

Lesson 3

Local Aggregation

Capture the fort

Wholeness of the Lesson

The communication cost is the single most factor that determines the efficiency of a MapReduce job. The best way to achieve better efficiency is by carefully planning and organizing the data so that only the least amount of data is shuffled across the network.

“Capturing the fort” is the very first course of action.

LOCAL AGGREGATION

An improvement on the basic wordcount algorithm is shown next. Only mapper is modified in this example.

An associative array is introduced inside the mapper to tally up term counts within a single input-split. Instead of emitting a key-value pair for each term in the input-split, this version emits a key-value pair for each **unique** term in the input-split.

Given the fact that some words appear frequently within an input-split, this can yield substantial savings in the number of intermediate key-value pairs emitted.

Example

Assume that an input-split contains the word “science” 10 times.

The wordcount algorithm will **emit** the pair (“science”, 1) ten times.

The “modified” algorithm shown next will **emit** only one pair: (“science”, 10).

LOCAL AGGREGATION

```
class Mapper
```

```
  method initialize()
```

```
    H = new AssociativeArray()
```

```
  method map(docid a; doc d)
```

```
    for all term t in record r do
```

```
       $H\{t\} = H\{t\} + 1.$ 
```

```
  method close()
```

```
    for all term t in H do
```

```
      Emit(t, H{t})
```

LOCAL AGGREGATION

Note that there is a reduction in the number of intermediate key-value pairs that need to be shuffled across the network from the **number of terms in the input-split** to the **number of unique terms in the input-split**

Two important design principles to follow:

- 1. One key, One emit principle.** A Mapper object will emit a key-value pair with a specific “key” only once.
- 2. Order 1 space complexity principle.** The size of the “value” is independent of the input-split.

In-mapper combining

With this technique, we are in essence incorporating combiner functionality directly inside the mapper. There is no need to run a separate combiner, since all opportunities for local aggregation are already exploited. This is a sufficiently common design pattern in MapReduce called, “**in-mapper combining**”. There are two main advantages to using this design pattern.

Advantages of the in-mapper combining pattern

First, it provides control over when local aggregation occurs and how it exactly takes place.

In contrast, the semantics of the combiner is under specified in MapReduce. For example, Hadoop makes no guarantees on how many times the combiner is applied, or that it is even applied at all. **The combiner is provided as a semantics-preserving optimization to the execution framework, which has the option of using it, perhaps multiple times, or not at all.**

Advantages of the in-mapper combining pattern

Second, in-mapper combining will typically be more efficient than using actual combiners.

One reason is the additional overhead associated with actually materializing the key-value pairs. Combiners reduce the amount of intermediate data that is shuffled across the network, but don't actually reduce the number of key-value pairs that are emitted by the mappers in the first place.

Advantages of the in-mapper combining pattern

In-mapper combining eliminates unnecessary object creation and destruction (garbage collection takes time), and furthermore, object serialization and deserialization (when intermediate key-value pairs fill the in-memory buffer holding map outputs and need to be temporarily spilled to disk).

Drawbacks to the in-mapper combining pattern

First, it breaks the functional programming underpinnings of MapReduce, since state is being preserved across multiple input key-value pairs (records). Preserving state across multiple input instances means that algorithmic behavior may depend on the order in which input key-value pairs are encountered. This creates the potential for ordering-dependent bugs, which are difficult to debug on large datasets in the general case (although the correctness of in-mapper combining for word count is easy to demonstrate).

Drawbacks to the in-mapper combining pattern

Second, there is a fundamental scalability bottleneck associated with the in-mapper combining pattern. It critically depends on having sufficient memory to store intermediate results until the mapper has completely processed all key-value pairs in an input split.

Drawbacks to the in-mapper combining pattern

One common solution to limiting memory usage when using the in-mapper combining technique is to “block” input key-value pairs and “flush” in-memory data structures periodically. Instead of emitting intermediate data only after every key-value pair has been processed, emit partial results after processing every n key-value pairs. This is straightforwardly implemented with a counter variable that keeps track of the number of input key-value pairs that have been processed.

Drawbacks to the in-mapper combining pattern

As an alternative, the mapper could keep track of its own memory footprint and flush intermediate key-value pairs once memory usage has crossed a certain threshold. In both approaches, either the block size or the memory usage threshold needs to be determined empirically: with too large a value, the mapper may run out of memory, but with too small a value, opportunities for local aggregation may be lost.

Main Point 1

The local aggregation is accomplished through in-mapper combining technique. In our daily life, we first organize and combine various ideas before talking to others in a spontaneous way. *The most fundamental combiner is the unification of the Self with itself, which gives rise to knower, process of knowing, and known.*

Average Problem

Consider a simple example: we have a large dataset where input keys are strings and input values are integers, and we wish to compute the mean of all integers associated with the same key. A real-world example might be a large user log from a popular website, where keys represent user ids and values represent some measure of activity such as elapsed time for a particular session - the task would correspond to computing the mean session length on a per-user basis, which would be useful for understanding user demographics.

Class Activity

Write a MapReduce algorithm to solve the average problem

Assume that record contains (... , userID,..., time, ...)

There are access methods available for all “fields” in the record.

Thus you can assume the following:

```
u = r.getUserId()
```

```
t = r.getTime()
```

Average Problem

```
class Mapper
```

```
    method map(docid a, doc d)
```

```
        //Note: each record r has one user Id and
```

```
        // one time. So no loop is needed.
```

```
        u = r.getUserId()
```

```
        t = r.getTime()
```

```
        Emit(u, t)
```

Average Problem

```
class Reducer
```

```
  method Reduce(string u, integers [t1, t2, ...])
```

```
    sum = 0; cnt = 0;
```

```
    for all integer t in integers [t1, t2, ...] do
```

```
      sum = sum + t; cnt = cnt + 1;
```

```
    avg = sum / cnt
```

```
    Emit(u, avg)
```

ALGORITHMIC CORRECTNESS ISSUES

In class exercise.

Create the in-mapper combining algorithm to solve the average problem.

ALGORITHMIC CORRECTNESS ISSUES

If each mapper emits the average, the algorithm will produce the wrong result

(Mapper Input)

(u1, 1) (u1, 4)

(u1, 2) (u1, 5)

(u1, 3)

(Mapper output)

(u1, 2) (u1, 4.5)

(Reducer output)

(u1, (2+4.5)/2)

The correct answer is (u1, 3).

ALGORITHMIC CORRECTNESS ISSUES

(Mapper Input)

(u1, 1) (u1, 4)

(u1, 2) (u1, 5)

(u1, 3)

Mapper emits the pair (sum, count) as value.

(u1, (6, 3)) (u1, (9, 2))

Reducer emits

(u1, (6+9)/5)

That is (u1, 3). That is correct!

ALGORITHMIC CORRECTNESS ISSUES

In the case of wordcount, the operation was addition. It is both associative and commutative.

The “average” is not associative.

You can modify an algorithm to “in-mapper combining” version without any change in the “emit” statements if the operations are associative and commutative.

In-Mapper Combining version of the average problem

```
class Mapper
```

```
    method initialize()
```

```
        H = new AssociativeArray()
```



```
method map(docid a, doc d)
```

```
    //Note: each record r has one user Id and  
    // one time. So no loop is needed.
```

```
    u = r.getUserId()
```

```
    t = r.getTime()
```

```
    p = new Pair(t, 1)
```

```
    if (H{u} is null)
```

```
        H{u} = p
```

```
    else
```

```
        H{u} = H{u} + p //elementwise addition
```

method close()

for all term u in H do

Emit(u , $H\{u\}$)

Average Problem

```
class Reducer
```

```
  method Reduce(string u, pair [(t1, c1), (t2, c2) ...])
```

```
    sum = 0; cnt = 0;
```

```
    for all pair (t, c) in [(t1, c1), (t2, c2) ...] do
```

```
      sum = sum + t; cnt = cnt + c;
```

```
    avg = sum / cnt
```

```
    Emit(u, avg)
```

Main Point 2

Using the In-mapper combining technique, each mapper is guaranteed to produce only one key-value pair for each key by combining the values. Bringing together all the relevant ideas, it is easy to convey the central idea in a concise and precise manner . *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.*

CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

1. It is important that safeguards be taken to maintain the consistency and correctness.
 2. In order to be an accurate representation of the part of the real world that it is modeling, a framework must guarantee certain features.
-
3. ***Transcendental consciousness:*** *is the source of all perfection in life.*
 4. ***Impulses within the Transcendental Field :*** *It is these impulses that structure and are within, everything in the universe.*
 5. ***Impulses within the Transcendental Field :*** *In unity consciousness the individual functions in perfect harmony with all the laws of nature.*

