Mining Massive Datasets

Lecture 4

Artur Andrzejak
http://pvs.ifi.uni-heidelberg.de

Note on Slides

A part of these slides come (either verbatim or in a modified form) from the book
Mining of Massive Datasets
by Jure Leskovec, Anand Rajaraman, Jeff Ullman (Stanford University).
For more information, see the website accompanying the book: http://www.mmds.org.

Spark:
DataFrames and Datasets

DataFrames in Spark

- **DataFrames** (DF) are tables with named and typed data columns
  - Similar to a dataframe in R, or Pandas (Python), or tables in DBMS/SQL
  - Impose a structure and schema on data
- Example

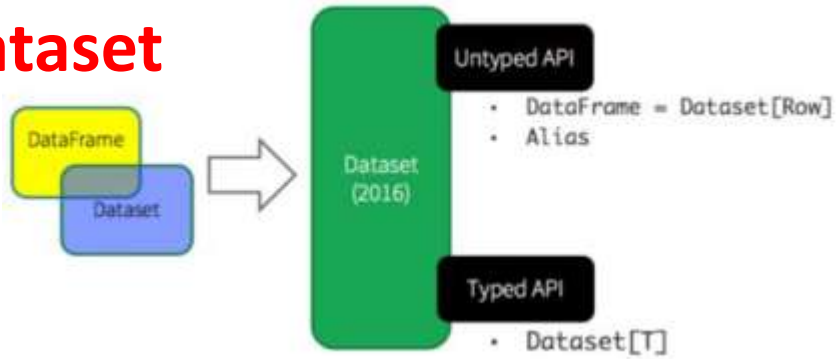
	Partition 1			Partition 2			Partition 3			Partition 4		
	Time (Str)	Site (Str)	Req (Int)	Time (Str)	Site (Str)	Req (Int)	Time (Str)	Site (Str)	Req (Int)	Time (Str)	Site (Str)	Req (Int)
Row 1	ts	m	1304	ts	d	3901	ts	m	1172	ts	m	2538
Row 2	ts	d	2237	ts	d	2491	ts	m	2137	ts	d	2837
Row 3	ts	m	1600	ts	d	2288	ts	d	3176	ts	d	3400

From: Databricks, 7 Steps for a Developer to Learn Apache Spark, p. 14, 2017

DataFrames and Datasets in 2016

From: Databricks, 7 Steps for a Developer to Learn Apache Spark, 2017

- Spark 2.0 unified data structures:
  DF became a specialized **Dataset**
- High-level DSL-like APIs
  for DFs and DS introduced



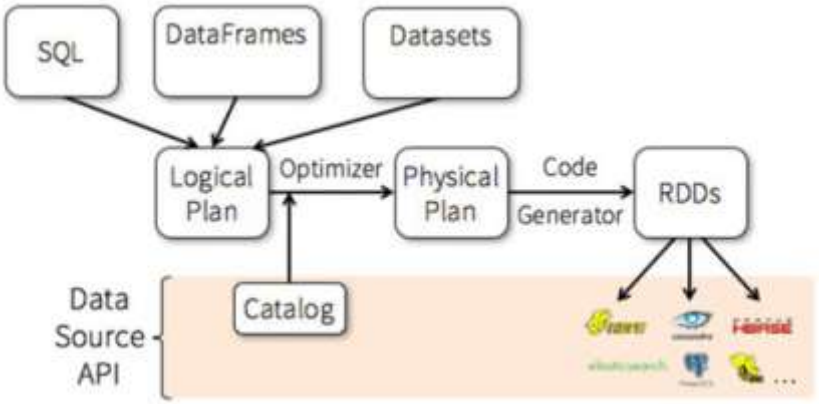
- Dataset is a strongly typed collection of *anything* T
  - APIs only for Scala and Java, not for Python
- DataFrame a is a collection of Row-objects
  - DataFrame = Dataset[Row]

	SQL	DataFrame	Dataset
Syntax errors	Runtime	Compile Time	Compile Time
Analysis errors	Runtime	Runtime	Compile Time

API ~ Domain Specific Language

- APIs for DFs and DSs provide high-level operators like sum, count, avg, min, max etc.
- Highly-efficient code generated (faster than for RDDs!)
- Example (in Scala):

From: Databricks, 7 Steps for a Developer to Learn Apache Spark, 2017




```
// a dataset with field names fname, lname, age, weight
// access using object notation
val seniorDS = peopleDS.filter( p => p.age > 55 )
// a dataframe with named columns fname, lname, age, weight
// access using col name notation
val seniorDF = peopleDF.where( peopleDF("age") > 55)
// equivalent Spark SQL code
val seniorDF = spark.sql("SELECT age from person where age > 35")
```

Creating DataFrames in PySpark

- A DF can be created from multiple sources …
  - By converting „normal“ RDDs
  - Loading from text, csv, json, xml, parquet files
  - Importing from DBMS (Hive, Cassandra, ..)
- Each DF has a **schema**: definition of names and types of columns
  - Schema can be set programmatically, or inferred from data

DataFrames from RDDs

```
// Read file with rows: <name, age>
filePath = „/home/immd-user/spark-2 …/examples/src/main/resources/people.txt“
parts = sc.textFile(filePath).map( lambda line: line.split(“,“) )

// Each row should become a tuple (name, age)
peopleRDD = parts.map( lambda p: (p[0], p[1].strip() )

from pyspark.sql import SparkSession
spark = SparkSession.builder.appName(„Exmpl").getOrCreate()
schema = StructType( [
                StructField (“name“,  StringType(), True),
                StructField ( „age“,     StringType(), True) ] )

dfPeople = spark.createDataFrame (peopleRDD, schema)
print ( dfPeople.take(5) )
```

Read DFs From Other Sources

filePath = „/home/immd-user/spark-2 …/examples/src/main/resources/people.txt“

```
// Read from CSV (comma separated values) files
df_csv = spark.read.csv(filePath, schema = schema)
print (df_csv.take(5) )

// Read from json – schema is inferred!
df_json =
spark.read.json("examples/src/main/resources/people.json")

df_json.show()
    # +----+-------+
    # | age|   name|
    # +----+-------+
    # |null|Michael|
    # |  30|   Andy|
    # |  19| Justin|
    # +----+-------+
```

DFs Operations

```
# Print schema
dfPeople.printSchema()
    # root
    # |-- age: string (nullable = true)
    # |-- name: string (nullable = true)
# Select only the "name" column
dfPeople.select("name").show()
    # |    name|
    # |Michael| …
# Group by age and count per group
dfPeople. groupBy("age").count().show()
    # | age|count|
    # |  19|    1|
    # |null|    1|
    # |  30|    1|
```

SQL: More User-Friendly

Standard DF API:

```
# Select people older than 25
dfPeople.filter(dfPeople['age'] > 25).show()
    # |age|name|
    # | 30|Andy|
```

Same result with **SQL**:

```
# Register the DataFrame as a SQL temporary view
dfPeople.createOrReplaceTempView("people")

sqlDF = spark.sql("SELECT * FROM people where age > 25")
sqlDF.show()
```

User Defined Functions (UDFs)

```
from pyspark.sql.functions import *
from pyspark.sql.types import *
df = sqlContext.read.parquet('hdfs:///.../stations.parquet')

lat2dir =  udf(lambda x: 'N' if x > 0 else 'S',   StringType())
lon2dir = udf(lambda x: 'E' if x > 0 else 'W',  StringType())

df.select(df.lat, lat2dir(df.lat).alias('latdir'),
        df.long, lon2dir(df.long).alias('longdir')).show()
```

```
|     lat          |latdir|      long       |longdir|
|37.329732         |    N|-121.901782|       W|
|37.330698         |    N|-121.888979|       W|
|   ...            |  ...  |    ...          |  ...     |
```

From: Jeffrey Aven: Sams teach yourself Apache Spark in 24 hours, 2017

Standard DFs Functions

There are many ready-to-use functions, similar to ones in SQL

Type	Sample Available Functions
String functions	`startswith, substr, concat, lower, upper, regexp_extract, regexp_replace`
Math functions	`abs, ceil, floor, log, round, sqrt`
Statistical functions	`avg, max, min, mean, stddev`
Date functions	`date_add, datediff, from_utc_timestamp`
Hashing functions	`md5, sha1, sha2`
Algorithmic functions	`soundex, levenshtein`
Windowing functions	`over, rank, dense_rank, lead, lag, ntile`

From: Jeffrey Aven: Sams teach yourself Apache Spark in 24 hours, 2017

More Resources on DataFrames

- Jeffrey Aven: Sams teach yourself Apache Spark in 24 hours, 2017, http://katalog.ub.uni-heidelberg.de/cgi-bin/titel.cgi?katkey=68102164 (free from univ. domain!)
- Databricks, 7 Steps for a Developer to Learn Apache Spark, 2017, https://goo.gl/chn8GE
- Spark Documentation: Spark SQL, DataFrames and Datasets Guide, https://goo.gl/HuHNEq
- Ankit Gupta: Complete Guide on DataFrame Operations in PySpark, 2016, https://goo.gl/mrNL4i
- Michael Armbrust, Wenchen Fan, Reynold Xin and Matei Zaharia: Introducing Apache Spark Datasets, 2016, https://goo.gl/VLu9dn

MapReduce Paradigm

Fundamentals

Programming Model: MapReduce

Warm-up task:

- We have a huge text document
- Count the number of times each distinct word appears in the file
- **Sample application:**
  - Analyze web server logs to find popular URLs

Task: Word Count

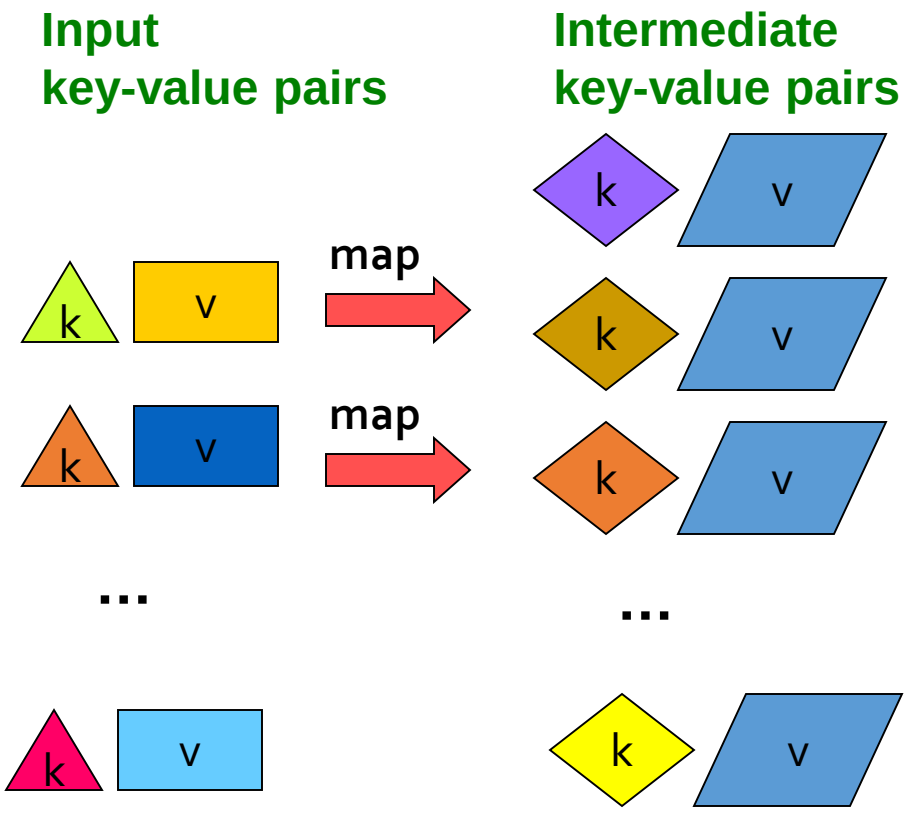
- Count occurrences of words:
  - **`words(doc.txt) | sort | uniq -c`**
    - where **`words`** takes a file and outputs the words in it, one per line
- This captures the essence of **MapReduce**
  - Great thing is that it is naturally parallelizable

MapReduce: Overview

- Sequentially read a lot of data
- **Map:**
  - Extract something you care about
- **Group by key:** Sort and Shuffle
- **Reduce:**
  - Aggregate, summarize, filter or transform
- Write the result

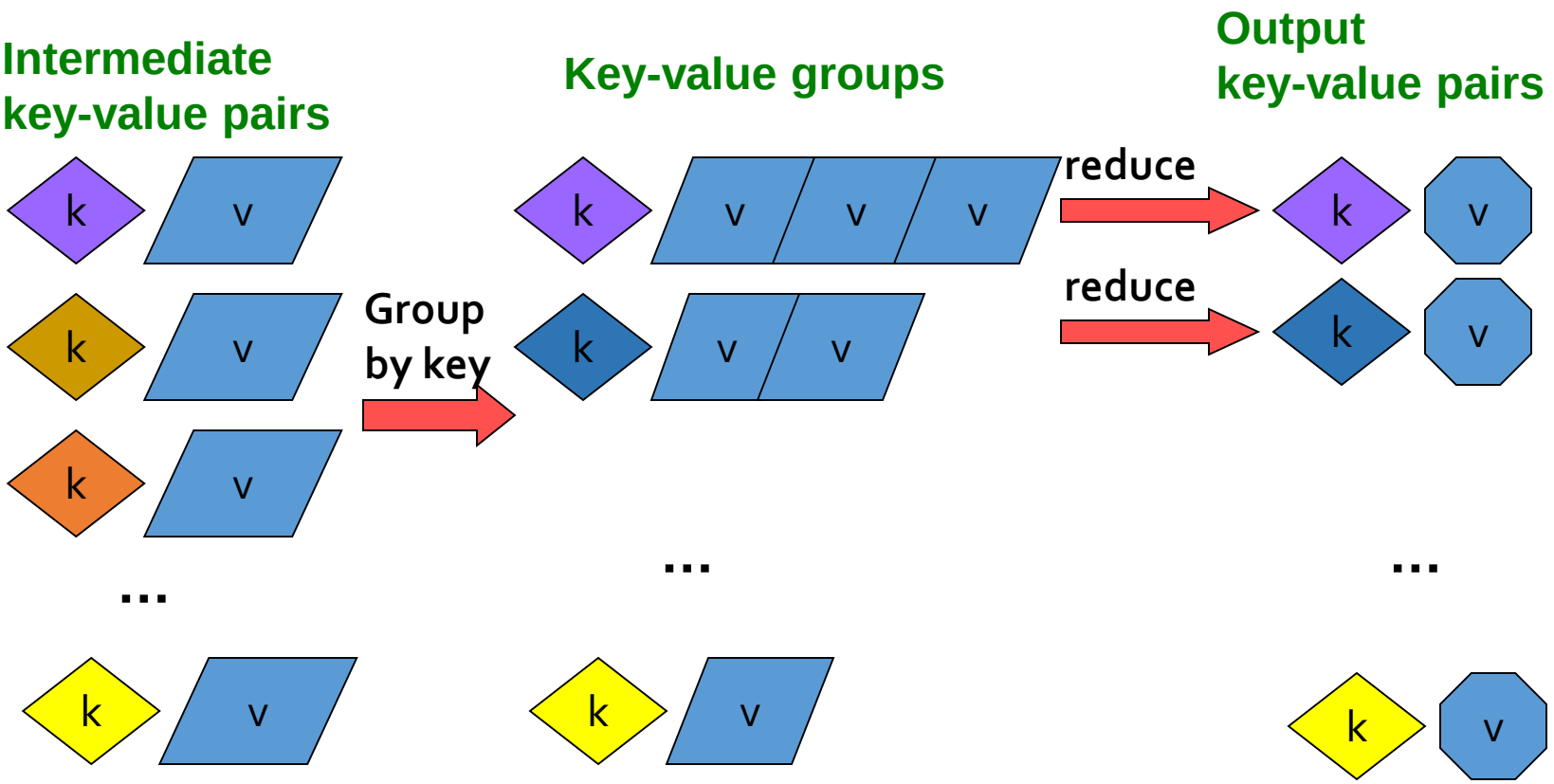
Outline stays the same, **Map** and **Reduce** change to fit the problem

MapReduce: The Map Step



This step in Spark?

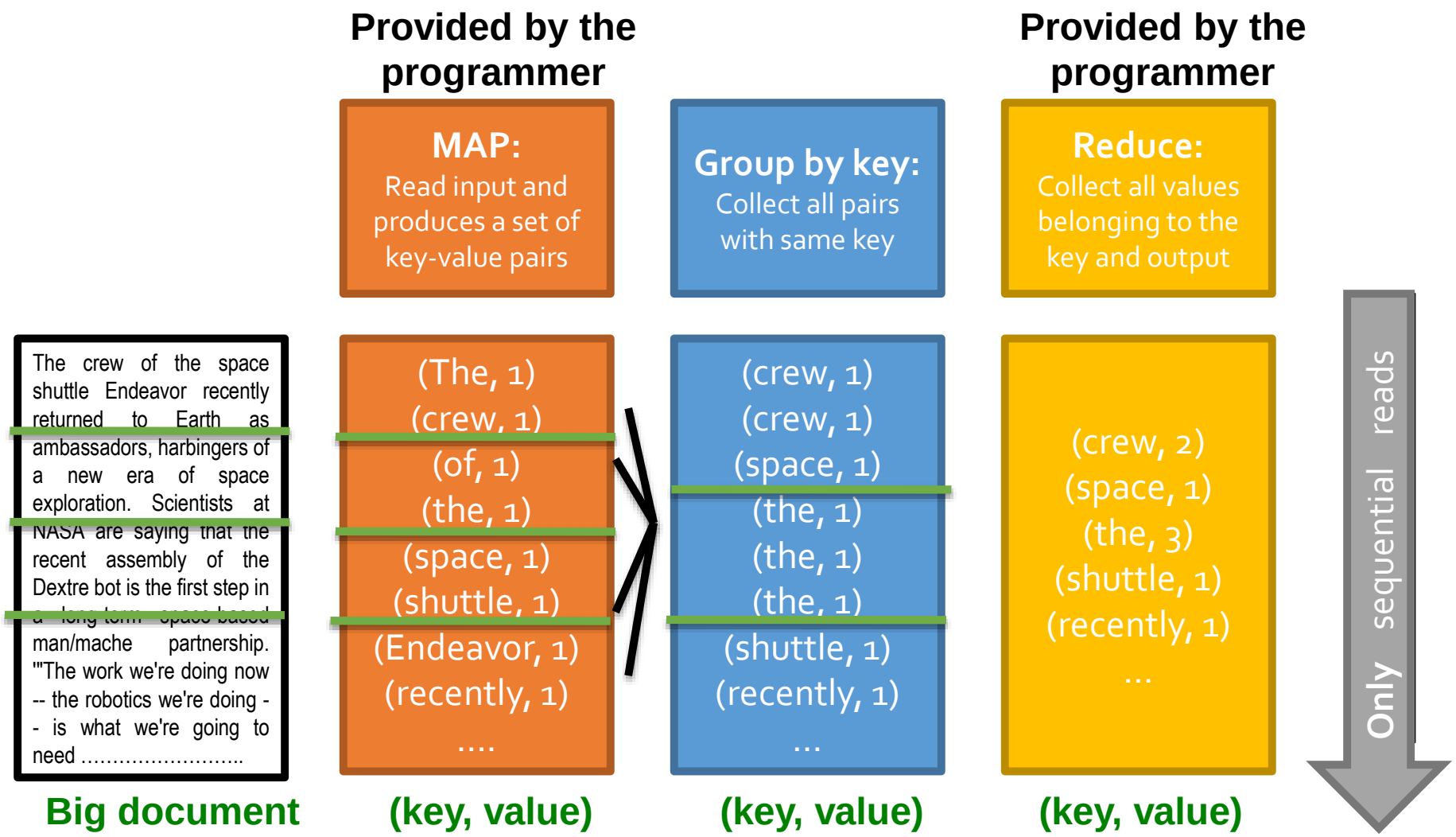
MapReduce: The Reduce Step



More Specifically

- **Input:** a set of key-value pairs
- Programmer specifies two methods:
  - **Map(k, v)** $\rightarrow$ <k', v'>*
    - Takes a key-value pair and outputs a set of key-value pairs
      - E.g., key is the filename, value is a single line in the file
    - There is one Map call for every *(k,v)* pair
  - **Reduce(k', <v'>*)** $\rightarrow$ <k', v''>*
    - **All values *v'* with same key *k'* are reduced together and processed in *v'* order**
    - There is one Reduce function call per unique key *k'*

MapReduce: Word Counting



Word Count Using MapReduce

```
map(key, value):
// key: document name; value: text of the document
   for each word w in value:
      emit(w, 1)
```

```
reduce(key, values):
// key: a word; value: an iterator over counts
      result = 0
      for each count v in values:
            result += v
      emit(key, result)
```

Map-Reduce: Environment

Map-Reduce environment takes care of:

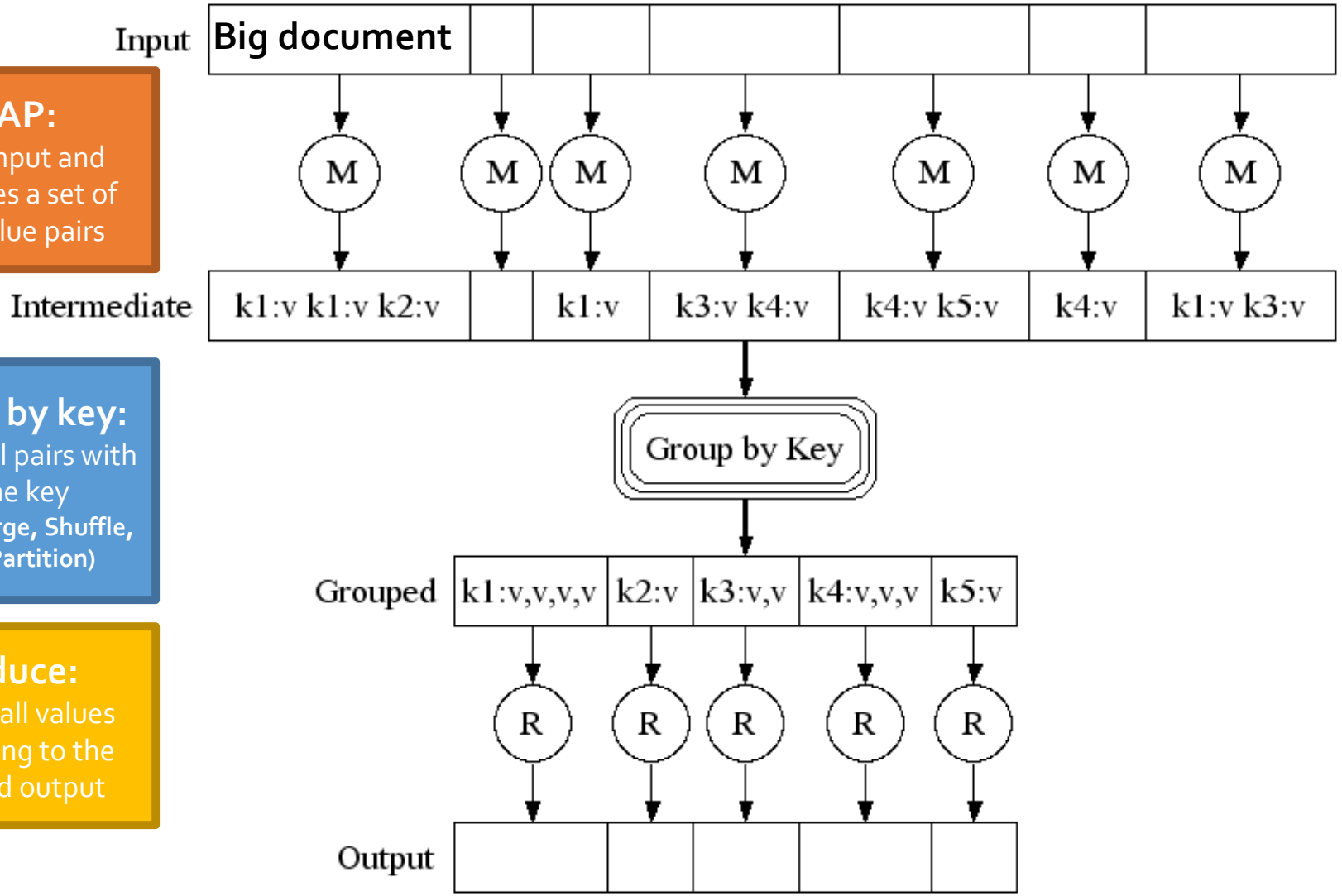
- Partitioning the input data
- Scheduling the program's execution across a set of machines
- Performing the **group by key** step
- Handling machine failures
- Managing required inter-machine communication

Map-Reduce: A diagram

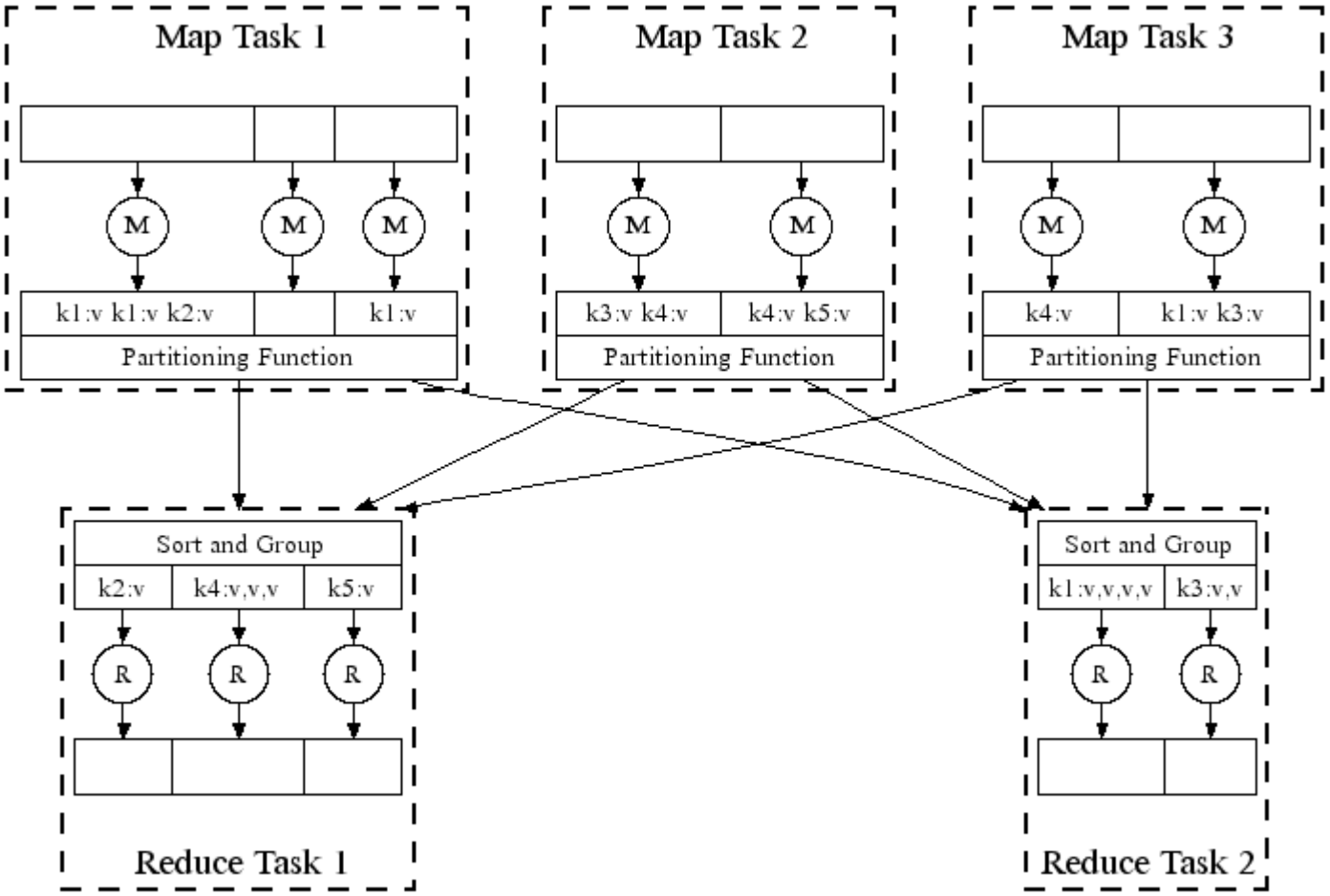
MAP:
Read input and produces a set of key-value pairs

Group by key:
Collect all pairs with same key
(Hash merge, Shuffle, Sort, Partition)

Reduce:
Collect all values belonging to the key and output



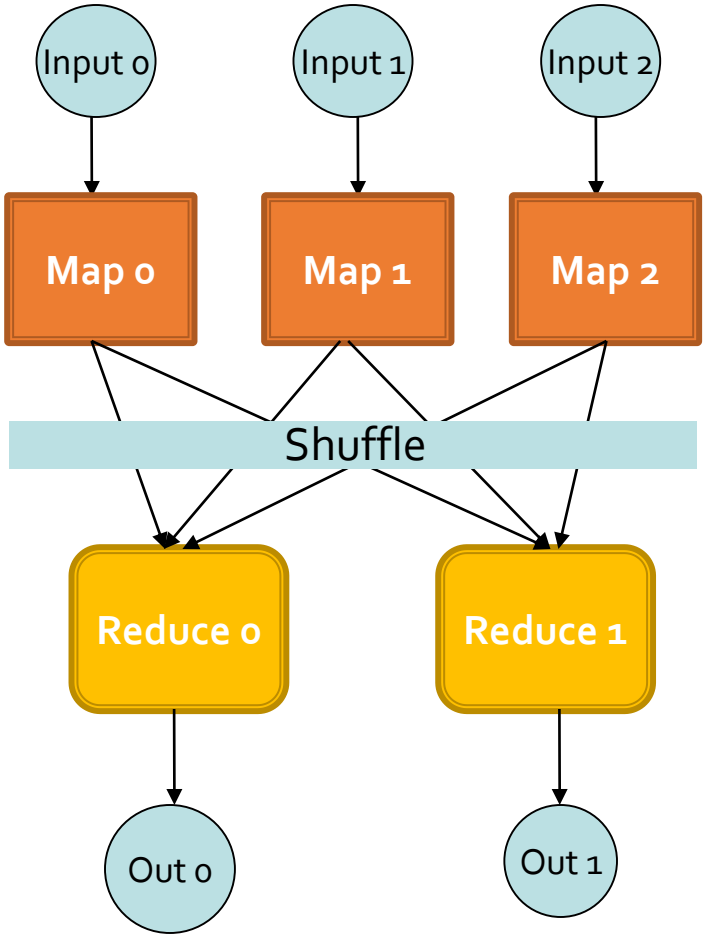
Map-Reduce: In Parallel



All phases are distributed with many tasks doing the work

Map-Reduce

- Programmer specifies:
  - Map and Reduce and input files
- **Workflow:**
  - Read inputs as a set of key-value-pairs
  - **Map** transforms input kv-pairs into a new set of k'v'-pairs
  - Sorts & Shuffles the k'v'-pairs to output nodes
  - All k'v'-pairs with a given k' are sent to the same **reduce**
  - **Reduce** processes all k'v'-pairs grouped by key into new k''v''-pairs
  - Write the resulting pairs to files
- All phases are distributed with many tasks doing the work



Data Flow

- **Input and final output are stored on a distributed file system (FS):**
  - Scheduler tries to schedule map tasks "close" to physical storage location of input data
- **Intermediate results are stored on local FS of Map and Reduce workers**
- **Output is often input to another MapReduce task**

MapReduce Paradigm

Advanced Concepts

Coordination: Master

- Master node takes care of coordination:
  - **Task status:** (idle, in-progress, completed)
  - **Idle tasks** get scheduled as workers become available
  - When a map task completes, it sends the master the location and sizes of its *R* intermediate files, one for each reducer
  - Master pushes this info (but not data!) to reducers
- Master pings workers periodically to detect failures

Dealing with Failures

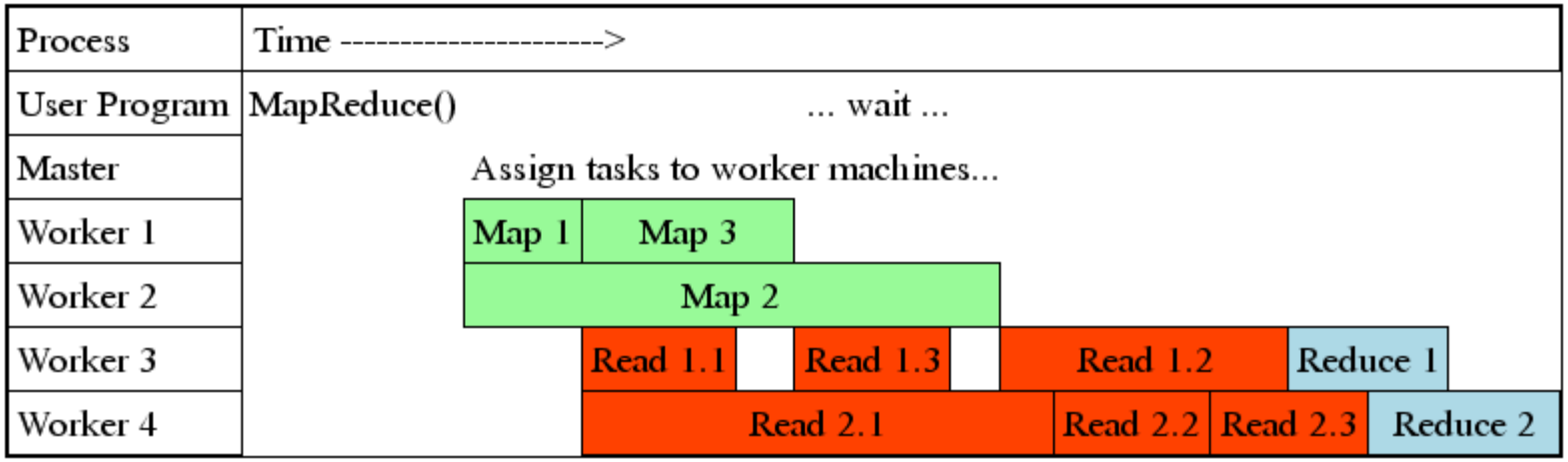
- **Map worker failure**
  - Map tasks completed or in-progress at worker are reset to idle
  - Reduce workers are notified when task is rescheduled on another worker
- **Reduce worker failure**
  - Only in-progress tasks are reset to idle
  - Reduce task is restarted
- **Master failure**
  - MapReduce task is aborted and client is notified

How many Map and Reduce jobs?

- *M* map tasks, *R* reduce tasks
- **Rule of a thumb:**
  - Make *M* much larger than the number of nodes in the cluster
  - One DFS chunk per map is common
  - Improves dynamic load balancing and speeds up recovery from worker failures
- **Usually *R* is smaller than *M***
  - Because output is spread across *R* files

Task Granularity & Pipelining

- **Fine granularity tasks:** map tasks >> machines
  - Minimizes time for fault recovery
  - Can do pipeline shuffling with map execution
  - Better dynamic load balancing

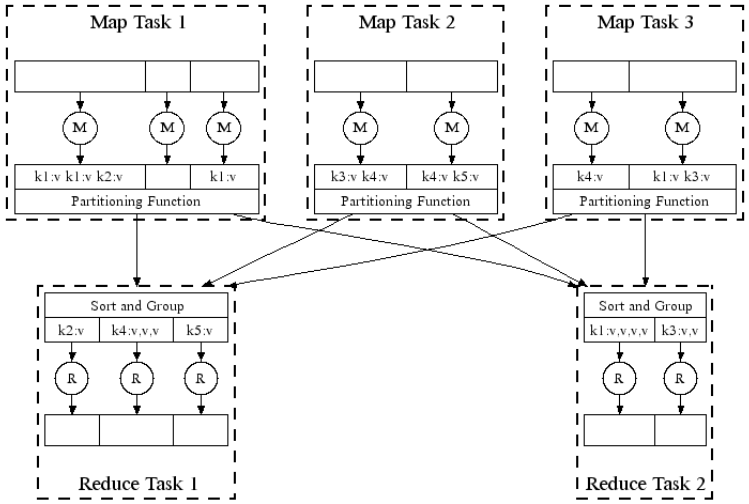


Refinements: Backup Tasks

- **Problem**
  - Slow workers significantly lengthen the job completion time:
    - Other jobs on the machine
    - Bad disks
    - Weird things
- **Solution**
  - Near end of phase, spawn backup copies of tasks
    - Whichever one finishes first “wins”
- **Effect**
  - Dramatically shortens job completion time

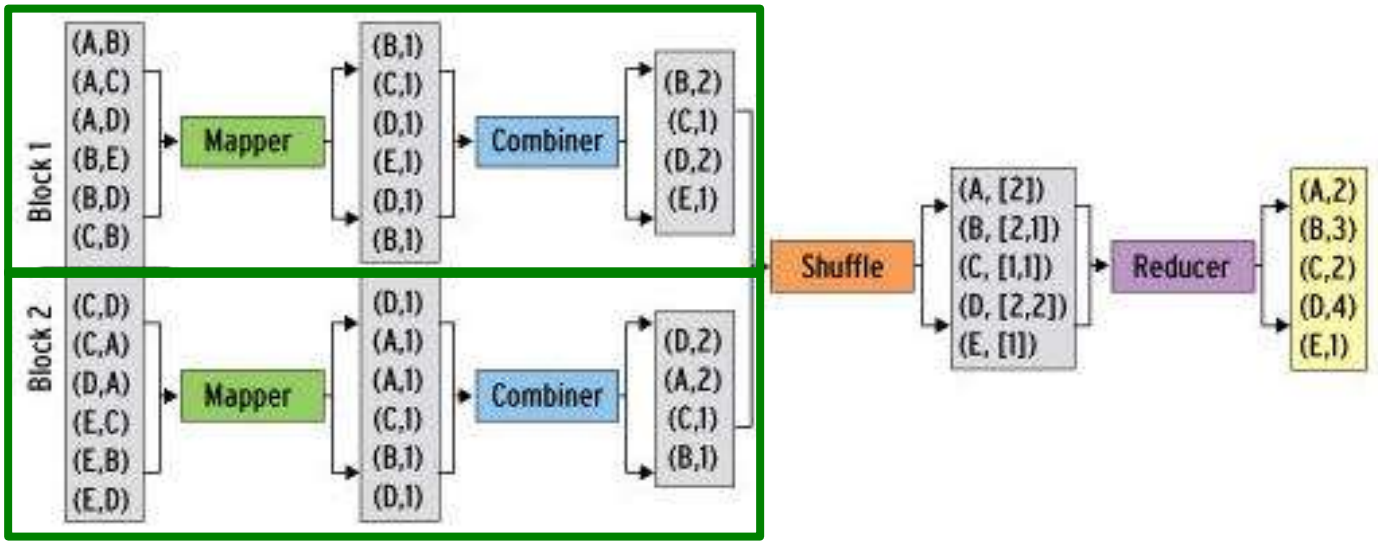
Refinement: Combiners

- Often a Map task will produce many pairs of the form $(k,v_1), (k,v_2), \ldots$ for the same key k
  - E.g., popular words in the word count example
- **Can save network time by pre-aggregating values in the mapper:**
  - combine(k, list(v_1)) → v_2
  - Combiner is usually same as the reduce function
- Works only if reduce function is commutative and associative



Refinement: Combiners

- **Back to our word counting example:**
  - Combiner combines the values of all keys of a single mapper (single machine):



  - Much less data needs to be copied and shuffled!

Refinement: Partition Function /1

- Partition Function controls how intermediate keys are assigned to reduce-workers
- System uses a default partition function:
  - **hash(key) mod R** (R = # of reduce workers)
  - If hash(key1) == hash(key2), then the same reduce worker will receive (key1, vals[]) and (key2, vals[])
- Overriding is useful to ensure that a group of reduce-tasks can write to a single file (or a local file system)
  - To override you need a custom implementation:

```
public static class MyPartitioner extends Partitioner<keyType>,
        <valType> {
    @Override getPartition(<keyType> key, <valType> val, int R) {
    ...}
}
```

User Defined Function (UDF)

Refinement: Partition Function /2

- E.g. in word count: you want that counts for all words from ‘a’ to ‘f’ should end up in a single file (or local file system)
- => Ensure that the reducer-tasks for keys ‘a’, ‘b’, …, ‘f’ are on the same machine (ID == 0):

```
getPartition(Text key, <valType> val, int R) {
    int firstLetter = key.charAt(0);                  // Unicode value
    if (97 <= firstLetter && firstLetter <= 102)   // ‘a’=97, ‘f’=102
            return 0;
    return 1    }
```

- A longer example:
  - http://hadooptutorial.wikispaces.com/Custom+partitioner

Expressing Algorithms in MapReduce

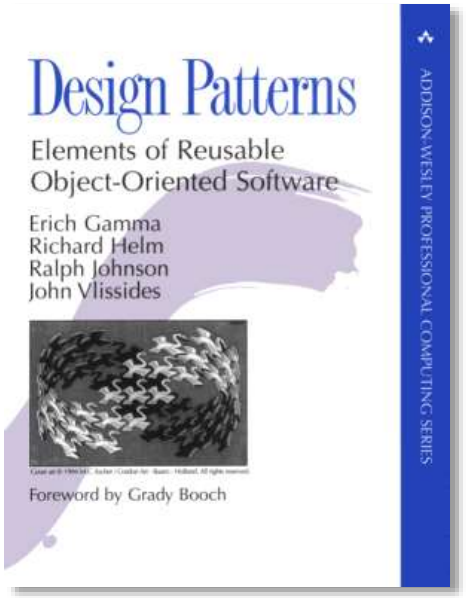
Example: Count 5-Word Sequences

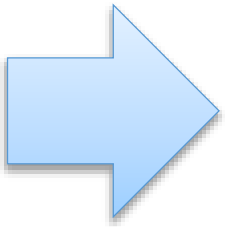
- **Statistical machine translation:**
  - Need to count number of times every 5-word sequence occurs in a large corpus of documents

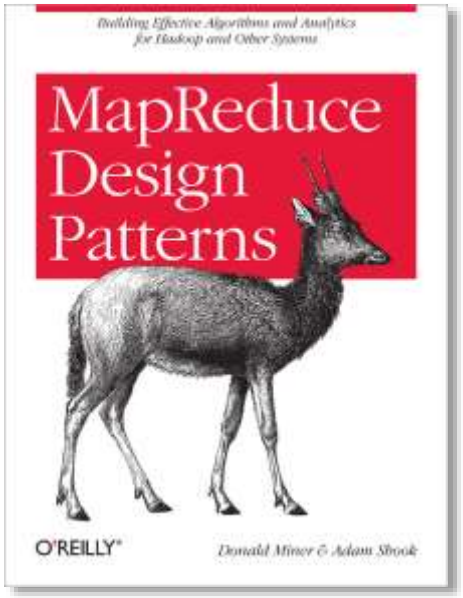
- **Very easy with MapReduce:**
  - **Map:**
    - Extract (5-word sequence, count) from document
  - **Reduce:**
    - Combine the counts

Design Patterns

- Design Patterns are “best solutions” for recurrent patterns of code => save thinking ☺
- Inspiration for MapReduce comes from Object-Oriented Design Patterns
  - Compared to OO-patterns: added consideration for performance





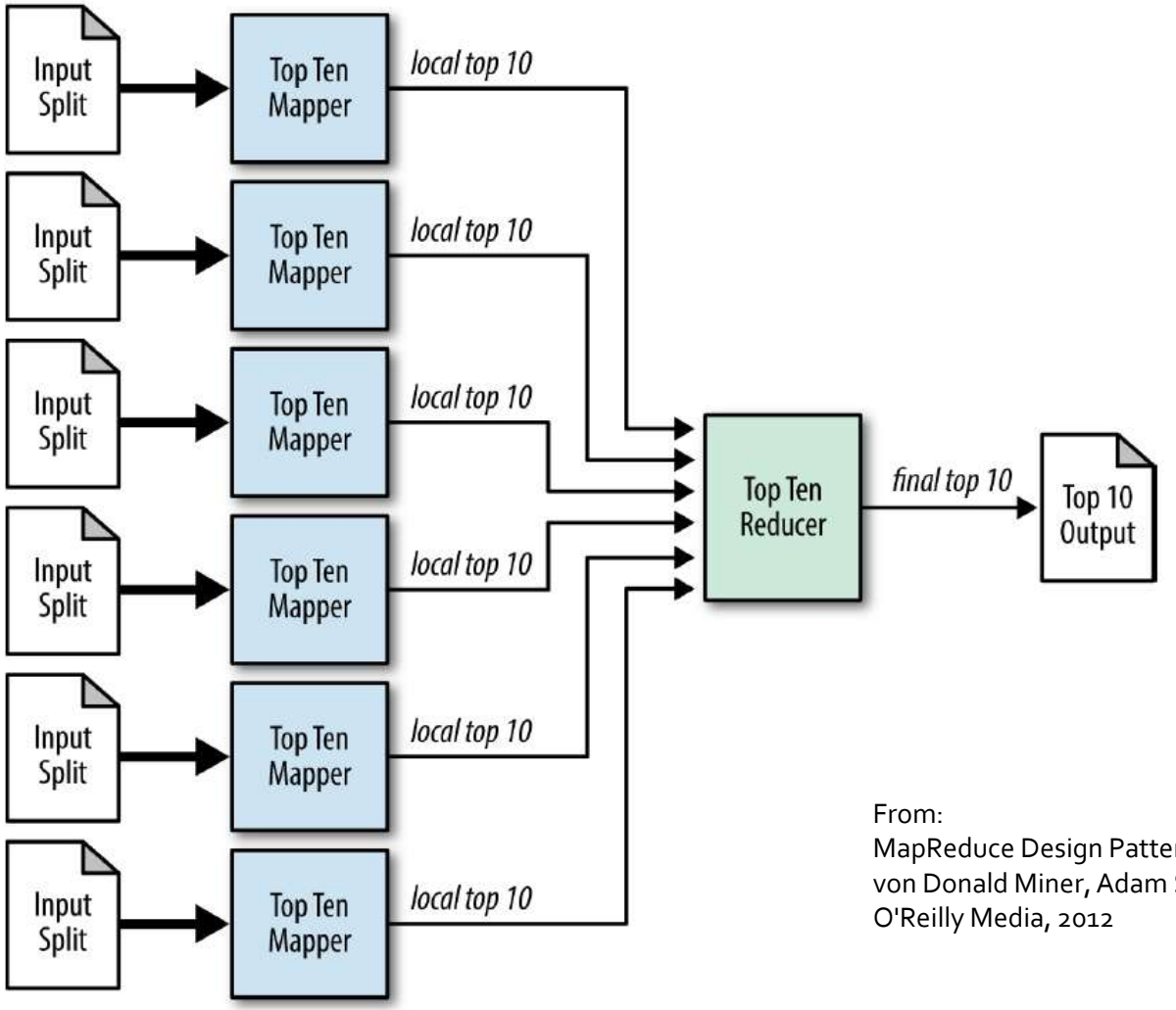


Example: "Top Ten"

- **Intent**
  - Retrieve a relatively small number of top k records, according to a ranking scheme in your data set
  - No matter how large the data
  - Generalization of maximum / minimum (k=1)
- **Motivation**
  - Finding outliers
  - Top ten lists are fun
  - Building dashboards
  - Sorting/Limit isn't going to work here

"Top Ten" with Map-Reduce

How to implement?



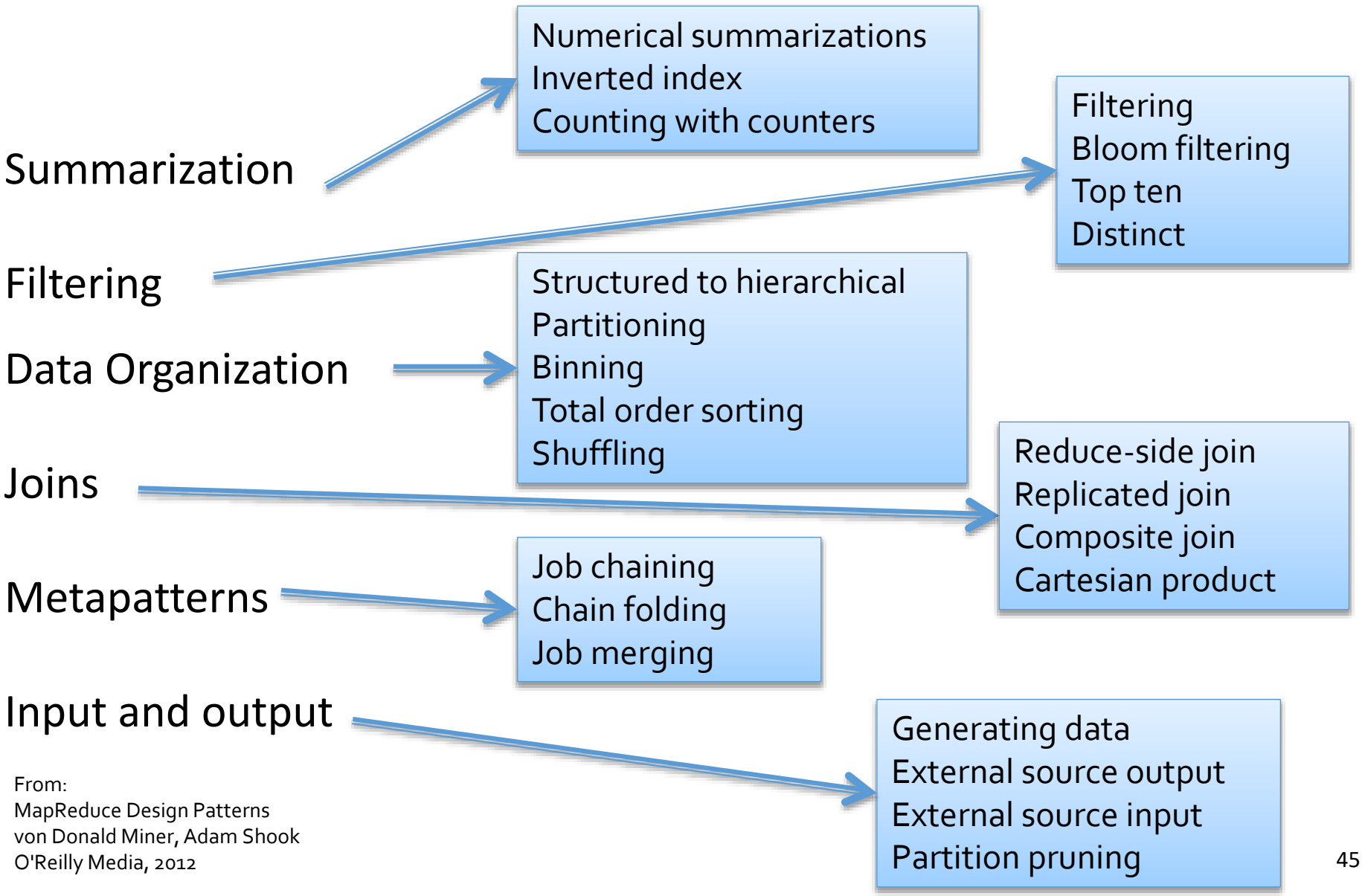
From:
MapReduce Design Patterns
von Donald Miner, Adam Shook
O'Reilly Media, 2012

"Top Ten" - Pseudocode

```
class mapper:
  setup ():
    initialize top k sorted list T (small local array of size k)
  map (key, record):
    compare record r against current "worst" element of T
    if r is better then
            insert r into T (according to ranking)
    if length(T) > k:
      truncate T to a length of k
  cleanup ():
    for record in T:
            emit null,record
class reducer:
  setup ():
    initialize top k sorted list T (small local array of size k)
  reduce (key, records):
    sort records (according to ranking)
    truncate records to length k
    for record in records:
            emit record
```

From:
MapReduce Design Patterns
von Donald Miner, Adam Shook
O'Reilly Media, 2012

Categories of Patterns (Book)



From:
MapReduce Design Patterns
von Donald Miner, Adam Shook
O'Reilly Media, 2012

Example: Join By Map-Reduce

- **Compute the natural join *R(A,B)* ⋈ *S(B,C)***
- R and S are each stored in files
- Tuples are pairs *(a,b)* (for R) or *(b,c)* (for S)

A	B
a_1	b_1
a_2	b_1
a_3	b_2
a_4	b_3

R

⋈

B	C
b_2	c_1
b_2	c_2
b_3	c_3

S

=

A	C
a_3	c_1
a_3	c_2
a_4	c_3

Map-Reduce Join

- **Use a hash function *h* from b-values to *1...k***
- **A Map process turns:**
  - Each input tuple *R(a,b)* into key-value pair *(b,(a,R))*
  - Each input tuple *S(b,c)* into *(b,(c,S))*

Only "marker" for S, R

- **Map processes** send each key-value pair with key *b* to reduce process *h(b)*
  - Hadoop does this automatically; just tell it what ***k*** is
- Each **Reduce process** matches all the pairs *(b,(a,R))* with all *(b,(c,S))* and outputs *(a,b,c)*

Thank you.

Questions?

Additional Materials and Further Reading

Hadoop Resources

- Releases from Apache download mirrors
  - http://www.apache.org/dyn/closer.cgi/lucene/hadoop/
- Nightly builds of source
  - http://people.apache.org/dist/lucene/hadoop/nightly/
- Source code from subversion
  - http://lucene.apache.org/hadoop/version_control.html

Cost of Map-Reduce

Cost Measures for Algorithms

- **In MapReduce we quantify the cost of an algorithm using**

1. *Communication cost* = total I/O of all processes
2. *Elapsed communication cost* = max of I/O along any path
3. (*Elapsed*) *computation cost* analogous, but count only running time of processes

Note that here the big-O notation is <u>not</u> the most useful (adding more machines is always an option)

Example: Cost Measures

- **For a map-reduce algorithm:**
  - **Communication cost =** input file size + 2 × (sum of the sizes of all files passed from Map processes to Reduce processes) + the sum of the output sizes of the Reduce processes
  - **Elapsed communication cost** is the sum of the largest input + output for any map process, plus the same for any reduce process

What Cost Measures Mean

- Either the I/O (communication) or processing (computation) cost dominates
  - Ignore one or the other
- Total cost tells what you pay in rent from your friendly neighborhood cloud
- Elapsed cost is wall-clock time using parallelism

Cost of Map-Reduce Join

- **Total communication cost**
  = $O(|R|+|S|+|R \bowtie S|)$
- **Elapsed communication cost** = O(s)
  - We're going to pick ***k*** and the number of Map processes so that the I/O limit ***s*** is respected
  - We put a limit ***s*** on the amount of input or output that any one process can have. ***s* could be:**
    - What fits in main memory
    - What fits on local disk
- With proper indexes, computation cost is linear in the input + output size
  - So computation cost is like comm. cost