DDAM, 2019
MAPREDUCE PATTERNS

Patrizio Dazzi

A DESIGN PATTERN

A design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. A design pattern is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations. Patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

DISTINCT VALUES

Problem Statement: There is a set of records that contain fields F and G. Count the total number of unique values of field F for each subset of records that have the same G (grouped by G).

```
1   Record 1: F=1, G={a, b}
2   Record 2: F=2, G={a, d, e}
3   Record 3: F=1, G={b}
4   Record 4: F=3, G={a, b}
5
6   Result:
7   a -> 3   // F=1, F=2, F=3
8   b -> 2   // F=1, F=3
9   d -> 1   // F=2
10  e -> 1   // F=2
```

Applications:

- Log Analysis, Unique Users Counting

DISTINCT VALUES

The first approach is to solve the problem in two stages. At the first stage Mapper emits dummy counters for each pair of F and G; Reducer calculates a total number of occurrences for each such pair. The main goal of this phase is to guarantee uniqueness of F values. At the second phase pairs are grouped by G and the total number of items in each group is calculated.

```
1  Record 1: F=1, G={a, b}
2  Record 2: F=2, G={a, d, e}
3  Record 3: F=1, G={b}
4  Record 4: F=3, G={a, b}
5
6  Result:
7  a -> 3   // F=1, F=2, F=3
8  b -> 2   // F=1, F=3
9  d -> 1   // F=2
10 e -> 1   // F=2
```

Phase I:

```
1  class Mapper
2     method Map(null, record [value f, categories [g1, g2,...]])
3        for all category g in [g1, g2,...]
4           Emit(record [g, f], count 1)
5
6  class Reducer
7     method Reduce(record [g, f], counts [n1, n2, ...])
8        Emit(record [g, f], null )
```

Phase II:

```
1  class Mapper
2     method Map(record [f, g], null)
3        Emit(value g, count 1)
4
5  class Reducer
6     method Reduce(value g, counts [n1, n2,...])
7        Emit(value g, sum( [n1, n2,...] ) )
```

CROSS-CORRELATION

Problem Statement: There is a set of tuples of items. For each possible pair of items calculate a number of tuples where these items co-occur. If the total number of items is N then N*N 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*N is quite small and such a matrix can fit in the memory of a single machine, then implementation is straightforward.

Applications:

- Text Analysis, Market Analysis

CROSS-CORRELATION (SOLUTION 1)

The first approach is to emit all pairs and dummy counters from Mappers and sum these counters on Reducer. The shortcomings are:

- The benefit from combiners is limited, as it is likely that all pair are distinct
- There is no in-memory accumulations

```
1  class Mapper
2     method Map(null, items [i1, i2,...] )
3        for all item i in [i1, i2,...]
4           for all item j in [i1, i2,...]
5              Emit(pair [i j], count 1)
6
7  class Reducer
8     method Reduce(pair [i j], counts [c1, c2,...])
9        s = sum([c1, c2,...])
10       Emit(pair[i j], count s)
```

CROSS-CORRELATION (SOLUTION 2)

The second approach is to group data by the first item in pair and maintain an associative array ("stripe") where counters for all adjacent items are accumulated. Reducer receives all stripes for leading item i, merges them, and emits the same result as in the Pairs approach.

- Generates fewer intermediate keys. Hence the framework has less sorting to do.
- Greately benefits from combiners.
- Performs in-memory accumulation. This can lead to problems, if not properly implemented.
- More complex implementation.
- In general, "stripes" is faster than "pairs"

```
1  class Mapper
2     method Map(null, items [i1, i2,...] )
3        for all item i in [i1, i2,...]
4           H = new AssociativeArray : item -> counter
5           for all item j in [i1, i2,...]
6              H{j} = H{j} + 1
7           Emit(item i, stripe H)
8
9  class Reducer
10    method Reduce(item i, stripes [H1, H2,...])
11       H = new AssociativeArray : item -> counter
12       H = merge-sum( [H1, H2,...] )
13       for all item j in H.keys()
14          Emit(pair [i j], H{j})
```

ITERATIVE MESSAGE PASSING

Problem Statement: There is a network of entities and relationships between them. It is required to calculate a state of each entity on the basis of properties of the other entities in its neighborhood. This state can represent a distance to other nodes, indication that there is a neighbor with the certain properties, characteristic of neighborhood density and so on.

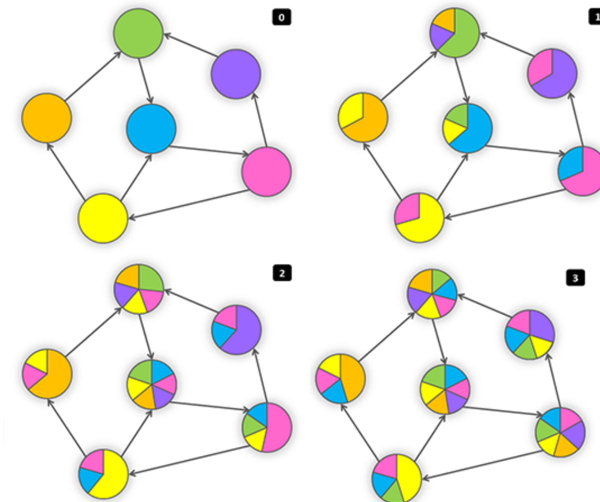
Applications:

- Graph Processing

ITERATIVE MESSAGE PASSING (SOLUTION)

- **Solution:** A network is stored as a set of nodes and each node contains a list of adjacent node IDs. Conceptually, MapReduce jobs are performed in iterative way and at each iteration each node sends messages to its neighbors. Each neighbor updates its state on the basis of the received messages. From the technical point of view, Mapper emits messages for each node using ID of the adjacent node as a key. As result, all messages are grouped by the incoming node and reducer is able to recompute state and rewrite node with the new state.

```
1  class Mapper
2     method Map(id n, object N)
3        Emit(id n, object N)
4        for all id m in N.OutgoingRelations do
5           Emit(id m, message getMessage(N))
6
7  class Reducer
8     method Reduce(id m, [s1, s2,...])
9        M = null
10       messages = []
11       for all s in [s1, s2,...] do
12           if IsObject(s) then
13              M = s
14           else                // s is a message
15              messages.add(s)
16       M.State = calculateState(messages)
17       Emit(id m, item M)
```



REPLICATED JOIN (MAP JOIN, HASH JOIN)

Let's assume that we join two sets – R and L, R is relative small. If so, R can be distributed to all Mappers and each Mapper can load it and index by the join key. The most common and efficient indexing technique here is a hash table. After this, Mapper goes through tuples of the set L and joins them with the corresponding tuples from R that are stored in the hash table. This approach is very effective because there is no need in sorting or transmission of the set L over the network, but set R should be quite small to be distributed to the all Mappers.

```
1  class Mapper
2     method Initialize
3        H = new AssociativeArray : join_key -> tuple from R
4        R = loadR()
5        for all [ join_key k, tuple [r1, r2,...] ] in R
6           H{k} = H{k}.append( [r1, r2,...] )
7
8     method Map(join_key k, tuple l)
9        for all tuple r in H{k}
10          Emit(null, tuple [k r l] )
```