( $\{(1, \text{"A"}), (2, \text{"B"}), (1, \text{"C"})\}$ ) returns  $\{(1, \{\text{"A"}, \text{"C"}\}), (2, \{\text{"B"}\})\}$ . The `groupBy` operation cannot be defined using a set former notation, but can be defined using structural recursion:

$$\begin{aligned} \text{groupBy}(X \uplus Y) &= \text{groupBy}(X) \uparrow_{\oplus} \text{groupBy}(Y) \\ \text{groupBy}(\{(k, a)\}) &= \{(k, \{a\})\} \\ \text{groupBy}(\{\}) &= \{\} \end{aligned}$$

where the parametric monoid  $\uparrow_{\oplus}$  merges groups associated with the same key using the monoid  $\oplus$  (equal to  $\uplus$  for `groupBy`):

$$\begin{aligned} X \uparrow_{\oplus} Y &\triangleq \{(k, a \oplus b) \mid (k, a) \in X, (k, b) \in Y\} \\ &\uplus \{(k, a) \mid (k, a) \in X, k \notin \pi_1(Y)\} \\ &\uplus \{(k, b) \mid (k, b) \in Y, k \notin \pi_1(X)\} \end{aligned} \quad (2)$$

where  $\pi_1(X) = \{k \mid (k, x) \in X\}$ . 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 using  $\pi_1(\text{groupBy}(\text{cMap}(\lambda x. \{(x, x)\}, X)))$ . Note also that `groupBy`( $X$ ) is not the same as the nesting  $\{(k, \{y \mid (k', y) \in X, k = k'\}) \mid (k, x) \in X\}$ , as the latter contains duplicate entries for the key  $k$ . Unlike nesting, unnesting a `groupBy` returns the input bag (proven in Appendix A):

$$\{(k, v) \mid (k, s) \in \text{groupBy}(X), v \in s\} = X \quad (3)$$

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

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

this term is not always a homomorphism on both inputs. Instead, MRQL provides a special homomorphic operation for equi-joins and outer-joins, `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  $\kappa$ , which returns a bag of type  $\{\kappa, (\{\alpha\}, \{\beta\})\}$ :

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

where the product of two monoids,  $\oplus \times \oplus$  is a monoid that, when applied to two pairs  $(x_1, x_2)$  and  $(y_1, y_2)$ , returns  $(x_1 \oplus y_1, x_2 \oplus y_2)$ . That is, the monoid  $\uparrow_{\oplus \times \oplus}$  merges two bags of type  $\{\kappa, (\{\alpha\}, \{\beta\})\}$  by unioning together their  $\{\alpha\}$  and  $\{\beta\}$  values that correspond to the same key  $\kappa$ . For example, `coGroup`( $\{(1, \text{"A"}), (2, \text{"B"}), (1, \text{"C"})\}, \{(1, \text{"D"}), (2, \text{"E"}), (3, \text{"F"})\}$ ) returns  $\{(1, (\{\text{"A"}, \text{"C"}\}, \{\text{"D"}\})), (2, (\{\text{"B"}\}, \{\text{"E"}\})), (3, (\{\}, \{\text{"F"}\}))\}$ . It can be proven (with a proof similar to that of Equation (3)) that both `coGroup` inputs can be derived from the `coGroup` result:

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

Aggregations are captured by the operation `reduce`( $\oplus, X$ ), which aggregates a bag  $X$  using a commutative monoid  $\oplus$ . For example, `reduce`( $+$ ,  $\{1, 2, 3\}$ ) = 6. This operation is in fact a general homomorphism that can be defined on any monoid  $\oplus$  with an identity function. We have:

$$\begin{aligned} \text{reduce}(\oplus, X) &= X \\ \text{reduce}(\uparrow_{\oplus}, X) &= \text{groupBy}(X) \end{aligned}$$

Finally, iteration `repeat`( $F, n, X$ ) over the bag  $X$  of type  $\{\alpha\}$  applies  $F$  of type  $\{\alpha\} \rightarrow \{\alpha\}$  to  $X$   $n$  times, yielding a bag of type  $\{\alpha\}$ :

$$\text{repeat}(F, n, X) = F(F(\dots F(X)))$$

An iteration is a homomorphism as long as  $F$  is a homomorphism, that is, when  $F(X \uplus Y) = F(X) \uplus F(Y)$ .

**Definition 1** (MRQL Algebra). *The MRQL algebra used throughout this paper consists of terms  $e$  that take the fol-*

lowing form:

$e ::=$	$\text{cMap}(f, e')$	<i>flatten-map</i>
	$\text{groupBy}(e')$	<i>group-by</i>
	$\text{coGroup}(e_1, e_2)$	<i>join</i>
	$\text{reduce}(\oplus, e')$	<i>aggregation</i>
	$(e_1, e_2)$	<i>pair</i>
	$S_i$	<i>a streaming input source</i>

where  $e'$ ,  $e_1$ , and  $e_2$  are also such terms. 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  $\text{cMap}$  in which the inner  $\text{cMap}$  is over a data source. This algebra does not include iteration,  $\text{repeat}(F, n, e)$ . Iterations have been discussed in the Introduction. In addition, for brevity, this algebra does not include terms for non-streaming input sources, general tuple and record construction and projection, bag union, singleton and empty bag, arithmetic operations, if-then-else expressions, boolean operations, etc.

In addition to these operations, there are a few more algebraic operations that can be expressed as homomorphisms, such as  $\text{orderBy}(X)$ , which is a  $\text{groupBy}$  followed by a sorting over the group-by key. This operation returns a list, which is represented by the non-commutative monoid list-append,  $++$ . Mixing multiple collection monoids and operations in the same algebra has been addressed by our previous work [14], and is left out from this paper to keep our analysis simple.

For example, the query  $q(S_1, S_2)$  used as the first example in the Introduction, is translated to the following algebraic term  $Q(S_1, S_2)$ :

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

where  $\text{avg}(s) = \text{reduce}(+, s) / \text{reduce}(+, \text{cMap}(\lambda v. \{1\}, s))$  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,  $\text{cMap}$  does not distribute over  $\uparrow_{\mathbb{B}}$ . One of our goals is to transform any composition of algebraic operations into a homomorphism.

## 5. QUERY NORMALIZATION

In an earlier work [13], we have presented general algorithms for unnesting nested queries. For example, consider the following nested query:

```
select x from x in X
where x.D > sum(select y.C from y in Y where x.A=y.B)
```

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  $\text{sum}(y.C)$ , and then to join the result with  $X$  on  $x.A=y.B$  using a right-outer join, removing all those matches whose  $x.D$  is below the sum. But, in our framework, this query is translated into:

```
cMap( λ(k,(xs,ys)). cMap( λx. if x.D > reduce(+,ys)
                        then {x}
                        else { }, xs),
    coGroup( cMap( λx. {(x.A,x)}, X ),
             cMap( λy. {(y.B,y.C)}, Y ) ) )
```

That is, the query unnesting is done with a left-outer join, which is captured concisely by the  $\text{coGroup}$  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 [13] for details).

The algebraic terms derived from MRQL queries can be normalized using the following rule:

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

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

$$\begin{aligned} &\text{cMap}(f, \text{cMap}(g, S)) \\ &= \{z \mid w \in \{y \mid x \in S, y \in g(x)\}, z \in f(w)\} \\ &= \{z \mid x \in S, y \in g(x), z \in f(y)\} \\ &= \{z \mid x \in S, z \in \{w \mid y \in g(x), w \in f(y)\}\} \\ &= \text{cMap}(\lambda x. \text{cMap}(f, g(x)), S) \end{aligned} \quad \square$$

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  $\text{cMap}$  between  $\text{groupBy}/\text{coGroup}$  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:*

$$\begin{aligned} q &::= (q_1, q_2) && \text{the query header} \\ &\quad | \text{reduce}(\oplus, \text{cMap}(f, e)) \\ &\quad | \text{cMap}(f, e) \\ e &::= \text{groupBy}(\text{cMap}(f, e')) && \text{the query body} \\ &\quad | \text{coGroup}(\text{cMap}(f_1, e_1), \text{cMap}(f_2, e_2)) \\ &\quad | S_i \end{aligned}$$

where the terms  $q_1$  and  $q_2$  have similar form as  $q$ , the terms  $e'$ ,  $e_1$ , and  $e_2$  have similar form as  $e$ , and  $f$ ,  $f_1$ , and  $f_2$  are anonymous functions that contain  $q$  terms. Here again, these functions should not contain any reference to a stream source,  $S_i$ .

The  $e$  terms form the *query body*, while the  $q$  term without the query body is called the *query header*. The query body is a tree of  $\text{groupBy}/\text{coGroup}$  operations connected via  $\text{cMaps}$ .

## 6. 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:

$$f(S_1 \uplus S'_1, \dots, S_n \uplus S'_n) = f(S_1, \dots, S_n) \otimes f(S'_1, \dots, S'_n) \quad (5)$$

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 homomorphism with a merge function  $\oplus$  under the environment  $\rho$ , 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 (5) can now be expressed as the judgment:

$$[S_1 : \mathbb{M}, \dots, S_n : \mathbb{M}] \vdash f(\bar{S}) : \otimes$$

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

$$X \square Y \triangleq \begin{cases} X & \text{if } Y = \square_z \text{ or } Y = X \\ Y & \text{if } X = \square_z \\ \text{error} & \text{otherwise} \end{cases}$$

where  $\square_z$  is the zero element of  $\square$  and depends on the type of the value. For example, for bags,  $\square_z$  is  $\{\}$ , which means that a term annotated with  $\square$  and returning a bag is either the empty bag or a bag that remains constant. We also use the annotation  $\square$  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  $\uparrow_{\square}$ , 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  $\{(k, s_1 \square s_2) \mid (k, s_1) \in X_1, (k, s_2) \in X_2\}$  in Definition (2) must have  $s_1 = s_2$  so that  $s_1 \square s_2 = s_1$ , otherwise it will be an error. This means that we can not have two different groups associated with the same key  $k$ . Given that a  $\text{groupBy}$  over a singleton gives a singleton group, each group returned from a  $\text{groupBy}$  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  $\text{coGroup}(X, Y)$  with the monoid  $\uparrow_{\square \times \mathbb{M}}$ , 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  $\kappa$ . This is because the  $X$  and  $Y$  values that are joined over the same key are merged with  $\square$  and  $\mathbb{M}$ , respectively, as indicated by  $\uparrow_{\square \times \mathbb{M}}$ .

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 denominator (below the line) contains the premises (separated by comma) and the numerator (above the line) is the conclusion. For example, the rule  $\frac{\rho \vdash f(x) : \otimes}{\rho \vdash x : \oplus}$  indicates that  $f(x_1 \oplus x_2) = f(x_1) \otimes f(x_2)$ . Figure 2 gives some of the inference rules. More rules will be given in Theorem 1. Rules (6d) through (6g) are derived directly from the algebraic definition of the operators. Rule (6a) retrieves the associated monoid of a variable  $v$  from the environment  $\rho$ . Rule (6b) indicates that if all the variables in a term  $e$  are invariant, then so is  $e$ . Rule (6h) indicates that all our joins are one-to-one or one-to-many. This is a very important restriction since many-to-many joins are not homomorphic. Rule (6i) indicates that a  $\text{cMap}$  over a  $\text{groupBy}$  returns a  $\text{groupBy}$  result as long as its functional argument is a  $\text{groupBy}$ . It can be proven as follows:

$$\begin{aligned} & \text{cMap}(f, X \uparrow_{\oplus} Y) \\ &= \{(\theta, z) \mid (k, x) \in X, (k, y) \in Y, (\theta, z) \in f(k, x \oplus y)\} \uplus \dots \\ &= \{(\theta, z) \mid (k, x) \in X, (k, y) \in Y, \\ & \quad (\theta, z) \in (f(k, x) \uparrow_{\otimes} f(k, y))\} \uplus \dots \end{aligned}$$

$$\begin{aligned} &= \{(\theta, z) \mid (k, x) \in X, (k, y) \in Y, \\ & \quad (\theta, z) \in \{(\theta, a \otimes b) \mid (\theta, a) \in f(k, x), \\ & \quad \quad (\theta, b) \in f(k, y)\}\} \uplus \dots \\ &= \{(\theta, a \otimes b) \mid (k, x) \in X, (k, y) \in Y, (\theta, a) \in f(k, x), \\ & \quad (\theta, b) \in f(k, y)\} \uplus \dots \\ &= \{(\theta, a \otimes b) \mid (\theta, a) \in \text{cMap}(f, X), (\theta, b) \in \text{cMap}(f, Y) \\ & \quad \uplus \dots \text{ (given that } k \text{ and } \theta \text{ are unique keys)}\} \\ &= \text{cMap}(f, X) \uparrow_{\otimes} \text{cMap}(f, Y) \end{aligned}$$

## 7. INJECTING LINEAGE TRACKING

There are two tasks that need to be accomplished to achieve our goal of transforming an algebraic term into a homomorphism: 1) transform the algebraic term in such a way that it propagates all keys used in joins and group-bys to the query output, and 2) pull the non-homomorphic parts out of the algebraic term so that it becomes a homomorphism. In this section, we address the first task.

We will transform the algebraic terms given in Definition 2 in such a way that they propagate the join and the group-by keys. That is, each value  $v$  returned by these terms is annotated with a lineage  $\theta$ , as a pair  $(\theta, v)$ , that takes the following form:

$$\theta ::= \begin{pmatrix} \kappa, \theta' \\ | \quad () \end{pmatrix}$$

which is the list of the join and group-by keys  $\kappa$  that are used in deriving the value (one key for each  $\text{groupBy}$  and  $\text{coGroup}$  operation). More specifically, if the output of a query  $q$  is  $\{t\}$  for some type  $t$ , then the transformed query will have output  $\{(\theta, t)\}$ . On the other hand, if the output is a non-collection type  $t$ , then the transformed query will also have output  $\{(\theta, t)\}$ , which separates the contributions to  $t$  for each combination of group-by/join keys. The query translation is accomplished by converting every  $\text{cMap}$  in a term to  $\text{cMap1}$ ,  $\text{cMap2}$ , or  $\text{cMap3}$ , where:

$$\text{cMap1}(f, X) \triangleq \{((k, \theta), b) \mid ((k, \theta), a) \in X, b \in f(k, a)\} \quad (7a)$$

$$\text{cMap2}(f, X) \triangleq \{((k', (k, \theta)), b) \mid ((k, \theta), a) \in X, (k', b) \in f(k, a)\} \quad (7b)$$

$$\text{cMap3}(f, X) \triangleq \{((k, ()), b) \mid a \in X, (k, b) \in f(a)\} \quad (7c)$$

The  $\text{cMap1}$  operation is a  $\text{cMap}$  that propagates the input lineage  $\theta$  to the output as is. The  $\text{cMap2}$  operation is a  $\text{cMap}$  that extends the input lineage  $(k, \theta)$  with the output key  $k'$ . To annotate an algebraic term  $q$  defined in Definition 2 with lineage  $\theta$ , the term is transformed into  $\mathcal{T}_q[q]$ :

$$\mathcal{T}_q[(q_1, q_2)] = (\mathcal{T}_q[q_1], \mathcal{T}_q[q_2]) \quad (8a)$$

$$\begin{aligned} & \mathcal{T}_q[\text{reduce}(\oplus, \text{cMap}(f, S_i))] \\ &= \{(((), \text{reduce}(\oplus, \text{cMap}(f, S_i))))\} \end{aligned} \quad (8b)$$

$$\begin{aligned} & \mathcal{T}_q[\text{reduce}(\oplus, \text{cMap}(f, e))] \\ &= \text{reduce}(\uparrow_{\oplus}, \text{cMap1}(f, \mathcal{T}_e[e])) \end{aligned} \quad (8c)$$

$$\mathcal{T}_q[\text{cMap}(f, S_i)] = \{(((), b) \mid a \in S_i, b \in f(a))\} \quad (8d)$$

$$\mathcal{T}_c[\text{cMap}(f, e)] = \text{cMap1}(f, \mathcal{T}_e[e]) \quad (8e)$$

$$\mathcal{T}_e[\text{groupBy}(c)] = \text{groupBy}(\mathcal{T}_c[c]) \quad (8f)$$

$$\mathcal{T}_e[\text{coGroup}(c_1, c_2)] = \text{coGroup}(\mathcal{T}_c[c_1], \mathcal{T}_c[c_2]) \quad (8g)$$

$$\mathcal{T}_c[\text{cMap}(f, S_i)] = \text{cMap3}(f, S_i) \quad (8h)$$

$$\mathcal{T}_c[\text{cMap}(f, e)] = \text{cMap2}(f, \mathcal{T}_e[e]) \quad (8i)$$



$\rho \vdash v : \rho(v)$	(6a)		
$\frac{\rho \vdash e : \square}{\forall v \in e : \rho \vdash v : \square}$	(6b)	$\frac{\rho \vdash \text{cMap}(f, X) : \uplus}{\rho \vdash X : \uplus, \rho[v : \square] \vdash f(v) : \square}$	(6f)
$\frac{\rho \vdash (x, y) : \oplus \times \otimes}{\rho \vdash x : \oplus, \rho \vdash y : \otimes}$	(6c)	$\frac{\rho \vdash \text{groupBy}(X) : \uparrow_{\uplus}}{\rho \vdash X : \uplus}$	(6g)
$\frac{\rho \vdash \text{reduce}(\oplus, X) : \oplus}{\rho \vdash X : \uplus}$	(6d)	$\frac{\rho \vdash \text{coGroup}(X, Y) : \uparrow_{\square \times \uplus}}{\rho \vdash X : \uplus, \rho \vdash Y : \uplus}$	(6h)
$\frac{\rho \vdash (X \uplus Y) : \uplus}{\rho \vdash X : \uplus, \rho \vdash Y : \uplus}$	(6e)	$\frac{\rho \vdash \text{cMap}(f, X) : \uparrow_{\otimes}}{\rho \vdash X : \uparrow_{\oplus}, \rho[k : \square, s : \oplus] \vdash f(k, s) : \uparrow_{\otimes}}$	(6i)

Figure 2: Monoid Inference Rules

Rule (8b) annotates every value of the input stream  $S_i$  with the empty lineage  $()$ . Rule (8c) indicates that total aggregations become group-by aggregations by aggregating the values of each group (associated with a different lineage  $\theta$ ).

We first prove that  $\text{cMap1}$ ,  $\text{cMap2}$ , and  $\text{cMap3}$  are homomorphisms, provided that their functional arguments are homomorphisms. More specifically, we prove the following judgments:

**Theorem 1** ( $\text{cMap1}$ ,  $\text{cMap2}$ , and  $\text{cMap3}$  Judgments).

$$\frac{\rho \vdash \text{cMap3}(f, X) : \uplus}{\rho \vdash X : \uplus, \rho[v : \square] \vdash f(v) : \square} \quad (9a)$$

$$\frac{\rho \vdash \text{cMap1}(\lambda(k, v). \{g(k, v)\}, X) : \uparrow_{\otimes}}{\rho \vdash X : \uparrow_{\uplus}, \rho[k : \square, v : \uplus] \vdash g(k, v) : \otimes} \quad (9b)$$

$$\frac{\rho \vdash \text{cMap1}(\lambda(k, (x, y)). \{g(k, (x, y))\}, X) : \uparrow_{\otimes}}{\rho \vdash X : \uparrow_{\square \times \uplus}, \rho[k : \square, x : \square, y : \uplus] \vdash g(k, (x, y)) : \otimes} \quad (9c)$$

$$\frac{\rho \vdash \text{cMap2}(f, X) : \uplus}{\rho \vdash X : \uparrow_{\uplus}, \rho[k : \square, v : \uplus] \vdash f(k, v) : \uplus} \quad (9d)$$

$$\frac{\rho \vdash \text{cMap2}(f, X) : \uplus}{\rho \vdash X : \uparrow_{\square \times \uplus}, \rho[k : \square, x : \square, y : \uplus] \vdash f(k, (x, y)) : \uplus} \quad (9e)$$

The proof is given in Appendix A. The  $\text{cMap1}$  in Judgement (9d) is over a  $\text{groupBy}$  (since  $X$  is annotated with  $\uparrow_{\uplus}$ ), and the  $\text{cMap1}$  in Judgement (9e) is over a  $\text{coGroup}$  that represents an one-to-many or many-to-many join (since  $X$  is annotated with  $\uparrow_{\square \times \uplus}$ ).

**Definition 3** (Query Merger Monoid). *The query merger monoid  $\mathcal{M}[q]$  of the query  $q$ , defined in Definition 2, is derived as follows:*

$$\begin{aligned} \mathcal{M}[(q_1, q_2)] &= \mathcal{M}[q_1] \times \mathcal{M}[q_2] \\ \mathcal{M}[\text{reduce}(\oplus, \text{cMap}(f, e))] &= \uparrow_{\oplus} \\ \mathcal{M}[\text{cMap}(f, S_i)] &= \uplus \\ \mathcal{M}[\text{cMap}(f, e)] &= \uparrow_{\otimes} \end{aligned}$$

where the monoid  $\uparrow_{\otimes}$  in the last equation (for  $e \neq S_i$ ) comes from  $\text{cMap1}(f, X) : \uparrow_{\otimes}$  in Equations (9b) and (9c).

**Definition 4** (Query Answer). *The query answer  $\mathcal{A}[q]_x$  of the query  $q$ , defined in Definition 2, is a function over  $x$  (the*

*current state*) derived as follows:

$$\begin{aligned} \mathcal{A}[(q_1, q_2)]_x &= (\mathcal{A}[q_1]_{\pi_1(x)}, \mathcal{A}[q_2]_{\pi_2(x)}) \\ \mathcal{A}[\text{reduce}(\oplus, \text{cMap}(f, e))]_x &= \text{reduce}(\oplus, \pi_2(x)) \\ \mathcal{A}[\text{cMap}(f, S_i)]_x &= \pi_2(x) \\ \mathcal{A}[\text{cMap}(f, e)]_x &= \pi_2(\text{reduce}(\uparrow_{\otimes}, T(x))) \end{aligned}$$

where the monoid  $\uparrow_{\otimes}$  in the last equation (for  $e \neq S_i$ ) comes from  $\text{cMap1}(f, X) : \uparrow_{\otimes}$  in Equations (9b) and (9c) and

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

The last reduce is actually a group-by with aggregation:

$$\text{reduce}(\uparrow_{\otimes}, S) = \text{cMap}(\lambda(k, s). \{(k, \text{reduce}(\otimes, s))\}, \text{groupBy}(S))$$

The following theorems prove that  $\mathcal{M}[q]$  and  $\mathcal{A}[q]_x$  have the desired properties:

**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:*

$$\mathcal{A}[q]_x = q \quad \text{for } x = \mathcal{T}_q[q] \quad (10)$$

**Theorem 3** (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  $\text{cMap1}$ ,  $\text{cMap2}$ , and  $\text{cMap3}$  term in  $\mathcal{T}_q[q]$  satisfies the premises of the judgments in Theorem 1:*

$$\rho[S_1 : \uplus, \dots, S_n : \uplus] \vdash \mathcal{T}_q[q] : \mathcal{M}[q] \quad (11)$$

The proofs of these theorems are given in Appendix A.

## 8. NON-HOMOMORPHIC TERMS

Theorem 2 indicates that the terms generated by the transformations (8a) through (8i) are homomorphisms as long as the premises of the judgements in Theorem 1 are true. These premises indicate that the  $\text{cMap}$  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 with the help of  $\text{kMap}$ :

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

which is a  $\text{cMap}$  that propagates the lineage  $\theta$  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  $\theta'$  from the old  $\theta$ . In our framework, we find the largest subterms in the algebraic term  $e$ , namely  $e_1, \dots, e_n$ , 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 is inferred to be a homomorphism. If it is, the node is replaced with a new variable. Thus,  $e$  is mapped to a term  $f(e_1, \dots, e_n)$ , for some term  $f$ , and the terms  $e_1, \dots, e_n$  are replaced with variables when  $f$  is pulled outwards:

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

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

$$\begin{aligned} & \text{cMap2}(g, \text{kMap}(f, X)) \rightarrow \text{cMap2}(\lambda(k, v). g(k, f(v)), X) \\ & \text{cMap1}(g, \text{kMap}(f, X)) \rightarrow \text{cMap1}(\lambda(k, v). g(k, f(v)), X) \\ & \text{groupBy}(\text{kMap}(f, X)) \rightarrow \text{kMap}(\lambda s. \text{map}(f, s), \text{groupBy}(X)) \\ & \text{coGroup}(\text{kMap}(f_x, X), \text{kMap}(f_y, Y)) \\ & \rightarrow \text{kMap}(\lambda(xs, ys). (\text{map}(f_x, xs), \text{map}(f_y, ys))), \\ & \quad \text{coGroup}(X, Y)) \end{aligned}$$

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.

## 9. AN EXAMPLE

Consider again the algebraic term  $Q(S_1, S_2)$ , presented at the end of Section 4. If we apply the transformations in Equations (8a) through (8i), 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:

$$\begin{aligned} & \text{cMap1}(\lambda(k, s). \{(k, \text{avg}(s))\}, \\ & \quad \text{groupBy}(\text{cMap2}(\lambda(j, (xs, ys)). g(xs, ys), \\ & \quad \quad \text{coGroup}(\text{cMap3}(\lambda x. \{(x.B, x)\}, S_1), \\ & \quad \quad \text{cMap3}(\lambda y. \{(y.C, y)\}, S_2)))))) \end{aligned}$$

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

We now check if the transformed query is a homomorphism. From Equation (9a), both coGroup inputs are annotated with  $\uplus$ , which means that, based on Equation (6h), coGroup is annotated with  $\uparrow_{\square \times \uplus}$ . From Equation (9d), the cMap2 operation is annotated with  $\uplus$  as long as  $g(xs, ys)$  is annotated with  $\uplus$ . Hence, the groupBy operation is annotated with  $\uparrow_{\uplus}$ , based on Equation (6g). Unfortunately, based on Equation (9c), the cMap1 term is not a homomorphism as is, because  $\text{avg}(s)$  is not a homomorphism over  $s$ . Based on the methods described in Section 8 that factor out non-homomorphic parts from terms, the cMap1 term is

broken into two terms  $a(h(S_1, S_2))$ , where  $h(S_1, S_2)$  is:

$$\text{cMap1}(\lambda(k, s). \{(k, (\text{reduce}(+, s), \text{reduce}(+, \text{cMap}(\lambda v. 1, s))))\}, \dots)$$

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

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

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

This is equivalent to the answer query given in the Introduction. Finally,  $h(S_1, S_2)$  is a homomorphism annotated with  $\uparrow_{\square \times ((+) \times (+))}$ , because the reduce terms are annotated with  $(+)$  (from Equation (6d)) and we have a pair of homomorphisms (Equation (6c)). This is equivalent to the merge function  $X \otimes Y$  given in the Introduction, implemented as a partitioned join combined with aggregation.

## 10. HANDLING DELETIONS

Our framework can be easily extended to handle deletion of existing data from an input stream, in addition to insertion of new data. To handle both insertions and deletions, a data stream  $S_i$  in our framework consists of an initial data set, followed by a continuous stream of incremental batches  $\Delta S_i$  and a continuous stream of decremental batches  $\Delta S'_i$  that arrive at regular time intervals  $\Delta t$ . Then, the stream data at time  $t + \Delta t$  is  $S_i \uplus \Delta S_i - \Delta S'_i$ , where  $X - Y$  is bag difference, which satisfies  $(X \uplus Y) - Y = X$ . (An element appears in  $X - Y$  as many times as it appears in  $X$ , minus the number of times it appears in  $Y$ .) This dual stream of updates, can be implemented as a single stream that contains updates tagged with  $+$  or  $-$ , to indicate insertion or deletion. Alternatively, a stream data source may monitor two separate directories, one for insertions and another for deletions, so that if a new file is created, it will be treated as a new batch of insertions or deletions, depending on the directory. In our framework, we require that  $\Delta S'_i \subseteq S_i$ , that is, we can only delete values that appear in the stream in larger or equal multiplicities. This restriction implies that there is a bag  $W_i$  such that  $S_i = W_i \uplus \Delta S'_i$ . Without this restriction, it would be hard to diminish the query results on  $S_i$  by the results on  $\Delta S_i$  to calculate the new results.

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

$$\begin{aligned} \widehat{\uplus} &= - & \widehat{+} &= - & \widehat{\square} &= \square \\ \widehat{\otimes} \times \widehat{\oplus} &= \widehat{\otimes} \times \widehat{\oplus} & \widehat{\uparrow}_{\oplus} &= \downarrow_{\oplus} \end{aligned}$$

where  $\downarrow_{\otimes}$  is defined as follows:

$$\begin{aligned} X \downarrow_{\otimes} Y &\triangleq \{ (k, a \otimes b) \mid (k, a) \in X, (k, b) \in Y, a \neq b \} \\ &\quad \uplus \{ (k, a) \mid (k, a) \in X, k \notin \pi_1(Y) \} \end{aligned}$$

Note that a diminisher is not a monoid.

**Theorem 4.** *For all  $X, Y : (X \otimes Y) \widehat{\otimes} Y = X$ .*

The proof is given in Appendix A. For  $X = \otimes_z$ , it implies  $Y \widehat{\otimes} Y = \otimes_z$ .

**Theorem 5.** *If  $[S_1 : \uplus, \dots, S_n : \uplus] \vdash f(\overline{S}) : \otimes$  and for all  $i : \Delta S'_i \subseteq S_i$ , then  $f(\overline{S - \Delta S'}) = f(\overline{S}) \widehat{\otimes} f(\overline{\Delta S'})$ .*

*Proof.* Since  $\Delta S'_i \subseteq S_i$ , then there must be  $W_i$  such that  $S_i = W_i \uplus \Delta S'_i$ . Thus,  $S_i - \Delta S'_i = W_i \uplus \Delta S'_i - \Delta S'_i \Rightarrow W_i = S_i - \Delta S'_i$ . Then,

$$\begin{aligned} f(\overline{S}) \widehat{\otimes} f(\overline{\Delta S'}) &= f(\overline{W \uplus \Delta S'}) \widehat{\otimes} f(\overline{\Delta S'}) \\ &= (f(\overline{W}) \otimes f(\overline{\Delta S'})) \widehat{\otimes} f(\overline{\Delta S'}) = f(\overline{W}) = f(\overline{S - \Delta S'}) \quad \square \end{aligned}$$

This theorem indicates that, to process the deletions  $\overline{\Delta S'}$ , we can simply use the same methods as for insertions, but using the diminishing function  $\widehat{\otimes}$  for merging the state, instead of  $\otimes$ . That is, we can use the same homomorphism  $h$  and answer function  $a$ , but now the state is diminished as follows:

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

For example, the join-groupby query used in the Introduction must now use the merging,  $X \uparrow_{(-) \times (-)} Y$ , equal to:

```
select ( $\theta$ , ( $sx-sy, cx-cy$ ))
from ( $\theta, (sx, cx)$ ) in  $X$ ,
      ( $\theta, (sy, cy)$ ) in  $Y$ 
where  $sx \neq sy$  and  $cx \neq cy$ 
union select  $x$  from  $x$  in  $X$  where  $\pi_1(x)$  not in  $\pi_1(Y)$ 
```

## 11. IMPLEMENTATION

We have implemented our incremental processing framework on Apache MRQL [28] on top of Apache Spark Streaming [40]. 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,  $\text{state}_i$ , of type  $\alpha_i$ . The operation  $\text{Incr}(i, s_0, F)$ , where  $F$  is a state transition function of type  $\alpha_i \rightarrow \alpha_i$  and  $s_0$  is the initial state of type  $\alpha_i$ , has the following semantics:

$$\begin{aligned} \text{state}_i &\leftarrow F(\text{state}_i) \\ \text{return } &\text{state}_i \end{aligned}$$

with  $\text{state}_i = s_0$ , initially. Note that the states are preserved across the *Inc* calls and are modified by these calls.

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:

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

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

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

For an iteration  $\text{repeat}(\lambda X. f(X, \overline{\Delta S}), n, g(\overline{S}))$ , we use the approximate solution, as described in the Introduction: we split  $f$  into a homomorphism  $h$  for some monoid  $\otimes$  and a function  $a$ , and we split  $g$  into a homomorphism  $h'$  for some monoid  $\oplus$  and a function  $a'$ . Then, the approximate solution

is implemented using the following physical plan:

$$\begin{aligned} &a(\text{Incr}(1, \otimes_z, \\ &\quad \lambda s. \text{repeat}(\lambda s'. s \otimes h(a(s'), \overline{\Delta S}), \\ &\quad \quad n - 1, \\ &\quad \quad s \otimes h(a'(\text{Incr}(2, \oplus_z, \\ &\quad \quad \quad \lambda s''. s'' \oplus h'(\overline{\Delta S}))))); \\ &\quad \overline{\Delta S}))) \end{aligned}$$

That is, the first iteration step is unrolled into the initial state of repeat to handle the case where  $X = a'(\text{state}')$  in the first call to  $h$ .

## 12. PERFORMANCE EVALUATION

The system described in this paper is available as part of the latest MRQL snapshot release (filed as a JIRA issue *MRQL-79: Add support for incremental query processing*) and is scheduled to be in the next official MRQL release soon. We have experimentally validated the effectiveness of our methods using three queries: *groupBy*, *join-groupBy*, and *k-means clustering*. The platform used for our evaluations is a small cluster of 9 Linux servers, connected through a Gigabit Ethernet switch. Each server has 4 Xeon cores at 3.2GHz with 4GB memory. For our experiments, we used Hadoop 2.2.0 (Yarn) and Spark 1.5.1. The cluster frontend 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 32 cores of the compute nodes for Spark tasks. The scripts to reproduce the experiments are available at <http://lambda.uta.edu/stream-results.zip>.

The data streams used in our experiments 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 experiments were repeated for increments of size 10K, 20K, 30K, and 40K tuples, always starting with a fresh state (constructed from the initial data only). The performance results are shown in Figure 3. The  $x$ -axis represents the time points  $\Delta t$  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 9 increments arrive at the time points  $1\Delta t$  through  $9\Delta t$ . The  $y$ -axis is the query execution time, and there are 4 plots, one for each increment size.

The *join-groupBy* and the *k-means clustering* queries are given in the Introduction. 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 10000. The *groupBy* initial dataset has size 1M tuples, while the two *join-groupBy* inputs have size 100K tuples. The datasets used for the *k-means* query consist of random  $(X, Y)$  points in 4 squares that have  $X$  in  $[2, 4]$  or  $[6, 8]$  and  $Y$  in  $[2, 4]$  or  $[6, 8]$ . Thus, the 4 centroids are expected to be  $(3, 3)$ ,  $(3, 7)$ ,  $(7, 3)$ , and  $(7, 7)$ . This also means that the state contains 4 centroids only. The initial dataset for *k-means* contains 1M points and the *k-means* query uses 10 iteration steps. The *k-means* incremental program uses the approximate solution described in the Introduction based on approximate floating point equality.

From Figure 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. Further-

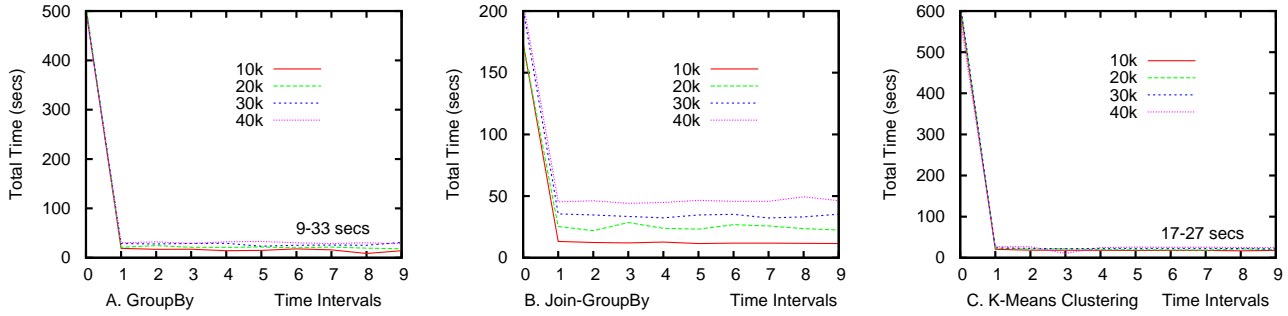


Figure 3: Incremental Query Evaluation of GroupBy, Join-GroupBy, and K-Means Clustering

more, 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 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 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 with 100K nodes and 1M edges, generated by the R-MAT algorithm using the Kronecker graph generator parameters:  $a=0.30$ ,  $b=0.25$ ,  $c=0.25$ , and  $d=0.20$ . (The MRQL PageRank query is given in [13].) As before, we used increments of sizes 10K-40K. The query used 10 iteration steps to converge. The results were disappointing: the initial data processing took 6495 secs, while the processing of each increment took about the same time (6150 secs). We believe that the reason for this performance is that the loop fixpoint  $X$  in the query contains the entire graph (each node in the state, in addition to the PageRank value, has a list of incoming neighbors), which results to large state increments (since the state depends on  $X$ ). We are planning to improve this query so that  $X$  will contain PageRank values only, while the new  $X$  is calculated by joining the current  $X$  with the original graph.

### 13. 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. 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. General distributed stream processing platforms, though, such as Storm, Spark, and Flink, cannot benefit from our techniques, because such techniques require program analysis, which is hard for non-declarative programs.

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

### 14. 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] 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.
- [3] 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.
- [4] 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.
- [5] 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.
- [6] 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.
- [7] 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.
- [8] 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.
- [9] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Symposium on Operating System Design and Implementation (OSDI)*, 2004.
- [10] P. Desikan, N. Pathak, J. Srivastava, and V. Kumar

- Incremental PageRank Computation on Evolving Graphs. In *International conference on World Wide Web (WWW)*, pages 1094–1095, 2005.
- [11] L. Fegaras. Supporting Bulk Synchronous Parallelism in Map-Reduce Queries. In *International Workshop on Data Intensive Computing in the Clouds (DataCloud)*, 2012.
- [12] 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.
- [13] 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.
- [14] L. Fegaras and D. Maier. Optimizing Object Queries Using an Effective Calculus. In *ACM Transactions on Database Systems (TODS)*, 25(4):457–516, 2000.
- [15] Apache Flink. <http://flink.apache.org/>.
- [16] Apache Giraph. <http://giraph.apache.org/>.
- [17] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhvani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative Machine Learning on MapReduce. In *IEEE International Conference on Data Engineering (ICDE)*, pages 231–242, 2011.
- [18] 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.
- [19] Apache Hadoop. <http://hadoop.apache.org/>.
- [20] Apache Hama. <http://hama.apache.org/>.
- [21] Apache Hive. <http://hive.apache.org/>.
- [22] 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.
- [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. In *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.
- [24] 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.
- [25] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential Dataflow. In *Conference on Innovative Data System Research (CIDR)*, 2013.
- [26] 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.
- [27] 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.
- [28] Apache MRQL (incubating). <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 *ACM SIGMOD International Conference on Management of Data*, pages 1099–1110, 2008.
- [30] D. Peng and F. Dabek. Large-scale Incremental Processing using Distributed Transactions and Notifications. In *Symposium on Operating System Design and Implementation (OSDI)*, 2010.
- [31] R. Power and J. Li. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *Symposium on Operating System Design and Implementation (OSDI)*, 2010.
- [32] A. Shinnar, D. Cunningham, B. Herta, and V. Sarawat. M3R: Increased Performance for In-Memory Hadoop Jobs In *Proceedings of the VLDB Endowment*, 5(12):1736–1747, 2012.
- [33] Apache Samza. <http://samza.apache.org/>.
- [34] Apache Spark. <http://spark.apache.org/>.
- [35] Apache Storm: A System for Processing Streaming Data in Real Time. <http://hortonworks.com/hadoop/storm/>.
- [36] Apache S4 (incubating): A Distributed Stream Computing Platform. <http://incubator.apache.org/s4/>.
- [37] 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.
- [38] L. G. Valiant. A Bridging Model for Parallel Computation. In *Communications of the ACM (CACM)*, 33(8):103–111, August 1990.
- [39] 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.
- [40] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

## APPENDIX

### A. PROOFS

*Proof of Equation (3).* We will use structural induction. The inductive step

$$\{(k, v) \mid (k, s) \in \text{groupBy}(X \uplus Y), v \in s\} = X \uplus Y$$

can be proven as follows. Let  $GX = \text{groupBy}(X)$  and  $GY = \text{groupBy}(Y)$ . Then, for

$$M(X, Y) = \{(k, a) \mid (k, a) \in X, (k, b) \in Y\} \uplus \{(k, a) \mid (k, a) \in X, k \notin \pi_1(Y)\} \quad (12)$$

we have,  $M(GX, GY) = GX$ , because for each group-by key  $k$ , there is at most one  $(k, a) \in GX$  and at most one  $(k, b) \in GY$ . Then, we have:

$$\begin{aligned} & \{(k, v) \mid (k, s) \in \text{groupBy}(X \uplus Y), v \in s\} \\ &= \{(k, v) \mid (k, s) \in (GX \uparrow_{\uplus} GY), v \in s\} \\ &= \{(k, v) \mid (k, s) \in (\{(k, a \uplus b) \mid (k, a) \in GX, (k, b) \in GY\} \\ & \quad \uplus \{(k, a) \mid (k, a) \in GX, k \notin \pi_1(GY)\}) \\ & \quad \uplus \{(k, b) \mid (k, b) \in GY, k \notin \pi_1(GX)\}), v \in s\} \\ &= \{(k, v) \mid (k, a) \in GX, (k, b) \in GY, v \in (a \uplus b)\} \\ & \quad \uplus \{(k, v) \mid (k, a) \in GX, k \notin \pi_1(GY), v \in a\} \\ & \quad \uplus \{(k, v) \mid (k, b) \in GY, k \notin \pi_1(GX), v \in b\} \\ &= \{(k, v) \mid (k, a) \in GX, (k, b) \in GY, v \in a\} \\ & \quad \uplus \{(k, v) \mid (k, a) \in GX, (k, b) \in GY, v \in b\} \\ & \quad \uplus \{(k, v) \mid (k, a) \in GX, k \notin \pi_1(GY), v \in a\} \\ & \quad \uplus \{(k, v) \mid (k, b) \in GY, k \notin \pi_1(GX), v \in b\} \\ &= \{(k, v) \mid (k, a) \in M(GX, GY), v \in a\} \\ & \quad \uplus \{(k, v) \mid (k, b) \in M(GY, GX), v \in b\} \\ &= \{(k, v) \mid (k, a) \in GX, v \in a\} \\ & \quad \uplus \{(k, v) \mid (k, b) \in GY, v \in b\} \\ &= X \uplus Y \quad (\text{from induction hypothesis}) \quad \square \end{aligned}$$

*Proof of Theorem 1 (cMap1, cMap2, and cMap3 Judgments).* For  $f(v) = \{g(v)\}$ , we have:

$$\begin{aligned} \text{cMap1}(f, X) &= \{((\theta, k), b) \mid ((\theta, k), a) \in X, b \in f(a)\} \\ &= \{((\theta, k), g(k, a)) \mid ((\theta, k), a) \in X\} \end{aligned}$$

Then, Judgement (9b) can be proven as follows:

$$\begin{aligned} & \text{cMap1}(f, X \uparrow_{\uplus} Y) \\ &= \{((\theta, k), g(k, c)) \mid ((\theta, k), c) \in (\{((\theta, k), a \uplus b) \mid ((\theta, k), a) \in X, \\ & \quad ((\theta, k), b) \in Y\} \uplus \dots)\} \\ &= \{((\theta, k), g(k, a \uplus b)) \mid ((\theta, k), a) \in X, ((\theta, k), b) \in Y\} \uplus \dots \\ &= \{((\theta, k), g(k, a) \otimes g(k, b)) \mid ((\theta, k), a) \in X, ((\theta, k), b) \in Y\} \\ & \quad \uplus \dots \\ &= \{((\theta, k), x \otimes y) \mid ((\theta, k), x) \in \text{cMap1}(f, X), \\ & \quad ((\theta, k), y) \in \text{cMap1}(f, Y)\} \uplus \dots \\ &= \text{cMap1}(f, X) \uparrow_{\otimes} \text{cMap1}(f, Y) \end{aligned}$$

Judgement (9c) can be proven as follows:

$$\begin{aligned} & \text{cMap1}(f, X \uparrow_{\square \times \uplus} Y) \\ &= \text{cMap1}(f, \{((k, \theta), (s_1 \square s'_1, s_2 \uplus s'_2)) \mid ((k, \theta), (s_1, s_2)) \in X, \\ & \quad ((k, \theta), (s'_1, s'_2)) \in Y\} \uplus \dots) \\ &= \{((k, \theta), g(k, (s_1 \square s'_1, s_2 \uplus s'_2))) \mid ((k, \theta), (s_1, s_2)) \in X, \\ & \quad ((k, \theta), (s'_1, s'_2)) \in Y\} \uplus \dots \\ &= \{((k, \theta), g(k, (s_1, s_2)) \otimes g(k, (s'_1, s'_2))) \mid ((k, \theta), (s_1, s_2)) \in X, \\ & \quad ((k, \theta), (s'_1, s'_2)) \in Y\} \uplus \dots \\ &= \{((k, \theta), x \otimes y) \mid ((k, \theta), x) \in \text{cMap1}(f, X), \\ & \quad ((k, \theta), y) \in \text{cMap1}(f, Y)\} \uplus \dots \\ &= \text{cMap1}(f, X) \uparrow_{\otimes} \text{cMap1}(f, Y) \end{aligned}$$

We now prove the cMap2 Judgments (9d) and (9e). First note that from Equation (12), if  $X : \uparrow_{\uplus}$  and  $Y : \uparrow_{\uplus}$ , then  $M(X, Y) = X$ . Judgement (9d) can be proven as follows:

$$\begin{aligned} & \text{cMap2}(f, X \uparrow_{\uplus} Y) \\ &= \text{cMap2}(f, \{((k, \theta), a \uplus b) \mid ((k, \theta), a) \in X, ((k, \theta), b) \in Y\} \\ & \quad \uplus \dots) \\ &= \{((k', (k, \theta)), d) \mid ((k, \theta), c) \in (\{((k, \theta), a \uplus b) \mid ((k, \theta), a) \in X, \\ & \quad ((k, \theta), b) \in Y\} \uplus \dots), \\ & \quad (k', d) \in f(k, c)\} \\ &= \{((k', (k, \theta)), d) \mid ((k, \theta), a) \in X, ((k, \theta), b) \in Y, \\ & \quad (k', d) \in f(k, a \uplus b)\} \uplus \dots \\ &= \{((k', (k, \theta)), d) \mid ((k, \theta), a) \in X, ((k, \theta), b) \in Y, \\ & \quad (k', d) \in (f(k, a) \uplus f(k, b))\} \uplus \dots \\ &= \{((k', (k, \theta)), d) \mid ((k, \theta), a) \in X, ((k, \theta), b) \in Y, \\ & \quad (k', d) \in f(k, a)\} \\ & \quad \uplus \{((k', (k, \theta)), d) \mid ((k, \theta), a) \in X, ((k, \theta), b) \in Y, \\ & \quad (k', d) \in f(k, b)\} \uplus \dots \\ &= \{((k', (k, \theta)), d) \mid ((k, \theta), a) \in X, ((k, \theta), b) \in Y, \\ & \quad (k', d) \in f(k, a)\} \\ & \quad \uplus \{((k', (k, \theta)), d) \mid ((k, \theta), a) \in X, (k, \theta) \notin \pi_1(Y), \\ & \quad (k', d) \in f(k, a)\} \\ & \quad \uplus \{((k', (k, \theta)), d) \mid ((k, \theta), a) \in X, ((k, \theta), b) \in Y, \\ & \quad (k', d) \in f(k, b)\} \\ & \quad \uplus \{((k', (k, \theta)), d) \mid ((k, \theta), b) \in Y, (k, \theta) \notin \pi_1(X), \\ & \quad (k', d) \in f(k, b)\} \\ &= \{((k', (k, \theta)), d) \mid ((k, \theta), a) \in X, (k', d) \in f(k, a)\} \\ & \quad \uplus \{((k', (k, \theta)), d) \mid ((k, \theta), b) \in Y, (k', d) \in f(k, b)\} \\ &= \text{cMap2}(f, X) \uplus \text{cMap2}(f, Y) \end{aligned}$$

Judgement (9e) can be proven as follows:

$$\begin{aligned} & \text{cMap2}(f, X \uparrow_{\square \times \uplus} Y) \\ &= \text{cMap2}(f, \{((k, \theta), (s_1 \square s'_1, s_2 \uplus s'_2)) \mid ((k, \theta), (s_1, s_2)) \in X, \\ & \quad ((k, \theta), (s'_1, s'_2)) \in Y\} \uplus \dots) \\ &= \text{cMap2}(f, \{((k, \theta), (s_1, s_2 \uplus s'_2)) \mid ((k, \theta), (s_1, s_2)) \in X, \\ & \quad ((k, \theta), (s_1, s'_2)) \in Y\} \uplus \dots) \end{aligned}$$

since  $s_1 \square s'_1$  has a valid value (equal to  $s_1$ ) if and only if  $s_1 = s'_1$ . Then, the proof of this judgement becomes similar to the proof of Judgement (9d).  $\square$

We will prove Theorem 2 using the following lemma:

**Lemmas 1.** *The following equations apply to terms in Definition 2 for a function  $f$  that satisfies the premises of Equations (9d) and (9e):*

$$G_1(\text{groupBy}(X)) = \text{groupBy}(T(X)) \quad (13a)$$

$$G_2(\text{coGroup}(X, Y)) = \text{coGroup}(T(X), T(Y)) \quad (13b)$$

$$T(\text{cMap2}(f, X)) = \text{cMap}(f, G_1(X)) \quad (13c)$$

(if  $X$  is a groupBy)

$$T(\text{cMap2}(f, X)) = \text{cMap}(f, G_2(X)) \quad (13d)$$

(if  $X$  is a coGroup)

where  $T$  removes the lineage  $\theta$ , while  $G_1$  and  $G_2$ , in addition to removing  $\theta$ , reconstruct the groupBy and coGroup result:

$$T(X) = \{ (k, a) \mid ((k, \theta), a) \in X \} \quad (14a)$$

$$G_1(X) = \text{groupBy}(\{ (k, a) \mid ((k, \theta), s) \in X, a \in s \}) \quad (14b)$$

$$G_2(X) = \text{coGroup}(\{ (k, a) \mid ((k, \theta), (s_1, s_2)) \in X, a \in s_1 \}, \\ \{ (k, b) \mid ((k, \theta), (s_1, s_2)) \in X, b \in s_2 \}) \quad (14c)$$

*Proof.* Proof of Equation (13a):

$$\begin{aligned} G_1(\text{groupBy}(X)) &= \text{groupBy}(\{ (k, a) \mid ((k, \theta), s) \in \text{groupBy}(X), a \in s \}) \\ &= \text{groupBy}(T(\{ ((k, \theta), a) \mid ((k, \theta), s) \in \text{groupBy}(X), \\ &\quad a \in s \})) \\ &= \text{groupBy}(T(X)) \end{aligned}$$

Proof of Equation (13b):

$$\begin{aligned} G_2(\text{coGroup}(X, Y)) &= \text{coGroup}(\{ (k, a) \mid ((k, \theta), (s_1, s_2)) \in \text{coGroup}(X, Y), \\ &\quad a \in s_1 \}, \\ &\quad \{ (k, b) \mid ((k, \theta), (s_1, s_2)) \in \text{coGroup}(X, Y), \\ &\quad b \in s_2 \}) \\ &= \text{coGroup}(T(\{ ((k, \theta), a) \mid ((k, \theta), (s_1, s_2)) \in \text{coGroup}(X, Y), \\ &\quad a \in s_1 \}), \\ &\quad T(\{ ((k, \theta), b) \mid ((k, \theta), (s_1, s_2)) \in \text{coGroup}(X, Y), \\ &\quad b \in s_2 \})) \\ &= \text{coGroup}(T(X), T(Y)) \end{aligned}$$

Proof of Equation (13c): From the premise of Equation (9d), we have  $f(k, X \uplus Y) = f(k, X) \uplus f(k, Y)$ . For  $s = \{a_1, \dots, a_n\}$ , we have  $\text{cMap}(\lambda a. f(k, \{a\}), s) = f(k, \{a_1\}) \uplus \dots \uplus f(k, \{a_n\}) = f(k, \{a_1, \dots, a_n\}) = f(k, s)$ , which implies:

$$f(k, s) = \{ (k', b) \mid a \in s, (k', b) \in f(k, \{a\}) \}$$

Hence,

$$\begin{aligned} \text{cMap}(f, G_1(X)) &= \text{cMap}(f, \text{groupBy}(\{ (k, a) \mid ((k, \theta), s) \in X, a \in s \})) \\ &= \{ (k', b) \mid (k, s) \in \text{groupBy}(\{ (k, a) \mid ((k, \theta), s) \in X, a \in s \}), \\ &\quad (k', b) \in f(k, s) \} \\ &= \{ (k', b) \mid (k, s) \in \text{groupBy}(\{ (k, a) \mid ((k, \theta), s) \in X, a \in s \}), \\ &\quad a \in s, (k', b) \in f(k, \{a\}) \} \\ &= \{ (k', b) \mid (k, a) \in (\{ (k, a) \mid ((k, \theta), s) \in X, a \in s \}), \\ &\quad (k', b) \in f(k, \{a\}) \} \\ &= \{ (k', b) \mid ((k, \theta), s) \in X, a \in s, (k', b) \in f(k, \{a\}) \} \\ &= \{ (k', b) \mid ((k, \theta), s) \in X, (k', b) \in f(k, s) \} \\ &= T(\{ ((k', (k, \theta)), b) \mid ((k, \theta), s) \in X, (k', b) \in f(k, s) \}) \\ &= T(\text{cMap2}(f, X)) \end{aligned}$$

Proof of Equation (13d): From the premise of Equation (9e),  $f(k, (X_1 \square X_2, Y_1 \uplus Y_2)) = f(k, (X_1, Y_1)) \uplus f(k, (X_2, Y_2))$ . Thus, we have:

$$\begin{aligned} \text{cMap}(f, G_2(X)) &= \text{cMap}(f, \text{coGroup}(\{ (k, a) \mid ((k, \theta), (s_1, s_2)) \in X, a \in s_1 \}, \\ &\quad \{ (k, b) \mid ((k, \theta), (s_1, s_2)) \in X, b \in s_2 \})) \\ &= \{ (k', v) \mid (k, (s_1, s_2)) \in \text{coGroup}(\{ (k, a) \mid ((k, \theta), (s_1, s_2)) \in X, a \in s_1 \}, \\ &\quad \{ (k, b) \mid ((k, \theta), (s_1, s_2)) \in X, b \in s_2 \}), \\ &\quad (k', v) \in f(k, (s_1, s_2)) \} \end{aligned}$$

Given a  $k$ , let  $X$  contain the values  $((k, \theta_1), (s_1^1, s_2^1)), \dots, ((k, \theta_n), (s_1^n, s_2^n))$ . Then, the previous  $\text{coGroup}(\dots)$  contains the value  $(k, (s_1^1 \square \dots \square s_1^n, s_2^1 \uplus \dots \uplus s_2^n))$ , and, thus,  $(k', v)$  is  $f(k, (s_1^1 \square \dots \square s_1^n, s_2^1 \uplus \dots \uplus s_2^n))$ , which is equal to  $f(k, (s_1^1, s_2^1)) \uplus \dots \uplus f(k, (s_1^n, s_2^n))$ . Therefore,

$$\begin{aligned} \text{cMap}(f, G_2(X)) &= \{ (k', v) \mid ((k, \theta), (s_1, s_2)) \in X, (k', v) \in f(k, (s_1, s_2)) \} \\ &= T(\{ ((k', (k, \theta)), v) \mid ((k, \theta), (s_1, s_2)) \in X, \\ &\quad (k', v) \in f(k, (s_1, s_2)) \}) \\ &= T(\text{cMap2}(f, X)) \quad \square \end{aligned}$$

**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:*

$$\mathcal{A}[q]_x = q \quad \text{for } x = \mathcal{T}_q[q] \quad (10)$$

*Proof.* To prove Equation (10), we first prove  $T(\mathcal{T}_c[c]) = c$  from Equations (8h) and (8i). If  $c = \text{cMap}(f, S_i)$ , then

$$\begin{aligned} T(\mathcal{T}_c[\text{cMap}(f, S_i)]) &= T(\text{cMap3}(f, S_i)) \\ &= \text{cMap}(f, S_i) \end{aligned}$$

If  $e = \text{cMap}(f, \text{groupBy}(c))$ , then using Equations (8i), (8f), (13a) and (13c):

$$\begin{aligned} T(\mathcal{T}_c[\text{cMap}(f, \text{groupBy}(c))]) &= T(\text{cMap2}(f, \mathcal{T}_c[\text{groupBy}(c)])) \\ &= T(\text{cMap2}(f, \text{groupBy}(\mathcal{T}_c[c]))) \\ &= \text{cMap}(f, G_1(\text{groupBy}(\mathcal{T}_c[c]))) \\ &= \text{cMap}(f, \text{groupBy}(T(\mathcal{T}_c[c]))) \\ &= \text{cMap}(f, \text{groupBy}(c)) \end{aligned}$$

The proof for  $e = \text{coGroup}(c_1, c_2)$  is similar (using Equations (8i), (8g), (13b) and (13d)).

We now prove Equation (10) using induction on  $q$ . Based on Definition 4, we have three cases. When  $q = (q_1, q_2)$ , then for  $x = \mathcal{T}_q[(q_1, q_2)] = (\mathcal{T}_q[q_1], \mathcal{T}_q[q_2])$ :

$$\begin{aligned} \mathcal{A}[(q_1, q_2)]_x &= (\mathcal{A}[q_1]_{\pi_1(x)}, \mathcal{A}[q_2]_{\pi_2(x)}) \\ &= (q_1, q_2) \quad (\text{from induction hypothesis}) \end{aligned}$$

For  $q = \text{cMap}(f, S_i)$ , we have:

$$\begin{aligned} \mathcal{A}[\text{cMap}(f, S_i)]_x &= \pi_2(x) \\ &= \pi_2(\mathcal{T}_q[\text{cMap}(f, S_i)]) \\ &= \pi_2(\{ ((i), b) \mid a \in S_i, b \in f(a) \}) \\ &= \text{cMap}(f, S_i) \end{aligned}$$

When  $q = \text{cMap}(f, \text{groupBy}(c))$ , we use the Judgment (9b), which requires  $g(k, v) : \otimes$ , where  $f(k, v) = \{g(k, v)\}$ . Using Equations (13c) and (13a), we have:

$$\begin{aligned} \mathcal{A}[\text{cMap}(f, \text{groupBy}(c))]_x &= \pi_2(\text{reduce}(\uparrow_{\otimes}, T(x))) \\ &= \pi_2(\text{reduce}(\uparrow_{\otimes}, T(\mathcal{T}_q[\text{cMap}(f, \text{groupBy}(c))]))) \\ &= \pi_2(\text{reduce}(\uparrow_{\otimes}, T(\text{cMap1}(f, \text{groupBy}(\mathcal{T}_c[c]))))) \\ &= \pi_2(\text{reduce}(\uparrow_{\otimes}, \{ (k, g(k, v)) \mid ((k, \theta), v) \in \text{groupBy}(\mathcal{T}_c[c]) \})) \end{aligned}$$

But, for  $\mathcal{T}_c[\llbracket c \rrbracket] = X \uplus Y$ , we have:

$$\begin{aligned}
& \pi_2(\text{reduce}(\uparrow_{\otimes}, \{ (k, g(k, v)) \mid ((k, \theta), v) \in \text{groupBy}(X \uplus Y) \})) \\
&= \pi_2(\text{reduce}(\uparrow_{\otimes}, \{ (k, g(k, x \uplus y)) \mid ((k, \theta), x) \in X, \\
&\quad ((k, \theta), y) \in Y \} \uplus \dots)) \\
&= \pi_2(\text{reduce}(\uparrow_{\otimes}, \{ (k, g(k, x) \otimes g(k, y)) \mid ((k, \theta), x) \in X, \\
&\quad ((k, \theta), y) \in Y \} \uplus \dots)) \\
&= \{ g(k, x) \otimes g(k, y) \mid ((k, \theta), x) \in X, ((k, \theta), y) \in Y \} \uplus \dots \\
&= \{ g(k, x \uplus y) \mid ((k, \theta), x) \in X, ((k, \theta), y) \in Y \} \uplus \dots \\
&= \text{cMap}(f, \{ (k, x \uplus y) \mid (k, x) \in T(X), (k, y) \in T(Y) \} \uplus \dots) \\
&= \text{cMap}(f, \text{groupBy}(T(X \uplus Y)))
\end{aligned}$$

Therefore,

$$\begin{aligned}
& \mathcal{A}[\text{cMap}(f, \text{groupBy}(c))]_x \\
&= \text{cMap}(f, \text{groupBy}(T(\mathcal{T}_c[\llbracket c \rrbracket]))) \\
&= \text{cMap}(f, \text{groupBy}(c))
\end{aligned}$$

The proof for  $q = \text{cMap}(f, \text{coGroup}(c_1, c_2))$  is similar. For  $q = \text{reduce}(\oplus, \text{cMap}(f, e))$ , and for  $e = \text{groupBy}(c)$  or  $e = \text{coGroup}(c_1, c_2)$  (the proof is easier for  $e = S_i$ ), we have:

$$\begin{aligned}
& \mathcal{A}[\text{reduce}(\oplus, \text{cMap}(f, e))]_x \\
&= \text{reduce}(\oplus, \pi_2(x)) \\
&= \text{reduce}(\oplus, \pi_2(\mathcal{T}_q[\text{reduce}(\oplus, \text{cMap}(f, e))])) \\
&= \text{reduce}(\oplus, \pi_2(\text{reduce}(\uparrow_{\oplus}, \text{cMap1}(f, \mathcal{T}_e[\llbracket e \rrbracket]))) \\
&= \text{reduce}(\oplus, \pi_2(\text{reduce}(\uparrow_{\oplus}, \mathcal{T}_q[\text{cMap}(f, e)]))) \\
&= \text{reduce}(\oplus, \text{cMap}(f, e))
\end{aligned}$$

from the previous proofs for  $q = \text{cMap}(f, \text{groupBy}(c))$  and  $q = \text{cMap}(f, \text{coGroup}(c_1, c_2))$ .  $\square$

**Theorem 3** (Homomorphism). *Any transformed term  $\mathcal{T}_q[\llbracket q \rrbracket]$ , where  $q$  is defined in Definition 2, is a homomorphism over the input streams, provided that each  $\text{cMap1}$ ,  $\text{cMap2}$ , and  $\text{cMap3}$  term in  $\mathcal{T}_q[\llbracket q \rrbracket]$  satisfies the premises of the judgments*

in Theorem 1:

$$\rho[S_1 : \uplus, \dots, S_n : \uplus] \vdash \mathcal{T}_q[\llbracket q \rrbracket] : \mathcal{M}[\llbracket q \rrbracket] \quad (11)$$

*Proof.* From Theorem 1, we can prove:

$$\begin{aligned}
& \mathcal{T}_e[\text{groupBy}(c)] : \uparrow_{\uplus} \\
& \mathcal{T}_e[\text{coGroup}(c_1, c_2)] : \uparrow_{\square \times \uplus} \\
& \mathcal{T}_e[\text{cMap}(f, e)] : \uplus
\end{aligned}$$

and then the  $\mathcal{T}_q[\llbracket q \rrbracket]$  monoid is derived from Definition 3.  $\square$

**Theorem 4.** *For all  $X, Y : (X \otimes Y) \widehat{\otimes} Y = X$ .*

*Proof.* We will prove this for  $\downarrow_{\widehat{\otimes}}$  only:

$$\begin{aligned}
& (X \uparrow_{\oplus} Y) \downarrow_{\widehat{\otimes}} Y \\
&= \{ (k, z \widehat{\oplus} y) \mid (k, z) \in (X \uparrow_{\oplus} Y), (k, y) \in Y, z \neq y \} \\
&\quad \uplus \{ (k, z) \mid (k, z) \in (X \uparrow_{\oplus} Y), k \notin \pi_1(Y) \} \\
&= \{ (k, (x \oplus y) \widehat{\oplus} y) \mid (k, x) \in X, (k, y) \in Y, \\
&\quad (k, y) \in Y, (x \oplus y) \neq y \} \\
&\quad \uplus \{ (k, z) \mid (k, z) \in (\{ (k, x \oplus y) \mid (k, x) \in X, (k, y) \in Y \} \\
&\quad \uplus \{ (k, x) \mid (k, x) \in X, k \notin \pi_1(Y) \} \\
&\quad \uplus \{ (k, y) \mid (k, y) \in Y, k \notin \pi_1(X) \}), \\
&\quad k \notin \pi_1(Y) \} \\
&= \{ (k, (x \oplus y) \widehat{\oplus} y) \mid (k, x) \in X, (k, y) \in Y, \\
&\quad (k, y) \in Y, (x \oplus y) \neq y \} \\
&\quad \uplus \{ (k, x) \mid (k, x) \in X, k \notin \pi_1(Y), k \notin \pi_1(Y) \} \\
&= \{ (k, x) \mid (k, x) \in X, (k, y) \in Y \} \\
&\quad \uplus \{ (k, x) \mid (k, x) \in X, k \notin \pi_1(Y) \} \\
&= X
\end{aligned}$$

$\square$