Map Reduce (contd.)

CompSci 590.03
Instructor: Ashwin Machanavajjhala
Recap: Map-Reduce

\[
\text{map}\ (k1,v1) \rightarrow \text{list}(k2,v2); \\
\text{reduce}\ (k2,\text{list}(v2)) \rightarrow \text{list}(k3,v3).
\]
This Class

• High Level Languages for Map Reduce

• Join Processing
HIGH LEVEL LANGUAGES
Word Count in Pig

Load A = ‘documents’ USING PigStorage(‘\t’) AS (id, docstring)
// load the data using a built in loader assuming data is (id, document string) delimited by tabs

B = FOREACH A GENERATE FLATTEN(Tokenize(docstring)) AS word
// Mapper UDF Tokenize generates a set of words
// FLATTEN: flattens a set into multiple records.

C = GROUP B BY word
// groups the data by word

D = FOREACH C GENERATE group, COUNT(B)
// Built in reduce function counts the number of times each word appears in B

STORE D
A = load 'student' AS (name:chararray, age:int, gpa:float);

DESCRIBE A;
A: {name: chararray, age: int, gpa: float}

DUMP A;
(John, 18, 4.0F)
(Mary, 19, 3.8F)
(Bill, 20, 3.9F)
(Joe, 18, 3.8F)

B = GROUP A BY age;

DESCRIBE B;
B: {group: int, A: {name: chararray, age: int, gpa: float}}

ILLUSTRATE B;
etc ...

<table>
<thead>
<tr>
<th>B</th>
<th>group: int</th>
<th>A: bag({name: chararray, age: int, gpa: float})</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td></td>
<td>{John, 18, 4.0}, (Joe, 18, 3.8)</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>{Bill, 20, 3.9}</td>
</tr>
</tbody>
</table>

DUMP B;
(18, {John, 18, 4.0F}, (Joe, 18, 3.8F))
(19, {Mary, 19, 3.8F})
(20, {Bill, 20, 3.9F})
Pig UDFs

• All user defined functions are written in java.

```java
package myudfs;
import java.io.IOException;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.Tuple;
import org.apache.pig.impl.util.WrappedIOException;

public class UPPER extends EvalFunc<String>
{
    public String exec(Tuple input) throws IOException {
        if (input == null || input.size() == 0)
            return null;
        try{
            String str = (String)input.get(0);
            return str.toUpperCase();
        }catch(Exception e){
            throw WrappedIOException.wrap("Caught exception processing input row ", e);
        }
    }
}
```

• See http://wiki.apache.org/pig/UDFManual
Algebraic UDFs

• Aggregate functions take a bag and return a scalar value
• Some aggregate functions (e.g., associative and commutative operations) can be computed incrementally in a distributed fashion.

```java
public interface Algebraic{
    public String getInitial();
    public String getIntermed();
    public String getFinal();
}
```
Other functions

- **COGROUP** // group multiple tables on the same value
- **FILTER** // discard records that do not satisfy some property
- **UNION** // union of two tables
- **SAMPLE** // randomly sample each record with probability p
- **DISTINCT** // remove duplicates
- **LIMIT** // return a subset of n (not random)

- See [http://pig.apache.org/docs/r0.7.0/piglatin_ref2.html](http://pig.apache.org/docs/r0.7.0/piglatin_ref2.html)
A = LOAD 'data1' AS (owner:chararray,pet:chararray);

DUMP A;
(Alice,turtle)
(Alice,goldfish)
(Alice,cat)
(Bob,dog)
(Bob,cat)

B = LOAD 'data2' AS (friend1:chararray,friend2:chararray);

DUMP B;
(Cindy,Alice)
(Mark,Alice)
(Paul,Bob)
(Paul,Jane)

X = COGROUP A BY owner, B BY friend2;

(Alice,{(Alice,turtle),(Alice,goldfish),(Alice,cat)}),{(Cindy,Alice),(Mark,Alice)})
(Bob,{(Bob,dog),(Bob,cat)}),{(Paul,Bob)})
(Jane,{},{(Paul,Jane)})
JOIN

A = LOAD 'data1' AS (a1:int,a2:int,a3:int);

DUMP A;
(1,2,3)
(4,2,1)
(8,3,4)
(4,3,3)
(7,2,5)
(8,4,3)

B = LOAD 'data2' AS (b1:int,b2:int);

DUMP B;
(2,4)
(8,9)
(1,3)
(2,7)
(2,9)
(4,6)
(4,9)

X = JOIN A BY a1, B BY b1;

DUMP X;
(1,2,3,1,3)
(4,2,1,4,6)
(4,3,3,4,6)
(4,2,1,4,9)
(4,3,3,4,9)
(8,3,4,8,9)
(8,4,3,8,9)
JOIN PROCESSING
JOINs

- A = JOIN B BY fieldB, C BY fieldC PARALLEL 20
  - Specify the number of reduce tasks

- A = JOIN B BY fieldB, C BY fieldC USING ‘replicated’
  - Can ask the system to use one of three ways to do join.
Join Types

**Fragment Replicated Join:**
- When one of the tables is small enough to fit in memory.
- Replicate the “small” table to all mappers containing the other “large” table.

**Skewed Join:**
- When one of the join attributes is very skewed.
- Keys with large number of keys are split into multiple reducers.

**Merge Join:**
- When two datasets are already sorted on the join key
- Use sort merge join.
Join as an Optimization Problem

- Objective: minimize job completion time
- Cost at a reducer:

  - Input-size dominated: Reducer input processing time is large
  - Output-size dominated: Reducer output processing time is large
Join-Matrix

$M_{ij} =$ pair of tuples that have $S.key = i$ and $T.key = j$

$M_{ij}$ is shaded if corresponding tuples appear in the join output.

Goal: find a mapping between join matrix cells to reducers that minimizes completion time.
Join Alternatives

- Standard join algorithm
- Group both tables by key, send all tuples with the same key to a single reducer
- Skew in 7 leads to skewed execution times in reducers.

R1: keys 5, 8
   Input: S1, S4
         T1, T5
   Output: 2 tuples

R2: key 7
   Input: S2, S3
         T2, T3, T4
   Output: 6 tuples

R3: key 9
   Input: S5, S6
         T6
   Output: 2 tuples

max - reducer - input = 5
max - reducer - output = 6
Join Alternatives

- Fine grained load balancing
  - Divide the cells in the join matrix equally amongst the reducers

- Leads to replication of tuples to multiple reducers
  - S2, S3 are sent to all reducers.
Join Alternatives

- Best of both worlds
- 7 is broken down into two reducers
- Limits replication of input as well as reduces output skew.

R1: key 1
   Input:  S1,S2,S3
   T1,T2
   Output: 3 tuples

R2: key 2
   Input:  S2,S3
   T3,T4
   Output: 4 tuples

R3: key 3
   Input:  S4,S5,S6
   T5,T6
   Output: 3 tuples

max-reducer-input = 5
max-reducer-output = 4
Computing a join

• Identify the regions in the join matrix that appear in the join.
  – Sufficient to identify a superset of the shaded cells in the join matrix

• Map regions of the join matrix to reducers such that each shaded cell is covered by a reducer.

• Develop a Map-reduce algorithm to assign tuples to the corresponding reducers.
Approach 1: Cross Product

• Cross Product: all cells in the join matrix are shaded
  – Superset of any join condition
Cross Product

How to cover the cross product by $r$ reducers?

- Need to cover all $|S| \times |T|$ cells using $r$ reducers
  - Max reducer output size $\geq |S| \times |T| / r$
  - Therefore, Max reducer input size $\geq 2 \sqrt{|S| \times |T| / r}$

- We can match these lower bounds by assigning square regions from the join matrix of side $= \sqrt{|S| \times |T| / r}$ cells.
  - $|S|$ and $|T|$ must be multiples of $\sqrt{|S| \times |T| / r}$

- Algorithms in the paper for optimal mapping to reducers for any given $|S|, |T|, r$.
  - At most $4 \sqrt{|S| \times |T| / r}$ max reducer input and max reducer output.
Join Algorithm

- Assign row ids from \{1, 2, ..., |S|\} and \{1, 2, ..., |T|\} to all rows in S and T, resp.

- Map phase:
  For \(x \in S\), let \(R = \{r_1, ..., r_k\}\) be the regions intersecting row \(x\).id. Generate tuples: one tuple \((r, x)\) for each \(r \in R\) Similarly generate tuples for \(y \in T\).

- Reduce phase:
  Perform the join (or cross product) of all the tuples input to the reducer.
Join Algorithm: 1-Bucket-Theta

- Problem: Need a new map step to assign ids to rows in S and T
- Instead, on seeing a new tuple in S or T, assign a random row id.
1-Bucket-Theta

• Since every cell in the entire cross product is sent to some reducer, any join algorithm can be implemented
  – By applying the appropriate join condition.

• If evaluating a join requires at least an $x$ fraction of all cells in the join matrix, then max reducer input $\geq 2 \sqrt{x|S||T|/r}$.

• 1-Bucket-Theta has max reducer input $\leq 4 \sqrt{|S||T|/r}$

• Hence, at most a factor of $2/\sqrt{x}$ off
  – Works well as long as $x$ is large (at least 50%)
Approach 2: Approximate the Join Matrix

True join matrix

Histogram boundaries

Candidate cells to be covered by algorithm
Approach 2

• Need more detailed statistics about $|S|$ and $|T|$.

• Need to know something about the join predicate:
  – Doesn’t work for black-box join operators.
  – Need to identify which blocks contain 0 cells that appear in the join.
    – Equijoins, band-joins, inequality joins …

• Paper shows a heuristic technique to divide candidate cells into reducers.
Summary

• High level languages help write complex programs without thinking about map and reduce

• Join operations can be optimized by dividing the join matrix into regions.