Map Reduce / Workflow Systems I

Alexander Schönhuth

UNIVERSITÄT
BIELEFELD
Faculty of Technology

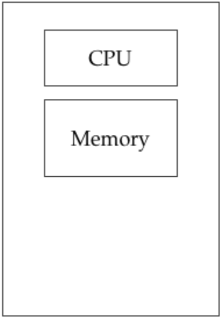
Bielefeld University
May 6, 2021

Learning Goals Today

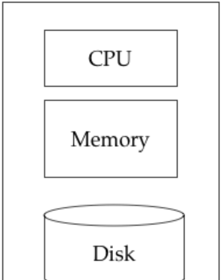
- Understand the technical challenges of parallelism / multi-node computation
- Understand the *MapReduce* paradigm
- Understand how to put the paradigm into effect in practice
- Understand the fundamental algorithms supported by MapReduce

Map Reduce: Introduction

MapReduce: Motivation I



Machine Learning, Statistics



Machine Learning, Statistics

"Classical" Data Mining

MapReduce: Motivation II

- Need to manage massive amounts of data quickly
- Within one particular application, data is massive
  - For example (web searches), even with high performance disk read bandwidth, just reading 10 billion web pages requires several days
- But operations can be very regular (do the same thing to each web page) ☞ exploit the parallelism
  - Many operations on databases (as supported by SQL, for example) can and need to be parallelized
  - Ranking web pages (*"PageRank"*) requires iterated multiplication of matrices with dimensions in the billions
  - Searching for "friend networks" in social networks require operations on graphs with billions of nodes and edges

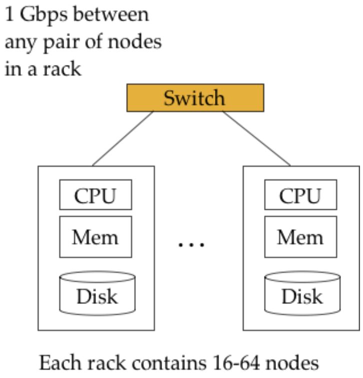
MapReduce: Motivation II

- New software stack: get parallelism not from single supercomputer, but from computing clusters
  - *First*, need to deal with storing data
    ☞ Distributed file systems (hardware based issues/solutions)
  - *Second*, new higher-level programming systems required
    ☞ *MapReduce*
  - *Third*, MapReduce reflects early attempts: ☞ More sophisticated *workflow systems*
- Here, we will deal predominantly with MapReduce first
- We will also consider most advanced workflow systems
- *Reminder:* it's about *analytics* in this course

MapReduce: Motivation III

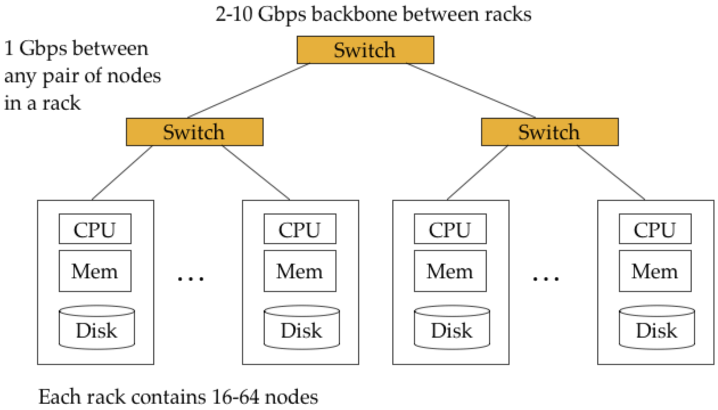
- MapReduce enables convenient execution of parallelizable operations on compute clusters and clouds
- MapReduce executes such operations in a *fault-tolerant* manner
- MapReduce is the origin of more general ideas
  - Systems supporting *acyclic workflows* in general
  - Systems supporting *recursive operations*

MapReduce: Motivation III



Adopted from `mmds.org`

MapReduce: Motivation III



Adopted from `mmds.org`

Distributed File Systems

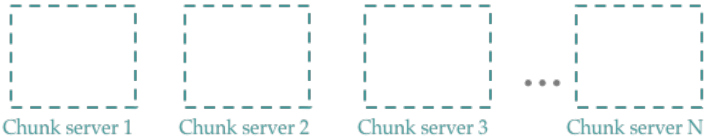
Distributed File Systems: Challenges and Characteristics

- *Node Failure:* Single nodes fail (e.g. by disk crash) or entire racks can fail (e.g. by network failure)
  ☞ no starting over every time: back up data
- *File Size:* can be huge
  ☞ how to distribute them?
- *Computation Time:* should not be dominated by input/output
  ☞ data should be as close as possible to compute nodes
- *Data:* does not change, new data only makes small appends
  ☞ otherwise DFS not suitable

Distributed File Systems: Summary

- Data is divided into *chunks* (usually of size 64 MB)
- Chunks are replicated (3 times is common)
- Chunk copies are distributed across the nodes
- A file called *master node* keeps track of where chunks went
- A *client library* provides file access; talks to master and connects to individual servers
- *Examples of DFS Implementations:*
  - *Google File System (GFS):* the original
  - *Hadoop Distributed File System (HDFS):* open source, used with Hadoop, a MapReduce implementation
  - *Colossus:* supposed to be an improvement over GFS; little has been published

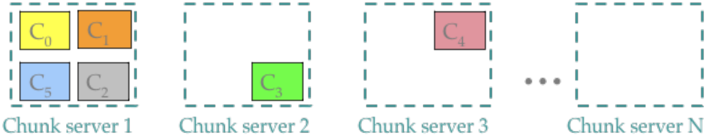
Distributed File Systems: Mode of Operation



Adopted from `mmds.org`

- Chunk servers correspond to nodes in racks

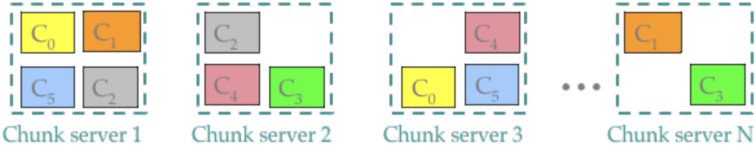
Distributed File Systems: Mode of Operation



Adopted from `mmds.org`

- One file ("File C") in 6 chunks, C0, C1, C2, C3, C4, C5

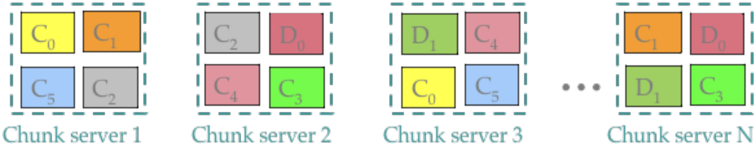
Distributed File Systems: Mode of Operation



Adopted from `mmds.org`

- Replicating each chunk twice and putting copies to different nodes prevents damage due to failure

Distributed File Systems: Mode of Operation



Adopted from `mmds.org`

- Fill servers up; computations are carried out immediately by chunk servers

Map Reduce: Workflow

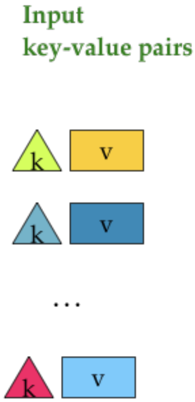
MapReduce: Workflow

1. Chunks are assigned to Map tasks, which turn each chunk into sequence of *key-value* pairs.
   - Key-value pair generation is specified by user
2. *Master controller* (automatic):
   - Key-value pairs are collected
   - Key-value pairs are sorted
   - Keys are divided among Reduce tasks
3. Reduce tasks combine values into final output
   - Reduce tasks are specified by user
   - Reduce tasks work on one key at a time

MapReduce: Running Example

- *Input:* One, or several huge documents
- *Desired Output:* Counts of all words appearing in the documents
- *Applications:*
  - Detecting plagiarism
  - Determining words characterizing documents for web searches
- Important: In the example, distinguish between
  - *Input key-value pairs* that reflect id-file pairs
  - *Intermediate key-value pairs* that reflect key-value pairs from Map tasks, as seen in the slide before
  - The latter ones are important for MapReduce

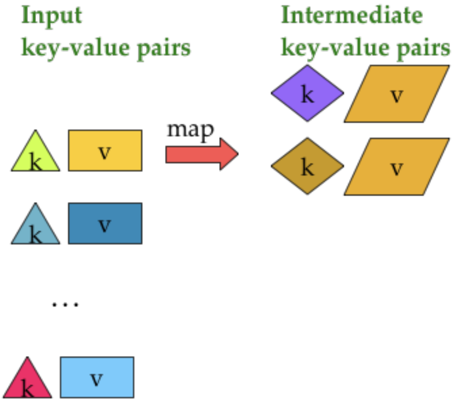
MapReduce: Map



Here, input key-value pairs refer to id-file (id-document) pairs

Adopted from `mmds.org`

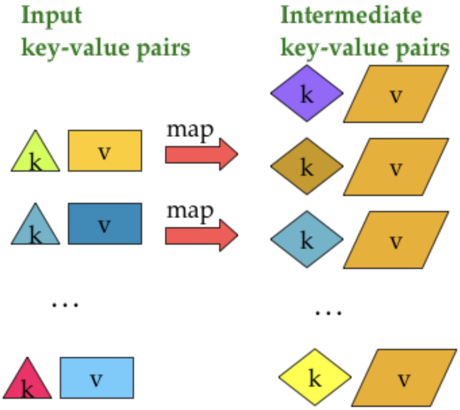
MapReduce: Map



Intermediate key-value pairs are the ones to be generated by a Map task

Adopted from `mmds.org`

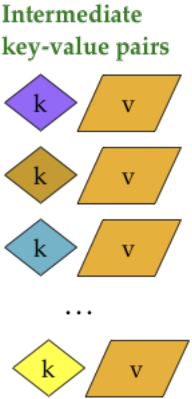
MapReduce: Map



Here: intermediate key-value pairs correspond to <'word',1> tuples

Adopted from `mmds.org`

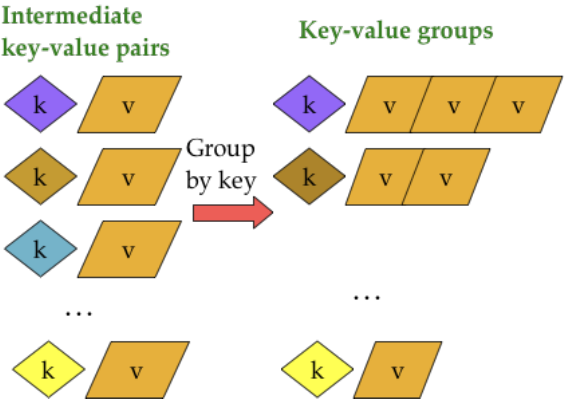
MapReduce: Reduce



Intermediate key-value pairs (<'word',1> tuples) generated by Map

Adopted from `mmds.org`

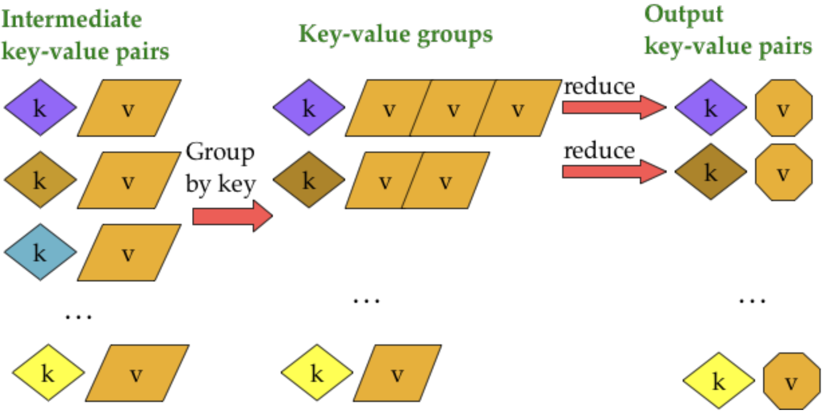
MapReduce: Reduce



Intermediate key-value pairs generated by Map

Adopted from `mmds.org`

MapReduce: Reduce



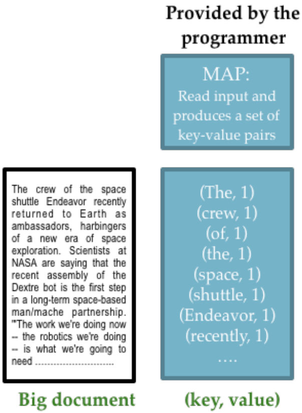
Output key-value pairs generated by Reduce: here <'word',count> tuples

Adopted from `mmds.org`

MapReduce: Formal Summary

- *Input:* A set of (key, value)-pairs $< k, v >$
  - $< k, v >$ usually correspond to file (v) and id (k) of the file
- *To be provided by programmer:*
  - $Map(< k, v >) \rightarrow < k', v' >^*$
    - Maps input pair $< k, v >$ to multi-set of key-value pairs $< k', v' >$
    - $< k', v' >$ is intermediate key-value in schematic on slides before
    - One Map call for each input key-value pair $< k, v >$
  - $Reduce(< k', v' >^*) \rightarrow < k', v'' >^*$
    - For each key k' all key-value pairs $< k', v' >$ are reduced together
    - One Reduce call for each unique key k'

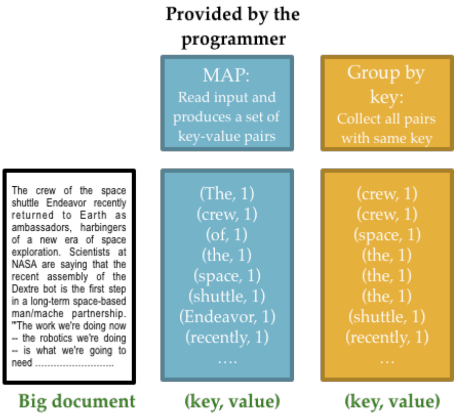
MapReduce Example: Word Counting



Intermediate key-value pairs correspond to <’word’,1> tuples

Adopted from `mmds.org`

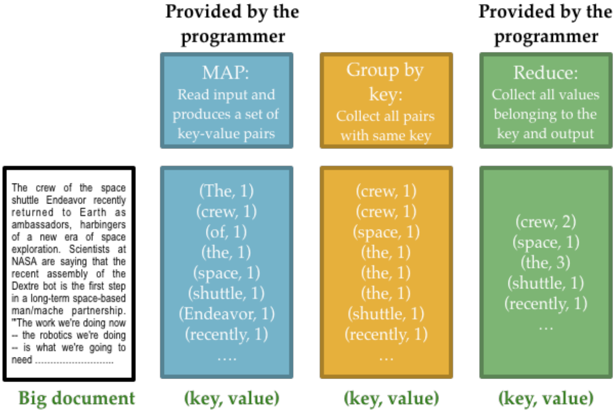
MapReduce Example: Word Counting



Intermediate key-value pairs are sorted and hashed by key (automatic)

Adopted from `mmds.org`

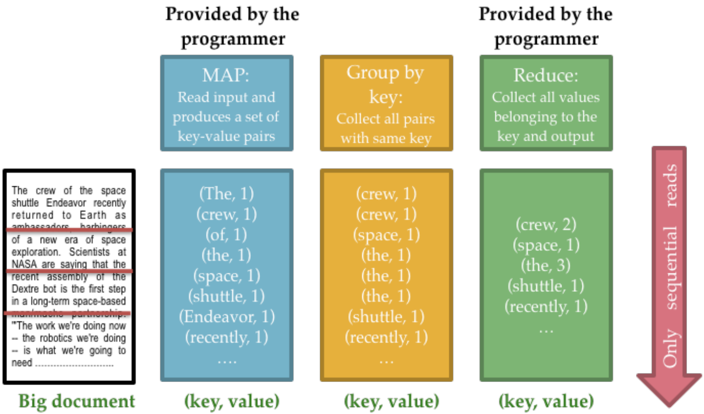
MapReduce Example: Word Counting



Reduce sums up all values for each key

Adopted from `mmds.org`

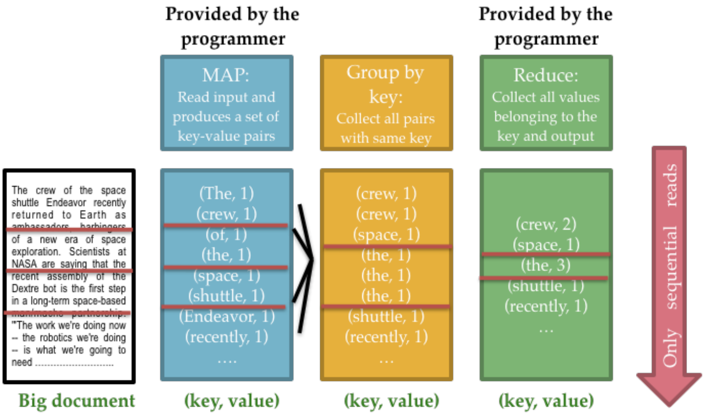
MapReduce Example: Word Counting



Map tasks are parallelized across nodes: one Map per chunk

Adopted from `mmds.org`

MapReduce Example: Word Counting



Reduce tasks are parallelized across nodes: one Reduce for a subset of keys

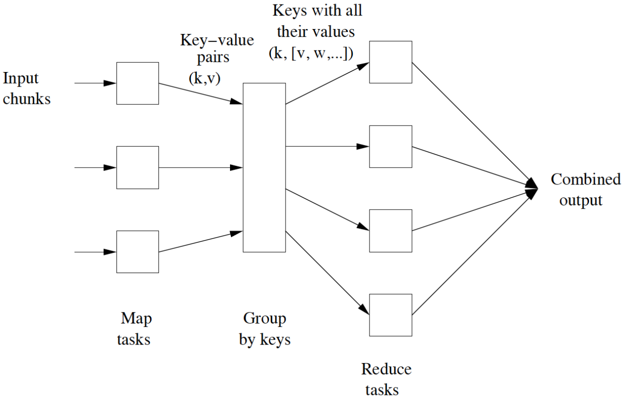
Adopted from `mmds.org`

EXAMPLE: WORD COUNTING CODE

```
map(key, value)
// key:  document name, value:  text of document
  foreach word w in value:
    emit(w,1)

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

MapReduce: Workflow Summary



Summary

Here $< k, v >$ refers to intermediate key-value pair earlier

Upon sorting key-value pairs are hashed

Adopted from `mmds.org`

Map Reduce: Execution

MapReduce: Host Size Example

- *Input:* Large web corpus with metadata file
  - Metadata file has entries: (URL, size, date,...)
- Would like to determine size for each host, which may encompass several URL's

- *Map:* For each entry, key-value pair: $< host(URL), size >$
- *Reduce:* Add up sizes for each host

MapReduce: Language Example

- *Input:* Many (possibly large) documents
- *Goal:* Count all 5-word sequences

- *Map:* Extract $< 5 - word - sequence, 1 >$ as key-value pairs
- *Reduce:* Add up counts across 5-word-sequence keys: *several such keys per document*

MapReduce: Language Example II

- *Input:* Many (possibly large) documents
- *Goal:* Count all 5-word sequences

- *Alternative Map:* Extract $< 5 - word - sequence, count >$ from each document, where *count* refers to number of appearances of 5-word-sequence in one document)
- *Alternative Reduce:* Add up counts across 5-word-sequence keys: *one key per document*

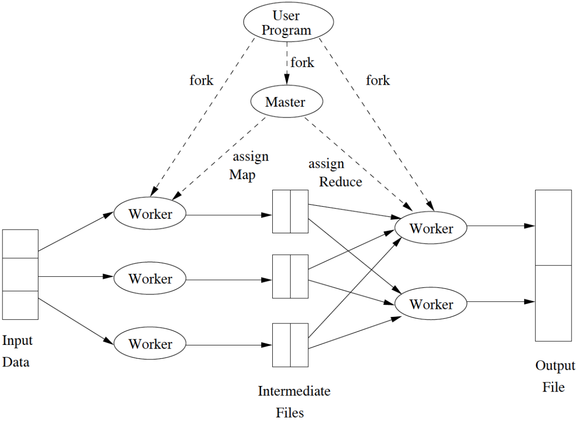
MAPREDUCE: COMBINERS

- The 'Alternative Map' is a strategy when Reduce tasks are associative
- In that case, some of the Reduce work can already done in the Map step
  - Adding is associative and commutative:

$$(a + b) + c = a + (b + c)$$
$$a + b = b + a$$

  - So, the Map task can generate $< key, count >$ per document instead of just *count* times many $< key, 1 >$ key-value pairs
- *Skew:* Runtime needed by Reduce tasks can vary substantially
  - Random assignment of keys to Reduce tasks balances out skew
  - Using more Reduce tasks than nodes leads to balanced work load per node

MapReduce: Execution



Execution of MapReduce program: overview

Adopted from `mmds.org`

MapReduce: Execution

- User needs to choose number of Map and Reduce tasks
  - One Map task per data chunk (so many more than nodes)
  - Less Reduce tasks: keep number of intermediate files low
  - One Master node
- Master keeps track of status of tasks (idle, in process, completed)
- Worker process reports to Master when finished; gets assigned a new task
- Master keeps track of location and sizes of files
- *Node Failures:*
  - When Worker nodes fail, Master reassigns tasks to other nodes
  - When Master node fails, entire process needs to be restarted

Map Reduce: Algorithms

MapReduce: Algorithms

- MapReduce does not necessarily cater to every problem that profits from parallelization
  - *Example:* Online retail sales: searches for products, recording sales
  - Require little computation, but modify underlying databases
- *Original Purpose:* Multiplying matrices required for PageRank (Google)
  - Matrix-vector multiplication
  - Matrix-matrix multiplication
- *Databases:* Relational algebra operations
  - Selection, projection
  - Union, intersection, difference
  - Natural join

MapReduce: Matrix-Vector Multiplication I

Let $M = (m_{ij}) \in \mathbb{R}^{m \times n}, v = (v_1, ..., v_n) \in \mathbb{R}^n$, for (very) large m, n.
We would like to compute $Mv =: x = (x_1, ..., x_m) \in \mathbb{R}^m$

$$x_i = \sum_{j=1}^{n} m_{ij} v_j \tag{1}$$

Assumptions:

- M, v stored as files in DFS
- coordinates i, j of entries m_{ij} discoverable (e.g. possible through explicit storage (i, j, m_{ij}))
- coordinates j of entries v_j discoverable

MapReduce: Matrix-Vector Multiplication II

We would like to compute $Mv = x = (x_1, ..., x_m) \in \mathbb{R}^m$

$$x_i = \sum_{j=1}^{n} m_{ij} v_j \tag{2}$$

Map

1. Take in suitably sized chunk of M and (entire) v
2. Generate key-value pairs $(i, m_{ij} v_j)$

Reduce

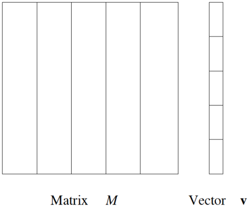
1. Sum all values of pairs with key i, yielding x_i

MapReduce: Matrix-Vector Multiplication III

We would like to compute $Mv =: x = (x_1, ..., x_m) \in \mathbb{R}^m$

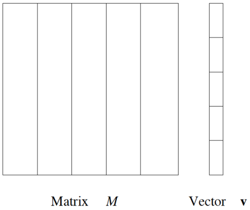
$$x_i = \sum_{j=1}^{n} m_{ij} v_j \tag{3}$$

Situation: Vector v too large to fit in main memory
Solution: Cut both M and v into stripes, process (chunks of) stripes



Adopted from `mmds.org`

MapReduce: Matrix-Vector Multiplication III



Adopted from `mmds.org`

Map

- Take in suitably sized chunk of stripe of M and stripe of v
- Generate key-value pairs $(i, m_{ij}v_j)$

Reduce

- Sum all values of pairs with key i, yielding x_i

MapReduce: Relational Algebras

MapReduce: Operations on large-scale data in database queries

- *Reminder: Relational Model*
  - A *relation* is a table with
  - column headers called *attributes*
  - rows called *tuples*
  - We write $R(A_1, A_2, ..., A_n)$ for a relation R with attributes $A_1, A_2, ..., A_n$

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

Relation *Links* (from `mmds.org`)

MapReduce: Relational Algebra Operations

- *Selection:* Apply condition C and select only tuples (rows) from R that satisfy C, denoted $\sigma_C(R)$
  - Choose only rows from R that refer to links leaving from or leading to a particular URL
- *Projection:* Choose a subset S of columns from R to generate new table $\pi_S(R)$
  - Generate table with only URL's that have incoming links

MapReduce: Relational Algebra Operations

Selection $\sigma_C(R)$

- **Map:** For each tuple t in R check whether C applies
  - If yes, generate key-value pair (t, t)
  - If not, do nothing
- **Reduce:** Reflects identity function, turns key-value pairs into output

Projection $\pi_S(R)$

- **Map:** For each tuple $t \in R$ compute tuple t' by removing attributes not from S. Generate key-value pair (t', t')
- **Reduce:** Two different t may turn into identical t', so there may be identical key-value pairs (t', t'), the system turns into $(t', [t', ..., t'])$ by grouping; output just (t', t'), yielding one key-value pair for each t'

MapReduce: Relational Algebra Operations

- *Union, Intersection, Difference:* Set operations applied to sets of tuples from two relations R and S
  - Imagine two tables, for links leaving from URL's in Europe and North America
  - Intersection: compute set of URL's that have incoming links from both Europe and North America
- *Natural Join:* Generate new table by joining tuples from two tables R and S when agreeing on attributes shared by two tables, yielding a new table $R \bowtie S$
  - Imagine two tables of links, one with links from Europe to Asia L_{EA}, and one from Asia to North America L_{AN}
  - Join two URL pairs when 'To' from first table agrees with 'From' from second table
  - This yields table $L_{EA} \bowtie L_{AN}$ with three columns

Relational Algebra Operations

Union, Intersection

- **Map:** For each tuple t from both R and S generate key-value pair (t, t)
- **Reduce:** After grouping, there will be two kinds of pairs: either $(t, [t])$ or $(t, [t, t])$
  - For *Union*, output everything
  - For *Intersection*, output (t, t) only for $(t, [t, t])$

Difference

- **Map:** For a tuple t in R, generate key-value pair (t, R), and for tuple t in S generate key-value pair (t, S) (use single bits for distinguishing R, S)
- **Reduce:** After grouping, three cases: $(t, [R])$, $(t, [R, S])$, $(t, [S])$. Output (t, t) only for $(t, [R])$

Relational Algebra Operations

Natural Join: $R(A,B) \bowtie S(B,C)$

- **Map:** For each tuple $t = (a,b)$ from R, generate key-value pair $(b,(R,a))$. For each tuple (b,c) from S, generate $(b,(S,c))$.
- **Reduce:** After grouping, each key value b has list of values being either of the form (R,a) or (S,c)
  - Construct all pairs of values where first component is like (R,a) and second component is like (S,c), yielding triples $(b,(R,a),(S,c))$
  - Turn triples into triples (a,b,c) being output

General Natural Join

Do like for relations with two attributes, by considering

- A attributes from R not in S
- B attributes both in R,S
- C attributes from S not in R

MapReduce: Matrix-Matrix Multiplication

Let $M = (m_{ij}) \in \mathbb{R}^{m \times n}, N = (n_{jl}) \in \mathbb{R}^{n \times k}$, for (very) large m, n, k.
We would like to compute $MN \in \mathbb{R}^{m \times k}$ where $(MN)_{il} = \sum_{j=1}^{n} m_{ij} n_{jl}$

- **Map:**
  - For each m_{ij}, generate all possible key-value pairs $((i, l), (M, j, m_{ij})$
  - For each n_{jl}, generate all possible key-value pairs $((i, l), (N, j, n_{jl})$
  - Thereby, M and N are stored by means of single bit
- **Reduce:** Need to work on list of values of keys (i, l):
  - Sort values [which are either (M, j, m_{ij}) or (N, j, n_{jl})] by j
  - After sorting, multiply each of two consecutive values m_{ij}, n_{jl}
  - Add up all the products

Remark: There are more efficient ways to multiply matrices using Natural Join (2.3.9)

Materials / Outlook

- See *Mining of Massive Datasets*, chapter 2.1–2.3
- As usual, see `http://www.mmds.org/` in general for further resources
- Next lecture: “Map Reduce / Workflow Systems II”
  - See *Mining of Massive Datasets* 2.4–2.6