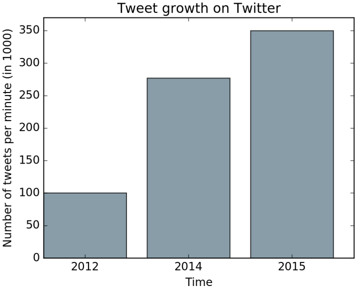
JOHANNES GUTENBERG UNIVERSITÄT MAINZ

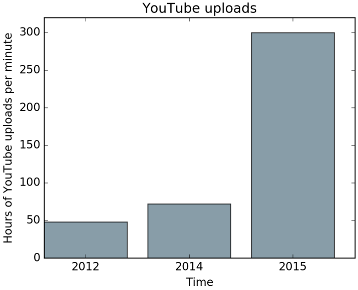
Mining of Massive Datasets

Steven Lang

July 2016

Motivation





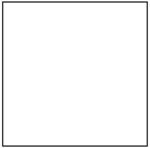
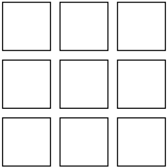
- Growing amounts of data
- Requires specific methods to analyze the data
- Result: Special computing models for clusters
- Concentrate on efficiently parallelizing operations

Hildebrandt, *Big Data: Algorithms, methods and techniques for analysing massive datasets*

Motivation

- Massive data has lead to novel approaches of data-mining
- Classic examples:
  - Ranking web pages by importance
  - Querying social-networks

Compute Cluster

	Trad. supercomputer	Cluster computer
scale	up $\Uparrow$	out $\Longleftrightarrow$
hardware	special	commodity
costs	intensive	low
failure rate (per node)	low	high
schematic		

Compute Cluster

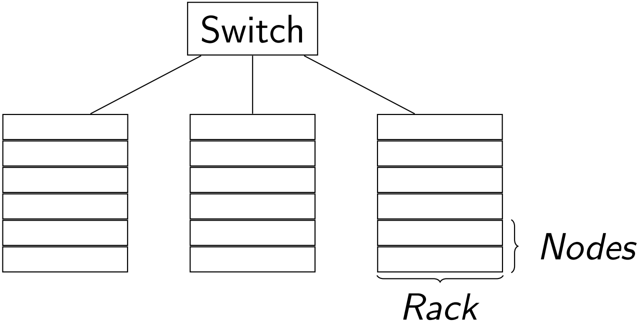


Figure: Example cluster with three racks and six nodes each

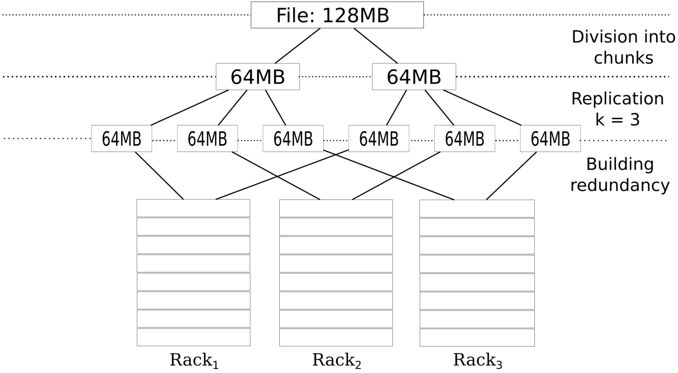
Compute Cluster

To get the most benefit out of these clusters one has to address:

- **Node failures** (availability of data)
  - Solution: Distributed file system (DFS)
- **Modularity of tasks**
  - Solution: Programming model called MapReduce

Distributed File System (DFS)

- File system for cluster computing
- Used to achieve consistent availability of data in clusters



MapReduce
Introduction

What is MapReduce?

- Programming model
- Allows scalability across complete compute cluster
- Takes care of:

MapReduce

Introduction

What is MapReduce?

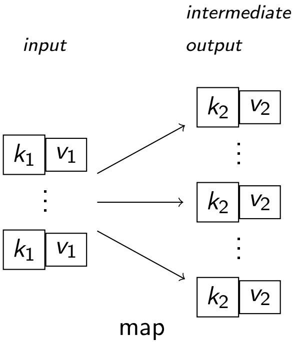
- Programming model
- Allows scalability across complete compute cluster
- Takes care of:
  - Parallelism
  - Task-coordination
  - Node-failures

MapReduce

Introduction

MR basically consists of two methods:

- **Map**: maps each input element to zero or more intermediate output elements

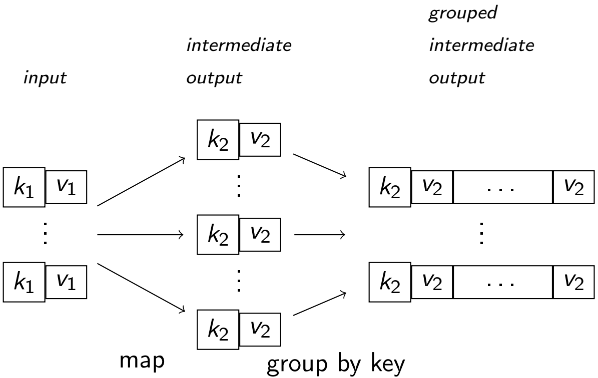


MapReduce

Introduction

MR basically consists of two methods:

- **Map**: maps each input element to zero or more intermediate output elements
- System groups intermediate results by key

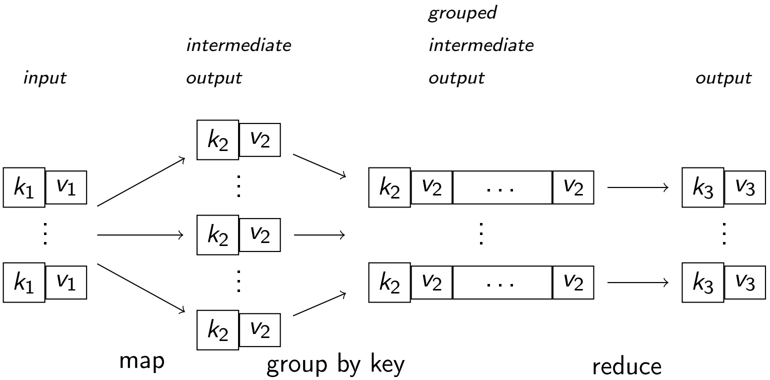


MapReduce

Introduction

MR basically consists of two methods:

- **Map**: maps each input element to zero or more intermediate output elements
- System groups intermediate results by key
- **Reduce**: reduces several map-outputs with the same key to one result output element



Terminology

- **Mapper**: The map-function
- **Reducer**: The reduce-function
- **Map-Task**: A process - takes one or more chunks of input data and executes the mapper
- **Reduce-Task**: A process - takes one or more key-value pairs and executes the reducer

Illustrating MapReduce with a common example: WordCount

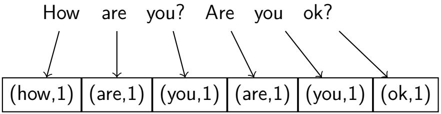
- Goal: Count words in a repository of documents

```
map(key, value):
  # Get words from document
  words = value.split(' ')

  # Emit each word
  for w in words:
    emit(w, 1)

reduce(key, value):
  # Sum up & emit
  s = sum(value)
  emit(key, s)
```

map:



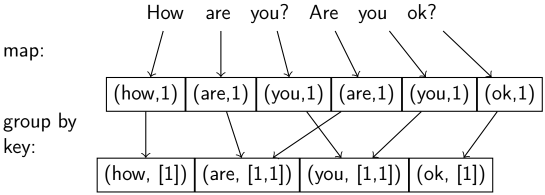
Illustrating MapReduce with a common example: WordCount

- Goal: Count words in a repository of documents

```
map(key, value):
  # Get words from document
  words = value.split(' ')

  # Emit each word
  for w in words:
    emit(w, 1)

reduce(key, value):
  # Sum up & emit
  s = sum(value)
  emit(key, s)
```



Illustrating MapReduce with a common example: WordCount

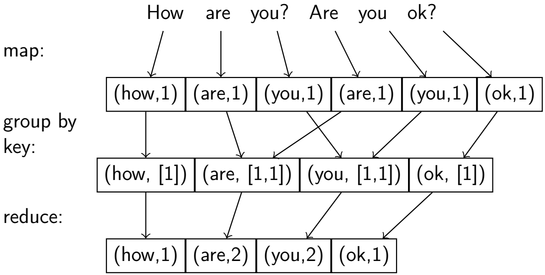
- Goal: Count words in a repository of documents

```
map(key, value):
  # Get words from document
   words = value.split(' ')

  # Emit each word
   for w in words:
      emit(w, 1)

reduce(key, value):
  # Sum up & emit
   s = sum(value)
   emit(key, s)

```



Coping with node failures

- **Worst case**: Master-node failure
  - Restart whole MR-Job

Coping with node failures

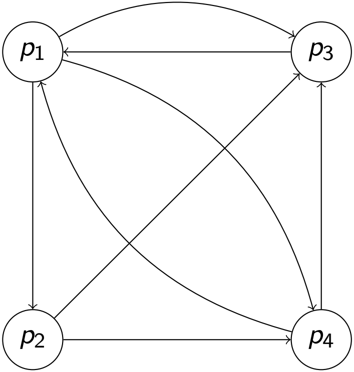
- **Worst case**: Master-node failure
  - Restart whole MR-Job
- **Less tragic**: Map-Node failure
  - Restart all its Map-Tasks
  - Notify Reduce-Tasks of new input-location

Coping with node failures

- **Worst case**: Master-node failure
  - Restart whole MR-Job
- **Less tragic**: Map-Node failure
  - Restart all its Map-Tasks
  - Notify Reduce-Tasks of new input-location
- **Most simple**: Reduce-Node failure
  - Reschedule its Reduce-Tasks at another Reduce-Node

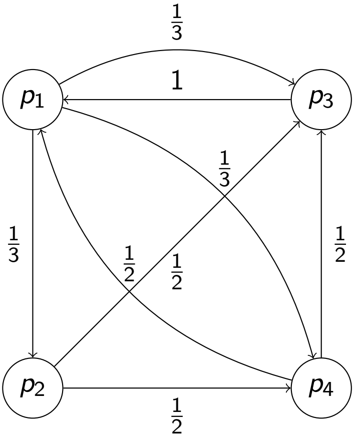
Application of MapReduce: Matrix-Vector-Multiplication

PageRank



Application of MapReduce: Matrix-Vector-Multiplication

PageRank



- $w(p_i, p_j) = \frac{1}{outdeg(p_i)}$
- $M \in \mathbb{Q}^{n \times n}$ Linkage matrix
- $V_k \in \mathbb{Q}^n$ Importance vector after the k-th step
- $V_0 = (\frac{1}{n}, \dots, \frac{1}{n})^T$
- $V_k = M \cdot V_{k-1}$

$$\begin{pmatrix} 0 & 0 & 1 & \frac{1}{2} \\ \frac{1}{3} & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{3} & \frac{1}{2} & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} \frac{1}{4} \\ \frac{1}{4} \\ \frac{1}{4} \\ \frac{1}{4} \end{pmatrix} = V_1$$

Application of MapReduce: Matrix-Vector-Multiplication

- PageRank: Large matrix-vector multiplications
- Input:
  - Matrix $M \in \mathbb{Q}^{n \times n}$
  - Vector $V \in \mathbb{Q}^{n}$
- Matrix-vector multiplication $X = M \cdot N$ denoted as:

$$x_i = \sum_{j=0}^{n} m_{ij} \cdot v_i$$

Application of MapReduce: Matrix-Vector-Multiplication

V fits into main memory

- m_{ij} will be stored as a triple (i, j, m_{ij}) on the DFS
- v_j will be stored as a tuple (j, v_j) on the DFS

Application of MapReduce: Matrix-Vector-Multiplication

V fits into main memory

- m_{ij} will be stored as a triple (i, j, m_{ij}) on the DFS
- v_j will be stored as a tuple (j, v_j) on the DFS

- Map-Task loads V into main memory
- Gets a chunk of M: $(i_k, j_k, m_{i_k j_k}), ..., (i_l, j_l, m_{i_l j_l})$
- For each (i, j, m_{ij}) emits the tuple $(i, m_{ij} \cdot v_j)$

Application of MapReduce: Matrix-Vector-Multiplication

V fits into main memory

- m_{ij} will be stored as a triple (i, j, m_{ij}) on the DFS
- v_j will be stored as a tuple (j, v_j) on the DFS

- Map-Task loads V into main memory
- Gets a chunk of M: $(i_k, j_k, m_{i_k j_k}), \ldots, (i_l, j_l, m_{i_l j_l})$
- For each (i, j, m_{ij}) emits the tuple $(i, m_{ij} \cdot v_j)$

- Reducer gets input $(i, [m_{i0} v_0, \ldots, m_{in} v_n,])$
  $\Rightarrow$ Has all terms to compute the i-th value x_i of X

Application of MapReduce: Matrix-Vector-Multiplication

V does ***not*** fit into main memory

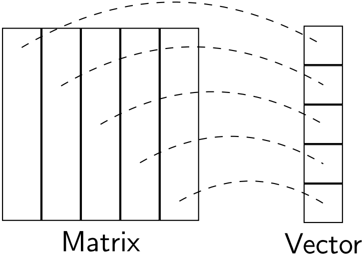


Figure: Striped matrix-vector-multiplication

Relational Algebra Operations

- Relational algebra operations can easily be implemented with the model of MapReduce

Example: $\sigma_C(R)$.

- **Mapper**: $\forall t \in R$, if $C(t) = \mathit{TRUE}$ emit (t, t).
- **Reducer**: Gets $(t, [t])$ as input, emits $(t, [t])$.

Related work:

- Pig[1]
- Hive[2]

[1] Gates *et al.*, "Building a High-level Dataflow System on Top of Map-Reduce: The Pig Experience"
[2] Thusoo *et al.*, "Hive - a petabyte scale data warehouse using Hadoop"

Summary

We have seen a few applications of MapReduce:

- WordCount
- Matrix-Vector-Multiplication
- Relational Algebra

How to optimize MapReduce jobs?

Communication Costs

- Increase performance $\hat{=}$ reducing communication between nodes
- Communication costs for a MapReduce job := size of input
- Concatenating MR-Jobs is possible:
  - Output of one MR-Job is input to the next one
  - Com. costs of a MR-Network is the sum of the inputs of each MR-Job

$$\boxed{input} \longrightarrow Job_1 \longrightarrow Job_2 \longrightarrow \cdots \longrightarrow Job_n \longrightarrow \boxed{output}$$

Communication Costs

Example: Natural Join

Relations:

R

A	B
a_1	b_1
a_2	b_2

S

B	C
b_2	c_1
b_3	c_2

- $|R| = r$, $|S| = s$
- Map-Tasks get chunks of R and S $\Rightarrow$ input-size: $r + s$

Communication Costs

Example: Cascaded Two-Way Joins

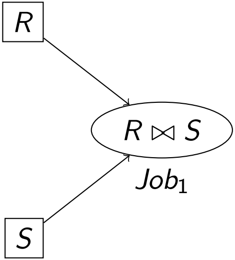


Figure: Schematic of cascaded two-way joins

Communication Costs

Example: Cascaded Two-Way Joins

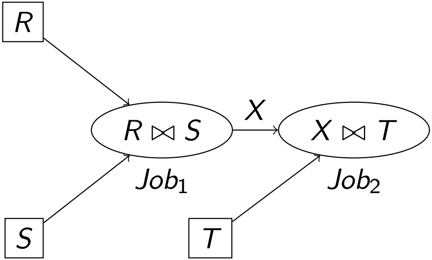


Figure: Schematic of cascaded two-way joins

Communication Costs

Example: Cascaded Two-Way Joins

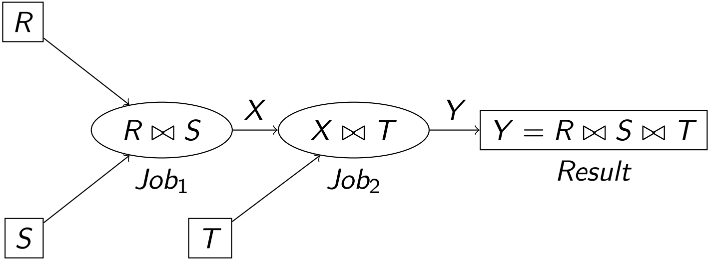


Figure: Schematic of cascaded two-way joins

Communication Costs

Example: Cascaded Two-Way Joins

Relations:

R

A	B
a_1	b_1
a_2	b_2

S

B	C
b_2	c_1
b_3	c_2

T

C	D
c_1	d_1
c_3	d_2

- R, S and T are of size r, s and t
- Cascaded two-way joins: $R \bowtie S \bowtie T$
- $p_b :=$ prob. that $tuple_R.B = tuple_S.B$
- $p_c :=$ prob. that $tuple_S.C = tuple_T.C$
- assume $p_b = p_c =: p$

Communication Costs

Example: Cascaded Two-Way Joins

$(R \bowtie S) \bowtie T$

- Input size of $R \bowtie S$: $r + s$
- Output size: $|R \bowtie S| \in \mathcal{O}(p \cdot r \cdot s)$
- Input size of $(R \bowtie S) \bowtie T$: $p \cdot r \cdot s + t$
- Com. costs $\in \mathcal{O}(r + s + t + p \cdot r \cdot s)$

Communication Costs

Example: Cascaded Two-Way Joins

$(R \bowtie S) \bowtie T$

- Input size of $R \bowtie S$: $r + s$
- Output size: $|R \bowtie S| \in \mathcal{O}(p \cdot r \cdot s)$
- Input size of $(R \bowtie S) \bowtie T$: $p \cdot r \cdot s + t$
- Com. costs $\in \mathcal{O}(r + s + t + p \cdot r \cdot s)$

$R \bowtie (S \bowtie T)$

- Input size of $S \bowtie T$: $s + t$
- Output size of $|S \bowtie T| \in \mathcal{O}(p \cdot s \cdot t)$
- Input size of $R \bowtie (S \bowtie T)$: $r + p \cdot s \cdot t$
- Com. costs $\in \mathcal{O}(r + s + t + p \cdot s \cdot t)$

Communication Costs

Example: Cascaded Two-Way Joins

$(R \bowtie S) \bowtie T$

- Input size of $R \bowtie S$: $r + s$
- Output size: $|R \bowtie S| \in \mathcal{O}(p \cdot r \cdot s)$
- Input size of $(R \bowtie S) \bowtie T$: $p \cdot r \cdot s + t$
- Com. costs $\in \mathcal{O}(r + s + t + p \cdot r \cdot s)$

$R \bowtie (S \bowtie T)$

- Input size of $S \bowtie T$: $s + t$
- Output size of $|S \bowtie T| \in \mathcal{O}(p \cdot s \cdot t)$
- Input size of $R \bowtie (S \bowtie T)$: $r + p \cdot s \cdot t$
- Com. costs $\in \mathcal{O}(r + s + t + p \cdot s \cdot t)$

Complexity Theory for MapReduce

Reducer Size and Replication Rate

Reducer size q

- Upper bound on value list for the reducer
- Low reducer size:
  - High degree of parallelism
  - Execute completely in main memory (avoid thrashing)

Complexity Theory for MapReduce

Reducer Size and Replication Rate

Reducer size q

- Upper bound on value list for the reducer
- Low reducer size:
  - High degree of parallelism
  - Execute completely in main memory (avoid thrashing)

Replication rate r

- $r = \frac{\#((k,v)_{out})}{\#((k,v)_{in})}$ of the mapper
- Low replication rate:
  - Reduces communication costs

Complexity Theory for MapReduce

Example: Similarity Joins

- Input:
  - 1 Million images
  - 1 MB each
- Goal: find similar pictures
  - $sim(P_i, P_j) > b$

Complexity Theory for MapReduce

Example: Similarity Joins

- Input:
  - 1 Million images
  - 1 MB each
- Goal: find similar pictures
  - $sim(P_i, P_j) > b$

Naive approach:

- **Mapper:** $\forall (i, P_i)$: emit $(\{i, j\}, P_i), \forall j \in \{1, 2, \ldots, 10^6\} \backslash \{i\}$
- **Reducer:** Gets $(\{i, j\}, [P_i, P_j])$, emits $(\{i, j\}, sim(P_i, P_j))$

Complexity Theory for MapReduce

Example: Similarity Joins

- Reducer size $q = 2MB$ (good!)
- Replication rate $r = 999.999$ (unbearable!)
- Amount of data to send along the cluster:

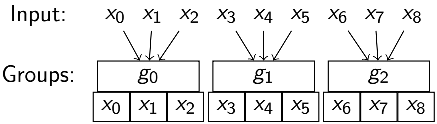
$$1.000.000MB \times 999.999 \times 1.000.000 \approx 10^{18} Bytes$$

- Takes approx. 300 years with gigabit-Ethernet

Complexity Theory for MapReduce

Example: Similarity Joins

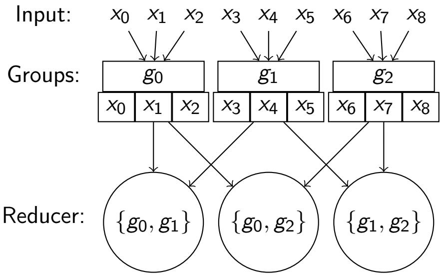
- Group pictures into g groups with $10^6/g$ pictures each



Complexity Theory for MapReduce

Example: Similarity Joins

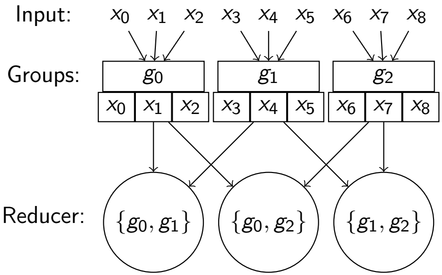
- Group pictures into g groups with $10^6/g$ pictures each
- **Mapper:**
  - Gets (i, P_i)
  - Emits $g-1$ key-value pairs: $(\{u, v\}, (i, P_i))$



Complexity Theory for MapReduce

Example: Similarity Joins

- Group pictures into g groups with $10^6/g$ pictures each
- **Mapper:**
  - Gets (i, P_i)
  - Emits $g-1$ key-value pairs: $(\{u,v\},(i,P_i))$
- **Reducer:**
  - Gets key $\{u,v\}$ with list $[(k,P_k)]$ of size $2\times 10^6/g$
  - $\forall (i,P_i),(j,P_j)$ with $group(i)\neq group(j)$, emits $(\{i,j\}, sim(P_i,P_j))$



Complexity Theory for MapReduce

Example: Similarity Joins

- Reducer size $q = 2GB$ (still good!)
- Replication rate $r = 999$ (bearable!)
- Amount of data to send along the cluster:

$$1.000.000MB \times 999 \times 1.000.000 \approx 10^{15}Bytes$$

- Takes approx. 3.6 Months with gigabit-Ethernet

Disadvantages of MapReduce

- Strict workflow
  - Algorithms needs to be translated into *map* and *reduce*
  - Building more complex workflows needs to be transformed into cascading MR-Jobs
- Iterative tasks generate high I/O overhead

$\Rightarrow$ Spark[3] captures these disadvantages

[3] Zaharia *et al.*, "Spark: Cluster Computing with Working Sets"

Spark: Cluster Computing with Working Sets

Introduction

What is Spark?

- Compute cluster framework
- Introduces dataset abstractions (RDDs)
- Ensures data integrity by lazy evaluation

Advantages over MapReduce:

- Efficient in iterative tasks
- Dynamic workflow

Resilient Distributed Datasets (RDDs)

file:

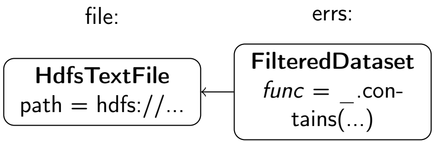
HdfsTextFile
path = hdfs://...

How to create RDDs?

- From a file

```
val file = spark.textFile("hdfs://...")
```

Resilient Distributed Datasets (RDDs)

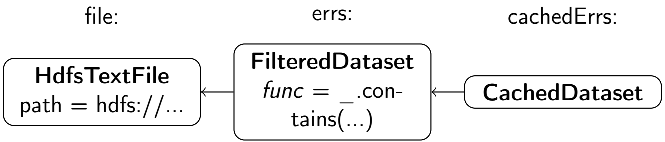


How to create RDDs?

- From a file
- From transformations

```
val file = spark.textFile("hdfs://...")
val errs = file.filter(_.contains("
ERROR"))
```

Resilient Distributed Datasets (RDDs)

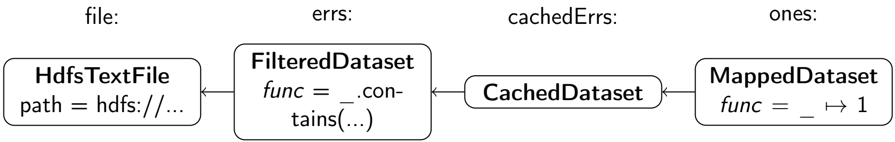


How to create RDDs?

- From a file
- From transformations
- By persistence changes

```
val file = spark.textFile("hdfs://...")
val errs = file.filter(_.contains("ERROR"))
val cachedErrs = errs.cache()
```

Resilient Distributed Datasets (RDDs)

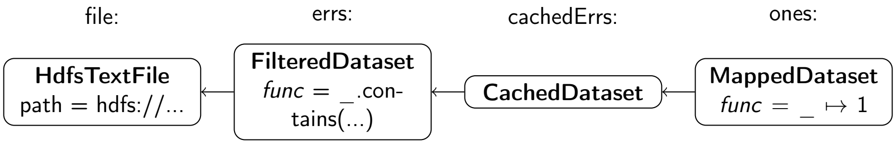


How to create RDDs?

- From a file
- From transformations
- By persistence changes
- By parallelizing

```
val file = spark.textFile("hdfs://...")
val errs = file.filter(_.contains("ERROR"))
val cachedErrs = errs.cache()
val ones = cachedErrs.map(_ => 1)
```

Resilient Distributed Datasets (RDDs)



How to create RDDs?

- From a file
- From transformations
- By persistence changes
- By parallelizing

```
val file = spark.textFile("hdfs://...")
val errs = file.filter(_.contains("ERROR"))
val cachedErrs = errs.cache()
val ones = cachedErrs.map(_ => 1)
val count = ones.reduce(_+_)
```

Resilient Distributed Datasets (RDDs)

Properties of **RDDs**:

- Lazyily evaluated
- Partitions can be recomputed on node failures
- Immutable collection of objects
- Distributed over different nodes

Performance Results

- Spark outperforms Hadoop[4] (Apache MR Framework) on iterative jobs (Example: Factor x20)

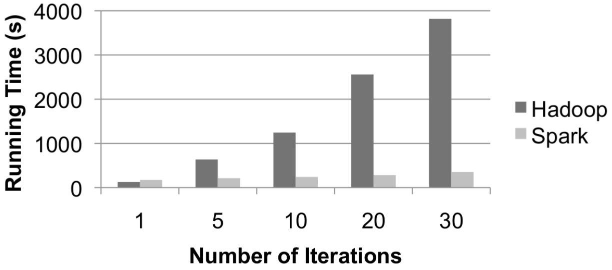


Figure: Logistic regression performance in Hadoop and Spark[5]

[4] *Apache Hadoop*

[5] Zaharia *et al.*, "Spark: Cluster Computing with Working Sets"

Conclusion

Covered topics:

- Compute clusters
- Distributed File System
- MapReduce
- Spark

Outlook:

- Massive data stream mining in real time: MOA[6]
- Scavanger[7]

---

[6] Bifet *et al.*, "MOA: Massive Online Analysis"

[7] Tyukin, Kramer, and Wicker, "Scavenger - A Framework for the Efficient Evaluation of Dynamic and Modular Algorithms"

References

- *Apache Hadoop*. http://hadoop.apache.org/. Accessed: 2016-07-14.
- Albert Bifet *et al.* "MOA: Massive Online Analysis". In: vol. 11. JMLR.org, Aug. 2010, pp. 1601–1604. URL: http://dl.acm.org/citation.cfm?id=1756006.1859903.
- Alan F. Gates *et al.* "Building a High-level Dataflow System on Top of Map-Reduce: The Pig Experience". In: vol. 2. 2. VLDB Endowment, Aug. 2009, pp. 1414–1425. DOI: 10.14778/1687553.1687568. URL: http://dx.doi.org/10.14778/1687553.1687568.
- Andreas Hildebrandt. *Big Data: Algorithms, methods and techniques for analysing massive datasets*. JGU Mainz.
- Ashish Thusoo *et al.* "Hive - a petabyte scale data warehouse using Hadoop". In: *ICDE*. Ed. by Feifei Li *et al.* IEEE, 2010, pp. 996–1005. ISBN: 978-1-4244-5444-0. URL: http://infolab.stanford.edu/~ragho/hive-icde2010.pdf.