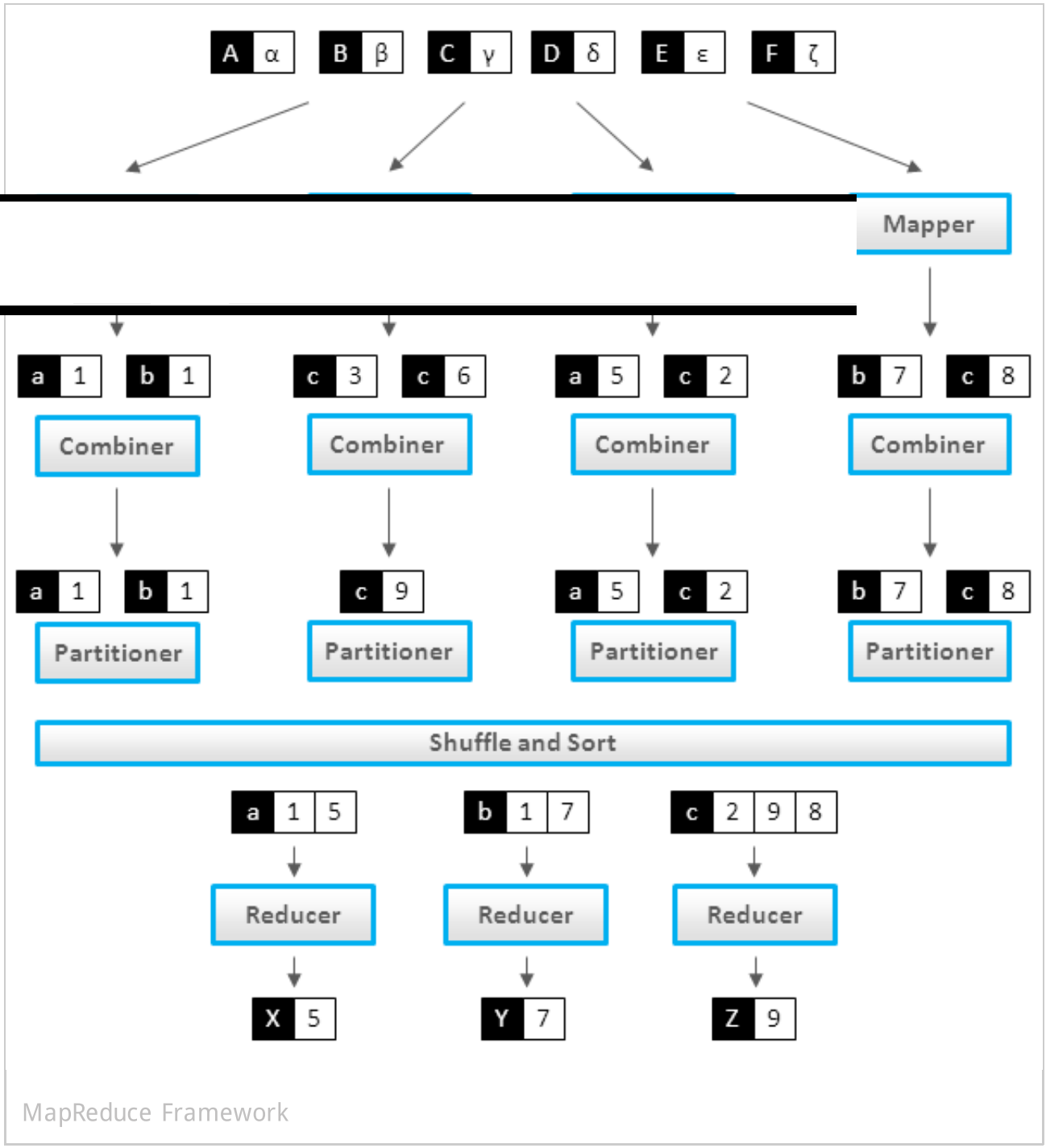
MapReduce Patterns, Algorithms, and Use Cases

highlyscalable.wordpress.com view original

In this article I digested a number of MapReduce patterns and algorithms to give a systematic view of the different techniques that can be found on the web or scientific articles. Several practical case studies are also provided. All descriptions and code snippets use the standard Hadoop's MapReduce model with Mappers, Reduces, Combiners, Partitioners, and sorting. This framework is depicted in the figure below.



MapReduce Framework

Basic MapReduce Patterns

Counting and Summing

Problem Statement: 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.

Solution:

Let start with something really simple. The code snippet below shows Mapper that simply emit "1" for each term it processes and Reducer that goes through the lists of ones and sum them up:

```
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)
```

The obvious disadvantage of this approach is a high amount of dummy counters emitted by the Mapper. The Mapper can decrease a number of counters via summing counters for each document:

```
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})
```

In order to accumulate counters not only for one document, but for all documents processed by one Mapper node, it is possible to leverage 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)
```

Applications:

Log Analysis, Data Querying

Collating

Problem Statement: 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.

Solution:

The solution is straightforward. 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.

Applications:

Inverted Indexes, ETL

Filtering ("Grepping"), Parsing, and Validation

Problem Statement: 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.

Solution: Solution is absolutely straightforward - Mapper takes records one by one and emits accepted items or their transformed versions.

Applications:

Log Analysis, Data Querying, ETL, Data Validation

Distributed Task Execution

Problem Statement: There is a large computational problem that can be divided into

multiple parts and results from all parts can be combined together to obtain a final result.

Solution: Problem description is split in a set of specifications and specifications are stored as input data for Mappers. Each Mapper takes a specification, performs corresponding computations and emits results. Reducer combines all emitted parts into the final result.

Case Study: Simulation of a Digital Communication System

There is a software simulator of a digital communication system like WiMAX that passes some volume of random data through the system model and computes error probability of throughput. Each Mapper runs simulation for specified amount of data which is 1/Nth of the required sampling and emit error rate. Reducer computes average error rate.

Applications:

Physical and Engineering Simulations, Numerical Analysis, Performance Testing

Sorting

Problem Statement: There is a set of records and it is required to sort these records by some rule or process these records in a certain order.

Solution: Simple sorting is absolutely straightforward - Mappers just emit all items as values associated with the sorting keys that are assembled as function of items. Nevertheless, in practice sorting is often used in a quite tricky way, that's why it is said to be a heart of MapReduce (and Hadoop). In particular, it is very common to use composite keys to achieve secondary sorting and grouping.

Sorting in MapReduce is originally intended for sorting of the emitted key-value pairs by key, but there exist techniques that leverage Hadoop implementation specifics to achieve sorting by values. See this blog for more details.

It is worth noting that if MapReduce is used for sorting of the original (not intermediate) data, it is often a good idea to continuously maintain data in sorted state using BigTable concepts. In other words, it can be more efficient to sort data once during insertion than sort them for each MapReduce query.

Applications:

ETL, Data Analysis

Not-So-Basic MapReduce Patterns

Iterative Message Passing (Graph Processing)

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.

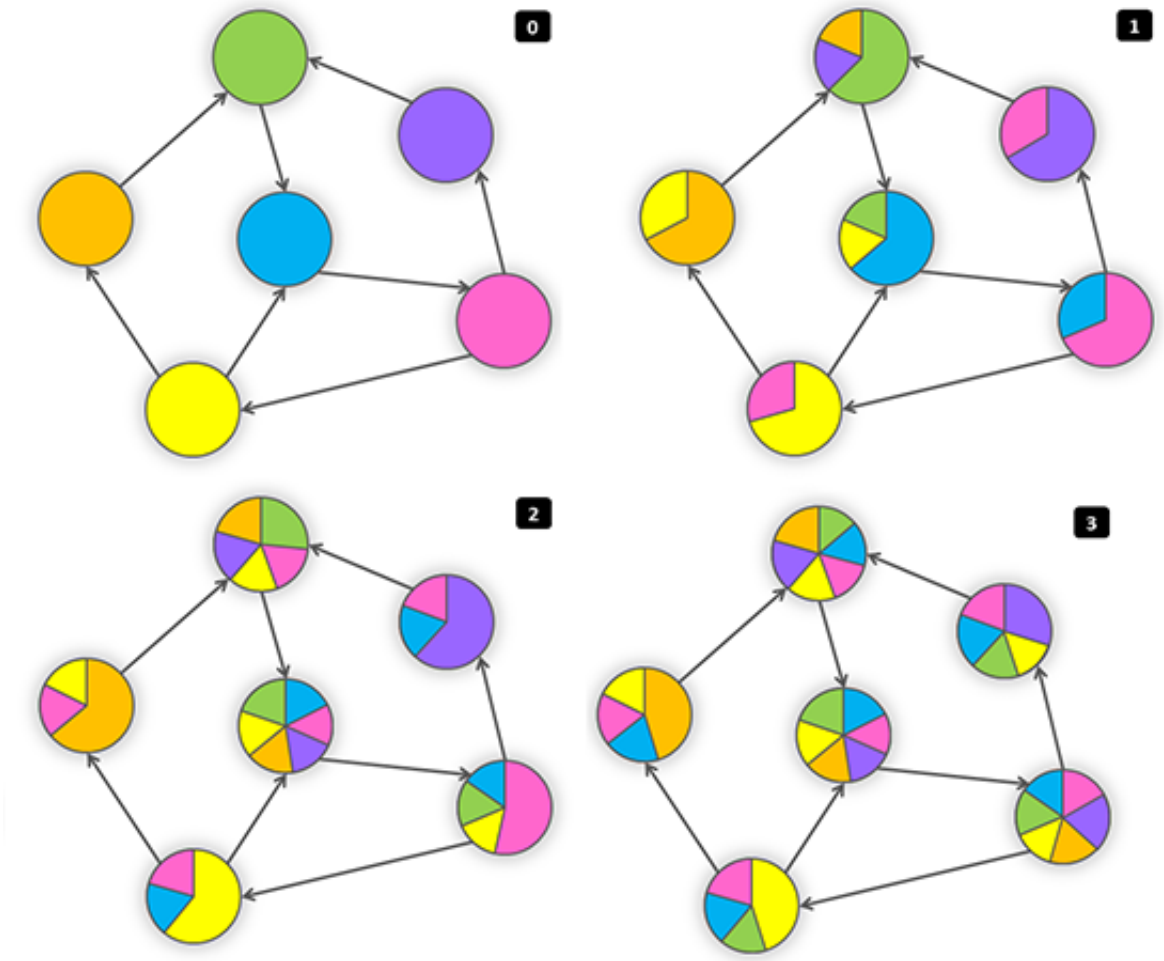
Solution: A network is stored as a set of nodes and each node contains a list of adjacent node IDs. Conceptually, MapReduce jobs are performed in iterative way and at each iteration each node sends messages to its neighbors. Each neighbor updates its state on the basis of the received messages. Iterations 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. This algorithm is shown in the figure below:

```
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)
```

It should be emphasized that state of one node rapidly propagates across all the network of network is not too sparse because all nodes that were “infected” by this state start to “infect” all their neighbors. This process is illustrated in the figure below:



Case Study: Availability Propagation Through The Tree of Categories

Problem Statement: This problem is inspired by real life eCommerce task. There is a tree of categories that branches out from large categories (like Men, Women, Kids) to smaller ones (like Men Jeans or Women Dresses), and eventually to small end-of-line categories (like Men Blue Jeans). End-of-line category is either available (contains products) or not. Some high level category is available if there is at least one available end-of-line category in its subtree. The goal is to calculate availabilities for all categories if availabilities of end-of-line categories are know.

Solution: This problem can be solved using the framework that was described in the previous section. We define getMessage and calculateState methods as follows:

```
class N
```

```
    State in {True = 2, False = 1, null = 0}, initialized 1 or 2 for end-of-line
categories, 0 otherwise

method getMessage(object N)
    return N.State

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

Case Study: Breadth-First Search

Problem Statement: There is a graph and it is required to calculate distance (a number of hops) from one source node to all other nodes in the graph.

Solution: Source node emits 0 to all its neighbors and these neighbors propagate this counter incrementing it by 1 during each hope:

```
class N
    State is distance, initialized 0 for source node, INFINITY for all other nodes

method getMessage(N)
    return N.State + 1

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

Case Study: 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,...]) )
```

It is worth mentioning that the schema we use is too generic and doesn't take advantage of the fact that state is a numerical value. In most of practical cases, we can perform aggregation of values on the Mapper side due to virtue of this fact. This optimization is illustrated in the code snippet below (for the PageRank algorithm):

```
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})

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)
```

Applications:

Graph Analysis, Web Indexing

Distinct Values (Unique Items Counting)

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

The problem can be a little bit generalized and formulated in terms of faceted search:

Problem Statement: There is a set of records. Each record has field F and arbitrary number of category labels G = {G1, G2, ...} . Count the total number of unique values of filed F for each subset of records for each value of any label. Example:

```
Record 1: F=1, G={a, b}
Record 2: F=2, G={a, d, e}
Record 3: F=1, G={b}
Record 4: F=3, G={a, b}

Result:
a -> 3   // F=1, F=2, F=3
b -> 2   // F=1, F=3
d -> 1   // F=2
e -> 1   // F=2
```

Solution I:

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.

Phase I:

```
class Mapper
    method Map(null, record [value f, categories [g1, g2,...]])
        for all category g in [g1, g2,...]
            Emit(record [g, f], count 1)

class Reducer
    method Reduce(record [g, f], counts [n1, n2, ...])
        Emit(record [g, f], null )
```

Phase II:

```
class Mapper
    method Map(record [f, g], null)
        Emit(value g, count 1)

class Reducer
    method Reduce(value g, counts [n1, n2,...])
        Emit(value g, sum( [n1, n2,...] ) )
```

Solution II:

The second solution requires only one MapReduce job, but it is not really scalable and its applicability is limited. The algorithm is simple - Mapper emits values and categories, Reducer excludes duplicates from the list of categories for each value and increment

counters for each category. The final step is to sum all counter emitted by Reducer. This approach is applicable if th number of record with the same f value is not very high and total number of categories is also limited. For instance, this approach is applicable for processing of web logs and classification of users - total number of users is high, but number of events for one user is limited, as well as a number of categories to classify by. It worth noting that Combiners can be used in this schema to exclude duplicates from category lists before data will be transmitted to Reducer.

```
class Mapper
    method Map(null, record [value f, categories [g1, g2,...] )
        for all category g in [g1, g2,...]
            Emit(value f, category g)

class Reducer
    method Initialize
        H = new AssociativeArray : category -> count
    method Reduce(value f, categories [g1, g2,...])
        [g1', g2',..] = ExcludeDuplicates( [g1, g2,..] )
        for all category g in [g1', g2',...]
            H{g} = H{g} + 1
    method Close
        for all category g in H do
            Emit(category g, count H{g})
```

Applications:

Log Analysis, Unique Users Counting

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.

Pairs Approach

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

```
class Mapper
    method Map(null, items [i1, i2,...] )
        for all item i in [i1, i2,...]
            for all item j in [i1, i2,...]
                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)
```

Stripes Approach

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”

```
class Mapper
    method Map(null, items [i1, i2,...] )
        for all item i in [i1, i2,...]
            H = new AssociativeArray : item -> counter
            for all item j in [i1, i2,...]
                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})
```

Applications:

Text Analysis, Market Analysis

References:

Relational MapReduce Patterns

In this section we go though the main relational operators and discuss how these operators can implemented in MapReduce terms.

Selection

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

Projection

Projection is just a little bit more complex than selection, but we should use a Reducer in this case to eliminate possible 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)
```

Union

Mappers are fed by all records of two sets to be united. Reducer is used to eliminate duplicates.

```
class Mapper
    method Map(rowkey key, tuple t)
        Emit(tuple t, null)

class Reducer
```

```
    method Reduce(tuple t, array n)   // n is an array of one or two nulls
      Emit(tuple t, null)
```

Intersection

Mappers are fed by all records of two sets to be intersected. Reducer emits only records that occurred twice. It is possible only if both sets contain this record because record includes primary key and can occur in one set only once.

```
class Mapper
    method Map(rowkey key, tuple t)
      Emit(tuple t, null)

class Reducer
    method Reduce(tuple t, array n)   // n is an array of one or two nulls
      if n.size() = 2
          Emit(tuple t, null)
```

Difference

Let's we have two sets of records - R and S. We want to compute difference R - S. Mapper emits all tuples and tag which is a name of the set this record came from. Reducer emits only records that came from R but not from S.

```
class Mapper
    method Map(rowkey key, tuple t)
      Emit(tuple t, string t.SetName)     // t.SetName is either 'R' or 'S'

class Reducer
    method Reduce(tuple t, array n) // array n can be ['R'], ['S'], ['R' 'S'], or ['S',
'R']
      if n.size() = 1 and n[1] = 'R'
```

```
        Emit(tuple t, null)
```

GroupBy and Aggregation

Grouping and aggregation can be performed in one MapReduce job as follows. Mapper extract from each tuple values to group by and aggregate and emits them. Reducer receives values to be aggregated already grouped and calculates an aggregation function.
Typical aggregation functions like sum or max can be calculated in a streaming fashion, hence don't require to handle all values simultaneously. Nevertheless, in some cases two phase MapReduce job may be required - see pattern Distinct Values as an example.

```
class Mapper
    method Map(null, tuple [value GroupBy, value AggregateBy, value ...])
        Emit(value GroupBy, value AggregateBy)
class Reducer
    method Reduce(value GroupBy, [v1, v2,...])
        Emit(value GroupBy, aggregate( [v1, v2,...] ) )  // aggregate() : sum(),
max(),...
```

Joining

Joins are perfectly possible in MapReduce framework, but there exist a number of techniques that differ in efficiency and data volumes they are oriented for. In this section we study some basic approaches. The references section contains links to detailed studies of join techniques.

Repartition Join (Reduce Join, Sort-Merge Join)

This algorithm joins of two sets R and L on some key k. Mapper goes through all tuples from R and L, extracts key k from the tuples, marks tuple with a tag that indicates a set this tuple came from ('R' or 'L'), and emits tagged tuple using k as a key. Reducer receives all tuples for a particular key k and put them into two buckets - for R and for L. When two

buckets are filled, Reducer runs nested loop over them and emits a cross join of the buckets. Each emitted tuple is a concatenation R-tuple, L-tuple, and key k. This approach has the following disadvantages:

- Mapper emits absolutely all data, even for keys that occur only in one set and have no pair in the other.
- Reducer should hold all data for one key in the memory. If data doesn't fit the memory, its Reducer's responsibility to handle this by some kind of swap.

Nevertheless, Repartition Join is a most generic technique that can be successfully used when other optimized techniques are not applicable.

```
class Mapper
    method Map(null, tuple [join_key k, value v1, value v2,...])
        Emit(join_key k, tagged_tuple [set_name tag, values [v1, v2, ...] ] )

class Reducer
    method Reduce(join_key k, tagged_tuples [t1, t2,...])
        H = new AssociativeArray : set_name -> values
        for all tagged_tuple t in [t1, t2,...]      // separate values into 2 arrays
            H{t.tag}.add(t.values)
        for all values r in H{'R'}                  // produce a cross-join of the two
arrays
            for all values l in H{'L'}
                Emit(null, [k r l] )
```

Replicated Join (Map Join, Hash Join)

In practice, it is typical to join a small set with a large one (say, a list of users with a list of log records). 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.

```
class Mapper
    method Initialize
        H = new AssociativeArray : join_key -> tuple from R
        R = loadR()
        for all [ join_key k, tuple [r1, r2,...] ] in R
            H{k} = H{k}.append( [r1, r2,...] )

    method Map(join_key k, tuple l)
        for all tuple r in H{k}
            Emit(null, tuple [k r l] )
```