Incremental Stream Processing of Nested-Relational Queries

Leonidas Fegaras

University of Texas at Arlington

http://lambda.uta.edu/

Goals

We want to

- convert nested-relational queries to incremental stream processing programs automatically
- derive incremental programs that return accurate results, not approximate answers — unlike most stream processing systems
  - retain a minimal state during streaming
  - derive an accurate snapshot answer periodically
- focus on distributed stream processing engines on Big Data streams

MRQL: A Query Language for Big Data Analytics

- A powerful and efficient query processing system for complex data analysis on Big Data
- More powerful than existing query languages
  - a richer data model (nested collections, trees, ...)
  - arbitrary query nesting
  - more powerful query constructs
  - user-defined types and functions
- Able to capture most complex data analysis tasks declaratively
- Focus on raw, read-only, complex data (eg, XML, JSON)
- Platform-independent
- A common front-end for the multitude of distributed processing frameworks emerging in the Hadoop ecosystem

MRQL Streaming and Incremental MRQL

data analysis queries (in MRQL)
Incremental MRQL
MRQL
MRQL Streaming
MapReduce
Spark
Flink
Hama
Spark Streaming
Storm (planned)
Flink Streaming (planned)
YARN (Cluster Resource Management)
HDFS (Hadoop Distributed File System)

- Many emerging Distributed Stream Processing Engines (DSPEs)
  Storm, Spark Streaming, Flink Streaming
- Most are based on *batch streaming*
  - continuous processing over streams of batch data
    (data that come in continuous large batches)
- Incremental MRQL: translates any batch query to an incremental DSPE program automatically

Why Bother?

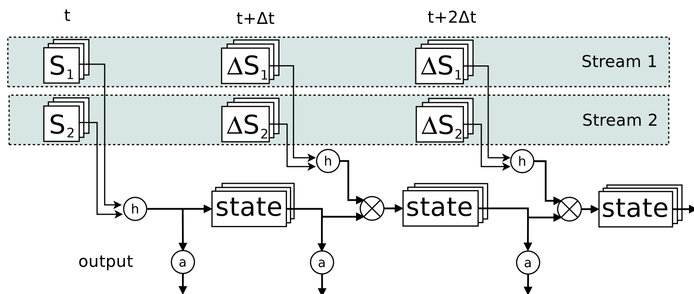
Incremental stream processing analyzes data in incremental fashion:

> Existing results on current data are reused and merged with the results of processing new data

Advantages:

- it can achieve better performance and may require less memory than batch processing
- it allows to process data streams in real-time with low latency
- it can be used for analyzing very large data incrementally
  > in batches that can fit in memory
  > enabling us to process more data with less hardware
- it can be used for incremental view maintenance in RDBs

Highlights of our Approach



An MRQL query $q(S_1, S_2)$ over two streams S_1 and S_2 is split into a homomorphism h and an answer function a:

$$q(S_1, S_2) = a(h(S_1, S_2))$$

where $h(S_1 \uplus \Delta S_1, S_2 \uplus \Delta S_2) = h(S_1, S_2) \otimes h(\Delta S_1, \Delta S_2)$,
for some monoid $\otimes$

Example

An MRQL query:

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

The homomorphism:

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

groups the results by the lineage x (the group-by key) and y (the join key)

Example (cont.)

The merge function $\oplus$ is a full outer join $⟗$:

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

The answer function:

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

The MRQL Algebra

Far more effective than the nested relational algebra

- **cMap**(f, S): $\{\beta\}$, for $f : \alpha \rightarrow \{\beta\}$ and $S : \{\alpha\}$

$$\mathrm{cMap}(\lambda x.\, \{x + 1\},\ \{1, 2, 3\}) \ = \ \{2, 3, 4\}$$

- **groupBy**$(S) : \{(\kappa, \{\alpha\})\}$, for $S : \{(\kappa, \alpha)\}$

$$\mathrm{groupBy}(\{(1, 10), (2, 20), (1, 30), (1, 40)\}) \ = \ \{(1, \{10, 30, 40\}), (2, \{20\})\}$$

- **coGroup**$(R, S) : \{(\kappa, (\{\alpha\}, \{\beta\}))\}$, for $R : \{(\kappa, \alpha)\}$ and $S : \{(\kappa, \beta)\}$

$$\begin{array}{l} \mathrm{coGroup}(\{(1, 10), (2, 20), (1, 30)\}, \{(1, 5), (2, 6), (3, 7)\}) \\ \quad = \ \{(1, (\{10, 30\}, \{5\})), (2, (\{20\}, \{6\}), (3, (\{\}, \{7\}))))\} \end{array}$$

- **reduce**$(\oplus, S) : \alpha$, for $S : \{\alpha\}$ and $\oplus : (\alpha, \alpha) \rightarrow \alpha$

$$\mathrm{reduce}(+, \{1, 2, 3\}) = 6$$

Term Normalization

- Fuse two cascaded cMaps into a nested cMap:

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

- Unnesting any nested query:

$$\begin{aligned} F(X, Y) &= \boxed{\mathrm{cMap}(\lambda x.\, g(\boxed{\mathrm{cMap}(\lambda y.\, h(x, y),\ Y)}), X)} \\ &\rightarrow \mathrm{cMap}(\lambda(k, (xs, ys)).\, F(xs, ys), \\ &\qquad\qquad \mathrm{coGroup}(\mathrm{cMap}(\lambda x.\, \{(k_1(x), x)\},\ X), \\ &\qquad\qquad\qquad\qquad \mathrm{cMap}(\lambda y.\, \{(k_2(y), y)\},\ Y))) \end{aligned}$$

provided that $k_1(x) \neq k_2(y)$ implies $h(x, y) = \{\}$

- *Normal form:* a tree of groupBy/coGroup nodes connected with cMap

Algebraic Terms

- Our algebraic operations are homomorphisms
  - groups from a groupBy/coGroup are merged with a full outer join $⟗$

$$\mathrm{groupBy}(X \uplus Y) \;=\; \mathrm{groupBy}(X) ⟗ \mathrm{groupBy}(Y)$$

- ... but cMap over groupBy or coGroup may not be a homomorphism
  - cMap distributes over $\uplus$ but doesn't distribute over $⟗$
- *Special case:* if g is a homomorphism then this is too:

$$\mathrm{cMap}(\lambda(k,s).\,\{(k,g(s))\},\, \mathrm{groupBy}(X))$$

  ie, when cMap propagates the groupBy key
- We are exploiting this property to the fullest

Transforming an Algebraic Term to a Homomorphism

- We transform each term to propagate the groupBy and coGroup keys to the output

  known as lineage tracking

- *lineage keys:* the groupBy/coGroup keys in the query
- Why?
  - the query results are grouped by the lineage keys
    - the current state is kept grouped by the lineage keys
    - the query results on the new data are grouped by the lineage keys
  - the state is combined with the new query results by joining them on the lineage keys using $⟗$

Lineage Tracking

- The lineage tree θ has the same shape as the tree of groupBy and coGroup operations in the term
- A lineage law transforms a term e of type $\{t\}$ to a term $[\![e]\!]$ of type $\{(\theta, t)\}$. Example:

$$[\![\mathrm{cMap}(f, \mathrm{groupBy}(e))]\!] \\ \quad\rightarrow \{\, ((k, \theta), w) \,|\, ((k, \theta), s) \in \mathrm{groupBy}(\mathrm{flip}([\![e]\!])),\, w \in f(k, s) \,\}$$

where flip maps $\{((k, v), \theta)\}$ to $\{((k, \theta), v)\}$.
It extends the lineage θ of e with the groupBy key k

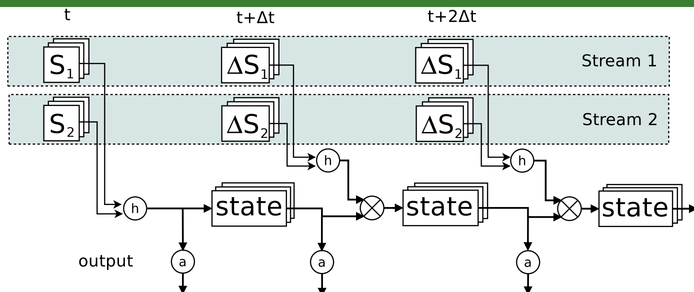
Transforming Terms to Homomorphisms

- What if a cMap term is not a homomorphism?
  1. split the cMap into two cMaps: one homomorphic and one not
  2. pull out and fuse all non-homomorphic cMaps at the root of the algebraic tree
- For a term $\mathrm{cMap}(\lambda v.\, e,\, X)$:
  1. find the largest subterms $e_1, \ldots, e_n$ in the algebraic term e that are homomorphisms
  2. replace these terms with new variables:

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

- All non-homomorphic parts from the cMap functionals are pulled outwards and combined, deriving two terms:
  - a homomorphism: performs the incremental computation
  - an answer function: removes the lineage, combines the lineage groups, and returns the query answer

Incremental Processing on Two Streams (revisited)



$$h(S_1 \uplus \Delta S_1, S_2 \uplus \Delta S_2) = h(S_1, S_2) \otimes h(\Delta S_1, \Delta S_2)$$

- The merging $\otimes$ is often an outer join $⟗$ that joins the curent state $h(S_1, S_2)$ with the new results $h(\Delta S_1, \Delta S_2)$
- The state remains partitioned on the lineage keys
- Only $h(\Delta S_1, \Delta S_2)$ needs to be distributed across workers based on the lineage keys
- The new state doesn't need to be repartitioned

Implementation and Current Work

- Incremental MRQL is implemented on Spark Streaming
- Tested on various SQL-like queries
  - an order of magnitude speed-up compared to batch processing that processes all the data
- Currently, we are working on iterative queries, such as PageRank and clustering
  - this time, we calculate approximate answers

Performance Evaluation