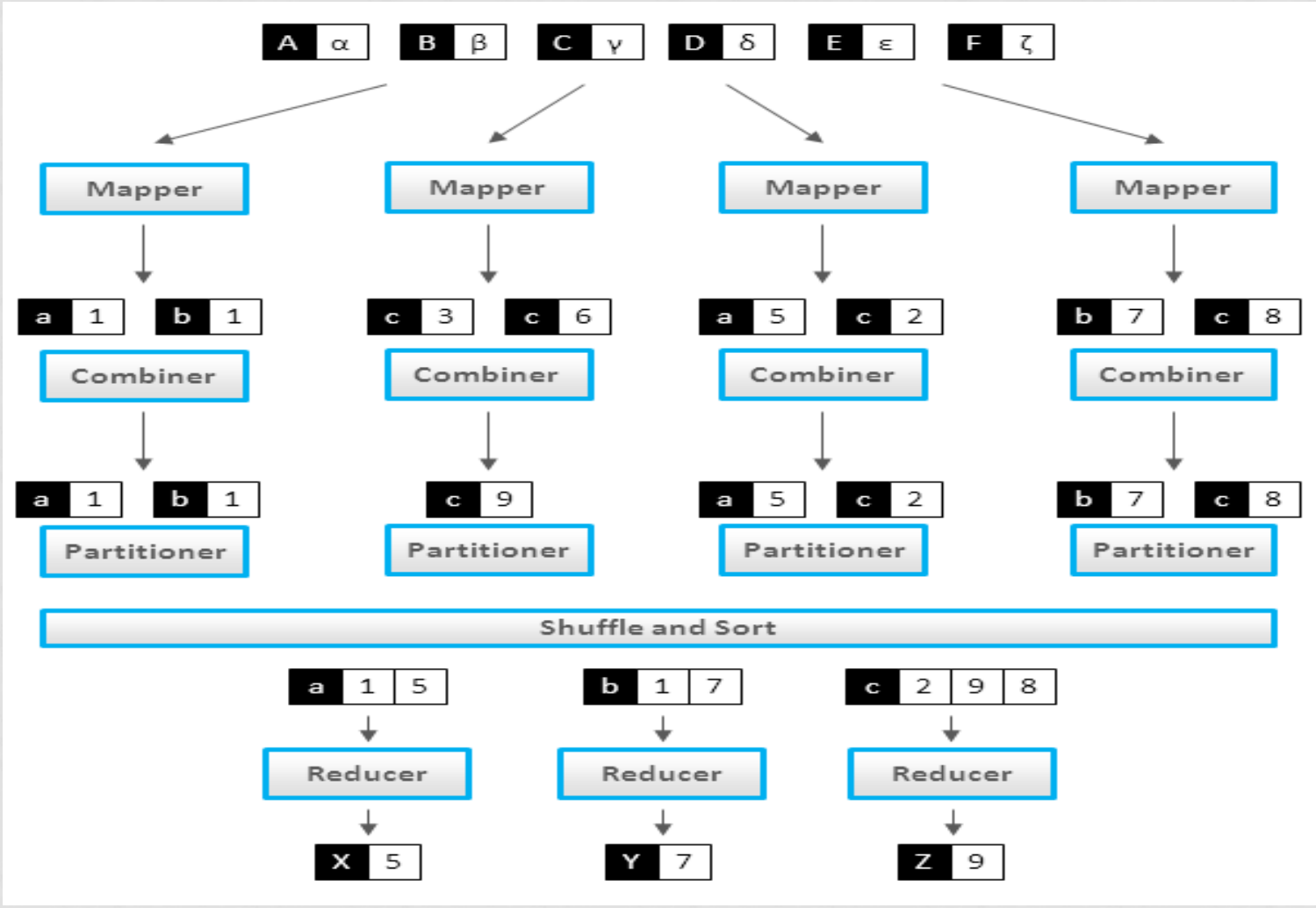
MAP REDUCE PATTERNS

HADOOP'S MAPREDUCE FRAMEWORK



COUNTING AND SUMMING

There is a number of documents where each document is a set of terms. It is required to calculate a total number of occurrences of each term in all documents. Alternatively, it can be an arbitrary function of the terms.

For instance, there is a log file where each record contains a response time and it is required to calculate an average response time. Eg. Log Analysis

```
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 WITH COMBINERS

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

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)
```

COLLATING

There is a set of items and some function of one item. It is required to save all items that have the same value of function into one file or perform some other computation that requires all such items to be processed as a group.

The most typical example is building of inverted indexes.

Mapper computes a given function for each item and emits value of the function as a key and item itself as a value. Reducer obtains all items grouped by function value and process or save them. In case of inverted indexes, items are terms (words) and function is a document ID where the term was found.

FILTERING ("GREPPING"), PARSING, AND VALIDATION

There is a set of records and it is required to collect all records that meet some condition or transform each record (independently from other records) into another representation. The later case includes such tasks as text parsing and value extraction, conversion from one format to another.

Eg: Log Analysis, Data Querying, ETL, Data Validation

Mapper takes records one by one and emits accepted items or their transformed versions.

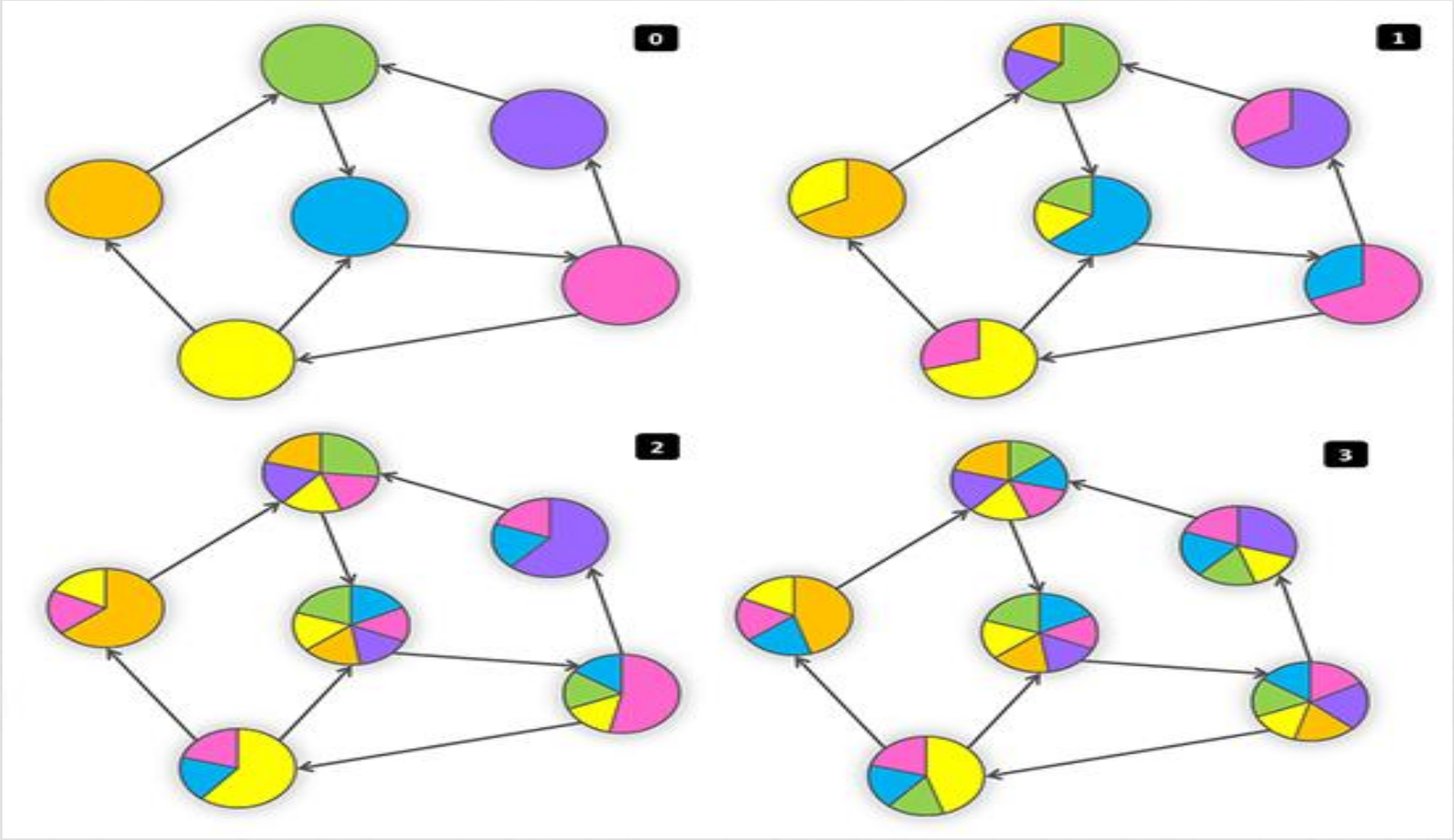
SECONDARY SORTING

- Simple sorting is absolutely straightforward – Mappers just emit all items as values associated with the sorting keys that are assembled as function of items.
- It is very common to use composite keys to achieve secondary sorting and grouping.
- Ex: On clickstream data, to analyze the click trends for each website country-wise

- (K1,V1) –> Map –> (K2,V2)
- (K2,List[V2]) –> Reduce –> (K3,V3)

- (LongWritable,Text) –> Map –> ({symbol,timestamp},price)
- ({symbol,timestamp},List[price]) –> Reduce –> (symbol,price)

ITERATIVE MESSAGE PASSING (GRAPH PROCESSING)



ITERATIVE MESSAGE PASSING (GRAPH PROCESSING)

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.

Ex: Graph Analysis, Web Indexing

Sol:

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. Iterations are terminated by some condition like fixed maximal number of iterations (say, network diameter) or negligible changes in states between two consecutive iterations. 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

ITERATIVE MESSAGE PASSING (GRAPH PROCESSING)

```
class Mapper
  method Map(id n, object N)
    Emit(id n, object N)
    for all id m in N.OutgoingRelations do
      Emit(id m, message getMessage(N))

class Reducer
  method Reduce(id m, [s1, s2,...])
    M = null
    messages = []
    for all s in [s1, s2,...] do
      if IsObject(s) then
        M = s
      else              // s is a message
        messages.add(s)
    M.State = calculateState(messages)
    Emit(id m, item M)
```

PAGERANK AND MAPPER-SIDE DATA AGGREGATION

This algorithm was suggested by Google to calculate relevance of a web page as a function of authoritativeness (PageRank) of pages that have links to this page. The real algorithm is quite complex, but in its core it is just a propagation of weights between nodes where each node calculates its weight as a mean of the incoming weights:

class N

State is PageRank

method getMessage(object N)

return N.State / N.OutgoingRelations.size()

method calculateState(state s, data [d1, d2,...])

return (sum([d1, d2,...]))

PAGERANK AND MAPPER-SIDE DATA AGGREGATION

```
class Mapper
  method Initialize
    H = new AssociativeArray

  method Map(id n, object N)
    p = N.PageRank  / N.OutgoingRelations.size()
    Emit(id n, object N)
    for all id m in N.OutgoingRelations do
      H{m} = H{m} + p

  method Close
    for all id n in H do
      Emit(id n, value H{n})
```

PAGERANK AND MAPPER-SIDE DATA AGGREGATION

```
class Reducer
  method Reduce(id m, [s1, s2,...])
    M = null
    p = 0
    for all s in [s1, s2,...] do
      if IsObject(s) then
        M = s
      else
        p = p + s
    M.PageRank = p
    Emit(id m, item M)
```