Big Data Computing Models
Map-Reduce

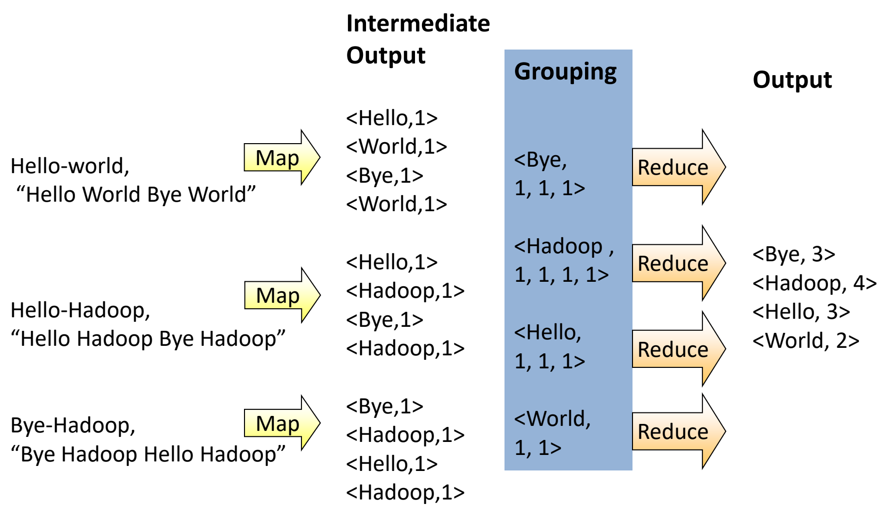
What is Map-Reduce?

- Parallel Programming Models
  - Message passing
    - Independent tasks encapsulating local data, interact by exchanging messages
  - Shared memory
    - Tasks share a common address space and interact by reading and writing this space asynchronously
  - Data parallelization
    - Perform the same operations to a set of data
    - Also referred to as "Embarrassingly parallel"
- Map reduce is a form of data parallelization

What is Map-Reduce?

- Map-reduce makes parallel programming easier
  - Coarse grained data parallel computing
  - User only writes a mapper and a reducer
  - MapReduce system takes care of the rest
- Use functional programming model
  - Function as an argument
  - Map/reduce (function (list))
    - Map (square (1 2 3 4 5)) = (1 4 9 16 25)
    - Reduce (+ (1 2 3 4 5)) = 15

Example -- Word Count



https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

Example -- Word Count

- Map and reduce program

```
map (key, value):
    // key: document name; value: text of document
    for each word w in value:  emit(w, 1)
reduce (key, values):
    // key: a word; values: the count of the word
    result = 0
    for each v in values:  result += v
    emit (key, result)
```

emit in java mapreduce:
  setup(Context context);
  context.write (…);

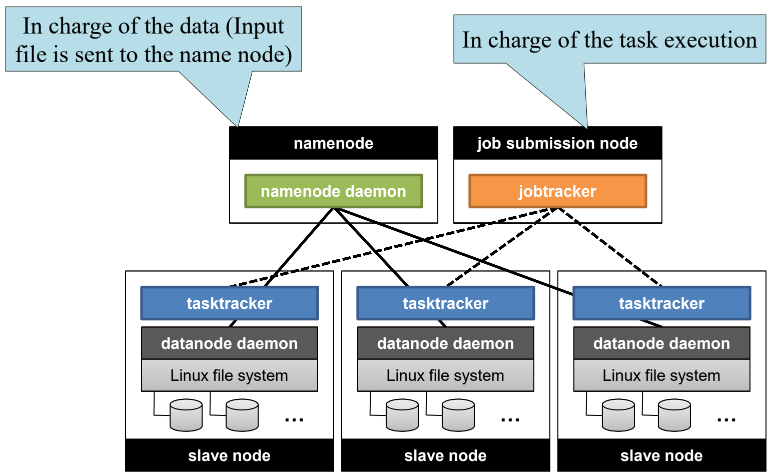
  - In a large document
    - Input to word count
      - Key: line number; value: text in the line

https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html#Example:_WordCount_v2.0

Map-Reduce Process

- Input
  - A list of key/value pairs: list(key, val)
- User supplies two functions:
  - Map-function (list(key, val)) → list(key', val')
    - list(key', val') is a list of intermediate key-value pairs generated from mapping function
    - Original list and new list may not be in the same size
    - One key' may have many different values
  - Reduce-function (list(key_i', val')) → v_i, for all i
- The execution framework handles everything else

Hadoop Map-Reduce Framework



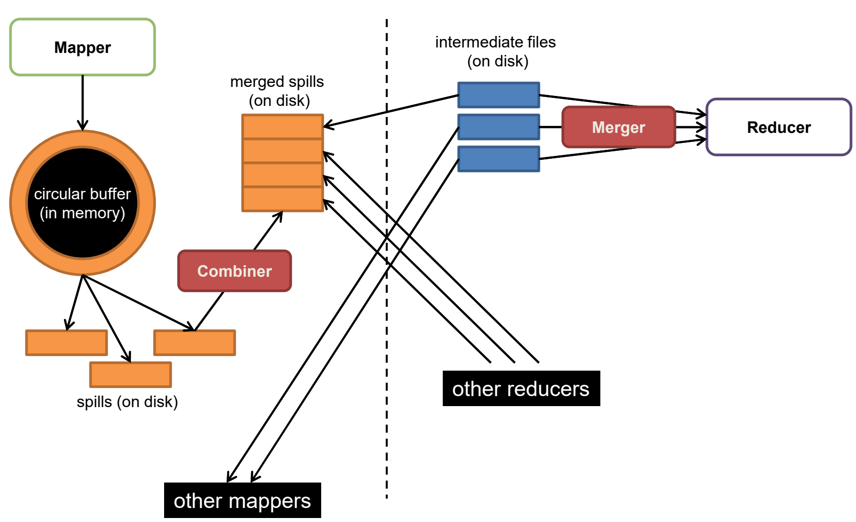
Hadoop Map-Reduce Framework

- Master-Worker structure
  - MapReduce job is sent to the master node JobTracker
  - JobTracker
    - Coordinate the execution of the tasks
      - Distributes the mapper task to TaskTrackers
      - Starts the TaskTrackers with the tasks
      - Wait for TaskTrackers' results
        - If some TaskTrakers have failed, redistribute the missed tasks
      - Determines reduce task allocation and sends TaskTrackers the tasks
        - Give reducer the intermediate file locations and sizes to reducers
      - Wait for TaskTrackers' results
        - If some TaskTrakers have failed, redistribute the missed tasks
    - Also periodically ping the TaskTrakers to keep track of their status

Hadoop Map-Reduce Framework

- Master-Worker structure
  - TaskTracker -- Map phase
    - Creates map task instances and runs them
      - Intermediate results are stored in a circular buffer in memory
      - When buffer is full, spill to local disk
    - Combiner combines intermediate results, sort them by key
      - Combiner mostly uses a hash function
    - Sends locations and sizes of intermediate files to the JobTracker
  - TaskTracker -- Reduce phase
    - Merger obtains intermediate results from mappers and merge them
    - Creates reduce task instances and runs them
    - Send the results to the JobTracker

Hadoop Map-Reduce Framework



Hadoop Map-Reduce Framework

- Master-Worker structure
  - Final output
    - TaskTracker responds to job issuer about the completion of the task
  - Final output may go for next map-reduce task
    - In this case, programmer does not combine the output files into one
    - Simply passes the output files (same name) as input to another MapReduce call
      - Cannot have shuffling from reduce to next round map
  - Map-reduce input and output
    - Always files (or a nosql-DB) that are partitioned into blocks and distributed over multiple nodes

Hadoop Map-Reduce Framework

- Synchronization
  - Reduce phase does not start till map phase is fully done
    - Choose to avoid complex issues in staring some reducers early
    - Use the idle workers to help speed up the remaining map phase
- When most subtasks (map or reduce) are done
  - Slow workers significantly delay completion time
    - Bad disks w/ soft errors transfer data slowly
    - Worker failure
  - Use multiple workers to execute the remaining tasks to ensure completion
    - Anyway these workers are idle

Hadoop Map-Reduce Framework

- Fault tolerance
  - Always assign one subtask to three workers
    - Even if all replicas fail ⇒ No results are generated ⇒ freed-up workers will re-execute the task
    - During a massive failure, lost 1600/1800 nodes, task finished fine
  - Do not handle master failure at this point
- When tasks fail
  - Map/Reduce functions sometimes fail for particular inputs
  - The worker sends the problem to the master node
    - Include sequence number of record being processed
  - If master sees two failures for the same record
    - Give up the subtask and stop all worker replicas for the same record

Hadoop Map-Reduce Framework

- Mini Reducer
  - Programmer may specify a "Combiner" function that does data merging on a single node
  - Typically, the same code is used to implement both the combiner and the reduce function
  - Example
    - Word count: Compute the cumulative count at each node
    - In order for this to work, the reduce function should be commutative and associative

Hadoop Map-Reduce Framework

- Important performance booster
  - Compress the mapper output to reduce the communication cost during shuffling
    - Tradeoff between compression/decompression cost and communication cost
    - Hadoop offers compression option

Evolution of Map Reduce Frameworks

What's New in MapReduce?

- Data parallel computation
  - In 80s, MIT developed the Connection Machine
    - Also called the thinking machine
    - The corresponding parallel programming paradigm is called data parallel programming (or more generally, the SIMD programming model)
    - Map (instructions (list))
  - In 80s, Teradata and Gamma project developed a new parallel database paradigm
    - Partition database row-wise and distribute partitions to nodes
    - Perform database operations in parallel in a cluster
  - What is new in MapReduce?

SIMD Programming

How is MapReduce different from SIMD

- Programming model
  - Define the basic data types
    - Can be scalar data type, or data structures
  - Define a special array of a basic data type
    - The special array is distributed over the computing nodes
      - E.g., in connection machine, it is called "shape"
      - E.g., shape data[N], sqdata[N];
  - Computation is instruction based
    - Computation in MapReduce is task based, more coarse-grained
      - E.g., Map: sqdata[i] = square (data[i]);
      - E.g., Reduce: sum = +sqdata[i];
  - Communication is specified based on array indices
    - Communication in MapReduce is specified by keys
      - E.g., data[i] = data[j] + data[j+1]

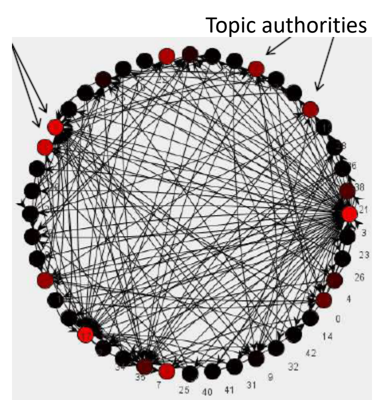
Issues in Hadoop -- No Loop

- ❖ MapReduce does not support iterative computing
  - ➢ And, fixed M-R-M-R flow (how about M-R-R-M)
- ❖ Multiple stages of MapReduce
  - ➢ User has to match I/O data files
    - ■ Create two JobConf objects, first job has the normal input, but output to temp, for example, and second job has temp as input and produces the normal output file
  - ➢ Or define job dependencies through jobControl object
    - ■ Job job1 = new Job(…); Job job2 = new Job(…);
    - ■ JobControl jbcntrl = new JobControl("jbcntrl");
      jbcntrl.addJob(job1); jbcntrl.addJob(job2);
      job2.addDependingJob(job1);
    - ■ jbcntrl.run();

Loop Example -- Page Ranking

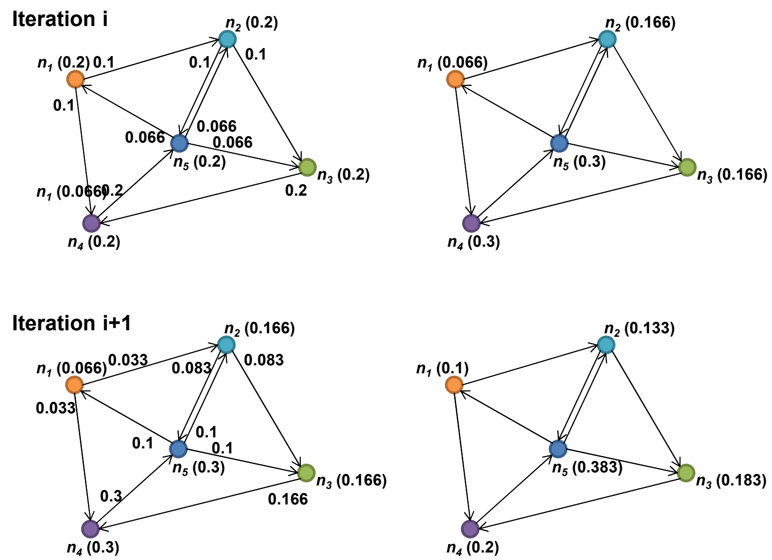
- ❖ A common task for Google
- ❖ Definition
  - ➢ Given a page x, with inlink from pages $t_1, t_2, \ldots, t_n$
    - ■ $C(t_i)$: out degree of page t_i
    - ■ α: probability of random jump
    - ■ N: the total number of nodes in the graph

$$PR(x) = \alpha\left(\frac{1}{N}\right) + (1-\alpha)\sum_{i=1}^{n}\frac{PR(t_i)}{C(t_i)}$$



Example -- Page Ranking

$$PR(x) = \alpha\left(\frac{1}{N}\right) + (1-\alpha)\sum_{i=1}^{n}\frac{PR(t_i)}{C(t_i)}$$



Page Ranking in Hadoop

- Page rank computation
  - Stage 0: Process each page *x*
    - Map phase:
      - Initiate *x*'s page rank, *x*.rank, to 1
      - Process page content, compute *x*.outlist (the list of *x*'s outgoing links), and compute *x*.#links (the total number of outgoing links in *x*)
      - Compute credit = *x*.rank / *x*.#links
      - For each *url* in *x*.outlist, create an item <key=*url*, val=credit>
    - Reduce phase:
      - Each reducer will receive <key=*y*, value=credit> from all its inlinks
      - Compute the sum of credits and assign it to *y*.rank
  - Note: After reduce, there is no shuffle and exchange
    - *y*.rank is the output and is to be used in the next iteration

Page Ranking in Hadoop

- Page rank computation
  - Stage *i*:
    - Map phase: (only distribute each rank received)
      - Need *x*.rank, *x*.outlist and *x*.#links
        - » *x*.rank is from the previous round
        - » *x*.outlist and *x*.#links has been generated in the first round
      - Compute credit = *x*.rank / *x*.#links
      - For each *url* in *x*.outlist, create an item <key=*url*, val=credit>
    - Reduce phase
      - Same as Stage 0
  - Termination check
    - No change of page rank for all pages

Issues in Hadoop -- No Loop

- MapReduce does not support iterative computing
    - User has to do it outside MapReduce $\Rightarrow$ Inefficient solution

      Why inefficient?
- Haloop MapReduce
  - Provide new constructs for expressing loops
    - $D_{i+1} = f(D_i, L)$
    - D_{i+1}: reducer output of the i-th iteration, which will be the mapper input of the (i+1)-th iteration
      - In Hadoop, they would be stored in HDFS Inefficient!
    - L: invariant data, does not change over the iterations
      - In Hadoop, they were in memory and would be gone after one MR

        Inefficient!
  - Termination
    - Programmer defines the termination condition
      - E.g., $D_{i+1} = D_i$, or $|D_{i+1} - D_i| < \epsilon$, or reach a certain bound, or …

Issues in Hadoop -- No Loop

- Haloop MapReduce
  - Workers maintain cache and local files
    - Cache the reducer outputs (D_{i-1}), which will be the input of the mapper in the next iteration   What are these in the page rank program?
    - Cache the invariant data (L) in the first iteration to allow reusing in later iterations   What are these in the page rank program?
    - Perform local termination evaluation
      - Master sends the evaluation criteria with the reduce tasks
      - Send the evaluation result to Master
  - Master
    - Repeats the map-reduce program
    - Coordinate the check of termination condition
      - Master acts as the single node reducer

Issues in Hadoop -- Data Placement

- Data placement problem
  - Sample problem: Join
    - Get the list of names and emails of customers with transaction \$ amount > \$1000 (for targeted customer retention)
      - Customer info table
        - Sorted by C.custId
      - Sales transaction table
        - Sorted by transaction time

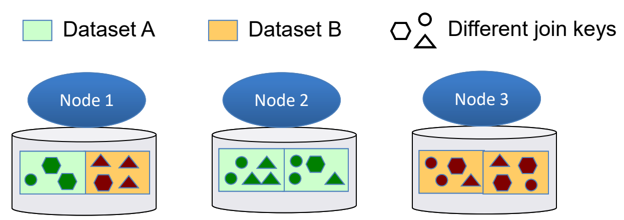
```
SELECT C.name, C.email
FROM Customers C, Sales S
WHERE C.custId = S.custId
AND S.amount > 1000
AND S.date BETWEEN
“12/1/15” AND “12/25/15”;
```

  - HDFS
    - Does not allow users to specify how the input files should be partitioned and placed
  - Improved systems: Hadoop++, Liah, CoHadoop, …

Issues in Hadoop -- Data Placement

- Hadoop
  - Data are partitioned according to the original input file order and placed on multiple nodes at random
  - Need to first perform MapReduce to align the keys in the two files

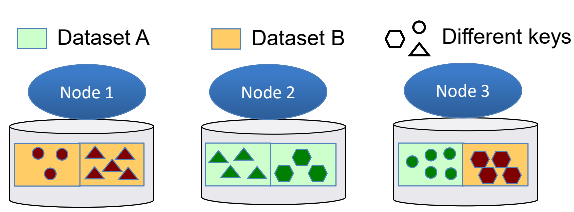
SELECT C.name, C.email
FROM Customers C, Sales S
WHERE C.custId = S.custId
AND S.amount > 1000
AND S.date BETWEEN
“12/1/15” AND “12/25/15”;



Issues in Hadoop -- Data Placement

- Hadoop++
  - Data are pre-partitioned into splits
    - User can determine which key to use for pre-partitioning (e.g. custId in the example)
    - A Trojan index describes each split
      - Tree based indexing of splits (key range, #records, …)
      - Stored on disk, allow easy cache in memory
      - Not available in Hadoop

SELECT C.name, C.email
FROM Customers C, Sales S
WHERE C.custId = S.custId
AND S.amount > 1000
AND S.date BETWEEN
“12/1/15” AND “12/25/15”;



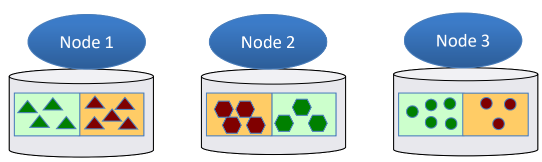
After prepartition:
Data with the same keys
are placed in the same split
(e.g., custID given by user)

Issues in Hadoop -- Data Placement

- ❖ Hadoop++
  - For PageRank example
    - Many links from a page are in the same domain
      ⇒ Partition the links by domain name instead of by the entire url
      ⇒ Can greatly reduce shuffling cost for each iteration

Issues in Hadoop -- Data Placement

- ❖ CoHadoop
  - Data are pre-partitioned based on the key (like Hadoop++)
  - Also support co-location for multiple datasets
    - Use a locator group concept
    - A data object can have a locator number
    - Multiple data objects with the same locator number are in one locator group and are co-located (blocks with the same key range are placed on the same node)



Issues in Hadoop

- Minor issue
  - Does not perform well on heterogeneous clusters
    - Hadoop attempts to distribute loads evenly
      - E.g., divide a data file evenly across the nodes
    - If the physical nodes have different computing capacity
      - The scheduler will reschedule the tasks for slow nodes to fast nodes
      - Require data movement from slow nodes to fast nodes
    - If partitioning of the data can go by the computing capacity
      - All nodes will progress in a similar rate
  - Easy to fix
    - E.g., using virtual nodes to make the system homogeneous
  - This can also be considered as a data placement problem

Issues in Hadoop

- Require detailed programming
  - Does not support SQL $\Rightarrow$ Need detailed coding for some standard DB operations
    - E.g., select, join, group-by, …
    - Complaints mainly from the DB community
  - Solution
    - Provide standard DB operations on top of MapReduce
    - E.g., Hive, Pig, Shark, …
      - Hive (HQL), originally by Facebook, then shifted to Apache
      - Pig (Pig Latin), originally by Yahoo, then shifted to Apache
      - Shark: Hive on Spark

Pig

- Pig: http://incubator.apache.org/pig
  - Developed by Yahoo!, now open source
  - Use Pig Latin, a dataflow language
    - Why not SQL?
      - Too complex to achieve efficient large data processing
      - Eliminate the non-common constructs, suitable for almost all commonly used queries
  - Support a fully-nestable data model
    - More natural and flexible than relational DB's flat tuples
    - Avoids expensive joins as much as possible
    - E.g., (yahoo , { finance email news })

Similar to the data model of some NoSQL DB systems

Pig -- Example

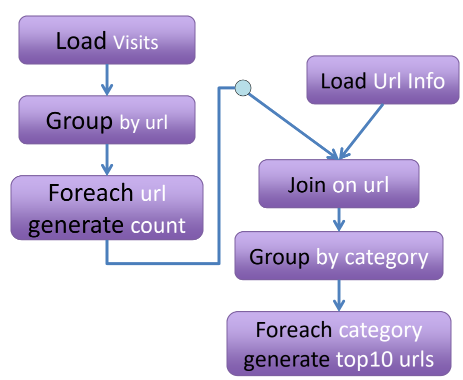
- Example
  - Given two tables: Visits, url-Info
    - Originally: Visits: ordered by time; url-info: ordered by url
  - Find the top 10 most visited pages in each category

Visits

User	Url	Time
Amy	cnn.com	8:00
Amy	bbc.com	10:00
Amy	flickr.com	10:05
Fred	cnn.com	12:00

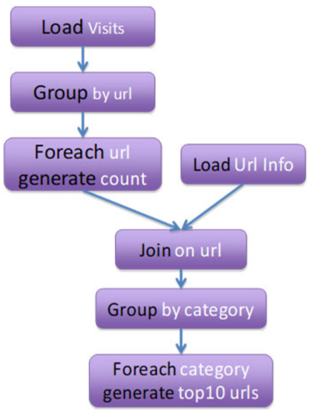
url-Info

Url	Category	PageRank
cnn.com	News	0.9
bbc.com	News	0.8
flickr.com	Photos	0.7
espn.com	Sports	0.9



Pig -- Example

- Example

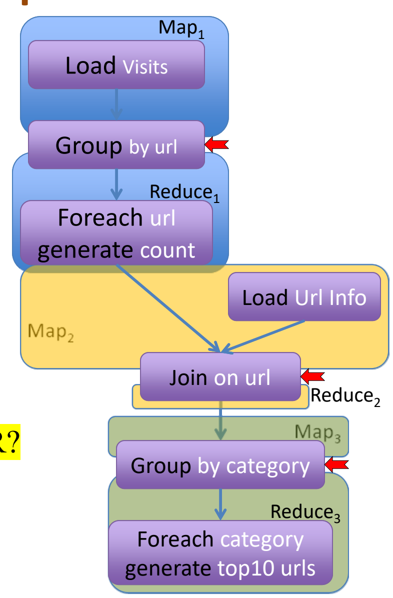



```
visits = load '/data/visits' as (user, url, time);
gVisits = group visits by url;
visitCounts = foreach gVisits
              generate url, count(visits);
urlInfo = load '/data/urlInfo' as
                  (url, category, pRank);
visitCounts = join visitCounts by url,
                     urlInfo by url;
gCategories = group visitCounts by category;
topUrls = foreach gCategories
              generate top(visitCounts,10);
store topUrls into '/data/topUrls';
```

* top, count are user defined functions

Pig -- Example

- Compile pig into map reduce
  - Each group and join forms a map-reduce boundary
    - Has to be done by both map and reduce phases
  - Other operations pipelined into map-reduce phases
    - E.g., foreach can be done in map or in reduce
  - What is the key for each M to R?



Hadoop Variants So Far

- Hive, Pig
  - Provide a high level language ⇒ Easier to program
  - Provide tools to transform the high level code to map reduce code
  - These systems are built on top of Hadoop ⇒ Still have the same problems as in Hadoop
- Haloop, Hadoop++
  - Manipulate data placement
  - Still on top of Hadoop
- How about completely change Hadoop
  - Spark

Spark

- Developed
  - By Matei Zaharia at Berkely as his PhD work
  - Then open sourced by Apache
- Major concept
  - RDD: Resilient Distributed Dataset
    - In Hadoop, datasets are stored as files on disk (HDFS)
      - Input/output datasets for a job have to be disk files
      - Data used crossed jobs are subsequently disk files
    - In Spark, each dataset is represented as an RDD
      - A Spark operation takes input RDDs and create output RDDs
    - RDD is in memory, distributed over multiple nodes
      - Persistent RDDs also have corresponding HDFS copies on disk
        - Asynchronously written to disk

Spark RDD

- Persist RDD
  - Call “persist” or “cache” methods to mark the RDD
    - By default, the RDD will be removed after the workflow
    - Can also call “unpersist” method to remove it
  - The persist RDD will be retained in memory
    - ⇒ Allows future operations to reuse the RDD
      ⇒ Much faster than disk accesses
      ⇒ Even if no loop support ⇒ No performance problem
    - If memory is abundant, 30-100 folds of speedup
  - Can choose different storage levels
    - (MEMORY_ONLY, DISK_ONLY, MEMORY_AND_DISK, etc.)
    - MEMORY_AND_DISK
      - Move some RDD partitions to disk when memory is full
      - Use LRU policy

Spark RDD

- RDD serialization
  - Hadoop persistent objects are writable
    - Has de/serialization costs
  - RDD avoids serialization (in memory)
    - Can be serialized as desired
    - Spark supports both java and kryo serialization (kryo does not support all data types, but is much faster)
- No replication
  - Spark gives up RDD replication
    - RDD is immutable once created, read only
    - RDD uses logging to achieve resilience

Spark RDD

- RDD resilience
  - No replication in RDD
    - Replication is not scalable in both computation performance and storage requirement (persistent RDDs are stored in HDFS)
  - Lineage: log the operations performed on RDDs
    - Represented by DAG
      - Nodes in DAG: RDDs;   Edges in DAG: Spark operations
  - Data recovery: RDD + lineage
    - A lost partition of an RDD can be recomputed automatically
    - Original RDD would have been an HDFS file
    - Operations has to be coarse grained to avoid logging overhead
  - Checkpoint RDDs
    - Prevent from having long lineage chains

Spark Operations

- Map-Reduce operations
  - map (func)
  - reduce(func)
  - collect()
    - Similar to reduce, with one reducer
    - Returns all the elements of the dataset at the driver (master)
  - count()
    - Not available in Hadoop
    - Returns the number of elements in the dataset

Spark Operations

- Add some DB operations
  - join(another-dataset)
  - groupby(column)
  - filter(func)
    - Similar to select (new dataset is a subset of source dataset)
    - func defines the selection criteria (func returns true/false)
- Add some Set operations
  - union(another-dataset)
  - intersection(another-dataset)
  - distinct()
    - Returns a new dataset that contains the distinct elements of the source dataset (remove duplicates)

Spark Operations

- Task
  - Unit operation
- Stage:
  - A sequence of tasks that does not have shuffle and exchange in between (e.g., map, filter, union, intersection)
  - Does not enforce synchronous operations
    - Workers can run asynchronously for operations in a stage
  - Otherwise, same as Hadoop, synchronous execution

Spark Operations

- In Hadoop

  N = #HDFS blocks for the dataset; M = #keys from mapper
  - N mappers, M reducers, C cores
  - Output of each mapper is sorted and divided into M objects
  - There are N*M objects to be shuffled
- In Spark
  - Initially, the same as Hadoop
  - Later: Output of mappers on the same core are merged
    - There are only C*M objects to be shuffled
      - This optimization is hardware dependent, but does not burden the user (since shuffle phase is anyway hidden from the user)
    - Note: Mini combiner in Hadoop is different, only merge data entries generated from the same mapper

Spark Data Placement

- RDD partitioning
  - Default: by hashing (HashPartitioner)
  - User can define data partitioning and co-location
- Partitioner object
  - Can specify the partition rules
    - Similar to Hadoop++
  - Each RDD can specify an optional Partitioner object
    - Two RDDs share the same Partitioner are co-located
    - Similar to locator in CoHadoop
    - Partitioner does not have to have partition rules, just for co-location

Spark

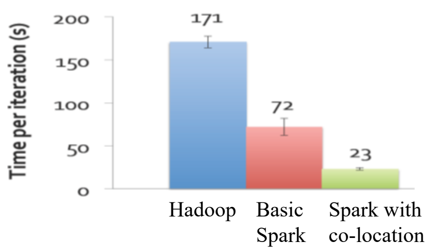
- Cluster architecture is the same as Hadoop
  - Driver = Master in Hadoop
    - Defines and invokes actions on RDDs
    - Tracks the RDDs' lineage
  - Workers
    - Store RDD partitions
    - Perform RDD transformations

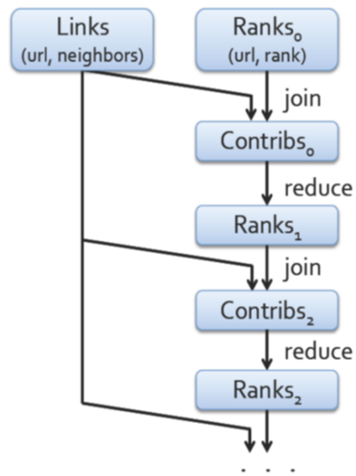
Spark

- Difference from Hadoop
  - Storage: Spark has in-memory store
    - Spark uses in-memory persistent storage
    - Also have copies on HDFS for persistent RDDs
  - No RDD replication
    - Resilience is via lineage logging
  - Computation: Spark has a more flexible model
    - Support loop, and can have any M-R combinations
    - Provide more operators, rather than just map and reduce
  - Support co-location and placement specification
  - Overall
    - ⇒ Fix many problems in Hadoop
    - ⇒ Offer the RDD solution ⇒ Unique in MapReduce evolution

Spark Performance

- Page *rank* example
  - Links: RDD indexed by url
    - Value: outgoing links
  - Ranks: RDD indexed by url
    - Value: page ranking
  - Computation requires repeated join of Links+Ranks





References

- MapReduce
  - MapReduce: Simplified data processing on large clusters
  - Parallel data processing with MapReduce: A survey
- Improvements
  - HaLoop: Efficient iterative data processing on large clusters
  - CoHadoop: Flexible Data Placement and Its Exploitation in Hadoop
  - Building a highlevel dataflow system on top of MapReduce: The Pig experience
  - Hive: A warehousing solution over a MapReduce framework
- Spark
  - Spark: Cluster computing with working sets
  - Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing