MapReduce in Spark

Krzysztof Dembczyński

Intelligent Decision Support Systems Laboratory (IDSS)
Poznań University of Technology, Poland

Software Development Technologies
Master studies, first semester
Academic year 2017/18 (winter course)

Review of the previous lectures

- Mining of massive datasets.
- Evolution of database systems.
- Dimensional modeling.
- ETL and OLAP systems:
  - Extraction, transformation, load.
  - ROLAP, MOLAP, HOLAP.
  - Challenges in OLAP systems: a huge number of possible aggregations to compute.

Outline

Outline

Motivation

- Traditional DBMS vs. NoSQL

Motivation

- Traditional DBMS vs. NoSQL
- New emerging applications: search engines, social networks, online shopping, online advertising, recommender systems, etc.

Motivation

- Traditional DBMS vs. NoSQL
- New emerging applications: search engines, social networks, online shopping, online advertising, recommender systems, etc.
- New computational challenges: WordCount, PageRank, etc.

Motivation

- Traditional DBMS vs. NoSQL
- New emerging applications: search engines, social networks, online shopping, online advertising, recommender systems, etc.
- New computational challenges: WordCount, PageRank, etc.
- New ideas:

Motivation

- Traditional DBMS vs. NoSQL
- New emerging applications: search engines, social networks, online shopping, online advertising, recommender systems, etc.
- New computational challenges: WordCount, PageRank, etc.
- New ideas:
  - Scaling-out instead of scaling-up

Motivation

- Traditional DBMS vs. NoSQL
- New emerging applications: search engines, social networks, online shopping, online advertising, recommender systems, etc.
- New computational challenges: WordCount, PageRank, etc.
- New ideas:
  - Scaling-out instead of scaling-up
  - Move-code-to-data

Outline

MapReduce-based systems

- Accessible – run on large clusters of commodity machines or on cloud computing services such as Amazon's Elastic Compute Cloud (EC2).

MapReduce-based systems

- Accessible – run on large clusters of commodity machines or on cloud computing services such as Amazon's Elastic Compute Cloud (EC2).
- Robust – are intended to run on commodity hardware; desinged with the assumption of frequent hardware malfunctions; they can gracefully handle most such failures.

MapReduce-based systems

- Accessible – run on large clusters of commodity machines or on cloud computing services such as Amazon's Elastic Compute Cloud (EC2).
- Robust – are intended to run on commodity hardware; desinged with the assumption of frequent hardware malfunctions; they can gracefully handle most such failures.
- Scalable – scales linearly to handle larger data by adding more nodes to the cluster.

MapReduce-based systems

- Accessible – run on large clusters of commodity machines or on cloud computing services such as Amazon's Elastic Compute Cloud (EC2).
- Robust – are intended to run on commodity hardware; desinged with the assumption of frequent hardware malfunctions; they can gracefully handle most such failures.
- Scalable – scales linearly to handle larger data by adding more nodes to the cluster.
- Simple – allow users to quickly write efficient parallel code.

MapReduce: Two simple procedures

- Word count: A basic operation for every search engine.
- Matrix-vector multiplication: A fundamental step in many algorithms, for example, in PageRank.

MapReduce: Two simple procedures

- Word count: A basic operation for every search engine.
- Matrix-vector multiplication: A fundamental step in many algorithms, for example, in PageRank.

- How to implement these procedures for efficient execution in a distributed system?

MapReduce: Two simple procedures

- Word count: A basic operation for every search engine.
- Matrix-vector multiplication: A fundamental step in many algorithms, for example, in PageRank.

- How to implement these procedures for efficient execution in a distributed system?
- How much can we gain by such implementation?

MapReduce: Two simple procedures

- Word count: A basic operation for every search engine.
- Matrix-vector multiplication: A fundamental step in many algorithms, for example, in PageRank.

- How to implement these procedures for efficient execution in a distributed system?
- How much can we gain by such implementation?
- Let us focus on the word count problem ...

Word count

- Count the number of times each word occurs in a set of documents:

Do as I say, not as I do.

Word	Count
as	2
do	2
i	2
not	1
say	1

Word count

- Let us write the procedure in pseudo-code for a single machine:

Word count

- Let us write the procedure in pseudo-code for a single machine:

```
define wordCount as Multiset;

for each document in documentSet {
    T = tokenize(document);

    for each token in T {
        wordCount[token]++;
    }
}

display(wordCount);
```

Word count

- Let us write the procedure in pseudo-code for many machines:

Word count

- Let us write the procedure in pseudo-code for many machines:
  - First step:

```
define wordCount as Multiset;

for each document in documentSubset {
    T = tokenize(document);
    for each token in T {
        wordCount[token]++;
    }
}

sendToSecondPhase(wordCount);
```

Word count

- Let us write the procedure in pseudo-code for many machines:
  - ▶ First step:

```
define wordCount as Multiset;

for each document in documentSubset {
    T = tokenize(document);
    for each token in T {
        wordCount[token]++;
    }
}

sendToSecondPhase(wordCount);
```

  - ▶ Second step:

```
define totalWordCount as Multiset;

for each wordCount received from firstPhase {
    multisetAdd(totalWordCount, wordCount);
}
```

Word count

- To make the procedure work properly across a cluster of distributed machines, we need to add a number of functionalities:
  - ▸ Store files over many processing machines (of phase one).
  - ▸ Write a disk-based hash table permitting processing without being limited by RAM capacity.
  - ▸ Partition the intermediate data (that is, wordCount) from phase one.
  - ▸ Shuffle the partitions to the appropriate machines in phase two.
  - ▸ Ensure fault tolerance.

MapReduce

- MapReduce programs are executed in two main phases, called mapping and reducing:

MapReduce

- MapReduce programs are executed in two main phases, called mapping and reducing:
  - ▸ Map: the map function is written to convert input elements to key-value pairs.

MapReduce

- MapReduce programs are executed in two main phases, called mapping and reducing:
  - ▶ Map: the map function is written to convert input elements to key-value pairs.
  - ▶ Reduce: the reduce function is written to take pairs consisting of a key and its list of associated values and combine those values in some way.
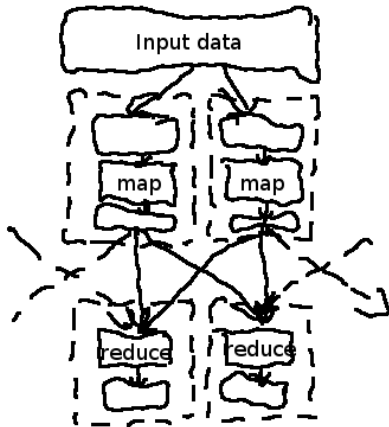
MapReduce

- The complete data flow:

	Input	Output
map	`(<k1, v1>)`	`list(<k2, v2>)`
reduce	`(<k2, list(<v2>)`	`list(<k3, v3>)`

MapReduce

Figure: The complete data flow

MapReduce

- The complete data flow:

MapReduce

- The complete data flow:
  - The input is structured as a list of key-value pairs: `list(<k1,v1>)`.

MapReduce

- The complete data flow:
  - ▸ The input is structured as a list of key-value pairs: `list(<k1,v1>)`.
  - ▸ The list of key-value pairs is broken up and each individual key-value pair, `<k1,v1>`, is processed by calling the map function of the mapper (the key `k1` is often ignored by the mapper).

MapReduce

- The complete data flow:
  - ▸ The input is structured as a list of key-value pairs: `list(<k1,v1>)`.
  - ▸ The list of key-value pairs is broken up and each individual key-value pair, `<k1,v1>`, is processed by calling the map function of the mapper (the key `k1` is often ignored by the mapper).
  - ▸ The mapper transforms each `<k1,v1>` pair into a list of `<k2,v2>` pairs.

MapReduce

- The complete data flow:
    - ▶ The input is structured as a list of key-value pairs: `list(<k1,v1>)`.
    - ▶ The list of key-value pairs is broken up and each individual key-value pair, `<k1,v1>`, is processed by calling the map function of the mapper (the key `k1` is often ignored by the mapper).
    - ▶ The mapper transforms each `<k1,v1>` pair into a list of `<k2,v2>` pairs.
    - ▶ The key-value pairs are processed in arbitrary order.

MapReduce

- The complete data flow:
  - ▶ The input is structured as a list of key-value pairs: `list(<k1,v1>)`.
  - ▶ The list of key-value pairs is broken up and each individual key-value pair, `<k1,v1>`, is processed by calling the map function of the mapper (the key `k1` is often ignored by the mapper).
  - ▶ The mapper transforms each `<k1,v1>` pair into a list of `<k2,v2>` pairs.
  - ▶ The key-value pairs are processed in arbitrary order.
  - ▶ The output of all the mappers are (conceptually) aggregated into one giant list of `<k2,v2>` pairs. All pairs sharing the same `k2` are grouped together into a new aggregated key-value pair: `<k2,list(v2)>`.

MapReduce

- The complete data flow:
  - ▸ The input is structured as a list of key-value pairs: `list(<k1,v1>)`.
  - ▸ The list of key-value pairs is broken up and each individual key-value pair, `<k1,v1>`, is processed by calling the map function of the mapper (the key `k1` is often ignored by the mapper).
  - ▸ The mapper transforms each `<k1,v1>` pair into a list of `<k2,v2>` pairs.
  - ▸ The key-value pairs are processed in arbitrary order.
  - ▸ The output of all the mappers are (conceptually) aggregated into one giant list of `<k2,v2>` pairs. All pairs sharing the same `k2` are grouped together into a new aggregated key-value pair: `<k2,list(v2)>`.
  - ▸ The framework asks the reducer to process each one of these aggregated key-value pairs individually.

WordCount in MapReduce

- Map:
  - For a pair <k1,document> produce a sequence of pairs <token,1>, where token is a token/word found in the document.

```
map( String filename , String document ) {
    List<String> T = tokenize(document);

    for each token in T {
      emit (( String ) token , ( Integer ) 1);
    }

}
```

WordCount in MapReduce

- Reduce
  - For a pair <word, list(1, 1, ..., 1)> sum up all ones appearing in the list and return <word, sum>, where sum is the sum of ones.

```
reduce(String token, List<Integer> values) {
    Integer sum = 0;

    for each value in values {
        sum = sum + value;
    }

    emit ((String)token, (Integer) sum);
}
```

Combiner and partitioner

- Beside map and reduce there are two other important elements that can be implemented within the MapReduce framework to control the data flow.

Combiner and partitioner

- Beside map and reduce there are two other important elements that can be implemented within the MapReduce framework to control the data flow.

- **Combiner** – perform local aggregation (the reduce step) on the map node.

Combiner and partitioner

- Beside map and reduce there are two other important elements that can be implemented within the MapReduce framework to control the data flow.

- **Combiner** – perform local aggregation (the reduce step) on the map node.

- **Partitioner** – divide the key space of the map output and assign the key-value pairs to reducers.

Outline

Spark

- Spark is a fast and general-purpose cluster computing system.

Spark

- Spark is a fast and general-purpose cluster computing system.
- It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs.

Spark

- Spark is a fast and general-purpose cluster computing system.
- It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs.
- It also supports a rich set of higher-level tools including:

Spark

- Spark is a fast and general-purpose cluster computing system.
- It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs.
- It also supports a rich set of higher-level tools including:
  - Spark SQL for SQL and structured data processing,

Spark

- Spark is a fast and general-purpose cluster computing system.
- It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs.
- It also supports a rich set of higher-level tools including:
  - Spark SQL for SQL and structured data processing,
  - MLlib for machine learning,

Spark

- Spark is a fast and general-purpose cluster computing system.
- It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs.
- It also supports a rich set of higher-level tools including:
  - ▶ Spark SQL for SQL and structured data processing,
  - ▶ MLlib for machine learning,
  - ▶ GraphX for graph processing,

Spark

- Spark is a fast and general-purpose cluster computing system.
- It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs.
- It also supports a rich set of higher-level tools including:
    - Spark SQL for SQL and structured data processing,
    - MLlib for machine learning,
    - GraphX for graph processing,
    - and Spark Streaming.

Spark

- Spark is a fast and general-purpose cluster computing system.
- It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs.
- It also supports a rich set of higher-level tools including:
  - Spark SQL for SQL and structured data processing,
  - MLlib for machine learning,
  - GraphX for graph processing,
  - and Spark Streaming.
- For more check `https://spark.apache.org/`

Spark

- Spark uses Hadoop which is a popular open-source implementation of MapReduce.

Spark

- Spark uses Hadoop which is a popular open-source implementation of MapReduce.
- Hadoop works in a master/slave architecture for both distributed storage and distributed computation.

Spark

- Spark uses Hadoop which is a popular open-source implementation of MapReduce.
- Hadoop works in a master/slave architecture for both distributed storage and distributed computation.
- Hadoop Distributed File System (HDFS) is responsible for distributed storage.

Installation of Spark

- Download Spark from

Installation of Spark

- Download Spark from
    `http://spark.apache.org/downloads.html`

Installation of Spark

- Download Spark from
    http://spark.apache.org/downloads.html
- Untar the spark archive:

Installation of Spark

- Download Spark from
  `http://spark.apache.org/downloads.html`
- Untar the spark archive:
  `tar xvfz spark-2.2.0-bin-hadoop2.7.tar`

Installation of Spark

- Download Spark from
    http://spark.apache.org/downloads.html
- Untar the spark archive:
    tar xvfz spark-2.2.0-bin-hadoop2.7.tar
- To play with Spark there is no need to install HDFS ...

Installation of Spark

- Download Spark from
    http://spark.apache.org/downloads.html
- Untar the spark archive:
    tar xvfz spark-2.2.0-bin-hadoop2.7.tar
- To play with Spark there is no need to install HDFS ...
- But, you can try to play around with HDFS.

- Create new directories:

HDFS

- Create new directories:
```
hdfs dfs -mkdir /user
```

HDFS

- Create new directories:

```
hdfs dfs -mkdir /user
hdfs dfs -mkdir /user/mynane
```

HDFS

- Create new directories:

  ```
  hdfs dfs -mkdir /user
  hdfs dfs -mkdir /user/mynane
  ```

- Copy the input files into the distributed filesystem:

HDFS

- Create new directories:
  ```
  hdfs dfs -mkdir /user
  hdfs dfs -mkdir /user/mynane
  ```
- Copy the input files into the distributed filesystem:
  ```
  hdfs dfs -put data.txt /user/myname/data.txt
  ```

HDFS

- Create new directories:
  ```
  hdfs dfs -mkdir /user
  hdfs dfs -mkdir /user/mynane
  ```
- Copy the input files into the distributed filesystem:
  ```
  hdfs dfs -put data.txt /user/myname/data.txt
  ```
- View the files in the distributed filesystem:

HDFS

- Create new directories:
  ```
  hdfs dfs -mkdir /user
  hdfs dfs -mkdir /user/mynane
  ```
- Copy the input files into the distributed filesystem:
  ```
  hdfs dfs -put data.txt /user/myname/data.txt
  ```
- View the files in the distributed filesystem:
  ```
  hdfs dfs -ls /user/myname/
  ```

HDFS

- Create new directories:
  ```
  hdfs dfs -mkdir /user
  hdfs dfs -mkdir /user/mynane
  ```
- Copy the input files into the distributed filesystem:
  ```
  hdfs dfs -put data.txt /user/myname/data.txt
  ```
- View the files in the distributed filesystem:
  ```
  hdfs dfs -ls /user/myname/
  hdfs dfs -cat /user/myname/data.txt
  ```

WordCount in Hadoop

```java
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;

public class WordCount {

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

    private final static IntWritable one = new IntWritable(1);
    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);
      }
    }
  }
  (...)
```

WordCount in Hadoop

```java
(...)
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);
    }
}

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);
}
}
```

WordCount in Spark

- The same code is much simpler in Spark
- To run the Spark shell type: `./bin/spark-shell`
- The code

```scala
val textFile = sc.textFile("~/data/all-bible.txt")
val counts = (textFile.flatMap(line => line.split(" "))
              .map(word => (word, 1))
              .reduceByKey(_ + _))
counts.saveAsTextFile("~/data/all-bible-counts.txt")
```

Alternatively:

```scala
val wordCounts = textFile.flatMap(line => line.split(" ")).groupByKey(identity).
    count()
```

Outline

Algorithms in Map-Reduce

- How to implement fundamental algorithms in MapReduce?
  - Matrix-vector multiplication.
  - Relational-Algebra Operations.
  - Matrix multiplication.

Matrix-vector Multiplication

- Let $\boldsymbol{A}$ to be large $n \times m$ matrix, and $\boldsymbol{x}$ a long vector of size m.
- The matrix-vector multiplication is defined as:

Matrix-vector Multiplication

- Let A to be large $n \times m$ matrix, and x a long vector of size m.
- The matrix-vector multiplication is defined as:

$$Ax = v,$$

where $v = (v_1, \ldots, v_n)$ and

$$v_i = \sum_{j=1}^{m} a_{ij} x_j.$$

Matrix-vector multiplication

- Let us first assume that m is large, but not so large that vector $\boldsymbol{x}$ cannot fit in main memory, and be part of the input to every Map task.

- The matrix $\boldsymbol{A}$ is stored with explicit coordinates, as a triple (i, j, a_{ij}).

- We also assume the position of element x_j in the vector $\boldsymbol{x}$ will be stored in the analogous way.

- **Map**:

Matrix-vector multiplication

- **Map**: each map task will take the entire vector x and a chunk of the matrix A. From each matrix element a_{ij} it produces the key-value pair $(i, a_{ij}x_j)$. Thus, all terms of the sum that make up the component v_i of the matrix-vector product will get the same key.

Matrix-vector multiplication

- **Map**: each map task will take the entire vector x and a chunk of the matrix A. From each matrix element a_{ij} it produces the key-value pair $(i, a_{ij}x_j)$. Thus, all terms of the sum that make up the component v_i of the matrix-vector product will get the same key.
- **Reduce**:

Matrix-vector multiplication

- **Map**: each map task will take the entire vector x and a chunk of the matrix A. From each matrix element a_{ij} it produces the key-value pair $(i, a_{ij}x_j)$. Thus, all terms of the sum that make up the component v_i of the matrix-vector product will get the same key.

- **Reduce**: a reduce task has simply to sum all the values associated with a given key i. The result will be a pair (i, v_i) where:

$$v_i = \sum_{j=1}^{m} a_{ij}x_j.$$
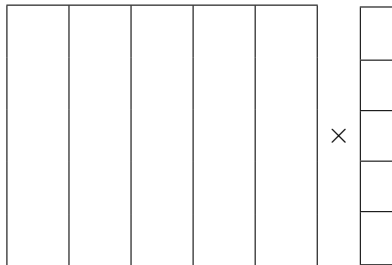
Matrix-vector multiplication in Spark

- The Spark code is quite simple:

```scala
val x = sc.textFile("~/data/x.txt").map(line => {val t = line.split(","); (t(0).
    trim.toInt, t(1).trim.toDouble)})
val vectorX = x.map{case (i,v) => v}.collect
val broadcastedX = sc.broadcast(vectorX)
val matrix = sc.textFile("~/data/M.txt").map(line => {val t = line.split(","); (t
    (0).trim.toInt, t(1).trim.toInt, t(2).trim.toDouble)})
val v = matrix.map { case (i,j,a) => (i, a * broadcastedX.value(j-1))}.reduceByKey(
    _ + _)
v.toDF.orderBy("_1").show
```

Matrix-Vector Multiplication with Large Vector v

- Divide the matrix into vertical stripes of equal width and divide the vector into an equal number of horizontal stripes, of the same height.

- The ith stripe of the matrix multiplies only components from the ith stripe of the vector.
- Thus, we can divide the matrix into one file for each stripe, and do the same for the vector.

Matrix-Vector Multiplication with Large Vector v

- Each Map task is assigned a chunk from one the stripes of the matrix and gets the entire corresponding stripe of the vector.
- The Map and Reduce tasks can then act exactly as in the case where Map tasks get the entire vector.

Relational-Algebra Operations

Example (Relation **Links**)

From	To
url1	url2
url1	url3
url2	url3
url2	url4
...	...

Relational-Algebra Operations

- Selection
- Projection
- Union, Intersection, and Difference
- Natural Join
- Grouping and Aggregation

Relational-Algebra Operations

- R, S - relation
- t, t' - a tuple
- $\mathcal{C}$ - a condition of selection
- A, B, C - subset of attributes
- a, b, c - attribute values for a given subset of attributes

- **Map**:

Selection

- **Map**: For each tuple t in R, test if it satisfies $\mathcal{C}$. If so, produce the key-value pair (t, t). That is, both the key and value are t.
- **Reduce**:

Selection

- **Map**: For each tuple t in R, test if it satisfies $\mathcal{C}$. If so, produce the key-value pair (t, t). That is, both the key and value are t.
- **Reduce**: The Reduce function is the identity. It simply passes each key-value pair to the output.

Projection

- **Map**:

Projection

- **Map**: For each tuple t in R, construct a tuple t' by eliminating from t those components whose attributes are not in A. Output the key-value pair (t', t').

- **Reduce**:

Projection

- **Map**: For each tuple t in R, construct a tuple t' by eliminating from t those components whose attributes are not in A. Output the key-value pair (t', t').

- **Reduce**: For each key t' produced by any of the Map tasks, there will be one or more key-value pairs (t', t'). The Reduce function turns $(t', [t', t', \ldots, t'])$ into (t', t'), so it produces exactly one pair (t', t') for this key t'.

Union

- **Map**:

Union

- **Map**: Turn each input tuple t either from relation R or S into a key-value pair (t, t).
- **Reduce**:

Union

- **Map**: Turn each input tuple t either from relation R or S into a key-value pair (t, t).
- **Reduce**: Associated with each key t there will be either one or two values. Produce output (t, t) in either case.

- **Map**:

Intersection

- **Map**: Turn each input tuple t either from relation R or S into a key-value pair (t, t).
- **Reduce**:

Intersection

- **Map**: Turn each input tuple t either from relation R or S into a key-value pair (t, t).
- **Reduce**: If key t has value list $[t, t]$, then produce (t, t). Otherwise, produce nothing.

Minus

- **Map**:

Minus

- **Map**: For a tuple t in R, produce key-value pair $(t, \texttt{name}(R))$, and for a tuple t in S, produce key-value pair $(t, \texttt{name}(S))$.
- **Reduce**:

Minus

- **Map**: For a tuple t in R, produce key-value pair $(t, \text{name}(R))$, and for a tuple t in S, produce key-value pair $(t, \text{name}(S))$.
- **Reduce**: For each key t, do the following.
  1. If the associated value list is $[\text{name}(R)]$, then produce (t, t).
  2. If the associated value list is anything else, which could only be $[\text{name}(R), \text{name}(S)]$, $[\text{name}(S), \text{name}(R)]$, or $[\text{name}(S)]$, produce nothing.

Natural Join

- Let us assume that we join relation $R(A, B)$ with relation $S(B, C)$ that share the same attribute B.

- **Map**:

Natural Join

- Let us assume that we join relation $R(A, B)$ with relation $S(B, C)$ that share the same attribute B.
- **Map**: For each tuple (a, b) of R, produce the key-value pair $(b, (\texttt{name}(R), a))$. For each tuple (b, c) of S, produce the key-value pair $(b, (\texttt{name}(S), c))$.
- **Reduce**:

Natural Join

- Let us assume that we join relation $R(A, B)$ with relation $S(B, C)$ that share the same attribute B.

- **Map**: For each tuple (a, b) of R, produce the key-value pair $(b, (\texttt{name}(R), a))$. For each tuple (b, c) of S, produce the key-value pair $(b, (\texttt{name}(S), c))$.

- **Reduce**: Each key value b will be associated with a list of pairs that are either of the form $(\texttt{name}(R), a)$ or $(\texttt{name}(S), c)$. Construct all pairs consisting of one with first component $\texttt{name}(R)$ and the other with first component S, say $(\texttt{name}(R), a)$ and $(\texttt{name}(S), c)$. The output for key b is $(b, [(a1, b, c1), (a2, b, c2), ...])$, that is, b associated with the list of tuples that can be formed from an R-tuple and an S-tuple with a common b value.

Grouping and Aggregation

- Let assume that we group a relation $R(A, B, C)$ by attributes A and aggregate values of B.
- **Map**:

- Let assume that we group a relation $R(A, B, C)$ by attributes A and aggregate values of B.
- **Map**: For each tuple (a, b, c) produce the key-value pair (a, b).
- **Reduce**:

Grouping and Aggregation

- Let assume that we group a relation $R(A, B, C)$ by attributes A and aggregate values of B.
- **Map**: For each tuple (a, b, c) produce the key-value pair (a, b).
- **Reduce**: Each key a represents a group. Apply the aggregation operator θ to the list $[b_1, b_2, \ldots, b_n]$ of B-values associated with key a. The output is the pair (a, x), where x is the result of applying θ to the list. For example, if θ is SUM, then $x = b_1 + b_2 + \ldots + b_n$, and if θ is MAX, then x is the largest of $b_1, b_2, \ldots, b_n$.

Matrix Multiplication

- If M is a matrix with element m_{ij} in row i and column j, and N is a matrix with element n_{jk} in row j and column k, then the product:

$$P = MN$$

is the matrix P with element p_{ik} in row i and column k, where:

$$pik =$$

Matrix Multiplication

- If M is a matrix with element m_{ij} in row i and column j, and N is a matrix with element n_{jk} in row j and column k, then the product:

$$P = MN$$

is the matrix P with element p_{ik} in row i and column k, where:

$$pik = \sum_j m_{ij} n_{jk}$$

Matrix Multiplication

- We can think of a matrix M and N as a relation with three attributes: the row number, the column number, and the value in that row and column, i.e.,:

$$M(I, J, V) \quad \text{and} \quad N(J, K, W)$$

with the following tuples, respectively:

$$(i, j, m_{ij}) \quad \text{and} \quad (j, k, n_{jk}).$$

- In case of sparsity of M and N, this relational representation is very efficient in terms of space.

- The product MN is almost a natural join followed by grouping and aggregation.

Matrix Multiplication

Matrix Multiplication

- **Map**:

Matrix Multiplication

- **Map**: Send each matrix element m_{ij} to the key value pair:

$$(j, (M, i, m_{ij})).$$

Analogously, send each matrix element n_{jk} to the key value pair:

$$(j, (N, k, n_{jk})).$$

- **Reduce**:

Matrix Multiplication

- **Map**: Send each matrix element m_{ij} to the key value pair:

$$(j, (M, i, m_{ij})).$$

  Analogously, send each matrix element n_{jk} to the key value pair:

$$(j, (N, k, n_{jk})).$$

- **Reduce**: For each key j, examine its list of associated values. For each value that comes from M, say (M, i, m_{ij}), and each value that comes from N, say (N, k, n_{jk}), produce the tuple

$$(i, k, v = m_{ij}n_{jk}),$$

  The output of the Reduce function is a key j paired with the list of all the tuples of this form that we get from j:

$$(j, [(i_1, k_1, v_1), (i_2, k_2, v_2), \ldots, (i_p, k_p, v_p)]).$$

Matrix Multiplication

- **Map**:

Matrix Multiplication

- **Map**: From the pairs that are output from the previous Reduce function produce p key-value pairs:

$$((i_1, k_1), v_1), ((i_2, k_2), v_2), \ldots, ((i_p, k_p), v_p).$$

- **Reduce**:

Matrix Multiplication

- **Map**: From the pairs that are output from the previous Reduce function produce p key-value pairs:

$$\left((i_1, k_1), v_1\right), \left((i_2, k_2), v_2\right), \ldots, \left((i_p, k_p), v_p\right).$$

- **Reduce**: For each key (i, k), produce the sum of the list of values associated with this key. The result is a pair

$$\left((i, k), v\right),$$

where v is the value of the element in row i and column k of the matrix

$$P = MN.$$

- **Map**:

Matrix Multiplication with One Map-Reduce Step

- **Map**: For each element m_{ij} of M, produce a key-value pair

$$((i, k), (M, j, m_{ij})),$$

  for $k = 1, 2, \ldots$, up to the number of columns of N.
  Also, for each element n_{jk} of N, produce a key-value pair

$$((i, k), (N, j, n_{jk})),$$

  for $i = 1, 2, \ldots$, up to the number of rows of M.

- **Reduce**:

Matrix Multiplication with One Map-Reduce Step

- **Reduce**: Each key (i, k) will have an associated list with all the values

$$(M, j, m_{ij}) \quad \text{and} \quad (N, j, n_{jk}),$$

for all possible values of j. We connect the two values on the list that have the same value of j, for each j:

- ▸ We sort by j the values that begin with M and sort by j the values that begin with N, in separate lists,
- ▸ The jth values on each list must have their third components, m_{ij} and n_{jk} extracted and multiplied,
- ▸ Then, these products are summed and the result is paired with (i, k) in the output of the Reduce function.

Outline

Summary

- Computational burden $\rightarrow$ data partitioning, distributed systems.
- New data-intensive challenges like search engines.
- MapReduce: The overall idea and simple algorithms.
- Spark: MapReduce in practice.
- Algorithms Using Map-Reduce
  - ▶ Relational-Algebra Operations,
  - ▶ Matrix multiplication.

Bibliography

- A. Rajaraman and J. D. Ullman. *Mining of Massive Datasets*.
  Cambridge University Press, 2011
  `http://infolab.stanford.edu/~ullman/mmds.html`

- J.Lin and Ch. Dyer. *Data-Intensive Text Processing with MapReduce*.
  Morgan and Claypool Publishers, 2010
  `http://lintool.github.com/MapReduceAlgorithms/`

- Ch. Lam. *Hadoop in Action*.
  Manning Publications Co., 2011

- `https://spark.apache.org`