HADOOP MAP REDUCE (JOINING DATA)

Master Program in Computer Science
University of Calabria

Prof. F. Ricca

Joining data

- Fundamental operation in databases
  - Needed in data analysis
  - Several join types
    - Natural join
    - (left,right,full) Outer join
    - Semi Join
- Might be necessary to join different sources
  - Some sources might be "big"
- How to implement join operations in Map Reduce?

Some join "patterns" (1)

- **Reduce-side join**
  - Aka repartitioned join (or the repartitioned sort-merge join)
  - The most common
  - Joining takes place in the reduce phase
    - *Often not very efficient*, we will discuss why
  - Supported in the DataJoin Package

- **Map-side join**
  - Aka Replicated join
  - Replicate (share) one of the sources in the mapper
    - Joining takes place in the map phase
    - Possible if some of the join sources is small enough

Some join "patterns" (2)

- **Reduce-side join with map-side filtering**
  - Combines the above two
  - Semi join-based
  - Based on *bloom filters* (or similar)

- **Secondary sort**
  - Aka value-to-key conversion
  - Joins with "group by"
  - Sort the values (in ascending or descending order) passed to each reducer
    - Not by key!!

Reduce-side Join

- The most general join technique
  - Not the most efficient joining
- Repartitioned join (or the repartitioned sort-merge join)
  - Mappers repartition the source
  - Shuffling&Sorting w.r.t the "key" done by the framework
  - Reducers (merge) joins the sources
- Some terminology
  - A **data source** is analogous to a table in relational databases
  - Record **tag** useful to persist some record information
  - The **group key** as the join key in a relational databases

Query 1: find all products sold (and their associated locations):

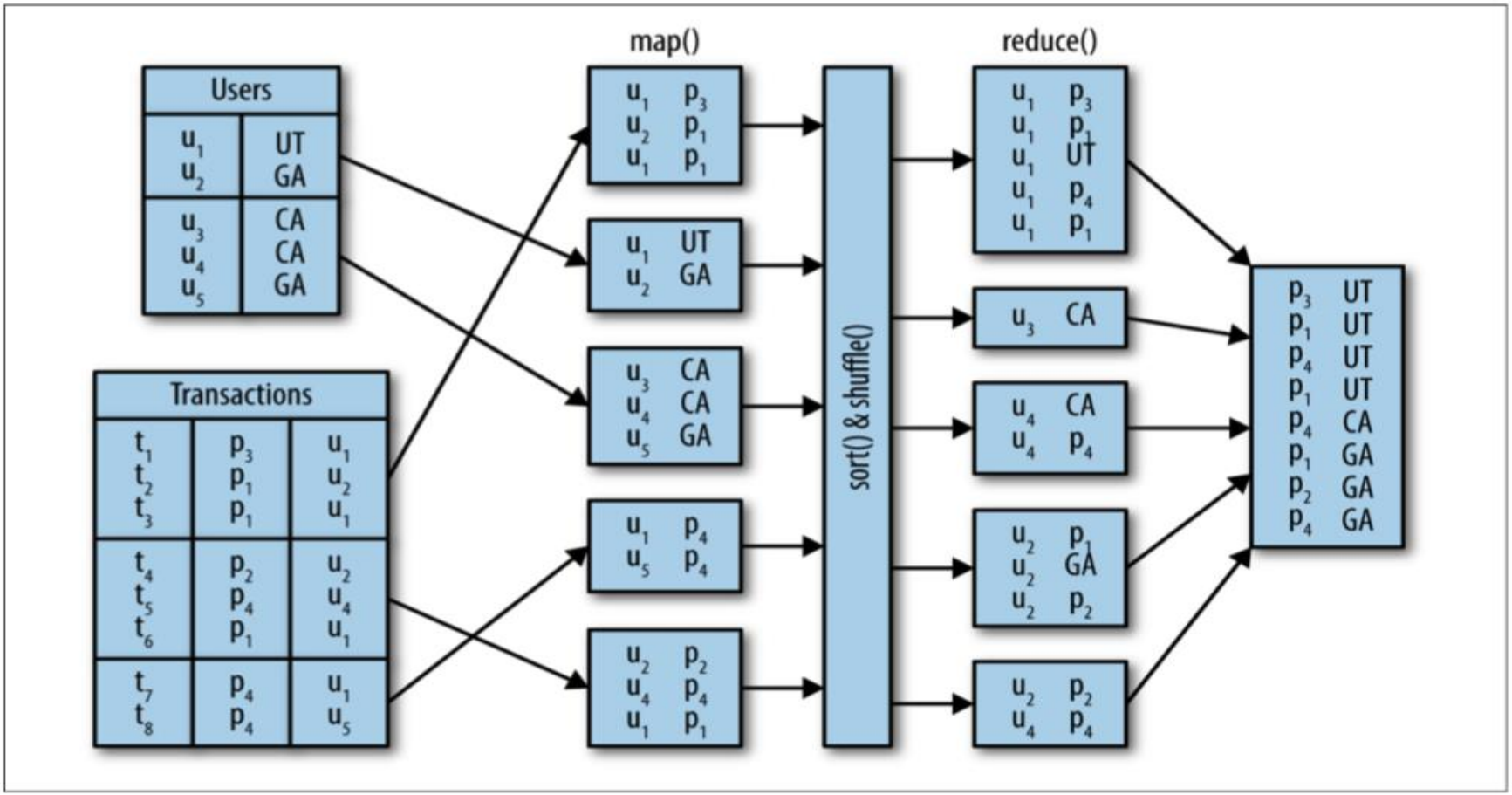
```
mysql> SELECT product_id, location_id
    -> FROM transactions LEFT OUTER JOIN users
    ->   ON transactions.user_id = users.user_id;
+------------+-------------+
| product_id | location_id |
+------------+-------------+
| p3         | UT          |
| p1         | GA          |
| p1         | UT          |
| p2         | GA          |
| p4         | CA          |
| p1         | UT          |
| p4         | UT          |
| p4         | GA          |
+------------+-------------+
8 rows in set (0.00 sec)
```

Reduce-side join steps

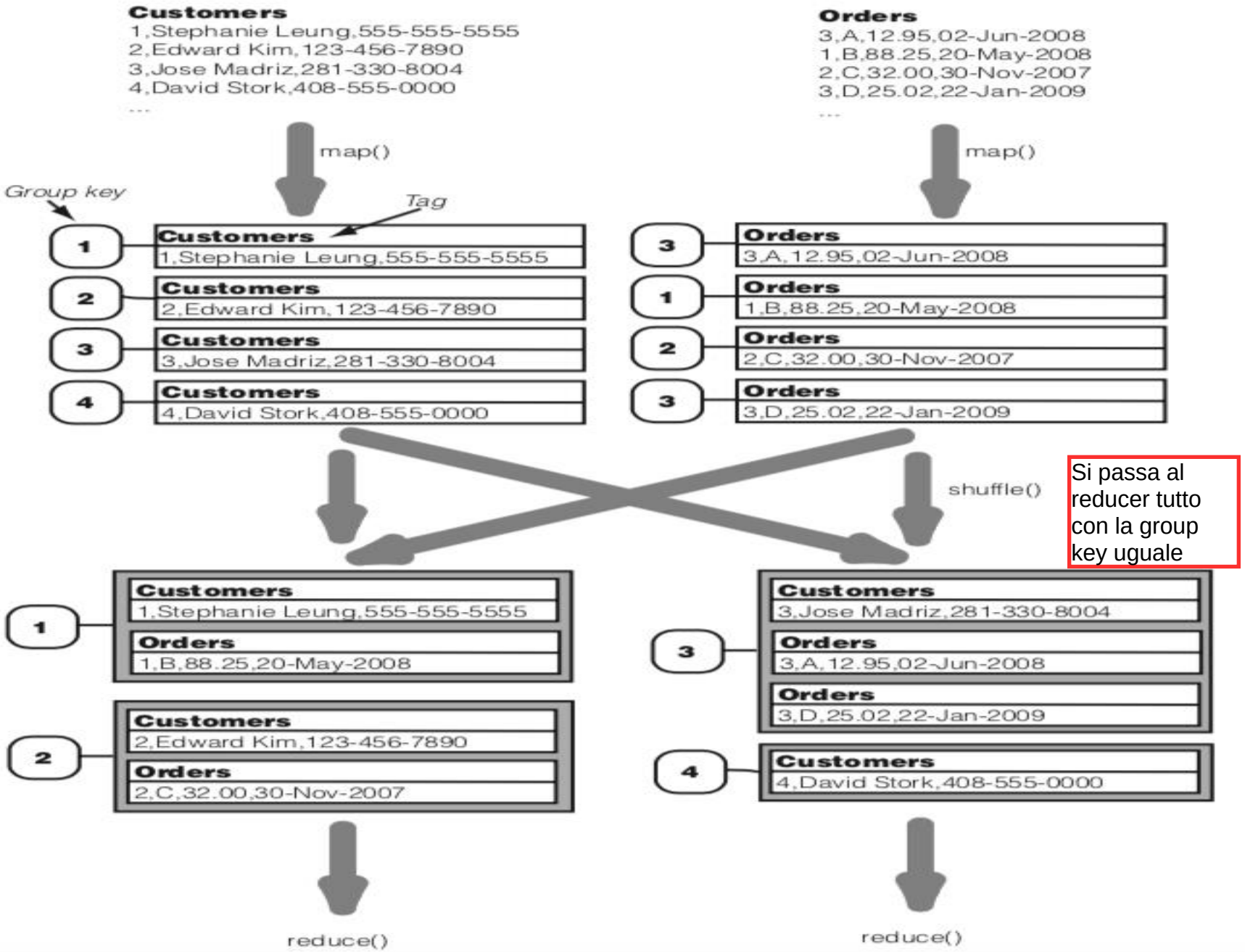
- The mapper(s):
  - Set the "group key" as key of the output
  - Select relevant columns and emit them as values
  - Maybe different mappers per different source
  - Tag the data by mentioning an origin

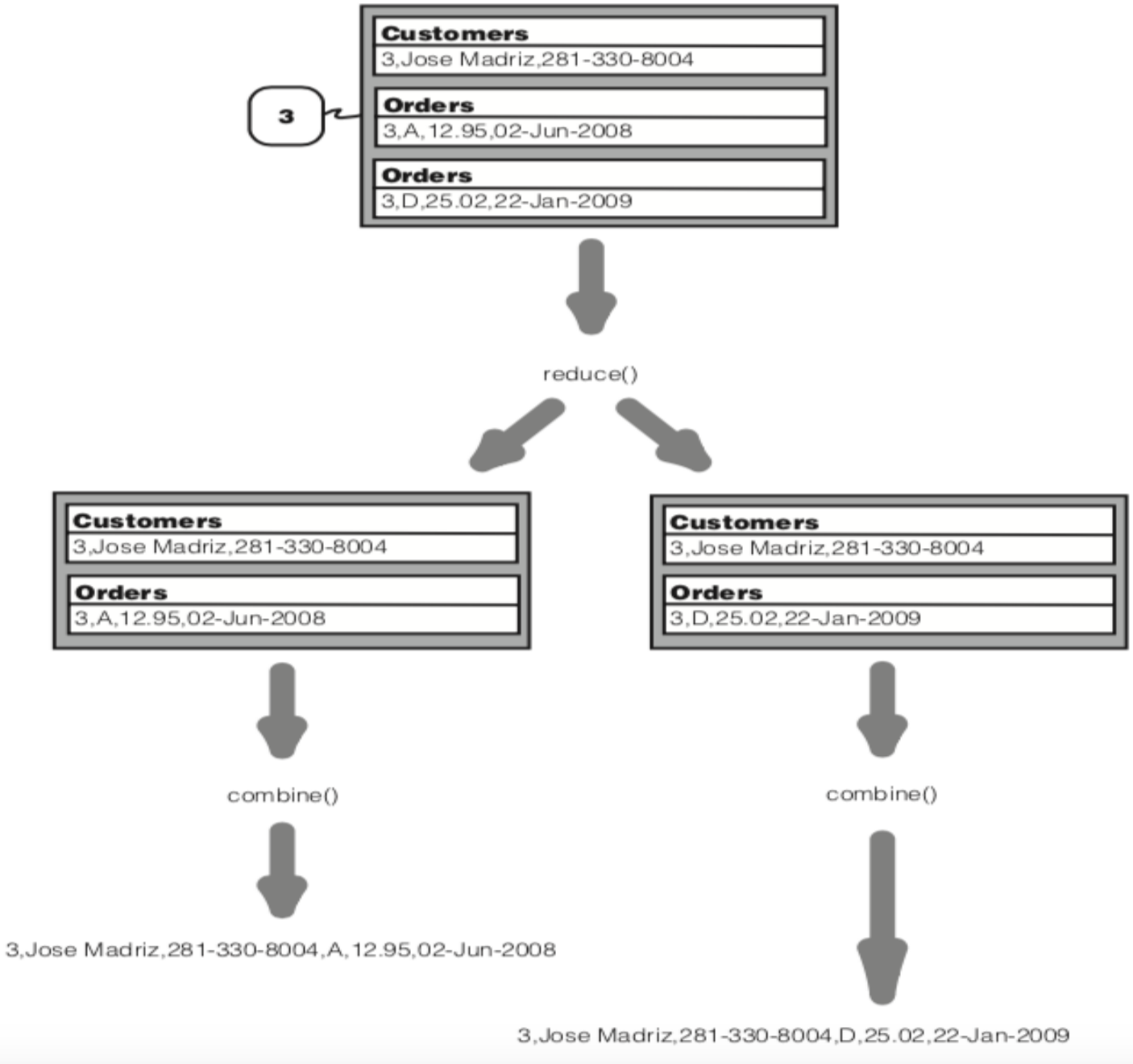
- The reducer(s)
  - Combine grouped values to generate the output
  - Perform the actual join
  - Decide what to preserve in the output "tuple"

Very simplified dataflow



Who tells the reducer(s) which value belongs to which source?


Customers
1,Stephanie Leung,555-555-5555
2,Edward Kim,123-456-7890
3,Jose Madriz,281-330-8004
4,David Stork,408-555-0000
...
Orders
3,A,12.95,02-Jun-2008
1,B,88.25,20-May-2008
2,C,32.00,30-Nov-2007
3,D,25.02,22-Jan-2009
...
map()
map()
Group key
Tag
1 Customers 1,Stephanie Leung,555-555-5555
2 Customers 2,Edward Kim,123-456-7890
3 Customers 3,Jose Madriz,281-330-8004
4 Customers 4,David Stork,408-555-0000
3 Orders 3,A,12.95,02-Jun-2008
1 Orders 1,B,88.25,20-May-2008
2 Orders 2,C,32.00,30-Nov-2007
3 Orders 3,D,25.02,22-Jan-2009
shuffle()
Si passa al reducer tutto con la group key uguale
1 Customers 1,Stephanie Leung,555-555-5555
Orders 1,B,88.25,20-May-2008
2 Customers 2,Edward Kim,123-456-7890
Orders 2,C,32.00,30-Nov-2007
3 Customers 3,Jose Madriz,281-330-8004
Orders 3,A,12.95,02-Jun-2008
Orders 3,D,25.02,22-Jan-2009
4 Customers 4,David Stork,408-555-0000
reduce()
reduce()


Customers
3,Jose Madriz,281-330-8004
3
Orders
3,A,12.95,02-Jun-2008
Orders
3,D,25.02,22-Jan-2009
reduce()
Customers
3,Jose Madriz,281-330-8004
Orders
3,A,12.95,02-Jun-2008
Customers
3,Jose Madriz,281-330-8004
Orders
3,D,25.02,22-Jan-2009
combine()
combine()
3,Jose Madriz,281-330-8004,A,12.95,02-Jun-2008
3,Jose Madriz,281-330-8004,D,25.02,22-Jan-2009

The mapper – simplified pseudocode

```
map(key, value) {
   String[] tokens = split(value, "\t");
   String id = tokens[1];
   //Let K1,..Kn be the column to project
   String cols =
   tokens[K1] + "\t" + … +"\t"+ tokens[Kn];
   //Let S be the source tag
   outputValue=Pair(S, col);
   emit(id,outputValue);
}
```

The reducer – simplified pseudocode

```
reduce(key, values) {
 for (Pair value: values)
     map.push(value.left, valur.right);
 combine(key, map);
}
combine(key, map) { // natural join
   Set customers = map.get("Customers")
   Set orders = map.get("Orders")
   for String c: customers
     for String o: orders
        emit(key, c + '\t' + o)
}
```

Map-side join (1)

- The reduce-side join technique is flexible
  - But it can also be quite inefficient
  - Joining doesn't take place until the reduce phase
  - Shuffling all data across the network
  - Often drop the majority of this data (join conditions)
- What if we eliminate the unnecessary data right in the map phase?
- Why not performing the entire join in the map phase?

Se lo faccio dai mapper posso farlo in parallelo meglio

Map-side join (2)

- The main obstacle to map-side join
  - Mappers only see the associated split
  - A record being processed by a mapper may be joined with a record not easily accessible (or even located) by that mapper
- Can we guarantee the accessibility of all the necessary data to a mapper?
  - YES - if the sources are (physically) organized in a sorted way
    - For example, the two sources of data are partitioned into the same number of partitions and the partitions are all sorted on the key and the key is the desired join key, so that each mapper can deterministically locate and retrieve all the data necessary to perform joining
    - Unfortunately, situations where we can apply this are very limited
  - YES - if one of the two sources is small enough!
    - Share one of the sources before starting the join operation

Farlo è costoso, quindi a meno che non sia già cosi la vedo dura

Map-side join (3)

- When joining big data often only one of the sources is big
  - The second source may be orders of magnitude smaller
  - Es. In a phone company's Customers data may have only tens of millions of records, but its transaction log can have billions of records containing detailed call history
- When the smaller source can fit in memory of a mapper
  - Copy the smaller source to all mappers, and
  - Perform join in the map phase
  - i.e., Replicated join in the database literature
- Hadoop has a mechanism called distributed cache
  - Designed to distribute files to all nodes in a cluster
  - When configuring a job: `DistributedCache.addCacheFile(…)`
  - Each mapper task call: `DistributedCache.getLocalCacheFiles(…)`

Map-side join – simplified pseudocode

```
//Store in memory the smaller "table"
Hash<String,String> joinData = new Hash<>();
//called once at the beginning to load the shared data
setup() {
  Path[] cacheFiles =
            DistributedCache.getLocalCacheFiles(conf);
  if (cacheFiles != null && cacheFiles.length > 0)
    joinData = readFromFile(cacheFiles[0])
}
map(key, value) {
  String joinValue=joinData.get(key);
  if(joinValue !=null ) // natural join
    emit(key,joinValue + '\t'+ value);
}
// No need for a reducer, wow!!!!
```

Reduce-side join with map-side filtering (1)

- Main limitation of map-side join
  - One of the join tables has to be small enough to fit in memory
  - If you may have a lot of data to analyze
    - You can't use replicated join no matter how you try achieve it!
- Main limitation of reduce-side join
  - the mapper only tags the data,
  - all data is shuffled across the network
  - Often most data is discarded in the reducer
- Can we combine map-side and reduce-side techniques?
  - i.e., we might push some work on the mapper side

Reduce-side join with map-side filtering (2)

- What is a semi-join?

```
SELECT s.id
FROM students s
WHERE s.id in
    (
    SELECT g.student_id
    FROM grades g
    )
```

- Basically you check for existence of values
  - At the basis of many optimizations techniques in databases

Reduce-side join with map-side filtering (3)

- Can we implement a semi-join on the mapper side?
  - How can we share one of the tables if it is so big?
  - Maybe we do not have to share the entire table
  - I would be ok to implement *a filter that discards most of the data*
- Suppose that we can (and indeed we can!)
- Mappers perform an (approximated) semi-join
  - To filter out useless data
  - Avoid (as much as possible) transmitting tuples that will be discarded
- Reducers perform the usual join
  - On less data!
  - The best of both words!

L'idea è: quello che posso pre-filtrare, filtro tutto brother

Bloom Filters (definition)

- A space-efficient probabilistic data structure
  - Conceived by Burton Howard Bloom in 1970
  - Approximate test of set membership
    - False positive matches are possible, but false negatives are not....
- A bloom filter works as a "probabilistic set"
  - Add elements (cannot remove elements)
  - The more elements that are added to the set
    - the larger the probability of false positives
  - The larger the structure (i.e., number of bits) of the filter
    - the smaller the error

Reduce-side join with map-side filtering (4)

1. Create bloom filters
2. Share it/them with the mappers
3. Implement approximated semi-join on the mapper side
   - Filter out as much as possible tuples that will not match
4. Perform the "true" join on the reducer

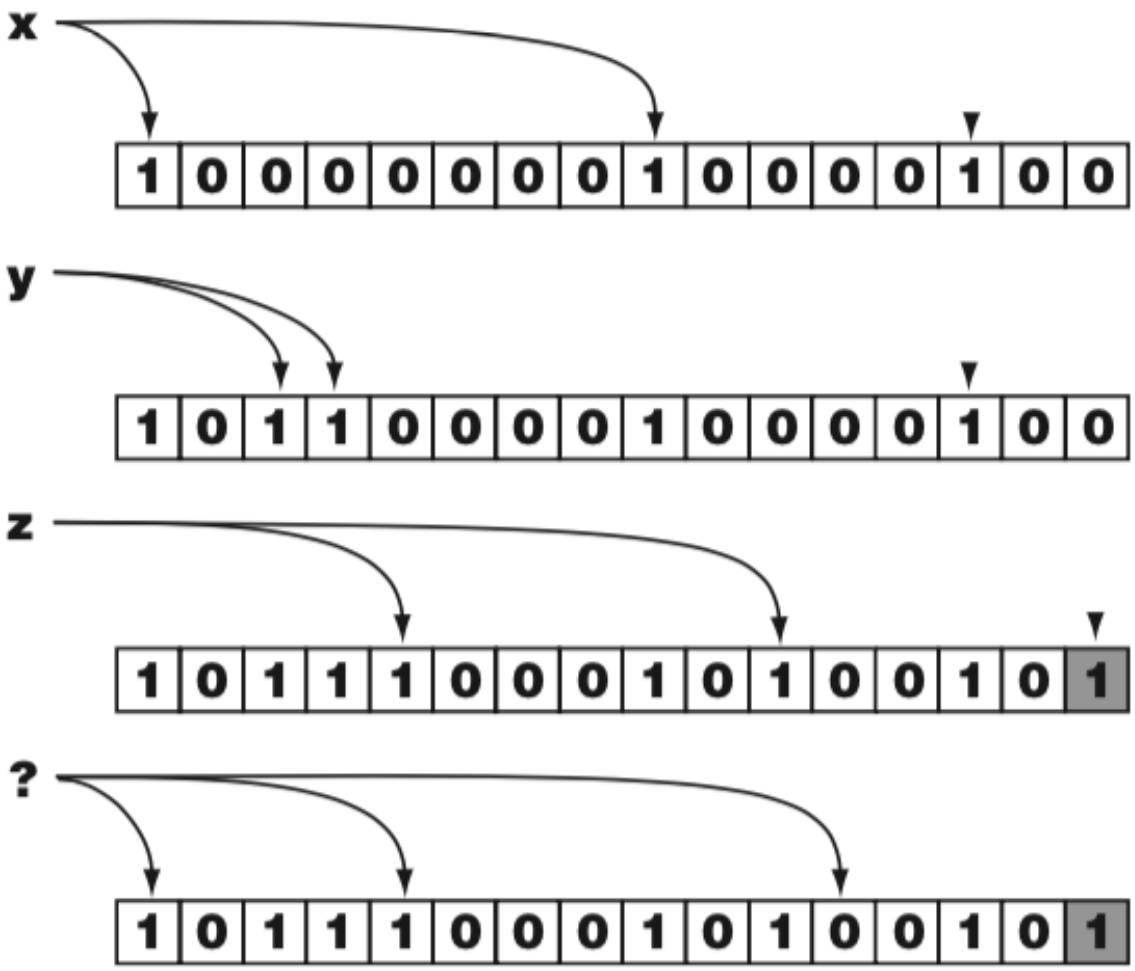
Why it works?

- A simple join example between two relations/tables
  - Join R(a, b) and S(a, c) on a common field a
  - size(R) = 1,000,000,000 (larger data set)
  - size(S) = 10,000,000 (smaller data set)
  - Basic join checks 10,000,000,000,000,000 records!!!!!
- Use a Bloom filter on relation S (the smaller data set) and filter data structure on relation R
  - Possibly eliminate the unneeded records from R
    - *It might reduce the set to 20,000,000*
  - The join becomes much faster and efficient!

Bloom filters definition

- Let B be a bit array of size m (m > 1), initialized with 0s
- Let {H1, H2, ..., Hk} be a set of k hash functions,
  if Hi(xj) = a, then set B[a] = 1
  - E.g., use SHA1 and MD5 as hash functions
- To check if x ∈ S, check B all k values Hi(x) must be 1
  - It is possible to have a false positive
    i.e., all k values are 1, but x is not in S

Bloom filter idealized



Array B:

initialized:

index	0	1	2	3	4	5	6	7	8	9
value	0	0	0	0	0	0	0	0	0	0

insert element a, H(a) = (2, 5, 6)

index	0	1	2	3	4	5	6	7	8	9
value	0	0	1	0	0	1	1	0	0	0

insert element b, H(b) = (1, 5, 8)

index	0	1	2	3	4	5	6	7	8	9
value	0	1	1	0	0	1	1	0	1	0

query element c

H(c) = (5, 8, 9) => c is not a member (since B[9]=0)

query element d

H(d) = (2, 5, 8) => d is a member (False positive)

query element e

H(e) = (1, 2, 6) => e is a member (False positive)

query element f

H(f) = (2, 5, 6) => f is a member (Positive)

Bloom filters false positive probability

- The probability of false positives is:

$$\left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k$$

Maggiore il dataset, maggiore è la possibilità di beccare un falso positivo.
Dalla formula si vede che più è grande n, più si avvicina a 1 il risultato

- What is the optimal number of hash functions?
- Given m (number of bits selected for Bloom filter) and n (size of big data set), the value of k (number of hash functions) that minimizes the probability of false positives

$$k = \frac{m}{n} ln(2) \qquad m = -\frac{nln(p)}{(ln(2))^2}$$

- Probability of getting 1 increases steadily with elements

$$1 - \left(1 - \frac{1}{m}\right)^{kn} \approx 1 - e^{-\frac{kn}{m}}$$

Bloom filter parameters

- For example
  - A set containing ten million URLs (n=10,000,000)
  - Storing the raw URLs (average URL length 100 bytes)
    - Would use 1 GB!
  - Allocating 8 bits per URL (m/n=8)
    - Requires a 10 MB Bloom filter
    - False positive rate of ~ 2%.
  - Allocating 10 bits per URL
    - Requires a 12.5 MB Bloom filter
    - False positive rate of only 0.8%!!

Available implementations (1)

org.apache.hadoop.util.bloom

Class BloomFilter

java.lang.Object
    org.apache.hadoop.util.bloom.Filter
        org.apache.hadoop.util.bloom.BloomFilter

All Implemented Interfaces:

`Writable`

Direct Known Subclasses:

`RetouchedBloomFilter`

```
@InterfaceAudience.Public
 @InterfaceStability.Stable
public class BloomFilter
extends org.apache.hadoop.util.bloom.Filter
```

Implements a *Bloom filter*, as defined by Bloom in 1970.

Constructor Summary

Constructors

Constructor and Description
`BloomFilter()` Default constructor - use with readFields
`BloomFilter(int vectorSize, int nbHash, int hashType)` Constructor

Modifier and Type	Method and Description
`void`	**`add`**`(org.apache.hadoop.util.bloom.Key key)` Adds a key to *this* filter.
`void`	**`and`**`(org.apache.hadoop.util.bloom.Filter filter)` Peforms a logical AND between *this* filter and a specified filter.
`int`	**`getVectorSize`**`()`
`boolean`	**`membershipTest`**`(org.apache.hadoop.util.bloom.Key key)` Determines wether a specified key belongs to *this* filter.
`void`	**`not`**`()` Performs a logical NOT on *this* filter.
`void`	**`or`**`(org.apache.hadoop.util.bloom.Filter filter)` Peforms a logical OR between *this* filter and a specified filter.
`void`	**`readFields`**`(DataInput in)` Deserialize the fields of this object from `in`.
`String`	**`toString`**`()`
`void`	**`write`**`(DataOutput out)` Serialize the fields of this object to `out`.
`void`	**`xor`**`(org.apache.hadoop.util.bloom.Filter filter)` Peforms a logical XOR between *this* filter and a specified filter.

Available implementations (2)

- The Guava library
  - First, you define your basic type (Person, in this case) that will be used in the Bloom filter
  - Then you define a Funnel of the basic type on which you want to use the Bloom filter.
    - A Funnel describes how to decompose a particular object type into primitive field values
  - The library provides the several hash functions
- See
  - https://github.com/google/guava

Secondary sort

- Sort the values passed to each reducer
  - Value-to-key conversion
  - Useful to implement joins with "group by"
  - Useful in applications (such as time series data) in which you want to sort your reducer data

Secondary sort by example

- Consider the temperature data from a scientific experiment
  - Data columns: year, month, day, and daily temperature, eg.:

```
2012, 01, 01, 5
2012, 01, 02, 45
...
2001, 11, 01, 46
2001, 11, 02, 47
...
```

- **Goal:** *output the temperature for every year-month with the values sorted in ascending order, e.g.:*

```
2012-01: 5, 10, 35, 45, ...
2001-11: 40, 46, 47, 48, ...
```

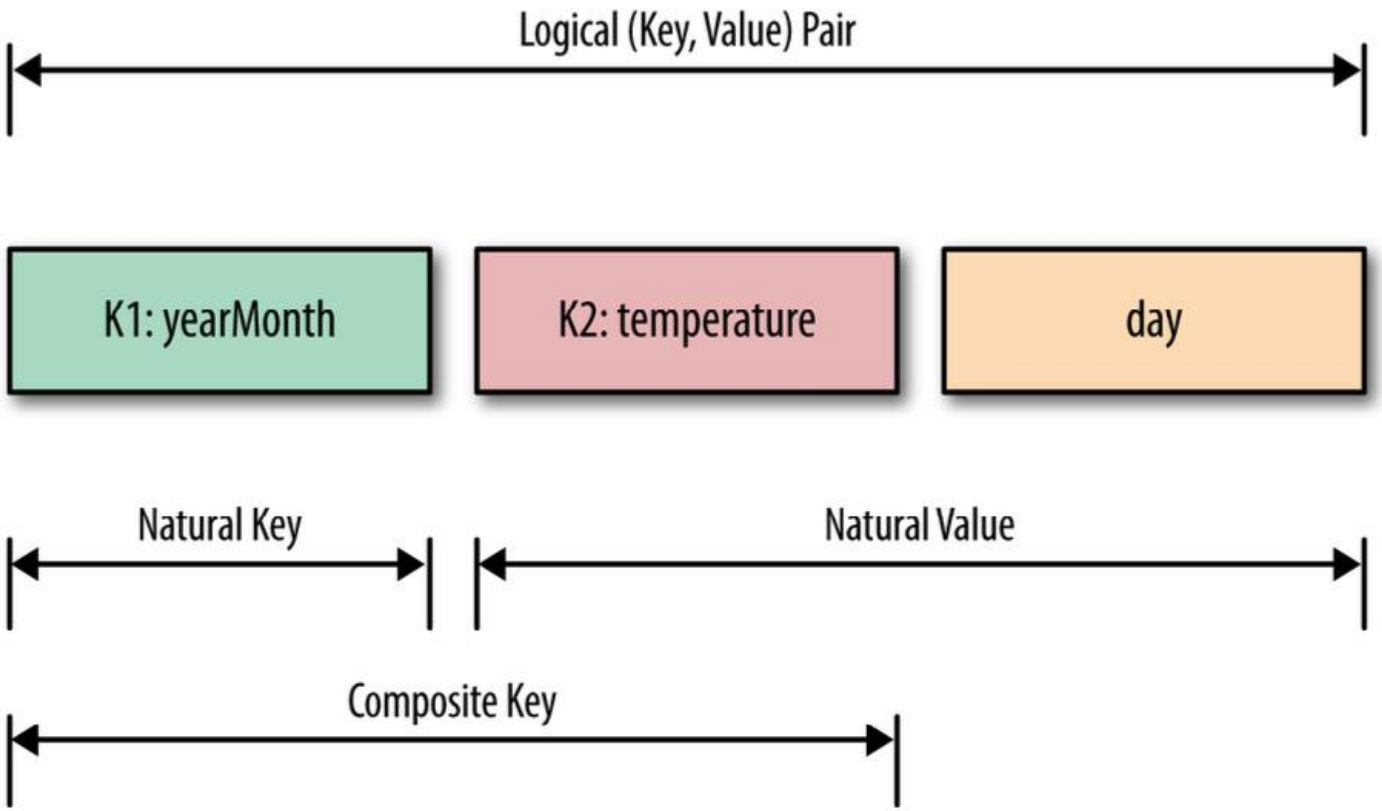
Possible solutions

- **In-reducer sort**
  - The reducer reads and buffers all to sort the values for a given key
  - This approach will not scale
    - The reducer easily runs out of memory
  - Work well if the number of values is small enough

- **Offload the sorting to the MapReduce framework**
  - Creating a *composite key* (natural key + some value)
  - Provide custom comparators and partitioners
    - E.g., partition the mapper's output by the natural key
  - Exploit map-reduce framework distributed sort!

Sorta per valore, splitti per chiave per ogni mapper

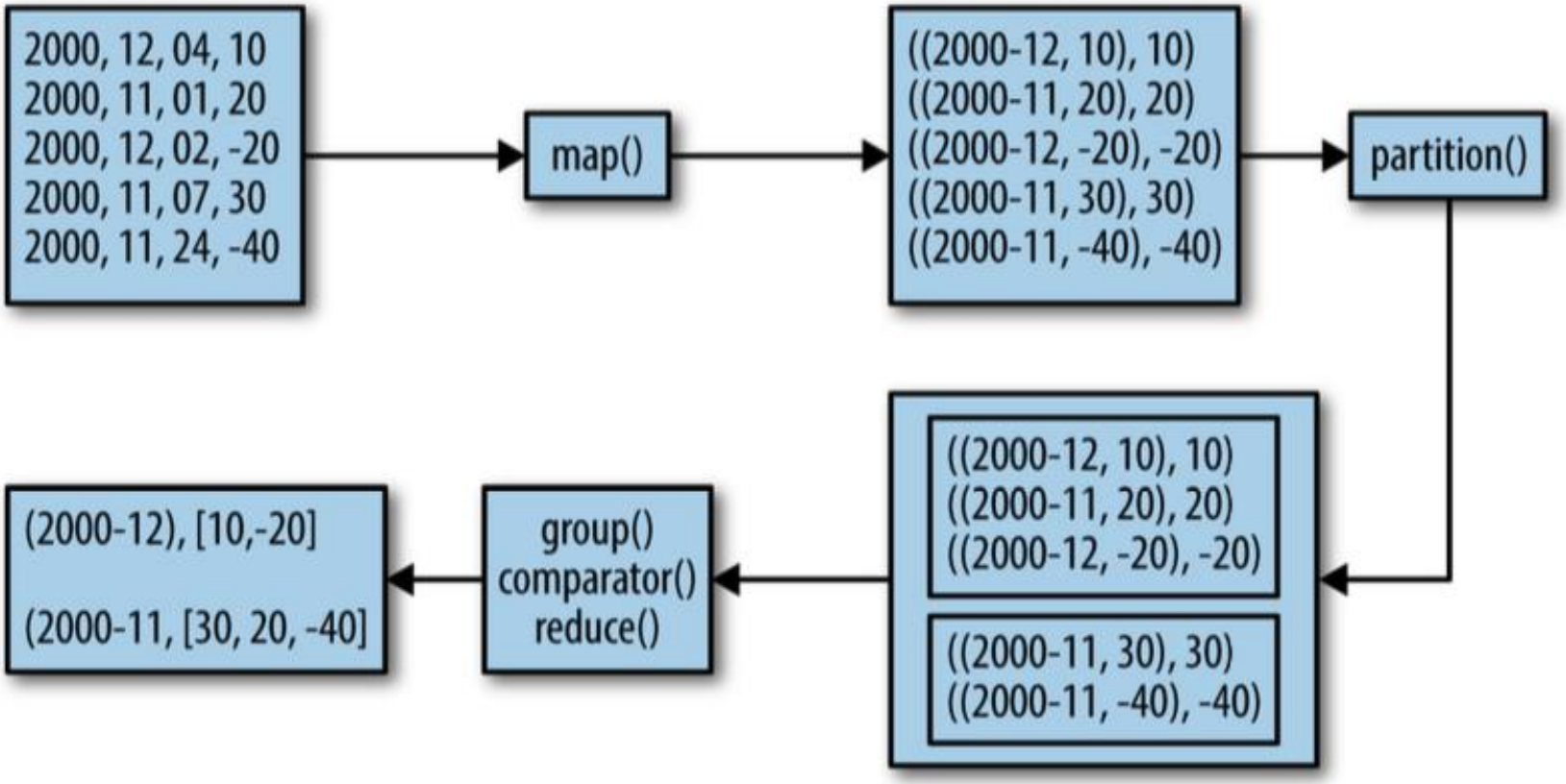
Composite vs natural key



Secondary sort – flow (1)

- The execution flow:
  - The mappers create (K,V) pairs
    - K is a composite key of (year,month,tempera ture) and
    - V is temperature
    - The (year,month) part of the composite key is the natural key
  - The partitioner class enables sends all natural keys to the same reducer
  - The grouping comparator class enables temperatures to arrive sorted at reducers
- The Secondary Sort design
  - “Scale out” no matter how many reducer values we want to sort
  - Uses MapReduce’s framework for sorting the reducers’ values rather than collecting them all and then sorting them in memory

Secondary sort – flow (2)



Secondary sort - ingredients

- Create a specific datatype for the composite key
  - `DateTemperaturePair class`
    - `implements Writable, WritableComparable<DateTemperaturePair>`
- Custom Partitioner
  - Decides which mapper's output goes to which reducer
    - `DateTemperaturePartitioner`
      - `extends Partitioner<DateTemperaturePair, Text>`
    - `job.setPartitionerClass(TemperaturePartitioner.class)`
- Grouping comparator
  - Controls which keys are grouped together for a single reduce() call
    - `DateTemperatureGroupingComparator class`
      - `extends WritableComparator`
    - `job.setGroupingComparatorClass (YearMonthGroupingComparator.class)`

```
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.io.WritableComparable;
...
public class DateTemperaturePair

implements Writable, WritableComparable<DateTemperaturePair> {

 private Text yearMonth = new Text();                    // natural key
 private Text day = new Text();
 private IntWritable temperature = new IntWritable(); // secondary key

 ...

 @Override
 /**
  * This comparator controls the sort order of the keys.
  */
 public int compareTo(DateTemperaturePair pair) {
     int compareValue = this.yearMonth.compareTo(pair.getYearMonth());
     if (compareValue == 0) {
         compareValue = temperature.compareTo(pair.getTemperature());
     }
     //return compareValue;     // sort ascending
     return -1*compareValue;   // sort descending
 }
 ...
}
```

```
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Partitioner;

public class DateTemperaturePartitioner
   extends Partitioner<DateTemperaturePair, Text> {

    @Override
    public int getPartition(DateTemperaturePair pair,
                            Text text,
                            int numberOfPartitions) {
        // make sure that partitions are non-negative
        return Math.abs(pair.getYearMonth().hashCode() % numberOfPartitions);
     }
}
```

```java
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.io.WritableComparator;

public class DateTemperatureGroupingComparator
    extends WritableComparator {

    public DateTemperatureGroupingComparator() {
        super(DateTemperaturePair.class, true);
    }

    @Override
    /**
     * This comparator controls which keys are grouped
     * together into a single call to the reduce() method
     */
    public int compare(WritableComparable wc1, WritableComparable wc2) {
        DateTemperaturePair pair = (DateTemperaturePair) wc1;
        DateTemperaturePair pair2 = (DateTemperaturePair) wc2;
        return pair.getYearMonth().compareTo(pair2.getYearMonth());
    }
}
```

```
map(key, value) {
    String[] tokens = value.split(",");
    // YYYY = tokens[0]
    // MM = tokens[1]
    // DD = tokens[2]
    // temperature = tokens[3]
    String yearMonth = tokens[0] + tokens[1];
    String day = tokens[2];
    int temperature = Integer.parseInt(tokens[3]);
    // prepare reducer key
    DateTemperaturePair reducerKey = new DateTemperaturePair();
    reducerKey.setYearMonth(yearMonth);
    reducerKey.setDay(day);
    reducerKey.setTemperature(temperature); // inject value into key
    // send it to reducer
    emit(reducerKey, temperature);
}
reduce(key, value) {
    StringBuilder sortedTemperatureList = new StringBuilder();
    for (Integer temperature : value) {
       sortedTemperatureList.append(temperature);
       sortedTemperatureList.append(",");
    }
    emit(key, sortedTemperatureList);
}
```

```
public class SecondarySortDriver extends Configured implements Tool {
    public int run(String[] args) throws Exception {
        Configuration conf = getConf();
        Job job = new Job(conf);
        job.setJarByClass(SecondarySortDriver.class);
        job.setJobName("SecondarySortDriver");

        Path inputPath = new Path(args[0]);
        Path outputPath = new Path(args[1]);

        FileInputFormat.setInputPaths(job, inputPath);
        FileOutputFormat.setOutputPath(job, outputPath);

        job.setOutputKeyClass(TemperaturePair.class);
        job.setOutputValueClass(NullWritable.class);

        job.setMapperClass(SecondarySortingTemperatureMapper.class);
        job.setReducerClass(SecondarySortingTemperatureReducer.class);
        job.setPartitionerClass(TemperaturePartitioner.class);
        job.setGroupingComparatorClass(YearMonthGroupingComparator.class);

        boolean status = job.waitForCompletion(true);
        theLogger.info("run(): status="+status);
        return status ? 0 : 1;
    }
```

LET'S START

(open, configure and run examples on eclipse)