Map reduce some Principles and patterns

Genoveva Vargas Solar
French Council of scientific research, LIG-LAFMIA, France
Genoveva.Vargas@imag.fr
http://www.vargas-solar.com/teaching

Map-reduce

- **Programming model** for expressing distributed computations on massive amounts of data
- **Execution framework** for large-scale data processing on clusters of commodity servers
- Market: any organization built around gathering, analyzing, monitoring, filtering, searching, or organizing content must tackle large-data problems
  - data- intensive processing is beyond the capability of any individual machine and requires clusters
  - large-data problems are fundamentally about ***organizing computations on dozens, hundreds, or even thousands of machines***

« Data represent the rising tide that lifts all boats—
more data lead to better algorithms and systems for
solving real-world problems »

Data processing

- Process the data to produce other data: analysis tool, business intelligence tool, ...
- This means
- • Handle large volumes of data
- • Manage thousands of processors
- • Parallelize and distribute treatments
  - Scheduling I/O
  - Managing Fault Tolerance
  - Monitor /Control processes

Map-Reduce provides all this easy!

Motivation

- The only feasible approach to tackling large-data problems is to ***divide and conquer***
  - To the extent that the sub-problems are independent, they can be tackled in parallel by different worker (threads in a processor core, cores in a multi-core processor, multiple processors in a machine, or many machines in a cluster)
  - Intermediate results from each individual worker are then combined to yield the final output

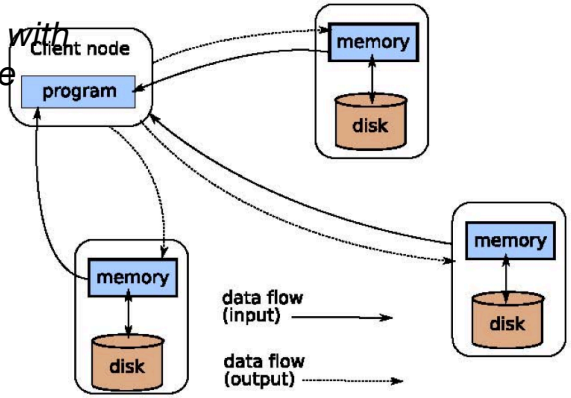
- Aspects to consider
  - How do we decompose the problem so that the smaller tasks can be executed in parallel?
  - How do we assign tasks to workers distributed across a potentially large number of machines? (some workers are better suited to running some tasks than others, e.g., due to available resources, locality constraints, etc.)
  - How do we ensure that the workers get the data they need?
  - How do we coordinate synchronization among the different workers?
  - How do we share partial results from one worker that is needed by another?
  - How do we accomplish all of the above in the face of software errors and hardware faults?

Motivation

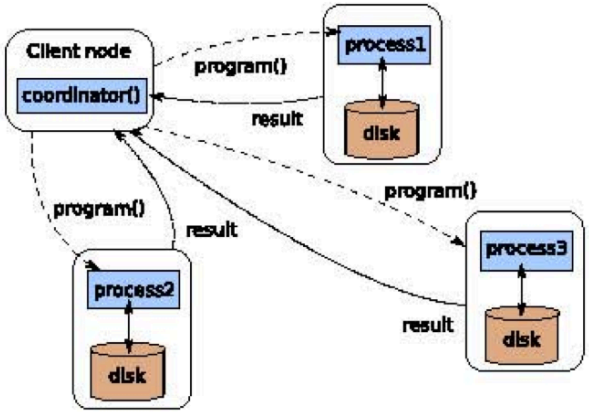
- OpenMP for shared memory parallelism or libraries implementing the Message Passing Interface (MPI) for cluster-level parallelism provide logical abstractions that hide details of operating system synchronization and communications primitives
  - → developers keep track of how resources are made available to workers
- Map-Reduce provides an abstraction hiding many system-level details from the programmer
  - → developers focus on ***what*** computations need to be performed, as opposed to ***how*** those computations are actually carried out or ***how*** to get the data to the processes
- **Yet**, organizing and coordinating large amounts of computation is only part of the challenge
  - Large-data processing requires bringing data and code together for computation to occur —no small feat for datasets that are terabytes and perhaps petabytes in size!

Approach

Centralized computing with distributed data storage



Run the program at the Client, get data from the distributed system
Downsides: important data flows, no use of the cluster computing resources



"push the program near the data"

- Instead of moving large amounts of data around, it is far more efficient, if possible, to move the code to the data
- The complex task of managing storage in such a processing environment is typically handled by a distributed file system that sits underneath MapReduce

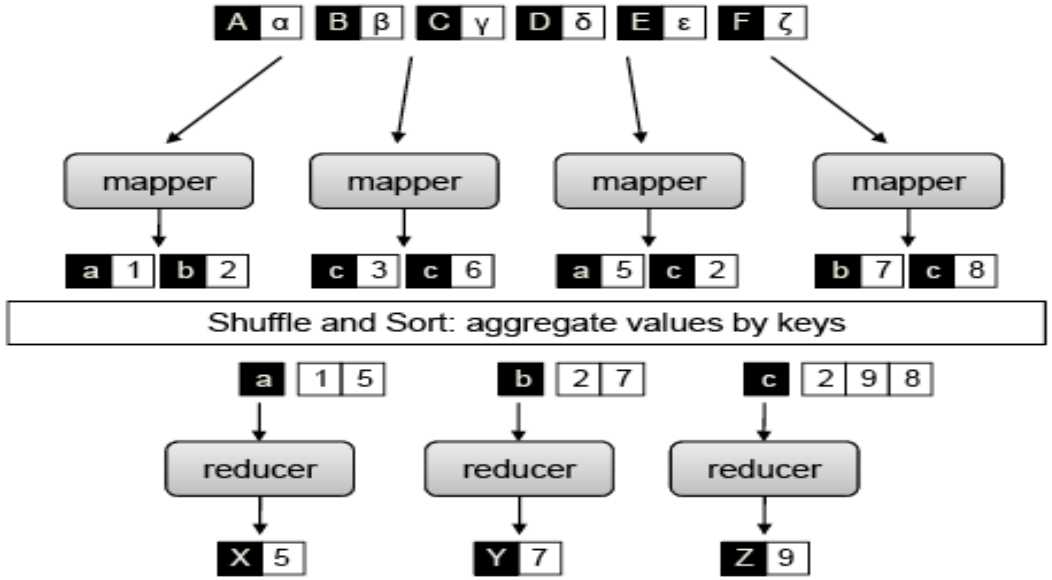
map-reduce principle

$$\text{map: } (k_1, v_1) \longrightarrow [(k_2, v_2)]$$
$$\text{reduce: } (k_2, [v_2]) \longrightarrow [(k_3, v_3)]$$

- **Stage 1**: Apply a user-specified computation over all input records in a dataset.
  - These operations occur in parallel and yield intermediate output (**key-value** pairs)
- **Stage 2**: Aggregate intermediate output by another user-specified computation
  - Recursively applies a function on every pair of the list

Map reduce example

Count the number of occurrences of every word in a text collection



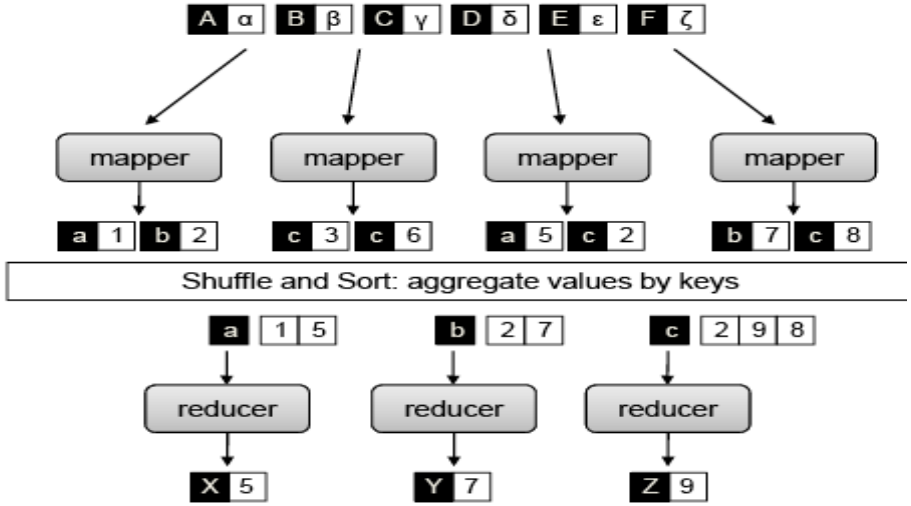
Key-value pairs

- Basic data structure in MapReduce, keys and values may be
  - primitive such as integers, floating point values, strings, and raw bytes
  - arbitrarily complex structures (lists, tuples, associative arrays, etc.)
- Part of the design of MapReduce algorithms involves imposing the key-value structure on arbitrary datasets
  - For a collection of web pages, keys may be URLs and values may be the actual HTML content.
  - For a graph, keys may represent node ids and values may contain the adjacency lists of those nodes

How does a map reduce job work?

- MapReduce job starts as data stored on the underlying distributed file system
- The mapper
  - is applied to every input key-value pair (split across an arbitrary number of files)
  - to generate an arbitrary number of intermediate key-value pairs.
- The reducer
  - is applied to all values associated with the same intermediate key
  - to generate output key-value pairs
- Intermediate data arrive at each reducer in order, sorted by the key: “group by” operation on intermediate keys
- Output key-value pairs from each reducer are written persistently back onto the distributed file system (intermediate key-value pairs are transient and not preserved)

Map reduce example




```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term t ∈ doc d do
4:             Emit(term t, count 1)

1: class Reducer
2:     method Reduce(term t, counts [c1, c2, ...])
3:         sum ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             sum ← sum + c
6:         Emit(term t, count sum)
```

Map reduce example

- Input key-values pairs take the form of (docid, doc) pairs stored on the distributed file system,
  - the former is a unique identifier for the document
  - the latter is the text of the document itself
- The **mapper** takes an input key-value pair, tokenizes the document, and emits an intermediate key-value pair for every word:
  - the word itself serves as the key, and the integer one serves as the value (denoting that we've seen the word once)
  - the MapReduce execution framework guarantees that all values associated with the same key are brought together in the reducer
- The reducer sums up all counts (ones) associated with each word
  - emits final key- value pairs with the word as the key, and the count as the value.
  - output is written to the distributed file system, one file per reducer

Implementation issues: mapper and reducer objects

- Mappers and reducers are objects that implement the `Map` and `Reduce` methods, respectively
- A `mapper object` is initialized for each map task
  - Associated with a particular sequence of key-value pairs called an ***input split***
  - `Map` method is called on each key-value pair by the execution framework
- A `reducer` object is initialized for each reduce task
  - `Reduce` method is called once per intermediate key
- MapReduce programs can contain no reducers, in which case mapper output is directly written to disk

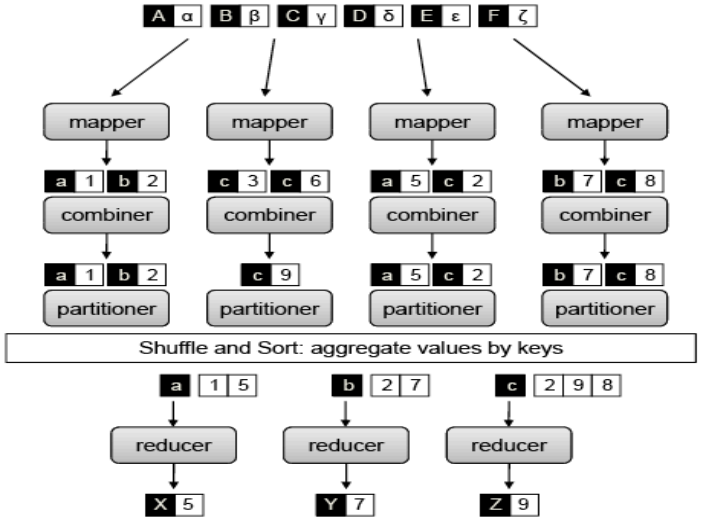
Implementation issues: mapper and reducer objects

- Mappers can emit an arbitrary number of intermediate key-value pairs, and they need not be of the same type as the input key-value pairs.
- Reducers can emit an arbitrary number of final key-value pairs, and they can differ in type from the intermediate key-value pairs.
- Mappers and reducers can have side effects
  - this is a powerful and useful feature
  - for example, preserving state across multiple inputs is central to the design of many MapReduce algorithms
- Since many mappers and reducers are run in parallel, and the distributed file system is a shared global resource, special care must be taken to ensure that such operations avoid synchronization conflicts

Existing solutions

- Google's MapReduce implementation BigTable
  - Input and output stored in a sparse, distributed, persistent multidimensional sorted map
- HBase is an open-source BigTable clone and has similar capabilities.
- Hadoop has been integrated with existing MPP (massively parallel processing) relational databases, which allows a programmer to write MapReduce jobs over database rows and dump output into a new database table

Map-reduce additional elements



- Partitioners are responsible for dividing up the intermediate key space and assigning intermediate key-value pairs to reducers
  - the partitioner species the task to which an intermediate key-value pair must be copied
- Combiners are an optimization in MapReduce that allow for local aggregation before the shuffle and sort phase

Partitioners and combiners

- ***Partitioners*** are responsible for dividing up the intermediate key space and assigning intermediate key-value pairs to reducers, i.e., the ***partitioner*** specifies the task to which an intermediate key-value pair must be copied
  - Reducers process keys in sorted order (which is how the "group by" is implemented)
  - The simplest ***partitioner*** involves computing the hash value of the key and then taking the mod of that value with the number of reducers
  - This assigns approximately the same number of keys to each reducer (dependent on the quality of the hash function)

- ***Combiners*** "mini-reducers" that take place on the output of the mappers, prior to the shuffle and sort phase
  - Optimize MapReduce allowing for local aggregation before the shuffle and sort phase
    - All key-value pairs need to be copied across the network, and so the amount of intermediate data will be larger than the input collection itself → inefficient
    - One solution is to perform local aggregation on the output of each mapper, i.e., to compute a local count for a word over all the documents processed by the mapper
  - Can emit any number of key-value pairs, but the keys and values must be of the same type as the mapper output

  With this modification (assuming the maximum amount of local aggregation possible), the number of intermediate key-value pairs will be at most the number of unique words in the collection times the number of mappers (and typically far smaller because each mapper may not encounter every word)

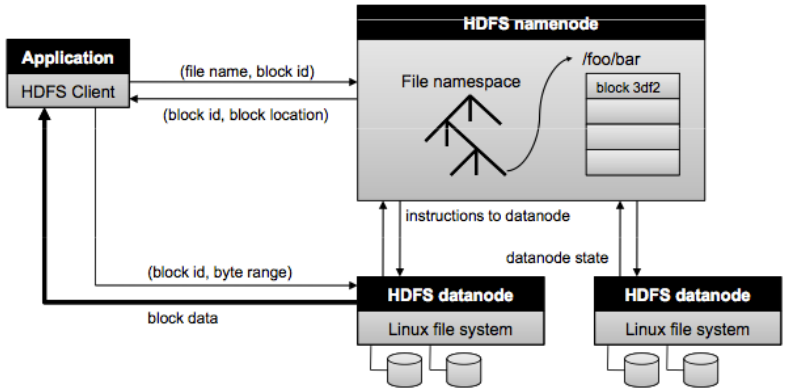
Execution framework

- Important idea behind MapReduce is separating the ***what*** of distributed processing from the ***how***
- A MapReduce program (job) consists of
  - code for mappers and reducers packaged together with
  - configuration parameters (such as where the input lies and where the output should be stored)
- Execution framework responsibilities: scheduling
  - Each MapReduce job is divided into smaller units called tasks
  - In large jobs, the total number of tasks may exceed the number of tasks that can be run on the cluster concurrently → manage tasks queues
  - Coordination among tasks belonging to different jobs

Distributed file system

- Abandons the separation of computation and storage as distinct components in a cluster
  - Google File System (GFS) supports Google's proprietary implementation of MapReduce;
  - In the open-source world, HDFS (Hadoop Distributed File System) is an open-source implementation of GFS that supports Hadoop
- The main idea is to divide user data into blocks and replicate those blocks across the local disks of nodes in the cluster
- Adopts a master–slave architecture
  - Master (`namenode` HDFS) maintains the file namespace (metadata, directory structure, file to block mapping, location of blocks, and access permissions)
  - Slaves (`datanode` HDFS) manage the actual data blocks

HFDS general architecture

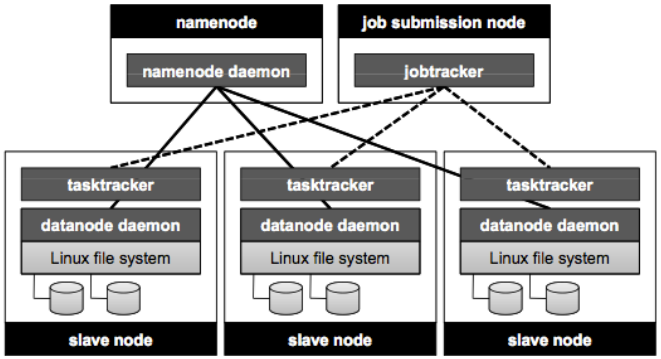


- An application client wishing to read a file (or a portion thereof) must first contact the namenode to determine where the actual data is stored
- The `namenode` returns the relevant `block id` and the `location` where the block is held (i.e., which `datanode`)
- The client then contacts the `datanode` to retrieve the data.
- HDFS lies on top of the standard OS stack (e.g., Linux): blocks are stored on standard single-machine file systems

HDFS properties

- HDFS stores three separate copies of each data block to ensure both reliability, availability, and performance
- In large clusters, the three replicas are spread across different physical racks,
  - HDFS is resilient towards two common failure scenarios individual `datanode` crashes and failures in networking equipment that bring an entire rack offline.
  - Replicating blocks across physical machines also increases opportunities to co-locate data and processing in the scheduling of MapReduce jobs, since multiple copies yield more opportunities to exploit locality
- To create a new file and write data to HDFS
  - The application client first contacts the `namenode`
  - The `namenode`
    - updates the file namespace after checking permissions and making sure the file doesn't already exist
    - allocates a new block on a suitable datanode
  - The application is directed to stream data directly to it
  - From the initial `datanode`, data is further propagated to additional replicas

Hadoop cluster architecture

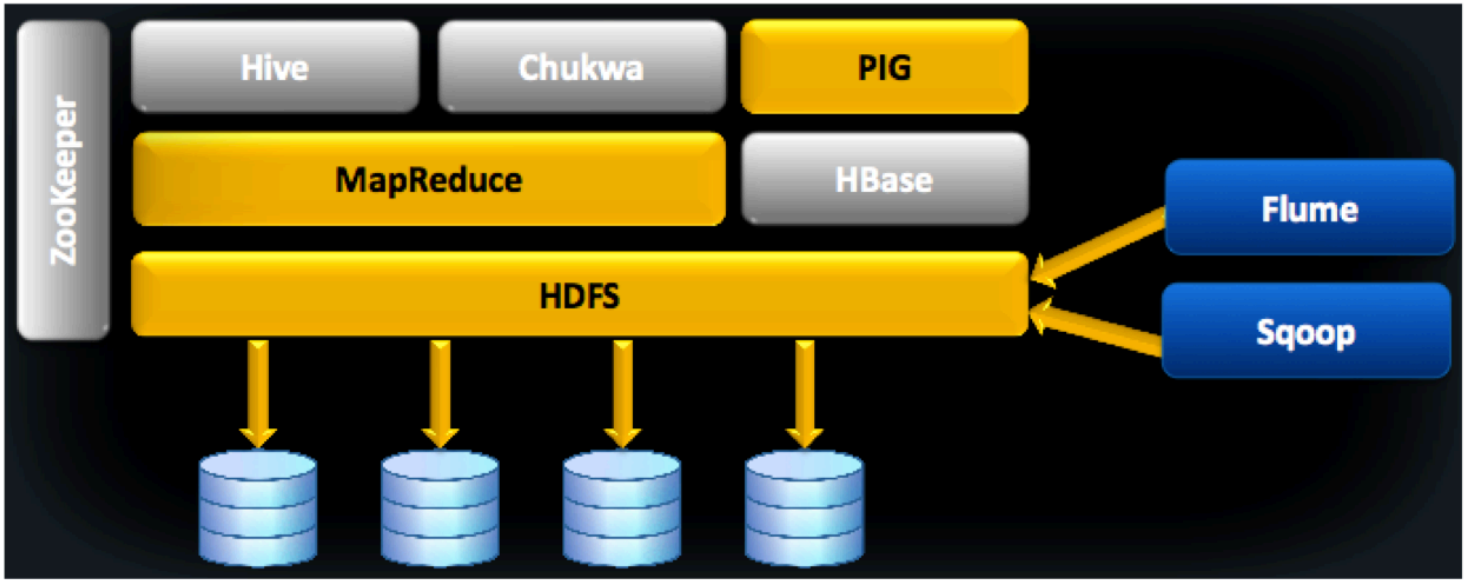


- The HDFS **namenode** runs the namenode daemon
- The job submission node runs the **jobtracker**, which is the single point of contact for a client wishing to execute a MapReduce job
- The **jobtracker**
  - Monitors the progress of running MapReduce jobs
  - Is responsible for coordinating the execution of the mappers and reducers
  - Tries to take advantage of data locality in scheduling map tasks

Hadoop cluster architecture

- `Tasktracker`
  - It accepts tasks (Map, Reduce, Shuffle, etc.) from `JobTracker`
  - Each `TaskTracker` has a number of slots for the tasks: these are execution slots available on the machine or machines on the same rack
  - It spawns a separate JVM for execution of the tasks
  - It indicates the number of available slots through the hearbeat message to the `JobTracker`

Hadoop infrastructure



The Apache™ Hadoop™ project develops open-source software
reliable, scalable, distributed computing.

Hadoop framework

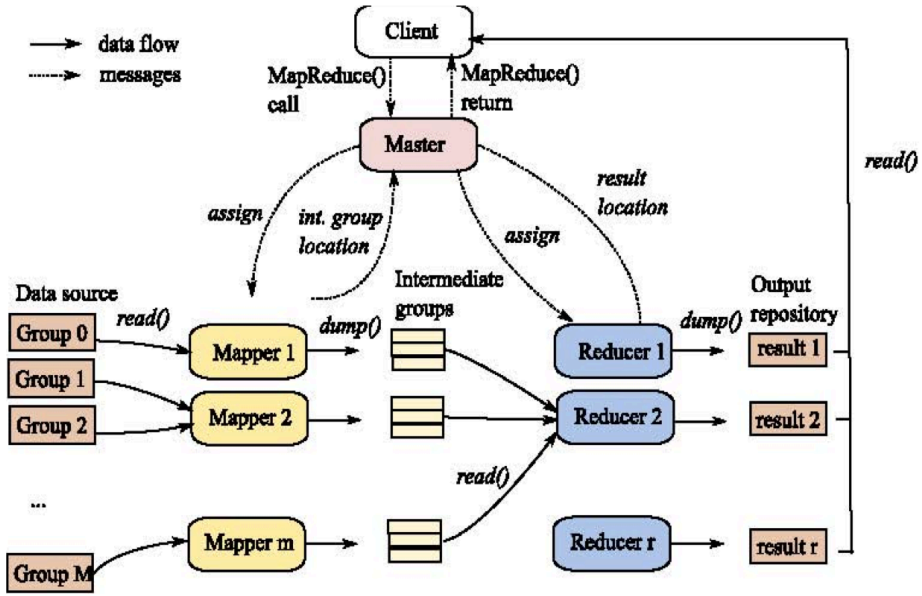
- **Hadoop Distributed File System** (HDFS): A distributed file system that provides high-throughput access to application data
- **Hadoop MapReduce**: A software framework for distributed processing of large data sets on compute clusters
- **HBase**: A scalable, distributed database that supports structured data storage for large tables
- **Hive**: A data warehouse infrastructure that provides data summarization and ad hoc querying
- **Chukwa**: A data collection system for managing large distributed systems
- **Pig**: A high-level data-flow language and execution framework for parallel computation
- **ZooKeeper**: A high-performance coordination service for distributed applications

Execution framework

- The developer submits a MapReduce program to the submission node of a cluster (in Hadoop, this is called the `jobtracker`)
- Execution framework takes care of everything else: it transparently handles all other aspects of distributed code execution, on clusters ranging from a single node to a few thousand nodes.
- Distribution, synchronization and failure are handled

Hadoop job execution

- The input is split in M groups
  - each group is assigned to a `mapper` (assignment is based on the data locality principle)
  - each `mapper` processes a group and stores the intermediate pairs locally
- Grouped instances are assigned to `reducers` thanks to a hash function
  - (***Shuffle***) intermediate pairs are sorted on their key by the reducer
  - grouped instances, submitted to the `reduce()` function



Hadoop job execution: mapper

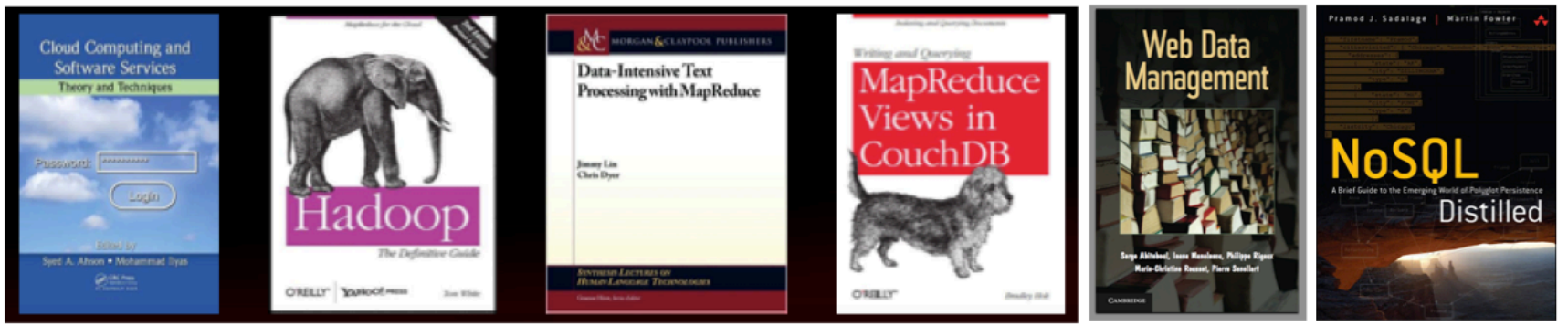
- In Hadoop, `mappers` are Java objects with a `Map` method (among others)
  - A `mapper` object is instantiated for every map task by the `tasktracker`
- The life-cycle of this object begins with
  - *Instantiation*, where a hook is provided in the API to run programmer-specified code
  - After *initialization*, the `Map` method is called (by the execution framework) on all key-value pairs in the input split.
  - Since these method calls occur in the context of the same Java object, it is possible to preserve state across multiple input key-value pairs within the same map task
  - After all key-value pairs in the input split have been processed, the `mapper` object provides an opportunity to run programmer-specified *termination* code

Hadoop job execution: reducer

- Each reducer object is *instantiated* for every reduce task.
  - The Hadoop API provides hooks for programmer-specified *initialization* and *termination* code.
  - After *initialization*, for each intermediate key in the partition (defined by the `partitioner`), the execution framework repeatedly calls the `Reduce` method with an intermediate key and an `iterator` over all values associated with that key
- The programming model also guarantees that intermediate keys will be presented to the `Reduce` method in sorted order
  - Since this occurs in the context of a single object, it is possible to preserve state across multiple intermediate keys (and associated values) within a single reduce task

Some books

- Hadoop The Definitive Guide – O'Reily 2011 – Tom White
- Data Intensive Text Processing with MapReduce – Morgan & Claypool 2010 –Jimmy Lin, Chris Dyer – pages 37-65
- Cloud Computing and Software Services Theory and Techniques– CRC Press 2011- Syed Ahson, Mohammad Ilyas – pages 93-137
- Writing and Querying MapReduce Views in CouchDB – O'Reily 2011 –Brandley Holt – pages 5-29
- NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence by Pramod J. Sadalage, Martin Fowler



DESIGNING MAP REDUCE ALGORITHMS

PATTERNS AND EXAMPLES

Context

- A large part of the power of MapReduce comes from its simplicity: in addition to preparing the input data, the programmer needs only to implement the mapper, the reducer, and optionally, the combiner and the partitioner
- any conceivable algorithm that a programmer wishes to develop must be expressed in terms of a small number of rigidly-defined components that must fit together in very specific ways.
- Synchronization is perhaps the most tricky aspect of designing MapReduce algorithms processes running on separate nodes in a cluster must, at some point in time, come together—for example, to distribute partial results from nodes that produced them to the nodes that will consume them.

Beyond the control of programmers

- Where a mapper or reducer runs (i.e., on which node in the cluster)
- 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

Under the control of programmers

- The ability to construct complex data structures as keys and values to store and communicate partial results.
- The ability to execute user-specified initialization code at the beginning of a map or reduce task, and the ability to execute user-specified termination code at the end of a map or reduce task.
- The ability to preserve state in both mappers and reducers across multiple input or intermediate keys.
- The ability to control the sort order of intermediate keys, and therefore the order in which a reducer will encounter particular keys.
- The ability to control the partitioning of the key space, and therefore the set of keys that will be encountered by a particular reducer.

Map-reduce phases

- Initialisation
- **Map**: *record reader, mapper, combiner, and partitioner*
- **Reduce:** *shuffle, sort, reducer, and output format*

- Partition input (key, value) pairs into chunks run map() tasks in parallel
- After all map()'s have been completed consolidate the values for each unique emitted key
- Partition space of output map keys, and run reduce() in parallel

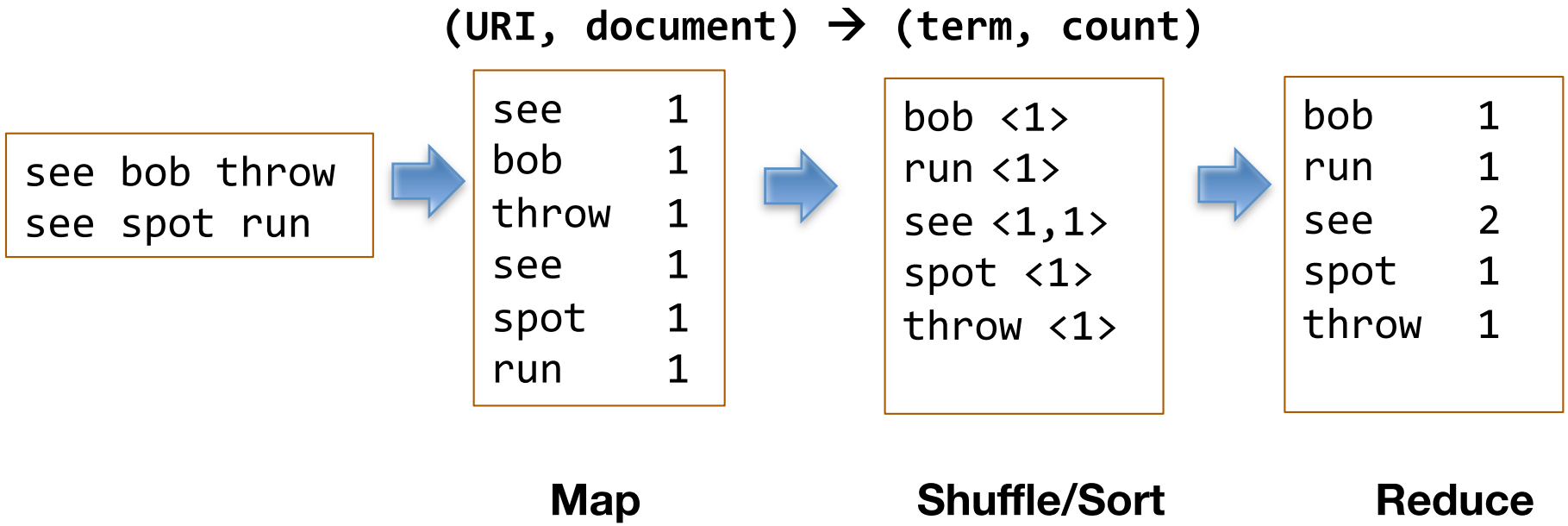
Map sub-phases

- *Record reader* translates an input split generated by input format into records
  - parse the data into records, but not parse the record itself
  - It passes the data to the mapper in the form of a key/value pair. Usually the key in this context is positional information and the value is the chunk of data that composes a record
- *Map* user-provided code is executed on each key/value pair from the record reader to produce zero or more new key/value pairs, called the intermediate pairs
  - The key is what the data will be grouped on and the value is the information pertinent to the analysis in the reducer
- *Combiner*, an optional localized reducer
  - Can group data in the map phase
  - It takes the intermediate keys from the mapper and applies a user-provided method to aggregate values in the small scope of that one mapper
- *Partitioner* takes the intermediate key/value pairs from the mapper (or combiner) and splits them up into shards, one shard per reducer

Reduce sub phases

- *Shuffle and sort* takes the output files written by all of the partitioners and downloads them to the local machine in which the reducer is running.
  - These individual data pieces are then sorted by key into one larger data list
  - The purpose of this sort is to group equivalent keys together so that their values can be iterated over easily in the reduce task
- *Reduce* takes the grouped data as input and runs a reduce function once per key grouping
  - The function is passed the key and an iterator over all of the values associated with that key
  - Once the reduce function is done, it sends zero or more key/value pair to the final step, the output format
- *Output format* translates the final key/value pair from the reduce function and writes it out to a file by a record writer

Counting words



Gold standard

- Linear scalability:
  - an algorithm running on twice the amount of data should take only twice as long
  - an algorithm running on twice the number of nodes should only take half as long
- Local aggregation: in the context of data-intensive distributed processing
  - the single most important aspect of synchronization is the exchange of intermediate results, from the processes that produced them to the processes that will ultimately consume them
  - Hadoop, intermediate results are written to local disk before being sent over the network
  - Since network and disk latencies are relatively expensive compared to other operations***, reductions in the amount of intermediate data translate into increases in algorithmic efficiency***
- Using the **combiner** and by taking advantage of the ability to **preserve state** across multiple inputs

  → it is possible to substantially reduce both the number and size of key-value pairs that need to be shuffled from the mappers to the reducers

Counting words basic algorithm

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term t ∈ doc d do
4:             Emit(term t, count 1)

1: class Reducer
2:     method Reduce(term t, counts [c1, c2, ...])
3:         sum ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             sum ← sum + c
6:         Emit(term t, count sum)
```

- the mapper emits an intermediate key-value pair for each term observed, with the term itself as the key and a value of one
- reducers sum up the partial counts to arrive at the final count

Local aggregation

Combiner technique

- Aggregate term counts across the documents processed by each map task
- Provide a general mechanism within the MapReduce framework to reduce the amount of intermediate data generated by the mappers
- Reduction in the number of intermediate key-value pairs that need to be shuffled across the network
  - from the order of total number of terms in the collection to the order of the number of ***unique*** terms in the collection

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1
6:         for all term t ∈ H do
7:             Emit(term t, count H{t})
```

In-mapper combining pattern: One step further

- The workings of this algorithm critically depends on the details of how map and reduce tasks in Hadoop are executed
- Prior to processing any input key-value pairs, the mapper's `Initialize` method is called
  - which is an API hook for user-specified code
  - We initialize an associative array for holding term counts
  - Since it is possible to preserve state across multiple calls of the `Map` method (for each input key-value pair), we can
    - continue to accumulate partial term counts in the associative array across multiple documents,
    - emit key-value pairs only when the mapper has processed all documents
- Transmission of intermediate data is deferred until the `Close` method in the pseudo-code

```
1: class Mapper
2:     method Initialize
3:         H ← new AssociativeArray
4:     method Map(docid a, doc d)
5:         for all term t ∈ doc d do
6:             H{t} ← H{t} + 1
7:     method Close
8:         for all term t ∈ H do
9:             Emit(term t, count H{t})
```

In-mapper combining pattern: advantages

- Provides control over when local aggregation occurs and how it exactly takes place
  - Hadoop makes no guarantees on how many times the combiner is applied, or that it is even applied at all
  - The execution framework has the option of using it, perhaps multiple times, or not at all
  - Such indeterminism is unacceptable, which is exactly why programmers often choose to perform their own local aggregation in the mappers

- In-mapper combining will typically be more efficient than using actual combiners.
  - One reason for this is the additional overhead associated with actually materializing the key-value pairs
    - Combiners reduce the amount of intermediate data that is shuffled across the network, but don't actually reduce the number of key-value pairs that are emitted by the mappers in the first place
    - The mappers will generate only those key-value pairs that need to be shuffled across the network to the reducers
    - Avoid unnecessary object creation and destruction (garbage collection takes time), and, object serialization and deserialization (when intermediate key-value pairs fill the in-memory buffer holding map outputs and need to be temporarily spilled to disk)

In-mapper combining pattern: limitations

- Breaks the functional programming underpinnings of MapReduce, since state is being preserved across multiple input key-value pairs
- There is a fundamental scalability bottleneck associated with the in-mapper combining pattern
  - It critically depends on having sufficient memory to store intermediate results until the mapper has completely processed all key-value pairs in an input split
  - One common solution to limiting memory usage is to “block” input key-value pairs and “flush” in-memory data structures periodically
  - Instead of emitting intermediate data only after every key-value pair has been processed, emit partial results after processing every n key-value pairs
    - Implemented with a counter variable that keeps track of the number of input key-value pairs that have been processed
    - The mapper could keep track of its own memory footprint and flush intermediate key-value pairs once memory usage has crossed a certain threshold
    - Memory size empirically determined: difficult due to concurrent access to memory

Some remarks

- In MapReduce algorithms, the extent to which efficiency can be increased through local aggregation depends
  - on the size of the intermediate key space
  - the distribution of keys themselves
  - the number of key-value pairs that are emitted by each individual map task

- Opportunities for aggregation come from having multiple values associated with the same key (whether one uses combiners or employs the in-mapper combining pattern)
  - In the word count example, local aggregation is effective because many words are encountered multiple times within a map task
  - In our word count example, we do not filter frequently-occurring words: therefore, without local aggregation, the reducer that is responsible for computing the count of *'the'* will have a lot more work to do than the typical reducer, and therefore will likely be a straggler
  - With local aggregation (either combiners or in-mapper combining), we substantially reduce the number of values associated with frequently-occurring terms, which alleviates the reduce straggler problem

DISCUSSION OF THE FIRST PART OF CHALLENGE II

COROLLARY CHALLENGE: WORD COUNT ON “STACKOVERFLOW” POSTS (*pp 7-11 of MapReduce design patterns book*)

Map – reduce design patterns

Summarization

Numerical

- Minimum, maximum, count, average, median-standard deviation

Inverted index

- Wikipedia inverted index

Counting with counters

- Count number of records, a small number of unique instances, summations
- Number of users per state

Filtering

Filtering

- Closer view of data, tracking event threads, distributed grep, data cleansing, simple random sampling, remove low scoring data

Bloom

- Remove most of nonwatched values, prefiltering data for a set membership check
- Hot list, Hbase query

Top ten

- Outlier analysis, select interesting data, catchy dashbords
- Top ten users by reputation

Distinct

- Deduplicate data, getting distinct values, protecting from inner join explosion
- Distinct user ids

Data organization

Structured to hierarchical

- Prejoining data, preparing data for Hbase or MongoDB
- Post/comment building for StackOverflow, Question/Answer building

Partitioning

- Partitioning users by last access date

Binning

- Binning by Hadoop-related tags

Total order sorting

- Sort users by last visit

Shuffling

- Anonymizing StackOverflow comments

Join

Reduce side join

- Multiple large data sets joined by foreign key
- User – comment join

Reduce side join with bloom filter

- Reputable user – comment join

Replicated join

- Replicated user – comment join

Composite join

- Composite user – comment join

Cartesian product

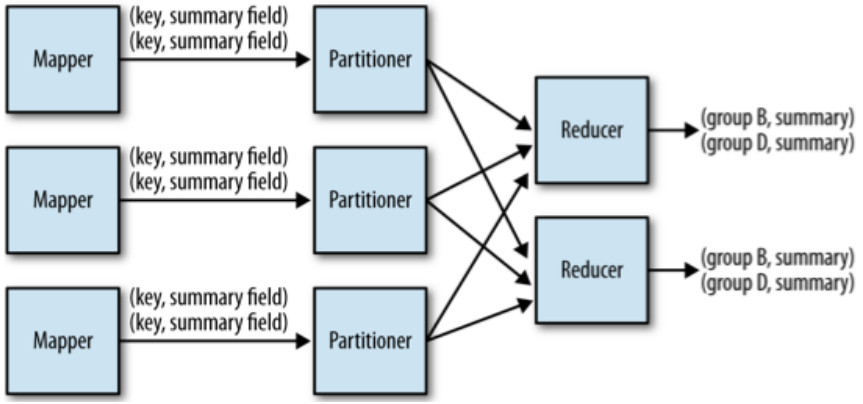
- Comment comparison

Numerical Summarization pattern

- The numerical summarizations pattern is a general pattern for calculating aggregate statistical values over a data collection
- *Intent*
  - Group records together by a key field and calculate a numerical aggregate per group to get a top-level view of the larger data set
  - θbe a generic numerical summarization function we wish to execute over some list of values $(v_1, v_2, v_3, \ldots, v_n)$ to find a value λ, i.e. $\lambda = \theta(v_1, v_2, v_3, \ldots, v_n)$. Examples of θ include a minimum, maximum, average, median, and standard deviation
- *Motivation and applicability*
  - Group logins by the hour of the day and perform a count of the number of records in each group, group advertisements by types to determine how affective ads are for better targeting
  - Dealing with numerical data or counting
  - The data can be grouped by specific fields

Structure

- The mapper outputs keys that consist of each field to group by, and values consisting of any pertinent numerical items
- The combiner can greatly reduce the number of intermediate key/value pairs to be sent across the network to the reducers for some numerical summarization functions
  - If the function θ is an associative and commutative operation, it can be used for this purpose
  - If you can arbitrarily change the order of the values and you can group the computation arbitrarily
- The reducer
  - receives a set of numerical values $(v_1, v_2, v_3, \ldots, v_n)$ associated with a group-by key records to perform the function $\lambda = \theta(v_1, v_2, v_3, \ldots, v_n)$
  - The value of λ is output with the given input key



Resemblances and performance analysis

Resemblances

SQL

The Numerical Aggregation pattern is analogous to using aggregates after a GROUP BY in SQL:

```
SELECT MIN(numericalcol1), MAX(numericalcol1),
       COUNT(*) FROM table GROUP BY groupcol2;
```

Pig

The GROUP ... BY expression, followed by a FOREACH ... GENERATE:

```
b = GROUP a BY groupcol2;
c = FOREACH b GENERATE group, MIN(a.numericalcol1),
       MAX(a.numericalcol1), COUNT_STAR(a);
```

Performance analysis

- Aggregations performed by jobs using this pattern typically perform well when the combiner is properly used
- These types of operations are what MapReduce was built for

Maximum, minimum and count example

- *Problem*
  - Given a list of user's comments, determine the first and last time a user commented and the total number of comments from that user
- *Principle*
  - After a grouping operation, the reducer simply iterates through all the values associated with the group and finds the min and max, as well as counts the number of members in the key grouping
  - Due to the associative and commutative properties, a combiner can be used to vastly cut down on the number of intermediate key/value pairs that need to be shuffled to the reducers

Maximum, minimum and count example

- `Mapper`
  - The mapper will pre-process input values by extracting the attributes from each input record: the creation date and the user identifier
  - The output key is the user ID and the value is three columns of our future output: the minimum date, the maximum date, and the number of comments this user has created
- `Reducer`
  - The reducer iterates through the values to find the minimum and maxi- mum dates, and sums the counts
- `Combiner`
  - As we are only interested in the count, minimum date, and maximum date, multiple comments from the same user do not have to be sent to the reducer.
  - The minimum and maximum comment dates can be calculated for each local map task without having an effect on the final minimum and maximum.

average example

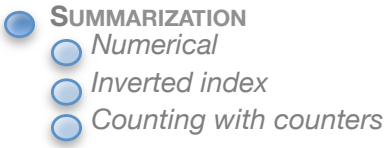
- *Problem*: given a list of user's comments, determine the average comment length per hour of day
  - The mapper processes each input record to calculate the average comment length based on the time of day
    - The output key is the hour of day
    - The output value is two columns, the comment count and the average length of the comments for that hour
    - Because the mapper operates on one record at a time, the count is simply 1 and the average length is equivalent to the comment length
  - The reducer iterates through all given values for the hour and keeps two local variables: a running count and running sum
    - For each value, the count is multiplied by the average and added to the running sum
    - The count is simply added to the running count

Median and standard deviation (CH-3$_a$)

- *Problem*
  - Given a list of user's comments, determine the median and standard deviation of comment lengths per hour of day
  - A median is the numerical value separating the lower and higher halves of a data set
    - This requires the data set to be complete, which in turn requires it to be shuffled
    - The data must also be sorted, which can present a barrier because MapReduce does not sort values
  - A standard deviation shows how much variation exists in the data from the average, thus requiring the average to be discovered prior to reduction

implementation

- Mapper processes each input record to calculate the median comment length within each hour of the day
  - The output key is the hour of day
  - The output value is a single value: the comment length
- The reducer iterates through the given set of values and adds each value to an in-memory list
  - The iteration also calculates a running sum and count
  - After iteration, the comment lengths are sorted to find the median value
    - If the list has an odd number of entries, the median value is set to the middle value
    - If the number is even, the middle two values are averaged
  - Next, the standard deviation is calculated by iterating through our sorted list after finding the mean from our running sum and count
  - A running sum of deviations is calculated by squaring the difference between each comment length and the mean.
  - The standard deviation is then calculated from this sum.
  - Finally, the median and standard deviation are output along with the input key



Inverted index pattern

- The inverted index pattern is commonly used as an example for MapReduce analytics
- *Intent*
  - Generate an index from a data set to allow for faster searches or data enrichment capabilities
- *Motivation and applicability*
  - Building an inverted index, a search engine knows all the web pages related to a keyword ahead of time and these results are simply displayed to the user
  - Inverted indexes should be used when quick search query responses are required. The results of such a query can be pre-processed and ingested into a database

Inverted index reminder

- Index data structure storing a mapping from content, such as words or numbers, to its locations in a database file, or in a document or a set of documents.
- The purpose of an inverted index is to allow fast full text searches, at a cost of increased processing when a document is added to the database

```
T[0] = "it is what it is"
T[1] = "what is it"
T[2] = "it is a banana”
```

```
"a":      {2}
"banana": {2}
"is":     {0, 1, 2}
"it":     {0, 1, 2}
"what":   {0, 1}
```

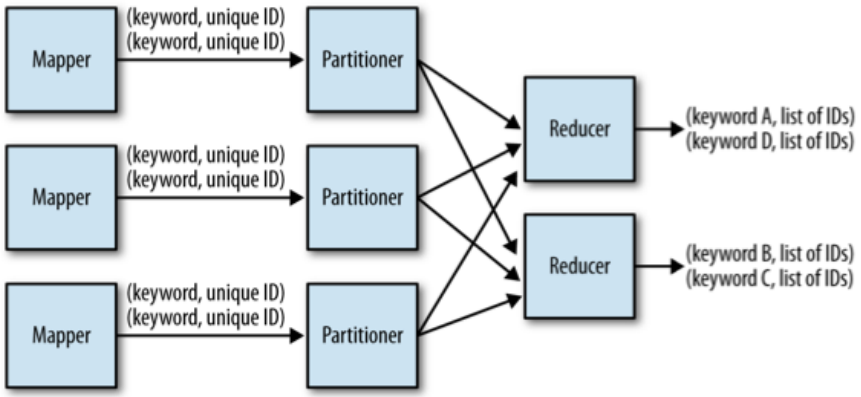
- A term search for the terms `"what"`, `"is"` and `"it"` would give the set `{0,1,2} ∩ {0,1,2} ∩ {0,1} = {0,1}`

```
"a":      {(2, 2)}
"banana": {(2, 3)}
"is":     {(0, 1), (0, 4), (1, 1), (2, 1)}
"it":     {(0, 0), (0, 3), (1, 2), (2, 0)}
"what":   {(0, 2), (1, 0)}
```

- If we run a phrase search for `"what is it"` we get hits for all the words in both document 0 and 1. But the terms occur consecutively only in document 1.

Structure

- The `mapper` outputs the desired fields for the index as the key and the unique identifier as the value
- The `combiner` can be omitted if you are just using the identity `reducer`
  - Some implementations concatenate the values associated with a group before outputting them to the file system. In this case, a combiner can be used
  - It won't have as beneficial an impact on byte count as the combiners in other patterns, but there will be an improvement
- The `partitioner`
  - Is responsible for determining where values with the same key will eventually be copied by a reducer for final output.
  - It can be customized for more efficient load balancing if the intermediate keys are not evenly distributed.
- The `reducer` will receive a set of unique record identifiers to map back to the input key
  - The identifiers can either be concatenated by some unique delimiter, leading to the output of one key/value pair per group, or



Performance analysis

- The performance of building an inverted index depends mostly
  - on the computational cost of parsing the content in the mapper
  - the cardinality of the index keys
  - the number of content identifiers per key

- If the number of unique keys and the number of identifiers is large, more data will be sent to the reducers. If more data is going to the reducers, you should increase the number of reducers to increase parallelism during the reduce phase

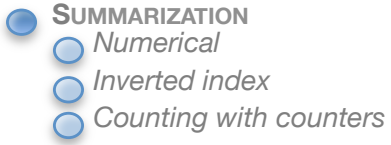
- Inverted indexes are particularly susceptible to hot spots in the index keys, since the index keys are rarely evenly distributed
  - For example, the reducer that handles the word “the” in a text search application is going to be particularly busy since “the” is seen in so much text
  - This can slow down your entire job since a few reducers will take much longer than the others
  - To avoid this problem, you might need to implement a custom partitioner, or omit common index keys that add no value to your end goal

Wikipedia inverted index (ch-3$_b$)

- *Problem*
  - We want to add *StackOverflow* links to each *Wikipedia* page that is referenced in a *StackOverflow* comment
  - Given a set of user's comments, build an inverted index of Wikipedia URLs to a set of answer post IDs

Implementation

- `Mapper`
  - Parses the posts from StackOverflow to output the row IDs of all answer posts that contain a particular Wikipedia URL
- `Reducer`
  - Iterates through the set of input values and appends each row ID to a `String`, delimited by a space character
  - The input key is output along with this concatenation
- `Combiner` can be used to do some concatenation prior to the reduce phase
  - Because all row IDs are simply concatenated together, the number of bytes that need to be copied by the reducer is more than in a numerical summarization pattern
  - The same code for the reducer class is used as the combiner



Counting with counters pattern

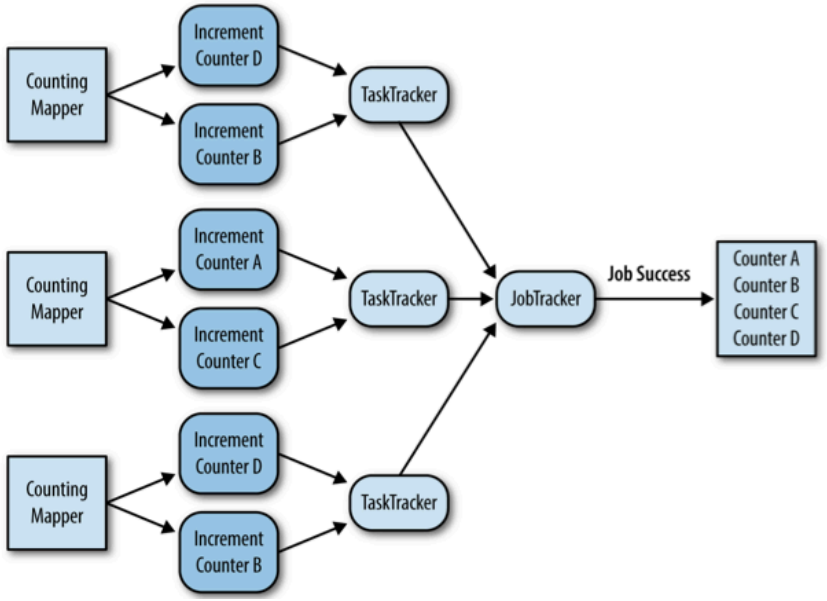
- This pattern utilizes the MapReduce framework's counters utility to calculate a global sum entirely on the map side without producing any output
- *Intent*
  - An efficient means to retrieve count summarizations of large data sets
- *Motivation and applicability*
  - Hourly ingest record counts can be post processed to generate helpful histograms. This can be executed in a simple "word count" manner but it can be done more efficiently using counters
  - Find the number of times your employees log into your heavily used public website every day

Principle and applicability

- *Principle*
  - Instead of writing any key value pairs at all, simply use the framework's counting mechanism to keep track of the number of input records
  - This requires no reduce phase and no summation!
  - The framework handles monitoring the names of the counters and their associated values, aggregating them across all tasks, as well as taking into account any failed task attempts
- *Applicability*
  - You have a desire to gather counts or summations over large data sets
  - The number of counters you are going to create is small—in the double digits

Structure

- The `mapper` processes each input record at a time to increment counters based on certain criteria.
  - The counter is either incremented by one if counting a single instance, or incremented by some number if executing a summation
  - These counters are then
    - aggregated by the `TaskTrackers` running the tasks and incrementally reported to the `JobTracker` for overall aggregation upon job success
    - The counters from any failed tasks are disregarded by the `JobTracker` in the final summation
- As this job is map only, there is no `combiner`, `partitioner`, or `reducer` required



Thanks Merci Gracias

Contact: Genoveva Vargas-Solar, CNRS, LIG-LAFMIA
Genoveva.Vargas@imag.fr
http://www.vargas-solar.com/teaching