XML Query Optimization in Map-Reduce

Leonidas Fegaras, Chengkai Li, Upa Gupta, and Jijo J. Philip

University of Texas at Arlington
Data Processing using Map-Reduce

- Introduced by Google in 2004
- It facilitates the parallel execution of ad-hoc, long-running large-scale data analysis tasks on a shared-nothing cluster of commodity computers connected through a high-speed network
- Several implementations:
  - Apache Hadoop, Google Sawzall, Microsoft Dryad, …
- Used extensively by companies (on a very large scale):
  - Yahoo!, Facebook, Google, ...
- There are some higher-level languages that make map-reduce programming easier:
  - HiveQL, PigLatin, Scope, Dryad/Linq, …
Data Processing using Map-Reduce

Still a controversial topic in the DB community
- parallel databases: need to model and load the data before processing
- map-reduce: better suited to a small number of ad-hoc queries over write-once raw data
  - better fault tolerance and ability to operate in heterogeneous environments
  - indexes may not be applicable:
    - they are not very useful when processing most of the data
    - the amortized cost of index creation/population may exceed the cost benefit of using the index

There are some recent systems that try to bridge the gap between map-reduce and RDB:
- Hive
- Pig
- HadoopDB
- Manimal
- Hadoop++
The Map-Reduce Model

- Very simple model: need to specify a map and a reduce task
  - the map task specifies how to process a single key/value pair to generate a set of intermediate key/value pairs
  - the reduce task specifies how to merge all intermediate values associated with the same intermediate key
- … but, many configuration parameters to adjust for better performance
Example with 3 Mappers and 2 Reducers

mapper 0

mapper 1

mapper 2

data split 0

data split 1

data split 2

group-by key

partition function: key mod 2

reducer 0

reducer 1

output 0

output 1

m : map

r : reduce

m :  map

r : reduce

partition function: key mod 2

XML Query Optimization in Map-Reduce

http://lambda.uta.edu/mrql/
Map-Reduce vs SQL

- Map-reduce programs are computationally complete
- Standard SQL (join, selection, projection, group-by, having, order-by) can be directly coded into map-reduce workflows
- Example:

```sql
select v.A, sum(v.B) from R v group by v.A
```

- Map-reduce pseudo-code in Hadoop:

```java
class Mapper
    method map ( key, v ):
        emit(v.A, v);

class Reducer
    method reduce ( key, values ):
        int c = 0;
        for each v ∈ values do c += v.B;
        emit(key,c);
```
Motivation

- Although the map-reduce model is simple, it is
  - hard to develop programs for complex data analysis tasks
  - hard to optimize general map-reduce programs expressed in a general-purpose programming language
- As is evident from the success of the RDB technology
  - program optimization would be more effective if the programs were written in a higher-level query language that hides the implementation details and is amenable to optimization
MRQL: a Map-Reduce Query Language

- An SQL-like query language for expressing map-reduce computations
- MRQL is powerful enough to express most common data analysis tasks over many forms of raw data:
  - XML text documents
  - line-oriented text documents with comma-separated values
- The MRQL syntax has been influenced by some functional query languages, such as ODMG OQL and XQuery
- The MRQL semantics is based on the work by the functional programming community on list comprehensions with group-by and order-by
- Our goal:

  To develop a novel framework for optimizing and evaluating map-reduce computations expressed in MRQL
Why not SQL?

Need to capture most map-reduce computations declaratively, within the query language, without using explicit calls to map-reduce jobs
  - otherwise, we may get suboptimal/error-prone/hard-to-maintain code

1. Must allow arbitrary queries/UDFs for the group-by and aggregation expressions (to make it m-r complete)

   \[
   \text{select } k, r(x) \\
   \text{from } x \text{ in } X \\
   \text{group by } k: m(x)
   \]

   after group-by, \( x \) is lifted to a bag (a group); \( r \) can be a query/UDF over bags

2. Must support hierarchical data and nested collections uniformly
   - need to optimize XML/JSON queries

3. Must allow arbitrary query nesting
   - otherwise, must explicitly use outer-joins/group-bys
     - ★ ugly
     - ★★ lost optimization opportunities

4. Must support recursion (for PageRank, etc)
Optimization

- We can leverage on the relational query optimization technology, but ...
  - the MRQL physical operators are different
  - cost factors are different
  - must deal with nested collections and nested queries

Example:

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

- A RDBMS may do a left-outer join between X and Y on x.A=y.B and group the result by the x key
  - requires 2 map-reduce jobs
- Often better: one reduce-side join, which requires 1 map-reduce job
Working with XML Data

- Each map worker is assigned a data split that consists of *data fragments*
  - for a text file: a single line in the file ⇒ a relational record
  - for an XML document: the choice for a suitable fragment size and structure may depend on the actual application

- XML fragmentation
  - cannot use existing XML parsers
    - a data split may start at an arbitrary point in the XML document
    - XML elements may cross data split boundaries
  - built on top of the existing Hadoop XML input format
  - uses a set of synchronization tags to start parsing a data split
  - uses a stream-based XPath processor to extract elements from a data split

```
source(  "article","incollection","book","inproceedings"),
Xpath( .[year=2009]/title ),
"dblp.xml" )
```
Physical Operators

The map-reduce operation:  \[ \text{MapReduce} \left( m, r \right) S \]

\[
\begin{align*}
\text{select} \; w \\
\text{from} \; z \; \text{in} \; (\text{select} \; r(\text{key},y) \\
\text{from} \; x \; \text{in} \; S, \\
(k,y) \; \text{in} \; m(x) \\
\text{group by} \; \text{key:} \; k), \\
w \; \text{in} \; z
\end{align*}
\]

Functional parameters:
- \( m \): map
- \( r \): reduce

Types:
- \( S \): \( \text{bag}(\alpha) \)
- \( m \): \( \alpha \rightarrow \text{bag}(\kappa \times \gamma) \)
- \( r \): \( \kappa \times \text{bag}(\gamma) \rightarrow \text{bag}(\beta) \)

Semantics:
\[ \text{MapReduce} \left( m, r \right) S = \text{concatMap}(r) \circ \text{groupBy} \circ \text{concatMap}(m) \]

where \( \text{groupBy}: \text{bag}(\alpha \times \beta) \rightarrow \text{bag}(\alpha \times \text{bag}(\beta)) \)
Implementation of MapReduce

- MapReduce \((m, r)S\) is implemented in Hadoop as a single map-reduce job:

```java
class Mapper
    method map (key, value):
        for each \((k, v) \in m(value)\) do emit(k, v);

class Reducer
    method reduce (key, values):
        \(B \leftarrow \emptyset\);
        for each \(w \in values\) do \(B \leftarrow B \cup \{w\}\);
        for each \(v \in r(key,B)\) do emit(key, v);
```

- For most cases, \(B\) is not materialized (using streaming)
Reduce-Side Join

Reduce-side join: \( \text{MapReduce2} ( m_x, m_y, r ) ( X, Y ) \)

\[
\begin{align*}
\text{select } w \\
\text{from } z \text{ in } ( \text{select } r(x',y') \\
\text{from } x \text{ in } X, \\
y \text{ in } Y, \\
(kx,x') \text{ in } m_x(x), \\
(ky,y') \text{ in } m_y(y) \\
\text{where } kx = ky \\
\text{group by } k: kx, \\
w \text{ in } z 
\end{align*}
\]

Functional parameters:
- \( m_x \): left map
- \( m_y \): right map
- \( r \): reduce

Implemented in Hadoop as a single map-reduce job that uses two map classes
- After map: X values are tagged with 0, Y values are tagged with 1
- The reducer separates the X from the Y values and calls \( r \)
Example

- Query:

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

- Physical plan:

```
MapReduce(\lambda p. select ( ( i.@category,count(p.watches.@open_auctions) ), p )
    from i in p.profile.interest,
    \lambda((cat,os),ps). { ( cat, os, count(ps) ) } )
( XMark )
```
The MRQL Optimizer

- Based on an algebra that uses a generalized join operator that does data nesting during the join
  - avoids the need for an explicit group-by to nest data for a nested query
- It uses a novel cost-based optimization framework to map the algebraic forms to efficient workflows of physical plan operators
  - uses a polynomial heuristic algorithm for query graph reduction
  - handles deeply nested queries, of any form and at any nesting level, and converts them to near-optimal join plans
  - optimizes dependent joins (used when traversing nested collections and XML data)
- There is no cost model yet
  - We plan to develop an adaptive optimization method to
    - incrementally reduce the query graph at run-time
    - enhance the reduce stage of a map-reduce operation to generate enough statistics to decide about the next graph reduction step
Current Status

- MRQL is implemented in Java on top of Hadoop 0.21.0
  [http://lambda.uta.edu/mrql/](http://lambda.uta.edu/mrql/)

- Can process the full MRQL syntax

- Can process two kinds of text documents:
  - XML documents
  - record-oriented text documents that contain basic values separated by user-defined delimiters

- Supports two kinds of joins:
  - reduce-side join
  - fragment-replicate join (using Hadoop's distributed cache)

- Supports four processing modes:
  - in-memory evaluation using Java vectors
  - Hadoop single node deployment
  - Hadoop cluster deployment
  - compilation to Java code for Hadoop cluster deployment