CSE6331: Cloud Computing

Leonidas Fegaras

University of Texas at Arlington

Cloud Computing at UTA: MRQL & DIQL

DISC (Data-Intensive Scalable Computing) Systems

- Designed for large-scale data processing on computer clusters
- Often work on raw data – indexes are uncommon
- They hide the details of distributed computing, fault tolerance, recovery, etc
- Many provide functional-style APIs
  - using powerful higher-order operations as building blocks
  - preventing interference among parallel tasks
- But some programmers are unfamiliar with functional programming
- Hard to develop complex applications when the focus is in optimizing performance
- Too many competing DISC platforms:

  Map-Reduce, Spark, Flink, Storm, ...
- Hard to tell which one will prevail in the near future
- ...or you can use a query language that is independent of the underlying DISC platform!

Limitations of Current DISC Query Languages

They are subsets of SQL.
But raw data is often nested, not normalized.

Most DISC query languages

- provide a limited syntax for operating on data collections
  simple joins and group-bys
- have limited support for nested collections and hierarchical data
- cannot express complex data analysis tasks that need iteration
- Example: Hive and DataFrames treat in-memory and distributed collections differently
  - to flatten a nested collection in a row in Spark DataFrames, one must 'explode' the collection
  - Hive supports 'lateral views' to avoid creating intermediate tables when exploding nested collections

Apache MRQL

Apache MRQL (incubating)

MRQL: a Map-Reduce Query Language

Web site: `http://mrql.incubator.apache.org/`
Wiki: `http://wiki.apache.org/mrql/`

History:

- Fall 2010: started at UTA as an academic research project
- March 2013: enters Apache Incubation
- 4 releases under Apache so far: latest MRQL 0.9.6

MRQL: Design Objectives

- Wanted to develop a powerful and efficient query processing system for complex data analysis applications on big data
  - more powerful than existing query languages
  - able to capture most complex data analysis tasks declaratively
  - able to work on read-only, raw (in-situ), complex data
  - HDFS as the physical storage layer
  - platform-independent:
    - the same query can run on multiple platforms on the same cluster
    - allowing developers to experiment with various platforms effortlessly
  - efficient!
- We envision MRQL to be:
  - a common front-end for the multitude of distributed processing frameworks emerging in the Hadoop ecosystem
  - a tool for comparing these systems (functionality & performance)

Is this yet Another SQL for Map-Reduce?

- MRQL is **NOT** SQL!
- MRQL is an SQL-like query language for large-scale, distributed data analysis on a computer cluster
- Unlike SQL, MRQL supports
  - a richer data model (nested collections, trees, ...)
  - arbitrary query nesting
  - more powerful query constructs
  - user-defined types and functions
- MRQL queries can run on multiple distributed processing platforms

  currently Apache Hadoop MapReduce, Hama, Spark, Flink, and (soon) Storm/Trinder
- The MRQL syntax and semantics have been influenced by
  - modern database query languages (mostly, XQuery and ODMG OQL)
  - functional programming languages (sequence comprehensions, algebraic data types, type inference)

Language Features

The MRQL query language:

- provides a rich type system that supports hierarchical data and nested collections uniformly
  - general algebraic datatypes – can be recursive
    - JSON and XML are user-defined types
  - pattern matching over data constructions (similar to Scala match)
  - local type inference (similar to Scala)
- allows nested queries at any level and at any place
  - no need for awkward nulls and outer-joins
- supports UDFs
  - provided that they don't have side effects
- allows to operate on the grouped data using queries
  - as is done in XQuery and Pig
  - improves SQL group-by with aggregation (which are too awkward)

Language features

The MRQL query language:

- supports custom aggregations and reductions using UDFs
  provided they have certain properties (associative & commutative)
- supports iteration declaratively
  to capture iterative algorithms, such as PageRank
- supports custom parsing and custom data fragmentation
- provides syntax-directed construction and deconstruction of data
  to capture domain-specific languages

The MRQL Data Model

MRQL supports the following types:

- a basic type: bool, short, int, long, float, double, string.
- a tuple $(t_1, \ldots, t_n)$,
- a record $< A_1 : t_1, ..., A_n : t_n >$,
- a list (sequence) $[t]$ or list(t),
- a bag (multiset) $\{t\}$ or bag(t),
- a user-defined type
- an algebraic data type T

Algebraic Data Types

- A data type T is defined at the top level:

```
data T = C1: t1 | ... | Cn: tn;
```

  where C1, ..., Cn are data constructors and t1, ..., tn are types
- It can be recursive
- For example, an integer list can be defined as follows:

```
data IList = Cons: (int,IList) | Nil: ();
```

- Then, `Cons(1,Cons(2,Nil()))` constructs the list [1,2]
- XML is a predefined data type, defined as:

```
data XML = Node: ( String, bag( (String,String) ),
                   list(XML) )
         | CData: String;
```

- For example, `<a x='1'><b>text</b></a>` is constructed using

```
Node('a',{('x','1')},[Node('b',{},[CData('text')])])
```

Patterns

- Patterns are used in select-queries and case statements
- A pattern can be:
  - a pattern variable that matches any data and binds the variable to data,
  - a constant basic value,
  - a * that matches any data,
  - a data construction $C(p_1, \ldots, p_n)$,
  - a tuple $(p_1, \ldots, p_n)$,
  - a record $< A_1 : p_1, \ldots, A_n : p_n >$,
  - a list $[p_1, \ldots, p_n]$
- A case statement takes the following form:

```
case e { p1: e1 | ... | pn: en }
```

- Example:

```
case e { Node(*,*,Node('a',*,cs)): cs | *: [] }
```

The Select-Query Syntax

- The select-query syntax takes the form: ([...] are optional)

```
select [ distinct ] e
from p in e, ..., p in e
[ where e ]
[ group by p: e [ having e ] ]
[ order by e [ limit e ] ]
```

- where e are expressions/queries and p are patterns
- Example:

```
select (n,cn)
from < name: n, children: cs > in Employees,
     < name: cn > in cs
```

- Same as:

```
select (e.name,c.name)
  from e in Employees,
       c in e.children
```

More Examples

```
select ( d, c, sum(s) )
from <dno:dn,salary:s> in Employees
group by (d,c): ( dn, salary>=100000 )
having avg(s) >= 80000

select (k,sum(o.TOTALPRICE))
 from o in Orders,
      c in Customers
 where o.CUSTKEY=c.CUSTKEY
 group by k: c.NAME;

select (c.NAME,avg(select o.TOTALPRICE
                   from o in Orders
                   where o.CUSTKEY=c.CUSTKEY))
  from c in Customers;
```

Simple example: matrix multiplication

A sparse matrix X is represented as a bag of (X_{ij}, i, j).

$$Z_{ij} = \sum_k X_{ik} * Y_{kj}$$

```
select ( sum(z), i, j )
  from (x,i,k) in X, (y,k,j) in Y, z = x*y
 group by i, j
```

An XML example

Group all persons according to their interests and the number of open auctions they watch. For each such group, return the number of persons in the group:

```
select ( cat, os, count(p) )
from p in XMARK,
     i in p.profile.interest
group by cat: i.@category,
          os: count(p.watches.@open_auctions)
```

Another XML example

Invert the citation graph in DBLP by grouping the items by their citations and by ordering these groups by the number of citations they received:

```
select ( select text(a.title)
           from a in DBLP
          where a.@key = x,
         count(a) )
from a in DBLP,
     c in a.cite
group by x: text(c)
order by inv(count(a))
```

Example: k-means clustering

Derive k clusters from a set of points P:

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

Example: the PageRank algorithm

Simplified PageRank:

- A graph node is associated with a PageRank and its outgoing links:

  $< \text{id: } 23,\ \text{rank: } 0.0,\ \text{adjacent: } \{\ 10,\ 45,\ 35\ \} >$

- Propagate the PageRank of a node to its outgoing links;
  each node gets a new PageRank by accumulating the propagated PageRanks from its incoming links:

```
repeat nodes = ...
  step select < id: m.id, rank: n.rank, adjacent: m.adjacent >
       from n in (select < id: key, rank: sum(c.rank) >
                  from c in ( select < id: a,
                                       rank: n.rank/count(n.adjacent) >
                              from n in nodes,
                                   a in n.adjacent )
                  group by key: c.id ),
            m in nodes
       where n.id = m.id
```

Architecture

Query translation stages:

1. type inference
2. query translation and normalization
3. simplification
4. algebraic optimization
5. plan generation
6. plan optimization
7. compilation to Java code

Algebraic operators

- Algebraic operations on bags:

groupBy	(X: $\{(\kappa, \alpha)\}$)	: $\{(\kappa, \{\alpha\})\}$
flatMap	(f: $\alpha \to \{\beta\}$, X: $\{\alpha\}$)	: $\{\beta\}$
reduce	($\oplus$: $(\alpha, \alpha) \to \alpha$, X: $\{\alpha\}$)	: α
union	(X: $\{\alpha\}$, Y: $\{\alpha\}$)	: $\{\alpha\}$

- Join: coGroup (X: $\{(\kappa, \alpha)\}$, Y: $\{(\kappa, \beta)\}$) : $\{(\kappa, \{\alpha\}, \{\beta\})\}$
- map-reduce(m,r,X) = flatMap(r,groupBy(flatMap(m,X)))
- List operations: orderBy, append
- Iteration: repeat (f: $\{\alpha\} \to \{\alpha\}$, X: $\{\alpha\}$) : $\{\alpha\}$

Query optimization

The query optimizer:

- uses a cost-based optimization framework to map algebraic terms to efficient workflows of physical operations
- handles dependent joins (used for nested collections)
- unnest deeply nested queries and converts them to join plans

Code Generation

- Run-time code generation (SQL, Hive, Spark SQL, MRQL, ...)
  - Run-time: checking, optimization, and code generation
  - Can embed values through parametric queries
- Two-stage code generation (DryadLINQ, Emma)
  - Compile-time: checking and static optimizations

    Generates a query graph
  - Run-time: cost-based optimizations and code generation
  - Embedding:
    - DryadLINQ broadcasts embedded values to workers
    - Emma uses Scala's run-time reflection to access embedded values
- Compile-time code generation (DIQL)
  - Compile-time: checking, static optimizations, and code generation
  - The optimizer can still do cost-based optimizations:
    1. picks few viable choices for query plans at compile-time
    2. generates conditional code that chooses a plan based on run-time statistics

Wish List for Query Embedding

PL: host programming language, QL: DISC query language

- No impedance mismatch:
  a QL must be fully embedded into the host PL
- The QL and PL data models must be equivalent
- . . . but a QL must work on distributed collections with special semantics
- Distributed and in-memory collections must be indistinguishable in the QL syntax
  - although they may be processed differently
- No null values in the QL data model
  - SQL uses 3-valued logic (very obscure)
  - most PLs do not provide a standardized way to treat nulls
  - need to handle null values before UDF calls or PL code
- $\Rightarrow$ no outer joins
- nulls in data (unknown/missing values) can be handled with PL code

DIQL (Data-Intensive Query Language)

An SQL-like query language for DISC systems that

- is deeply embedded in Scala
- is optimized and translated to Java byte code at compile-time
- is designed to support multiple Scala-based APIs for DISC processing
  - currently: Spark, Flink, and Twitter's Cascading/Scalding
- can uniformly work on both distributed and in-memory collections using the same syntax
- allows seamless mixing of native Scala code with SQL-like query syntax
  - can use any Scala pattern, access any Scala variable, and embed any functional Scala code
  - can use the core Scala libraries and tools, and user-defined classes
- has compositional semantics based on monoid homomorphisms

The DIQL Syntax

pattern: p ::= *any Scala pattern, including a refutable pattern*

qualifier: q ::= p **<−** e *generator over a dataset*
| p **<−−** e *generator over a small dataset*
| p = e *binding*

expression: e ::= *any Scala functional expression*
| **select** [**distinct**] e
from $q, \ldots, q$
[**where** e]
[**group by** p [: e] [**having** e]]
[**order by** e]
| $\oplus/e$ *aggregation*

The Group-By Semantics

- Based on comprehensions with 'order by' and 'group by' [Wadler and Peyton Jones 2007]
- Pattern variables are essential to the group-by semantics
- A group-by lifts each pattern variable defined in the **from**-clause from some type t to a $\{t\}$

  this $\{t\}$ contains all the variable values associated with the same group-by key

Example: Matrix Multiplication

- A sparse matrix M is a bag of (v, i, j), for $v = M_{ij}$
- Matrix multiplication of X and $Y = \sum_k X_{ik} * Y_{kj}$:

```
select ( +/z, i, j )
from (x,i,k) <- X, (y,k_,j) <- Y, z = x*y
where k == k_
group by (i, j)
```

- before the group-by: `z` $= X_{ik} * Y_{kj}$
- after group-by:

  `z` is lifted to a bag of values $X_{ik} * Y_{kj}$;
  for each group `(i, j)`, the bag `z` contains $X_{ik} * Y_{kj}$, for all k

- `+/z` sums up all `z` values, for each group `(i, j)`

How can we Express Outer Joins in DIQL?

Outer semijoins can be simply expressed as nested queries
In SQL:

```
select c.name
from Customers c left outer join Orders o on o.cid = c.cid
group by c.cid
having o.price is null or c.account >= sum(o.price)
```

in DIQL:

```
select c.name from c <- Customers
where c.account >= +/(select o.price from o <- Orders
                      where o.cid == c.cid)
```

The DIQL query processor unnests any nested query to a coGroup

Full Outer Join: Matrix Addition

Matrix addition $X + Y$ is equivalent to a full outer join:

```
select ( +/(x++y), i, j )
from (x,i,j) <- X group by (i,j)
from (y,i_,j_) <- Y group by (i_,j_)
```

- `X` is grouped by `(i,j)`
- `Y` is grouped by `(i_,j_)`
- with `(i,j)==(i_,j_)`

This is a coGroup!

Mixing SQL-like Syntax with Scala Code

A Scala class that represents a graph node:

```
case class Node ( id: Long, adjacent: List[Long] )
```

In Spark, a graph is a distributed collection of type RDD[Node].
Query: transform a graph so that each node is linked to the neighbors of its neighbors:

```
q("""
let graph = select Node( n, ns )
            from line <- sc.textFile("graph.txt"),
                 n::ns = line.split(",").toList.map(_.toLong)
in select Node( x, ++/ys )
   from Node(x,xs) <- graph,
        a <- xs,
        Node(y,ys) <- graph
   where y == a
   group by x
""")
```

Monoids as a Formal Basis for DISC Systems

- The results of data-parallel computations must be independent of
  - the way we divide the data into partitions and
  - the way we combine the partial results to obtain the final result

  $\Rightarrow$ data-parallel computations must be *associative*
- They can be expressed as *monoid homomorphisms*
  - A monoid has an associative merge function and an identity
  - A collection monoid has also a unit function

    $(\uplus, \{\}, \lambda x.\{x\})$ for bags
  - A monoid homomorphism maps a collection monoid to a monoid
  - Captures data parallelism

    $$H(P_1 \uplus P_2 \uplus \cdots \uplus P_n) = H(P_1) \oplus H(P_2) \oplus \cdots \oplus H(P_n)$$

  - Some monoid homomorphisms are not allowed

    can't convert a bag to a list
- Collection monads (aka, ringads) have a monoidal structure too
  - but are not a good basis for practical data-centric languages
  - require various extensions (for group-by, aggregations, etc)

The Monoid Algebra

Generalizes the nested relational algebra (here shown on bags only)

- **flatMap** of type $(\alpha \to \{\beta\}, \{\alpha\}) \to \{\beta\}$

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

  Monoid: $\uplus$
- **groupBy** of type $\{(\kappa, \alpha)\} \to \{(\kappa, \{\alpha\})\}$

$$\text{groupBy}(\{(1,a),(2,b),(1,c),(1,d)\}) \ = \ \{(1,\{a,c,d\}),(2,\{b\})\}$$

  It returns an indexed set (aka, a key-value map).
  Monoid: *indexed set union* (a full outer join that unions the groups of matching keys)
- **orderBy** of type $\{(\kappa, \alpha)\} \to_{\ll} (\kappa, \{\alpha\})$
  Monoid: merges sorted lists by unioning the groups of matching keys

The Monoid Algebra (cont.)

- **coGroup** of type $(\{(\kappa, \alpha)\}, \{(\kappa, \beta)\}) \rightarrow \{(\kappa, (\{\alpha\}, \{\beta\}))\}$

$$\begin{aligned}&\text{coGroup}(\{(1, a), (2, b), (1, c)\},\\ &\qquad \{(1, 5), (2, 6), (3, 7)\})\\ &= \{(1, (\{a, c\}, \{5\})), (2, (\{b\}, \{6\}), (3, (\{\}, \{7\})))\}\end{aligned}$$

  A lossless inner/outer equi-join.

  Monoid: a full outer join that unions groups pairwise

- **reduce** of type $((\alpha, \alpha) \rightarrow \alpha, \{\alpha\}) \rightarrow \alpha$

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

  Monoid: $+$

The Essence of Data Parallelism

- Divide-and-conquer = groupBy-and-flatMap
- Map-Reduce = flatMap-groupBy-flatMap

$$\text{mapReduce}(m, r)(X) \quad = \quad \text{flatMap}(r, \text{groupBy}(\text{flatMap}(m, X)))$$

for a map function m of type $(k_1, \alpha) \rightarrow \{(k_2, \beta)\}$
and a reduce function r of type $(k_2, \{\beta\}) \rightarrow \{(k_3, \gamma)\}$

Term Normalization

- Fuse two cascaded flatMaps into a nested flatMap:

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

- *Normal form:* a tree of groupBy/coGroup nodes connected via a single flatMap

Query Unnesting

- Deriving joins and unnesting any nested query:

$$F(X,Y) \quad = \quad \boxed{\mathrm{flatMap}(\lambda x.\, g(\boxed{\mathrm{flatMap}(\lambda y.\, h(x,y),\ Y)}),X)}$$

$$\rightarrow \quad \mathrm{flatMap}(\lambda(k,(xs,ys)).\, F(xs,ys),\ \mathrm{coGroup}(\mathrm{flatMap}(\lambda x.\, \{(k_1(x),x)\},\ X),\ \mathrm{flatMap}(\lambda y.\, \{(k_2(y),y)\},\ Y)))$$

provided that there are key functions k_1 and k_2 such that
$k_1(x) \neq k_2(y) \ \Rightarrow \ h(x,y) = \{\,\}$

eg, $h(x,y) =$ **if** $k_1(x) == k_2(y)$ **then** e **else** $\{\,\}$

- coGroups are implemented as distributed partitioned joins or broadcast joins

Algebraic Terms

- Our monoid algebraic operations are homomorphisms
- ... but flatMap over groupBy or coGroup may not be a homomorphism
  - groupBy returns an indexed set
  - flatMap distributes over $\uplus$ but doesn't distribute over indexed set union
- *Special case:* if g is a homomorphism then so is this:

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

  ie, flatMap must propagate the groupBy key
- This is the basis for our incrementalization method

Incremental Stream Processing

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)

We want to:

- convert any batch DISC query to an 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

Why Bother?

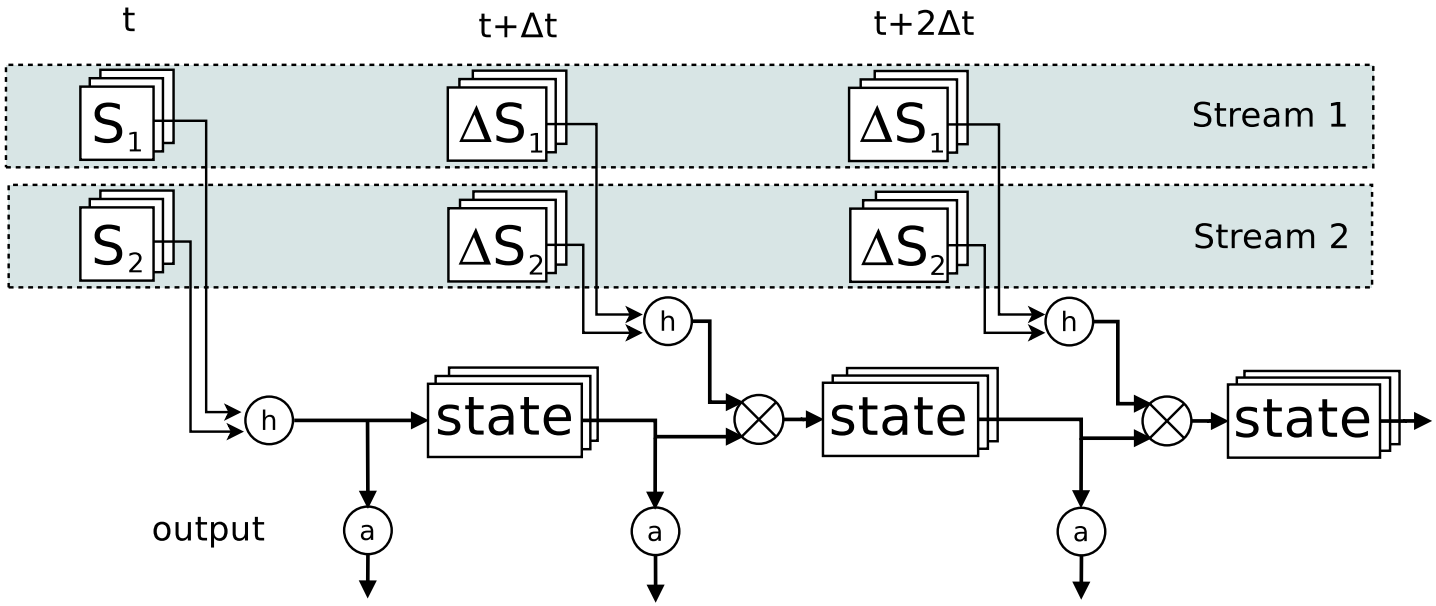
Incremental stream processing analyzes data in incremental fashion:

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

Advantages:

- it can achieve better performance and 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

Incrementalization using Homomorphisms



A 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 $\otimes$

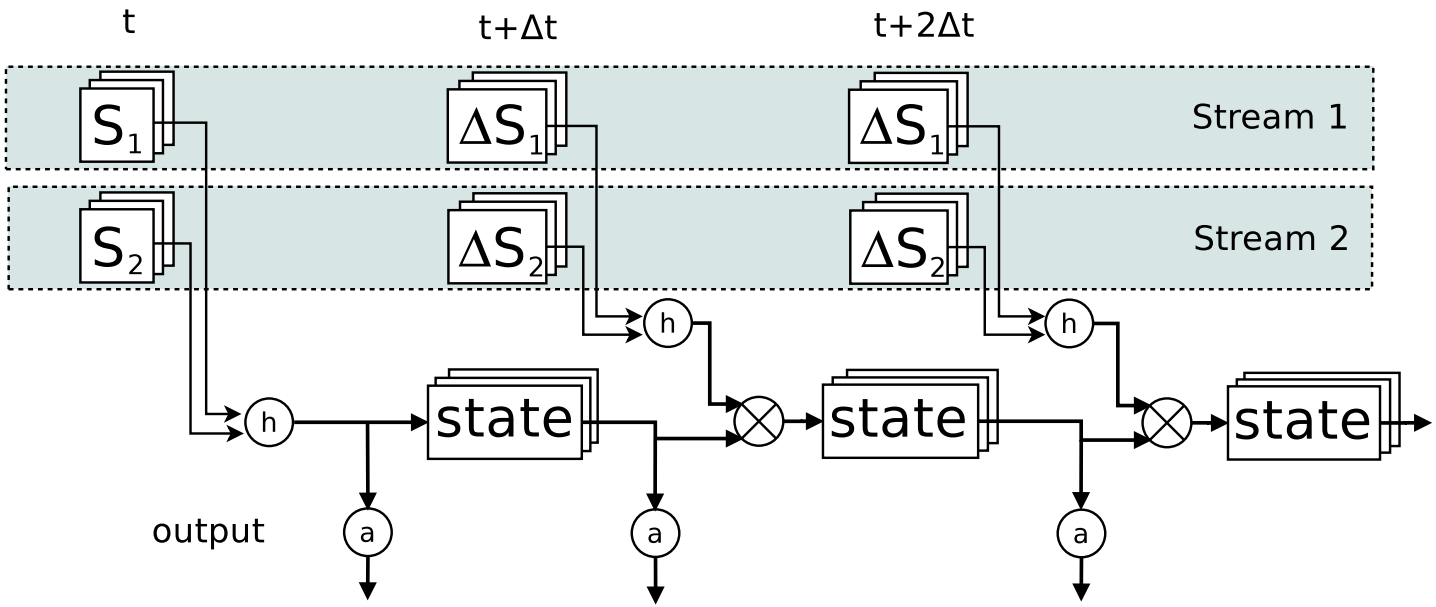
Transforming an Algebraic Term to a Homomorphism

- 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 will be 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 indexed set union

Incremental Processing



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

- In most cases, the merging $\otimes$ is an indexed set union that joins the current 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 the workers based on the lineage keys
- We may store the state as a key-value map and update it in place