Supporting Bulk Synchronous Parallelism in Map-Reduce Queries

Leonidas Fegaras

University of Texas at Arlington

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

DataCloud 2012

Data Processing Using 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

  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:

  Hive, Pig, 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

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

Drawback:

- The I/O of a MR job is done through DFS (Distributed File System)

It simplifies fault-tolerance but imposes a high overhead for

- complex MR workflows
- repetitive MR jobs

  needed for graph analysis, such as PageRank

The Bulk Synchronous Parallelism (BSP) Model

- Introduced by Leslie Valliant in 1989
- A BSP program is a parallel computation that consists of a sequence of *supersteps*
  - each superstep is evaluated in parallel by every peer
  - and consists of three stages:
    1. a local computation
    2. a process communication
    3. barrier synchronization
- Implementations for cloud computing:

  Google's Pregel, Apache's Giraph and Hama
- Drawback:

  the local state of each peer must not exceed its memory capacity

  $\Rightarrow$ the entire graph, as well as the auxiliary data needed for processing the graph, must fit in the total memory of the cluster

Goals

- Want to run large-scale data analysis programs in both modes: MR and BSP
  - without modifying the programs
  - if enough resources are available and performance is preferred over resilience $\Rightarrow$ BSP
  - ... otherwise $\Rightarrow$ MR
- This can be done if data analysis programs were expressed in a higher-level declarative form
  - hides implementation details
  - offers opportunities for optimization
  - easier to learn, maintain, adapt
  - preferred by many programmers anyway
- Our goal is to translate and optimize MRQL queries to BSP jobs

  translating MRQL to MR workflows is done in our earlier work
- Our approach is to translate the already optimized MR plans generated by MRQL to BSP

Related Work

- Many higher-level MR languages:

  Hive, Pig, PACT/Nephele, SCOPE, Dryad/Linq,...
- Systems that improve the evaluation of MR workflows and repetitive jobs:

  Haloop, Twister, SystemML, ...
- Systems that use distributed memory for parallel computing:

  Main Memory MR (M3R), Piccolo, GraphLab, ...
- *Spark*: provides primitives for in-memory cluster computing
  - based on Resilient Distributed Datasets (RDDs):
    fault-tolerant, parallel data structures to explicitly share intermediate results in memory
  - *Shark*: runs Hive queries on Spark
- *Asterix*: a scalable platform to store, manage, and analyze large volumes of semistructured data

  uses its own distributed data store, Hyracks
  translates iterative jobs to Datalog, then to Hyracks operations

MRQL: the Map-Reduce Query Language

- Has SQL-like syntax, but is far more powerful than SQL
- Has been influenced by functional query languages, such as OQL and XQuery
- Its semantics is based on list comprehensions with group-by/order-by
- 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
  - XML and JSON documents
  - binary encoded documents
- Uses a novel cost-based optimization framework to map algebraic forms to efficient workflows of physical plan operators
- Handles deeply nested queries, of any form and at any nesting level
- Handles dependent joins (used for nested collections)

Matrix Multiplication in MRQL

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

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

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

MRQL evaluates it using two MR jobs

Complex algorithms, such as matrix factorization, require many repetitive matrix multiplications
MRQL may fuse consecutive computations, eliminating intermediate matrices

K-Means Clustering in MRQL

It clusters the data points by their closest centroid, and, for each cluster, a new centroid is calculated from the average values of its points

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

MRQL needs one MR job for each repeat step

PageRank in MRQL

Each node in nodes has an id, a PageRank, and outgoing links:

$$< \text{id: int, rank: float, links: } \{ \text{ int } \} >$$

```
repeat nodes = ...
  step select ( < id: m.id, rank: n.rank, links: m.links >,
                abs((n.rank-m.rank)/m.rank) > 0.1 )
       from n in (select < id: key, rank: sum(c.rank) >
                    from c in ( select < id: a, rank: n.rank/count(n.links) >
                                  from n in nodes, a in n.links )
                   group by key: c.id),
            m in nodes
      where n.id = m.id
```

MRQL needs one MR job for each repeat step

The MR Operation

A MR job:

$$\text{mapReduce}(m, r)\, S$$

transforms a data set S of type $\{\alpha\}$ into a data set of type $\{\beta\}$.

Types:

m:	$\alpha \rightarrow \{\, (\kappa, \gamma) \,\}$	*the map function*
r:	$(\kappa, \{\gamma\}) \rightarrow \{\beta\}$	*the reduce function*

Semantics:

$$\text{mapReduce}(m, r)\, S = \text{cmap}(r)\,(\text{groupBy}(\text{cmap}(m)\, S))$$

where:

$\text{cmap}(f) : \; \{\alpha\} \rightarrow \{\beta\}$, given that $f : \; \alpha \rightarrow \{\beta\}$

$\text{groupBy} : \; \{(\kappa, \alpha)\} \rightarrow \{(\kappa, \{\alpha\})\}$

The BSP Operation

A BSP job:

bsp(superstep, initstate) S

maps a dataset S of type V into a new dataset of type V, by repeating a superstep of type:

superstep: $(\{M\}, V, K) \rightarrow (\{(I, M)\}, V, K, \text{boolean})$

The superstep is evaluated by every peer participating in the BSP computation and

- it maps the peer's local snapshot V to a new local snapshot V
- it receives messages $\{M\}$ and sends messages $\{(I, M)\}$ to peers I
- the superstep logic is controlled by a DFA:
    - the DFA state is of type K
    - a superstep makes a transition from a state K to a new state K
    - the initial state is initstate

BSP Synchronization

The superstep type again:

superstep: $(\{M\}, V, K) \rightarrow (\{(I, M)\}, V, K, \text{boolean})$

Synchronization:

- The returned flag indicates whether the peer wants to terminate this BSP computation or use barrier synchronization and continue by repeating the superstep
  - only if *all* peers agree to exit (when they all return true), then every peer should exit the BSP computation
  - if there is at least one peer who wants to continue, then every peer must do barrier synchronization and repeat the superstep

MR-to-BSP Transformation

A MR job can be evaluated using a BSP job that has two supersteps: one for the map and one for the reduce task:

$$
\begin{array}{l}
\text{mapReduce}(m, r)\ S \\
= \text{bsp}(\ \lambda(ms, as, is\_map).\ \textbf{if}\ is\_map \\
\qquad\qquad\qquad\qquad \textbf{then}\ (\ \text{cmap}(\lambda(k,c).\ \{\ (\ \text{shuffle}(k), (k,c)\ )\ \}) \\
\qquad\qquad\qquad\qquad\qquad\qquad (\text{cmap}(\lambda(k,c).\ m(c))\ as), \\
\qquad\qquad\qquad\qquad\qquad \{\ \}, \text{false}, \text{false}\ ) \\
\qquad\qquad\qquad\qquad \textbf{else}\ (\ \{\ \}, \text{cmap}(r)\ (\text{groupBy}(ms)), \text{false}, \text{true}\ ), \\
\qquad \text{true}\ )\ S
\end{array}
$$

The DFA state is a boolean flag *is_map* that indicates whether we are in map or reduce mode

The map step shuffles the map results to the reducers, using the function shuffle that maps a key k to a peer

Mapping Repeat Loops

repeat $S = S_0$
  step loopstep(S)

Suppose that the loopstep is translated to a BSP operation:

$\text{loopstep}(S) = \text{bsp}(s, k_0)\ S$

where s is the superstep. Then, the repeat loop can be translated to:

$$
\begin{array}{l}
\text{bsp}(\ \lambda(ms, vs, k).\ \textbf{let}\ (ts, bs, k, exit) \leftarrow s(ms, vs, k) \\
\qquad\qquad \textbf{in if}\ exit \\
\qquad\qquad\quad \textbf{then}\ (\ \{\ \}, \\
\qquad\qquad\qquad\quad \text{cmap}(\lambda(t,(v,b)).\ \{(t,v)\})\ bs, \\
\qquad\qquad\qquad\quad k_0, \\
\qquad\qquad\qquad\quad \forall(t,(v,b)) \in bs : \neg b\ ) \\
\qquad\qquad\quad \textbf{else}\ (\ ts,\ bs,\ k,\ \text{false}\ ), \\
\quad k_0\ )\ S
\end{array}
$$

Normalization

Translating an MRQL query to a BSP job:

1. the query is translated and optimized to a workflow of MR jobs that consists of MR algebraic forms
2. each MR algebraic form is mapped to a BSP job
3. the BSP workflow is normalized to a single BSP job

$$
\begin{array}{l}
\mathrm{bsp}(s_2, k_2)\ (\ \mathrm{bsp}(s_1, k_1)\ S\ ) \\
= \mathrm{bsp}(\ \lambda(ms, as, (\mathit{first}, k)). \\
\qquad\qquad\qquad \textbf{if}\ \mathit{first} \\
\qquad\qquad\qquad \textbf{then let}\ (ts, bs, k', b)\ \leftarrow\ s_1(ms, as, k), \\
\qquad\qquad\qquad\qquad\qquad \mathrm{exit} \leftarrow \mathrm{synchronize}(b) \qquad\qquad \textit{// poll peers if they} \\
\qquad\qquad\qquad\qquad \textbf{in}\ (\ ts,\ bs, \qquad\qquad\qquad\qquad\qquad\quad \textit{// all agree to exit} \\
\qquad\qquad\qquad\qquad\qquad (\ \neg\mathrm{exit},\ \textbf{if}\ \mathrm{exit}\ \textbf{then}\ k_2\ \textbf{else}\ k'\ ), \\
\qquad\qquad\qquad\qquad\qquad \mathrm{false}\ ) \\
\qquad\qquad\qquad \textbf{else let}\ (ts, bs, k', b)\ \leftarrow\ s_2(ms, as, k) \\
\qquad\qquad\qquad\qquad \textbf{in}\ (\ ts,\ bs,\ (\ \mathrm{false},\ k'\ ),\ b\ ), \\
\qquad\quad (\mathrm{true}, k_1)\ )\ S
\end{array}
$$

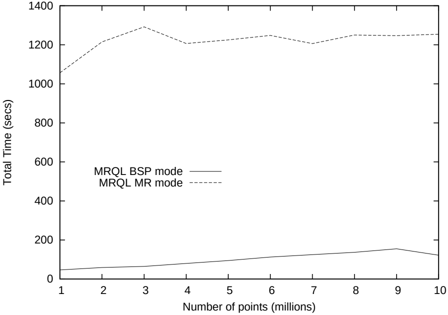
Current Status

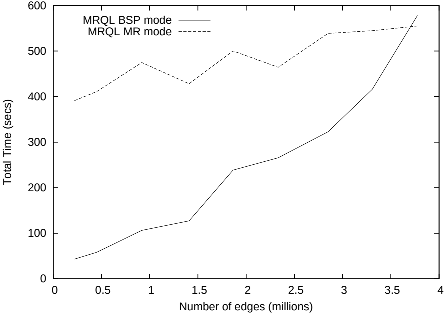
- MRQL is implemented on top of Apache Hadoop and Hama
- It is available at `http://lambda.uta.edu/mrql/`
- Open source at `https://github.com/fegaras/mrql`
- It can execute MRQL queries in two modes:
  - using the MR framework on Apache Hadoop, or
  - using the BSP framework on Apache Hama

Performance Evaluation

Setup: 8 nodes/32 cores

K-Means clustering (left) and PageRank (right) using MR and BSP:





Future Work

- Develop a cost model to decide between MR and BSP, based on available resources
- Apply MRQL to large-scale scientific data analysis

    array-based query optimization
- Map MRQL to Spark or Hyracks