Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters

Hung-chih Yang, Ali Dasdan
Yahoo!
Ruey-Lung Hsiao, D. Stott Parker
UCLA
Outline

1. Introduction
2. Map-Reduce
3. Map-Reduce-Merge: extending Map-Reduce
   • Implementation
4. Applications to Relational Data Processing
5. Optimizations & Enhancements
6. Case Studies
7. Conclusion
Introduction

• New challenges of data processing
  • Vast amount of data collected from the World Wide Web
  • Cannot rely on generic DBMS to reduce costs and improve efficiency

• Solutions of Search Engine companies
  • Use customized parallel data processing systems
  • Use large clusters of shared- nothing commodity nodes
  • Eg: Google’s GFS, Map-Reduce
    • Microsoft’s Dryad
    • Yahoo!’s Hadoop (open-source)
Introduction

• Properties of Data-intensive systems
  • Simple
    • Adopt only a selected subset of database principles
  • Generic and cost effective
  • Deployed on large clusters of shared nothing commodities
  • Refactoring of data processing into two primitives:
    • Map function
    • Reduce function
  • Map-Reduce allows users not to worry about the nuisance details
    • Coordinating parallel sub-tasks
    • Maintaining distributed file storage
Motivation

• Map-Reduce framework is best at handling *homogeneous datasets*
  • Joining multiple heterogeneous datasets is not efficient in map-reduce
• Extending Map-Reduce to process heterogeneous datasets simultaneously
• Join-enabled map-reduce systems can provide a parallel and cost effective alternative
  • Can include relational algebra in the subset of database principles
Map-Reduce

- Input dataset stored in GFS
  - Mapper
    - Read splits of input dataset
    - Apply map function to input records
    - Produce intermediate key/value sets
    - Partition the intermediate sets into no of reducers sets
  - Reducer
    - Read their part of intermediate sets from mappers
    - Apply reduce function to the values of a same key
    - Output final results

Signature of Map-Reduce function:

Map: \((k1, v1) \rightarrow [(k2, v2)]\)

Reduce: \((k2, [v2]) \rightarrow [v3]\)
Join Using Map Reduce

• Use homogenization procedure
  • Apply one map/reduce task on each dataset
  • Insert a data-source tag into every value
  • Extract a key attribute common for all heterogeneous datasets
  • Transformed datasets now have two common attributes
    • Key and data-source

• Problems
  • Take a lot of extra disk space and incur excessive map-reduce communications
  • Limited only to queries that can be rendered as equi-joins
Join using Map-Reduce: Homogenization

Collect records with same key

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1, &quot;Value1&quot;</td>
</tr>
<tr>
<td>85</td>
<td>1, &quot;Value2&quot;</td>
</tr>
<tr>
<td>320</td>
<td>1, &quot;Value3&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>2, &quot;Value4&quot;</td>
</tr>
<tr>
<td>54</td>
<td>2, &quot;Value5&quot;</td>
</tr>
<tr>
<td>320</td>
<td>2, &quot;Value6&quot;</td>
</tr>
</tbody>
</table>

Dataset 1

Dataset 2
Why go to Map-Reduce-Merge?

• Map-Reduce cant support relational algebra efficiently without sacrificing the existing generality and simplicity.
• Need to process heterogeneous datasets simultaneously
• The existing join technique takes lots of extra disk space, incurs excessive map-reduce communications and limited to queries that are equi-join.
• By adding a merge phase to this process, a variety of hierarchical workflows for data processing can be achieved
• Can embed programming logic into each phase.
Map-Reduce-Merge

Signatures

\[
\begin{align*}
\text{map: } (k_1, v_1)_\alpha & \rightarrow [(k_2, v_2)]_\alpha \\
\text{reduce: } (k_2, [v_2])_\alpha & \rightarrow (k_2, [v_3])_\alpha \\
\text{merge: } ((k_2, [v_3])_\alpha, (k_3, [v_4])_\beta) & \rightarrow [(k_4, v_5)]_\gamma
\end{align*}
\]

- $\alpha, \beta, \gamma$ represent dataset lineages
- Reduce function produces a key/value list instead of just values
- Merge function reads data from both lineages

These three primitives can be used to implement the parallel version of several join algorithm
Example

Algorithm 1 Map function for the Employee dataset.
1: map(const Key& key, /* emp.id */
2: const Value& value /* emp.info */) {
3:   emp.id = key;
4:   dept.id = value.dept.id;
5:   /* compute bonus using emp.info */
6:   output.key = (dept.id, emp.id);
7:   output.value = (bonus);
8:   Emit(output.key, output.value);
9: }

Algorithm 2 Map function for the Department dataset.
1: map(const Key& key, /* dept.id */
2: const Value& value /* dept.info */) {
3:   dept.id = key;
4:   bonus.adjustment = value.bonus.adjustment;
5:   Emit((dept.id), (bonus.adjustment));
6: }

Algorithm 3 Reduce function for the Employee dataset.
1: reduce(const Key& key, /* (dept.id, emp.id) */
2: const Value& value /* dept.info */) {
3:   /* an iterator for a bonuses collection */
4:   bonus.sum = /* sum up bonuses for each emp.id */
5:   Emit(key, (bonus.sum));
6: }

Algorithm 4 Reduce function for the Department dataset.
1: reduce(const Key& key, /* (dept.id) */
2: const Value& value /* dept.info */) {
3:   /* an iterator on a bonus.adjustments collection */
4:   /* aggregate bonus.adjustments and */
5:   compute a final bonus.adjustment */
6:   Emit(key, (bonus.adjustment));
7: }
Implementation of Merge Modules

• Partition Selector
  • Determine from which reducers this merger retrieves its input data based on the merger number

• Processor function
  • Process data from one source only
  • Users can define two processor functions

• Merger function
  • Process two pairs of key/values

• Configurable iterator
  • A merger has two logical iterators
  • Control their relative movement against each others
Applications to Relational Data Processing

Map-Reduce-Merge can be used to implement primitive and derived relational operators:

1. Projection
2. Aggregation
3. Selection
4. Set Operations: Union, Intersection, Difference
5. Cartesian Product
6. Rename
7. Join
Map-Reduce-Merge Implementations of Relational Join Algorithms

- Sort-Merge Join
- Hash Join
- Block Nested-Loop Join

Eg: Hash Join
Optimizations and Enhancements

• Optimal Reduce-Merge Connections
• Combining Phases
  • Reduce-Map, Merge-Map
  • Reduce-Merge
  • Reduce-Merge-Map

Enhancements:
• Map-Reduce-Merge Library
• Map-Reduce-Merge Workflow
Case Studies

• Join Web-Graph

• Map-Reduce-Merge Workflow for TPC-H Query-2
  • Involves 5 tables, 1 nested query, 1 aggregate and group by clause and 1 order by operator.
Case study: Map-Reduce-Merge Workflow for TPC-H Query 2

Figure 7: A join tree for TPC-H Query 2. It is implemented with 13 passes of Map-Reduce-Merge modules (10 mappers, 10 reducers, and 4 mergers).

Figure 8: The join tree of Fig. 7 is re-implemented with 6 passes of combined Map-Reduce-Merge modules (5 mappers, 4 reduce-merge-mappers, 1 reduce-mapper, and 1 reducer).
Conclusion

• MapReduce & GFS represent a paradigm shift in data processing: use a simplified interface instead of overly-general DBMS

• Map-Reduce-Merge adds the ability to execute arbitrary relational algebra queries

• Next steps:
  • Develop SQL-like interface and
  • A Query Optimizer