Big Data Analysis

Prof. Yanlei Diao
University of Massachusetts Amherst
I. Today’s Big Data Systems

• Want to process lots of data, unstructured or structured
• Want to parallelize across hundreds/thousands of commodity computers
  – New definition of cluster computing: large numbers of low-end processors working in parallel to solve a computing problem.
  – Parallel DB: a small number of high-end servers.
• Want to make this easy
MapReduce (Programming & Computation)

- Clean abstraction for programmers
- Automatic parallelization & distribution
- Fault-tolerance
- Status and monitoring tools

Programming Model

• Borrows from functional programming
• Users implement an interface of two functions:

  - map  (in_key, in_value) ->
    list(out_key, intermediate_value)

  - reduce (out_key, list(intermediate_value) ->
    list(out_value)
map

- Input: a key-value pair. E.g.,
  - A line out of files (filename, line),
  - A row of a database (row_id, row),
  - A document (doc_name, document)
- map( ) produces one or more intermediate values along with an output key from the input.
- map( ) is stateless: one input leaves no state that would affect the processing of the next input.
reduce

• After the map phase is over, all the intermediate values for a given output key are collected into a list
• \texttt{reduce()} combines those intermediate values into one or more \textit{final values} for that same output key
• \texttt{reduce()} can be \textit{stateful}: it operates on all the intermediate values of a certain key
Example: Count Word Occurrences

```java
map(String input_key, String input_value):
    // input_key: document name
    // input_value: document contents
    for each word w in input_value:
        EmitIntermediate(w, "1");

reduce(String output_key, Iterator intermediate_values):
    // output_key: a word
    // output_values: a list of counts
    int result = 0;
    for each v in intermediate_values:
        result += ParseInt(v);
    Emit(AsString(result));
```

How do we implement this using a relational DBMS? Customized data loading (data may be used only once), then Group By.
Click Stream Analysis: Page Frequencies

Clicks(time, url, referral_url, user_id, geo_info...)

map(String tuple_id, String tuple):
    EmitIntermediate(url, "1");

reduce(String url, Iterator list_tuples):
    int result = 0;
    for each t in list_tuples:
        result += ParseInt(t);
    Emit(AsString(result));

Select count(*)
From Clicks
Group By url;
MapReduce: Computation Model

Simple data partitioning

Map: input → list(k, v)

Groups:
- k1:v, k1:v, k2:v
- k1:v
- k3:v, k4:v
- k4:v, k5:v
- k4:v
- k1:v, k3:v

Logical partitioning: group all (key, value) by key

Group by Key

Grouped
- k1:v,v,v,v
- k2:v
- k3:v,v
- k4:v,v,v
- k5:v

Reduce: (k, list(v)) → list(v’)

Output

Yanlei Diao, University of Massachusetts Amherst
Example: Sessionization on click streams

Input

| <u1,t1,p7> | <u2,t2,p2> | <u3,t3,p4> | <u2,t4,p1> | <u2,t5,p6> | <u3,t6,p9> | <u1,t7,p2> |

Intermediate

| M | M | M | M | M | M | M |

Map: extracts key-value pairs
- Extracts user id as key
- Uses <time,page> as value

Group by Key

Grouped

| u1:<t1,p7>,<t7,p2> | u2:<t2,p2>,<t4,p1>,<t5,p6> | u3:<t3,p4>,<t6,p9> |

Reduce: for each group
- Breaks user clicks into sessions

Output

| u1:<t1,p7> | u1:<t7,p2> | u2:<t2,p2>,<t4,p1>,<t5,p6> | u3:<t3,p4>,<t6,p9> |

Yanlei Diao, University of Massachusetts Amherst
Optimization: Incremental Computation

• “Combiner” functions can be applied earlier, e.g., right after map() finishes on the same machine
• Causes a mini-reduce phase to occur before the real reduce phase, to save bandwidth
• Common examples: word frequency, url frequency
• Also called partial aggregation

Under what conditions is it sound to use a combiner?
Hadoop Implementation of MapReduce

Node 1

- Data Load
- Map ()
- Local Sort
- * Combiner
- Map Write

Node 2

- Data Load
- Map ()
- Local Sort
- * Combiner
- Map Write

Merge & * Combine
Reduce ()
Final Write

Shuffle

Partition w. a local sort
Multi-pass merge

* optional

Yanlei Diao, University of Massachusetts Amherst
Fault Tolerance

• Failure is the norm rather than exception!
  – Each node has prob. $p$ to fail. What is the chance that the job involving $n$ nodes fails?

• Fine-grained fault tolerance: *materialize* map output onto local disk before the map task completes

• Master detects worker failures
  – Re-executes completed & in-progress `map()` tasks
  – Re-executes in-progress `reduce()` tasks

• Master notices particular input key/values cause crashes in `map()`, and skips those values on re-execution.
  – Effect: Can work around bugs in third-party libraries!
Refinement: Exploiting Locality

- Master scheduling policy:
  - Asks GFS for locations of replicas of input file blocks
  - Map tasks typically split into 64MB (GFS block size)
  - Map tasks scheduled so GFS input block replica are on same machine or same rack

- Effect
  - Thousands of machines read input at local disk speed
  - Without this, rack switches limit read rate
II. Comparison to Parallel Databases

Let us consider structured data here.
- Of course, MapReduce can also handle text processing!

1. A closer look at internal implementation of MapReduce
   - Extract (key, value) using map()
   - Group data by key
   - Then apply reduce() to each group

2. Implementing relational operators using MapReduce
   - Parallel sorting?
   - Parallel Join?
   - Parallel group by-aggregation?

3. MapReduce query plans

Yanlei Diao, University of Massachusetts Amherst
1. Analysis of Open Source Hadoop

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>Number of reduce tasks per node</td>
</tr>
<tr>
<td>C</td>
<td>Map input chunk size</td>
</tr>
<tr>
<td>F</td>
<td>Merge factor that controls how often on-disk files are merged</td>
</tr>
</tbody>
</table>

**Symbol | Description**

(1) System Settings

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>Number of reduce tasks per node</td>
</tr>
<tr>
<td>C</td>
<td>Map input chunk size</td>
</tr>
<tr>
<td>F</td>
<td>Merge factor that controls how often on-disk files are merged</td>
</tr>
</tbody>
</table>

(2) Workload Description

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Input data size</td>
</tr>
<tr>
<td>K_m</td>
<td>Ratio of output size to input size for the map function</td>
</tr>
<tr>
<td>K_r</td>
<td>Ratio of output size to input size for the reduce function</td>
</tr>
</tbody>
</table>

(3) Hardware Resources

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Number of nodes in the cluster</td>
</tr>
<tr>
<td>B_m</td>
<td>Output buffer size per map task</td>
</tr>
<tr>
<td>B_r</td>
<td>Shuffle buffer size per reduce task</td>
</tr>
</tbody>
</table>

(4) Symbols Used in the Analysis

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>Bytes read and written per node, $U = U_1 + \ldots + U_5$ where $U_i$ is the number of bytes of the following types: 1: map input; 2: map internal spills; 3: map output; 4: reduce internal spills; 5: reduce output</td>
</tr>
<tr>
<td>S_i</td>
<td>Number of sequential I/O requests per node for IO type $i$</td>
</tr>
<tr>
<td>T</td>
<td>Time measurement for startup and I/O cost</td>
</tr>
<tr>
<td>h</td>
<td>Height of the tree structure for multi-pass merge</td>
</tr>
</tbody>
</table>

---

Yanlei Diao, University of Massachusetts Amherst
Sort-Merge Implementation of Group-By

1. Data Load
2. Map( )
3. Local Sort
4. * Combine
5. Map Write

... Map Task

... Map Task

Shuffle

... Reduce Task

... Reduce Task

* optional
Analysis of Multi-Pass Merge

- Used in a mapper for sorting if map output exceeds memory size
- Used in a reducer unless all data fits in memory

Yanlei Diao, University of Massachusetts Amherst
Effect of the Merge Factor F

Yanlei Diao, University of Massachusetts Amherst
2. Implementing Relational Operators

- **Selection**: R.a > “abc”
  - **ParallelDB**: if range partitioned, use a few nodes and indexes
  - **MapReduce**: scan all nodes, map() only.
    - *Can be dominated by start-up cost. No indexes.*

- Most other operators need **repartitioning** data:
  - **ParallelDB**: explicit partitioning function
  - **MapReduce**: more complicated
    1. Implicit partitioning function, \( \text{fn} \), controls data shuffling to reducers.
      (Default is hash partitioning. Can be changed to range partitioning.)
    2. Each reducer uses an additional mechanism to group data by the key.

- Consider the task to range partition data and sort data in each range. What is the key in the MR programming model?
Join Operators

• **Equijoin:** \( R.a = S.a \)
  - **ParallelDB:** parallel hash join.
    • I/O and network costs?
  - **MapReduce:** the programming interface is not natural for joins.
    1. map() annotates tuples with ‘r’ and ‘s’,
    2. the system groups all data by the join attribute using sort-merge,
    3. reduce() joins ‘r’ and ‘s’ tuples with the same value of the join attribute.
    • I/O and network costs?
    • *Is it better to change the programming model to make join more natural?*

• **Non-equijoin:** \( R.a < S.a \)
  - **ParallelDB:** fragment-replication
  - **MapReduce:** simulates fragment-replication. If replicate S,
    • replicate each S tuple \( m \) times in the mapper
    • tweak the partitioning function, \( fn \), for shuffling so that these \( m \) copies go to different reducers.
Group By Aggregation

- **Scalar aggregate**: count(), sum()
  - **ParallelDB**: partial aggregation + final aggregation
  - **MapReduce**: map() is empty; use combiner() for partial aggregation; use reduce() for final aggregation

- **Group by aggregation**: $G_{R.a, aggr(R.b)}$
  - **ParallelDB**: unary input version of parallel hash join
  - **MapReduce**:
    - map() simply emits tuples;
    - the system groups data by R.a;
    - reduce() computes sum.
    - should use the combiner() for partial aggregation earlier.
MapReduce Query Plans

- How many rounds of map reduce jobs?
- In each round, what is in map(), what is in reduce()?

```
SELECT DISTINCT U.city
FROM Users U, Clicks C
WHERE U.uid=C.uid
AND C.url LIKE '%google%';
```

**Round 1:**
**Key:** uid
**Map:** (1) selection, (2) create ‘u’, ‘c’ tuples with labels
**Reduce:** (1) join tuples within each group, (2) emit cities

```
≥
σ
U.uid=C.uid
Π
city
DupElim
```

**Round 2:**
**Key:** city
**Map:** emit
**Reduce:** emit a tuple in each group
More on Query Plans

```
SELECT url, count(*)
FROM    Clicks C
GROUP BY url
HAVING count(*) > 1000
ORDER BY count(*) DESC;
```

Round 2:
- **Key**: count
- **Map**: emit
- **Shuffle**: range partitioning (set manually)
- **Reduce**: local sort
  or (a simple but bad plan)
- **Key**: fixed
- **Map**: emit
- **Reduce**: sort all in a single reducer

Round 1:
- **Key**: url
- **Map**: emit
- **Reduce**: (1) count, (2) selection, (3) emit (url, count)

Second-order fn
→ a single tuple, or
→ a set of tuples (unnesting results)