Hadoop MapReduce (part b)

ECE 454 / 751: Distributed Computing

Instructor: Dr. Wojciech Golab

wgolab@uwaterloo.ca

Slides are derived from online materials:
https://highlyscalable.wordpress.com/2012/02/01/mapreduce-patterns/

Learning objectives

MapReduce programming patterns in pseudo-code:

- counting and summing
- selection
- projection
- inverted index
- cross-correlation

Counting and summing

The simplest solution outputs a (key, 1) tuple for every instance of a key.

```
class Mapper
   method Map(docid id, doc d)
      for all term t in doc d do
         Emit(term t, count 1)

class Reducer
   method Reduce(term t, counts [c1, c2,...])
      sum = 0
      for all count c in [c1, c2,...] do
          sum = sum + c
      Emit(term t, count sum)
```

Counting and summing (cont.)

Instead of emitting one key-value pair for each word in the input, the mapper can aggregate the counts for each document. This requires additional memory for the map task.

```
class Mapper
   method Map(docid id, doc d)
      H = new AssociativeArray
      for all term t in doc d do
          H{t} = H{t} + 1
      for all term t in H do
         Emit(term t, count H{t})
```

Counting and summing (cont.)

Alternatively, a combiner can be used to aggregate counters across all the documents processed by a map task. The combiner is executed before the shuffle phase, and is usually identical to the reducer.

```
class Mapper
   method Map(docid id, doc d)
      for all term t in doc d do
         Emit(term t, count 1)

class Combiner
   method Combine(term t, counts [c1, c2,...])
      sum = 0
      for all count c in [c1, c2,...] do
          sum = sum + c
      Emit(term t, count sum)
```

Selection

Selection returns a subset of the input elements that satisfy some predicate (e.g., x < 10). Only a mapper is required, and it is possible to run the job without a reducer at all, in which case the framework will generate one output file per map task.

```
class Mapper
   method Map(rowkey key, tuple t)
      if t satisfies the predicate
         Emit(tuple t, null)
```

Projection

Projection returns a subset of the fields of each input element (e.g., from <x,y,z> to <x,y>). The reducer is needed only to eliminate duplicates.

```
class Mapper
   method Map(rowkey key, tuple t)
      tuple g = project(t)  // extract required fields to tuple g
      Emit(tuple g, null)

class Reducer
   method Reduce(tuple t, array n)   // n is an array of nulls
      Emit(tuple t, null)
```

Inverted index

Given a set of text documents with distinct identifiers, produce a mapping from term to document ID.

```
class Mapper
   method Map(docid id, doc d)
      for all term t in doc d do
         Emit(term t, docid id)

class Reducer
   method Reduce(term t, docids [id1, id2, ...])
      Emit(term t, list of docids)
```

Cross-correlation

Problem statement:

- There is a set of tuples of items (e.g., sentences of words). For each possible pair of items, calculate the number of tuples where these items co-occur. If the total number of items is N, then N^2 values should be reported.
- This problem appears in text analysis (say, items are words and tuples are sentences), market analysis (customers who buy *this* tend to also buy *that*). If N^2 is quite small and such a matrix can fit in the memory of a single machine, then the implementation is straightforward.
- Otherwise, it is possible to scale out the computation using MR.

Cross-correlation (cont.)

Pairs Approach: simplest and generally slowest (why?)

```
class Mapper
   method Map(void, items [i1, i2,...] )
      for all item i in [i1, i2,...]
         for all item j in [i1, i2,...] such that j > i
            Emit(pair [i j], count 1)

class Reducer
   method Reduce(pair [i j], counts [c1, c2,...])
      s = sum([c1, c2,...])
      Emit(pair[i j], count s)
```

Cross-correlation (cont.)

Stripes Approach: faster (why?) but more complex and requires more memory for map-side aggregation (i.e., computing H).

```
class Mapper
   method Map(void, items [i1, i2,...] )
      for all item i in [i1, i2,...]
         H = new AssociativeArray : item -> counter
         for all item j in [i1, i2,...] such that j > i
            H{j} = H{j} + 1
         Emit(item i, stripe H)

class Reducer
   method Reduce(item i, stripes [H1, H2,...])
      H = new AssociativeArray : item -> counter
      H = merge-sum( [H1, H2,...] )
      for all item j in H.keys()
         Emit(pair [i j], H{j})
```

Cross-correlation (cont.)

- Two examples are given shortly. For simplicity, the first example uses short records with two items per record. The second example uses longer records with three items per record.
- The examples assume that the mapper function in the previous two slides is optimized as follows:
  1. The mapper emits nothing if i = j because we are not interested in cross-correlations between an item and itself.
  2. The mapper eliminates duplicates by ensuring that i < j.
- Both examples assume that a combiner is not used. To prepare for the midterm and final exam, repeat this exercise on your own assuming that a combiner is used.

Example 1: pairs

R1 = “a dog”, R2 = “a cat”

Mapper:

Input: R1
R2

Output: <[a, dog], 1>
<[a, cat], 1>

Reducer:

Input: <[a, dog], [1]>
<[a, cat], [1]>

Output: <[a, dog], 1>
<[a, cat], 1>

Example 1: stripes

R1 = “a dog”, R2 = “a cat”

Mapper:

Input: R1
R2

Output: <[a], {dog:1}>
<[a], {cat:1}>

Reducer:

Input: <[a], [{dog:1}, {cat:1}]>
Output: <[a, dog], 1>
<[a, cat], 1>

Example 2: pairs

R1 = “a big dog”, R2 = “a small cat”

Mapper:

Input: R1
R2

Output: <[a, big], 1> <[a, dog], 1> <[big, dog], 1>
<[a, small], 1> <[a, cat], 1> <[cat, small], 1>

Reducer:

Input: <[a, big], [1]> <[a, dog], [1]> <[big, dog], [1]>
<[a, small], [1]> <[a, cat], [1]> <[cat, small], [1]>

Output: <[a, big], 1> <[a, cat], 1> <[a, dog], 1>
<[a, small], 1> <[big, dog], 1> <[cat, small], 1>

Example 2: stripes

R1 = “a big dog”, R2 = “a small cat”

Mapper:

Input: R1
R2

Output: <[a], {big:1, dog:1} > <[big], {dog:1}>
<[a], {small:1, cat:1} > <[cat], {small:1}>

Reducer:

Input: <[a], [{big:1, dog:1}, {small:1, cat:1}]>
<[big], [{dog:1}]> <[cat], [{small:1}]>

Output: <[a, big], 1> <[a, cat], 1> <[a, dog], 1>
<[a, small], 1> <[big, dog], 1> <[cat, small], 1>