MONOIDS
(DESIGN EFFICIENT MAP REDUCE ALGORITHMS)

Master Program in Computer Science
University of Calabria

Prof. F. Ricca

Monoids

- An algebraic structure
  - Associative binary operation
  - Identity element
- Design principle for Efficient Map/Reduce
  - Monoidify! [Lin 2013]
    - "Make the output of the mapper a monoid"
  - More flexibility from commutative monoids

…more on this later

Running example

```
SELECT key, value
FROM mytable;
```

key	value
key1	10
key1	20
key1	30
key2	40
key2	60
key3	20
key3	30

```
SELECT key, AVG(value)
FROM mytable GROUP BY key;
```

key	value
key1	20
key2	50
key3	25

A bad programming style (mapper)

```
/**
 * @param key is a string object
 * @param value is a long associated with key
 */
map(String key, Long value) {
    emit(key, value);
}
```

A bad programming style (reducer)

```java
/**
 * @param key is a string object
 * @param values is a list of longs: [i1, i2, ...]
 */
reduce(String key, List<Long> list) {
    Long sum = 0;
    Integer count = 0;
    for (Long i : list) {
        sum = sum + i;
        count++;
    }
    double average = sum/count;
    emit(key, average);
}
```

A bad programming style (comment)

- The algorithm is not very efficient
  - Too much work required by shuffle & sort of the framework!

- We cannot use the reducer as a combiner
  - The mean of means of is not the same as the mean

- We know already…
  - It is possible to modify this in a better solution!

Good programming (mapper)

```
/**
 * @param key is a string object
 * @param value is a Pair(long : sum, int: count) associated with key
 */
map(String key, Long value) {
    emit(key, Pair(value, 1));
}
```

- The key is the same as before
- The value is a pair of (sum, count)

- This output has a precise algebraic property!

Good programming (combiner)

```
/**
 * @param key is a string object
 * @param value is a list = [(v1, c1), (v2, c2), ...]
 */
combine(String key, List<Pair<Long, Integer>> list) {
    Long sum = 0;
    Integer count = 0;
    for (Pair<Long, Integer> pair : list) {
        sum += pair.v;
        count += pair.c
    }
    emit(key, new Pair(sum, count));
}
```

- Performs a "local reduce" on the output of the mapper!
- Shuffle & sort "a few" values

Good programming (reducer)

```
/**
* @param key a string object
* @param value is a list = [(v1, c1), (v2, c2), ...]
*/
reduce(String key, List<Pair<Long, Integer>> list) {
    Long sum = 0;
    Integer count = 0;
    for (Pair<Long, Integer> pair : list) {
        sum += pair.v;
        count += pair.c
    }
    Pair<Long, Integer>  partialPair = new Pair<Long, Integer>(sum, count);
    emit(key, partialPair);
}
```

What is the algebraic property?

- In the good example the output of the mapper is a $\boxed{\text{monoid}}$

- A *monoid* is a triple (S, f, e) satisfying
    - S is a set
    - f: S × S → S is a binary operation, say •
    - e ∊ S is the identity element
  - *Closure:*
    - for all a and b in S, the result of the operation a • b is also in S
  - *Associativity:*
    - for all a, b, and c in S, it holds (a • b) • c = a • (b • c)

questo la stiamo facendo nel combiner

  - *Identity element*:
    - for all a in S, the following two equations hold:
    - e • a = a   and a • e = a

Let's check intuitively

- Operation is memberwise sum
  - (a,b) + (c,d) = (a+c,b+d)
- Identity element: (0,0)
  - Es. (1,1) + (0,0) = (1,1)

ad sempio, la media
non va bene

Other examples

- Addition over set of integers
  - 1+0=0+1=1
  - a+(b+c) = (a+b)+c
  - a+b is a number
- Maximum over Set of Integers → Monoid
  - MAX(a, MAX(b,c)) = MAX(MAX(a,b),c)
  - MAX(a,0) = MAX(0,a) = a
  - MAX(a,b) is a number
- Subtraction over Set of Integers → NOT a Monoid
  - $(1-2)-3 \neq 1-(2-3)$
  - Not associative!

Commutative monoids

- A triple (S, f, e) is a *commutative* monoid if
  - Is a monoid and
  - Is *Commutative:*
    - for all a, b, and c in S, it holds (a • b) = (b • a)

- A triple (S, f, e) is a *idempotent* monoid if
  - Is a monoid and
  - Is *Idempotent:*
    - for all a in S, it holds (a • a) = a

- Observation:
  - Combiners can be used when the function you want to apply is a commutative monoid

Cosa succede se non facciamo sort?

Se non ricostruiamo l'ordine degli elementi prima di fare le operazioni di reduce, è un problema? La risposta è se l'operazione deve essere commutativa.

Commutatività   --> L'ordine non è importante.

Additional examples

- Concatenation over Lists → Monoid
  - L + [] = L
  - [] + L = L
  - (L1 + L2) + L3 = L1 + (L2 + L3)
  - Is it commutative?
- Union/Intersection over set of integers?
- Median over set of integers?

Why monoids?

- We can write very general code in terms of the algebraic construction, and then use it over all of the different operations
- Monoids can build "fold" operations
  - Operations that collapse a sequence of other operations into a single value
- Any data structure which is a monoid is a data structure with a meaningful fold operation:
  - Monoids encapsulate the requirements of foldability

Monoidify!

"One principle for designing efficient MapReduce algorithms can be precisely articulated as follows: create a monoid out of the intermediate value emitted by the mapper. Once we monoidify the object, proper use of combiners and the in-mapper combining techniques becomes straightforward."

Jimmy Lin

noh la domanda che ha fatto ricca

La struttura ad albero di elementi intermedi ecc se funziona bene con map reduce

Altra roba importante: calcolare la cosa dei monoidi ma con mapreduce (attenzione, monoide NON E' commutativo)