Scalable Algorithm Design

The MapReduce Programming Model

Pietro Michiardi

Eurecom

- Jimmy Lin and Chris Dyer, "Data-Intensive Text Processing with MapReduce," Morgan & Claypool Publishers, 2010.
`http://lintool.github.io/MapReduceAlgorithms/`
- Tom White, "Hadoop, The Definitive Guide," O'Reilly / Yahoo Press, 2012
- Anand Rajaraman, Jeffrey D. Ullman, Jure Leskovec, "Mining of Massive Datasets", Cambridge University Press, 2013

Key Principles

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 [1, 3]
- **Some numbers ($\sim$ 2012):**
  - Data processed by Google every day: 100+ PB
  - Data processed by Facebook every day: 10+ PB

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**:
  - Sharing: manage a common/global state
  - Shared nothing: independent entities, no common state
- **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(s) failures [Several!]
  - Cascading effect

Implications of Failures

- **Failures are part of everyday life**
  - Mostly due to the scale and shared environment
- **Sources of Failures**
  - Hardware / Software
  - Electrical, Cooling, ...
  - 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 collocated
  - → **Data Locality Principle**
- **Distributed filesystems are necessary**

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 data
- **Typically, data is collected "elsewhere" and copied to the distributed filesystem**
  - E.g.: Apache Flume, Hadoop Sqoop, $\cdots$
- **Data-intensive applications**
  - Read and process the whole Web (e.g. PageRank)
  - Read and process the whole Social Graph (e.g. LinkPrediction, a.k.a. "friend suggest")
  - Log analysis (e.g. Network traces, Smart-meter data, $\cdots$)

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
- **BUT: In-depth knowledge of the framework is key**
  - Custom data reader/writer
  - Custom data partitioning
  - Memory utilization
- **Auxiliary components**
  - Hadoop Pig
  - Hadoop Hive
  - Cascading/Scalding
  - ... and many many more!

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
- **Embarrassingly parallel problems**
  - Simple definition: independent (shared nothing) computations on fragments of the dataset
  - How to to decide if a problem is embarrassingly parallel or not?
- **MapReduce is a first attempt, not the final answer**

The Programming Model

Functional Programming Roots

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

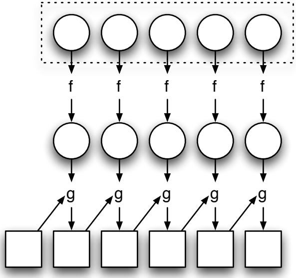


Figure: Illustration of *map* and *fold*.

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 (an accumulator)
  - 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** elements of the list, also the fold phase can proceed in parallel
- **Associative and commutative operations**
  - Allow performance gains through local aggregation and reordering

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:**
  - Grouping intermediate results happens 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 and strict
- **There are several important problems that can be adapted to MapReduce**
  - 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

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 involves**:
  - Imposing the key-value structure on arbitrary datasets[2]
    - 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

[2] There's more about it: here we only look at the input to the map function.

A Generic MapReduce Algorithm

- **The programmer defines a mapper and a reducer as follows**[3]:
  - map: $(k_1, v_1) \rightarrow [(k_2, v_2)]$
  - reduce: $(k_2, [v_2]) \rightarrow [(k_3, v_3)]$

- **In words**:
  - A dataset stored on an underlying **distributed** filesystem, which is split in a number of **blocks** 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

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

Where the magic happens

- **Implicit between the map and reduce phases is a parallel "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[4]
- **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

[4] Think of **iterative algorithms**.

A Simplified view of MapReduce

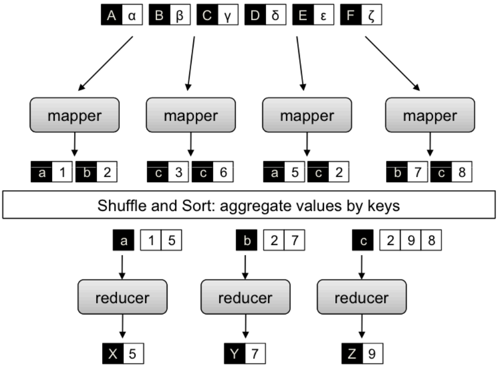


Figure: Mappers are applied to all input key-value pairs, to generate an arbitrary number of intermediate pairs. Reducers are applied to all intermediate values associated with the same intermediate key. Between the map and reduce phase lies a barrier that involves a large distributed sort and group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

Basic Design Patterns

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
  - → The designer needs to "bend" the framework

- **Learn by examples**
  - "Design patterns"
  - "Shuffle" 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 pairs 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 algorithms 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

- **Basic design patterns**[5]
  - Local Aggregation
  - Pairs and Stripes
  - Order inversion

[5] You will see them in action during the DAY2 laboratory session.

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

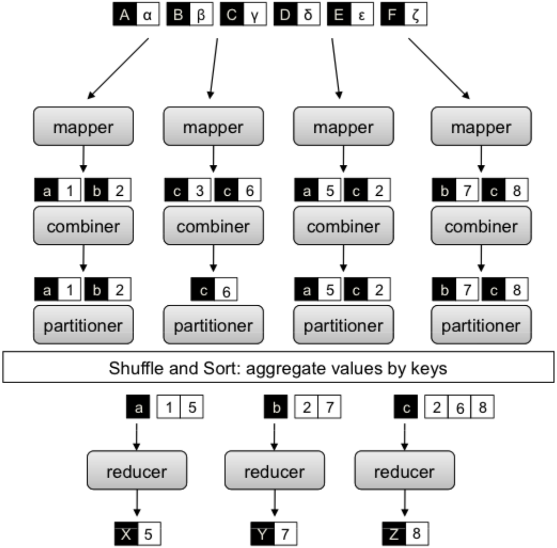
- **Combiners and preserving state across inputs**
  - Reduce the number and size of key-value pairs to be shuffled

Combiners

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

- **Back to our running 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

Combiners: an illustration



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-Memory 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-Memory Combiners

- **Taking the idea one step further**
  - Exploit implementation details in Hadoop[6]
  - 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

[6] Forward reference! We'll see more tomorrow.

In-Memory 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-Memory Combiners

- **Summing up: a first "design pattern", *in-memory combining***
  - Provides control over when local aggregation occurs
  - Designer 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-Memory Combiners

- **Precautions**
  - In-memory 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-memory 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 arise 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 Associative 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 pairs
  - 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 way 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

- **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 [r_1, r_2, ...])
        sum ← 0
        cnt ← 0
        for all integer r ∈ integers [r_1, r_2, ...] do
            sum ← sum + r
            cnt ← cnt + 1
        EMIT(string t, pair (sum, cnt))                ▷ Separate sum and count

class REDUCER
    method REDUCE(string t, pairs [(s_1, c_1), (s_2, c_2) ...])
        sum ← 0
        cnt ← 0
        for all pair (s, c) ∈ pairs [(s_1, c_1), (s_2, c_2) ...] 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)
```

Advanced technique

- **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
    $S \leftarrow$ new ASSOCIATIVEARRAY
    $C \leftarrow$ new ASSOCIATIVEARRAY
  method MAP(string t, integer r)
    $S\{t\} \leftarrow S\{t\} + r$
    $C\{t\} \leftarrow C\{t\} + 1$
  method CLOSE
    for all term $t \in S$ **do**
      EMIT(term t, pair $(S\{t\}, C\{t\})$)

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 hundreds of thousands of words, or even billions of worlds in some specific cases
- **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 related to co-occurring words
    - This requires **modifying 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 output 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 ∈ doc d do
            H ← new AssociativeArray
            for all term u ∈ Neighbors(w) do
                H{u} ← 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, ...])
        H_f ← new AssociativeArray
        for all stripe H ∈ stripes [H_1, H_2, H_3, ...] 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
    - In particular, intermediate ones, that fly over the network
  - 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 stripes 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
  - Greatly benefits from combiners, as the key space is the vocabulary
  - Suffers from memory paging problems, if not properly engineered

Computing relative frequencies

- **"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 frequencies

- **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 frequencies: 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 frequencies: 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 joint 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 frequencies: 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 represents 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 frequencies: 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 marginal
  - 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 [Optional]

Motivations

- **Examples of graph problems**
  - Clustering
  - Matching 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, [4] indicate a fundamentally new design pattern to analyze graphs in MapReduce
  - New trend: graph databases, graph processing systems[7]

[7] If you're interested, we'll discuss this off-line.

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
  - [5] dispelled the notion of sparseness of real-world graphs

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, [d_1, d_2, . . .])
        d_min ← ∞
        M ← ∅
        for all d ∈ counts [d_1, d_2, . . .] 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 distance 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 to 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
  - $\rightarrow$ 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 [4] 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

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

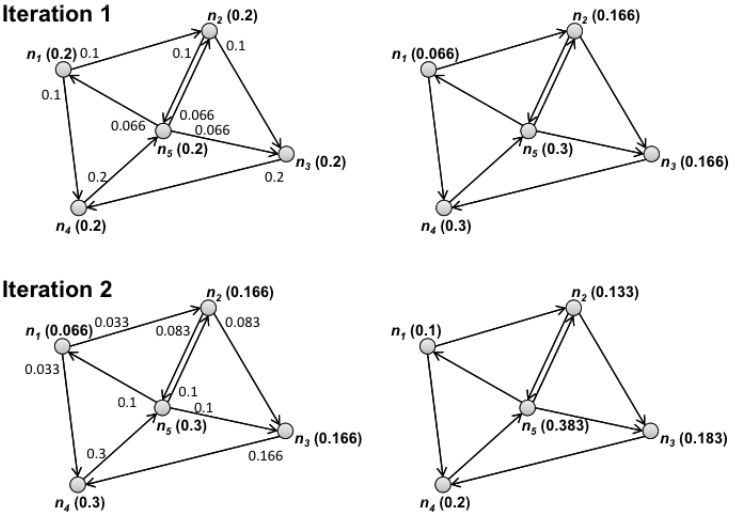
- **PageRank is defined recursively, hence we need an iterative algorithm**
  - A node receives “contributions” from all pages that link to it
- **Consider the set of nodes $L(n)$**
  - A random surfer at m arrives at n with probability $1/C(m)$
  - Since the PageRank value of m is the probability that 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

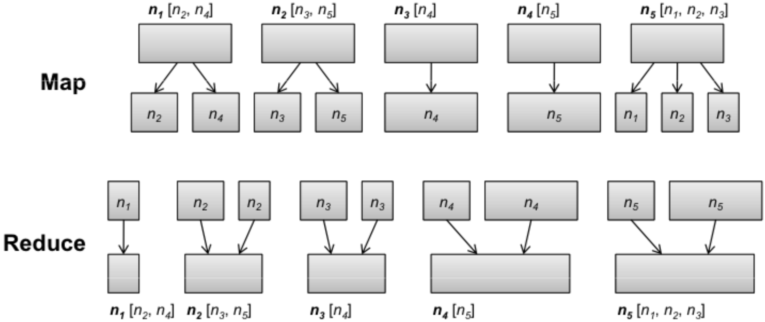
```
class Mapper
    method Map(nid n, node N)
        p ← N.PageRank/|N.AdjacencyList|
        Emit(nid n, N)                                  ▷ Pass along graph structure
        for all nodeid m ∈ N.AdjacencyList do
            Emit(nid m, p)                              ▷ Pass PageRank mass to neighbors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

```
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
  - For each 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 algorithm
    - 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**
  - Convergence: [6, 2]

References

References I

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

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

[3] 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.

References II

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

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

[6] 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.