Lesson 5

Order Inversion and Secondary Sorting

Simplification by Expanding the Context

WHOLENESS OF THE LESSON

In mapreduce algorithms, quite often it is necessary to compute an aggregate value before computing individual values that contribute to the aggregate. Further, quite often we need to sort the values corresponding to a key. It is possible to accomplish both these tasks effortlessly.

Finer levels of intelligence are more expanded but at the same time more discriminating. For this reason, action that arises from a higher level of consciousness spontaneously computes the best path for success and fulfillment.

Computing Relative Frequencies

The pair and stripe algorithms computes the absolute frequencies.

Quite often relative frequencies may provide much deeper insight than the absolute frequencies.

Stripe approach for relative frequency matrix

Computing relative frequencies Is quite easy.

We just need to compute the “total” of all the values in the stripe and then divide each value of the stripe by the total.

Relative frequencies (Stripe)

```
class Mapper
    method Map(docid a; doc d)
        for all term u in record r do
            H = new AssociativeArray
            for all term v in Window(u) do
                H{v} = H{v} + 1 .          //Tally words co-occurring with u
            Emit(u, H)
class Reducer
    method Reduce(u, [H1, H2, H3, …])
        $H_f$ = new AssociativeArray
        for all stripe H in [H1;H2;H3; …] do
            $H_f = H_f + H$                //Element-wise addition
        s = Sum($H_f$)                     // Sum of all elements in $H_f$
        Emit(u, $H_f$ /s)                  //  Divide each element by s
```

Relative frequencies (Pair)

Problem to be solved:

We have at any time only one cell value. We do not have the sum of all values in the i-th row when we process any individual cell in the i-th row. So we cannot compute the relative frequency.

Solution: **Compute sum of all values in the i-th row, before we process any individual cell in the i-th row.**

Relative frequencies (Pair)

Modification needed in the Mapper class:

Every time a key-value pair of the form ((u, v), 1) is emitted, emit a key-value pair ((u, *), 1).

```
class Mapper
    method Map(docid a; doc d)
        for all term u in record r do
            for all term v in Window(u) do
                Emit((u, v), 1)
                Emit((u, *), 1)
```

Relative frequencies (Pair)

Ordering Rules.

1. The key (u, v) must be ordered such that the primary order is based on u and the secondary order is based on v.

2. (u, *) < (u, v) for all v. In other words, * < v.

Relative frequencies (Pair)

Note 1. * stands for a “special token” such that it is not part of “actual data” and * has value smaller than all possible values for the “events”.

Example 1.

If we coded our “events” as positive integers, we can choose 0 as the * or “the special token”.

Example 2.

If we coded our “events” as a string of alphanumeric characters, we can choose the string “*” as the * or “the special token”.

Relative frequencies (Pair)

Partition Rule.

The partition must just depend upon the u of the key (u, v).

```
int getPartition(key k, int numReducers)
    return Math.abs(k.left.hashCode())% numReducers
```

Relative frequencies (Pair)

```
int compareTo(Object O1, Object O2)
    k = compareTo(O1.left, O2.left)
    if (k != 0) return k
    return compareTo(O1.right, O2.right)
```

Relative Frequencies

Example

(apple, *)	[10, 15, 5, 10]	sum = 40
(apple, grape)	[2, 6, 2]	f(apple,grape) = 10/40
(apple, peach)	[3, 1]	f(apple, peach) = 4/40
…		
(apple, sberry)	[1,2,2,1]	f(apple, sberry) = 6/40
(beets, *)	[30, 10, 60]	sum = 100

(Note: f stands for relative frequency.)

In class quiz

Q1. Is this in-mapper combining or not?

Q2. What is wrong?

(apple, *)	[10, 15, 5, 10]	sum = 40
(apple, grape)	[2, 6, 2]	f(apple, grape) = 10/40
(apple, peach)	[3, 1]	f(apple, peach) = 4/40
…		
(apple, sberry)	[1,2,1, 1, 1]	f(apple, sberry) = 6/40
(beets, *)	[30, 10, 60]	sum = 100

Relative Frequencies (Pair)

```
class Mapper
    method Map(docid a, doc d)
        for all term u in record r do
            for all term v in Window(u) do
                Emit((u, v), 1)
                Emit((u, *), 1)
```

Relative Frequencies (Pair)

```
class Reducer
    method initialize()
        sum = 0
    method Reduce(Pair (u, v), Integer [c1, c2, …])
        s = 0
        for all count c in counts [c1, c2, …] do
            s = s + c
        if (v == “*”)
            sum = s
        else
            Emit((u, v), s / sum)
```

Relative Frequencies (Pair)

```
int getPartition(key k, int numReducers)
    return Math.abs(k.left.hashCode())% numReducers
```

```
int compareTo(Object O1, Object O2)
    k = compareTo(O1.left, O2.left)
    if (k != 0) return k
    return compareTo(O1.right, O2.right)
```

Main Point 1

Order inversion technique is in some sense unique to mapreduce algorithms. It allows us to compute any aggregate function before computing the individual parts. Thus, we can at the very least avoid one additional mapreduce task.

Through the regular practice of the TM technique the functioning of the mind and body are brought together to operate in accord with all the laws of Nature.

Order Inversion

In the pair approach to compute the relative frequencies, we were able to compute the sum, an “aggregate value”, before computing the values of the “parts” or the individual items. This design pattern is known as order inversion.

Sorting Values

In Hadoop, intermediate key-value pairs are sorted by the key during the shuffle and sort phase. However there is no built-in mechanism for sorting values.

Google's implementation has built-in functionality for sorting values.

Sorting Values

In Hadoop, intermediate key-value pairs are sorted by the key during the shuffle and sort phase. So if we want any part of the value to be sorted, all we need to do is make it part of the key.

After the shuffle and sort phase, we can regroup the key and value whichever way you prefer.

Sorting Values

Consider the example of weather sensor data from different sensors s1, s2, s3, …collected at different times t1, t2, t3, and so on. The actual data is stored in a record and sent to local weather center.

Sorting Values

Thus at time t1, weather center will receive all the data for t1 in some order.

(t1, s1, d11)

(t1, s2, d12)

(t1, s3, d13)

…

At time t2, weather center will receive all the data for t2 in some order.

(t2, s1, d11)

(t2, s2, d12)

(t2, s3, d13)

Sorting Values

The weather center would like to group the data by the senor and then would like to sort the data within each group by time.

Sensor 1 data:

(t1, s1, d11)

(t2, s1, d21)

(t3, s1, d31)

. . .

Sensor 2 data:

(t1, s2, d11)

(t2, s2, d21)

(t3, s2, d31)

. . .

Rule for determining the key

Anything that needs to be sorted or grouped must be part of the key. **Sorting is accomplished by the user-defined comparator** and **grouping is accomplished by overloading getPartition method**.

```
class Mapper
    method map(docid a, doc d)
        t = getTime()
        s = getSensorID()
        d = getData()
        Emit((s, t), d)
```

Sorting Values

```
class Reducer
    method reduce((s, t), [d])
        Emit((s, t), d)
```

Sorting Values

```
int getPartition(key k, int numReducers)
    return Math.abs(k.left.hashCode())% numReducers
```

```
int compareTo(Object O1, Object O2)
    k = compareTo(O1.left, O2.left)
    if (k != 0) return k
    return compareTo(O1.right, O2.right)
```

Sorting Values Quiz

Consider a key with three “fields”.

Class O {private X x; private Y y; private Z z}

Q1. Write a comparator so that objects of the class O are sorted by x in the descending order, followed by y in the ascending order and then z in the descending order.

Answer to the Quiz

```
int compareTo(Object o1, Object o2)
    k = compareTo(o1.x, o2.x)
    if (k != 0) return  -k        // desc
    k = compareTo(o1.y, o2.y)
    if (k != 0) return k          // asc
    return  -compareTo(o1.z, o2.z)     // desc
```

Example

	Fname	Lname	Score
1	Jose	Rock	80
2	Abel	Cox	80
3	Jose	Rock	50
4	Bea	Cox	100
5	Bea	Rock	70
6	Jose	Cox	40
Order above 6 records: Primary Fname Asc, Secondary Lname Desc, Ternary Score Asc			
Answer			
2	Abel	Cox	80
5	Bea	Rock	70
4	Bea	Cox	100
1	Jose	Rock	50
3	Jose	Rock	80
6	Jose	Cox	40

SUMMARY

Ultimately, controlling synchronization in the MapReduce programming model boils down to effective use of the following techniques:

1. Constructing complex keys and values that bring together data necessary for a computation. This is used in all of the above design patterns.
2. Executing user-specified initialization and termination code in either the Mapper and/or Reducer. For example, in-mapper combining depends on emission of intermediate key-value pairs in the map task termination code.

(Optional)
"Hadoop-The definitive guide by Tom White"

Each map task has a circular memory buffer that it writes the output to. The buffer is 100 MB by default (the size can be tuned by changing the mapreduce.task.io.sort.mb property). When the contents of the buffer reach a certain threshold size (mapreduce.map.sort.spill.percent, which has the default value 0.80, or 80%), a background thread will start to spill the contents to disk.

Map outputs will continue to be written to the buffer while the spill takes place, but if the buffer fills up during this time, the map will block until the spill is complete. Spills are written in round-robin fashion to the directories specified by the mapreduce.cluster.local.dir property, in a job-specific subdirectory. Before it writes to disk, the thread first divides the data into partitions corresponding to the reducers that they will ultimately be sent to. Within each partition, the background thread performs an in-memory sort by key, and if there is a combiner function, it is run on the output of the sort.

Running the combiner function makes for a more compact map output, so there is less data to write to local disk and to transfer to the reducer.“

So if this sorted result is given as input to the combiner then the combiner output must be sorted. For example in the W1D4 solution , cell (**Combiner 1 – Input Split 2)**

<rat, 1>, <cat, 2>, <bat, 1>

must read

<bat, 1>, <cat, 2>, <rat, 1>

and so on.

Main Point 2

In the Hadoop implementation, values are not sorted. However, it is not a limitation. Using the technique of value to key conversion, it is possible to sort values as well. *Purification of the path is a natural part of the evolutionary process and is due to the invincible nature of creative intelligence.*

CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

1. It is important that necessary values are computed in one mapreduce job if possible.
2. In order to be handle various real world problems, mapreduce supports efficient design patterns.

---

3. ***Transcendental consciousness:*** *is the field of pure orderliness. Even a chaotic mind is capable of diving into this field and benefit immediately from the orderly influence that comes from contact with this field. Hidden problems and pockets of disorder are spontaneously neutralized through this process.*
4. ***Impulses within the Transcendental Field:*** *The repeated collapse of infinity to a point and expansion of point to infinity with infinite frequency within the transcendental field produces the "hum" of creation, the integrated and even flow of all the forces underlying the manifest universe.*
5. ***Wholeness moving within itself:*** *In Unity Consciousness, the unfoldment of creation from within the unmanifest is appreciated as the expression of one's own unbounded Self.*