An Optimization Framework for Map-Reduce Queries

Leonidas Fegaras

University of Texas at Arlington

http://lambda.uta.edu/mrql/

Work in collaboration with Chengkai Li and Upa Gupta
Data Processing with Map-Reduce (MR)

- MR 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. It hides the details of parallelization, data distribution, fault-tolerance, and load balancing.

- Several implementations:
  - Apache Hadoop, Google Sawzall, Microsoft Dryad, ...

- Used extensively by companies on a very large scale:
  - Yahoo! manages more than 42,000 Hadoop nodes holding 200 PBs
  - Yahoo!'s biggest cluster: 4,000 nodes
  - More than 22 organizations are running PB-scale Hadoop clusters

- Some higher-level languages that make MR programming easier:
  - HiveQL, PigLatin, Scope, Dryad/Linq, ...

- Preferred by MR programmers:
  - Pig is used for over 60% of Yahoo! MR jobs
  - Hive is used for 90% of Facebook MR jobs
Alternatives to MR

- Compared to a parallel RDB, the MR framework:
  - is better suited to large-scale data analysis on write-once in-situ data
  - often used to process data as is
  - offers better fault tolerance and the ability to operate in heterogeneous environments
- An MR alternative: BSP (Bulk Synchronous Parallel Architecture)
  - Apache Hama
  - Pregel for graph processing
  - still work in progress
  - supports a more flexible API
  - needs checkpointing for fault-tolerance

An Optimization Framework for Map-Reduce Queries http://lambda.uta.edu/mrql/
The MR Programming Framework

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
An Example with 3 Mappers and 2 Reducers

mapper 0

mapper 1

mapper 2

data split 0

m m m 3 v1
m m m 5 v2
m m m 3 v3

mapper 0

mapper 1

mapper 2

data split 1

m m m 4 v4
m m m 5 v5
m m m 2 v6

mapper 0

mapper 1

mapper 2

data split 2

m m m 2 v7
m m m 3 v8
m m m 3 v9

mapper 0

mapper 1

mapper 2

data split 0

m m m 3 v1
m m m 5 v2
m m m 3 v3

mapper 0

mapper 1

mapper 2

data split 1

m m m 4 v4
m m m 5 v5
m m m 2 v6

mapper 0

mapper 1

mapper 2

data split 2

m m m 2 v7
m m m 3 v8
m m m 3 v9

mapper 0

mapper 1

mapper 2

output 0

output 1

reducer 0

reducer 1

reducer 0

reducer 1

output 0

output 1

m : map  r : reduce

group-by key

partition function: key mod 2

An Optimization Framework for Map-Reduce Queries http://lambda.uta.edu/mrql/
Background: MR vs SQL

- MR programs are computationally complete
- Regular SQL (join, selection, projection, group-by, having, order-by) can be directly coded using workflows of MR jobs

Example:

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

can be coded in MR as:

```
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);
```
Background: Reduce-Side Join

Example:

\[
\begin{align*}
\text{select} & \ x.C, y.D \\
\text{from} & \ X \text{ as } x, Y \text{ as } y \\
\text{where} & \ x.A = y.B
\end{align*}
\]

can be coded in MR as:

```java
class Mapper1
    method map ( key, x )
        emit(x.A,(1,x));

class Mapper2
    method map ( key, y )
        emit(y.B,(2,y));

class Reducer
    method reduce ( key, values )
        for each (1, x) ∈ values
            for each (2, y) ∈ values
                emit(key,(x.C,y.D));
```

An Optimization Framework for Map-Reduce Queries http://lambda.uta.edu/mrql/
Motivation

Although the MR model is simple, it is hard to develop, optimize, and maintain non-trivial MR applications coded in a general-purpose programming language. To achieve good performance one needs to:

- tune many configuration parameters
- use custom serializers, comparators, and partitioners
- use special optimization techniques, such as in-mapper combining and in-reducer streaming

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.
Many higher-level languages:
HiveQL, PigLatin, PACT/Nephele, SCOPE, Dryad/Linq,…

*HadoopDB*: a hybrid scheme between MR and parallel databases

*Manimal*: analyzes the MR code to find opportunities for using $B^+$-tree indexes, projections, and data compression

*Asterix*: a scalable platform to store, manage, and analyze large volumes of semistructured data
  uses its own distributed data store, Hyracks

*YSmart*: intra-query optimization by factoring out correlated operations
Goal

Build a query processing system that translates SQL-like data analysis queries to efficient MR jobs

- HDFS as the physical storage layer
  - no indexing, no data partitioning/clustering
  - no normalization
  - no data statistics
- Hadoop as the run-time engine
  - no extensions

In the future, we may relax some of these restrictions
Goal

Build a query processing system that translates SQL-like data analysis queries to efficient MR jobs

- HDFS as the physical storage layer
  - no indexing, no data partitioning/clustering
  - no normalization
  - no data statistics
- Hadoop as the run-time engine
  - no extensions

In the future, we may relax some of these restrictions

But, if we use SQL and completely hide the MR layer, why don’t we just use a parallel RDB?

- MR is already used extensively; we can’t change this
- a good query processor may simplify and improve the way programmers develop data analysis applications and will make MR computing friendlier to non-expert programmers
- MR is actually good!
What is Wrong with Existing MR Query Languages?

Two major problems (to be justified next):

1. Current MR query languages have limited expressive power, forcing users to plug-in custom MR scripts into their queries. This may result to suboptimal, error-prone, and hard-to-maintain code.

2. Current MR query processors apply traditional relational query optimization techniques that may be suboptimal in a MR environment.
A MR job over a relation \( R \) groups the tuples of \( R \) using the map function \( m \) and then applies the reduce function \( r \) to each group:

```sql
select k, r(group-values)
from R as v
group by k: m(v)
```

Current MR SQL-like query languages do not allow \( m \) and \( r \) to be nested queries and do not support access to the entire group, \textit{group-values}, other than performing simple aggregations over the group elements.

Example: calculate one step of the k-means clustering algorithm by deriving \( k \) new centroids from the old

```sql
select avg(s.X) as X, avg(s.Y) as Y, avg(s.Z) as Z
from Points as s
group by (select * from Centroids as c order by distance(c,s))[0]
```
How can we Reach MR-Completeness?

A MR query language must
- allow nested queries and UDFs at any level and at any place
  provided that UDFs are pure (no side effects)
- allow to operate on all the grouped data using queries
  as is done for ODMG OQL and XQuery
- support custom aggregations/reductions using UDFs
  provided that they are pure, associative, and commutative
- support recursion or transitive closure declaratively
  to capture graph algorithms, such as PageRank
- support hierarchical data and nested collections uniformly
  allowing us to query JSON and XML data
- support custom parsing and custom data fragmentation
  given that MR does not support nested data parallelism
What is Wrong with Existing MR Query Languages?

Two major problems:

1. Current MR query languages have limited expressive power, forcing users to plug-in custom MR scripts into their queries. This may result in suboptimal, error-prone, and hard-to-maintain code.

An Optimization Framework for Map-Reduce Queries http://lambda.uta.edu/mrql/
What is Wrong with Existing MR Query Languages?

Two major problems:

2. Current MR query processors apply traditional relational query optimization techniques that may be suboptimal in a MR environment.
Example from TPCH:

```sql
select c.CUSTKEY, c.NAME, \textit{avg}(o.TOTALPRICE)
from Orders as o, Customer as c
where o.CUSTKEY = c.CUSTKEY
group by c.CUSTKEY, c.NAME
```

Hive evaluates this query using a join followed by a MR job (for the group-by), a total of 2 MR jobs.

This query can also be evaluated using just one reduce-side join.

the set of orders fed to the reducer contains complete groups, which are ready for aggregation.

This is true for all queries whose join attributes are also group-by attributes.

Can we patch a relational system to generate this join/group-by plan?
By definition, a relational operation must return flat tuples

\[ \Rightarrow \] A group-by must always be combined with aggregation within the same operator

Consider nested queries (which are important for MR-completeness).

For simple correlated queries, such as

\[
\begin{align*}
\text{select } & f(\text{select } h(x,y) \\
& \text{ from } Y \text{ as } y \text{ where } x.A = y.B) \\
& \text{ from } X \text{ as } x
\end{align*}
\]

(where \( f \) and \( h \) denote some code)

it is not inconceivable that an RDB optimizer could be patched to generate a reduce-side join that incorporates the group-by at the reduce stage
Can we Patch an RDB ...?

But what about this double-nested query:

```
select f(select g(select h(x,y,z) from Z as z where z.C=y.D)
       from Y as y where x.A=y.B)
from X as x
```

One good plan is to use two reduce-side joins (2 MR jobs only):

1. a join between X and Y that emits a nested set of pairs (x,ys):
   for each x, the set ys contains all the matching values y

2. a join between the result and Z, which computes the final query result
   at the reduce stage

The reducer of the 2\textsuperscript{nd} join receives two sets (mixed) from the two inputs: the set XY from the first join, which is the nested set (x, ys), and the set Zs of z tuples, and evaluates in memory:

```
select f(select g(select h(x, y, z) from z ∈ Zs)
       from y ∈ ys)
from (x, ys) ∈ XY
```
The reduce-side join plan again:

1. A join between X and Y that emits a nested set of pairs \((x, y_s)\):
   - For each \(x\), \(y_s\) contains all the matching values \(y\)
2. A join between the result and Z, which computes the final query result at the reduce stage

It is impossible to capture the first join as a purely relational operator

⇒ need nested relations!

Side note:

- Some MR query languages use outer-joins with group-bys to simulate nested queries
- Bad idea!
  - May miss opportunities of using a combined join/group-by
MRQL: the Map-Reduce Query Language

Oh great, yet another query language!

- The MRQL syntax has been influenced by some functional query languages, such as ODMG OQL and XQuery
- The MRQL semantics is based on list comprehensions with group-by and order-by
- It is implemented in Java on top of Hadoop
- Allows arbitrary query nesting, UDFs, custom aggregations, and custom parsers
- Can operate on complex data, such as nested collections and trees
- Can process:
  - record-oriented text documents that contain basic values separated by user-defined delimiters
  - XML and JSON documents
  - binary encoded documents

Note: This work is about optimizing MR queries. It can apply to other suitable languages, such as OQL and XQuery
A MR job:

\[ \text{MapReduce}(m, r) \ S \]

transforms a data set \( S \) of type \( \{\alpha\} \) into a data set of type \( \{\beta\} \) using a map function \( m \) and a reduce function \( r \) with types:

\[
\begin{align*}
m & : \alpha \to \{(\kappa, \gamma)\} \\
r & : (\kappa, \{\gamma\}) \to \{\beta\}
\end{align*}
\]

Semantics:

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

where:

\[
\begin{align*}
\text{concatMap}(f) & : \{\alpha\} \to \{\beta\}, \text{ given that } f : \alpha \to \{\beta\} \\
\text{groupBy} & : \{(\kappa, \alpha)\} \to \{(\kappa, \{\alpha\})\}
\end{align*}
\]

concatMap generalizes \( \pi, \sigma, \) and \( \mu \)
MRQL is MR-Complete

Any

\[ \text{MapReduce}(m, r) S \]

can be expressed in MRQL as:

\[
\begin{align*}
\text{select} & \quad w \\
\text{from} & \quad z \ \text{in} \ ( \text{select} \quad r(\text{key}, \text{y}) \\
& \quad \text{from} \quad x \ \text{in} \ S, \\
& \quad (k, y) \ \text{in} \ m(x) \\
& \quad \text{group by} \quad \text{key: k}), \\
& \quad w \ \text{in} \ z
\end{align*}
\]
The reduce-side join

ReduceSideJoin\( (m_x, m_y, r) (X, Y) \)

joins the data set \( X \) of type \( \{\alpha\} \) with the data set \( Y \) of type \( \{\beta\} \) to form a data set of type \( \{\gamma\} \), where

\[
\begin{align*}
m_x &: \alpha \rightarrow \{(\kappa, \alpha')\} \\
m_y &: \beta \rightarrow \{(\kappa, \beta')\} \\
r &: (\{\alpha'\}, \{\beta'\}) \rightarrow \{\gamma\}
\end{align*}
\]

The mappers \( m_x \) and \( m_y \) calculate the join keys \( \kappa \) and the reducer \( r \) combines the tuples from \( X \) and \( Y \) that correspond to the same join key. Its semantics is given in terms of concatMap, groupBy, and union.

Other join implementations: MapJoin (1 map job), MapJoinReduce (1 MR job), and BlockNestedLoop (1 map job)
Most important algebraic operators: concatMap, groupBy, union, and join

The MRQL join is a restricted version of ReduceSideJoin. It joins the bag $X$ of type $\{\alpha\}$ with the bag $Y$ of type $\{\beta\}$ to form a bag of type $\{\gamma\}$:

$$\text{join}(k_x, k_y, r)(X, Y)$$

where

$$
\begin{align*}
k_x & : \alpha \rightarrow \kappa \\
k_y & : \beta \rightarrow \kappa \\
r & : (\{\alpha\}, \{\beta\}) \rightarrow \{\gamma\}
\end{align*}
$$
Algebraic Optimization

Some optimizations:

- Fusing a join with a group-by if the group-by key is the same as the join key:

  \[
  \text{join}( \pi_1, k_y, r ) \ ( \text{groupBy}(X), Y ) \\
  = \text{join}( \pi_1, k_y, \lambda(xs, ys).r( \text{groupBy}(xs), ys ) ) \ ( X, Y )
  \]

  where \( \pi_1(x, y) = x \)

- Converting a self-join into a simple MapReduce operation that operates over the input data set once:

  \[
  \text{join}( k_x, k_y, r ) \ ( X, X ) \\
  = \text{MapReduce}( \lambda x. \{(k_x(x), (1, x)), (k_y(x), (2, x))\}, \\
  \lambda(k, s). r( \text{select } x \text{ from } (1, x) \in s, \\
  \text{select } x \text{ from } (2, x) \in s ) ) \ X
  \]
Example: The PageRank Algorithm

A web graph is represented as a set of links, where each link has a source, a destination, the total number of its outgoing links, and its current PageRank.

One step of the PageRank algorithm derives a new set of edges from the old set, changing only their rank:

```
SELECT m.source, m.dest, m.count, c.rank
FROM (SELECT n.dest, SUM(n.rank/n.count) AS rank
      FROM Graph AS n
      GROUP BY n.dest) AS c,
       Graph AS m
WHERE m.source = c.dest
```

(SQL query)

Needs just 1 MR job:

- fuse the join with the group-by ⇒ a self-join over Graph
- convert the self-join to a single MR job
The Complete PageRank in MRQL

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init</td>
<td>Preprocessing: 1 MR job</td>
</tr>
<tr>
<td>Step</td>
<td>Repeat step: 1 MR job</td>
</tr>
<tr>
<td>Post</td>
<td>Postprocessing: 1 MR job</td>
</tr>
</tbody>
</table>

```ml
graph = select (key, n.to)
    from n in source(line, "graph.csv", ...)
    group by key: n.id;

size = count(graph);

select (x.id, x.rank)
from x in (repeat
    nodes = select <id: key, rank: 1.0/size, adjacent: al>
        from (key, al) in graph
    init step: 1 MR job

    step select (<id: m.id, rank: n.rank, adjacent: m.adjacent>,
        abs((n.rank-m.rank)/m.rank) > 0.1)
        from n in (select <id: key, rank: 0.25/size+0.85*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)

order by x.rank desc;
```
The MRQL Query Optimizer

- uses a novel cost-based optimization framework to map 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
- handles dependent joins (used for nested collections and XML data)

Our cost model is currently incomplete
We plan to develop an adaptive optimization system to

- incrementally reduce the query graph at run-time
- extend the reduce stage of a map-reduce operation to generate enough statistics to decide about the next graph reduction step
How does MRQL Compare with Hive?

- For simple join/group-by queries: they are about the same
- For queries that need optimization, such as fusing joins with group-bys, self-joins etc: MRQL is a clear winner
- For complex/nested queries, nested data, complex aggregations: no competition
Performance Evaluation

PageRank evaluation over DBLP (865MB XML) and the Hungarian Web (734MBs, 500K nodes 14M links)
Setup: 8 nodes/32 cores

![Graph A](B) PageRank over DBLP data with/without optimization

![Graph B](C) PageRank over the Hungarian Web with/without optimization
Future Work

- Develop a comprehensive cost model
- Self-tuning
- Want to query both raw data and structured data, such as RDBs and key/value indexes, in the same query language
- Want to capture scientific data & computations
  - need to introduce the concept of data neighborhood
  - need to be able to access ‘adjacent’ data (e.g., for data smoothing)
- Want to define complex custom parsers declaratively
  - need to go beyond Regular Expressions to capture LALR grammars