Tutorial: MapReduce

Theory and Practice of Data-intensive Applications

Pietro Michiardi

Eurecom

Introduction

What is MapReduce

- **A programming model**:
  - Inspired by functional programming
  - Allows expressing distributed computations on massive amounts of data
- **An execution framework**:
  - Designed for large-scale data processing
  - Designed to run on clusters of commodity hardware

What is this Tutorial About

- **Design of scalable algorithms with MapReduce**
  - Applied algorithm design and case studies
- **In-depth description of MapReduce**
  - Principles of functional programming
  - The execution framework
- **In-depth description of Hadoop**
  - Architecture internals
  - Software components
  - Cluster deployments

Motivations

Big Data

- **Vast repositories of data**
  - Web-scale processing
  - Behavioral data
  - Physics
  - Astronomy
  - Finance

- **"The fourth paradigm" of science** [6]
  - Data-intensive processing is fast becoming a necessity
  - Design algorithms capable of scaling to real-world datasets

- **It's not the algorithm, it's the data!** [2]
  - More data leads to better accuracy
  - With more data, accuracy of different algorithms converges

Key Ideas Behind MapReduce

Scale out, not up!

- **For data-intensive workloads, a large number of commodity servers is preferred over a small number of high-end servers**
  - Cost of super-computers is not linear
  - But datacenter efficiency is a difficult problem to solve [3, 5]

- **Some numbers:**
  - Data processed by Google every day: 20 PB
  - Data processed by Facebook every day: 15 TB

Implications of Scaling Out

- **Processing data is quick, I/O is very slow**
  - 1 HDD = 75 MB/sec
  - 1000 HDDs = 75 GB/sec
- **Sharing vs. Shared nothing**:
  - High-performance computing focus: distribute the *workload*
  - Shared nothing focus: distribute the *data*
- **Sharing is difficult**:
  - Synchronization, deadlocks
  - Finite bandwidth to access data from SAN
  - Temporal dependencies are complicated (restarts)

Failures are the norm, not the exception

- LALN data [DSN 2006]
  - Data for 5000 machines, for 9 years
  - Hardware: 60%, Software: 20%, Network 5%
- DRAM error analysis [Sigmetrics 2009]
  - Data for 2.5 years
  - 8% of DIMMs affected by errors
- Disk drive failure analysis [FAST 2007]
  - Utilization and temperature major causes of failures
- Amazon Web Service failure [April 2011]
  - Cascading effect

Implications of Failures

- **Failures are part of everyday life**
  - Mostly due to the scale and shared environment
- **Sources of Failures**
  - Hardware / Software
  - Preemption
  - Unavailability of a resource due to overload
- **Failure Types**
  - Permanent
  - Transient

Move Processing to the Data

- **Drastic departure from high-performance computing model**
  - HPC: distinction between processing nodes and storage nodes
  - HPC: CPU intensive tasks
- **Data intensive workloads**
  - Generally not processor demanding
  - The network becomes the bottleneck
  - MapReduce assumes processing and storage nodes to be colocated: *Data Locality*
- **Distributed filesystems are necessary**

EURECOM
Sophia Antipolis

Process Data Sequentially and Avoid Random Access

- **Data intensive workloads**
  - Relevant datasets are too large to fit in memory
  - Such data resides on disks
- **Disk performance is a bottleneck**
  - Seek times for random disk access are *the* problem
    - Example: 1 TB DB with 10^{10} 100-byte records. Updates on 1% requires 1 month, reading and rewriting the whole DB would take 1 day[1]
  - Organize computation for sequential reads

[1] From a post by Ted Dunning on the Hadoop mailing list

Implications of Data Access Patterns

- **MapReduce is designed for**
  - *batch processing*
  - involving (mostly) *full scans* of the dataset
- **Typically, data is collected "elsewhere" and copied to the distributed filesystem**
- **Data-intensive applications**
  - Read and process the whole Internet dataset from a crawler
  - Read and process the whole Social Graph

Hide System-level Details

- **Separate the *what* from the *how***
  - MapReduce abstracts away the "distributed" part of the system
  - Such details are handled by the framework
- **In-depth knowledge of the framework is key**
  - Custom data reader/writer
  - Custom data partitioning
  - Memory utilization
- **Auxiliary components**
  - Hadoop Pig
  - Hadoop Hive
  - Cascading

Seamless Scalability

- **We can define scalability along two dimensions**
  - In terms of data: given twice the amount of data, the same algorithm should take no more than twice as long to run
  - In terms of resources: given a cluster twice the size, the same algorithm should take no more than half as long to run
- **Embarassingly parallel problems**
  - Simple definition: independent (shared nothing) computations on fragments of the dataset
  - It's not easy to decide whether a problem is embarrassingly parallel or not
- **MapReduce is a first attempt, not the final answer**

Part One

The MapReduce Framework

Preliminaries

Divide and Conquer

- **A feasible approach to tackling large-data problems**
  - Partition a large problem into smaller sub-problems
  - Independent sub-problems executed in parallel
  - Combine intermediate results from each individual worker
- **The workers can be**:
  - Threads in a processor core
  - Cores in a multi-core processor
  - Multiple processors in a machine
  - Many machines in a cluster
- **Implementation details of divide and conquer are complex**

Divide and Conquer: How to?

- Decompose the original problem in smaller, parallel tasks
- Schedule tasks on workers distributed in a cluster
  - Data locality
  - Resource availability
- Ensure workers get the data they need?
- Coordinate synchronization among workers?
- Share partial results
- Handle failures?

The MapReduce Approach

- **Shared memory approach** (OpenMP, MPI, ...)
  - Developer needs to take care of (almost) everything
  - Synchronization, Concurrency
  - Resource allocation
- **MapReduce: a shared nothing approach**
  - Most of the above issues are taken care of
  - Problem decomposition and sharing partial results need particular attention
  - Optimizations (memory and network consumption) are tricky

The MapReduce Programming model

Functional Programming Roots

- **Key feature: higher order functions**
  - Functions that accept other functions as arguments
  - **Map** and **Fold**

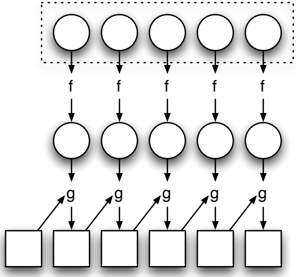


Figure: Illustration of *map* and *fold*.

Functional Programming Roots

- **map phase:**
  - Given a list, *map* takes as an argument a function f (that takes a single argument) and applies it to all element in a list
- **fold phase:**
  - Given a list, fold takes as arguments a function g (that takes two arguments) and an initial value
  - g is first applied to the initial value and the first item in the list
  - The result is stored in an intermediate variable, which is used as an input together with the next item to a second application of g
  - The process is repeated until all items in the list have been consumed

Functional Programming Roots

- **We can view map as a transformation over a dataset**
  - This transformation is specified by the function f
  - Each functional application happens in isolation
  - The application of f to each element of a dataset can be parallelized in a straightforward manner

- **We can view fold as an aggregation operation**
  - The aggregation is defined by the function g
  - Data locality: elements in the list must be “brought together”
  - If we can group element of the list, also the fold phase can proceed in parallel

- **Associative and commutative operations**
  - Allow performance gains through local aggregation and reordeing

Functional Programming and MapReduce

- **Equivalence of MapReduce and Functional Programming:**
  - The map of MapReduce corresponds to the map operation
  - The reduce of MapReduce corresponds to the fold operation

- **The framework coordinates the map and reduce phases:**
  - How intermediate results are grouped for the reduce to happen in parallel

- **In practice:**
  - User-specified computation is applied (in parallel) to all input records of a dataset
  - Intermediate results are aggregated by another user-specified computation

What can we do with MapReduce?

- **MapReduce "implements" a subset of functional programming**
  - The programming model appears quite limited
- **There are several important problems that can be adapted to MapReduce**
  - In this tutorial we will focus on illustrative cases
  - We will see in detail "design patterns"
    - How to transform a problem and its input
    - How to save memory and bandwidth in the system

Mappers and Reducers

Data Structures

- **Key-value pairs are the basic data structure in MapReduce**
  - Keys and values can be: integers, float, strings, raw bytes
  - They can also be arbitrary data structures

- **The design of MapReduce algorithms involes**:
  - Imposing the key-value structure on arbitrary datasets
    - E.g.: for a collection of Web pages, input keys may be URLs and values may be the HTML content
  - In some algorithms, input keys are not used, in others they uniquely identify a record
  - Keys can be combined in complex ways to design various algorithms

A MapReduce job

- **The programmer defines a mapper and a reducer as follows**[2]:
  - map: $(k_1, v_1) \rightarrow [(k_2, v_2)]$
  - reduce: $(k_2, [v_2]) \rightarrow [(k_3, v_3)]$

- **A MapReduce job consists in**:
  - A dataset stored on the underlying distributed filesystem, which is split in a number of files across machines
  - The mapper is applied to every input key-value pair to generate intermediate key-value pairs
  - The reducer is applied to all values associated with the same intermediate key to generate output key-value pairs

[2] We use the convention $[\cdots]$ to denote a list.

Where the magic happens

- **Implicit between the map and reduce phases is a distributed "group by" operation on intermediate keys**
  - Intermediate data arrive at each reducer in order, sorted by the key
  - No ordering is guaranteed across reducers

- **Output keys from reducers are written back to the distributed filesystem**
  - The output may consist of *r* distinct files, where *r* is the number of reducers
  - Such output may be the input to a subsequent MapReduce phase

- **Intermediate keys are transient**:
  - They are not stored on the distributed filesystem
  - They are "spilled" to the local disk of each machine in the cluster

A Simplified view of MapReduce

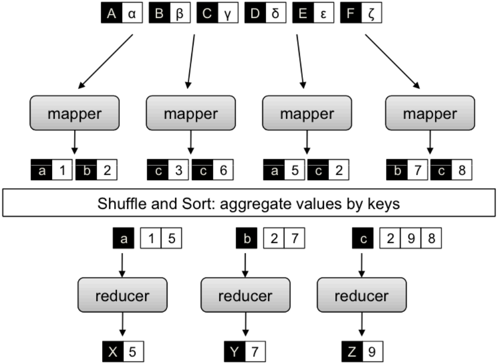


Figure: Mappers are applied to all input key-value pairs, to generate an arbitrary number of intermediate pairs. Reducers are applied to all intermediate values associated with the same intermediate key. Between the map and reduce phase lies a barrier that involves a large distributed sort and group by.

"Hello World" in MapReduce

```
class MAPPER
    method MAP(docid a, doc d)
        for all term t ∈ doc d do
            EMIT(term t, count 1)

class REDUCER
    method REDUCE(term t, counts [c1, c2, ...])
        sum ← 0
        for all count c ∈ counts [c1, c2, ...] do
            sum ← sum + c
        EMIT(term t, count sum)
```

Figure: Pseudo-code for the word count algorithm.

"Hello World" in MapReduce

- **Input:**
  - Key-value pairs: (docid, doc) stored on the distributed filesystem
  - docid: unique identifier of a document
  - doc: is the text of the document itself
- **Mapper:**
  - Takes an input key-value pair, tokenize the document
  - Emits intermediate key-value pairs: the word is the key and the integer is the value
- **The framework:**
  - Guarantees all values associated with the same key (the word) are brought to the same reducer
- **The reducer:**
  - Receives all values associated to some keys
  - Sums the values and writes output key-value pairs: the key is the word and the value is the number of occurrences

Implementation and Execution Details

- **The partitioner is in charge of assigning intermediate keys (words) to reducers**
  - Note that the partitioner can be customized
- **How many map and reduce tasks?**
  - The framework essentially takes care of map tasks
  - The designer/developer takes care of reduce tasks
- **In this tutorial we will focus on Hadoop**
  - Other implementations of the framework exist: Google, Disco, ...

Restrictions

- **Using external resources**
  - E.g.: Other data stores than the distributed file system
  - Concurrent access by many map/reduce tasks
- **Side effects**
  - Not allowed in functional programming
  - E.g.: preserving state across multiple inputs
  - State is kept internal
- **I/O and execution**
  - External side effects using distributed data stores (e.g. BigTable)
  - No input (e.g. computing π), no reducers, never no mappers

The Execution Framework

The Execution Framework

- **MapReduce program, a.k.a. a job**:
  - Code of mappers and reducers
  - Code for combiners and partitioners (optional)
  - Configuration parameters
  - All packaged together

- **A MapReduce job is submitted to the cluster**
  - The framework takes care of eveything else
  - Next, we will delve into the details

Scheduling

- **Each Job is broken into tasks**
  - Map tasks work on fractions of the input dataset, as defined by the underlying distributed filesystem
  - Reduce tasks work on intermediate inputs and write back to the distributed filesystem

- **The number of tasks may exceed the number of available machines in a cluster**
  - The scheduler takes care of maintaining something similar to a queue of pending tasks to be assigned to machines with available resources

- **Jobs to be executed in a cluster requires scheduling as well**
  - Different users may submit jobs
  - Jobs may be of various complexity
  - Fairness is generally a requirement

Scheduling

- **The scheduler component can be customized**
  - As of today, for Hadoop, there are various schedulers
- **Dealing with stragglers**
  - Job execution time depends on the slowest map and reduce tasks
  - Speculative execution can help with slow machines
    - But data locality may be at stake
- **Dealing with skew in the distribution of values**
  - E.g.: temperature readings from sensors
  - In this case, scheduling cannot help
  - It is possible to work on customized partitioning and sampling to solve such issues [Advanced Topic]

Data/code co-location

- **How to feed data to the code**
  - In MapReduce, this issue is intertwined with scheduling and the underlying distributed filesystem
- **How data locality is achieved**
  - The scheduler starts the task on the node that holds a particular block of data required by the task
  - If this is not possible, tasks are started elsewhere, and data will cross the network
    - Note that usually input data is replicated
  - Distance rules [11] help dealing with bandwidth consumption
    - Same rack scheduling

Synchronization

- In MapReduce, synchronization is achieved by the "shuffle and sort" bareer
  - Intermediate key-value pairs are grouped by key
  - This requires a distributed sort involving all mappers, and taking into account all reducers
  - If you have m mappers and r reducers this phase involves up to $m \times r$ copying operations

- **IMPORTANT: the reduce operation cannot start until all mappers have finished**
  - This is different from functional programming that allows "lazy" aggregation
  - In practice, a common optimization is for reducers to **pull** data from mappers as soon as they finish

Errors and faults

Using quite simple mechanisms, the MapReduce framework deals with:

- **Hardware failures**
  - Individual machines: disks, RAM
  - Networking equipment
  - Power / cooling
- **Software failures**
  - Exceptions, bugs
- **Corrupt and/or invalid input data**

Partitioners and Combiners

Partitioners

- **Partitioners are responsible for**:
  - Dividing up the intermediate key space
  - Assigning intermediate key-value pairs to reducers
  - → Specify the task to which an intermediate key-value pair must be copied

- **Hash-based partitioner**
  - Computes the hash of the key modulo the number of reducers r
  - This ensures a roughly even partitioning of the key space
    - ★ However, it ignores values: this can cause imbalance in the data processed by each reducer
  - When dealing with complex keys, even the base partitioner may need customization

Combiners

- **Combiners are an (optional) optimization**:
  - Allow local aggregation before the “shuffle and sort” phase
  - Each combiner operates in isolation
- **Essentially, combiners are used to save bandwidth**
  - E.g.: word count program
- **Combiners can be implemented using local data-structures**
  - E.g., an associative array keeps intermediate computations and aggregation thereof
  - The map function only emits once all input records (even all input splits) are processed

Partitioners and Combiners, an Illustration

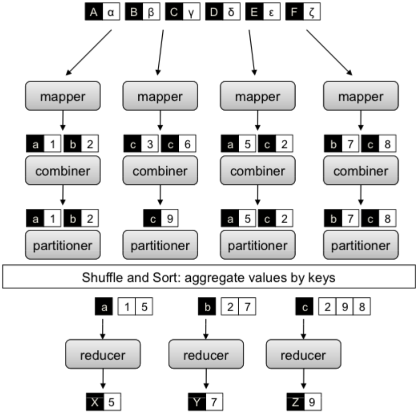


Figure: Complete view of MapReduce illustrating combiners and partitioners.

Note: in Hadoop, partitioners are executed before combiners.

The Distributed Filesystem

Colocate data and computation!

- **As dataset sizes increase, more computing capacity is required for processing**
- **As compute capacity grows, the link between the compute nodes and the storage nodes becomes a bottleneck**
  - One could eventually think of special-purpose interconnects for high-performance networking
  - This is often a costly solution as cost does not increase linearly with performance
- **Key idea: abandon the separation between compute and storage nodes**
  - This is exactly what happens in current implementations of the MapReduce framework
  - A distributed filesystem is not mandatory, but highly desirable

Distributed filesystems

- **In this tutorial we will focus on HDFS, the Hadoop implementation of the Google distributed filesystem (GFS)**

- **Distributed filesystems are not new!**
  - HDFS builds upon previous results, tailored to the specific requirements of MapReduce
  - Write once, read many workloads
  - Does not handle concurrency, but allow replication
  - Optimized for throughput, not latency

HDFS

- **Divide user data into blocks**
  - Blocks are big! [64, 128] MB
  - Avoids problems related to metadata management
- **Replicate blocks across the local disks of nodes in the cluster**
  - Replication is handled by storage nodes themselves (similar to chain replication) and follows distance rules
- **Master-slave architecture**
  - `NameNode`: master maintains the namespace (metadata, file to block mapping, location of blocks) and maintains overall health of the file system
  - `DataNode`: slaves manage the data blocks

HDFS, an Illustration

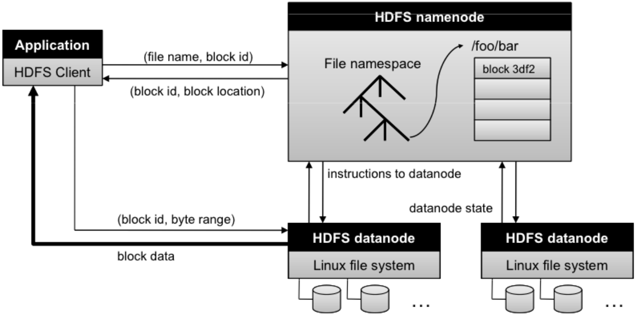


Figure: The architecture of HDFS.

HDFS I/O

- **A typical read from a client involves**:
  1. Contact the `NameNode` to determine where the actual data is stored
  2. `NameNode` replies with block identifiers and locations (*i.e.*, which `DataNode`)
  3. Contact the `DataNode` to fetch data

- **A typical write from a client involves**:
  1. Contact the `NameNode` to update the namespace and verify permissions
  2. `NameNode` allocates a new block on a suitable `DataNode`
  3. The client directly streams to the selected `DataNode`
  4. Currently, HDFS files are immutable

- **Data is never moved through the `NameNode`**
  - Hence, there is no bottleneck

HDFS Replication

- **By default, HDFS stores 3 sperate copies of each block**
  - This ensures reliability, availability and performance
- **Replication policy**
  - Spread replicas across differen racks
  - Robust against cluster node failures
  - Robust against rack failures
- **Block replication benefits MapReduce**
  - Scheduling decisions can take replicas into account
  - Exploit better data locality

HDFS: more on operational assumptions

- **A small number of large files is preferred over a large number of small files**
  - Metadata may explode
  - Input splits fo MapReduce based on individual files
    - → Mappers are launched for every file
    - ⋆ High startup costs
    - ⋆ Inefficient “shuffle and sort”
- **Workloads are batch oriented**
- **Not full POSIX**
- **Cooperative scenario**

Part Two

Hadoop implementation of MapReduce

Preliminaries

From Theory to Practice

- **The story so far**
  - Concepts behind the MapReduce Framework
  - Overview of the programming model
- **Hadoop implementation of MapReduce**
  - HDFS in details
  - Hadoop I/O
  - Hadoop MapReduce
    - Implementation details
    - Types and Formats
    - Features in Hadoop
  - Hadoop Streaming: Dumbo
- **Hadoop Deployments**

Terminology

- **MapReduce:**
  - `Job`: an execution of a Mapper and Reducer across a data set
  - `Task`: an execution of a Mapper or a Reducer on a slice of data
  - `Task Attempt`: instance of an attempt to execute a task
  - **Example:**
    - Running "Word Count" across 20 files is one job
    - 20 files to be mapped = 20 map tasks + some number of reduce tasks
    - At least 20 attempts will be performed... more if a machine crashes

- **Task Attempts**
  - Task attempted at least once, possibly more
  - Multiple crashes on input imply discarding it
  - Multiple attempts may occur in parallel (speculative execution)
  - Task ID from TaskInProgress is not a unique identifier

HDFS in details

The Hadoop Distributed Filesystem

- **Large dataset(s) outgrowing the storage capacity of a single physical machine**
  - Need to partition it across a number of separate machines
  - Network-based system, with all its complications
  - Tolerate failures of machines
- **Hadoop Distributed Filesystem[10, 11]**
  - Very large files
  - Streaming data access
  - Commodity hardware

HDFS Blocks

- **(Big) files are broken into block-sized chunks**
  - `NOTE`: A file that is smaller than a single block does not occupy a full block's worth of underlying storage
- **Blocks are stored on independent machines**
  - Reliability and parallel access
- **Why is a block so large?**
  - Make transfer times larger than seek latency
  - E.g.: Assume seek time is 10ms and the transfer rate is 100 MB/s, if you want seek time to be 1% of transfer time, then the block size should be 100MB

NameNodes and DataNodes

- **NameNode**
  - Keeps metadata in RAM
  - Each block information occupies roughly 150 bytes of memory
  - Without `NameNode`, the filesystem cannot be used
    - Persistence of metadata: synchronous and atomic writes to NFS
- **Secondary NameNode**
  - Merges the namespce with the edit log
  - A useful trick to recover from a failure of the `NameNode` is to use the NFS copy of metadata and switch the secondary to primary
- **DataNode**
  - They store data and talk to clients
  - They report periodically to the `NameNode` the list of blocks they hold

Anatomy of a File Read

- **NameNode is only used to get block location**
  - Unresponsive DataNode are discarded by clients
  - Batch reading of blocks is allowed
- **"External" clients**
  - For each block, the NameNode returns a set of DataNodes holding a copy thereof
  - DataNodes are sorted according to their proximity to the client
- **"MapReduce" clients**
  - TaskTracker and DataNodes are colocated
  - For each block, the NameNode usually[3] returns the local DataNode

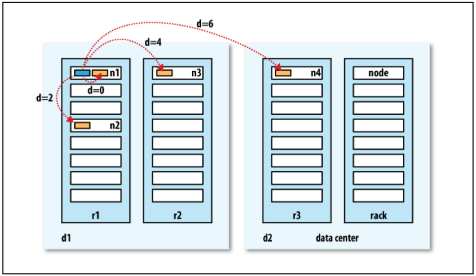
[3] Exceptions exist due to stragglers.

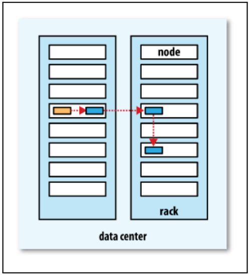
Anatomy of a File Write

- **Details on replication**
  - Clients ask `NameNode` for a list of suitable `DataNodes`
  - This list forms a `pipeline`: first `DataNode` stores a copy of a block, then forwards it to the second, and so on

- **Replica Placement**
  - Tradeoff between reliability and bandwidth
  - Default placement:
    - First copy on the "same" node of the client, second replica is off-rack, third replica is on the same rack as the second but on a different node
    - Since Hadoop 0.21, replica placement can be customized

Network Topology and HDFS





HDFS Coherency Model

- **Read your writes is not guaranteed**
  - The namespace is updated
  - Block contents may not be visible after a write is finished
  - Application design (other than MapReduce) should use `sync()` to force synchronization
  - `sync()` involves some overhead: tradeoff between robustness/consistency and throughput

- **Multiple writers (for the same block) are not supported**
  - Instead, different blocks can be written in parallel (using MapReduce)

Hadoop I/O

I/O operations in Hadoop

- **Reading and writing data**
  - From/to HDFS
  - From/to local disk drives
  - Across machines (inter-process communication)
- **Customized tools for large amounts of data**
  - Hadoop does not use Java native classes
  - Allows flexibility for dealing with custom data (e.g. binary)
- **What's next**
  - Overview of what Hadoop offers
  - For an in depth knowledge, use [11]

Data Integrity

- **Every I/O operation on disks or the network may corrupt data**
  - Users expect data not to be corrupted during storage or processing
  - Data integrity usually achieved with checksums
- **HDFS transparently checksums all data during I/O**
  - HDFS makes sure that storage overhead is roughly 1%
  - `DataNodes` are in charge of checksumming
    - With replication, the last replica performs the check
    - Checksums are timestamped and logged for statistcs on disks
  - Checksumming is also run periodically in a separate thread
    - Note that thanks to replication, error correction is possible

Compression

- **Why using compression**
  - Reduce storage requirements
  - Speed up data transfers (across the network or from disks)

- **Compression and Input Splits**
  - IMPORTANT: use compression that supports splitting (e.g. bzip2)

- **Splittable files, Example 1**
  - Consider an uncompressed file of 1GB
  - HDFS will split it in 16 blocks, 64MB each, to be processed by separate Mappers

Compression

- **Splittable files, Example 2 (gzip)**
  - Consider a compressed file of 1GB
  - HDFS will split it in 16 blocks of 64MB each
  - Creating an `InputSplit` for each block will not work, since it is not possible to read at an arbitrary point

- **What's the problem?**
  - This forces MapReduce to treat the file as a single split
  - Then, a single Mapper is fired by the framework
  - For this Mapper, only 1/16-th is local, the rest comes from the network

- **Which compression format to use?**
  - Use bzip2
  - Otherwise, use `SequenceFiles`
  - See Chapter 4 (page 84) [11]

Serialization

- **Transforms structured objects into a byte stream**
  - For transmission over the network: Hadoop uses RPC
  - For persistent storage on disks

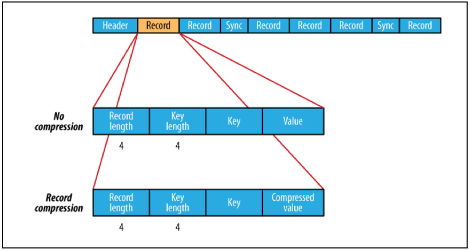
- **Hadoop uses its own serialization format, `Writable`**
  - Comparison of types is crucial (Shuffle and Sort phase): Hadoop provides a custom `RawComparator`, which avoids deserialization
  - Custom `Writable` for having full control on the binary representation of data
  - Also "external" frameworks are allowed: enter **Avro**

- **Fixed-lenght or variable-length encoding?**
  - Fixed-lenght: when the distribution of values is uniform
  - Variable-length: when the distribution of values is not uniform

Sequence Files

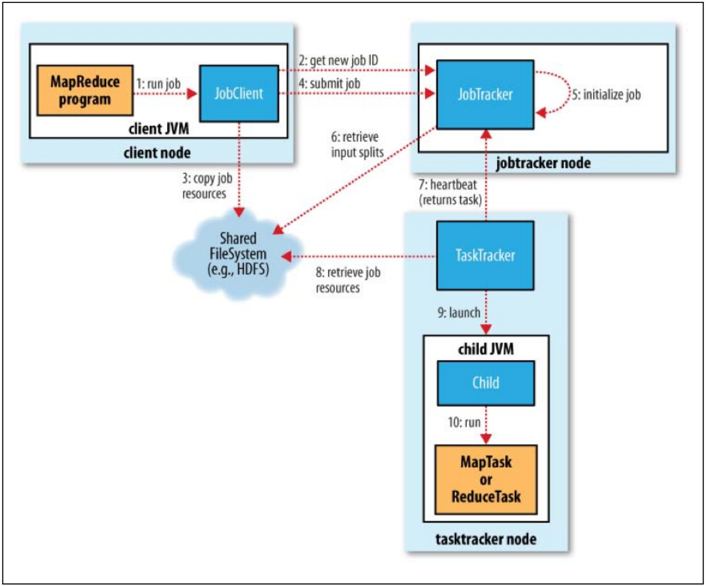
- **Specialized data structure to hold custom input data**
  - Using blobs of binaries is not efficient

- **`SequenceFiles`**
  - Provide a persistent data structure for binary key-value pairs
  - Also work well as containers for smaller files so that the framework is more happy (remember, better few large files than lots of small files)
  - They come with the `sync()` method to introduce sync points to help managing `InputSplits` for MapReduce



How Hadoop MapReduce Works

Anatomy of a MapReduce Job Run



Job Submission

- **`JobClient` class**
  - The `runJob()` method creates a new instance of a **JobClient**
  - Then it calls the `submitJob()` on this class
- **Simple verifications on the Job**
  - Is there an output directory?
  - Are there any input splits?
  - Can I copy the JAR of the job to HDFS?
- **NOTE: the JAR of the job is replicated 10 times**

Job Initialization

- **The `JobTracker` is responsible for:**
  - Create an object for the job
  - Encapsulate its tasks
  - Bookkeeping with the tasks' status and progress

- **This is where the scheduling happens**
  - `JobTracker` performs scheduling by maintaining a queue
  - Queueing disciplines are pluggable

- **Compute mappers and reducers**
  - `JobTracker` retrieves input splits (computed by `JobClient`)
  - Determines the number of Mappers based on the number of input splits
  - Reads the configuration file to set the number of Reducers

Task Assignment

- **Hearbeat-based mechanism**
  - `TaskTrackers` periodically send hearbeats to the `JobTracker`
  - `TaskTracker` is alive
  - Heartbeat contains also information on availability of the `TaskTrackers` to execute a task
  - `JobTracker` piggybacks a task if `TaskTracker` is available

- **Selecting a task**
  - `JobTracker` first needs to select a job (*i.e.* scheduling)
  - `TaskTrackers` have a fixed number of slots for map and reduce tasks
  - `JobTracker` gives priority to map tasks (WHY?)

- **Data locality**
  - `JobTracker` is topology aware
    - Useful for map tasks
    - Unused for reduce tasks

Task Execution

- **Task Assignement is done, now `TaskTrackers` can execute**
  - Copy the JAR from the HDFS
  - Create a local working directory
  - Create an instance of `TaskRunner`

- **`TaskRunner` launches a child JVM**
  - This prevents bugs from stalling the `TaskTracker`
  - A new child JVM is created per `InputSplit`
    - Can be overriden by specifying JVM Reuse option, which is very useful for custom, in-memory, combiners

- **Streaming and Pipes**
  - User-defined map and reduce methods need not to be in Java
  - Streaming and Pipes allow C++ or python mappers and reducers
  - We will cover Dumbo

Handling Failures

In the real world, code is buggy, processes crash and machine fails

- **Task Failure**
  - Case 1: map or reduce task throws a runtime exception
    - ★ The child JVM reports back to the parent `TaskTracker`
    - ★ `TaskTracker` logs the error and marks the TaskAttempt as failed
    - ★ `TaskTracker` frees up a slot to run another task
  - Case 2: Hanging tasks
    - ★ `TaskTracker` notices no progress updates (timeout = 10 minutes)
    - ★ `TaskTracker` kills the child JVM[4]
  - `JobTracker` is notified of a failed task
    - ★ Avoids rescheduling the task on the same `TaskTracker`
    - ★ If a task fails 4 times, it is not re-scheduled[5]
    - ★ Default behavior: if any task fails 4 times, the job fails

[4] With streaming, you need to take care of the orphaned process.

[5] Exception is made for speculative execution

Handling Failures

- **`TaskTracker` Failure**
  - Types: crash, running very slowly
  - Heartbeats will not be sent to `JobTracker`
  - `JobTracker` waits for a timeout (10 minutes), then it removes the `TaskTracker` from its scheduling pool
  - `JobTracker` needs to reschedule even *completed* tasks (WHY?)
  - `JobTracker` needs to reschedule tasks in progress
  - `JobTracker` may even blacklist a `TaskTracker` if too many tasks failed

- **`JobTracker` Failure**
  - Currently, Hadoop has no mechanism for this kind of failure
  - In future releases:
    - Multiple `JobTrackers`
    - Use ZooKeeper as a coordination mechanisms

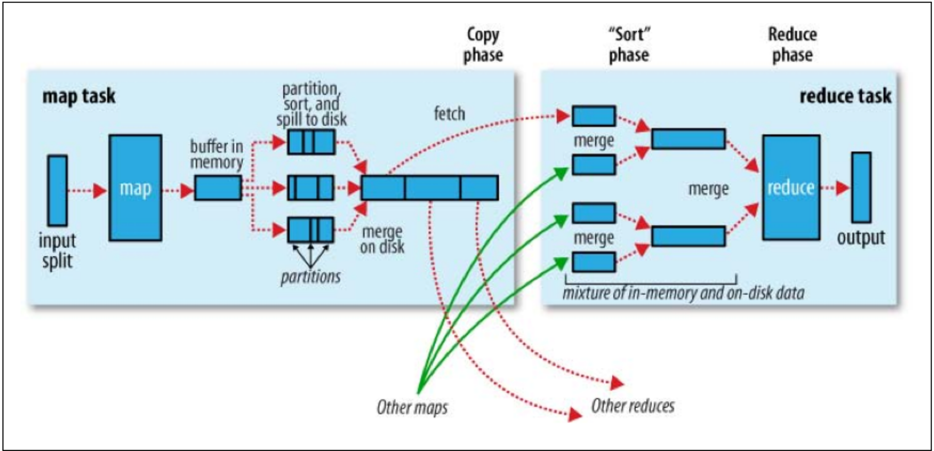
Scheduling

- **FIFO Scheduler (default behavior)**
  - Each job uses the whole cluster
  - Not suitable for shared production-level cluster
    - Long jobs monopolize the cluster
    - Short jobs can hold back and have no guarantees on execution time
- **Fair Scheduler**
  - Every user gets a fair share of the cluster capacity over time
  - Jobs are placed in to pools, one for each user
    - Users that submit more jobs have no more resources than oterhs
    - Can guarantee minimum capacity per pool
  - Supports preemption
  - "Contrib" module, requires manual installation
- **Capacity Scheduler**
  - Hierarchical queues (mimic an oragnization)
  - FIFO scheduling in each queue
  - Supports priority

Shuffle and Sort

- **The MapReduce framework guarantees the input to every reducer to be sorted by key**
  - The process by which the system sorts and transfers map outputs to reducers is known as shuffle
- **Shuffle is the most important part of the framework, where the "magic" happens**
  - Good understanding allows optimizing both the framework and the execution time of MapReduce jobs
- **Subject to continuous refinements**

Shuffle and Sort: the Map Side



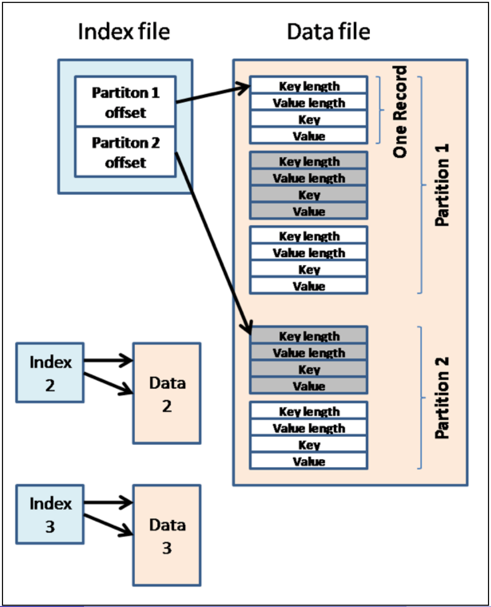
Shuffle and Sort: the Map Side

- **The output of a map task is not simply written to disk**
  - In memory buffering
  - Pre-sorting
- **Circular memory buffer**
  - 100 MB by default
  - Threshold based mechanism to spill buffer content to disk
  - Map output written to the buffer while spilling to disk
  - If buffer fills up while spilling, the map task is blocked
- **Disk spills**
  - Written in round-robin to a local dir
  - Output data is parttioned corresponding to the reducers they will be sent to
  - Within each partition, data is sorted (in-memory)
  - Optionally, if there is a combiner, it is executed just after the sort phase

Shuffle and Sort: the Map Side

- **More on spills and memory buffer**
  - Each time the buffer is full, a new spill is created
  - Once the map task finishes, there are many spills
  - Such spills are merged into a single partitioned and sorted output file
- **The output file partitions are made available to reducers over HTTP**
  - There are 40 (default) threads dedicated to serve the file partitions to reducers

Shuffle and Sort: the Map Side



Shuffle and Sort: the Reduce Side

- **The map output file is located on the local disk of tasktracker**
- **Another tasktracker (in charge of a reduce task) requires input from many other TaskTracker (that finished their map tasks)**
  - How do reducers know which tasktrackers to fetch map output from?
    - When a map task finishes it notifies the parent tasktracker
    - The tasktracker notifies (with the heartbeat mechanism) the jobtracker
    - A thread in the reducer polls periodically the jobtracker
    - Tasktrackers do not delete local map output as soon as a reduce task has fetched them (WHY?)
- **Copy phase: a pull approach**
  - There is a small number (5) of copy threads that can fetch map outputs in parallel

Shuffle and Sort: the Reduce Side

- **The map outputs are copied to the the trasktracker running the reducer in memory (if they fit)**
  - Otherwise they are copied to disk

- **Input consolidation**
  - A background thread merges all partial inputs into larger, sorted files
  - Note that if compression was used (for map outputs to save bandwidth), decompression will take place in memory

- **Sorting the input**
  - When all map outputs have been copied a merge phase starts
  - All map outputs are sorted maintaining their sort ordering, in rounds

Hadoop MapReduce Types and Formats

MapReduce Types

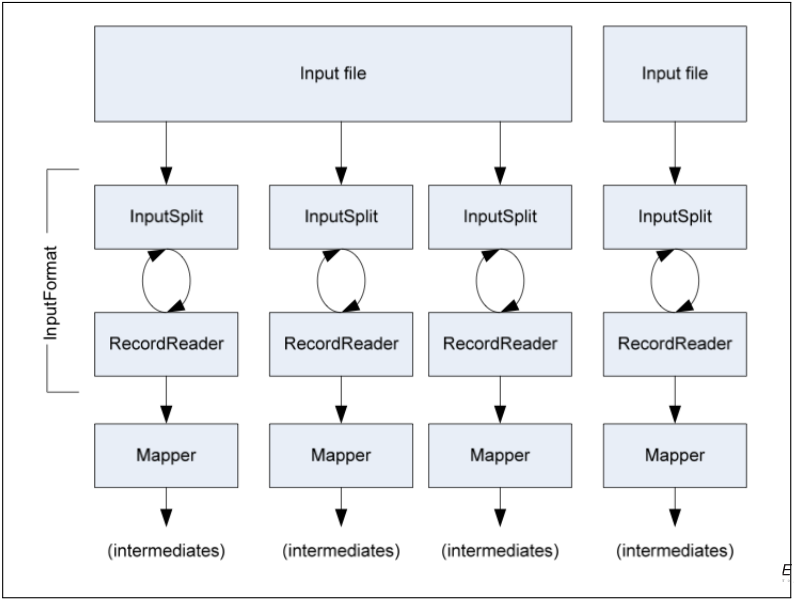
- **Input / output to mappers and reducers**
  - map: $(k1, v1) \rightarrow [(k2, v2)]$
  - reduce: $(k2, [v2]) \rightarrow [(k3, v3)]$
- **In Hadoop, a mapper is created as follows:**
  - `void map(K1 key, V1 value, OutputCollector<K2, V2> output, Reporter reporter)`
- **Types:**
  - K types implement `WritableComparable`
  - V types implement `Writable`

What is a `Writable`

- **Hadoop defines its own classes for strings (`Text`), integers (`intWritable`), etc...**
- **All keys are instances of `WritableComparable`**
  - Why comparable?
- **All values are instances of `Writable`**

EURECOM

Getting Data to the Mapper

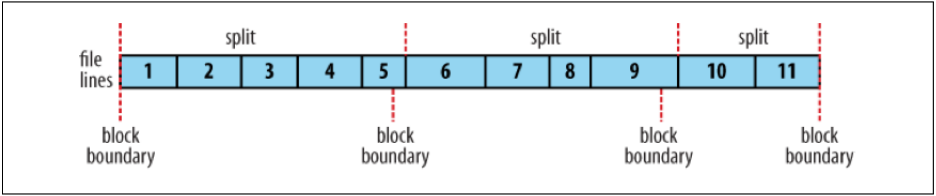


Reading Data

- **Datasets are specified by `InputFormats`**
  - `InputFormats` define input data (e.g. a file, a directory)
  - `InputFormats` is a factory for `RecordReader` objects to extract key-value records from the input source

- **`InputFormats` identify partitions of the data that form an `InputSplit`**
  - `InputSplit` is a (**reference to a**) chunk of the input processed by a single map
    - Largest split is processed first
  - Each split is divided into records, and the map processes each record (a key-value pair) in turn
  - Splits and records are logical, they are not physically bound to a file

The relationship between `InputSplit` and HDFS blocks



`FileInputFormat` and Friends

- **`TextInputFormat`**
  - Traeats each `newline`-terminated line of a file as a value
- **`KeyValueTextInputFormat`**
  - Maps `newline`-terminated text lines of “key” SEPARATOR “value”
- **`SequenceFileInputFormat`**
  - Binary file of key-value pairs with some additional metadata
- **`SequenceFileAsTextInputFormat`**
  - Same as before but, maps `(k.toString(), v.toString())`

Filtering File Inputs

- **`FileInputFormat` reads all files out of a specified directory and send them to the mapper**
- **Delegates filtering this file list to a method subclasses may override**
  - Example: create your own "`xyzFileInputFormat`" to read `*.xyz` from a directory list

Record Readers

- **Each `InputFormat` provides its own `RecordReader` implementation**
- **`LineRecordReader`**
  - Reads a line from a text file
- **`KeyValueRecordReader`**
  - Used by `KeyValueTextInputFormat`

Input Split Size

- **`FileInputFormat` divides large files into chunks**
  - Exact size controlled by `mapred.min.split.size`
- **Record readers receive file, offset, and length of chunk**
  - Example

On the top of the Crumpetty Tree→	(0, On the top of the Crumpetty Tree)
The Quangle Wangle sat,→	(33, The Quangle Wangle sat,)
But his face you could not see,→	(57, But his face you could not see,)
On account of his Beaver Hat.→	(89, On account of his Beaver Hat.)

- **Custom `InputFormat` implementaions may override split size**

Sending Data to Reducers

- **Map function receives `OutputCollector` object**
  - `OutputCollector.collect()` receives key-value elements
- **Any (`WritableComparable`, `Writable`) can be used**
- **By defalut, mapper output type assumed to be the same as the reducer output type**

WritableComparator

- **Compares `WritableComparable` data**
  - Will call the `WritableComparable.compare()` method
  - Can provide fast path for serialized data

- **Configured through:**
  `JobConf.setOutputValueGroupingComparator()`

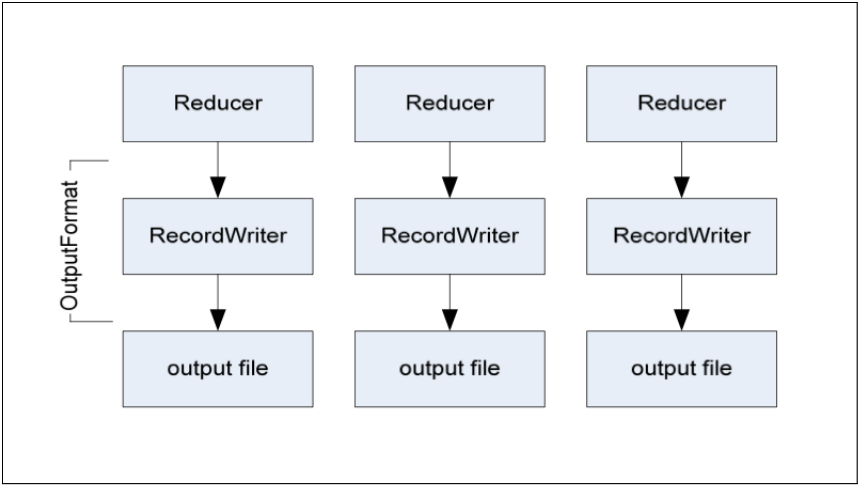
Partiotioner

- **`int getPartition(key, value, numPartitions)`**
  - Outputs the partition number for a given key
  - One partition == all values sent to a single reduce task
- **`HasPartitioner` used by default**
  - Uses `key.hashCode()` to return partion number
- **`JobConf` used to set `Partitioner` implementation**

The Reducer

- `void reduce(k2 key, Iterator<v2> values, OutputCollector<k3, v3> output, Reporter reporter )`
- **Keys and values sent to one partition all go to the same reduce task**
- **Calls are sorted by key**
  - "Early" keys are reduced and output before "late" keys

Writing the Output



Writing the Output

- **Analogous to `InputFormat`**
- **`TextOutputFormat` writes "key value `<newline>`" strings to output file**
- **`SequenceFileOutputFormat` uses a binary format to pack key-value pairs**
- **`NullOutputFormat` discards output**

Hadoop MapReduce Features

Developing a MapReduce Application

Preliminaries

- **Writing a program in MapReduce has a certain flow to it**
  - Start by writing the map and reduce functions
    - Write unit tests to make sure they do what they should
  - Write a driver program to run a job
    - The job can be run from the IDE using a small subset of the data
    - The debugger of the IDE can be used
  - Evenutally, you can unleash the job on a cluster
    - Debugging a distributed program is challenging

- **Once the job is running properly**
  - Perform standard checks to improve performance
  - Perform task profiling

Configuration

- **Before writing a MapReduce program, we need to set up and cofigure the development environment**
  - Components in Hadoop are configured with an ad hoc API
  - `Configuration` class is a collection of *properties* and their values
  - Resources can be combined into a configuration

- **Configuring the IDE**
  - In the IDE create a new project and add all the JAR files from the top level of the distribution and form the *lib* directory
  - For Eclipse there are also available plugins
  - Commercial IDE also exist (Karmasphere)

- **`Alternatives`**
  - Switch configurations (local, cluster)
  - `Alternatives` (see Cloudera documentation for Ubuntu) is very effective

Local Execution

- **Use the `GenericOptionsParser`, `Tool` and `ToolRunner`**
  - These helper classes makes it easy to intervene on job configurations
  - These are additional configurations to the `core` configuration
- **The `run()` method**
  - Constructs and configure a `JobConf` object and launch it
- **How many reducers?**
  - In a local execution, there is a single (eventually none) reducer
  - Even by setting a number of reducer larger than one, the option will be ignored

Cluster Execution

- **Packaging**
- **Launching a Job**
- **The WebUI**
- **Hadoop Logs**
- **Running Dependent Jobs, and Oozie**

Hadoop Deployments

Setting up a Hadoop Cluster

- **Cluster deployment**
  - Private cluster
  - Cloud-based cluster
  - AWS Elasitc MapReduce

- **Outlook:**
  - Cluster specification
    - Hardware
    - Network Topology
  - Hadoop Configuration
    - Memory considerations

Cluster Specification

- **Commodity Hardware**
  - Commodity $\neq$ Low-end
    - False economy due to failure rate and maintenance costs
  - Commodity $\neq$ High-end
    - High-end machines perform better, which would imply a smaller cluster
    - A single machine failure would compromise a large fraction of the cluster

- **A 2010 specification**:
  - 2 quad-cores
  - 16-24 GB ECC RAM
  - 4×1 TB SATA disks[6]
  - Gigabit Ethernet

[6] Why not using RAID instead of JBOD?

Cluster Specification

- **Example:**
  - Assume your data grows by 1 TB per week
  - Assume you have three-way replication in HDFS
  - → You need additional 3TB of raw storage per week
  - Allow for some overhead (temporary files, logs)
  - → This is a new machine per week

- **How to dimension a cluster?**
  - Obviously, you won't buy a machine per week!!
  - The idea is that the above back-of-the-envelope calculation is that you can project over a 2 year life-time of your system
  - → You would need a 100-machine cluster

- **Where should you put the various components?**
  - Small cluster: NameNode and JobTracker can be colocated
  - Large cluster: requires more RAM at the NameNode

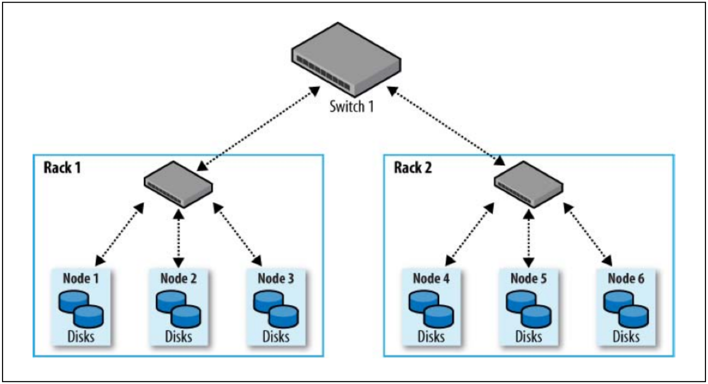
Cluster Specification

- **Should we use 64-bit or 32-bit machines?**
  - NameNode should run on a 64-bit machine: this avoids the 3GB Java heap size limit on 32-bit machines
  - Other components should run on 32-bit machines to avoid the memory overhead of large pointers

- **What's the role of Java?**
  - Recent releases (Java6) implement some optimization to eliminate large pointer overhead

  $\rightarrow$ A cluster of 64-bit machines has no downside

Cluster Specification: Network Topology



Cluster Specification: Network Topology

- **Two-level network topology**
  - Switch redundancy is not shown in the figure
- **Typical configuration**
  - 30-40 servers per rack
  - 1 GB switch per rack
  - Core switch or router with 1GB or better
- **Features**
  - Aggregate bandwidth between nodes on the same rack is much larger than for nodes on different racks
  - Rack awareness
    - Hadoop should know the cluster topology
    - Benefits both HDFS (data placement) and MapReduce (locality)

Hadoop Configuration

- **There are a handful of files for controlling the operation of an Hadoop Cluster**
  - See next slide for a summary table

- **Managing the configuration across several machines**
  - All machines of an Hadoop cluster must be in sync!
  - What happens if you dispatch an update and some machines are down?
  - What happens when you add (new) machines to your cluster?
  - What if you need to patch MapReduce?

- **Common practice: use configuration management tools**
  - Chef, Puppet, ...
  - Declarative language to specify configurations
  - Allow also to install software

Hadoop Configuration

Filename	Format	Description
hadoop-env.sh	Bash script	Environment variables that are used in the scripts to run Hadoop.
core-site.xml	Hadoop configuration XML	I/O settings that are common to HDFS and MapReduce.
hdfs-site.xml	Hadoop configuration XML	Namenode, the secondary namenode, and the datanodes.
mapred-site.xml	Hadoop configuration XML	Jobtracker, and the tasktrackers.
masters	Plain text	A list of machines that each run a secondary namenode.
slaves	Plain text	A list of machines that each run a datanode and a tasktracker.

Table: Hadoop Configuration Files

Hadoop Configuration: memory utilization

- **Hadoop uses a lot of memory**
  - Default values, for a typical cluster configuration
    - DataNode: 1 GB
    - TaskTracker: 1 GB
    - Child JVM map task: $2 \times$ 200MB
    - Child JVM reduce task: $2 \times$ 200MB
- **All the moving parts of Hadoop (HDFS and MapReduce) can be individually configured**
  - This is true for cluster configuration but also for job specific configurations
- **Hadoop is fast when using RAM**
  - Generally, MapReduce Jobs are not CPU-bound
  - Avoid I/O on disk as much as you can
  - Minimize network traffic
    - Customize the partitioner
    - Use compression ($\rightarrow$ decompression is in RAM)

Elephants in the cloud!

- **May organization run Hadoop in private clusters**
  - Pros and cons

- **Cloud based Hadoop installations (Amazon biased)**
  - Use Cloudera + Whirr
  - Use Elastic MapReduce

Hadoop on EC2

- **Launch instances of a cluster on demand, paying by hour**
  - CPU, in general bandwidth is used from within a datacenter, hence it's free

- **Apache Whirr project**
  - Launch, terminate, modify a running cluster
  - Requires AWS credentials

- **Example**
  - Launch a cluster `test-hadoop-cluster`, with one master node (`JobTracker` and `NameNode`) and 5 worker nodes (`DataNodes` and `TaskTrackers`)

  → `hadoop-ec2 launch-cluster test-hadoop-cluster 5`
  - See project webpage and Chapter 9, page 290 [11]

AWS Elastic MapReduce

- **Hadoop as a service**
  - Amazon handles everything, which becomes transparent
  - How this is done remains a mistery

- **Focus on What not How**
  - All you need to do is to package a MapReduce Job in a JAR and upload it using a Web Interface
  - Other Jobs are available: python, pig, hive, ...
  - Test your jobs locally!!!

Part Three

Algorithm Design in MapReduce

Preliminaries

Algorithm Design

- **Developing algorithms involve:**
  - Preparing the input data
  - Implement the mapper and the reducer
  - Optionally, design the combiner and the partitioner

- **How to recast existing algorithms in MapReduce?**
  - It is not always obvious how to express algorithms
  - Data structures play an important role
  - Optimization is hard

  $\rightarrow$ The designer needs to “bend” the framework

- **Learn by examples**
  - “Design patterns”
  - Synchronization is perhaps the most tricky aspect

Algorithm Design

- **Aspects that are not under the control of the designer**
  - *Where* a mapper or reducer will run
  - *When* a mapper or reducer begins or finishes
  - *Which* input key-value pairs are processed by a specific mapper
  - *Which* intermediate key-value paris are processed by a specific reducer

- **Aspects that can be controlled**
  - Construct data structures as keys and values
  - Execute user-specified initialization and termination code for mappers and reducers
  - Preserve state across multiple input and intermediate keys in mappers and reducers
  - Control the sort order of intermediate keys, and therefore the order in which a reducer will encounter particular keys
  - Control the partitioning of the key space, and therefore the set of keys that will be encountered by a particular reducer

Algorithm Design

- **MapReduce jobs can be complex**
  - Many algorithms cannot be easily expressed as a single MapReduce job
  - Decompose complex algorithms into a sequence of jobs
    - Requires orchestrating data so that the output of one job becomes the input to the next
  - Iterative algorithms require an external driver to check for convergence

- **Optimizations**
  - Scalability (linear)
  - Resource requirements (storage and bandwidth)

- **Outline**
  - Local Aggregation
  - Pairs and Stripes
  - Order inversion
  - Graph algorithms

Local Aggregation

Local Aggregation

- **In the context of data-intensive distributed processing, the most important aspect of synchronization is the exchange of intermediate results**
  - This involves copying intermediate results from the processes that produced them to those that consume them
  - In general, this involves data transfers over the network
  - In Hadoop, also disk I/O is involved, as intermediate results are written to disk

- **Network and disk latencies are expensive**
  - Reducing the amount of intermediate data translates into algorithmic efficiency

- **Combiners and preserving state across inputs**
  - Reduce the number and size of key-value pairs to be shuffled

Combiners

- **Combiners are a general mechanism to reduce the amount of intermediate data**
  - They could be thought of as “mini-reducers”

- **Example: word count**
  - Combiners aggregate term counts across documents processed by each map task
  - If combiners take advantage of all opportunities for local aggregation we have at most $m \times V$ intermediate key-value pairs
    - ★ m: number of mappers
    - ★ V: number of unique terms in the collection
  - Note: due to Zipfian nature of term distributions, not all mappers will see all terms

EURECOM

Word Counting in MapReduce

class Mapper
    method Map(docid a, doc d)
        for all term $t \in$ doc d **do**
            Emit(term t, count 1)

class Reducer
    method Reduce(term t, counts $[c_1, c_2, \ldots]$)
        $sum \leftarrow 0$
        for all count $c \in$ counts $[c_1, c_2, \ldots]$ **do**
            $sum \leftarrow sum + c$
        Emit(term t, count sum)

In-Mapper Combiners

- **In-Mapper Combiners, a possible improvement**
  - Hadoop does not guarantee combiners to be executed
- **Use an associative array to cumulate intermediate results**
  - The array is used to tally up term counts within a single document
  - The `Emit` method is called only after all `InputRecords` have been processed
- **Example (see next slide)**
  - The code emits a key-value pair for each unique term in the document

In-Mapper Combiners

```
class Mapper
    method Map(docid a, doc d)
        H ← new AssociativeArray
        for all term t ∈ doc d do
            H{t} ← H{t} + 1                ▷ Tally counts for entire document
        for all term t ∈ H do
            Emit(term t, count H{t})
```

In-Mapper Combiners

- **Taking the idea one step further**
  - Exploit implementation details in Hadoop
  - A Java mapper object is created for each map task
  - JVM reuse must be enabled

- **Preserve state within and across calls to the `Map` method**
  - `Initialize` method, used to create a across-map persistent data structure
  - `Close` method, used to emit intermediate key-value pairs only when all map task scheduled on one machine are done

EURECOM

In-Mapper Combiners

```
class Mapper
    method Initialize
        H ← new AssociativeArray
    method Map(docid a, doc d)
        for all term t ∈ doc d do
            H{t} ← H{t} + 1                    ▷ Tally counts across documents
    method Close
        for all term t ∈ H do
            Emit(term t, count H{t})
```

In-Mapper Combiners

- **Summing up: a first "design pattern", *in-mapper combining***
  - Provides control over when local aggregation occurs
  - Design can determine how exactly aggregation is done

- **Efficiency vs. Combiners**
  - There is no additional overhead due to the materialization of key-value pairs
    - Un-necessary object creation and destruction (garbage collection)
    - Serialization, deserialization when memory bounded
  - Mappers still need to emit all key-value pairs, combiners only reduce network traffic

In-Mapper Combiners

- **Precautions**
  - In-mapper combining breaks the functional programming paradigm due to state preservation
  - Preserving state across multiple instances implies that algorithm behavior might depend on execution order
    - Ordering-dependent bugs are difficult to find

- **Scalability bottleneck**
  - The in-mapper combining technique strictly depends on having sufficient memory to store intermediate results
    - And you don't want the OS to deal with swapping
  - Multiple threads compete for the same resources
  - A possible solution: "block" and "flush"
    - Implemented with a simple counter

Further Remarks

- **The extent to which efficiency can be increased with local aggregation depends on the size of the intermediate key space**
  - Opportunities for aggregation araise when multiple values are associated to the same keys
- **Local aggregation also effective to deal with reduce stragglers**
  - Reduce the number of values associated with frequently occuring keys

Algorithmic correctness with local aggregation

- **The use of combiners must be thought carefully**
  - In Hadoop, they are optional: the correctness of the algorithm cannot depend on computation (or even execution) of the combiners
- **In MapReduce, the reducer input key-value type must match the mapper output key-value type**
  - Hence, for combiners, both input and output key-value types must match the output key-value type of the mapper
- **Commutative and Associatvie computations**
  - This is a special case, which worked for word counting
    - ★ There the combiner code is actually the reducer code
  - In general, combiners and reducers are not interchangeable

Algorithmic Correctness: an Example

- **Problem statement**
  - We have a large dataset where input keys are strings and input values are integers
  - We wish to compute the mean of all integers associated with the same key
    - In practice: the dataset can be a log from a website, where the keys are user IDs and values are some measure of activity

- **Next, a baseline approach**
  - We use an identity mapper, which groups and sorts appropriately input key-value paris
  - Reducers keep track of running sum and the number of integers encountered
  - The mean is emitted as the output of the reducer, with the input string as the key

- **Inefficiency problems in the shuffle phase**

Example: basic MapReduce to compute the mean of values

```
class Mapper
    method Map(string t, integer r)
        Emit(string t, integer r)

class Reducer
    method Reduce(string t, integers [r1, r2, ...])
        sum ← 0
        cnt ← 0
        for all integer r ∈ integers [r1, r2, ...] do
            sum ← sum + r
            cnt ← cnt + 1
        r_avg ← sum/cnt
        Emit(string t, integer r_avg)
```

Algorithmic Correctness: an Example

- **Note: operations are not distributive**
  - `Mean`(1,2,3,4,5) $\neq$ `Mean`(`Mean`(1,2), `Mean`(3,4,5))
  - Hence: a combiner cannot output partial means and hope that the reducer will compute the correct final mean
- **Next, a failed attempt at solving the problem**
  - The combiner partially aggregates results by separating the components to arrive at the mean
  - The sum and the count of elements are packaged into a pair
  - Using the same input string, the combiner emits the pair

Example: Wrong use of combiners

```
class Mapper
    method Map(string t, integer r)
        Emit(string t, integer r)
```

```
class Combiner
    method Combine(string t, integers [r1, r2, ...])
        sum ← 0
        cnt ← 0
        for all integer r ∈ integers [r1, r2, ...] do
            sum ← sum + r
            cnt ← cnt + 1
        Emit(string t, pair (sum, cnt))                ▷ Separate sum and count
```

```
class Reducer
    method Reduce(string t, pairs [(s1, c1), (s2, c2) ...])
        sum ← 0
        cnt ← 0
        for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
            sum ← sum + s
            cnt ← cnt + c
        r_avg ← sum/cnt
        Emit(string t, integer r_avg)
```

Algorithmic Correctness: an Example

- **What's wrong with the previous approach?**
  - Trivially, the input/output keys are not correct
  - Remember that combiners are optimizations, the algorithm should work even when "removing" them

- **Executing the code omitting the combiner phase**
  - The output value type of the mapper is integer
  - The reducer expects to receive a list of integers
  - Instead, we make it expect a list of pairs

- **Next, a correct implementation of the combiner**
  - Note: the reducer is similar to the combiner!
  - Exercise: verify the correctness

Example: Correct use of combiners

```
class MAPPER
    method MAP(string t, integer r)
        EMIT(string t, pair (r, 1))

class COMBINER
    method COMBINE(string t, pairs [(s1, c1), (s2, c2) ...])
        sum ← 0
        cnt ← 0
        for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
            sum ← sum + s
            cnt ← cnt + c
        EMIT(string t, pair (sum, cnt))

class REDUCER
    method REDUCE(string t, pairs [(s1, c1), (s2, c2) ...])
        sum ← 0
        cnt ← 0
        for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
            sum ← sum + s
            cnt ← cnt + c
        r_avg ← sum/cnt
        EMIT(string t, integer r_avg)
```

Algorithmic Correctness: an Example

- **Using in-mapper combining**
  - Inside the mapper, the partial sums and counts are held in memory (across inputs)
  - Intermediate values are emitted only after the entire input split is processed
  - Similarly to before, the output value is a pair

```
class Mapper
    method Initialize
        S ← new AssociativeArray
        C ← new AssociativeArray
    method Map(string t, integer r)
        S{t} ← S{t} + r
        C{t} ← C{t} + 1
    method Close
        for all term t ∈ S do
            Emit(term t, pair (S{t}, C{t}))
```

EURECOM

Pairs and Stripes

Pairs and Stripes

- **A common approach in MapReduce: build complex keys**
  - Data necessary for a computation are naturally brought together by the framework
- **Two basic techniques:**
  - *Pairs*: similar to the example on the average
  - *Stripes*: uses in-mapper memory data structures
- **Next, we focus on a particular problem that benefits from these two methods**

Problem statement

- **The problem: building word co-occurrence matrices for large corpora**
  - The co-occurrence matrix of a corpus is a square $n \times n$ matrix
  - n is the number of unique words (*i.e.*, the vocabulary size)
  - A cell m_{ij} contains the number of times the word w_i co-occurs with word w_j within a specific context
  - Context: a sentence, a paragraph a document or a window of m words
  - NOTE: the matrix may be symmetric in some cases

- **Motivation**
  - This problem is a basic building block for more complex operations
  - Estimating the distribution of discrete joint events from a large number of observations
  - Similar problem in other domains:
    - Customers who buy *this* tend to also buy *that*

Observations

- **Space requirements**
  - Clearly, the space requirement is $O(n^2)$, where n is the size of the vocabulary
  - For real-world (English) corpora n can be hundres of thousands of words, or even billion of worlds
- **So what's the problem?**
  - If the matrix can fit in the memory of a single machine, then just use whatever naive implementation
  - Instead, if the matrix is bigger than the available memory, then paging would kick in, and any naive implementation would break
- **Compression**
  - Such techniques can help in solving the problem on a single machine
  - However, there are scalability problems

Word co-occurrence: the Pairs approach

- **Input to the problem**
  - Key-value pairs in the form of a `docid` and a `doc`
- **The mapper:**
  - Processes each input document
  - Emits key-value pairs with:
    - Each co-occurring word pair as the key
    - The integer one (the count) as the value
  - This is done with two nested loops:
    - The outer loop iterates over all words
    - The inner loop iterates over all neighbors
- **The reducer:**
  - Receives pairs relative to co-occurring words
    - This requires modifing the partitioner
  - Computes an absolute count of the joint event
  - Emits the pair and the count as the final key-value output
    - Basically reducers emit the cells of the matrix

Word co-occurrence: the Pairs approach

```
class Mapper
    method Map(docid a, doc d)
        for all term w ∈ doc d do
            for all term u ∈ Neighbors(w) do
                Emit(pair (w, u), count 1)        ▷ Emit count for each co-occurrence

class Reducer
    method Reduce(pair p, counts [c1, c2, ...])
        s ← 0
        for all count c ∈ counts [c1, c2, ...] do
            s ← s + c                              ▷ Sum co-occurrence counts
        Emit(pair p, count s)
```

Word co-occurrence: the Stripes approach

- **Input to the problem**
  - Key-value pairs in the form of a `docid` and a `doc`
- **The mapper:**
  - Same two nested loops structure as before
  - Co-occurrence information is first stored in an associative array
  - Emit key-value pairs with words as keys and the corresponding arrays as values
- **The reducer:**
  - Receives all associative arrays related to the same word
  - Performs an element-wise sum of all associative arrays with the same key
  - Emits key-value output in the form of word, associative array
    - Basically, reducers emit rows of the co-occurrence matrix

Word co-occurrence: the Stripes approach

class MAPPER
    method MAP(docid a, doc d)
        for all term $w \in$ doc d **do**
            $H \leftarrow$ new ASSOCIATIVEARRAY
            for all term $u \in$ NEIGHBORS(w) **do**
                $H\{u\} \leftarrow H\{u\} + 1$ ▷ Tally words co-occurring with w
            EMIT(Term w, Stripe H)

class REDUCER
    method REDUCE(term w, stripes $[H_1, H_2, H_3, \ldots]$)
        $H_f \leftarrow$ new ASSOCIATIVEARRAY
        for all stripe $H \in$ stripes $[H_1, H_2, H_3, \ldots]$ **do**
            SUM(H_f, H) ▷ Element-wise sum
        EMIT(term w, stripe H_f)

Pairs and Stripes, a comparison

- **The pairs approach**
  - Generates a large number of key-value pairs (also intermediate)
  - The benefit from combiners is limited, as it is less likely for a mapper to process multiple occurrences of a word
  - Does not suffer from memory paging problems

- **The pairs approach**
  - More compact
  - Generates fewer and shorted intermediate keys
    - The framework has less sorting to do
  - The values are more complex and have serialization/deserialization overhead
  - Greately benefits from combiners, as the key space is the vocabulary
  - Suffers from memory paging problems, if not properly engineered

Order Inversion

Computing relative frequenceies

- **“Relative” Co-occurrence matrix construction**
  - Similar problem as before, same matrix
  - Instead of absolute counts, we take into consideration the fact that some words appear more frequently than others
    - ★ Word w_i may co-occur frequently with word w_j simply because one of the two is very common
  - We need to convert absolute counts to relative frequencies $f(w_j|w_i)$
    - ★ What proportion of the time does w_j appear in the context of w_i?

- **Formally, we compute:**

$$f(w_j|w_i) = \frac{N(w_i, w_j)}{\sum_{w'} N(w_i, w')}$$

  - $N(\cdot, \cdot)$ is the number of times a co-occurring word pair is observed
  - The denominator is called the marginal

Computing relative frequenceies

- **The stripes approach**
  - In the reducer, the counts of all words that co-occur with the conditioning variable (w_i) are available in the associative array
  - Hence, the sum of all those counts gives the marginal
  - Then we divide the the joint counts by the marginal and we're done

- **The pairs approach**
  - The reducer receives the pair (w_i, w_j) and the count
  - From this information alone it is not possible to compute $f(w_j|w_i)$
  - Fortunately, as for the mapper, also the reducer can preserve state across multiple keys
    - ★ We can buffer in memory all the words that co-occur with w_i and their counts
    - ★ This is basically building the associative array in the stripes method

Computing relative frequenceies: a basic approach

- **We must define the sort order of the pair**
  - In this way, the keys are first sorted by the left word, and then by the right word (in the pair)
  - Hence, we can detect if all pairs associated with the word we are conditioning on (w_i) have been seen
  - At this point, we can use the in-memory buffer, compute the relative frequencies and emit

- **We must define an appropriate partitioner**
  - The default partitioner is based on the hash value of the intermediate key, modulo the number of reducers
  - For a complex key, the raw byte representation is used to compute the hash value
    - ★ Hence, there is no guarantee that the pair (dog, aardvark) and (dog,zebra) are sent to the same reducer
  - What we want is that all pairs with the same left word are sent to the same reducer

Computing relative frequenceies: order inversion

- **The key is to properly sequence data presented to reducers**
  - If it were possible to compute the marginal in the reducer before processing the join counts, the reducer could simply divide the joint counts received from mappers by the marginal
  - The notion of "before" and "after" can be captured in the ordering of key-value pairs
  - The programmer can define the sort order of keys so that data needed earlier is presented to the reducer before data that is needed later

Computing relative frequenceies: order inversion

- **Recall that mappers emit pairs of co-occurring words as keys**

- **The mapper:**
  - additionally emits a "special" key of the form $(w_i, *)$
  - The value associated to the special key is one, that represtns the contribution of the word pair to the marginal
  - Using combiners, these partial marginal counts will be aggregated before being sent to the reducers

- **The reducer:**
  - We must make sure that the special key-value pairs are processed before any other key-value pairs where the left word is w_i
  - We also need to modify the partitioner as before, *i.e.*, it would take into account only the first word

Computing relative frequenceies: order inversion

- **Memory requirements:**
  - Minimal, because only the marginal (an integer) needs to be stored
  - No buffering of individual co-occurring word
  - No scalability bottleneck

- **Key ingredients for order inversion**
  - Emit a special key-value pair to capture the margianl
  - Control the sort order of the intermediate key, so that the special key-value pair is processed first
  - Define a custom partitioner for routing intermediate key-value pairs
  - Preserve state across multiple keys in the reducer

Graph Algorithms

Preliminaries and Data Structures

Motivations

- **Examples of graph problems**
  - Graph search
  - Graph clustering
  - Minimum spanning trees
  - Matching problems
  - Flow problems
  - Element analysis: node and edge centralities
- **The problem: big graphs**
- **Why MapReduce?**
  - Algorithms for the above problems on a single machine are not scalable
  - Recently, Google designed a new system, Pregel, for large-scale (incremental) graph processing
  - Even more recently, [7] indicate a fundamentally new design pattern to analyze graphs in MapReduce

Graph Representations

- **Basic data structures**
  - Adjacency matrix
  - Adjacency list
- **Are graphs sparse or dense?**
  - Determines which data-structure to use
    - Adjacency matrix: operations on incoming links are easy (column scan)
    - Adjacency list: operations on outgoing links are easy
    - The shuffle and sort phase can help, by grouping edges by their destination reducer
  - [8] dispelled the notion of sparseness of real-world graphs

Parallel Breadth-First-Search

Parallel Breadth-First Search

- **Single-source shortest path**
  - Dijkstra algorithm using a global priority queue
    - Maintains a globally sorted list of nodes by current distance
  - How to solve this problem in parallel?
    - "Brute-force" approach: breadth-first search

- **Parallel BFS: intuition**
  - Flooding
  - Iterative algorithm in MapReduce
  - Shoehorn message passing style algorithms

Parallel Breadth-First Search

```
class Mapper
    method Map(nid n, node N)
        d ← N.Distance
        Emit(nid n, N)                                  ▷ Pass along graph structure
        for all nodeid m ∈ N.AdjacencyList do
            Emit(nid m, d + 1)                          ▷ Emit distances to reachable nodes

class Reducer
    method Reduce(nid m, [d1, d2, . . .])
        d_min ← ∞
        M ← ∅
        for all d ∈ counts [d1, d2, . . .] do
            if IsNode(d) then
                M ← d                                   ▷ Recover graph structure
            else if d < d_min then                      ▷ Look for shorter distance
                d_min ← d
        M.Distance ← d_min                              ▷ Update shortest distance
        Emit(nid m, node M)
```

Parallel Breadth-First Search

- **Assumptions**
  - Connected, directed graph
  - Data structure: adjacency list
  - Distance to each node is stored alongside the adjacency list of that node

- **The pseudo-code**
  - We use *n* to denote the node id (an integer)
  - We use *N* to denote the node adjacency list and current distance
  - The algorithm works by mapping over all nodes
  - Mappers emit a key-value pair for each neighbor on the node's adjacency list
    - The key: node id of the neighbor
    - The value: the current distace to the node plus one
    - If we can reach node *n* with a distance *d*, then we must be able to reach all the nodes connected ot *n* with distance $d + 1$

Parallel Breadth-First Search

- **The pseudo-code (continued)**
  - After shuffle and sort, reducers receive keys corresponding to the destination node ids and distances corresponding to all paths leading to that node
  - The reducer selects the shortest of these distances and update the distance in the node data structure

- **Passing the graph along**
  - The mapper: emits the node adjacency list, with the node id as the key
  - The reducer: must distinguish between the node data structure and the distance values

Parallel Breadth-First Search

- **MapReduce iterations**
  - The first time we run the algorithm, we “discover” all nodes connected to the source
  - The second iteration, we discover all nodes connected to those
  - → Each iteration expands the “search frontier” by one hop
  - How many iterations before convergence?

- **This approach is suitable for small-world graphs**
  - The diameter of the network is small
  - See [7] for advanced topics on the subject

Parallel Breadth-First Search

- **Checking the termination of the algorithm**
  - Requires a “driver” program which submits a job, check termination condition and eventually iterates
  - In practice:
    - Hadoop counters
    - Side-data to be passed to the job configuration

- **Extensions**
  - Storing the actual shortest-path
  - Weighted edges (as opposed to unit distance)

The story so far

- **The graph structure is stored in an adjacency lists**
  - This data structure can be augmented with additional information
- **The MapReduce framework**
  - Maps over the node data structures involving only the node's internal state and it's local graph structure
  - Map results are "passed" along outgoing edges
  - The graph itself is passed from the mapper to the reducer
    - This is a very costly operation for large graphs!
  - Reducers aggregate over "same destination" nodes
- **Graph algorithms are generally iterative**
  - Require a driver program to check for termination

PageRank

Introduction

- **What is PageRank**
  - It's a measure of the relevance of a Web page, based on the structure of the hyperlink graph
  - Based on the concept of random Web surfer

- **Formally we have:**

$$P(n) = \alpha\left(\frac{1}{|G|}\right) + (1-\alpha)\sum_{m \in L(n)} \frac{P(m)}{C(m)}$$

  - $|G|$ is the number of nodes in the graph
  - α is a random jump factor
  - $L(n)$ is the set of out-going links from page n
  - $C(m)$ is the out-degree of node m

PageRank in Details

- **PageRank is defined recursively, hence we need an interative algorithm**
  - A node receives "contributions" from all pages that link to it

- **Consider the set of nodes $L(n)$**
  - A random surfer at m arrives at n with probability $1/C(m)$
  - Since the PageRank value of m is the probability that the random surfer is at m, the probability of arriving at n from m is $P(m)/C(m)$

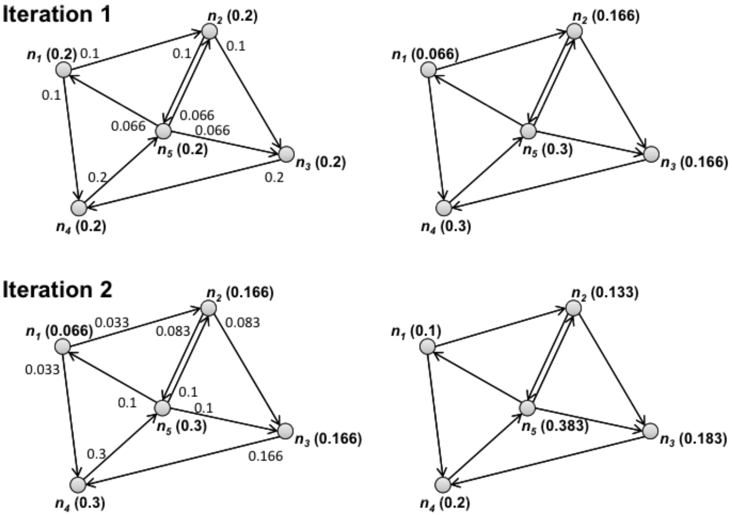
- **To compute the PageRank of n we need:**
  - Sum the contributions from all pages that link to n
  - Take into account the random jump, which is uniform over all nodes in the graph

PageRank in MapReduce

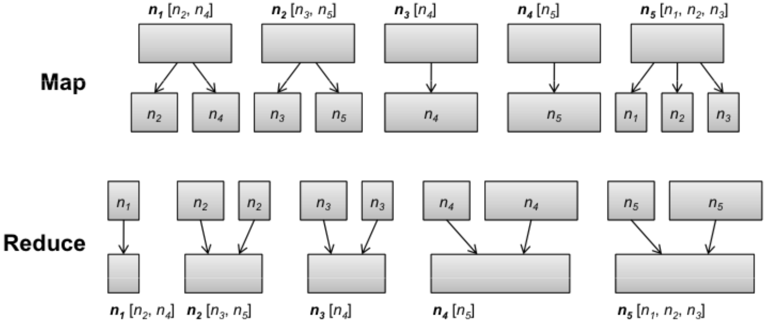
```
class Mapper
    method Map(nid n, node N)
        p ← N.PageRank/|N.AdjacencyList|
        Emit(nid n, N)                              ▷ Pass along graph structure
        for all nodeid m ∈ N.AdjacencyList do
            Emit(nid m, p)                          ▷ Pass PageRank mass to neighbors

class Reducer
    method Reduce(nid m, [p1, p2, ...])
        M ← ∅
        for all p ∈ counts [p1, p2, ...] do
            if IsNode(p) then
                M ← p                               ▷ Recover graph structure
            else
                s ← s + p                           ▷ Sum incoming PageRank contributions
        M.PageRank ← s
        Emit(nid m, node M)
```

PageRank in MapReduce



PageRank in MapReduce



PageRank in MapReduce

- **Sketch of the MapReduce algorithm**
  - The algorithm maps over the nodes
  - Foreach node computes the PageRank mass the needs to be distributed to neighbors
  - Each fraction of the PageRank mass is emitted as the value, keyed by the node ids of the neighbors
  - In the shuffle and sort, values are grouped by node id
    - Also, we pass the graph structure from mappers to reducers (for subsequent iterations to take place over the updated graph)
  - The reducer updates the value of the PageRank of every single node

PageRank in MapReduce

- **Implementation details**
  - Loss of PageRank mass for sink nodes
  - Auxiliary state information
  - One iteration of the algorith
    - Two MapReduce jobs: one to distribute the PageRank mass, the other for dangling nodes and random jumps
  - Checking for convergence
    - Requires a driver program
    - When updates of PageRank are "stable" the algorithm stops

- **Further reading on convergence and attacks**
  - Convergenge: [9, 4]
  - Attacks: Adversarial Information Retrieval Workshop [1]

References I

[1] Adversarial information retrieval workshop.

[2] Michele Banko and Eric Brill.
Scaling to very very large corpora for natural language disambiguation.
In *Proc. of the 39th Annual Meeting of the Association for Computational Linguistic (ACL)*, 2001.

[3] Luiz Andre Barroso and Urs Holzle.
The datacebter as a computer: An introduction to the design of warehouse-scale machines.
Morgan & Claypool Publishers, 2009.

[4] Monica Bianchini, Marco Gori, and Franco Scarselli.
Inside pagerank.
In *ACM Transactions on Internet Technology*, 2005.

References II

[5] James Hamilton.
Cooperative expendable micro-slice servers (cems): Low cost, low power servers for internet-scale services.
In *Proc. of the 4th Biennal Conference on Innovative Data Systems Research (CIDR)*, 2009.

[6] Tony Hey, Stewart Tansley, and Kristin Tolle.
The fourth paradigm: Data-intensive scientific discovery.
Microsoft Research, 2009.

[7] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii.
Filtering: a method for solving graph problems in mapreduce.
In *Proc. of SPAA*, 2011.

References III

[8] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos.
Graphs over time: Densification laws, shrinking diamters and possible explanations.
In *Proc. of SIGKDD*, 2005.

[9] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd.
The pagerank citation ranking: Bringin order to the web.
In *Stanford Digital Library Working Paper*, 1999.

[10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler.
The hadoop distributed file system.
In *Proc. of the 26th IEEE Symposium on Massive Storage Systems and Technologies (MSST)*. IEEE, 2010.

References IV

[11] Tom White.
Hadoop, The Definitive Guide.
O'Reilly, Yahoo, 2010.