Department of Computer Science
University of Cyprus

EPL646 – Advanced Topics in Databases

Lecture 16

Big Data Management VI
(MapReduce Programming)

Credits: Pietro Michiardi (Eurecom): Scalable Algorithm Design, Apache MapReduce Tutorial

Demetris Zeinalipour
http://www.cs.ucy.ac.cy/~dzeina/courses/epl646

Outline of Lecture

- MapReduce "Hello World" Program Explained
  - Wordcount in MR, Example Execution, Pseudocode
  - Mean Computation in MR, JAVA API Preview
- Operational Issues:
  - What to configure and what not
- Combiners and In-Memory Combiners
- Relational Operators in MR
  - Selection/Projection
  - Union / Intersection / Set Difference
  - Join /Aggregation

Introduction to Hadoop Programming

- In the previous lecture we learnt how a MapReduce program executes in a Hadoop Environment without actually seeing the program.
- In this lecture we will learn more about the basic principles on how to write MapReduce Programs in Hadoop.
- To validate some of the ideas in this lecture, ensure that Hadoop is installed, configured and running. More details:
  - Single Node Setup for first-time users.
  - Cluster Setup for large, distributed clusters.
- In our laboratory we will use a Single Node Setup (consult the image that has been circulated by the TA).
  - Hadoop v2 requires Java 7 or greater
  - Hadoop v3 requires Java 8 – our labs & assignments ☺
  - New Features: HDFS Erasure encoding, YARN v2 Timeline service (HBase store) Opportunistic containers, 2 Namenodes, default ports changed, Filesystem Connectors (e.g., Microsoft Azure Data Lake), Intra-DataNode Balancer

MapReduce "Hello World" (WordCount 1/2)

```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
```

As of JAVA 7 Generic Example: no way to verify, at compile time, how the class is used (e.g., as Integer, String, etc. ☹

```
public class Box {
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```

Input(k,v) *Output(k',v')*

```
public class WordCount {
  public static class TokenizerMapper  extends Mapper<Object, Text, Text, IntWritable> {

    private final static IntWritable one = new IntWritable(1); // optimized serialization of JAVA.Integer() class
    private Text word = new Text();

    public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken()); context.write(word, one);
      }
    } // map
}
```

Input(k,v)

Output(k',v')

Applications may override the [run(org.apache.hadoop.mapreduce.Mapper.Context)]() method to exert greater control on map processing e.g. field delimiters, etc.

MapReduce "Hello World" (WordCount 2/2)

Input(k,v) Output(k',v')

```
public static class IntSumReducer  extends Reducer<Text,IntWritable,Text,IntWritable> {
 private IntWritable result = new IntWritable();
 public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {
   int sum = 0;
   for (IntWritable val : values) {
     sum += val.get();
   }
   result.set(sum);
   context.write(key, result);
 }
}
```

Output(k',v')

> Implementing this interface allows an object to be the target of the "for-each loop" statement. (as of 1.5)

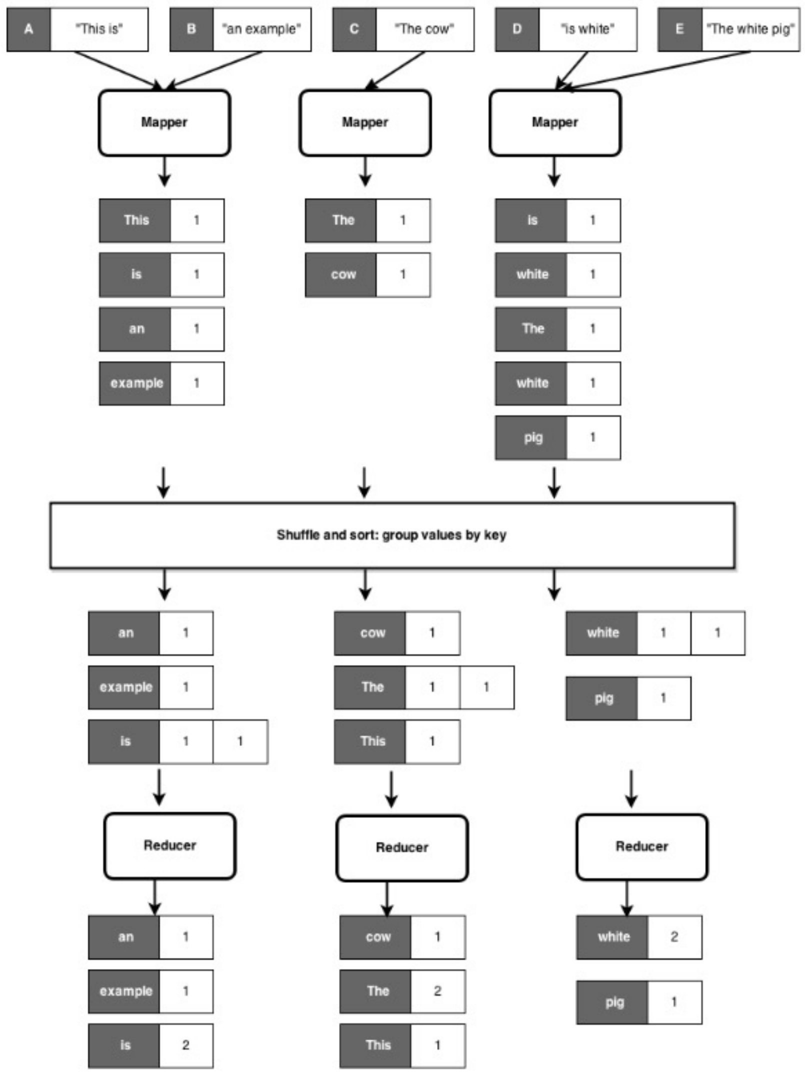
As of JAVA 5 Enhanced For Loop: iterate through all the elements of a ***Collection***.

```
public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();
  Job job = Job.getInstance(conf, "word count");
  job.setJarByClass(WordCount.class);
  job.setMapperClass(TokenizerMapper.class);
  job.setCombinerClass(IntSumReducer.class);
  job.setReducerClass(IntSumReducer.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);
  FileInputFormat.addInputPath(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));
  System.exit(job.waitForCompletion(true) ? 0 : 1);
 }
}
```

Cleanup(), setup() are not mandatory in Mapper (see next slide)!

Multi-threading is possible with **MultithreadedMapper** when the Mapper is **not CPU bound** (the time to complete the task is **not** determined by the slow CPU but the Mapper logic).
=> *If using more CPU power would speedup the program execution.*

Execution (Wordcount, Anagram)



In laboratory you saw the **anagram problem**

sort

Map("eilnst", Silent")
Map("eilnst", "Listen")

Reduce("eilnst", [Silent, Listen])

=> Each reducer takes care of each key.

Remember!
The Map() and Reduce() blocks are the personal loops of the tasks, i.e.,

- 1 Map per partitioned data group.
- 1 Reduce per unique key.

Mapper Functions (Reducer Similar)

Mapper Functions

•protected void **setup**(org.apache.hadoop.mapreduce.Mapper.Context context) throws IOException, InterruptedException

- **Called once at the beginning of the task.**

•protected void **map(KEYIN** key, VALUEIN value, org.apache.hadoop.mapreduce.Mapper.Context context) throws IOException, InterruptedException

- **Called once for each key/value pair in the input split. Most applications should override this, but the default is the identity function (k,v) => (k,v).**

•protected void **cleanup(**org.apache.hadoop.mapreduce.Mapper.Context context) throws IOException, InterruptedException

- **Called once at the end of the task.**

•public void **run(**org.apache.hadoop.mapreduce.Mapper.Context context) throws IOException, InterruptedException

- **Expert users can override this method for more complete control over the execution of the Mapper.**

•Methods inherited from class java.lang.Object: clone, equals, finalize, getClass, hashCode, toString, thread control: notify, notifyAll, wait

•How to run the code – More details:
https://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapreduce/Mapper.html#Mapper()

Word Count Pseudocode (easier for next slides)

"Hello World" in "Map Reduce"

```
class MAPPER
    method MAP(offset a, line l)
        for all term t ∈ line l do
            EMIT(term t, count 1)

class REDUCER
    method REDUCE(term t, counts [c1, c2, ...])
        sum ← 0
        for all count c ∈ counts [c1, c2, ...] do
            sum ← sum + c
        EMIT(term t, count sum)
```

Both MAPPER and REDUCER could skip EMIT(), e.g., the case of filter

Iterator <> TABLE (we need to begin from the beginning in the iterator if we want to rewind, iterator.reset())

Example: Mean Computation

Problem:

- We have a large dataset where **input keys** are **strings** and **input values are integers (e.g., http://www.cs.ucy.ac.cy, 10s)**
- We wish to compute the mean of all integers associated with the same key
- Almost identical to “word count” (now “word average”)!

```
class Mapper
    method Map(string t, integer r)
        Emit(string t, integer r)
```

Identity emit function (could be omitted as it is default)

```
class Reducer
    method Reduce(string t, integers [r1, r2, . . .])
        sum ← 0
        cnt ← 0
        for all integer r ∈ integers [r1, r2, . . .] do
            sum ← sum + r
            cnt ← cnt + 1
        ravg ← sum/cnt
        Emit(string t, integer ravg)
```

Note!: A good idea here would be to use a symmetric hash (e.g., MD5), i.e., key=md5(URL) instead of key=URL to minimize string comparisons. At the end we could URL = reverse-md5(md5(URL))

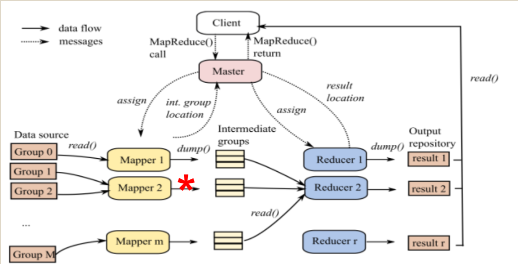
Operational Issues

Aspects that are not under the control of the designer

- *Where* a mapper or reducer will run
- *When* a mapper or reducer begins or finishes
- *Which* input key-value pairs are processed by a specific mapper
- *Which* intermediate key-value pairs are processed by a specific reducer

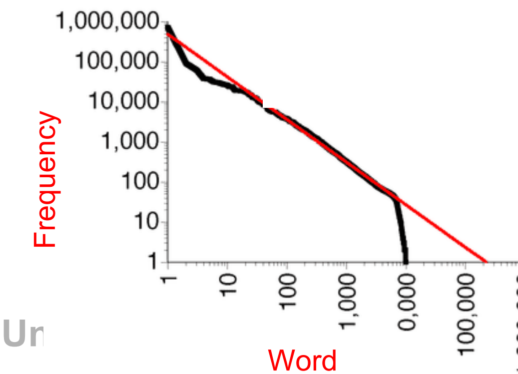
Aspects that can be controlled

- Construct data structures (intermediate results) as keys and values
- Execute user-specified **initialization** (setup()) and termination code (cleanup()) for mappers and reducers
- **Preserve state** across multiple input and intermediate keys in mappers and reducers (**in-memory combiners** – discussed next)
- Control the **sort order** of **intermediate keys**, and therefore the order **in which a reducer** will encounter particular keys.
- Control the **partitioning** **of the key space**, and therefore the set of keys that **will be encountered by a particular reducer.**



Combiners (optional)

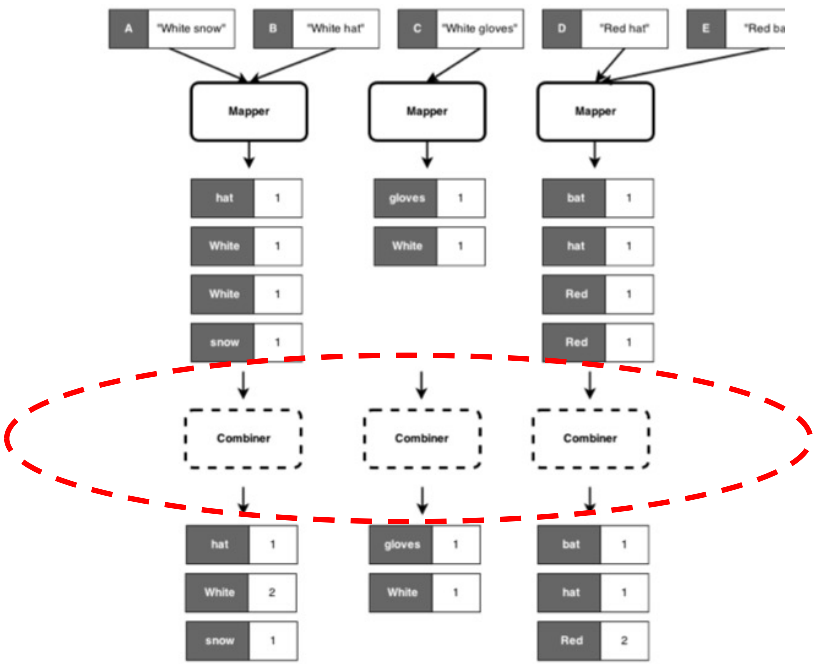
- **Combiners are a general mechanism to reduce the amount of intermediate data (after the map task)**
  - They could be thought of as “mini-reducers” before data is shipped to reducers
  - Reduce the number and size of key-value pairs to be shuffled
- **Back to our running example: word count**
  - Combiners aggregate term counts across documents processed by each map task
  - If combiners take advantage of all opportunities for local aggregation we have at most $\boldsymbol{m} \times \boldsymbol{V}$ intermediate key-value pairs
    - m: number of mappers
    - V : number of unique terms in the collection
  - Note: due to Zipfian nature of term distribution not all mappers will see all terms.



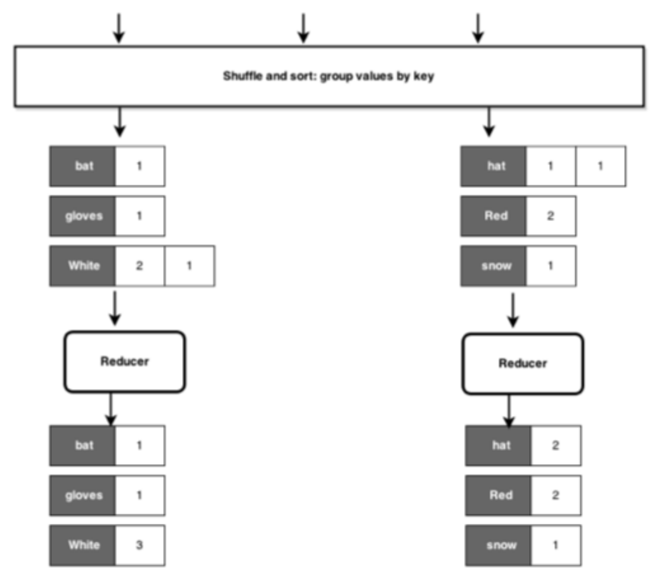
Combiners (optional)

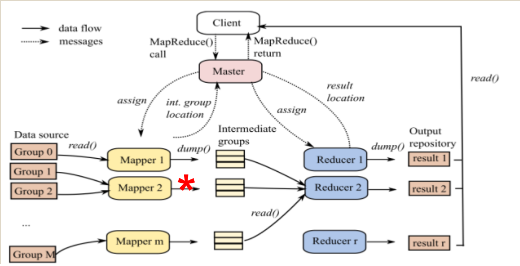
- **The use of combiners must be thought carefully**
  - In Hadoop, they are **optional:** the correctness of the algorithm cannot depend on computation (or even execution) of the combiners
  - In Apache Spark, they're mostly automatic

Combiners



Shuffling
mapreduce.reduce.shuffle.





Combiners (optional)

- **Hadoop does not guarantee combiners to be executed**
  - Actually, combiners might only be called if the number of map output records is greater than a threshold, *i.e.*, 4
- ***Problem:*** *Can we enforce the execution of aggregation at the end of the Map phase?*
  - ☺ Yes, by implementing the aggregation logic in Mapper.
  - ☹ Not always very good as it the function state:
    - In-memory combining breaks the functional programming paradigm due to **state preservation.**
    - In-memory combining strictly depends on having sufficient memory to store intermediate results
    - A possible solution: “block” and “flush”
  - Nevertheless, let’s see an example…

In-Memory Combiners (inside Mapper)

class MAPPER
  method MAP(offset a, line l)
    $H \leftarrow$ new HashMap
    for all term $t \in$ line l **do**
      $H\{t\} \leftarrow H\{t\} + 1$
    for all term $t \in H$ **do**
      EMIT(term t, count $H\{t\}$)

- **We use a hash map to accumulate intermediate results**
  - The data structure is also know as “associative array” or “dictionary”
  - The array is used to tally up (aggregate) term counts within a single “document”
  - The Emit method is called only after all InputRecords have been processed

Selections (σ) in MapReduce

- **Revision of** *Relational Algebra Operators*
  – *http://www2.cs.ucy.ac.cy/~dzeina/courses/epl342/schedule.html (Lecture 8 and 9).*
- **In practice, selections do not need a full-blown MapReduce implementation**
  – They can be implemented in the map phase alone
  – Actually, they could also be implemented in the reduce portion!
  – Remember that the input to **Reduce** is an **Iterator** (it is constructed as the packets arrive at the reducer not a fully constructed list on which).
- **A MapReduce implementation of $\sigma_C(R)$**
  – For each tuple ***t*** in ***R***, check if ***t*** satisfies ***C***
  – If so, emit a key/value pair (**t, NULL**)
  – (VOID) Identity Reducer

Projections (π) in MapReduce

- **A note on *duplicates* in projections:**
  - **Relational Algebra (π)** generates NO duplicates (RA operates with sets). Notation we use: $\pi_{DISTINCT\ S}(R)$
  - **SQL (SELECT): generates duplicates (SQL operates with multisets).** Notation we use SELECT: $\pi_S(R)$. Of course there is also SELECT DISTINCT, again notated with $\pi_{DISTINCT\ S}(R)$
- **How to implement Projections in MR?**
  - **$\pi_S(R)$ (ALL):** Keeps Duplicates => Only requires map task.
  - **$\pi_{DISTINCT\ S}(R)$:** Removes Duplicates => Requires map + reduce
- **$\pi_{DISTINCT\ S}(R)$ Implementation**

MAP

- For each tuple ***t*** in *R*, construct a tuple **t'** that contains only the S columns.
- Emit a key/value pair (**t',NULL**)

REDUCE

- Foreach key **t'** obtained from mappers (**t'**, **[<NULL>]**), take the key only.
- Emit a key/value pair (***t', NULL***)

Unions (∪) in MapReduce

$R \cup S = \{ x \mid x \in R \vee x \in S \}$

– Relations ***R*** and ***S*** *must* have the same schema!

R: 1, 2, 2, 2, 3, 4, 4
S: 2, 3, 4, 4, 4, 5

R **UNION** S:
1, 2, 3, 4, 5
R **UNION** ALL S:
1, 2, 2, 2, 3, 4, 4, 2, 3, 4, 4, 4, 5

- **A note on *duplicates* in unions:**
  - **$R \cup_{ALL} S$:** Keeps Duplicates => Requires Map Task only
  - **$R \cup S$:** Removes Duplicates => Requires Map + Reduce Task
- **Outline of $R \cup S$ Implementation:**
  - Map tasks will be assigned chunks from either *R* or *S* *
  - Mappers don't do much, just pass by to reducers. Reducers do duplicate elimination **(not necessary in** $R \cup_{ALL} S$**)**
  - * Note: Hadoop MapReduce supports reading multiple inputs.
- **How to implement $R \cup S$ in MR?**

  MAP: For each tuple **t** in R and S, emit a key/value pair (**t, NULL**) // identity function

  REDUCE

  – Foreach key **t'** obtained from mappers (**t'**, **[<NULL>]**), take the key only.

  – Emit a key/value pair (***t'***, ***NULL***) **// i.e., Either an R tuple or an S tuple**.

  – Also works when R and/or S have duplicates => Still generates (**t'**, **[<NULL>]**),

Intersection (∩) in MapReduce

$$R \cap S = \{ x \mid x \in R \wedge x \in S \}$$

– Relations ***R*** and ***S*** *must* have the same schema!

Examples:
R: 1, **2**, 2, 2, **3**, **4**, **4**
S: **2**, **3**, **4**, **4**, 4, 5

R ***INTERSECT*** *S:* 2, 3, 4
R ***INTERSECT ALL*** *S:* 2, 3, 4, 4

- **A note on *duplicates* in intersections:**
  - **R ∩ S :** Removes Duplicates => Map + Reduce
  - **R $\cap_{ALL}$ S:** Keeps Duplicates=> Map+Reduce (not available in most DBMS)
- **Outline of** R ∩ S **Implementation:**
  - Map tasks will be assigned chunks from either *R* or *S*
  - Mappers don't do much, just pass by to reducers. Reducers do duplicate elimination **(not necessary in** R $\cap_{ALL}$ S**)**
- **How to implement** R ∩ S **in MR (R, S: no duplicates)**

  MAP: For each tuple **t** in R and S, emit a key/value pair (**t, t**)

  REDUCE
  - Foreach key **t'** obtained from the mappers, if **(t', [t', t']) (i.e., these 2 entries must have come from R and S)** then emit the key/value pair **(t', NULL)**
  - Otherwise, **emit nothing // i.e., (t', [t']) OR (t', [<NULL>])**
  - **// If R, S contain duplicates, I must annotate the t in map (t,'R'), (t,'S'), … to avoid self-intersection within R and within S, respectively.**

Set Difference (-) Revision

$R - S = \{ x \mid x \in R \wedge x \notin S \}$ Note: $R - S \neq S - R$

– Relations ***R*** and ***S*** *must* have the same schema!

(a) STUDENT

Fn	Ln
Susan *	Yao
Ramesh*	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

|Student|=7

INSTRUCTOR

Fname	Lname
John	Smith
Ricardo	Browne
Susan *	Yao
Francis	Johnson
Ramesh*	Shah

|Instructor|=5

|Student– Instructor| = 5

(d)

Fn	Ln
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

|Instructor – Student| = 3

(e)

Fname	Lname
John	Smith
Ricardo	Browne
Francis	Johnson

Set Difference (-) in MR

- **Outline of R - S Implementation:**
  - The map function passes tuples from R and S to the reducer
  - **it must inform the reducer whether the tuple came from R or S!**

- **How to implement R - S in MR?**

  MAP:

  - For a tuple ***t*** in *R* emit a key/value pair **(*t*, 'R')** and for a tuple ***t*** in *S*, emit a key/value pair **(*t*,'S')**

  REDUCE:

  - For each key **t**, do the following:
    - If the input is **(t,['R']), then emit (*t, NULL*)**
    - If the input is **(t,['R', 'S'])** or **(t,['S', 'R'])**, or **(t,['S']), don't emit anything!**

Join ($\bowtie$,$\otimes$) in MapReduce

$$R \otimes_{<attr>} S \;=\; \sigma_{<attr>}(R \times S)$$

- **This topic is subject to continuous refinements**
  – There are many JOIN operators and many different implementations
- **Let's look at two relations** $R(A, \boldsymbol{B})$ **and** $S(\boldsymbol{B}, C)$
  – We must find tuples that agree on their *B* components
  – We shall use the ***B*-value** of tuples from either relation as **the key**
  – **The value** will be the **other component** and the name of the relation
  – That way the reducer knows from which relation each tuple is coming from
- **How to implement R ⊗ S in MR?**

MAP:

– For each tuple (a,b) of R emit the key/value pair **(b, ('R',a))**

– For each tuple (b,c) of S emit the key/value pair **(b, ('S',c))**

REDUCE:

– Each key **b** will be associated to a list of pairs that are either **('R', a)** or **('S', c)**

– Generate key/value pairs **$(b,[(a_1,b,c_1), (a_2,b,c_2), \ldots, (a_n,b,c_n)]$** and emit the **unique triples** (a,b,c) => the final unique step would be best to be implemented with a second MR job – see assignment!

Aggregation (γ) in MapReduce

- We already discussed Aggregates, remember the Mean Example.
  - **Map:** The map operation prepares the grouping
  - **Reduce:** The reducer computes the aggregation.
  - **Simplifying assumptions:** one grouping attribute and one aggregation function. – easy to lift these assumptions and generalize the discussion.
- Different Types of Aggregates :
  - **Distributive Aggregates:** COUNT,SUM,MAX,MIN,AVG(S/C) **=>** *reduce uses a rolling computation of the aggregate.*
  - **Holistic Aggregates:** MEDIAN, MEAN **=>** *reduce has to retain all incoming tuples (coming through the iterator) (k,[v1,…,vn]) and then compute the aggregate.*
- **How to implement $\gamma_{A,\theta(B)}R$ in MR?**

  MAP: For a tuple ***t (a,b,c)*** in *R* emit a key/value pair **(a,b)**

  REDUCE: For each key **t,** do the following:
  - Distributive Aggregates: Apply **θ** on the incoming tuples **[b1,…bn]** on-the-fly
  - Holistic Aggregates: Accumulate the incoming tuples in a table. At the end apply **θ** on the constructed table.
  - Emit the key/value pair **(a,x)** where **x = θ([b1,…,bn])**

Generalizing MR Operators

- Having developed operators for **basic RA operators**, one could now develop **higher-level declarative languages (e.g., SQL, PIG)** that translate into **MR jobs**!
- Example **Apache HUE** on top of **HIVE** (Hadoop / HDFS)