# Lecture 6
# Map Reduce II

Alexander Schönhuth

UNIVERSITÄT BIELEFELD
Faculty of Technology

Bielefeld University

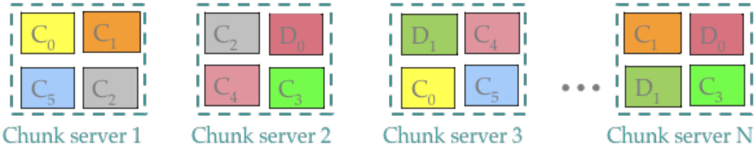
April 27, 2023

# Learning Goals Today

- Understand how to put the paradigm into effect in practice
- Understand the fundamental algorithms supported by MapReduce
- Get to know idea of workflow systems and some examples

***Map Reduce: Reminder***

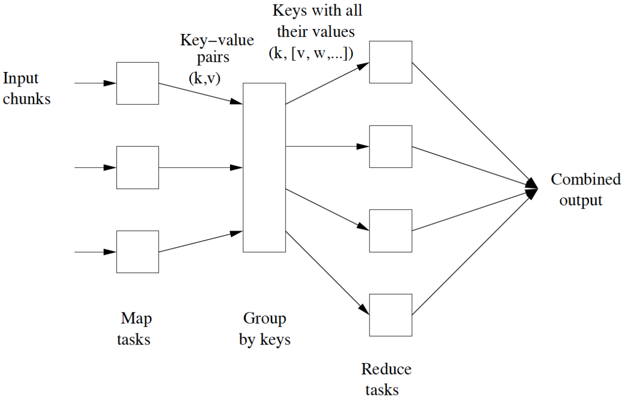
# DISTRIBUTED FILE SYSTEMS: MODE OF OPERATION




Adopted from `mmds.org`

- Replicating each chunk (at least) twice and putting copies to different nodes prevents damage due to failure
- Fill servers up; computations are carried out immediately by chunk servers

# 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`

# EXAMPLE: COUNTING WORDS IN DOCUMENTS

*Code for Map and Reduce tasks*

```
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)
```

*Map Reduce: Execution*

# MapReduce: Host Size Example

- *Input:* Large web corpus with metadata file
  - Metadata file has entries: (URL, size, date,...)
  - URL's belong to hosts; hosts may control several URL's
  - Host of URL can be determined
- Would like to determine size for each host
  - Size of a host is sum of the sizes of its 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 (= 1's) across 5-word-sequence keys
  - There may be several identical key-value pairs per document
  - ☞ number of appearances of 5-word-sequence in document

# MapReduce: Language Example II

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

- *Alternative Map:*
  - Generate only one $< 5 - word - sequence, count >$ per document
  - *count* is number of appearances of sequences in document
- *Alternative Reduce:*
  - Add up counts across 5-word-sequence keys
  - One key per document where value is count in document

# MapReduce: Combiners

- 'Alternative Map' reflects strategy for *associative* Reduce tasks
- In that case, some Reduce work can be performed in Map step
- Adding is associative and commutative:

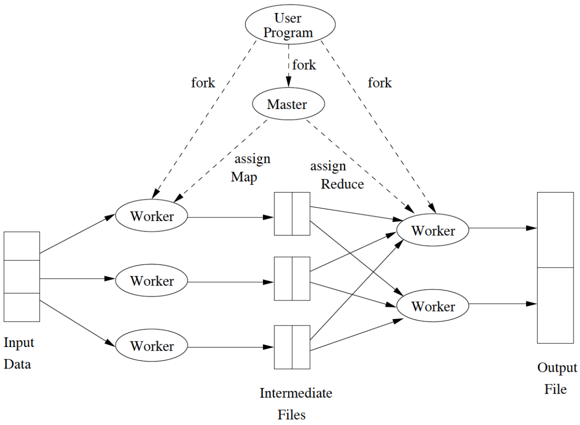
$$(a + b) + c = a + (b + c)$$
$$a + b = b + a$$

- So, the Map task can generate $< key, count >$ per document instead of just *count* times many $< key, 1 >$ key-value pairs

# MapReduce: Skew

- *Skew:* Runtime of Reduce tasks can vary substantially
- Runtime depends on number of key-value pairs
- Nodes have to carry out several Reduce tasks
- *Goal:* Achieve that runtime per node is similar
- *Strategy:* Random assignment of keys to Reduce tasks
  - Random assignment balances out skew
  - The more Reduce tasks, the more balanced by random assignment

# MapReduce: Execution




Execution of MapReduce program: overview

Adopted from `mmds.org`

# MapReduce: Execution

- User needs to design Map and Reduce tasks
  - One Map task per data chunk
    ☞ Each node holds several chunks
    ☞ Many more Map tasks than nodes
  - Varying Reduce tasks: control number of intermediate files
  - One Master node
- Master keeps track of status of tasks (idle, in process, completed)
- Worker signals Master termination; 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
  - MapReduce *never (!)* modifies original data (chunks themselves)
- *Original Purpose:* Multiplying matrices 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 $Mv = x$:

$$\begin{pmatrix} m_{11} & \dots & m_{1n} \\ \vdots & \ddots & \vdots \\ m_{mn} & \dots & m_{mn} \end{pmatrix} \times \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} = \begin{pmatrix} x_1 \\ \vdots \\ x_m \end{pmatrix} \in \mathbb{R}^m \tag{1}$$

that is

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

for each $i = 1, ..., m$.

# 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 $Mv =: x = (x_1, ..., x_m) \in \mathbb{R}^m$

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

Assumptions:

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

# MapReduce: Matrix-Vector Multiplication II

Compute $x_i = \sum_{j=1}^{n} m_{ij} v_j$ for each $i = 1, ..., m$

**Map**

1. Take in suitably sized chunk of $M$ and (entire) $v$
   - Chunk of $M$ = horizontal slice of $M$:

$$\begin{pmatrix} m_{i_1 1} & \dots & m_{i_1 n} \\ \vdots & \ddots & \vdots \\ m_{i_2 1} & \dots & m_{i_2 n} \end{pmatrix} \tag{4}$$

     that is, submatrix of $M$ on subset of rows $1 \leq i_1 < i_2 \leq m$
   - Processing chunk enables computation of $x_i, i = i_1, ..., i_2$
2. Generate key-value pairs

$$(i, m_{ij} v_j) \quad \text{for} \quad i_1 \leq i \leq i_2, 1 \leq j \leq n$$

# MapReduce: Matrix-Vector Multiplication II

Compute $x_i = \sum_{j=1}^{n} m_{ij} v_j$ for each $i = 1, ..., m$

**Map**

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

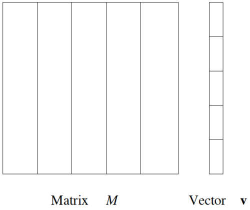
**Reduce**

1. Sum all values of pairs with key $i$
2. When processing chunk with $i = i_1, ..., i_2$, yields $x_i, i = i_1, ..., i_2$

# MapReduce: Matrix-Vector Multiplication III

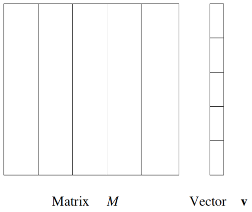
Compute $x_i = \sum_{j=1}^{n} m_{ij} v_j$ for each $i = 1, ..., m$

**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, To)*

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
  - *Example:* Choose only rows leading to 'url3'
  - Yields smaller subtable as a result
- *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
    ☞ Project to 'To' column
  - Resulting table has only one column
    ☞ All URL's in one-column table have link from other URL

# 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
  - *Example:* Selecting rows leading to 'url3'
    ☞ Generate tuples $((url1, url3), (url1, url3))$ and $((url2, url3), (url2, url3))$
- **Reduce:** Reflects identity function, turns key-value pairs into output

# MapReduce: Relational Algebra Operations

**Projection** $\pi_S(R)$

- **Map:** For each tuple $t \in R$ compute tuple $t'$ by removing attributes not from $S$. Generate key-value pair $(t', t')$
  - *Example:* Project to 'To' column
    ☞ Generate pairs
    $((url2), (url2)), ((url3), (url3)), ((url3), (url3)), ((url4), (url4))$
- **Reduce:** Two different $t$ may turn into identical $t'$ (example: 'url3'), 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** $R(A, B) \bowtie S(B, C)$**:**

"$(a, b)$ from $R$ and $(b, c)$ from $S$ get $(a, b, c)$ in $R(A, B) \bowtie S(B, C)$"

- **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

# Relational Algebra Operations

**General Natural Join on more than 3 attributes**

Do like for relations with two attributes, by considering

- Type $A$ attributes: in $R$, but not in $S$
- Type $B$ attributes: both in $R, S$
- Type $C$ attributes: in $S$, but 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 $N$ reflect single bit, e.g. $M \leftrightarrow 0, N \leftrightarrow 1$

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

# 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 $m_{ij}$ and $n_{jl}$, generate all possible key-value pairs $((i,l),(M,j,m_{ij})$ and $((i,l),(N,j,n_{jl})$
- **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 $j$
  - Yields

$$(M,1,m_{i1}),(N,1,n_{1l}),(M,2,m_{i2}),(N,2,n_{2l}),...,(M,n,m_{in}),(N,n,n_{nl}) \tag{5}$$

  - After sorting, multiply each of two consecutive values $m_{ij}, n_{jl}$
  - Add up all the products ☞ yields $\sum_{j=1}^{n} m_{ij} n_{jl}$

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

*Workflow Systems*

# Workflow Systems: Introduction

- Workflow systems generalize MapReduce
- Just as much as MapReduce:
  - They're built on distributed file systems
  - They orchestrate large numbers of tasks with only small input provided by the user
  - They automatically handle failures
- In addition:
  - Single tasks can do other things than just Map or Reduce
  - Tasks interact in more complex ways

# Workflow Systems: Flow Graph

- A *function* represents arbitrary functionality within a workflow
  - and not just 'Map' or 'Reduce'
- Functions are represented as *nodes* of the *flow graph*
- Arcs $a \rightarrow b$ for two functions $a, b$ mean that the output of function $a$ is provided to function $b$ as input
- *Note:* The same function could be used by many tasks

# Workflow Systems

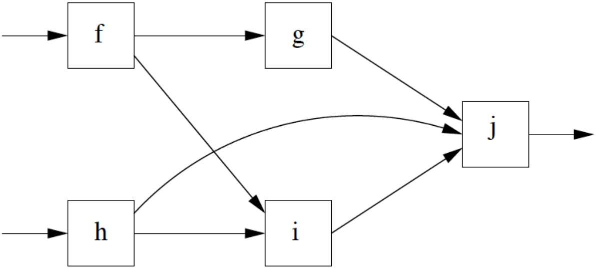



Figure: More complex workflow than MapReduce

Adopted from `mmds.org`

# Workflow Systems: Acyclic Flow Graph

- It is easier to deal with *acyclic flow graphs*
  - This means that one cannot return to functions
- *Blocking Property:* tasks only generate output upon completion
  - Blocking property easily applicable only in acyclic workflows
- *Simple Example of Workflow:* Cascades of Map-Reduce jobs
  - Output of Map jobs generated only after all Map tasks are completed
  - Reduce can work only on complete output anyway

# Popular Workflow Systems

- *Spark:* developed by UC Berkeley
- *TensorFlow:* Google's system, primarily developed for neural network computations
- *Pregel:* also by Google, for handling *recursive* (i.e. cyclic) workflows
- *Snakemake:* easy-to-use workflow system, inspired by MakeFile logic/functionality

# Spark

- State-of-the-art workflow system:
  - Very efficient with failures
  - Very efficient in grouping tasks among nodes
  - Very efficient in scheduling execution of functions
- Basic concept: *Resilient Distributed Dataset (RDD)*
  - Generalizes key-value pair type of data: RDD is a file of objects of one type
  - *Distributed:* broken into chunks held at different nodes
  - *Resilient:* recoverable from losses of (even all) chunks
- *Transformations* (steps of functions) turn RDD into others
- *Actions* turn other data (from surrounding file system) into RDD's and vice versa

# Spark: Transformations

**Remark:** For the following, consider equivalent methods in Python

- *Map* takes a function as parameter and applies it to every element of an RDD, generating a new one
  - Turns one object into exactly another object, but not several ones
  - Remember: Map from MapReduce generates several key-value pairs from one object
- *Flatmap* is like Map from MapReduce, and generalizes it from key-value pairs to general object types (not implemented in Python)
- *Filter* takes a predicate as input
  - Predicate is true or false for elements of RDD
  - So RDD is filtered for objects for which predicate applies
  - Yields a 'filtered RDD'

# Spark: Reduce and Relational Database Operations

- *Reduce* is an action, and takes as parameter a function that
  - applies to two elements of a particular type $T$
  - returns one element of type $T$
  - and is applied repeatedly until a single element remains
  - Works for associative and commutative operations
- Many *Relational Database Operations* are implemented in Spark:
  - Process RDD's reflecting tuples of relations
  - *Examples:* Join, GroupByKey

# Spark: Implementation Details

- Spark is similar like MapReduce in handling data (chunks are called *splits*)
- *Lazy evaluation* allows to apply several transformations consecutively to splits:
  - No intermediate formation of entire RDD's
  - Contradicts blocking property, because partial output is passed on to new functions
- *Resilience* (despite lazy evaluation) is maintained by *lineages of RDD's*
- Beneficial trade-off of more complex recovery of failures versus greater speed overall
  - Note that greater speed reduces probability of failures

# TensorFlow

- Open-source system developed (initially) by Google for machine-learning applications
- Programming interface for writing sequences of steps
- Data are *tensors*, which are multidimensional matrices
- Power comes from built-in operations applicable to tensors

# Recursive Workflows

*Examples:*

- Calculating fixed-points ($M\bar{v} = \bar{v}$ for a matrix $M$ and $v$) by iterative application of $M$ to $v$

$$v \to Mv \to M^2v \to \ldots \to M^tv \to M^{t+1}v \to \ldots \overset{t\to\infty}{\to} \bar{v} \qquad (6)$$

- Gradient descent, e.g. required in TensorFlow for determining optimal sets of parameters for machine learning models
- *Lack of blocking property*:
  - Flow graphs have cycles
  - Tasks may provide their output as input to other tasks whose output in turn results in more input to the first task
  - So generation of output only when task is done does not work
  - *Recovery from failures needs to be reorganized*

# Transitive Closure: Definition

Definition [Transitive Closure]:
Let $R(X, Y)$ be a relation.

- $R(X, Y)$ is *transitive* if $(x, z) \in R$ and $(z, y) \in R$ imply that $(x, y) \in R$ as well
- The *transitive closure* $\overline{R(X, Y)}$ of $R(X, Y)$ is the *smallest set of tuples to be added* to $R(X, Y)$ that renders the resulting set of tuples transitive

# Recursive Workflows: Example

- Directed graph stored as relation $E(X, Y)$, listing arcs from $X$ to $Y$
- Want to compute relation $P(X, Y)$, listing paths from $X$ to $Y$
- $P$ is transitive closure of $E$
- *Reminder:*
  - Natural Join $P(X, Z) \bowtie P(Z, Y)$, for given $x \in X, y \in Y$ generates $(x, z, y)$ for all applicable $z \in Z$, so possibly generates several $(x, z_1, y), (x, z_2, y), \ldots$
  - Project $\pi_{X,Y}$: all $(x, z_1, y), (x, z_2, y), \ldots$ become one $(x, y)$

# Recursive Workflows: Example

- *Reminder:*
  - Natural Join $P(X, Z) \bowtie P(Z, Y)$, for given $x \in X, y \in Y$ generates $(x, z, y)$ for all applicable $z \in Z$, so possibly generates several $(x, z_1, y), (x, z_2, y), ...$
  - Project $\pi_{X,Y}$: all $(x, z_1, y), (x, z_2, y), ...$ become one $(x, y)$
- *Algorithm:*
  - *Start:* $P(X, Y) = E(X, Y)$
  - *Iteration:* Add to $P$ tuples

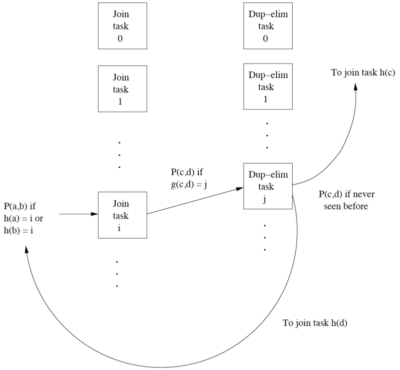
$$\pi_{X,Y}(P(X, Z) \bowtie P(Z, Y)) \tag{7}$$

as pairs of nodes $X$ and $Y$ s.t. for some node $Z$ there is path from $X$ to $Z$ and from $Z$ to $Y$

# Example: Transitive Closure

$P(a, b)$ corresponds to $(a, b)$

- $n$ Join tasks, corresponding to buckets of hash function $h$
- Tuple $P(a, b)$ is assigned to Join tasks $h(a)$ and $h(b)$
- $i$-th Join task receives $P(a, b)$
  - Store $P(a, b)$ locally
  - If $h(a) = i$ look for tuples $P(x, a)$ and produce $P(x, b)$
  - If $h(b) = i$ look for tuples $P(b, y)$ and produce $P(a, y)$




Transitive closure by recursive tasks

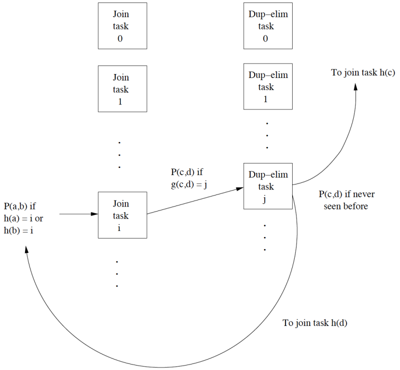
Adopted from `mmds.org`

UNIVERSITÄT BIELEFELD

# Example: Transitive Closure

$P(a, b)$ corresponds to $(a, b)$

- *i*-th Join task receives $P(a, b)$
  - Store $P(a, b)$ locally
  - If $h(a) = i$ look for tuples $P(x, a)$ and produce $P(x, b)$
  - If $h(b) = i$ look for tuples $P(b, y)$ and produce $P(a, y)$
- *Additional explanation:*
  - $h(a) = i$, so $(a, b)$ and $(x, a)$ get stored at Join task $i \Rightarrow$ Generate $(x, b)$
  - $h(b) = i$, so $(a, b)$ and $(b, y)$ get stored at Join task $i \Rightarrow$ Generate $(a, y)$

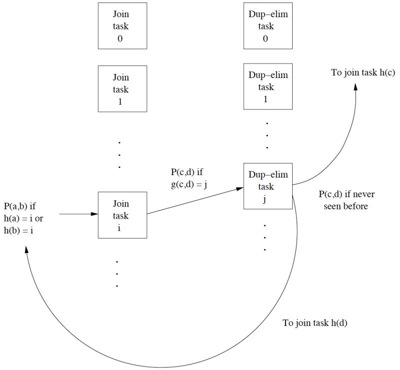



Transitive closure by recursive tasks

Adopted from `mmds.org`

# Recursive Workflows: Example

- $m$ Dup-elim tasks, corresponding to buckets of hash function $g$
- $P(c,d)$ (as output of Join task) is sent to Dup-elim task $j = g(c,d)$
- Dup-elim task $j$ checks whether $P(c,d)$ was received before
  - If *yes*, $P(c,d)$ is ignored (and not stored)
  - If *not*, $P(c,d)$ is stored locally,
  - *and* sent to Join tasks $h(c)$ and $h(d)$

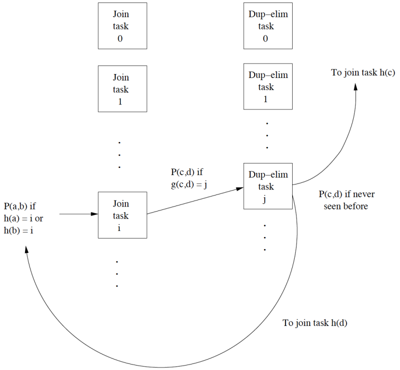



Transitive closure by recursive tasks

Adopted from `mmds.org`

# Recursive Workflows: Example

- Every Join task has $m$ output files
- Every Dup-elim task has $n$ output files
- Initially, tuples $E(a, b)$ are sent to Dup-elim tasks $g(a, b)$




Transitive closure by recursive tasks

Adopted from `mmds.org`

# Recursive Workflows: Failure Handling

- *Iterated MapReduce:* Application is repeated execution / sequence of MapReduce job(s) (“HaLoop”)
- *Spark Approach:* Lazy evaluation, lineage mechanisms, option to store intermediate results
- *Bulk Synchronous Systems:* Graph-based model using “periodic checkpointing”

# Bulk Synchronous Systems: Pregel

- System views data as *graph*:
  - *Nodes* (roughly) reflect tasks
  - *Arcs:* from nodes whose output (messages) are input to other nodes
- *Supersteps:*
  - All messages received by any of the nodes from the previous superstep are processed
  - All messages generated are sent to their destinations
- *Advantage:* Sending messages means communication costs, bundling them reduces costs
- *Failure Management:* Checkpointing entire computation by making copy after each superstep
- May be beneficial to checkpoint periodically after number of supersteps

# Snakemake

- Create *reproducible* and *scalable* data analyses
- Workflows described in human readable, Python based language
- Seamlessly scale to server, cluster, grid and cloud environments
- Integrating descriptions of required software, deployable to any execution environment

# Materials / Outlook

- See *Mining of Massive Datasets*, chapter 2.1–2.4
- As usual, see `http://www.mmds.org/` in general for further resources
- Next lecture: "Map Reduce / Workflow Systems II"
  - See *Mining of Massive Datasets* 2.5–2.6

# Example / Illustration