

Map Reduce / Workflow Systems II

Alexander Schönhuth



Bielefeld University
May 20, 2020

LEARNING GOALS TODAY

- ▶ Get to know idea of workflow systems and some examples
- ▶ Understand the definition of *communication cost*
- ▶ Understand the definition of *wall clock time*
- ▶ Get to know theory and intuition of *complexity theory* for MapReduce

LEARNING GOALS TODAY

- ▶ Get to know idea of workflow systems and some examples
- ▶ Understand the definition of *communication cost*
- ▶ Understand the definition of *wall clock time*
- ▶ Get to know theory and intuition of *complexity theory* for MapReduce

LEARNING GOALS TODAY

- ▶ Get to know idea of workflow systems and some examples
- ▶ Understand the definition of *communication cost*
- ▶ Understand the definition of *wall clock time*
- ▶ Get to know theory and intuition of *complexity theory* for MapReduce

LEARNING GOALS TODAY

- ▶ Get to know idea of workflow systems and some examples
- ▶ Understand the definition of *communication cost*
- ▶ Understand the definition of *wall clock time*
- ▶ Get to know theory and intuition of *complexity theory* for MapReduce

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: 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: 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: 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: 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: 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

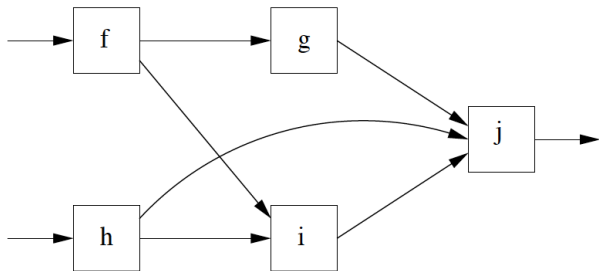


Figure: More complex workflow than MapReduce

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

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

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

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

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

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

- ▶ 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

- ▶ 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: 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: 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: 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: 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: 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: 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

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

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

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

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 ($Mv = v$ for a matrix M and v) by iterative application of M to v
- ▶ 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*

RECURSIVE WORKFLOWS

Examples:

- ▶ Calculating fixed-points ($Mv = v$ for a matrix M and v) by iterative application of M to v
- ▶ 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*

RECURSIVE WORKFLOWS

Examples:

- ▶ Calculating fixed-points ($Mv = v$ for a matrix M and v) by iterative application of M to v
- ▶ 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*

RECURSIVE WORKFLOWS

Examples:

- ▶ Calculating fixed-points ($Mv = v$ for a matrix M and v) by iterative application of M to v
- ▶ 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*

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 (see below)
- ▶ *Algorithm:*
 - ▶ *Start:* $P(X, Y) = E(X, Y)$
 - ▶ *Iteration:* Add to P tuples

$$\pi_{X,Y}(P(X, Z) \bowtie P(Z, Y)) \quad (1)$$

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

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 (see below)
- ▶ *Algorithm:*
 - ▶ *Start:* $P(X, Y) = E(X, Y)$
 - ▶ *Iteration:* Add to P tuples

$$\pi_{X,Y}(P(X, Z) \bowtie P(Z, Y)) \quad (1)$$

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

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 (see below)
- ▶ *Algorithm:*
 - ▶ *Start:* $P(X, Y) = E(X, Y)$
 - ▶ *Iteration:* Add to P tuples

$$\pi_{X,Y}(P(X, Z) \bowtie P(Z, Y)) \quad (1)$$

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

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 (see below)
- ▶ *Algorithm:*
 - ▶ *Start:* $P(X, Y) = E(X, Y)$
 - ▶ *Iteration:* Add to P tuples

$$\pi_{X,Y}(P(X, Z) \bowtie P(Z, Y)) \quad (1)$$

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

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 (see below)
- ▶ *Algorithm:*
 - ▶ *Start:* $P(X, Y) = E(X, Y)$
 - ▶ *Iteration:* Add to P tuples

$$\pi_{X,Y}(P(X, Z) \bowtie P(Z, Y)) \quad (1)$$

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

TRANSITIVE CLOSURE: DEFINITION

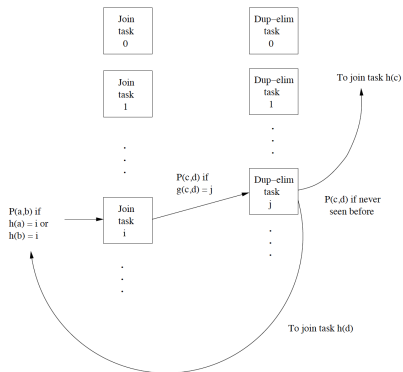
DEFINITION [TRANSITIVE CLOSURE]:

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*

EXAMPLE: TRANSITIVE CLOSURE

- ▶ 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 tasks 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 $P(a, y)$

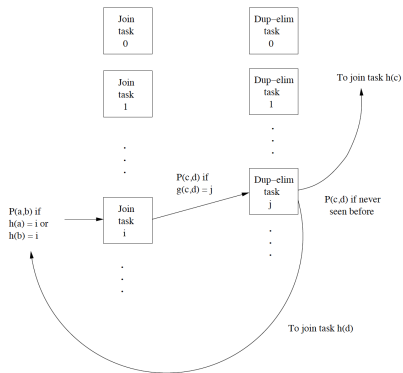


Transitive closure by recursive tasks

Adopted from mmds.org

EXAMPLE: TRANSITIVE CLOSURE

- ▶ 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 tasks 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 $P(a, y)$

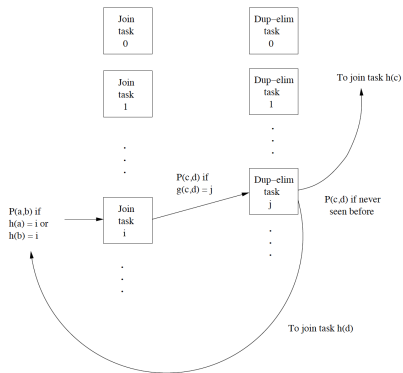


Transitive closure by recursive tasks

Adopted from mmds.org

EXAMPLE: TRANSITIVE CLOSURE

- ▶ 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 tasks 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 $P(a, y)$

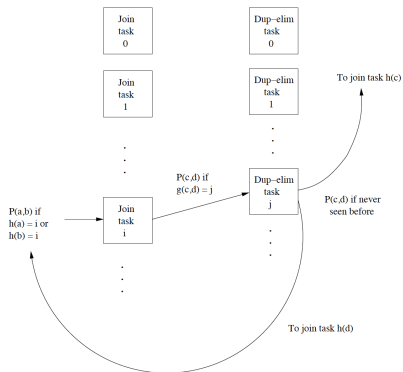


Transitive closure by recursive tasks

Adopted from mmds.org

EXAMPLE: TRANSITIVE CLOSURE

- ▶ 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 tasks 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 $P(a, y)$

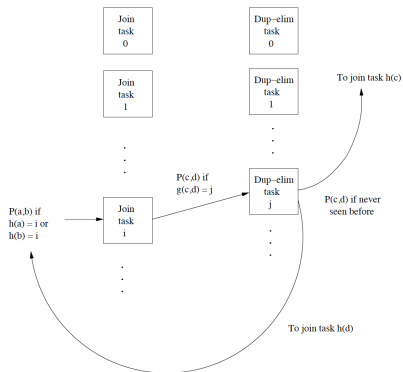


Transitive closure by recursive tasks

Adopted from mmds.org

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 $h(d)$

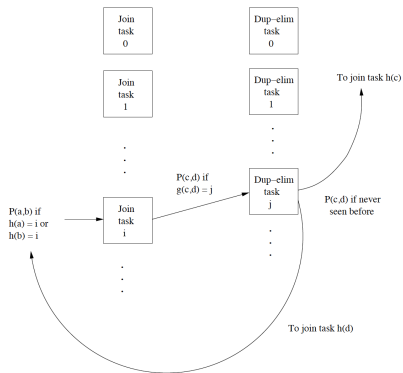


Transitive closure by recursive tasks

Adopted from mmds.org

RECURSIVE WORKFLOWS: EXAMPLE

- ▶ 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 $h(d)$

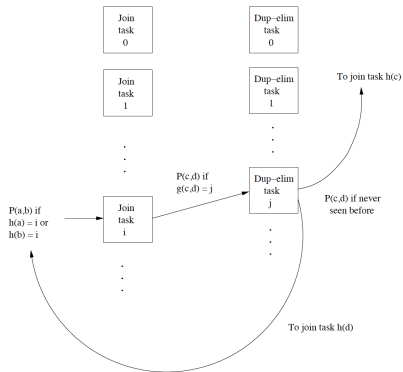


Transitive closure by recursive tasks

Adopted from mmds.org

RECURSIVE WORKFLOWS: EXAMPLE

- ▶ 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 $h(d)$

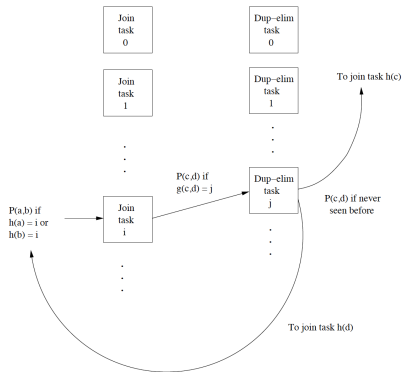


Transitive closure by recursive tasks

Adopted from mmds.org

RECURSIVE WORKFLOWS: EXAMPLE

- ▶ 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 $h(d)$

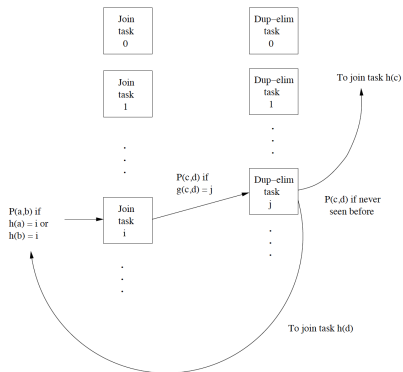


Transitive closure by recursive tasks

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 $g(a, b)$

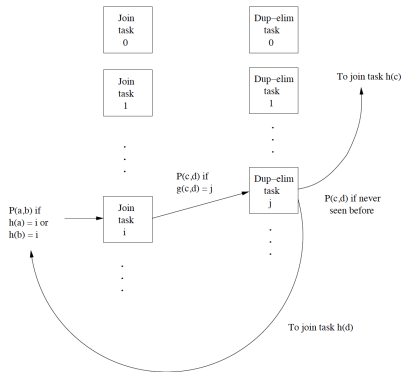


Transitive closure by recursive tasks

Adopted from mmds.org

RECURSIVE WORKFLOWS: 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 $g(a, b)$

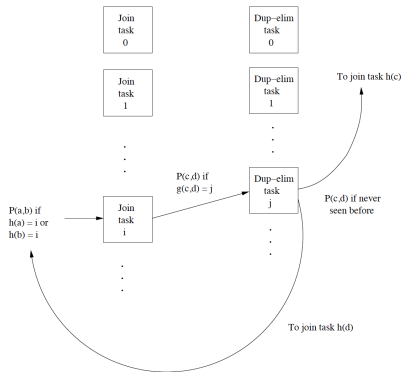


Transitive closure by recursive tasks

Adopted from mmds.org

RECURSIVE WORKFLOWS: 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

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

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

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

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

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

The Communication-Cost Model

COMMUNICATION COST

Situation

- ▶ Algorithm implemented by acyclic network of tasks:
 - ▶ Map tasks feeding Reduce tasks
 - ▶ Cascade of several MapReduce jobs
 - ▶ More general workflow structure (e.g. Fig. 1)

COMMUNICATION COST

Situation

- ▶ Algorithm implemented by acyclic network of tasks:
 - ▶ Map tasks feeding Reduce tasks
 - ▶ Cascade of several MapReduce jobs
 - ▶ More general workflow structure (e.g. Fig. 1)

DEFINITION [COMMUNICATION COST]:

- ▶ The *communication cost of a task* is the size of the input it receives
- ▶ The *communication cost of an algorithm* is the sum of the communication costs of its tasks

COMMUNICATION COST

Situation

- ▶ Algorithm implemented by acyclic network of tasks:
 - ▶ Map tasks feeding Reduce tasks
 - ▶ Cascade of several MapReduce jobs
 - ▶ More general workflow structure (e.g. Fig. 1)

DEFINITION [COMMUNICATION COST]:

- ▶ The *communication cost of a task* is the size of the input it receives
- ▶ The *communication cost of an algorithm* is the sum of the communication costs of its tasks

COMMUNICATION COST

Why Communication Cost?

- ▶ Computing communication cost is the way to measure the complexity of distributed algorithm
- ▶ Neglect time necessary for tasks to execute
- ▶ Importance of communication cost:
 - ▶ Tasks tend to be simple (often linear in size of input)
 - ▶ Interconnect speed of compute cluster (typically 1 Gbit/sec) slow compared with speed processors execute instructions
 - ▶ Often there is competition for the interconnect when several nodes are communicating
 - ▶ Moving data from disk to memory may exceed runtime

COMMUNICATION COST

Why Communication Cost?

- ▶ Computing communication cost is the way to measure the complexity of distributed algorithm
- ▶ Neglect time necessary for tasks to execute
- ▶ Importance of communication cost:
 - ▶ Tasks tend to be simple (often linear in size of input)
 - ▶ Interconnect speed of compute cluster (typically 1 Gbit/sec) slow compared with speed processors execute instructions
 - ▶ Often there is competition for the interconnect when several nodes are communicating
 - ▶ Moving data from disk to memory may exceed runtime

COMMUNICATION COST

Why Communication Cost?

- ▶ Computing communication cost is the way to measure the complexity of distributed algorithm
- ▶ Neglect time necessary for tasks to execute
- ▶ Importance of communication cost:
 - ▶ Tasks tend to be simple (often linear in size of input)
 - ▶ Interconnect speed of compute cluster (typically 1 Gbit/sec) slow compared with speed processors execute instructions
 - ▶ Often there is competition for the interconnect when several nodes are communicating
 - ▶ Moving data from disk to memory may exceed runtime

COMMUNICATION COST

Why Communication Cost?

- ▶ Computing communication cost is the way to measure the complexity of distributed algorithm
- ▶ Neglect time necessary for tasks to execute
- ▶ Importance of communication cost:
 - ▶ Tasks tend to be simple (often linear in size of input)
 - ▶ Interconnect speed of compute cluster (typically 1 Gbit/sec) slow compared with speed processors execute instructions
 - ▶ Often there is competition for the interconnect when several nodes are communicating
 - ▶ Moving data from disk to memory may exceed runtime

COMMUNICATION COST

Why Communication Cost?

- ▶ Computing communication cost is the way to measure the complexity of distributed algorithm
- ▶ Neglect time necessary for tasks to execute
- ▶ Importance of communication cost:
 - ▶ Tasks tend to be simple (often linear in size of input)
 - ▶ Interconnect speed of compute cluster (typically 1 Gbit/sec) slow compared with speed processors execute instructions
 - ▶ Often there is competition for the interconnect when several nodes are communicating
 - ▶ Moving data from disk to memory may exceed runtime

COMMUNICATION COST

Why Communication Cost?

- ▶ Computing communication cost is the way to measure the complexity of distributed algorithm
- ▶ Neglect time necessary for tasks to execute
- ▶ Importance of communication cost:
 - ▶ Tasks tend to be simple (often linear in size of input)
 - ▶ Interconnect speed of compute cluster (typically 1 Gbit/sec) slow compared with speed processors execute instructions
 - ▶ Often there is competition for the interconnect when several nodes are communicating
 - ▶ Moving data from disk to memory may exceed runtime

Why not Output Size?

- ▶ Output often is input to another task anyway
- ▶ Output rarely large in comparison with input or intermediate data

COMMUNICATION COST

Why Communication Cost?

- ▶ Computing communication cost is the way to measure the complexity of distributed algorithm
- ▶ Neglect time necessary for tasks to execute
- ▶ Importance of communication cost:
 - ▶ Tasks tend to be simple (often linear in size of input)
 - ▶ Interconnect speed of compute cluster (typically 1 Gbit/sec) slow compared with speed processors execute instructions
 - ▶ Often there is competition for the interconnect when several nodes are communicating
 - ▶ Moving data from disk to memory may exceed runtime

Why not Output Size?

- ▶ Output often is input to another task anyway
- ▶ Output rarely large in comparison with input or intermediate data

REMINDER: NATURAL JOIN

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

REMINDER: NATURAL JOIN

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

REMINDER: NATURAL JOIN

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

REMINDER: NATURAL JOIN

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

COMMUNICATION COST: NATURAL JOIN EXAMPLE

Suppose we are joining $R(A, B) \bowtie S(B, C)$ with R, S of sizes r and s .

- ▶ *Map*: Chunks of files R, S are input to Map tasks
☞ communication cost of Map is $r + s$ (in practice mostly disk to memory)
- ▶ *Reduce*: Input to Reduce tasks is all $(r + s)$ many key-value pairs generated by Map tasks
☞ communication cost for Reduce is $O(r + s)$
- ▶ *Output of Reduce* could be much larger than $O(r + s)$ (up to $O(rs)$), depending on how many tuples are to be generated for each key b

COMMUNICATION COST: NATURAL JOIN EXAMPLE

Suppose we are joining $R(A, B) \bowtie S(B, C)$ with R, S of sizes r and s .

- ▶ *Map*: Chunks of files R, S are input to Map tasks
↳ communication cost of Map is $r + s$ (in practice mostly disk to memory)
- ▶ *Reduce*: Input to Reduce tasks is all $(r + s)$ many key-value pairs generated by Map tasks
↳ communication cost for Reduce is $O(r + s)$
- ▶ *Output of Reduce* could be much larger than $O(r + s)$ (up to $O(rs)$), depending on how many tuples are to be generated for each key b

COMMUNICATION COST: NATURAL JOIN EXAMPLE

Suppose we are joining $R(A, B) \bowtie S(B, C)$ with R, S of sizes r and s .

- ▶ *Map*: Chunks of files R, S are input to Map tasks
 - ☞ communication cost of Map is $r + s$ (in practice mostly disk to memory)
- ▶ *Reduce*: Input to Reduce tasks is all $(r + s)$ many key-value pairs generated by Map tasks
 - ☞ communication cost for Reduce is $O(r + s)$
- ▶ *Output of Reduce* could be much larger than $O(r + s)$ (up to $O(rs)$), depending on how many tuples are to be generated for each key b

COMMUNICATION COST: NATURAL JOIN EXAMPLE

Suppose we are joining $R(A, B) \bowtie S(B, C)$ with R, S of sizes r and s .

- ▶ *Map*: Chunks of files R, S are input to Map tasks
 - ☞ communication cost of Map is $r + s$ (in practice mostly disk to memory)
- ▶ *Reduce*: Input to Reduce tasks is all $(r + s)$ many key-value pairs generated by Map tasks
 - ☞ communication cost for Reduce is $O(r + s)$
- ▶ *Output of Reduce* could be much larger than $O(r + s)$ (up to $O(rs)$), depending on how many tuples are to be generated for each key b

COMMUNICATION COST EXAMPLE: $R(A, B) \bowtie S(B, C)$

Let sizes of relations R and S be r and s .

Map

- ▶ Each chunk of the files holding R and S is fed to one task
 - ☞ Communication cost is $r + s$
- ▶ Nodes hold chunks already from file distribution step: no internode communication, only disk-to-memory costs
- ▶ All Map tasks perform a simple transformation, so only negligible computation cost
- ▶ Output about as large as input

COMMUNICATION COST EXAMPLE: $R(A, B) \bowtie S(B, C)$

Let sizes of relations R and S be r and s .

Map

- ▶ Each chunk of the files holding R and S is fed to one task
 - ↳ Communication cost is $r + s$
- ▶ Nodes hold chunks already from file distribution step: no internode communication, only disk-to-memory costs
- ▶ All Map tasks perform a simple transformation, so only negligible computation cost
- ▶ Output about as large as input

COMMUNICATION COST EXAMPLE: $R(A, B) \bowtie S(B, C)$

Let sizes of relations R and S be r and s .

Reduce

- ▶ Receives and divides input into tuples from R and S
- ▶ For each key, pairs each tuple from R with the ones from S
- ▶ Output size can vary: can be larger or smaller than $O(r + s)$
 - ▶ Many different B -values: output is small
 - ▶ Few B -values: output much larger
- ▶ Output large: computation cost could be much larger than $O(r + s)$
- ▶ Often output is further subsequently aggregated at further nodes
 - ☞ Communication cost greater than computation cost

COMMUNICATION COST EXAMPLE: $R(A, B) \bowtie S(B, C)$

Let sizes of relations R and S be r and s .

Reduce

- ▶ Receives and divides input into tuples from R and S
- ▶ For each key, pairs each tuple from R with the ones from S
- ▶ Output size can vary: can be larger or smaller than $O(r + s)$
 - ▶ Many different B-values: output is small
 - ▶ Few B-values: output much larger
- ▶ Output large: computation cost could be much larger than $O(r + s)$
- ▶ Often output is further subsequently aggregated at further nodes
- ▶ Communication cost greater than computation cost

COMMUNICATION COST EXAMPLE: $R(A, B) \bowtie S(B, C)$

Let sizes of relations R and S be r and s .

Reduce

- ▶ Receives and divides input into tuples from R and S
- ▶ For each key, pairs each tuple from R with the ones from S
- ▶ Output size can vary: can be larger or smaller than $O(r + s)$
 - ▶ Many different B-values: output is small
 - ▶ Few B-values: output much larger
- ▶ Output large: computation cost could be much larger than $O(r + s)$
- ▶ Often output is further subsequently aggregated at further nodes
 - ☞ Communication cost greater than computation cost

WALL-CLOCK TIME

DEFINITION [WALL-CLOCK TIME]:

The *wall-clock time* is defined to be the time for the entire parallel algorithm to finish.

WALL-CLOCK TIME

DEFINITION [WALL-CLOCK TIME]:

The *wall-clock time* is defined to be the time for the entire parallel algorithm to finish.

Example: Careless reasoning could make one assign all tasks to one node, which minimizes communication cost. But the wall-clock time is (likely to be) at its maximum.

EXAMPLE: MULTIWAY JOIN

Consider computing $R(A, B) \bowtie S(B, C) \bowtie T(C, D)$. For simplicity, let us assume that

- ▶ the probability that an R - and an S -tuple agree on B
- ▶ the probability that an S - and a T -tuple agree on C

are equal. Let p be that probability.

EXAMPLE: MULTIWAY JOIN

Consider computing $R(A, B) \bowtie S(B, C) \bowtie T(C, D)$. For simplicity, let us assume that

- ▶ the probability that an R - and an S -tuple agree on B
- ▶ the probability that an S - and a T -tuple agree on C

are equal. Let p be that probability.

Joining R and S first:

- ▶ Communication cost is $O(r + s)$ (see before)
- ▶ Size of output is prs
- ▶ Hence joining $R \bowtie S$ with T is $O((r + s) + (t + prs))$

EXAMPLE: MULTIWAY JOIN

Consider computing $R(A, B) \bowtie S(B, C) \bowtie T(C, D)$. For simplicity, let us assume that

- ▶ the probability that an R - and an S -tuple agree on B
- ▶ the probability that an S - and a T -tuple agree on C

are equal. Let p be that probability.

Joining R and S first:

- ▶ Communication cost is $O(r + s)$ (see before)
- ▶ Size of output is prs
- ▶ Hence joining $R \bowtie S$ with T is $O((r + s) + (t + prs))$

EXAMPLE: MULTIWAY JOIN

Consider computing $R(A, B) \bowtie S(B, C) \bowtie T(C, D)$. For simplicity, let us assume that

- ▶ the probability that an R - and an S -tuple agree on B
- ▶ the probability that an S - and a T -tuple agree on C

are equal. Let p be that probability.

Joining R and S first:

- ▶ Communication cost is $O(r + s)$ (see before)
- ▶ Size of output is prs
- ▶ Hence joining $R \bowtie S$ with T is $O((r + s) + (t + prs))$

EXAMPLE: MULTIWAY JOIN

Consider computing $R(A, B) \bowtie S(B, C) \bowtie T(C, D)$. For simplicity, let us assume that

- ▶ the probability that an R - and an S -tuple agree on B
- ▶ the probability that an S - and a T -tuple agree on C

are equal. Let p be that probability.

Joining R and S first:

- ▶ Communication cost is $O(r + s)$ (see before)
- ▶ Size of output is prs
- ▶ Hence joining $R \bowtie S$ with T is $O((r + s) + (t + prs))$

EXAMPLE: MULTIWAY JOIN

Consider computing $R(A, B) \bowtie S(B, C) \bowtie T(C, D)$. For simplicity, let us assume that

- ▶ the probability that an R - and an S -tuple agree on B
- ▶ the probability that an S - and a T -tuple agree on C

are equal. Let p be that probability.

Joining R and S first:

- ▶ Communication cost is $O(r + s)$ (see before)
- ▶ Size of output is prs
- ▶ Hence joining $R \bowtie S$ with T is $O((r + s) + (t + prs))$

Joining S and T first:

- ▶ yields $O((s + t) + (r + pst))$ by analogous considerations

EXAMPLE: MULTIWAY JOIN

Consider computing $R(A, B) \bowtie S(B, C) \bowtie T(C, D)$. For simplicity, let us assume that

- ▶ the probability that an R - and an S -tuple agree on B
- ▶ the probability that an S - and a T -tuple agree on C

are equal. Let p be that probability.

Joining R and S first:

- ▶ Communication cost is $O(r + s)$ (see before)
- ▶ Size of output is prs
- ▶ Hence joining $R \bowtie S$ with T is $O((r + s) + (t + prs))$

Joining S and T first:

- ▶ yields $O((s + t) + (r + pst))$ by analogous considerations

$R(A, B) \bowtie S(B, C) \bowtie T(C, D)$ IN ONE MAPREDUCE

Let p be the probability that an R - and an S -tuple agree on B , matching the probability for an S - and a T -tuple to agree on C .

- ▶ Hash B - and C -values, using functions h and g
- ▶ Let b and c be the number of buckets for h and g
- ▶ Let k be the number of Reducers; require that $bc = k$
 - ▶ Each reducer corresponds to a pair of buckets
 - ▶ Reducer corresponding to bucket pair (i, j) joins tuples $R(u, v), S(v, w), T(w, x)$ whenever $h(v) = i, g(w) = j$
- ▶ Hence Map tasks send R - and T -tuples to more than one reducer
 - ▶ R -tuples $R(u, v)$ go to all reducers $(h(v), y)$
 - ↔ goes to c reducers
 - ▶ T -tuples $T(w, x)$ go to all reducers $(z, g(w))$
 - ↔ goes to b reducers

$R(A, B) \bowtie S(B, C) \bowtie T(C, D)$ IN ONE MAPREDUCE

Let p be the probability that an R - and an S -tuple agree on B , matching the probability for an S - and a T -tuple to agree on C .

- ▶ Hash B - and C -values, using functions h and g
- ▶ Let b and c be the number of buckets for h and g
- ▶ Let k be the number of Reducers; require that $bc = k$
 - ▶ Each reducer corresponds to a pair of buckets
 - ▶ Reducer corresponding to bucket pair (i, j) joins tuples $R(u, v), S(v, w), T(w, x)$ whenever $h(v) = i, g(w) = j$
- ▶ Hence Map tasks send R - and T -tuples to more than one reducer
 - ▶ R -tuples $R(u, v)$ go to all reducers $(h(v), y)$
↔ goes to c reducers
 - ▶ T -tuples $T(w, x)$ go to all reducers $(z, g(w))$
↔ goes to b reducers

$R(A, B) \bowtie S(B, C) \bowtie T(C, D)$ IN ONE MAPREDUCE

Let p be the probability that an R - and an S -tuple agree on B , matching the probability for an S - and a T -tuple to agree on C .

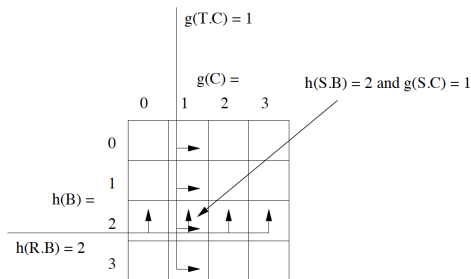
- ▶ Hash B - and C -values, using functions h and g
- ▶ Let b and c be the number of buckets for h and g
- ▶ Let k be the number of Reducers; require that $bc = k$
 - ▶ Each reducer corresponds to a pair of buckets
 - ▶ Reducer corresponding to bucket pair (i, j) joins tuples $R(u, v), S(v, w), T(w, x)$ whenever $h(v) = i, g(w) = j$
- ▶ Hence Map tasks send R - and T -tuples to more than one reducer
 - ▶ R -tuples $R(u, v)$ go to all reducers $(h(v), y)$
↔ goes to c reducers
 - ▶ T -tuples $T(w, x)$ go to all reducers $(z, g(w))$
↔ goes to b reducers

$R(A, B) \bowtie S(B, C) \bowtie T(C, D)$ IN ONE MAPREDUCE

Let p be the probability that an R - and an S -tuple agree on B , matching the probability for an S - and a T -tuple to agree on C .

- ▶ Hash B - and C -values, using functions h and g
- ▶ Let b and c be the number of buckets for h and g
- ▶ Let k be the number of Reducers; require that $bc = k$
 - ▶ Each reducer corresponds to a pair of buckets
 - ▶ Reducer corresponding to bucket pair (i, j) joins tuples $R(u, v), S(v, w), T(w, x)$ whenever $h(v) = i, g(w) = j$
- ▶ Hence Map tasks send R - and T -tuples to more than one reducer
 - ▶ R -tuples $R(u, v)$ go to all reducers $(h(v), y)$
 - ↳ goes to c reducers
 - ▶ T -tuples $T(w, x)$ go to all reducers $(z, g(w))$
 - ↳ goes to b reducers

MULTIWAY JOIN: ONE MAPREDUCE II

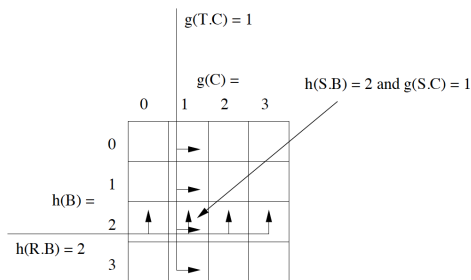


Sixteen reducers for a 3-way join

Adopted from mmds.org

- ▶ $h(v) = 2, g(w) = 1$
- ▶ S -tuple $S(v, w)$ goes to reducer for key (2, 1)
- ▶ R -tuple $R(u, v)$ goes to reducers for keys (2, 0), ..., (2, 3)
- ▶ T -tuple $T(w, x)$ goes to reducers for keys (0, 1), ..., (3, 1)

MULTIWAY JOIN: ONE MAPREDUCE II

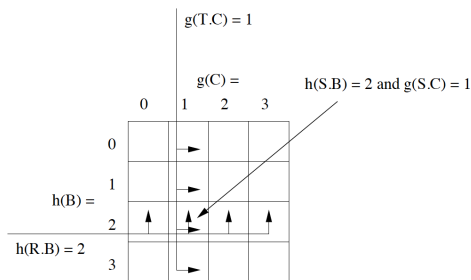


Sixteen reducers for a 3-way join

Adopted from mmds.org

- ▶ $h(v) = 2, g(w) = 1$
- ▶ S -tuple $S(v, w)$ goes to reducer for key $(2, 1)$
- ▶ R -tuple $R(u, v)$ goes to reducers for keys $(2, 0), \dots, (2, 3)$
- ▶ T -tuple $T(w, x)$ goes to reducers for keys $(0, 1), \dots, (3, 1)$

MULTIWAY JOIN: ONE MAPREDUCE II

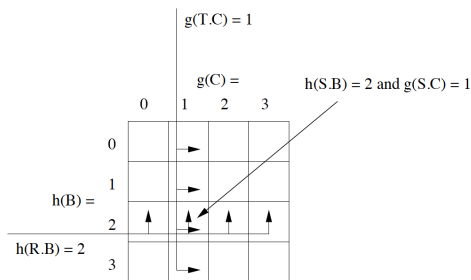


Sixteen reducers for a 3-way join

Adopted from mmds.org

- ▶ $h(v) = 2, g(w) = 1$
- ▶ S -tuple $S(v, w)$ goes to reducer for key $(2, 1)$
- ▶ R -tuple $R(u, v)$ goes to reducers for keys $(2, 0), \dots, (2, 3)$
- ▶ T -tuple $T(w, x)$ goes to reducers for keys $(0, 1), \dots, (3, 1)$

MULTIWAY JOIN: ONE MAPREDUCE II



Sixteen reducers for a 3-way join

Adopted from mmds.org

- ▶ $h(v) = 2, g(w) = 1$
- ▶ S -tuple $S(v, w)$ goes to reducer for key $(2, 1)$
- ▶ R -tuple $R(u, v)$ goes to reducers for keys $(2, 0), \dots, (2, 3)$
- ▶ T -tuple $T(w, x)$ goes to reducers for keys $(0, 1), \dots, (3, 1)$

MULTIWAY JOIN: ONE MAPREDUCE III

Communication cost:

- ▶ Moving tuples to proper reducers is sum of
 - ▶ s to send tuples $S(v, w)$ to $(h(v), g(w))$
 - ▶ rc to send tuples $R(u, v)$ to $(h(v), y)$ for each of the c possible $g(w) = y$
 - ▶ bt to send tuples $T(w, x)$ to $(z, g(w))$ for each of the b possible $h(b) = z$
- ▶ Additional (constant) cost $r + s + t$ to make each tuple input to one of the Map tasks (constant)

MULTIWAY JOIN: ONE MAPREDUCE III

Communication cost:

- ▶ Moving tuples to proper reducers is sum of
 - ▶ s to send tuples $S(v, w)$ to $(h(v), g(w))$
 - ▶ rc to send tuples $R(u, v)$ to $(h(v), y)$ for each of the c possible $g(w) = y$
 - ▶ bt to send tuples $T(w, x)$ to $(z, g(w))$ for each of the b possible $h(b) = z$
- ▶ Additional (constant) cost $r + s + t$ to make each tuple input to one of the Map tasks (constant)

MULTIWAY JOIN: ONE MAPREDUCE III

Communication cost:

- ▶ Moving tuples to proper reducers is sum of
 - ▶ s to send tuples $S(v, w)$ to $(h(v), g(w))$
 - ▶ rc to send tuples $R(u, v)$ to $(h(v), y)$ for each of the c possible $g(w) = y$
 - ▶ bt to send tuples $T(w, x)$ to $(z, g(w))$ for each of the b possible $h(b) = z$
- ▶ Additional (constant) cost $r + s + t$ to make each tuple input to one of the Map tasks (constant)

MULTIWAY JOIN: ONE MAPREDUCE III

Communication cost:

- ▶ Moving tuples to proper reducers is sum of
 - ▶ s to send tuples $S(v, w)$ to $(h(v), g(w))$
 - ▶ rc to send tuples $R(u, v)$ to $(h(v), y)$ for each of the c possible $g(w) = y$
 - ▶ bt to send tuples $T(w, x)$ to $(z, g(w))$ for each of the b possible $h(b) = z$
- ▶ Additional (constant) cost $r + s + t$ to make each tuple input to one of the Map tasks (constant)

MULTIWAY JOIN: ONE MAPREDUCE III

Communication cost:

- ▶ *Goal:* Select b and c , subject to $bc = k$, to minimize $s + cr + bt$
- ▶ Using Lagrangian multiplier λ yields to solve for
 - ▶ $r - \lambda b = 0$
 - ▶ $t - \lambda c = 0$
- ▶ It follows that $rt = \lambda^2 bc$, that is $rt = \lambda^2 k$, yielding further $\lambda = \sqrt{\frac{rt}{k}}$
- ▶ So, minimum communication cost at $c = \sqrt{\frac{kt}{r}}$ and $b = \sqrt{\frac{kr}{t}}$
- ▶ Substituting into $s + cr + bt$ yields $s + 2\sqrt{krt}$
- ▶ Adding $r + s + t$ yields $r + 2s + t + 2\sqrt{krt}$, which is usually dominated by $2\sqrt{krt}$

MULTIWAY JOIN: ONE MAPREDUCE III

Communication cost:

- ▶ *Goal:* Select b and c , subject to $bc = k$, to minimize $s + cr + bt$
- ▶ Using Lagrangian multiplier λ yields to solve for
 - ▶ $r - \lambda b = 0$
 - ▶ $t - \lambda c = 0$
- ▶ It follows that $rt = \lambda^2 bc$, that is $rt = \lambda^2 k$, yielding further $\lambda = \sqrt{\frac{rt}{k}}$
- ▶ So, minimum communication cost at $c = \sqrt{\frac{kt}{r}}$ and $b = \sqrt{\frac{kr}{t}}$
- ▶ Substituting into $s + cr + bt$ yields $s + 2\sqrt{krt}$
- ▶ Adding $r + s + t$ yields $r + 2s + t + 2\sqrt{krt}$, which is usually dominated by $2\sqrt{krt}$

MULTIWAY JOIN: ONE MAPREDUCE III

Communication cost:

- ▶ *Goal:* Select b and c , subject to $bc = k$, to minimize $s + cr + bt$
- ▶ Using Lagrangian multiplier λ yields to solve for
 - ▶ $r - \lambda b = 0$
 - ▶ $t - \lambda c = 0$
- ▶ It follows that $rt = \lambda^2 bc$, that is $rt = \lambda^2 k$, yielding further $\lambda = \sqrt{\frac{rt}{k}}$
- ▶ So, minimum communication cost at $c = \sqrt{\frac{kt}{r}}$ and $b = \sqrt{\frac{kr}{t}}$
- ▶ Substituting into $s + cr + bt$ yields $s + 2\sqrt{krt}$
- ▶ Adding $r + s + t$ yields $r + 2s + t + 2\sqrt{krt}$, which is usually dominated by $2\sqrt{krt}$

MULTIWAY JOIN: ONE MAPREDUCE III

Communication cost:

- ▶ *Goal:* Select b and c , subject to $bc = k$, to minimize $s + cr + bt$
- ▶ Using Lagrangian multiplier λ yields to solve for
 - ▶ $r - \lambda b = 0$
 - ▶ $t - \lambda c = 0$
- ▶ It follows that $rt = \lambda^2 bc$, that is $rt = \lambda^2 k$, yielding further $\lambda = \sqrt{\frac{rt}{k}}$
- ▶ So, minimum communication cost at $c = \sqrt{\frac{kt}{r}}$ and $b = \sqrt{\frac{kr}{t}}$
- ▶ Substituting into $s + cr + bt$ yields $s + 2\sqrt{krt}$
- ▶ Adding $r + s + t$ yields $r + 2s + t + 2\sqrt{krt}$, which is usually dominated by $2\sqrt{krt}$

MULTIWAY JOIN: ONE MAPREDUCE III

Communication cost:

- ▶ *Goal:* Select b and c , subject to $bc = k$, to minimize $s + cr + bt$
- ▶ Using Lagrangian multiplier λ yields to solve for
 - ▶ $r - \lambda b = 0$
 - ▶ $t - \lambda c = 0$
- ▶ It follows that $rt = \lambda^2 bc$, that is $rt = \lambda^2 k$, yielding further $\lambda = \sqrt{\frac{rt}{k}}$
- ▶ So, minimum communication cost at $c = \sqrt{\frac{kt}{r}}$ and $b = \sqrt{\frac{kr}{t}}$
- ▶ Substituting into $s + cr + bt$ yields $s + 2\sqrt{krt}$
- ▶ Adding $r + s + t$ yields $r + 2s + t + 2\sqrt{krt}$, which is usually dominated by $2\sqrt{krt}$

Complexity Theory for MapReduce

MAPREDUCE: COMPLEXITY THEORY

Idea

- ▶ *Reminder:* A “reducer” is the execution of a Reduce task on a single key and the associated value list
- ▶ *Important considerations:*
 - ▶ Keep communication cost low
 - ▶ Keep wall-clock time low
 - ▶ Execute each reducer in main memory
- ▶ *Intuition:*
 - ▶ The less communication, the less parallelism, so
 - ▶ the more wall-clock time
 - ▶ the more main memory needed
- ▶ *Goal:* Develop encompassing complexity theory

MAPREDUCE: COMPLEXITY THEORY

Idea

- ▶ *Reminder:* A “reducer” is the execution of a Reduce task on a single key and the associated value list
- ▶ *Important considerations:*
 - ▶ Keep communication cost low
 - ▶ Keep wall-clock time low
 - ▶ Execute each reducer in main memory
- ▶ *Intuition:*
 - ▶ The less communication, the less parallelism, so
 - ▶ the more wall-clock time
 - ▶ the more main memory needed
- ▶ *Goal:* Develop encompassing complexity theory

MAPREDUCE: COMPLEXITY THEORY

Idea

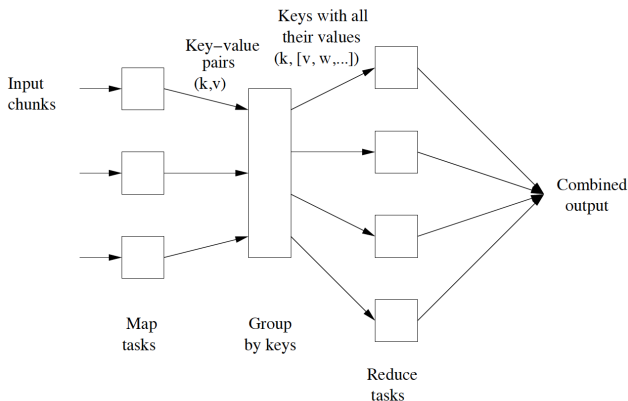
- ▶ *Reminder:* A “reducer” is the execution of a Reduce task on a single key and the associated value list
- ▶ *Important considerations:*
 - ▶ Keep communication cost low
 - ▶ Keep wall-clock time low
 - ▶ Execute each reducer in main memory
- ▶ *Intuition:*
 - ▶ The less communication, the less parallelism, so
 - ▶ the more wall-clock time
 - ▶ the more main memory needed
- ▶ *Goal:* Develop encompassing complexity theory

MAPREDUCE: COMPLEXITY THEORY

Idea

- ▶ *Reminder:* A “reducer” is the execution of a Reduce task on a single key and the associated value list
- ▶ *Important considerations:*
 - ▶ Keep communication cost low
 - ▶ Keep wall-clock time low
 - ▶ Execute each reducer in main memory
- ▶ *Intuition:*
 - ▶ The less communication, the less parallelism, so
 - ▶ the more wall-clock time
 - ▶ the more main memory needed
- ▶ *Goal:* Develop encompassing complexity theory

REDUCER SIZE: INFORMAL EXPLANATION



Reducer size: maximum length of list $[v,w,\dots]$ after grouping keys

Adopted from mmds.org

REDUCER SIZE

DEFINITION [REDUCER SIZE]:

The *reducer size* q is the upper bound on the number of values to appear in the list of a single key.

REDUCER SIZE

DEFINITION [REDUCER SIZE]:

The *reducer size* q is the upper bound on the number of values to appear in the list of a single key.

Motivation

- ▶ Small reducer size forces to have many reducers
- ▶ Further creating many Reduce tasks implies high parallelism, hence small wall-clock time
- ▶ Sufficiently small reducer size allows to have all data in main memory

REDUCER SIZE

DEFINITION [REDUCER SIZE]:

The *reducer size* q is the upper bound on the number of values to appear in the list of a single key.

Motivation

- ▶ Small reducer size forces to have many reducers
- ▶ Further creating many Reduce tasks implies high parallelism, hence small wall-clock time
- ▶ Sufficiently small reducer size allows to have all data in main memory

REDUCER SIZE

DEFINITION [REDUCER SIZE]:

The *reducer size* q is the upper bound on the number of values to appear in the list of a single key.

Motivation

- ▶ Small reducer size forces to have many reducers
- ▶ Further creating many Reduce tasks implies high parallelism, hence small wall-clock time
- ▶ Sufficiently small reducer size allows to have all data in main memory

REPLICATION RATE

DEFINITION [REPLICATION RATE]:

The *replication rate* r is the number of all key-value pairs generated by Map tasks, divided by the number of inputs.

REPLICATION RATE

DEFINITION [REPLICATION RATE]:

The *replication rate* r is the number of all key-value pairs generated by Map tasks, divided by the number of inputs.

Motivating Example

- ▶ One-pass matrix multiplication algorithm:
 - ▶ Matrices involved are $n \times n$
 - ▶ *Reminder*: Key-value pairs for MN are $((i, k), (M, j, m_{ij})), j = 1, \dots, n$ and $((i, k), (N, j, n_{jk})), j = 1, \dots, n$
- ▶ Replication rate r is equal to n :
 - ▶ Inputs are all m_{ij} and n_{jk}
 - ▶ For each m_{ij} , one generates key-value pairs for $(i, k), k = 1, \dots, n$
 - ▶ For each n_{jk} , one generates key-value pairs for $(i, k), i = 1, \dots, n$
- ▶ Reducer size is $2n$: for each key (i, k) there are n values from each m_{ij} and n values from each n_{jk}

REPLICATION RATE

DEFINITION [REPLICATION RATE]:

The *replication rate* r is the number of all key-value pairs generated by Map tasks, divided by the number of inputs.

Motivating Example

- ▶ One-pass matrix multiplication algorithm:
 - ▶ Matrices involved are $n \times n$
 - ▶ *Reminder*: Key-value pairs for MN are $((i, k), (M, j, m_{ij}))$, $j = 1, \dots, n$ and $((i, k), (N, j, n_{jk}))$, $j = 1, \dots, n$
- ▶ Replication rate r is equal to n :
 - ▶ Inputs are all m_{ij} and n_{jk}
 - ▶ For each m_{ij} , one generates key-value pairs for (i, k) , $k = 1, \dots, n$
 - ▶ For each n_{jk} , one generates key-value pairs for (i, k) , $i = 1, \dots, n$
- ▶ Reducer size is $2n$: for each key (i, k) there are n values from each m_{ij} and n values from each n_{jk}

REPLICATION RATE

DEFINITION [REPLICATION RATE]:

The *replication rate* r is the number of all key-value pairs generated by Map tasks, divided by the number of inputs.

Motivating Example

- ▶ One-pass matrix multiplication algorithm:
 - ▶ Matrices involved are $n \times n$
 - ▶ *Reminder*: Key-value pairs for MN are $((i, k), (M, j, m_{ij}))$, $j = 1, \dots, n$ and $((i, k), (N, j, n_{jk}))$, $j = 1, \dots, n$
- ▶ Replication rate r is equal to n :
 - ▶ Inputs are all m_{ij} and n_{jk}
 - ▶ For each m_{ij} , one generates key-value pairs for (i, k) , $k = 1, \dots, n$
 - ▶ For each n_{jk} , one generates key-value pairs for (i, k) , $i = 1, \dots, n$
- ▶ Reducer size is $2n$: for each key (i, k) there are n values from each m_{ij} and n values from each n_{jk}

REPLICATION RATE

DEFINITION [REPLICATION RATE]:

The *replication rate* r is the number of all key-value pairs generated by Map tasks, divided by the number of inputs.

Motivating Example

- ▶ One-pass matrix multiplication algorithm:
 - ▶ Matrices involved are $n \times n$
 - ▶ *Reminder*: Key-value pairs for MN are $((i, k), (M, j, m_{ij}))$, $j = 1, \dots, n$ and $((i, k), (N, j, n_{jk}))$, $j = 1, \dots, n$
- ▶ Replication rate r is equal to n :
 - ▶ Inputs are all m_{ij} and n_{jk}
 - ▶ For each m_{ij} , one generates key-value pairs for (i, k) , $k = 1, \dots, n$
 - ▶ For each n_{jk} , one generates key-value pairs for (i, k) , $i = 1, \dots, n$
- ▶ Reducer size is $2n$: for each key (i, k) there are n values from each m_{ij} and n values from each n_{jk}

REPLICATION RATE

DEFINITION [REPLICATION RATE]:

The *replication rate* r is the number of all key-value pairs generated by Map tasks, divided by the number of inputs.

Motivating Example

- ▶ One-pass matrix multiplication algorithm:
 - ▶ Matrices involved are $n \times n$
 - ▶ *Reminder*: Key-value pairs for MN are $((i, k), (M, j, m_{ij}))$, $j = 1, \dots, n$ and $((i, k), (N, j, n_{jk}))$, $j = 1, \dots, n$
- ▶ Replication rate r is equal to n :
 - ▶ Inputs are all m_{ij} and n_{jk}
 - ▶ For each m_{ij} , one generates key-value pairs for (i, k) , $k = 1, \dots, n$
 - ▶ For each n_{jk} , one generates key-value pairs for (i, k) , $i = 1, \dots, n$
- ▶ Reducer size is $2n$: for each key (i, k) there are n values from each m_{ij} and n values from each n_{jk}

REPLICATION RATE

DEFINITION [REPLICATION RATE]:

The *replication rate* r is the number of all key-value pairs generated by Map tasks, divided by the number of inputs.

Motivating Example

- ▶ One-pass matrix multiplication algorithm:
 - ▶ Matrices involved are $n \times n$
 - ▶ *Reminder*: Key-value pairs for MN are $((i, k), (M, j, m_{ij}))$, $j = 1, \dots, n$ and $((i, k), (N, j, n_{jk}))$, $j = 1, \dots, n$
- ▶ Replication rate r is equal to n :
 - ▶ Inputs are all m_{ij} and n_{jk}
 - ▶ For each m_{ij} , one generates key-value pairs for (i, k) , $k = 1, \dots, n$
 - ▶ For each n_{jk} , one generates key-value pairs for (i, k) , $i = 1, \dots, n$
- ▶ Reducer size is $2n$: for each key (i, k) there are n values from each m_{ij} and n values from each n_{jk}

EXAMPLE: SIMILARITY JOIN

Situation

- ▶ Given large set X of elements
- ▶ Given similarity measure $s(x, y)$ for measuring similarity between $x, y \in X$
- ▶ Measure is symmetric: $s(x, y) = s(y, x)$
- ▶ *Output* of the algorithm: all pairs x, y where $s(x, y) \geq t$ for threshold t
- ▶ *Exemplary input*: 1 million images (i, P_i) where
 - ▶ i is ID of image
 - ▶ P_i is picture itself
 - ▶ Each picture is 1MB

EXAMPLE: SIMILARITY JOIN

Situation

- ▶ Given large set X of elements
- ▶ Given similarity measure $s(x, y)$ for measuring similarity between $x, y \in X$
- ▶ Measure is symmetric: $s(x, y) = s(y, x)$
- ▶ *Output* of the algorithm: all pairs x, y where $s(x, y) \geq t$ for threshold t
- ▶ *Exemplary input*: 1 million images (i, P_i) where
 - ▶ i is ID of image
 - ▶ P_i is picture itself
 - ▶ Each picture is 1MB

EXAMPLE: SIMILARITY JOIN

MapReduce: Bad Idea

- ▶ Use keys (i, j) for pair of pictures $(i, P_i), (j, P_j)$
- ▶ *Map*: generates $((i, j), [P_i, P_j])$ as input for
- ▶ *Reduce*: computes $s(P_i, P_j)$ and decides whether $s(P_i, P_j) \geq t$
- ▶ Reducer size q is small: 2 MB; expected to fit in main memory
- ▶ *However*, each picture makes part of 999 999 key-value pairs, so

$$r = 999\,999$$

- ▶ Hence, number of bytes communicated from Map to Reduce is

$$10^6 \times 999\,999 \times 10^6 = 10^{18}$$

that is one exabyte

EXAMPLE: SIMILARITY JOIN

MapReduce: Bad Idea

- ▶ Use keys (i, j) for pair of pictures $(i, P_i), (j, P_j)$
- ▶ *Map*: generates $((i, j), [P_i, P_j])$ as input for
- ▶ *Reduce*: computes $s(P_i, P_j)$ and decides whether $s(P_i, P_j) \geq t$
- ▶ Reducer size q is small: 2 MB; expected to fit in main memory
- ▶ *However*, each picture makes part of 999 999 key-value pairs, so

$$r = 999\,999$$

- ▶ Hence, number of bytes communicated from Map to Reduce is

$$10^6 \times 999\,999 \times 10^6 = 10^{18}$$

that is one exabyte

EXAMPLE: SIMILARITY JOIN

MapReduce: Bad Idea

- ▶ Use keys (i, j) for pair of pictures $(i, P_i), (j, P_j)$
- ▶ *Map*: generates $((i, j), [P_i, P_j])$ as input for
- ▶ *Reduce*: computes $s(P_i, P_j)$ and decides whether $s(P_i, P_j) \geq t$
- ▶ Reducer size q is small: 2 MB; expected to fit in main memory
- ▶ *However*, each picture makes part of 999 999 key-value pairs, so

$$r = 999\,999$$

- ▶ Hence, number of bytes communicated from Map to Reduce is

$$10^6 \times 999\,999 \times 10^6 = 10^{18}$$

that is one exabyte

EXAMPLE: SIMILARITY JOIN

MapReduce: Bad Idea

- ▶ Use keys (i, j) for pair of pictures $(i, P_i), (j, P_j)$
- ▶ *Map*: generates $((i, j), [P_i, P_j])$ as input for
- ▶ *Reduce*: computes $s(P_i, P_j)$ and decides whether $s(P_i, P_j) \geq t$
- ▶ Reducer size q is small: 2 MB; expected to fit in main memory
- ▶ *However*, each picture makes part of 999 999 key-value pairs, so

$$r = 999\,999$$

- ▶ Hence, number of bytes communicated from Map to Reduce is

$$10^6 \times 999\,999 \times 10^6 = 10^{18}$$

that is one exabyte

EXAMPLE: SIMILARITY JOIN

MapReduce: Bad Idea

- ▶ Use keys (i, j) for pair of pictures $(i, P_i), (j, P_j)$
- ▶ *Map*: generates $((i, j), [P_i, P_j])$ as input for
- ▶ *Reduce*: computes $s(P_i, P_j)$ and decides whether $s(P_i, P_j) \geq t$
- ▶ Reducer size q is small: 2 MB; expected to fit in main memory
- ▶ *However*, each picture makes part of 999 999 key-value pairs, so

$$r = 999\,999$$

- ▶ Hence, number of bytes communicated from Map to Reduce is

$$10^6 \times 999\,999 \times 10^6 = 10^{18}$$

that is one exabyte



EXAMPLE: SIMILARITY JOIN

MapReduce: Better Idea

- ▶ Group images into g groups, each of $10^6/g$ pictures
- ▶ *Map*: For each (i, P_i) generate $g - 1$ key-value pairs
 - ▶ Let u be group of P_i
 - ▶ Let v be one of the other groups
 - ▶ Keys are sets $\{u, v\}$ (set, so no order!)
 - ▶ Value is (i, P_i)
 - ▶ Overall: $(\{u, v\}, (i, P_i))$ as key-value pair
- ▶ *Reduce*: Consider key $\{u, v\}$
 - ▶ Associated value list has $2 \times \frac{10^6}{g}$ values
 - ▶ Consider (i, P_i) and (j, P_j) when i, j are from different groups
 - ▶ Compute $s(P_i, P_j)$
 - ▶ Compute $s(P_i, P_j)$ for P_i, P_j from same group on processing keys $\{u, u + 1\}$

EXAMPLE: SIMILARITY JOIN

MapReduce: Better Idea

- ▶ Group images into g groups, each of $10^6/g$ pictures
- ▶ *Map*: For each (i, P_i) generate $g - 1$ key-value pairs
 - ▶ Let u be group of P_i
 - ▶ Let v be one of the other groups
 - ▶ Keys are sets $\{u, v\}$ (set, so no order!)
 - ▶ Value is (i, P_i)
 - ▶ Overall: $(\{u, v\}, (i, P_i))$ as key-value pair
- ▶ *Reduce*: Consider key $\{u, v\}$
 - ▶ Associated value list has $2 \times \frac{10^6}{g}$ values
 - ▶ Consider (i, P_i) and (j, P_j) when i, j are from different groups
 - ▶ Compute $s(P_i, P_j)$
 - ▶ Compute $s(P_i, P_j)$ for P_i, P_j from same group on processing keys $\{u, u + 1\}$

EXAMPLE: SIMILARITY JOIN

MapReduce: Better Idea

- ▶ Group images into g groups, each of $10^6/g$ pictures
- ▶ *Map*: For each (i, P_i) generate $g - 1$ key-value pairs
 - ▶ Let u be group of P_i
 - ▶ Let v be one of the other groups
 - ▶ Keys are sets $\{u, v\}$ (set, so no order!)
 - ▶ Value is (i, P_i)
 - ▶ Overall: $(\{u, v\}, (i, P_i))$ as key-value pair
- ▶ *Reduce*: Consider key $\{u, v\}$
 - ▶ Associated value list has $2 \times \frac{10^6}{g}$ values
 - ▶ Consider (i, P_i) and (j, P_j) when i, j are from different groups
 - ▶ Compute $s(P_i, P_j)$
 - ▶ Compute $s(P_i, P_j)$ for P_i, P_j from same group on processing keys $\{u, u + 1\}$

EXAMPLE: SIMILARITY JOIN

MapReduce: Better Idea

- ▶ *Replication rate* is $g - 1$
 - ▶ Each input element (i, P_i) is turned into $g - 1$ key-value pairs
- ▶ *Reducer size* is $2 \times \frac{10^6}{g}$
 - ▶ Number of values on list for reducer
 - ▶ This yields $2 \times \frac{10^6}{g} \times 10^6$ bytes stored at Reducer node

EXAMPLE: SIMILARITY JOIN

MapReduce: Better Idea

- ▶ *Replication rate* is $g - 1$
 - ▶ Each input element (i, P_i) is turned into $g - 1$ key-value pairs
- ▶ *Reducer size* is $2 \times \frac{10^6}{g}$
 - ▶ Number of values on list for reducer
 - ▶ This yields $2 \times \frac{10^6}{g} \times 10^6$ bytes stored at Reducer node

EXAMPLE: SIMILARITY JOIN

MapReduce: Better Idea

▶ *Example* $g = 1000$:

- ▶ Input is 2 GB, fits into main memory
- ▶ Communication cost:

$$\underbrace{(10^3 \times 999)}_{\text{number of reducers}} \times \underbrace{(2 \times 10^3 \times 10^6)}_{\text{reducer size}} \approx 10^{15} \quad (2)$$

- ▶ 1000 times less than brute-force
 - ▶ Half a million reducers: maximum parallelism at Reduce nodes
- ▶ *Computation cost* is independent of g
- ▶ Always all-vs-all comparison of pictures
 - ▶ Computing $s(P_i, P_j)$ for all i, j

EXAMPLE: SIMILARITY JOIN

MapReduce: Better Idea

▶ *Example* $g = 1000$:

▶ Input is 2 GB, fits into main memory

▶ Communication cost:

$$\underbrace{(10^3 \times 999)}_{\text{number of reducers}} \times \underbrace{(2 \times 10^3 \times 10^6)}_{\text{reducer size}} \approx 10^{15} \quad (2)$$

▶ 1000 times less than brute-force

▶ Half a million reducers: maximum parallelism at Reduce nodes

▶ *Computation cost* is independent of g

▶ Always all-vs-all comparison of pictures

▶ Computing $s(P_i, P_j)$ for all i, j

EXAMPLE: SIMILARITY JOIN

MapReduce: Better Idea

▶ *Example* $g = 1000$:

- ▶ Input is 2 GB, fits into main memory
- ▶ Communication cost:

$$\underbrace{(10^3 \times 999)}_{\text{number of reducers}} \times \underbrace{(2 \times 10^3 \times 10^6)}_{\text{reducer size}} \approx 10^{15} \quad (2)$$

- ▶ 1000 times less than brute-force
- ▶ Half a million reducers: maximum parallelism at Reduce nodes
- ▶ *Computation cost* is independent of g
 - ▶ Always all-vs-all comparison of pictures
 - ▶ Computing $s(P_i, P_j)$ for all i, j

EXAMPLE: SIMILARITY JOIN

MapReduce: Better Idea

▶ *Example* $g = 1000$:

- ▶ Input is 2 GB, fits into main memory
- ▶ Communication cost:

$$\underbrace{(10^3 \times 999)}_{\text{number of reducers}} \times \underbrace{(2 \times 10^3 \times 10^6)}_{\text{reducer size}} \approx 10^{15} \quad (2)$$

- ▶ 1000 times less than brute-force
 - ▶ Half a million reducers: maximum parallelism at Reduce nodes
- ▶ *Computation cost* is independent of g
- ▶ Always all-vs-all comparison of pictures
 - ▶ Computing $s(P_i, P_j)$ for all i, j

EXAMPLE: SIMILARITY JOIN

MapReduce: Better Idea

▶ *Example* $g = 1000$:

- ▶ Input is 2 GB, fits into main memory
- ▶ Communication cost:

$$\underbrace{(10^3 \times 999)}_{\text{number of reducers}} \times \underbrace{(2 \times 10^3 \times 10^6)}_{\text{reducer size}} \approx 10^{15} \quad (2)$$

- ▶ 1000 times less than brute-force
 - ▶ Half a million reducers: maximum parallelism at Reduce nodes
- ▶ *Computation cost* is independent of g
- ▶ Always all-vs-all comparison of pictures
 - ▶ Computing $s(P_i, P_j)$ for all i, j

MAPREDUCE: GRAPH MODEL

Goal: Proving lower bounds on replication rate as function of reducer size, for many problems. Therefore:

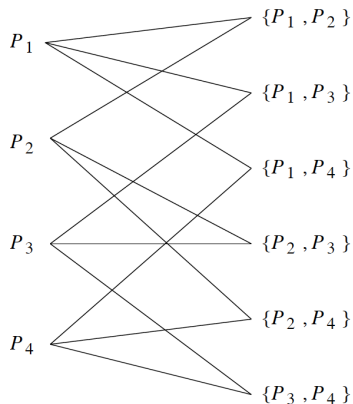
MAPREDUCE: GRAPH MODEL

Goal: Proving lower bounds on replication rate as function of reducer size, for many problems. Therefore:

Graph Model:

- ▶ Graph describes how outputs depend on inputs
- ▶ Reducers operate independently: each output has one reducer that receives all input required to compute output
- ▶ *Model foundation:*
 - ▶ There is a set of inputs
 - ▶ There is a set of outputs
 - ▶ Outputs depends on inputs: many-to-many relationship

MAPREDUCE: GRAPH MODEL EXAMPLE



Graph for similarity join with four pictures

Adopted from mmds.org

MAPREDUCE: GRAPH MODEL MATRIX MULTIPLICATION

Graph Model Matrix Multiplication

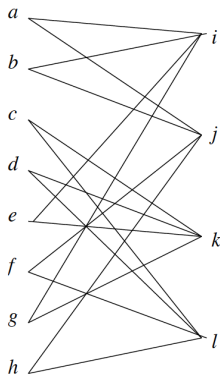
- ▶ Multiplying $n \times n$ matrices M and N makes
 - ▶ $2n^2$ inputs $m_{ij}, n_{jk}, 1 \leq i, j, k \leq n$
 - ▶ n^2 outputs $p_{ik} := (MN)_{ik}, 1 \leq i, k \leq n$
- ▶ Each output p_{ik} needs $2n$ inputs $m_{i1}, m_{i2}, \dots, m_{in}$ and $n_{1k}, n_{2k}, \dots, n_{nk}$
- ▶ Each input relates to n outputs: e.g. m_{ij} to $p_{i1}, p_{i2}, \dots, p_{in}$

MAPREDUCE: GRAPH MODEL MATRIX MULTIPLICATION

Graph Model Matrix Multiplication

- ▶ Multiplying $n \times n$ matrices M and N makes
 - ▶ $2n^2$ inputs $m_{ij}, n_{jk}, 1 \leq i, j, k \leq n$
 - ▶ n^2 outputs $p_{ik} := (MN)_{ik}, 1 \leq i, k \leq n$
- ▶ Each output p_{ik} needs $2n$ inputs $m_{i1}, m_{i2}, \dots, m_{in}$ and $n_{1k}, n_{2k}, \dots, n_{nk}$
- ▶ Each input relates to n outputs: e.g. m_{ij} to $p_{i1}, p_{i2}, \dots, p_{in}$

MAPREDUCE: GRAPH MODEL MATRIX MULTIPLICATION II



$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} i & j \\ k & l \end{bmatrix}$$

Input-output relationship graph for multiplying 2x2 matrices

Adopted from mmds.org

MAPREDUCE: MAPPING SCHEMAS

A *mapping schema* with a given reducer size q is an assignment of inputs to reducers such that

- ▶ No reducer receives more than q inputs
- ▶ For every output, there is a reducer that receives all inputs required to generate the output

MAPREDUCE: MAPPING SCHEMAS

A *mapping schema* with a given reducer size q is an assignment of inputs to reducers such that

- ▶ No reducer receives more than q inputs
- ▶ For every output, there is a reducer that receives all inputs required to generate the output

Consideration: The existence of a mapping schema for a given q distinguishes problems that can be solved in a *single* MapReduce job from those that cannot.

MAPPING SCHEMA: EXAMPLE

Consider computing similarity of p pictures, divided into g groups.

- ▶ Number of outputs: $\binom{p}{2} = \frac{p(p-1)}{2} \approx \frac{p^2}{2}$
- ▶ Reducer receives $2p/g$ inputs
 - ☞ necessary reducer size is $q = 2p/g$
- ▶ Replication rate is $r = g - 1 \approx g$:

$$r = 2p/q$$

☞ r inversely proportional to q which is common

- ▶ In a mapping schema for reducer size q :
 - ▶ Each reducer is assigned exactly $2p/g$ inputs
 - ▶ In all cases, every output is covered by some reducer

MAPPING SCHEMA: EXAMPLE

Consider computing similarity of p pictures, divided into g groups.

- ▶ Number of outputs: $\binom{p}{2} = \frac{p(p-1)}{2} \approx \frac{p^2}{2}$
- ▶ Reducer receives $2p/g$ inputs
 - ☞ necessary reducer size is $q = 2p/g$
- ▶ Replication rate is $r = g - 1 \approx g$:

$$r = 2p/q$$

☞ r inversely proportional to q which is common

- ▶ In a mapping schema for reducer size q :
 - ▶ Each reducer is assigned exactly $2p/g$ inputs
 - ▶ In all cases, every output is covered by some reducer

MAPPING SCHEMA: EXAMPLE

Consider computing similarity of p pictures, divided into g groups.

- ▶ Number of outputs: $\binom{p}{2} = \frac{p(p-1)}{2} \approx \frac{p^2}{2}$
- ▶ Reducer receives $2p/g$ inputs
 - ☞ necessary reducer size is $q = 2p/g$
- ▶ Replication rate is $r = g - 1 \approx g$:

$$r = 2p/q$$

☞ r inversely proportional to q which is common

- ▶ In a mapping schema for reducer size q :
 - ▶ Each reducer is assigned exactly $2p/g$ inputs
 - ▶ In all cases, every output is covered by some reducer

MAPPING SCHEMAS: NOT ALL INPUTS PRESENT

Example: Natural Join $R(A, B) \bowtie S(B, C)$, where many possible tuples $R(a, b), S(b, c)$ are missing.

- ▶ Theoretically $q = |A| \cdot |C|$ (keys were $b \in B$)
- ▶ But in practice many tuples $(a, b), (b, c)$ are missing for each b , so q possibly much smaller than $|A| \cdot |C|$

MAPPING SCHEMAS: NOT ALL INPUTS PRESENT

Example: Natural Join $R(A, B) \bowtie S(B, C)$, where many possible tuples $R(a, b), S(b, c)$ are missing.

- ▶ Theoretically $q = |A| \cdot |C|$ (keys were $b \in B$)
- ▶ But in practice many tuples $(a, b), (b, c)$ are missing for each b , so q possibly much smaller than $|A| \cdot |C|$

Main Consideration: One can increase q because of the missing inputs, without that inputs do no longer fit into main memory in practice

MAPPING SCHEMAS: LOWER BOUNDS ON REPLICATION RATE

Technique for proving lower bounds on replication rates

1. Prove upper bound $g(q)$ on how many outputs a reducer with q inputs can cover
 - ☞ This may be difficult in some cases
2. Determine total number of outputs O
3. Let there be k reducers with $q_i < q, i = 1, \dots, k$ inputs
 - ☞ observe that $\sum_{i=1}^k g(q_i)$ needs to be no less than O
4. Manipulate the inequality $\sum_{i=1}^k g(q_i) \geq O$ to get a lower bound on $\sum_{i=1}^k q_i$
5. Dividing the lower bound on $\sum_{i=1}^k q_i$ by number of inputs is lower bound on replication rate

MAPPING SCHEMAS: LOWER BOUNDS ON REPLICATION RATE

Technique for proving lower bounds on replication rates

1. Prove upper bound $g(q)$ on how many outputs a reducer with q inputs can cover
 - ☞ This may be difficult in some cases
2. Determine total number of outputs O
3. Let there be k reducers with $q_i < q, i = 1, \dots, k$ inputs
 - ☞ observe that $\sum_{i=1}^k g(q_i)$ needs to be no less than O
4. Manipulate the inequality $\sum_{i=1}^k g(q_i) \geq O$ to get a lower bound on $\sum_{i=1}^k q_i$
5. Dividing the lower bound on $\sum_{i=1}^k q_i$ by number of inputs is lower bound on replication rate

MAPPING SCHEMAS: LOWER BOUNDS ON REPLICATION RATE

Technique for proving lower bounds on replication rates

1. Prove upper bound $g(q)$ on how many outputs a reducer with q inputs can cover
 - ☞ This may be difficult in some cases
2. Determine total number of outputs O
3. Let there be k reducers with $q_i < q, i = 1, \dots, k$ inputs
 - ☞ observe that $\sum_{i=1}^k g(q_i)$ needs to be no less than O
4. Manipulate the inequality $\sum_{i=1}^k g(q_i) \geq O$ to get a lower bound on $\sum_{i=1}^k q_i$
5. Dividing the lower bound on $\sum_{i=1}^k q_i$ by number of inputs is lower bound on replication rate

MAPPING SCHEMAS: LOWER BOUNDS ON REPLICATION RATE

Technique for proving lower bounds on replication rates

1. Prove upper bound $g(q)$ on how many outputs a reducer with q inputs can cover
 - ☞ This may be difficult in some cases
2. Determine total number of outputs O
3. Let there be k reducers with $q_i < q, i = 1, \dots, k$ inputs
 - ☞ observe that $\sum_{i=1}^k g(q_i)$ needs to be no less than O
4. Manipulate the inequality $\sum_{i=1}^k g(q_i) \geq O$ to get a lower bound on $\sum_{i=1}^k q_i$
5. Dividing the lower bound on $\sum_{i=1}^k q_i$ by number of inputs is lower bound on replication rate

MAPPING SCHEMAS: LOWER BOUNDS ON REPLICATION RATE

Technique for proving lower bounds on replication rates

1. Prove upper bound $g(q)$ on how many outputs a reducer with q inputs can cover
 - ☞ This may be difficult in some cases
2. Determine total number of outputs O
3. Let there be k reducers with $q_i < q, i = 1, \dots, k$ inputs
 - ☞ observe that $\sum_{i=1}^k g(q_i)$ needs to be no less than O
4. Manipulate the inequality $\sum_{i=1}^k g(q_i) \geq O$ to get a lower bound on $\sum_{i=1}^k q_i$
5. Dividing the lower bound on $\sum_{i=1}^k q_i$ by number of inputs is lower bound on replication rate

LOWER BOUNDS: EXAMPLE ALL-PAIRS PROBLEM

- ▶ Recall that $r \leq 2p/q$ was upper bound on replication rate for all-pairs problem
- ▶ *Here:* Lower bound on r that is half the upper bound

LOWER BOUNDS: EXAMPLE ALL-PAIRS PROBLEM

► *Steps from slide before:*

- Step 1: reducer with q inputs cannot cover more than $\binom{q}{2} \approx q^2/2$ outputs
- Step 2: overall $\binom{p}{2} \approx p^2/2$ outputs must be covered
- Step 3: So, the inequality approximately evaluates as

$$\sum_{i=1}^k q_i^2/2 \geq p^2/2 \quad \iff \quad \sum_{i=1}^k q_i^2 \geq p^2$$

- Step 4: From $q \geq q_i$, we obtain

$$q \sum_{i=1}^k q_i \geq p^2 \quad \iff \quad \sum_{i=1}^k q_i \geq \frac{p^2}{q}$$

- Step 5: Noting that $r = (\sum_{i=1}^k q_i)/p$, we obtain

$$r \geq \frac{p}{q}$$

which is half the size of upper bound

LOWER BOUNDS: EXAMPLE ALL-PAIRS PROBLEM

► *Steps from slide before:*

- Step 1: reducer with q inputs cannot cover more than $\binom{q}{2} \approx q^2/2$ outputs
- Step 2: overall $\binom{p}{2} \approx p^2/2$ outputs must be covered
- Step 3: So, the inequality approximately evaluates as

$$\sum_{i=1}^k q_i^2/2 \geq p^2/2 \quad \iff \quad \sum_{i=1}^k q_i^2 \geq p^2$$

- Step 4: From $q \geq q_i$, we obtain

$$q \sum_{i=1}^k q_i \geq p^2 \quad \iff \quad \sum_{i=1}^k q_i \geq \frac{p^2}{q}$$

- Step 5: Noting that $r = (\sum_{i=1}^k q_i)/p$, we obtain

$$r \geq \frac{p}{q}$$

which is half the size of upper bound

LOWER BOUNDS: EXAMPLE ALL-PAIRS PROBLEM

► *Steps from slide before:*

- Step 1: reducer with q inputs cannot cover more than $\binom{q}{2} \approx q^2/2$ outputs
- Step 2: overall $\binom{p}{2} \approx p^2/2$ outputs must be covered
- Step 3: So, the inequality approximately evaluates as

$$\sum_{i=1}^k q_i^2/2 \geq p^2/2 \quad \iff \quad \sum_{i=1}^k q_i^2 \geq p^2$$

► Step 4: From $q \geq q_i$, we obtain

$$q \sum_{i=1}^k q_i \geq p^2 \quad \iff \quad \sum_{i=1}^k q_i \geq \frac{p^2}{q}$$

► Step 5: Noting that $r = (\sum_{i=1}^k q_i)/p$, we obtain

$$r \geq \frac{p}{q}$$

which is half the size of upper bound

LOWER BOUNDS: EXAMPLE ALL-PAIRS PROBLEM

► *Steps from slide before:*

- Step 1: reducer with q inputs cannot cover more than $\binom{q}{2} \approx q^2/2$ outputs
- Step 2: overall $\binom{p}{2} \approx p^2/2$ outputs must be covered
- Step 3: So, the inequality approximately evaluates as

$$\sum_{i=1}^k q_i^2/2 \geq p^2/2 \quad \iff \quad \sum_{i=1}^k q_i^2 \geq p^2$$

► Step 4: From $q \geq q_i$, we obtain

$$q \sum_{i=1}^k q_i \geq p^2 \quad \iff \quad \sum_{i=1}^k q_i \geq \frac{p^2}{q}$$

► Step 5: Noting that $r = (\sum_{i=1}^k q_i)/p$, we obtain

$$r \geq \frac{p}{q}$$

which is half the size of upper bound

LOWER BOUNDS: EXAMPLE ALL-PAIRS PROBLEM

► *Steps from slide before:*

- Step 1: reducer with q inputs cannot cover more than $\binom{q}{2} \approx q^2/2$ outputs
- Step 2: overall $\binom{p}{2} \approx p^2/2$ outputs must be covered
- Step 3: So, the inequality approximately evaluates as

$$\sum_{i=1}^k q_i^2/2 \geq p^2/2 \quad \iff \quad \sum_{i=1}^k q_i^2 \geq p^2$$

► Step 4: From $q \geq q_i$, we obtain

$$q \sum_{i=1}^k q_i \geq p^2 \quad \iff \quad \sum_{i=1}^k q_i \geq \frac{p^2}{q}$$

► Step 5: Noting that $r = (\sum_{i=1}^k q_i)/p$, we obtain

$$r \geq \frac{p}{q}$$

which is half the size of upper bound

MATERIALS / OUTLOOK

- ▶ See *Mining of Massive Datasets*, chapter 2.4–2.5
- ▶ For deepening your understanding, voluntary *homework*: please read through 2.6.7
- ▶ As usual, see <http://www.mmds.org/> in general for further resources
- ▶ Next lecture: “MapReduce / Workflow Systems III; Mining Data Streams I”
 - ▶ See *Mining of Massive Datasets* 2.6; 4.1–4.7