Distributed Data Management
Summer Semester 2013
TU Kaiserslautern

Dr.-Ing. Sebastian Michel

smichel@mmci.uni-saarland.de
Lecture 2+

MAP REDUCE
Lots of Data

• E.g. Google:
  – Billions of Websites
    (around 50 billion, Spring 2013)
  – TBs of data

• And not only websites:
  – Videos (Youtube), images,
    user profiles, email

http://flickr.com/photos/jurvetson/157722937/
Scale-out

• Many machines (hundreds, thousands)

• Commodity machines
  – Cheap but not super reliable
  – Anyway, there are so many, failures will happen!
    • 1 server fails, say, once a year.... Imagine you have 10,000 machines
Map and Reduce: Key Idea

• Spread task of processing data on machines
• According to map and reduce rules/functions
• No need to deal with node failures, load balancing, etc. System takes care of this.

• **Map phase**: Data is put to a number of machines. Output is partitioned (sorted) by a key (e.g., a term)
• **Reduce**: For each key-group, data is aggregated (reduced)
Map Reduce from High Level

DATA

MAP

Intermediate Results

MAP

REDUCE

Result

MAP

REDUCE

Result

MAP

REDUCE

Result

D

Result

A

Result

T

Result

A

Result

Distributed Data Management, SoSe 2013, S. Michel
Brief History of MapReduce

• First described in an article in 2004.
  – MapReduce paradigm and how it is used in Google (Google file system, etc.)

• Many MapReduce implementations
• Hadoop is arguable the most prominent one
• Will look at MR in general and Hadoop specifically
Architectural Issues

• Data lies in a distributed file system
• Block based, big chunks (usually 64MB or 128MB)
• Chunks are replicated and distributed over machines

• If possible, data processing is moved to data hosting machines.
Functional Programming: Map

Expression : map

Of type : (a -> b) -> [a] -> [b]

Definition:

• map f [] = []

• map f (x:xs) = f x : map f xs

Example (using Hugs98 Haskell):

• map (\x-> x*x) [1,2,3,4] → [1,4,9,16]
Map

Observation:

Execution of function f can be done fully in parallel!

Then: Output is aggregated (reduced).
Functional Programming: Reduce (aka. fold)

Expression: \texttt{foldl} (note: there is also \texttt{foldr}=right)

Of type: \((\texttt{a} \rightarrow \texttt{b} \rightarrow \texttt{a}) \rightarrow \texttt{a} \rightarrow \texttt{[b]} \rightarrow \texttt{a}\)

Definition:

\begin{itemize}
  \item \texttt{foldl} \texttt{f} \texttt{z} \texttt{[]} = \texttt{z}
  \item \texttt{foldl} \texttt{f} \texttt{z} (\texttt{x}:\texttt{xs}) = \texttt{foldl} \texttt{f} (\texttt{f} \texttt{z} \texttt{x}) \texttt{xs}
\end{itemize}

Example:

\begin{itemize}
  \item \texttt{foldl} (+) 0 \texttt{[1,2,3,4,5]} \rightarrow \texttt{15}
\end{itemize}
Note on “Functional Programming”

• what was commonly restricted to functional prog. languages is getting more and more “standard”

• Python, Ruby, Scala (Java++), Clojure, C#, C++ (11)

• E.g., Ruby:
  
  \[
  [1,2,3,4,5].map{|x| x**2 } => [1, 4, 9, 16, 25]
  \]

  \[
  [1,2,3,4,5].inject(0){|x,a| x+a} => 15
  \]
Going Distributed: Key Principle

• Many data chunks
• Map function on each of the chunks
• Map process outputs data with keys
  => Partitions based on keys
• Aggregate (fold/reduce) mapped data per key

• E.g., count number occurrences of each terms in set of documents.
Map and Reduce: Types

- Map \((k1,v1) \rightarrow \text{list}(k2,v2)\)
- Reduce\((k2, \text{list}(v2)) \rightarrow \text{list}(k3, v3)\)

- For instance:
  - \(k1=\) document identifier
  - \(v1=\) document content
  - \(k2=\) term
  - \(v2=\) count
  - \(k3=\) term
  - \(v3=\) final count

(keys allow grouping data to machines/tasks)
Move Computation to Data

• Data is stored in a **distributed file system** (for Google: GFS=Google File System)
• Large chunks (**blocks**)
• **Master** node of GFS knows locations
• Can/should! **initiate computation** at such nodes
Computation (Workflow)

- A master node controls computation
  - this is where you submit your job (task) to
  - computes necessary map and reduce tasks
  - selects and activates worker nodes

- Worker node
  - for map; selected if possible close to data
  - reduce; consumed intermediate results and creates final output
Example: Grep

- Given: file
- Want: all lines that contain certain pattern

- Map(String key, String value)
  ```java
  if value.contains(pattern):
    emit(value, "")
  ```

This is a **map only** task (no reducer; no grouping by key): output is written directly to distributed file system
MapReduce: Example Map + Count

• Data Part 1
  – “One ring to rule them all, one ring to find them,

• Data Part 2
  – “One ring to bring them all and in the darkness bind them.”
**Map Line to Terms and Counts**

**Line 1**

```
{"one"=>["1", "1"],
 "ring"=>["1", "1"],
 "to"=>["1", "1"],
 "rule"=>["1"],
 "them"=>["1", "1"],
 "all"=>["1"],
 "find"=>["1"]}
```

**Line 2**

```
{"one"=>["1"],
 "ring"=>["1"],
 "to"=>["1"],
 "bring"=>["1"],
 "them"=>["1", "1"],
 "all"=>["1"],
 "and"=>["1"],
 "in"=>["1"],
 "the"=>["1"],
 "darkness"=>["1"],
 "bind"=>["1"]}
```
Group by Term

{"one"=>["1", "1"],
"ring"=>["1", "1"],
....

{"one"=>["1"],
"ring"=>["1"],
...

{"one"=>[["1","1"],["1"]],
"ring"=>[["1","1"],["1"]],
...

Distributed Data Management, SoSe 2013, S. Michel
Sum Up

```json
{"one"=>["1","1"],["1"],
 "ring"=>["1","1"],["1"],
 ...

{"one"=>["3"],
 "ring"=>["3"],
 ...
```
Example: Wordcount

Map(String key, String value)
  for each word w in value:
    emit(w, 1)

Reduce(String key, Iterator values)
  int result=0
  for each v in values:
    result += v
  emit(result)

Note: depends also in which context you want to count, e.g.,
- overall occurrences of word in collection
- or number of documents in which word occurs
- or number of sentences in collection where word occurs
- ...
Example: Inverted Index

• Given: set of documents
• Want: A -> list of document ids in which A occurs, for each term A

• How can this be computed in MapReduce?
Example: Inverted Index

• Why useful?
  – Consider Google-style query: A B C
  – How to find relevant documents? Parse through all? No.
  – Which documents are relevant for the result? Check (pre-computed inv. index):
Example: Co-occurrences

• Given: text file
• Want: for terms a, b, how often does a and b occur close together, e.g., within sentence?
• That is, output = ([a,b], count)

• How can this be computed?
Example: Co-occurrences (Cont’d)

• **Solution 1: pairs approach**
  – mapper for string s:
    • **for all** term pairs (a,b) in s: emit({a,b}, 1)
  – reducer just aggregates counts

• **Solution 2: “stripes” approach**
  – mapper for string s:
    • **collect all** t_i that co-occur with a
      • emit (a,{t_1, t_2, .... t_n})
  – reducer aggregates