“Map Reduce” Computing Paradigm .3

pm jat @ daiict

Announcement

- A List of research articles is made available at course site. May refer while planning your term papers https://moodle.daiict.ac.in/mod/resource/view.php?id=1579

Writing Map-Reduce programs for a problem!

- In map-reduce program, we always require breaking a problem in map-reduce tasks.
- In some cases like performing aggregate operations on a data file, map-reduce programming quiet straight forward.
- However as logic deviates from this, require iterations etc, solving it through map-reduce becomes bit challenging.
- Let us look into more examples!
  - JOIN, SORT, TOP-N, ?

Computing Join using Map-Reduce

- Consider following two files[3] (small files to keep it simple):

```
users(user_id, state_id) //user and state
transactions(prod_id, user_id)
```

    //products that a user bought

- And want to perform following operation (count number of states in which a product is sold):

```
SELECT product_id, count(distinct state_id)
    FROM transactions JOIN users
    ON transactions.user_id = users.user_id
    group by product_id;
```

- It requires Join and then aggregation.

Computing Join using Map-Reduce

- Said computation can be performed in two steps, i.e. “two map-reduce jobs” in pipeline
- MR Job-1: JOIN only

```
SELECT state_id, product_id FROM users u
JOIN transaction t ON u.user_id = t.user_id
```

- MR Job-2; aggregation
  - `SELECT product_id, count(distinct state_id) FROM [result-mr1] group by product_id`
- Map-Reduce **allows us sequencing multiple map-reduce jobs** in rive program. Output of one MR Job becomes input to another MR Job.
- We already have seen how to perform aggregation, let us try understanding some simple ways to perform join.

Computing Join using Map-Reduce

- JOIN of Users and Transactions: Input and Output of the operations is depicted here for some sample data!

Users	
UID	State
U1	UT
U2	GA
U3	CA
U4	CA
U5	GA

Transactions	
UID	PID
U1	P3
U2	P1
U1	P1
U2	P2
U4	P4
U1	P1
U1	P4
U5	P4

JOIN

transaction_users			
t.UID	PID	u.UID	State
U1	P3	U1	UT
U2	P1	U2	GA
U1	P1	U1	UT
U2	P2	U2	GA
U4	P4	U4	CA
U1	P1	U1	UT
U1	P4	U1	UT
U5	P4	U5	GA

Join Strategy – "Reducer Side Join"

- Let us look into a common strategy to perform joins on map-reduce, called as "Reducer Side Join" and "Repartition Join"
- This strategy motivates from "partitioned sort-merge join" used in the parallel RDBMS literature [4].
- In this strategy, **we have two mappers – one for each input file**.
- Each mappers produces "Key, Value"; where key is joining attribute (in case of both mappers), and values part contains projected attributes from respected file that are to be included in join result.
- Outputs of both the mappers are combined and shuffled to reducers.
- Reduces then actually performs the join as following.

Mappers in “Reducer side Join”

- Two mappers. One for each file.
- Output Key for Mappers is Joining Attribute!
- Output from both mappers are **combined** and shuffled to reducers
- Note that output values from mapper are tagged with “L” and “R” indicating source (Left or Right file respectively)

Users	
UID	State
U1	UT
U2	GA
U3	CA
U4	CA
U5	GA

Mapper User

Transactions	
UID	PID
U1	P3
U2	P1
U1	P1
U2	P2
U4	P4
U1	P1
U1	P4
U5	P4

Mapper Transaction

Key	Value
U1	L,UT
U2	L,GA
U3	L,CA
U4	L,CA
U5	L,GA
U1	R,P3
U2	R,P1
U1	R,P1
U2	R,P2
U4	R,P4
U1	R,P1
U1	R,P4
U5	R,P4

Mappers in “Reducer side Join”

```
public class UserMapper {
    public void map(rowid, row) {
        tokens = row.split(",");
        //tokens[0] = user_id, tokens[1] = state_id
        output_key = tokens[0]; //user_id
        output_value= "L," + tokens[1]; //state_id
        write(output_key, output_value);
    }
}

public class TransactionMapper {
    public void map(rowid, row) {
        tokens = row.split(",");
        //tokens[0] = user_id, tokens[1] = product_id
        output_key = tokens[0]; //user_id
        output_value= "R," + tokens[1]; //product_id
        write(output_key, output_value);
    }
}
```

Mappers in “Reducer side Join”

JOIN at Reducers

- Here is how output of mappers are shuffled (based on user id)
- Reducer, now does actual Join
- A brute force approach is “Nested-Loop Join”

Key	Value
U1	L,UT
U2	L,GA
U3	L,CA
U4	L,CA
U5	L,GA
U1	R,P3
U2	R,P1
U1	R,P1
U2	R,P2
U4	R,P4
U1	R,P1
U1	R,P4
U5	R,P4

U1: (R,P3), (R,P1), (L,UT), (R,P1),(R,P4)

U1	UT	U1	P3
U1	UT	U1	P1
U1	UT	U1	P1
U1	UT	U1	P4

U2: (L,GA), (R,P1), (R,P2)

U2	GA	U2	P1
U2	GA	U2	P2

U3: (L,CA)

U3	CA		

U4: (L,CA), (R,P4)

U4	CA	U4	P4

U5: (R,P4), (L,GA)

U5	GA	U5	P4

Joining Reducer pseudo code

```
public class JoinReducer {
    public void reduce(key, values) {
        userID = key;
        //iterate through, separate tuples from two mappers
        //assumption: there is single tuple from User Mapper
        state = "unfound";
        List prod_ids = new ArrayList();
        for(val : values) {
            tokens = val.split(",");
            tag = tokens[0];
            if tag == "L"
                state = tokens[1];
            else
                prod_ids.add(tokens[1]);
        }
        //perform join
        for(prod_id : prod_ids) {
            write(state, prod_id);
        }
        //note: it happens to be a RIGHT JOIN
    }
```

"Nested-Loop Join" –
a brute force approach

Joined output at Reducer

U1: (R,P3), (R,P1), (L,UT), (R,P1),(R,P4)

U1	UT	U1	P3
U1	UT	U1	P1
U1	UT	U1	P1
U1	UT	U1	P4

U2: (L,GA), (R,P1), (R,P2)

U2	GA	U2	P1
U2	GA	U2	P2

U3: (L,CA)

U3	CA		

U4: (L,CA), (R,P4)

U4	CA	U4	P4

U5: (R,P4), (L,GA)

U5	GA	U5	P4

State	PID
UT	P3
UT	P1
UT	P4
UT	P1
CA	P4
GA	P1
GA	P2
GA	P4

You may see result sorted on state as it is the key in reducer output

Improve upon “Join Algorithm”

- As we have seen, shown approach has two shortcoming
  – Assumes that single row generated from “Left” file. Which may be correct though in most cases (recall one of relation has single tuple in joins on relations – FK-PK pairs)
  – If list is huge, it may not fit in primary memory of reducer node.
- Article [4] call this as “Standard Repartition Join”; the article also suggest many ways to improve upon this.
- Here, I am sharing an approach that has found to be used in an implementation in text [3]

Improve upon "Join Algorithm" [3]

- Strategy goes as following
  - Have a customized "mapper-key" such that we take advantage of sorting (as part of shuffling)
  - Key generated by mappers should be such that state tuple from users come first in the value list for a given user-id (Refer next slide)
  - Create a customized partitioning function – required for customized "mapper key"

Mappers with customized key [3]

Users	
UID	State
U1	UT
U2	GA
U3	CA
U4	CA
U5	GA

Mapper User

Transactions	
UID	PID
U1	P3
U2	P1
U1	P1
U2	P2
U4	P4
U1	P1
U1	P4
U5	P4

Mapper Transaction

Key	Value
U1,1	L,UT
U2,1	L,GA
U3,1	L,CA
U4,1	L,CA
U5,1	L,GA
U1,2	R,P3
U2,2	R,P1
U1,2	R,P1
U2,2	R,P2
U4,2	R,P4
U1,2	R,P1
U1,2	R,P4
U5,2	R,P4

Mappers with customized key [3]

- **See here how values are arranged!**
- Output of user mapper is ordered before all outputs of transaction mapper for a key!
- Now join algorithm can work on this assertion

Key	Value
U1,1	L,UT
U2,1	L,GA
U3,1	L,CA
U4,1	L,CA
U5,1	L,GA
U1,2	R,P3
U2,2	R,P1
U1,2	R,P1
U2,2	R,P2
U4,2	R,P4
U1,2	R,P1
U1,2	R,P4
U5,2	R,P4

U1: (L,UT), (R,P3), (R,P1),(R,P1),(R,P4)

U1	UT	U1	P3
U1	UT	U1	P1
U1	UT	U1	P1
U1	UT	U1	P4

U2: (L,GA), (R,P1), (R,P2)

U2	GA	U2	P1
U2	GA	U2	P2

U3: (L,CA)

U3	CA		

U4: (L,CA), (R,P4)

U4	CA	U4	P4

U5: (L,GA), (R,P4)

U5	GA	U5	P4

Modified “Reduce Algorithm”

```
public class JoinReducer {

    public void reduce(key, values) {
        iterator = values.iterator();
        stateID = "unfound";
        if (iterator.hasNext()) {
            //firstPair must be state pair
            firstPair = iterator.next();
            if (firstPair.getLeftElement().equals("L")) {
                stateID = firstPair.getRightElement();
            }
        }
        while (iterator.hasNext()) {
            //the remaining elements must be product pair
            productPair = iterator.next();
            productID = productPair.getRightElement();
            write(stateID, productID);
        }
    }
```

Improve upon "Join Algorithm" [3]

- Complete implementation from book [3] is available at: https://github.com/mahmoudparsian/data-algorithms-book/tree/master/src/main/java/org/dataalgorithms/chap04/mapreduce

- Note this implementation has following things to be noted

  (1) Defines a Custom Key (Class for Key) with implementing "Comparator interface" [in Java terminology]

  In C++, it can stated that we require defining a class that overloads <, >, and == operators

  (2) Defining "Customized Partition" (shuffling) Function

  (3) Attaching multiple mappers in a Map Reduce job.

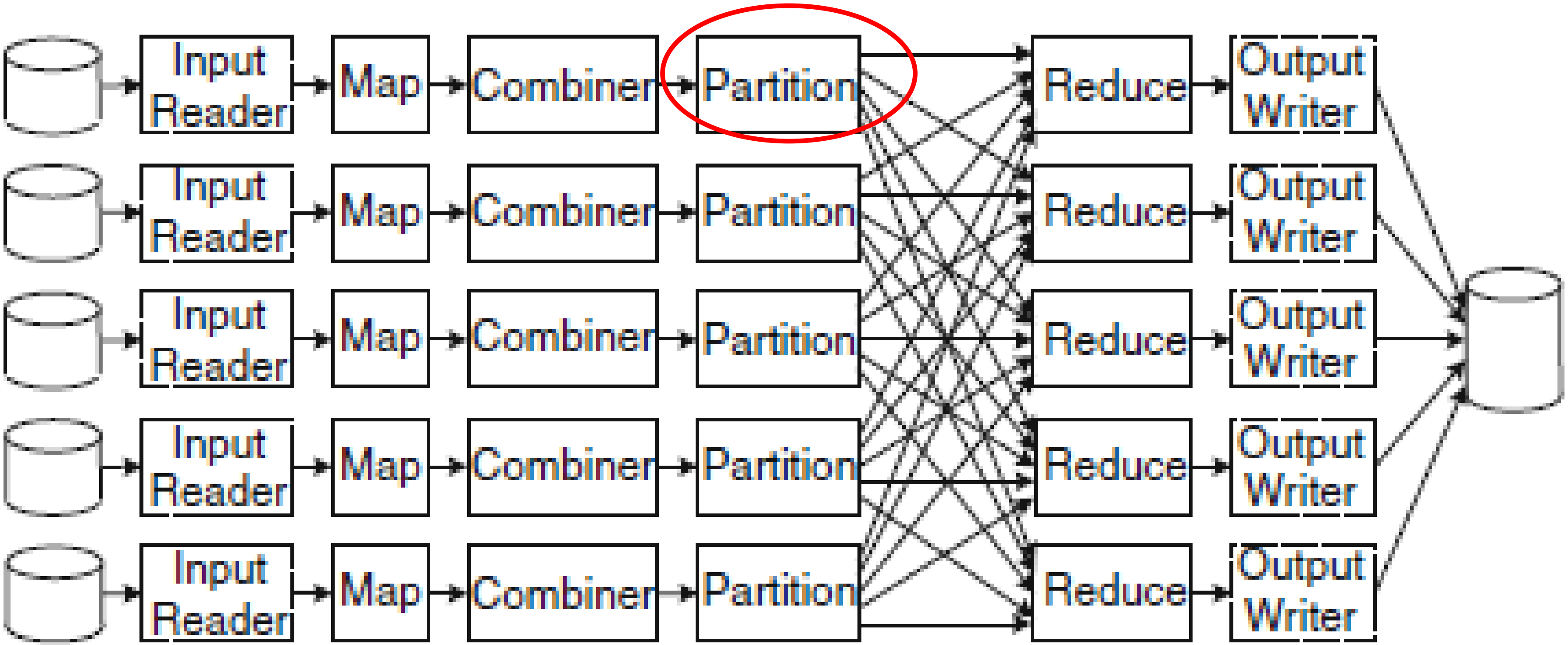
User defined Comparator for Grouping

- For user defined keys, we always require overloading comparison operations.
- It is done as following in Java

```java
public class SecondarySortGroupComparator
    implements RawComparator<PairOfStrings> {
    @Override
    public int compare(PairOfStrings first, PairOfStrings second) {
        return first.getLeftElement().compareTo(second.getLeftElement());
    }
```

Customize Partition Function

- In some case we need to customize the partition function; Typically when Partitioning based on key is not enough; or Key is composite and user defined



(MR Job-1) "Customized Partition"

- In our solution, we have added additional information (1, and 2) in our map out key for ordering tuples from two mappers.
- However we do not want using this added information for shuffling (partitioning) purpose. Therefore we create a partitioning function that hashes based on actual key only (i.e. user_id)

```java
public class SecondarySortPartitioner
    extends Partitioner<PairOfStrings, Object> {
    @Override
    public int getPartition(PairOfStrings key,
                            Object value,
                            int numberOfPartitions) {
        return (key.getLeftElement().hashCode()
                & Integer.MAX_VALUE) % numberOfPartitions;
    }
```

(MR Job-1) Join – Driver

```java
public static void main( String[] args ) throws Exception {
    Path transactions = new Path(args[0]);  // input
    Path users = new Path(args[1]);         // input
    Path output = new Path(args[2]);        // output

    Configuration conf = new Configuration();
    Job job = new Job(conf);
    job.setJarByClass(LeftJoinDriver.class);
    job.setJobName("Phase-1: Left Outer Join");

    // "secondary sort" is ha
    // 1. how the mapper generated keys will be partitioned
    job.setPartitionerClass(SecondarySortPartitioner.class);

    // 2. how the natural keys (generated by mappers) will be grouped
    job.setGroupingComparatorClass(SecondarySortGroupComparator.class);

    // 3. how PairOfStrings will be sorted
    job.setSortComparatorClass(PairOfStrings.Comparator.class);
```

Partitioning and Comparator getting specified

(MR Job-1) Join – Driver

```
job.setReducerClass(LeftJoinReducer.class);

job.setOutputKeyClass(Text.class);
job.setOutputValueClass(Text.class);
job.setOutputFormatClass(Sequenc
```

Multiple Mappers are getting Added!

```
// define multiple mappers: one for users and one for transactions
MultipleInputs.addInputPath(job, transactions,
        TextInputFormat.class, LeftJoinTransactionMapper.class);
MultipleInputs.addInputPath(job, users,
        TextInputFormat.class, LeftJoinUserMapper.class);
job.setMapOutputKeyClass(PairOfStrings.class);
job.setMapOutputValueClass(PairOfStrings.class);
FileOutputFormat.setOutputPath(job, output);

if (job.waitForCompletion(true)) {
    return;
}
else {
    throw new Exception("Phase-1: Left Outer Join Job Failed");
```

MR JOIN – Map Side

- Possible only if one of “file” can fit in the memory of mapper
- Suppose we want to join R and S on r.a=s.b
- Basic flow in MAP function goes as following-
  - R is loaded in memory once (let R be small enough to be loaded in in memory) and made available all instances of Map
  - For each record in S, perform lookup of s.a in R
  - If found, join and emit
- This is basically a “hash join” in database terminology (while other one if “sort-merge” join).

“Secondary Sort” using MR [3]

- Problem (weather data from book “data algorithms” [3])

```
Format:
        <year><,><month><,><day><,><temperature>

Example:
        2012, 01, 01, 35
        2011, 12, 23, -4
```

Input

```
Format:
        <year><-><month>: <temperature1><,><temperature2><,> .
        where temperature1 <= temperature2 <= ...

Example:
        2012-01:  5, 10, 35, 45, ...
        2001-11: 40, 46, 47, 48, ...
        2005-08: 38, 50, 52, 70, ...
```

Output

“Secondary Sort”

- Map-Reduce framework performs sort on “key”, and guarantees keys are sorted, but
- Does not guarantee any order for values within a key.
- So we play a trick here, as following
  - We add temperature to the key
  - Specify “partition function” for customized “shuffling”, so that it uses “year-month” only for grouping purpose.
  - We also need to define a comparator class, that gets used for comparing two “key objects” for grouping purpose!

Sort – Map Reduce functions

```
void map(key, value) {
    tokens = line.split(",");
    // YYYY, MM, DD, temperature
    yearMonth = tokens[0] + "-" + tokens[1];
    temperature = toInteger(tokens[3]);
    reducer_key = (yearMonth, temperature);
    write(reducer_key, temperature);
}
```

```
void reduce(key, values) {
    tmp_str = new String();
    for (value : values) {
        tmp_str += value + ",";
    }
    write(key.getYearMonth(), tmp_str);
}
```

Implementation of Sort

- Require creating Reducer (Map output) Key class, that
  - Wraps YYYY, MM, Temperature
  - Implements Comparator interface for grouping
- Require defining a Partition class that defines partition based on YYYY-MM only while key contains temperature also!
- Complete source code from the book can be accessed from https://github.com/mahmoudparsian/data-algorithms-book/tree/master/src/main/java/org/dataalgorithms/chap01/mapreduce

Exercise #: Top N

- Suppose you have following data file:
  CustNo, OrderAmountSum and want to compute top N X's while in descending order of count

- For example:

```
SELECT CustNo, OrderAmountSum FROM OrderSums
ORDER BY OrderAmountSum DESC LIMIT 10;
```

- At Mapper
  - We maintain a “Sorted Tree of Size N”
  - …
  - Map end producing ToP N for all of its data!

Top N: MR Strategy

- Compute top N at each mapper, and
- Shuffle output of all mappers to a “Single Reducer”
- To do this all map outputs may have same key.
- Single reducer should be fine here as normally N is smaller; say 10; for 1000 mappers, total data records for a reducer is 10000; not very large!
- Data records of top N from records from all mappers are aggregated, and final top N are computed!

Top N: MR Strategy

Multiple Mappers are getting Added!

Data Split 1

CID	AmtSum
C1	3500
C2	19000
C3	31000
C4	5000

Data Split 2

CID	AmtSum
C5	5500
C6	40000
C7	2300
C8	21000
C9	7600

Final Top-3 at Mapper 1

Key	Value
19000	C2,19000
31000	C3,31000
5000	C4,5000

Final Top-3 at Mapper 2

Key	Value
40000	C6,40000
21000	C8,21000
7600	C9,7600

Output Mapper 1

Key	Value
NULL	C2,19000
NULL	C3,31000
NULL	C4,5000

Output Mapper 2

Key	Value
NULL	C6,40000
NULL	C8,21000
NULL	C9,7600

Aggregated Top-3 at Reducer

Key	Value
31000	C3,31000
40000	C6,40000
21000	C8,21000

Output Reducer

Key	Value
Null	C3,31000
Null	C6,40000
Null	C8,21000

Top N: Map function

- Have a “Sorted Tree” object, globally available to **all map calls on a mapper** (mapper object)
  - Key of tree is “data” that to be sored – order amount in this case.
  - Rest of record is stored as value
- Initialized to empty on construction of Mapper class
  - Simply put data record into a tree
  - Smallest is removed if size > N
- This keeps on going till all records of a mapper are done
- At the end; values is tree will be top N of local within the mapper. This is outputted!

Top N Mapper

```
public class TopNMapper {
    private int N = 10; // default
    private SortedMap topN = new TreeMap();

    public void map(rowid, row) {
        tokens = row.split(",");
        data = tokens[1];
        topN.put(data, row);
        if (topN.size() > N) {
            topN.remove(topN.firstKey());
        }
    }

    protected void setup() {
        this.N = getNfromConfig();// default is top 10
    }

    protected void cleanup(){
        for (String str : top.values()) {
            write(NULL, str);
        }
    }
}
```

CRUX of CODE!
Sorted Map of size N is maintained

Top N: Reduce function

- Runs on a single reducers!
- Almost same as Map function
  - a “Sorted Tree” object, globally available to all reduce calls on the reducer
- Final Top N is built as following-
  - Simply put data record into a tree
  - Smallest is removed if size > N
- This keeps on going till all records of a mapper are done
- At the end; values is tree will be top N; this is outputted!

Top N Reducer

```
public class TopNReducer {

    private int N = 10; // default
    private SortedMap topN = new TreeMap();

    public void reduce(key, values) {
        for (value : values) {
            tokens = value.split(",");
            data =  tokens[1];
            topN.put(data, value);
            // keep only top N
            if (topN.size() > N) {
                topN.remove(topN.firstKey());
            }
        }
        List keys = new ArrayList(top.keySet());
        for(int i=keys.size()-1; i>=0; i--){
            write(NULL, topN.get(keys.get(i)));
        }
    }

    protected void setup() {
        this.N = getNfromConfig();// default is top 10
    }
}
```

Exercise #: Top N

- Complete source from book [3] is available at https://github.com/mahmoudparsian/data-algorithms-book/tree/master/src/main/java/org/dataalgorithms/chap03/mapreduce

- **“Apache Spark” and “Spark-SQL”**

Sources/References

[1] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: Simplified data processing on large clusters." (2004)

[2] Doulkeridis, Christos, and Kjetil NØrvåg. "A survey of large-scale analytical query processing in MapReduce." *The VLDB Journal—The International Journal on Very Large Data Bases* 23.3 (2014): 355-380.

[3] Parsian, Mahmoud. Data algorithms: recipes for scaling up with Hadoop and Spark, O'Reilly, 2015

[4] Blanas, Spyros, et al. "A comparison of join algorithms for log processing in mapreduce." *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 2010.