COMP 9313 REVIEW

Nov 9th 2018: 12:00~4:30

- ❑ Topic 1: MapReduce (Chapters 2-4)
- ❑ Topic 2: Spark Core and GraphX (Chapters 6 and 7)

Nov 11th 2018: 5:00~9:30

- ❑ Topic 3: Mining Data Streams (Chapter 8)
- ❑ Topic 4: Finding Similar Items (Chapter 9)
- ❑ Topic 5: Recommender Systems (Chapter 11)
- ❑ Summary + Outlook

Exam Questions

- Question 1 MapReduce
  - Part A: MapReduce concepts
  - Part B: MapReduce algorithm design
- Question 2 Spark
  - Part A: Spark concepts
  - Part B: Show output of the given code
  - Part C: Spark algorithm design
    - Spark Core
    - Spark GraphX
- Question 3 Finding Similar Items
  - Shingling, Min Hashing, LSH
- Question 4 Mining Data Streams
  - Sampling, DGIM, Bloom filter
- Question 5 Recommender Systems

COMP 9313 REVIEW

20181109

Outline

- Topic 1: MapReduce (Chapters 2-4)
  - Part A: MapReduce concepts
  - Part B: MapReduce algorithm design
- Topic 2: Spark Core and GraphX (Chapters 6 and 7)
  - Part A: Spark concepts
  - Part B: Show output of the given code
  - Part C: Spark algorithm design
    - Spark Core
    - Spark GraphX

Topic 1: MapReduce

- ***Part A: MapReduce concepts***
- Part B: MapReduce algorithm design

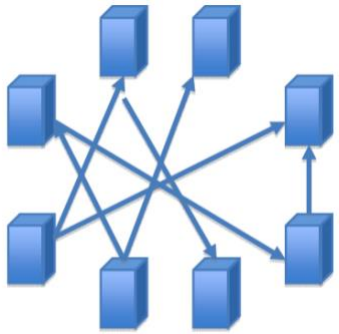
Map and Reduce Functions

- Programmers specify two functions:
  - **map** (k_1, v_1) → list [<k_2, v_2>]
    - Map transforms the input into key-value pairs to process
  - **reduce** (k_2, [v_2]) → [<k_3, v_3>]
    - Reduce aggregates the list of values for each key
    - All values with the same key are sent to the same reducer
- Optionally, also:
  - combine (k_2, [v_2]) → [<k_3, v_3>]
    - Mini-reducers that run in memory after the map phase
    - Used as an optimization to reduce network traffic
  - partition (k_2, number of partitions) → partition for k_2
    - Often a simple hash of the key, e.g., hash(k_2) mod n
    - Divides up key space for parallel reduce operations
  - Grouping comparator: controls which keys are grouped together for a single call to Reducer.reduce() function
- The execution framework handles everything else…

Understanding MapReduce

- Map>>
  - (K1, V1) →
    - Info in
    - Input Split
  - list (K2, V2)
    - Key / Value out (intermediate values)
    - One list per local node
    - Can implement local Reducer (or Combiner)

Shuffle/Sort>>



Reduce

(K2, list(V2)) →

- Shuffle / Sort phase precedes Reduce phase
- Combines Map output into a list

list (K3, V3)

- Usually aggregates intermediate values

(input) <k1, v1> → **map** → <k2, v2> → **combine** → <k2, list(V2)> → **reduce** → <k3, v3> *(output)*

WordCount - Mapper

Let’s count number of each word in documents (e.g., Tweets/Blogs)

- Reads input pair <k1,v1>
  - The input to the mapper is in format of <docID, docText>:

    <D1,“Hello World” >,<D2,“Hello Hadoop Bye Hadoop” >
- Outputs pairs <k2, v2>
  - The output of the mapper is in format of <term, 1>:

<Hello, 1><World, 1><Hello, 1><Hadoop, 1><Bye, 1><Hadoop, 1>

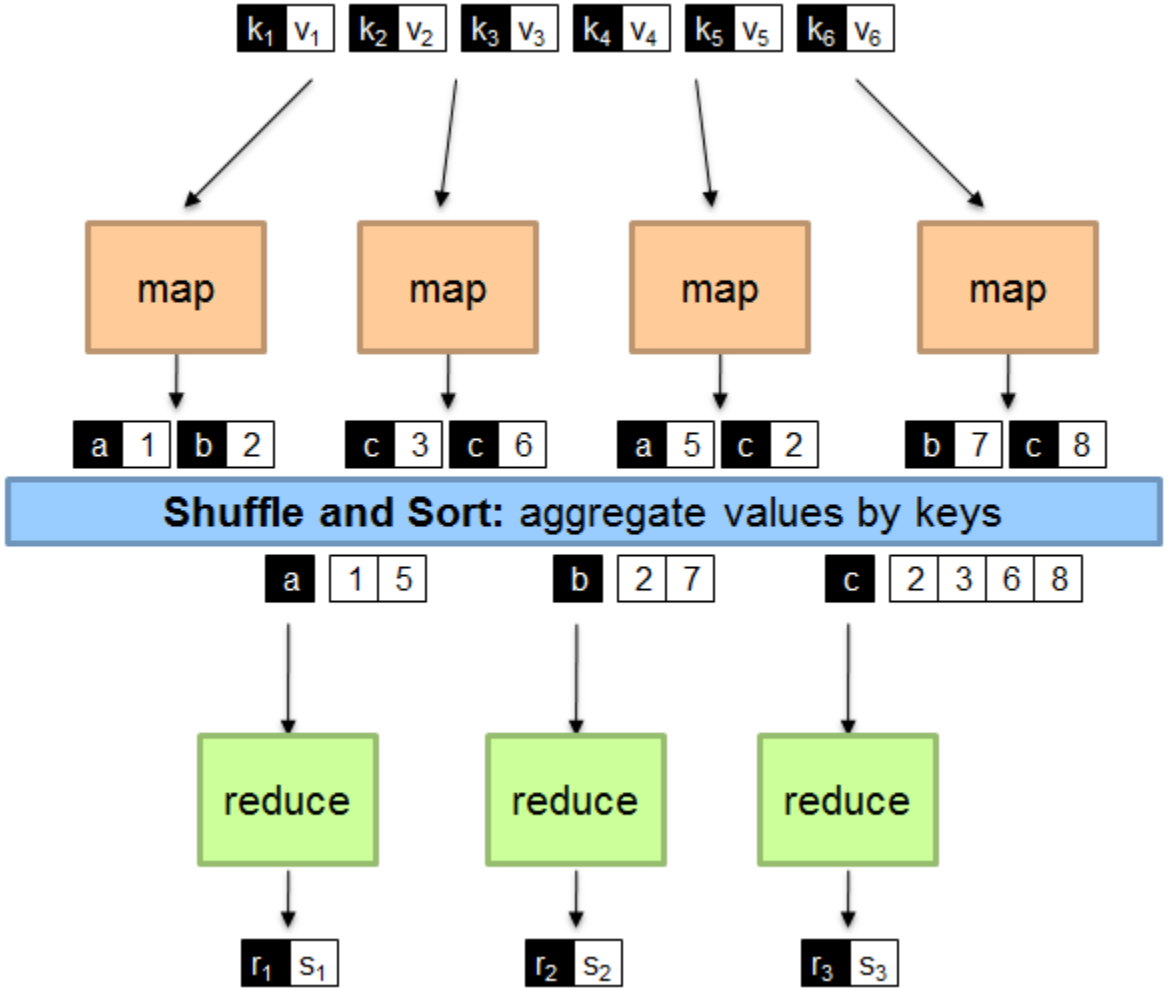
- After **shuffling and sort**, reducer receives <k2, list(v2)>

<Hello, {1, 1}><World, {1}><Hadoop, {1, 1}><Bye, {1}>

- The output is in format of <k3, v3>:

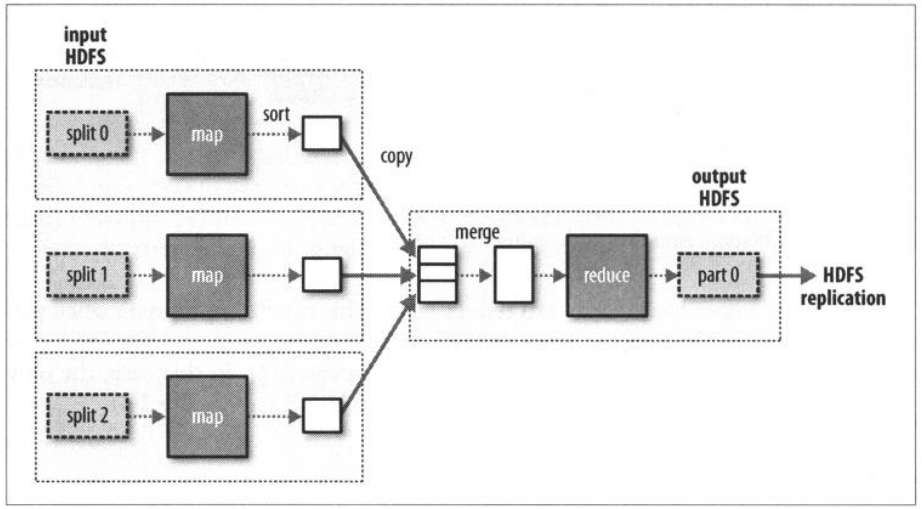
<Hello, 2><World, 1><Hadoop, 2><Bye, 1>

A Brief View of MapReduce



Shuffle and Sort

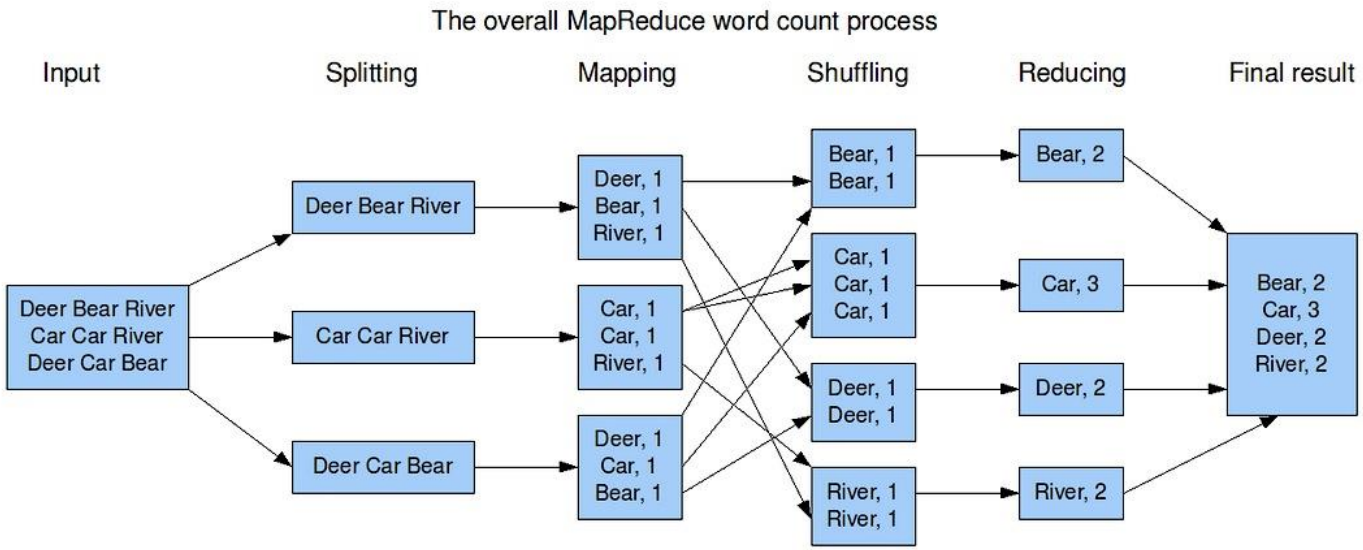
- Shuffle
  - Input to the Reducer is the sorted output of the mappers. In this phase the framework fetches the relevant partition of the output of all the mappers, via **HTTP**.
- Sort
  - The framework groups Reducer inputs by keys (since different Mappers may have output the same key) in this stage.
- Hadoop framework handles the Shuffle and Sort step .



“Hello World” in MapReduce

```
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 s)
```



Combiners

- Often a Map task will produce many pairs of the form $(k,v_1), (k,v_2), \ldots$ for the same key *k*
  - E.g., popular words in the word count example
- **Combiners** are a general mechanism to *reduce the amount of intermediate data, thus saving network time*
  - They could be thought of as "mini-reducers"
- Warning!
  - The use of combiners must be thought carefully
    - Optional in Hadoop: the correctness of the algorithm cannot depend on computation (or even execution) of the combiners
    - A combiner operates on each map output key. It must have the same output key-value types as the Mapper class.
    - A combiner can produce summary information from a large dataset because it replaces the original Map output
  - **Works only if reduce function is commutative and associative**
    - In general, reducer and combiner are not interchangeable

Combiner Design

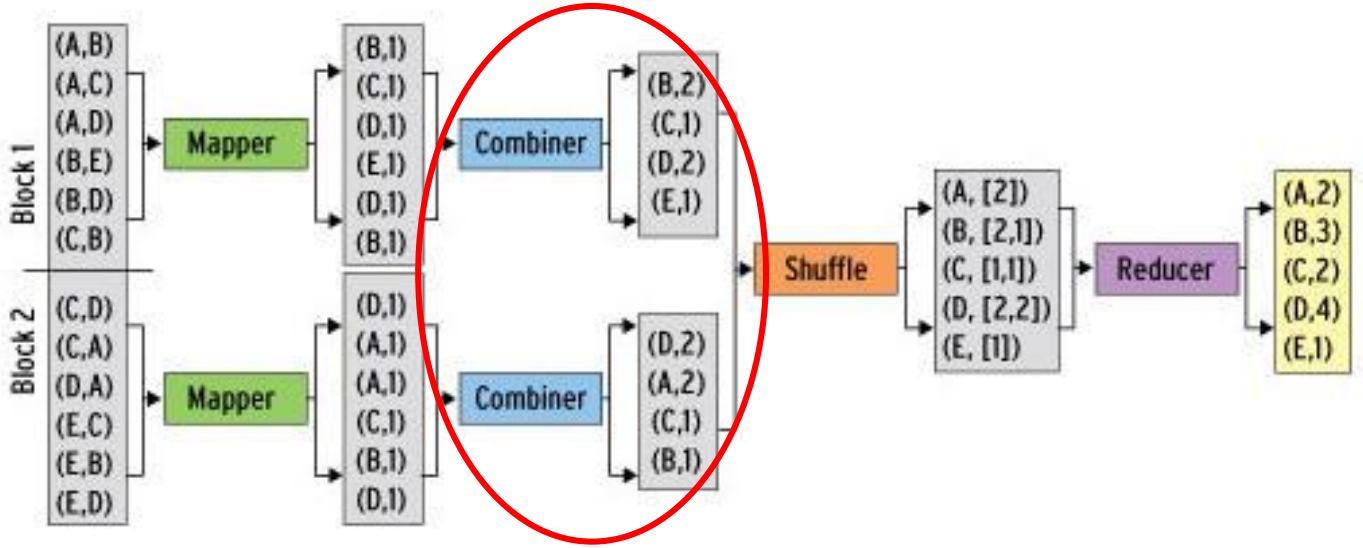
- Both input and output data types must be consistent with the output of mapper (or input of reducer)
- Combiners and reducers share same method signature
  - Sometimes, reducers can serve as combiners
  - Often, not…
- Hadoop do not guarantee how many times it will call combiner function for a particular map output record
  - It is just optimization
  - The number of calling (even zero) does not affect the output of Reducers

```
max(0, 20, 10, 25, 15) = max(max(0, 20, 10), max(25, 15)) = max(20, 25) = 25
```

- **Applicable on problems that are commutative and associative**
  - Commutative: max(a, b) = max(b, a)
  - Associative: max (max(a, b), c) = max(a, max(b, c))

Combiners in WordCount

Combiner combines the values of all keys of a single mapper node (single machine):



Much less data needs to be copied and shuffled!

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: not all mappers will see all terms

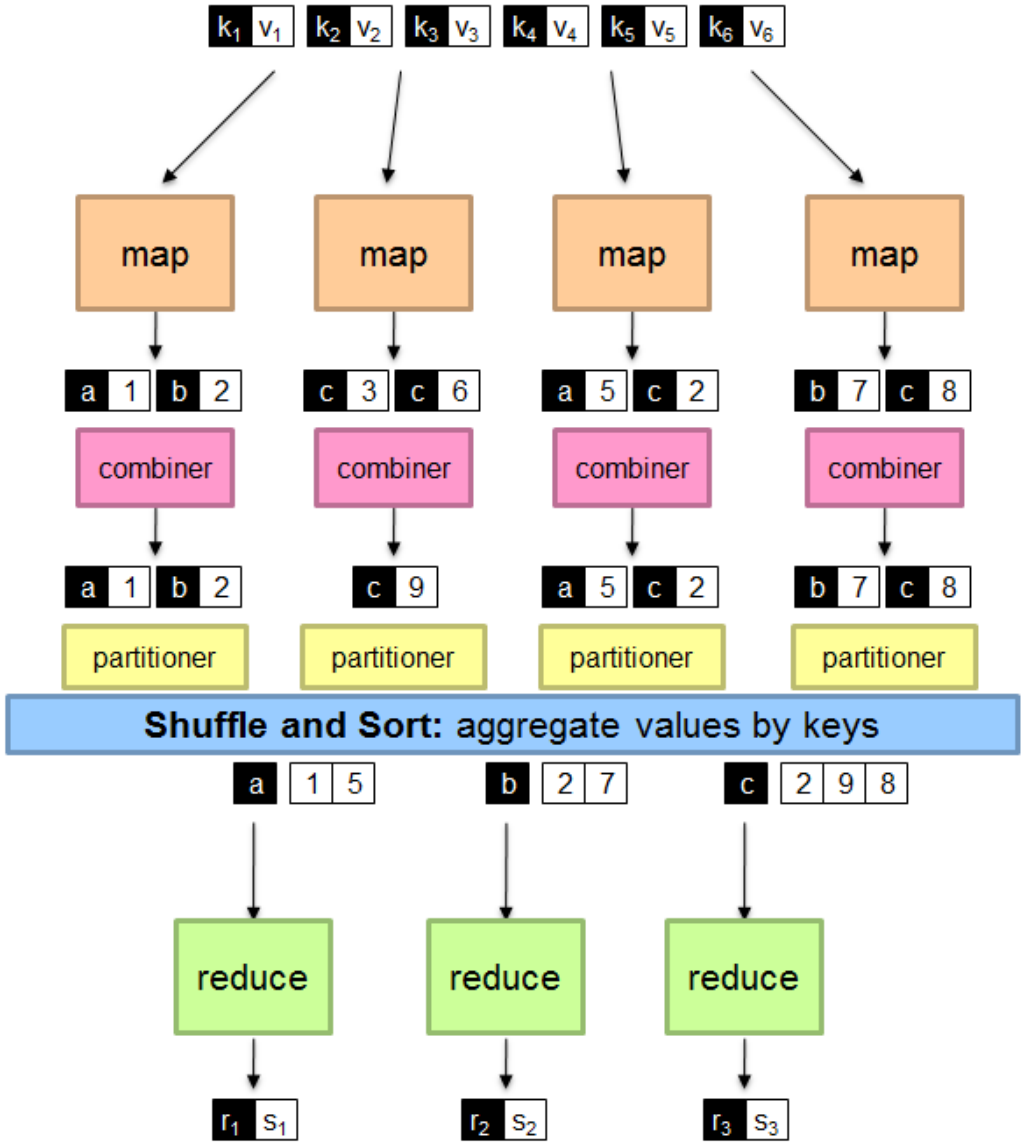
Partitioner

Partitioner controls the partitioning of the keys of the intermediate map-outputs.

- The key (or a subset of the key) is used to derive the partition, typically by a ***hash function***.
- ***The total number of partitions is the same as the number of reduce tasks for the job***.
  - This controls which of the m reduce tasks the intermediate key (and hence the record) is sent to for reduction.

- System uses **HashPartitioner** by default:
  - hash(key) mod R
- Sometimes useful to override the hash function:
  - E.g., ***hash(hostname(URL)) mod R*** ensures URLs from a host end up in the same output file
- Job sets Partitioner implementation (in Main)

A Brief View of MapReduce



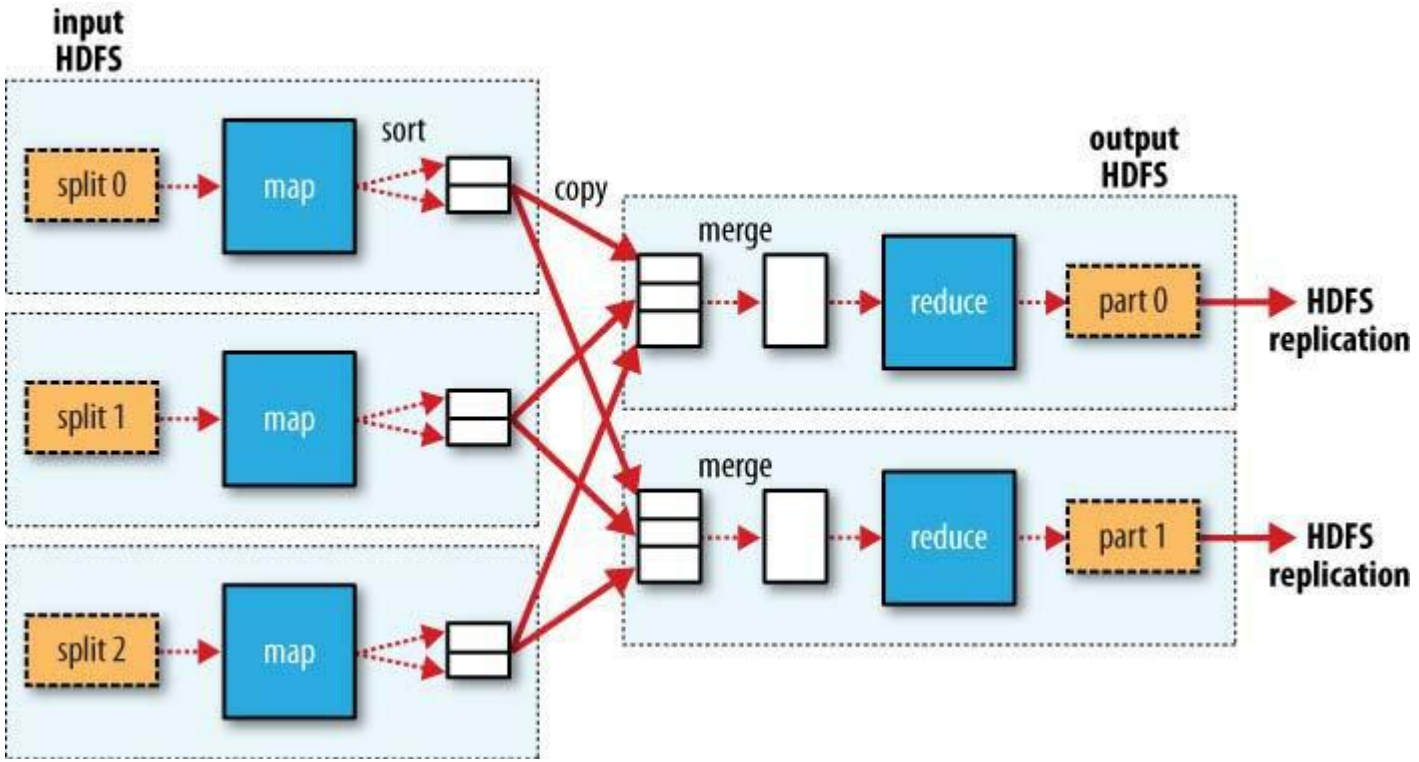
More Detailed MapReduce Dataflow?

When there are multiple reducers, the map tasks partition their output:

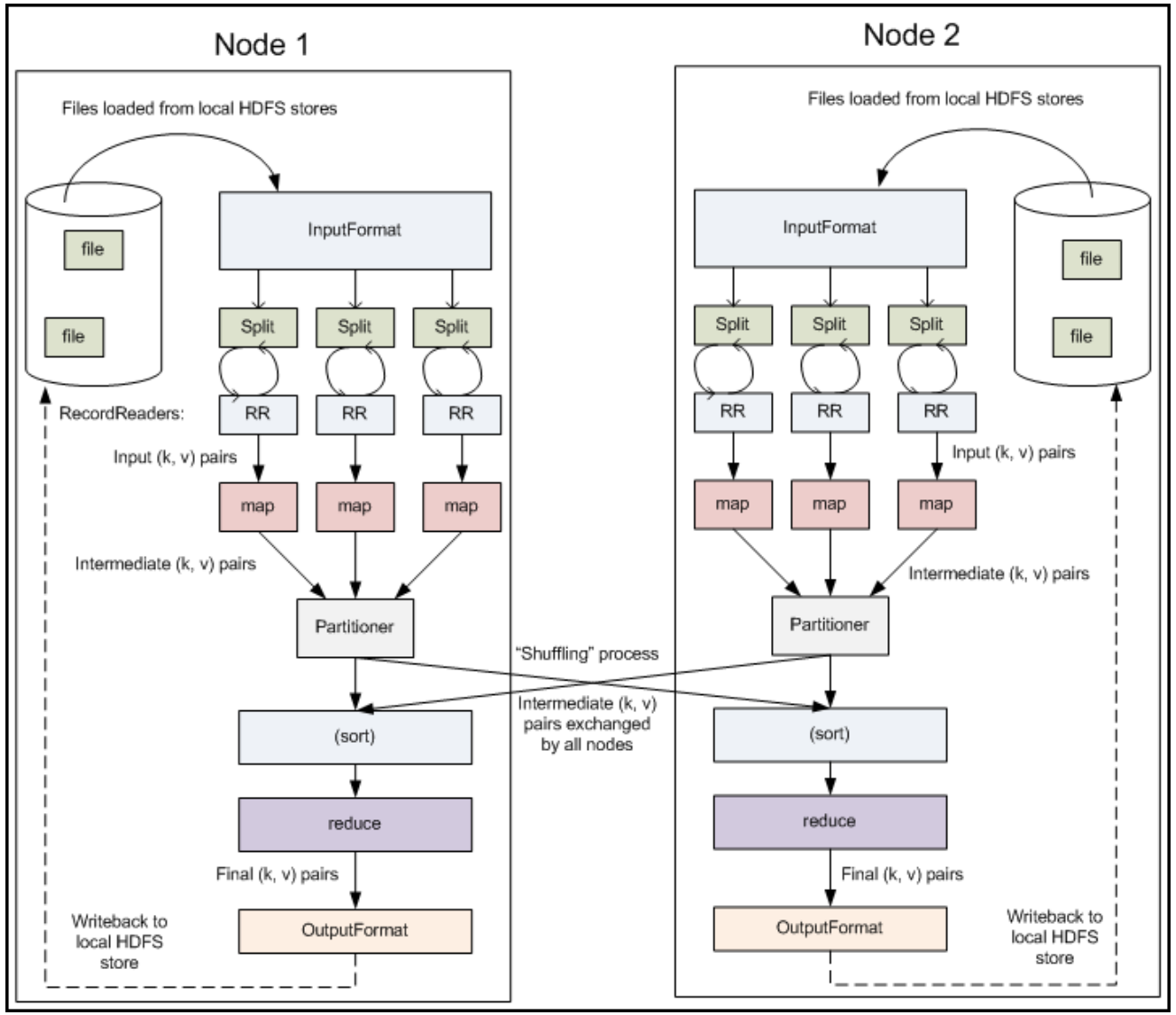
One partition for each reduce task

The records for every key are all in a single partition

Partitioning can be controlled by a user-defined partitioning function



MapReduce Data Flow



Serialization

- Process of turning structured objects into a byte stream for transmission over a network or for writing to persistent storage
- Deserialization is the reverse process of serialization
- Requirements
  - Compact
    - To make efficient use of storage space
  - Fast
    - The overhead in reading and writing of data is minimal
  - Extensible
    - We can transparently read data written in an older format
  - Interoperable
    - We can read or write persistent data using different language

Issues with MapReduce on Graph Processing

- MapReduce Does not support iterative graph computations:
  - External driver. Huge I/O incurs
  - No mechanism to support global data structures that can be accessed and updated by all mappers and reducers
    - Passing information is only possible within the local graph structure – through adjacency list
    - Dijkstra's algorithm on a single machine: a global priority queue that guides the expansion of nodes
    - Dijkstra‘s algorithm in Hadoop, no such queue available. Do some “wasted” computation instead
- MapReduce algorithms are often impractical on large, dense graphs.
  - The amount of intermediate data generated is on the order of the number of edges.
  - For dense graphs, MapReduce running time would be dominated by copying intermediate data across the network.

Number of Maps and Reduces

- Maps
  - The number of maps is usually driven by the total size of the inputs, that is, **the total number of blocks of the input files**.
  - The right level of parallelism for maps seems to be around 10-100 maps per-node, although it has been set up to 300 maps for very cpu-light map tasks.
  - If you expect 10TB of input data and have a blocksize of 128MB, you'll end up with 82,000 maps, unless Configuration.set(MRJobConfig.NUM_MAPS, int) (which only provides a hint to the framework) is used to set it even higher.

- Reduces
  - The right number of reduces seems to be 0.95 or 1.75 multiplied by (<*no. of nodes*> * <*no. of maximum containers per node*>)
  - With 0.95 all of the reduces can launch immediately and start transferring map outputs as the maps finish. With 1.75 the faster nodes will finish their first round of reduces and launch a second wave of reduces doing a much better job of load balancing.
  - Use job.setNumReduceTasks(int) to set the number

Topic 1: MapReduce

- Part A: MapReduce concepts
- ***Part B: MapReduce algorithm design***

MapReduce Algorithm Design Patterns

- **“In-mapper combining”**, where the functionality of the combiner is moved into the mapper.
  - Scalability issue (not suitable for huge data) : More memory required for a mapper to store intermediate results
- The related patterns **“pairs” and “stripes”** for keeping track of joint events from a large number of observations.
- **“Order inversion”**, where the main idea is to convert the sequencing of computations into a sorting problem.
  - You need to guarantee that all key-value pairs relevant to the same term are sent to the same reducer
- **“Value-to-key conversion”**, which provides a scalable solution for secondary sorting.
  - Grouping comparator

Design Pattern 1: In-mapper Combining

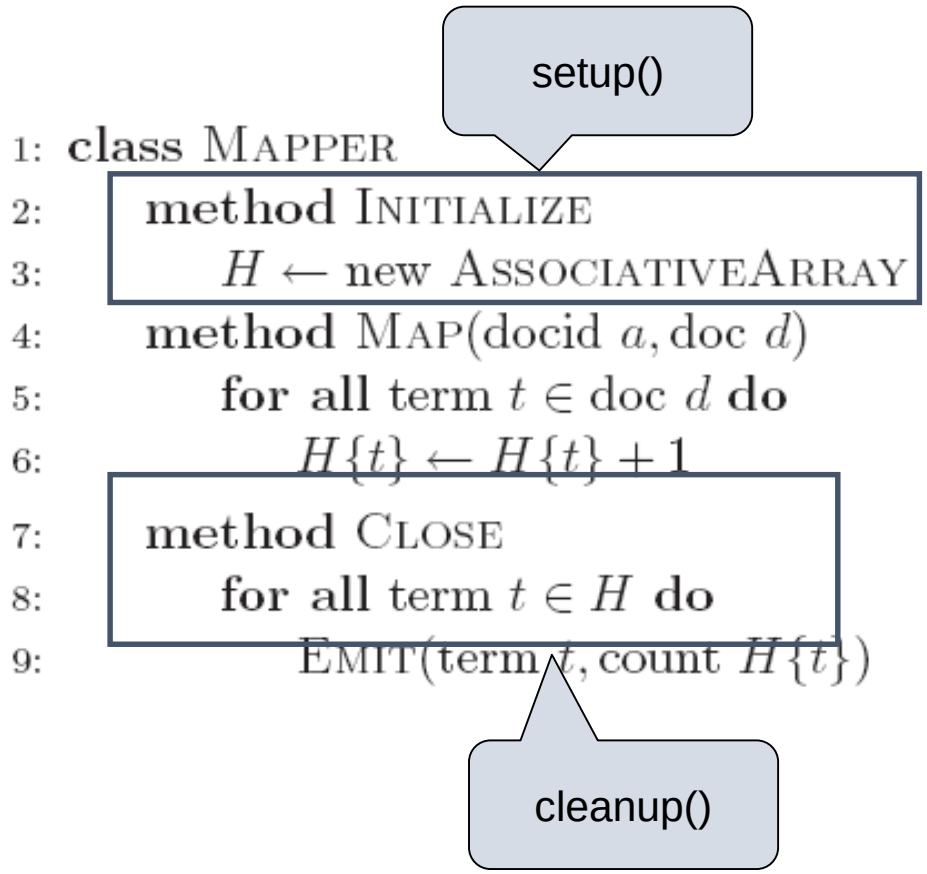
Design Pattern for Local Aggregation

- “In-mapper combining”
  - Fold the functionality of the combiner into the mapper by preserving state across multiple map calls
- Advantages
  - Speed
  - Why is this faster than actual combiners?
- Disadvantages
  - Explicit memory management required
  - Potential for order-dependent bugs

Lifecycle of Mapper/Reducer

- Lifecycle: setup -> map -> cleanup
  - setup(): called once at the beginning of the task
  - map(): do the map
  - cleanup(): called once at the end of the task.
  - We do not invoke these functions
- In-mapper Combining:
  - Use **setup()** to initialize the state preserving data structure
  - Use **clearnup()** to emit the final key-value pairs

Word Count: Version 2



▷ Tally counts *across* documents

Design Pattern 2: Pairs vs Stripes

First Try: "Pairs"

- Each mapper takes a sentence
  - Generate all co-occurring term pairs
  - For all pairs, emit (a, b) → count
- Reducers sum up counts associated with these pairs
- Use combiners!

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

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

“Pairs” Analysis

- Advantages
  - Easy to implement, easy to understand

- Disadvantages
  - Lots of pairs to sort and shuffle around
  - Not many opportunities for combiners to work

Another Try: "Stripes"

- Idea: group together pairs into an associative array

(a, b) → 1
(a, c) → 2
(a, d) → 5
(a, e) → 3
(a, f) → 2

a → { b: 1, c: 2, d: 5, e: 3, f: 2 }

- Each mapper takes a sentence:
  - Generate all co-occurring term pairs
  - For each term, emit a → { b: $count_b$, c: $count_c$, d: $count_d$ … }
- Reducers perform element-wise sum of associative arrays

```
    a → { b: 1,         d: 5, e: 3 }
+   a → { b: 1, c: 2, d: 2,         f: 2 }
    a → { b: 2, c: 2, d: 7, e: 3, f: 2 }
```

Key: cleverly-constructed data structure brings together partial results

“Stripes” Analysis

- Advantages
  - Far less sorting and shuffling of key-value pairs
  - Can make better use of combiners

- Disadvantages
  - More difficult to implement
  - Underlying object more heavyweight
  - Fundamental limitation in terms of size of event space

Design Pattern 3: Order Inversion

$f(w_j|w_i)$: "Pairs"

- Better solutions?

(a, *) → 32

Reducer holds this value in memory, rather than the stripe

$(a, b_1) \rightarrow 3$
$(a, b_2) \rightarrow 12$
$(a, b_3) \rightarrow 7$
$(a, b_4) \rightarrow 1$
…

$(a, b_1) \rightarrow 3 / 32$
$(a, b_2) \rightarrow 12 / 32$
$(a, b_3) \rightarrow 7 / 32$
$(a, b_4) \rightarrow 1 / 32$
…

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

$f(w_j|w_i)$: "Pairs" – Order Inversion

- A better solution based on order inversion

- 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 (define sort order)
  - We also need to guarantee that all pairs associated with the same word are sent to the same reducer (use partitioner)

$f(w_j|w_i)$: "Pairs" – Order Inversion

- Example:
  - The reducer finally receives:

key	**values**	
(dog, ∗)	[6327, 8514, . . .]	compute marginal: $\sum_{w'} N(\text{dog}, w') = 42908$
(dog, aardvark)	[2,1]	$f(\text{aardvark}\|\text{dog}) = 3/42908$
(dog, aardwolf)	[1]	$f(\text{aardwolf}\|\text{dog}) = 1/42908$
. . .		
(dog, zebra)	[2,1,1,1]	$f(\text{zebra}\|\text{dog}) = 5/42908$
(doge, ∗)	[682, . . .]	compute marginal: $\sum_{w'} N(\text{doge}, w') = 1267$
. . .		

  - The pairs come in order, and thus we can compute the relative frequency immediately.

Design Pattern 4: Value-to-key Conversion

Secondary Sort

- MapReduce sorts input to reducers by key
  - Values may be arbitrarily ordered

- What if want to sort value as well?
  - E.g., $k \rightarrow (v_1, r), (v_3, r), (v_4, r), (v_8, r)\ldots$
  - Google's MapReduce implementation provides built-in functionality
  - Unfortunately, Hadoop does not support

- Secondary Sort: sorting values associated with a key in the reduce phase, also called “value-to-key conversion”

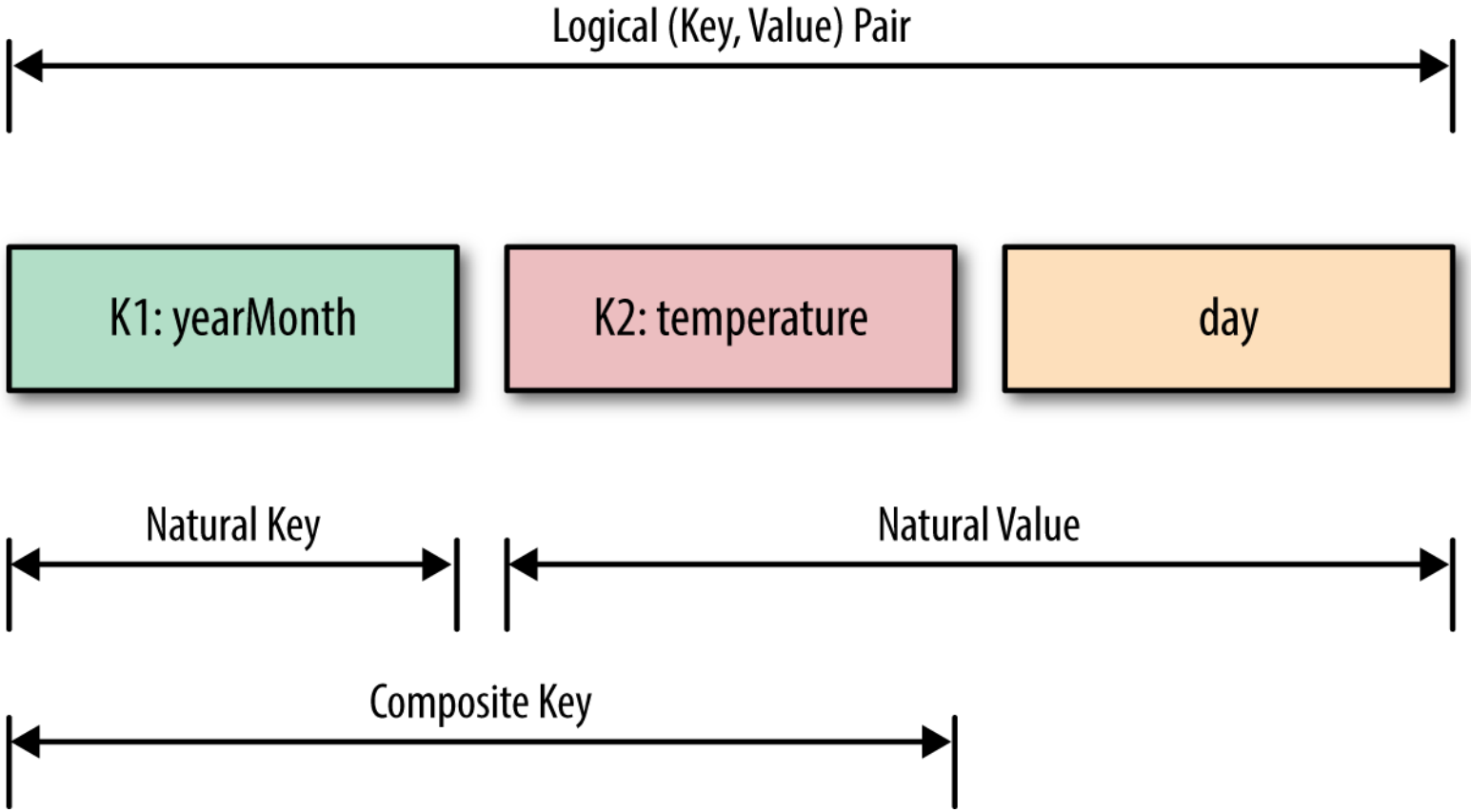
Solutions to the Secondary Sort Problem

- Use the *Value-to-Key Conversion* design pattern:
  - form a composite intermediate key, (K, V), where V is the secondary key. Here, K is called a *natural key*. To inject a value (i.e., V) into a reducer key, simply create a composite key
    - K: year-month
    - V： temperature data

Let the MapReduce execution framework do the sorting (rather than sorting in memory, let the framework sort by using the cluster nodes).

- Preserve state across multiple key-value pairs to handle processing. Write your own partitioner: partition the mapper's output by the natural key (year-month).

Secondary Sorting Keys



Application: Building Inverted Index

Boolean Text Retrieval: Inverted Index

- The inverted index of a document collection is basically a data structure that
  - attaches each distinctive term with a list of all documents that contains the term.
  - The documents containing a term are sorted in the list

- Thus, in retrieval, it takes constant time to
  - find the documents that contains a query term.
  - multiple query terms are also easy handle as we will see soon.

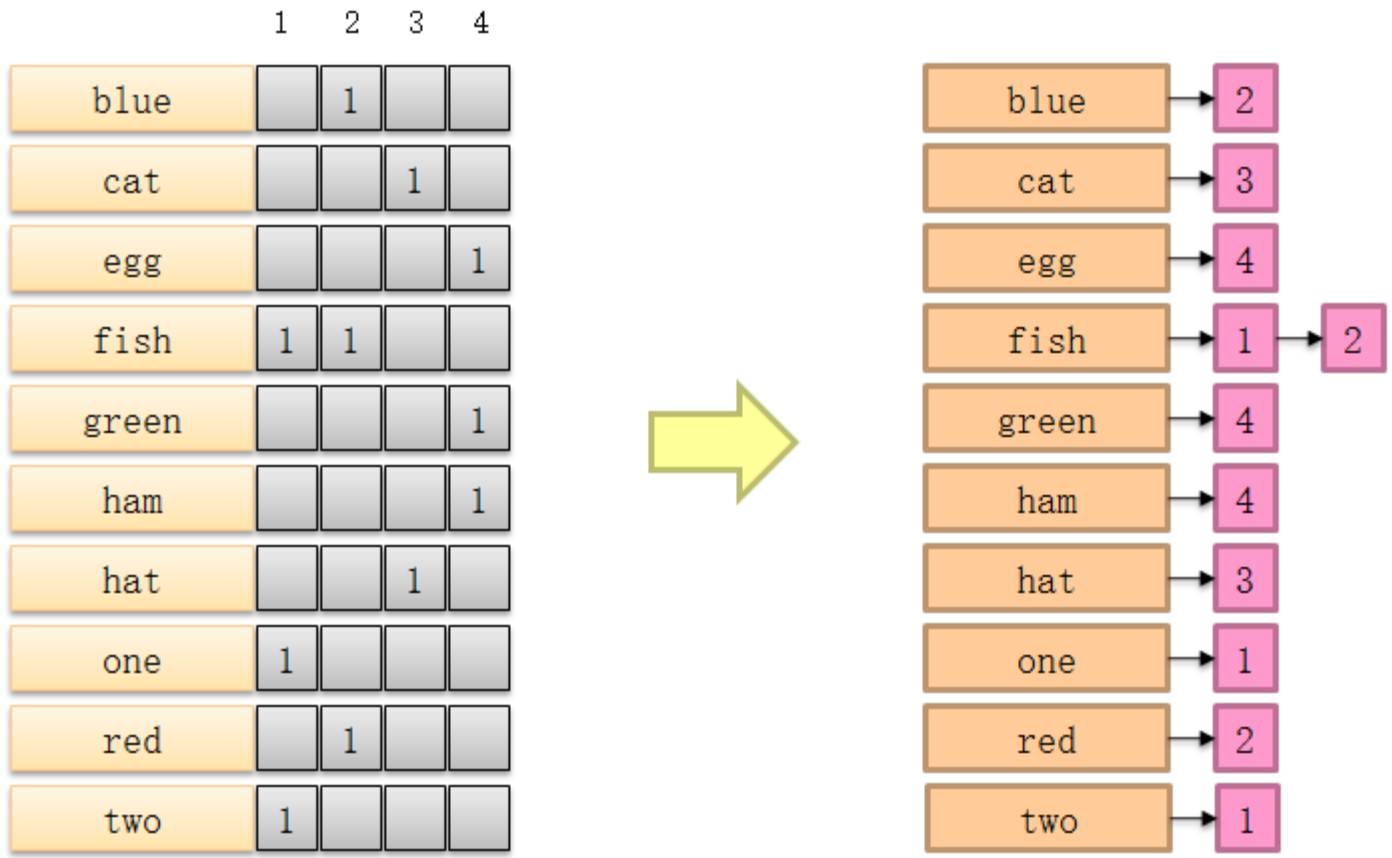
Boolean Text Retrieval: Inverted Index

Doc 1 one fish, two fish

Doc 2 red fish, blue fish

Doc 3 cat in the hat

Doc 4 green eggs and ham



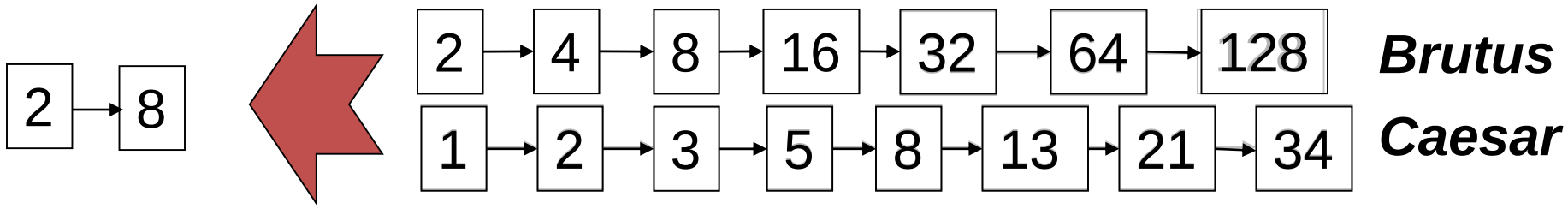
Search Using Inverted Index

- Given a query **q**, search has the following steps:
  - Step 1 (vocabulary search): find each term/word in q in the inverted index.
  - Step 2 (results merging): Merge results to find documents that contain all or some of the words/terms in q.
  - Step 3 (Rank score computation): To rank the resulting documents/pages, using:
    - content-based ranking
    - link-based ranking
    - Not used in Boolean retrieval

Boolean Query Processing: AND

Consider processing the query: ***Brutus*** *AND* ***Caesar***

- Locate ***Brutus*** in the Dictionary;
  - Retrieve its postings.
- Locate ***Caesar*** in the Dictionary;
  - Retrieve its postings.
- "Merge" the two postings:
  - Walk through the two postings simultaneously, in time linear in the total number of postings entries



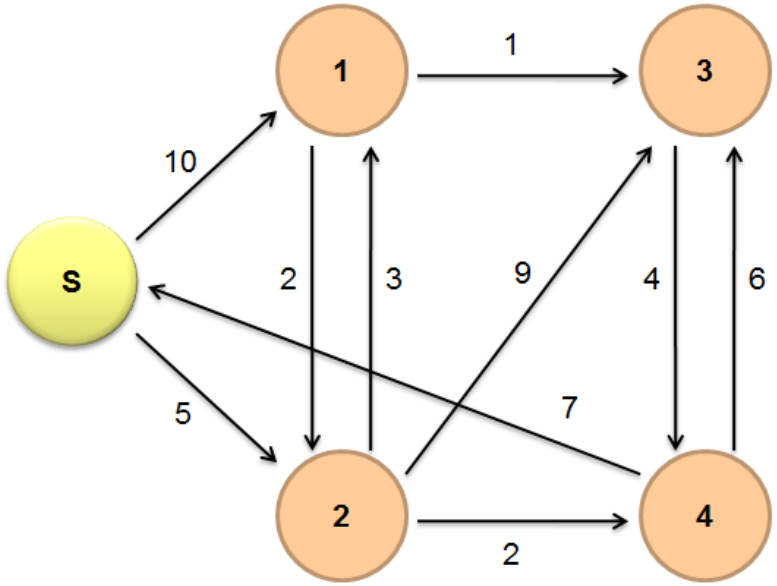
If the list lengths are x and y, the merge takes O(x+y) operations.

Crucial: postings sorted by docID.

Graph Data Processing in MapReduce

Single-Source Shortest Path (SSSP)

- **Problem:** find shortest path from a source node to one or more target nodes
  - Shortest might also mean lowest weight or cost
- Dijkstra's Algorithm:
  - For a given source node in the graph, the algorithm finds the shortest path between that node and every other



Implementation in MapReduce

- The actual checking of the termination condition must occur outside of MapReduce.
- The driver (main) checks to see if a termination condition has been met, and if not, repeats.
- Hadoop provides a lightweight API called "**counters**".
  - It can be used for counting events that occur during execution, e.g., number of corrupt records, number of times a certain condition is met, or anything that the programmer desires.
  - ***Counters can be designed to count the number of nodes that have distances of ∞ at the end of the job***, the driver program can access the final counter value and check to see if another iteration is necessary.

Example (only distances)

- Input file:

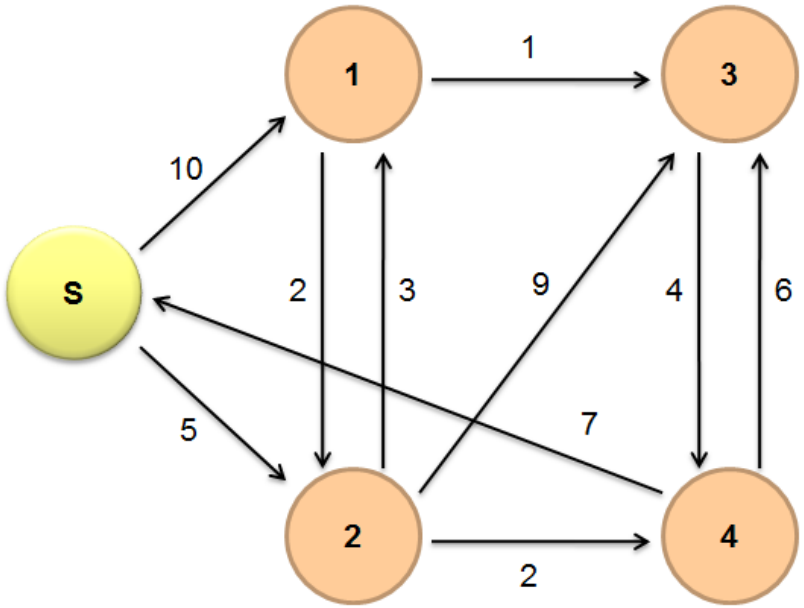
s --> 0 | n1: 10, n2: 5

n1 --> ∞ | n2: 2, n3:1

n2 --> ∞ | n1: 3, n3:9, n4:2

n3 --> ∞ | n4:4

n4 --> ∞ | s:7, n3:6



Iteration 1

- **Map:**

Read s --> 0 | n1: 10, n2: 5

Emit: (n1, 10), (n2, 5), and the adjacency list (s, n1: 10, n2: 5)

The other lists will also be read and emit, but they do not contribute, and thus ignored

- **Reduce:**

Receives: (n1, 10), (n2, 5), (s, <0, (n1: 10, n2: 5)>)

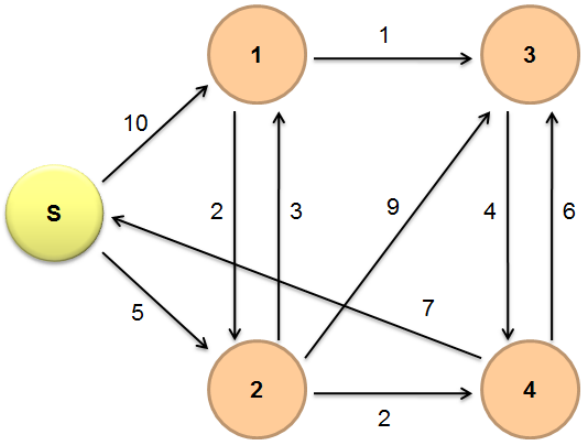
The adjacency list of each node will also be received, ignored in example

Emit:

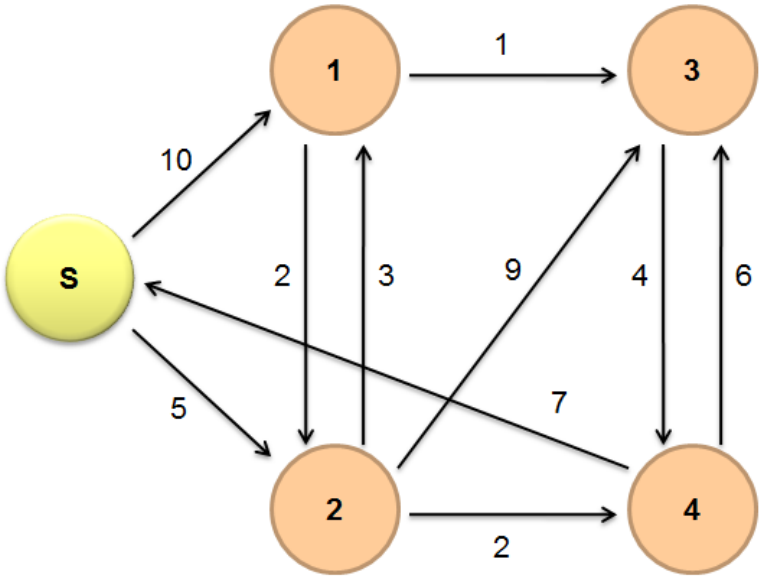
s --> 0 | n1: 10, n2: 5

n1 --> 10 | n2: 2, n3:1

n2 --> 5 | n1: 3, n3:9, n4:2



Iteration 2



- **Map:**

Read: n1 --> 10 | n2: 2, n3:1

Emit: (n2, 12), (n3, 11), (n1, <10, (n2: 2, n3:1)>)

Read: n2 --> 5 | n1: 3, n3:9, n4:2

Emit: (n1, 8), (n3, 14), (n4, 7), (n2, <5, (n1: 3, n3:9, n4:2)>)

Ignore the processing of the other lists

- **Reduce:**

Receives: (n1, (8, <10, (n2: 2, n3:1)>)), (n2, (12, <5, n1: 3, n3:9, n4:2>)), (n3, (11, 14)), (n4, 7)

Emit:

n1 --> 8 | n2: 2, n3:1

n2 --> 5 | n1: 3, n3:9, n4:2

n3 --> 11 | n4:4

n4 --> 7 | s:7, n3:6

Iteration 3

- **Map:**

Read: n1 --> 8 | n2: 2, n3:1

Emit: (n2, 10), (n3, 9), (n1, <8, (n2: 2, n3:1)>)

Read: n2 --> 5 | n1: 3, n3:9, n4:2 (**Again!**)

Emit: (n1, 8), (n3, 14), (n4, 7), (n2, <5, (n1:3, n3:9, n4:2)>)

Read: n3 --> 11 | n4:4

Emit: (n4, 15), (n3, <11, (n4:4)>)

Read: n4 --> 7 | s:7, n3:6

Emit: (s, 14), (n3, 13), (n4, <7, (s:7, n3:6)>)

- **Reduce:**

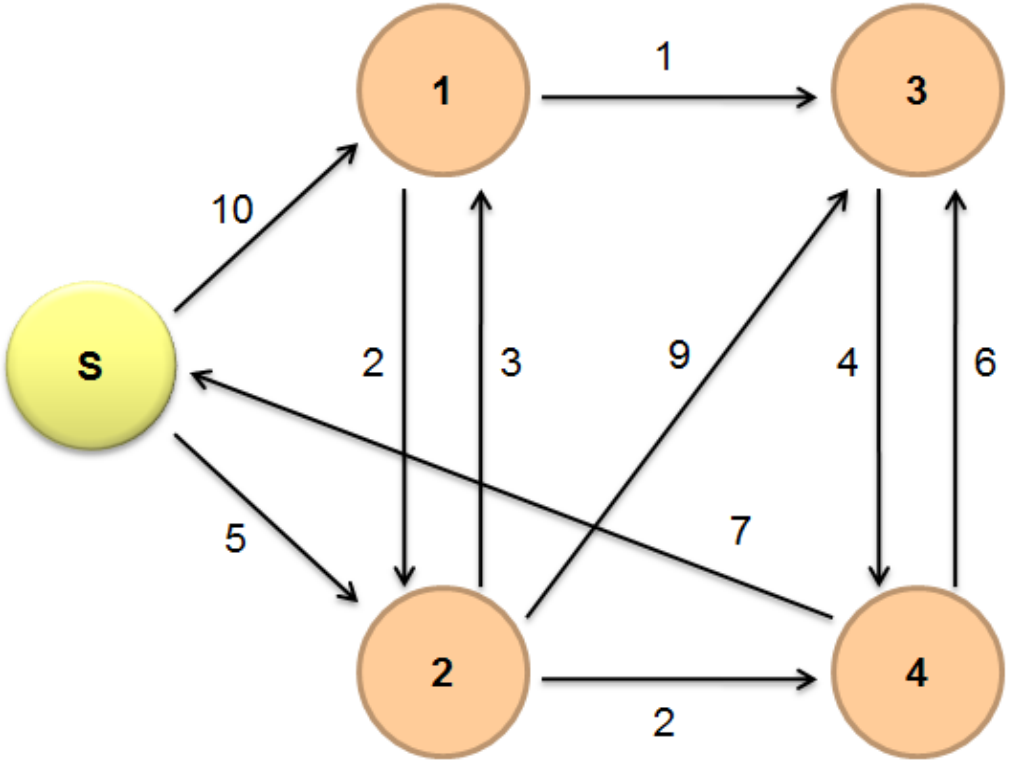
Emit:

n1 --> 8 | n2: 2, n3:1

n2 --> 5 | n1: 3, n3:9, n4:2

n3 --> 9 | n4:4

n4 --> 7 | s:7, n3:6



Iteration 4

- **Map:**

Read: n1 --> 8 | n2: 2, n3:1 (**Again!**)

Emit: (n2, 10), (n3, 9), (n1, <8, (n2: 2, n3:1)>)

Read: n2 --> 5 | n1: 3, n3:9, n4:2 (**Again!**)

Emit: (n1, 8), (n3, 14), (n4, 7), (n2, <5, (n1: 3, n3:9, n4:2)>)

Read: n3 --> 9 | n4:4

Emit: (n4, 13), (n3, <9, (n4:4)>)

Read: n4 --> 7 | s:7, n3:6 (**Again!**)

Emit: (s, 14), (n3, 13), (n4, <7, (s:7, n3:6)>)

- **Reduce:**

Emit:

n1 --> 8 | n2: 2, n3:1

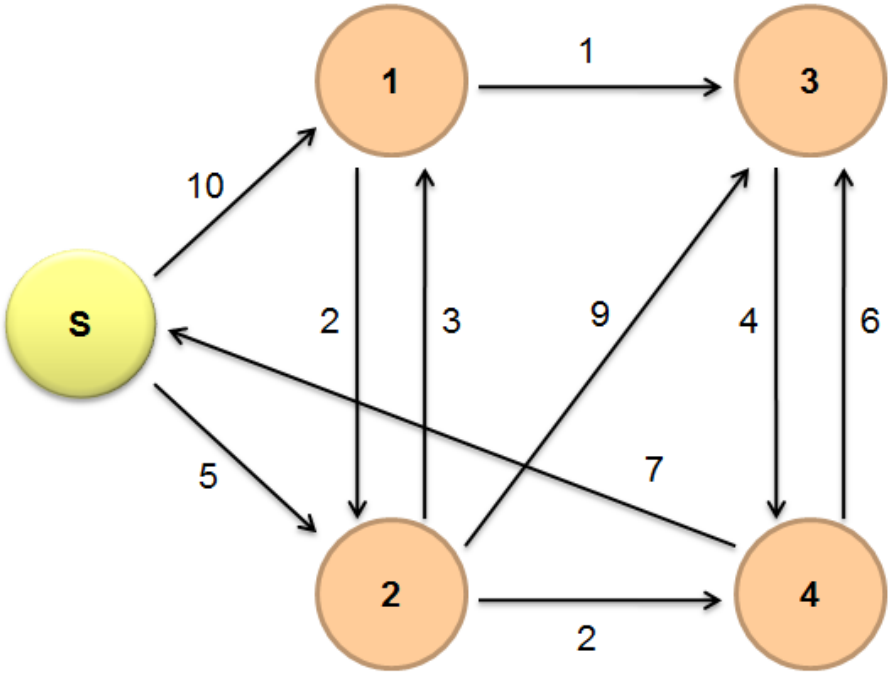
n2 --> 5 | n1: 3, n3:9, n4:2

n3 --> 9 | n4:4

n4 --> 7 | s:7, n3:6

No updates. Terminate.

In order to avoid duplicated computations, you can use a status value to indicate whether the distance of the node has been modified in the previous iteration.



Dijkstra's Algorithm

```
1:  DIJKSTRA(G, w, s)
2:      d[s] ← 0
3:      for all vertex v ∈ V do
4:          d[v] ← ∞
5:      Q ← {V}
6:      while Q ≠ ∅ do
7:          u ← EXTRACTMIN(Q)
8:          for all vertex v ∈ u.ADJACENCYLIST do
9:              if d[v] > d[u] + w(u, v) then
10:                 d[v] ← d[u] + w(u, v)
```

Practice: Design MapReduce Algorithms

- Counting total enrollments of two specified courses

- Input Files: A list of students with their enrolled courses

  Jamie: COMP9313, COMP9318

  Tom: COMP9331, COMP9313

  … …

- Mapper selects records and outputs initial counts
  - Input: Key – student, value – a list of courses
  - Output: (COMP9313, 1), (COMP9318, 1), …

- Reducer accumulates counts
  - Input: (COMP9313, [1, 1, …]), (COMP9318, [1, 1, …])
  - Output: (COMP9313, 16), (COMP9318, 35)

Practice: Design MapReduce Algorithms

- Remove duplicate records
- Input: a list of records

  2013-11-01 aa
  2013-11-02 bb
  2013-11-03 cc
  2013-11-01 aa
  2013-11-03 dd

- Mapper
  - Input (record_id, record)
  - Output (record, “”)
    - E.g., (2013-11-01 aa, “”), (2013-11-02 bb, “”), …
- Reducer
  - Input (record, [“”, “”, “”, …])
    - E.g., (2013-11-01 aa, [“”, “”]), (2013-11-02 bb, [“”]), …
  - Output (record, “”)

Practice: Design MapReduce Algorithms

- Assume that in an online shopping system, a huge log file stores the information of each transaction. Each line of the log is in format of "userID**\t** product**\t** price**\t** time". Your task is to use MapReduce to find out the top-5 expensive products purchased by each user in 2016

- Mapper:
  - Input(transaction_id, transaction)
  - initialize an associate array H(UserID, priority queue Q of log record based on price)
  - map(): get local top-5 for each user
  - cleanup(): emit the entries in H

- Reducer:
  - Input(userID, list of queues[])
  - get top-5 products from the list of queues

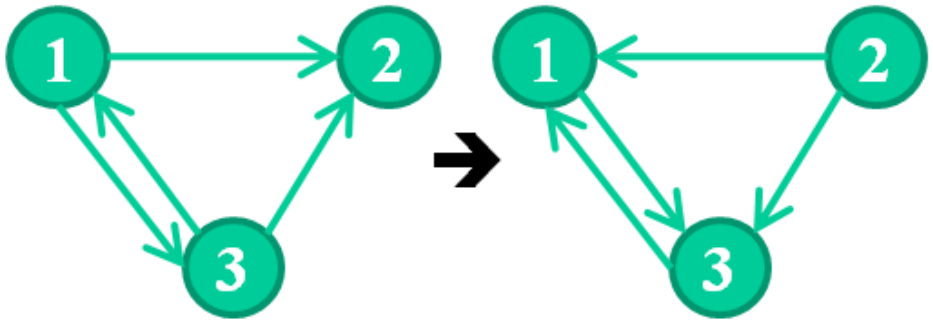
Practice: Design MapReduce Algorithms

- Reverse graph edge directions & output in node order

Input: adjacency list of graph (3 nodes and 4 edges)

(3, [1, 2])
(1, [2, 3]) ➔

(1, [3])
(2, [1, 3])
(3, [1])



- Note, the node_ids in the output values are also sorted. But Hadoop only sorts on keys!

- Solutions: Secondary sort

Practice: Design MapReduce Algorithms

- Map
  - Input: (3, [1, 2]), (1, [2, 3]).
  - Intermediate: (1, [3]), (2, [3]), (2, [1]), (3, [1]). (reverse direction)
  - Output: (<1, 3>, [3]), (<2, 3>, [3]), (<2, 1>, [1]), (<3, 1>, [1]).
    - Copy node_ids from value to key.
- Partition on Key.field1, and Sort on whole Key (both fields)
  - Input: (<1, 3>, [3]), (<2, 3>, [3]), (<2, 1>, [1]), (<3, 1>, [1])
  - Output: (<1, 3>, [3]), (<2, 1>, [1]), (<2, 3>, [3]), (<3, 1>, [1])
- Grouping comparator
  - Merge according to part of the key
  - Output: (<1, 3>, [3]), (<2, 1>, [1, 3]), (<3, 1>, [1])
    this will be the reducer's input
- Reducer
  - Merge according to part of the key
  - Output: (1, [3]), (2, [1, 3]), (3, [1])

Practice: Design MapReduce Algorithms

- Calculate the common friends for each pair of users in Facebook. Assume the friends are stored in format of Person->[List of Friends], e.g.: A -> [B C D], B -> [A C D E], C -> [A B D E], D -> [A B C E], E -> [B C D]. Your result should be like:
  - (A B) -> (C D)
  - (A C) -> (B D)
  - (A D) -> (B C)
  - (B C) -> (A D E)
  - (B D) -> (A C E)
  - (B E) -> (C D)
  - (C D) -> (A B E)
  - (C E) -> (B D)
  - (D E) -> (B C)

Practice: Design MapReduce Algorithms

- Mapper:
  - Input(user u, List of Friends [f_1, f_2, …,])
  - map(): for each friend f_i, emit (<u, f_i>, List of Friends [f_1, f_2, …,])

- Reducer:
  - Input(user u, list of friends lists[])
  - Get the intersection from all friends lists

- Example: http://stevekrenzel.com/articles/finding-friends

Question 1 MapReduce

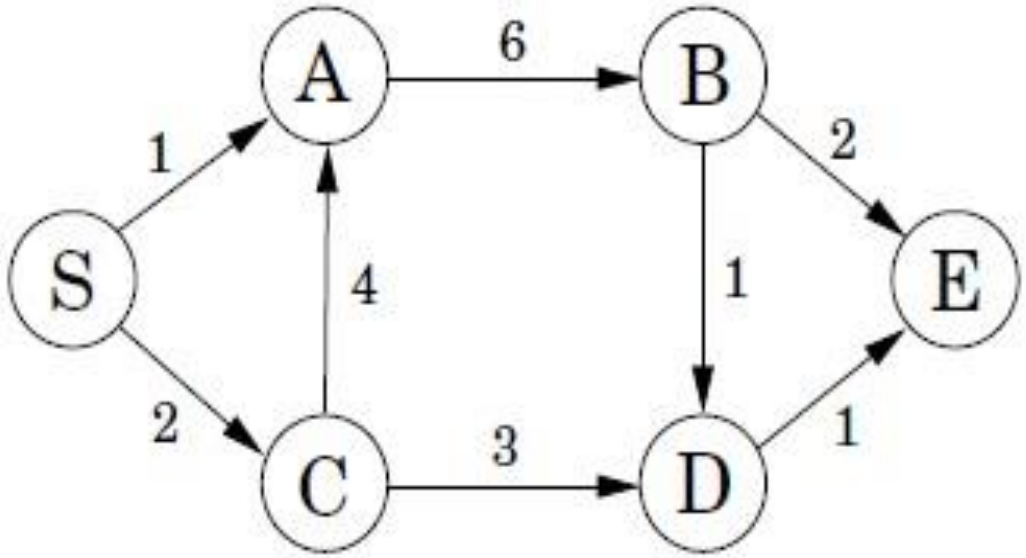
Assume that you are given a data set crawled from a location-based social network, in which each line of the data is in format of (userID, a list of locations the user has visited <loc1, loc2, …>). Your task is to compute for each location the set of users who have visited it, and the users are sorted in ascending order according to their IDs.

Solution

```
class Pair
        userID, locID
        int compareTo(Pair p)
                int ret = this.locID.compareTo(p.getLoc)
                if(ret == 0) ret = this.userID.compareTo(p.getUser)
                return ret

class Mapper
        method Map(userID, list of locations)
                foreach loc in the list of locations
                        Emit( (loc, userID), userID)

class Partitioner
                method int getPartition(key, value, int numPartitions)
                        return key.first.hashCode() & Integer.MAX_VALUE % numPartitions

Class PairGroupingComparator extends WritableComparator
                method int compare(WritableComparable wc1, WritableComparable wc2)
                        return ((Pair) wc1).getLoc().compareTo(((Pair) wc1).getLoc())

class Reducer
                method Reduce(key, userList [])
                        Emit(key.getLoc(), userList)
```

Question 1 MapReduce

- Given the following graph, assume that you are using the single shortest path algorithm to compute the shortest path from node S to node E. Show the output of the mapper (sorted results of all mappers) and the reducer (only one reducer used) in each iteration (including both the distances and the paths).



Solution

1.
Mapper:
(A, 1), (C, 2)
Reducer:
A: 1 | S->A | B:6
C: 2 | S->C | A:4, D:3

2.
Mapper:
(B, 7), (A, 6), (D, 5)
Reducer:
B: 7 | S->A->B | D:1, E:2
D: 5 | S->C->D | E:1

3.
Mapper:
(E, 9), (D, 8), (E, 6)
Reducer:
E: 6 |S->C->D->E | empty

Algorithm terminates

Question 1 MapReduce

Assume that in an online shopping system, a huge log file stores the information of each transaction. Each line of the log is in format of "userID**\t**product**\t** price**\t** time". Your task is to use MapReduce to find out the top-5 most expensive products purchased by each user in 2016.

```
class Mapper
        initialize an associate array H(integer UserID, priority queue Q of log
record based on price)
      method Map(key, log record R)
            if R.time == 2016
                  H(R.userID).add(R)
                  if(H(R.userID).size >5)
                        H(R.userID).remove(first element)
        method CleanUp()
                foreach(entry E in H)
                      Emit(E.userID, E.Q)
class Combiner ???
class Reducer
      method Reduce(userID, list of queues[])
            P <- get top 5 products from the list of queues
            Emit(userID, P)
```

Question 1 MapReduce

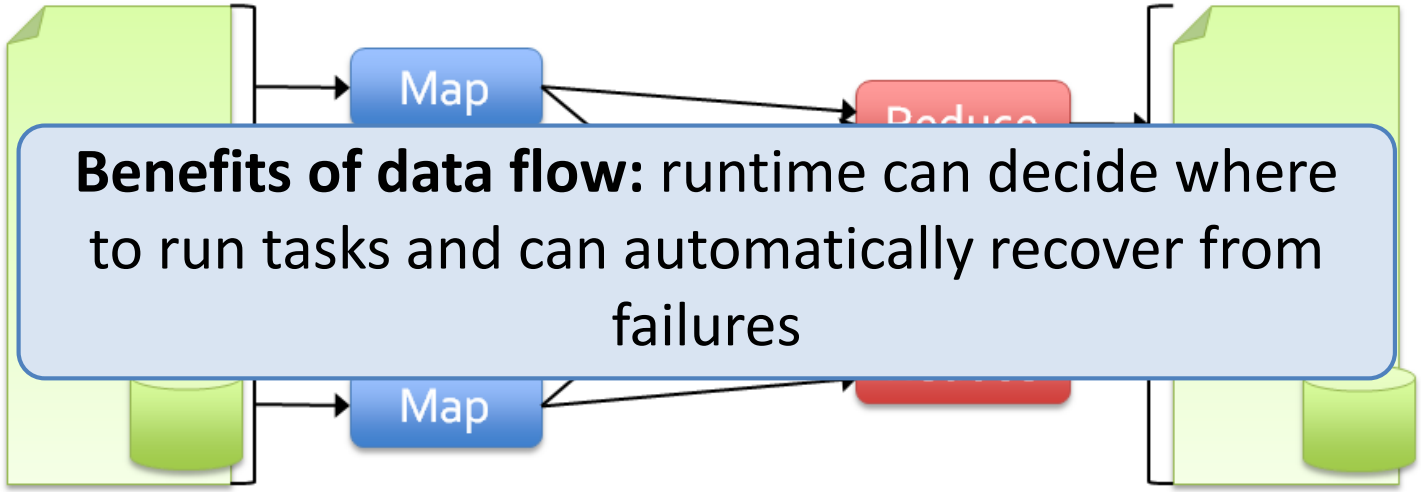
Given a large text file, find the top-k words that appear the most frequently.

Answer: Do word count first, and then in each reducer, output only the top-k words. In the second round map/reduce task, read the local top-k and compute the global top-k.

Topic 2: Spark Core and GraphX

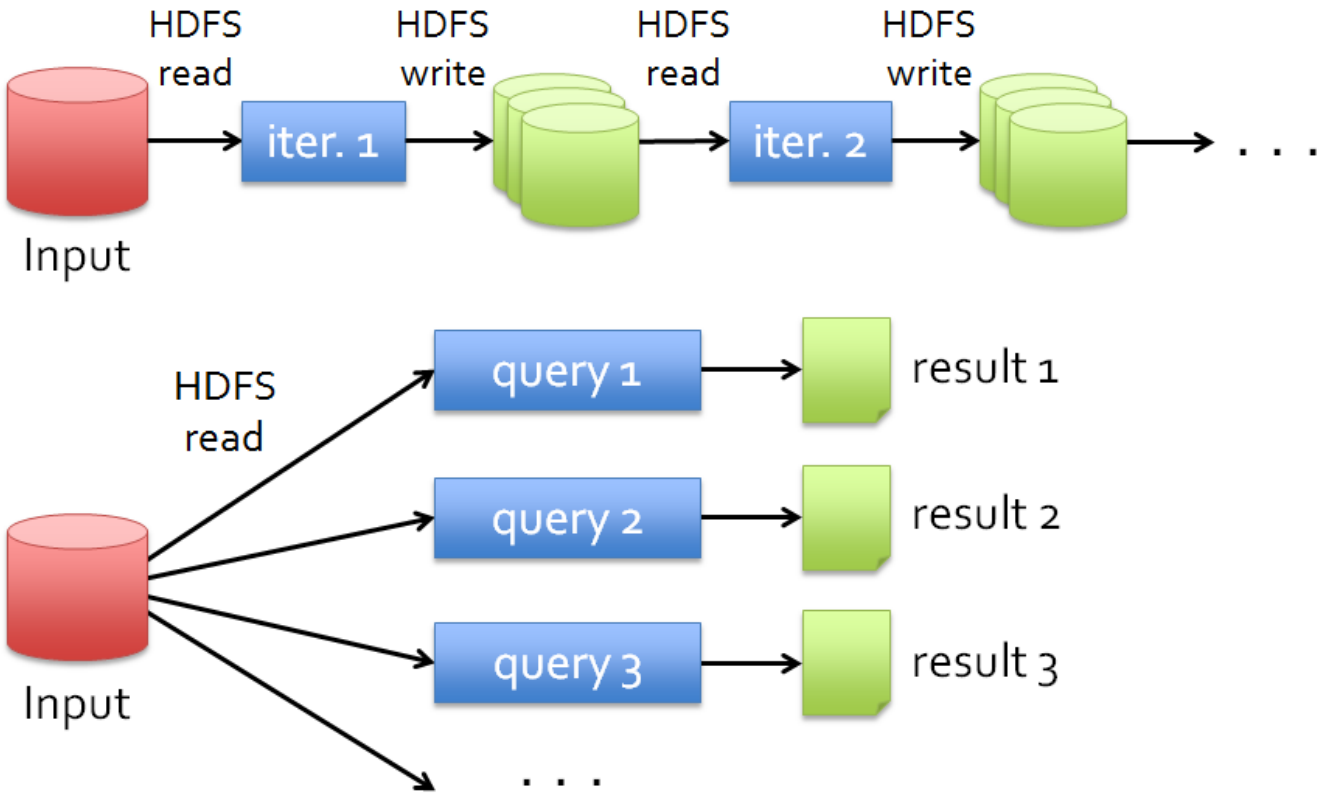
- ***Part A: Spark concepts***
- Part B: Show output of the given code
- Part C: Spark algorithm design
  - Spark Core
  - Spark GraphX

Limitations of MapReduce



- As a general programming model:
  - It is more suitable for one-pass computation on a large dataset
  - Hard to compose and nest multiple operations
  - No means of expressing iterative operations
- As implemented in Hadoop
  - All datasets are read from disk, then stored back on to disk
  - All data is (usually) triple-replicated for reliability
  - Not easy to write MapReduce programs using Java

Data Sharing in MapReduce



Slow due to replication, serialization, and disk IO

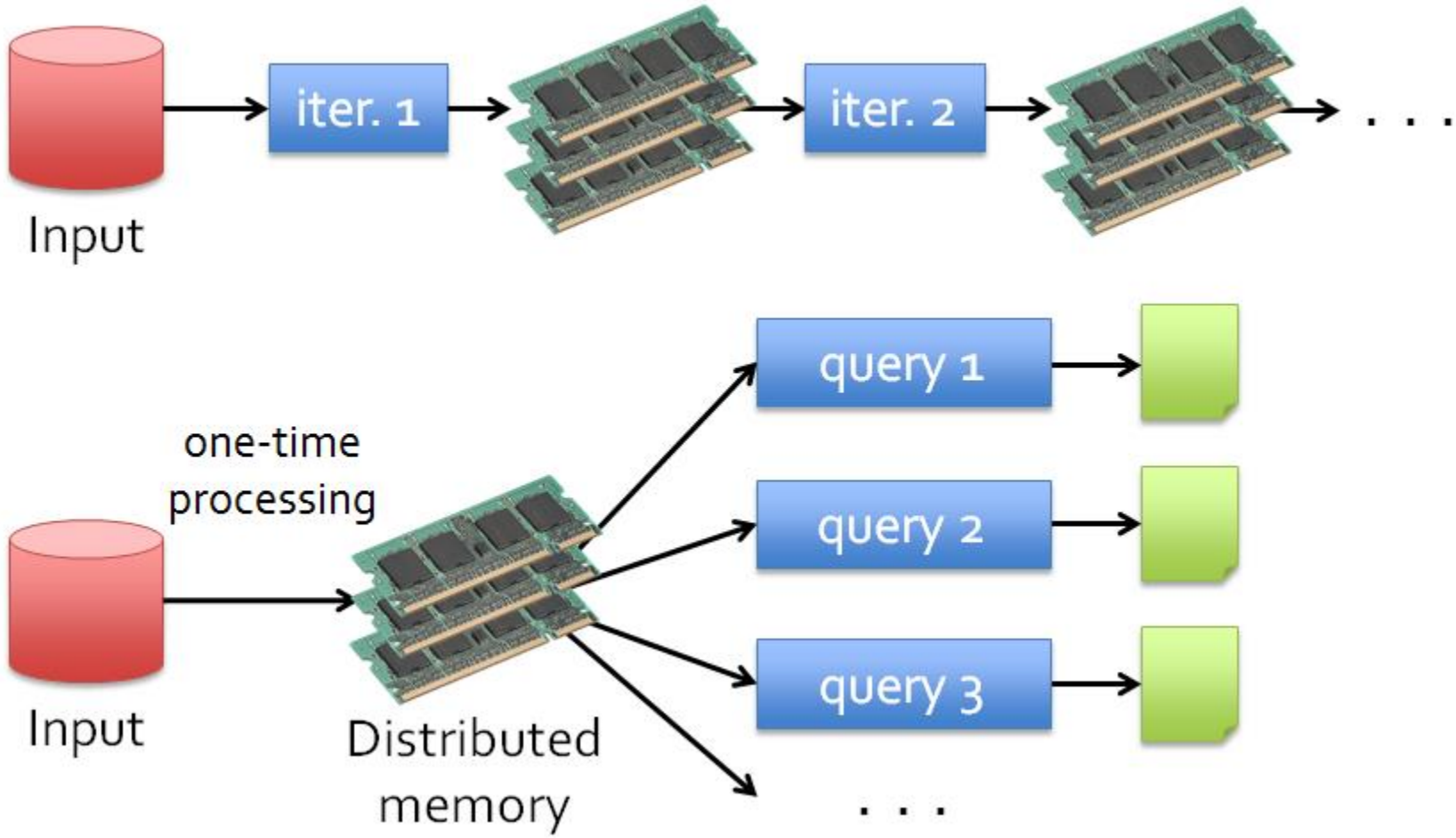
- Complex apps, streaming, and interactive queries all need one thing that MapReduce lacks:

  Efficient primitives for **data sharing**

Goals of Spark

- Keep more data in-memory to improve the performance!
- Extend the MapReduce model to better support two common classes of analytics apps:
  - Iterative algorithms (machine learning, graphs)
  - Interactive data mining
- Enhance programmability:
  - Integrate into Scala programming language
  - Allow interactive use from Scala interpreter

Data Sharing in Spark Using RDD



10-100× faster than network and disk

What is Spark

- Fast and expressive cluster computing system interoperable with Apache Hadoop
- Improves efficiency through:
  - **In-memory** computing primitives
  - General computation graphs
- Improves usability through:
  - Rich APIs in Scala, Java, Python
  - Interactive shell

→ Up to 100 × faster (10 × on disk)

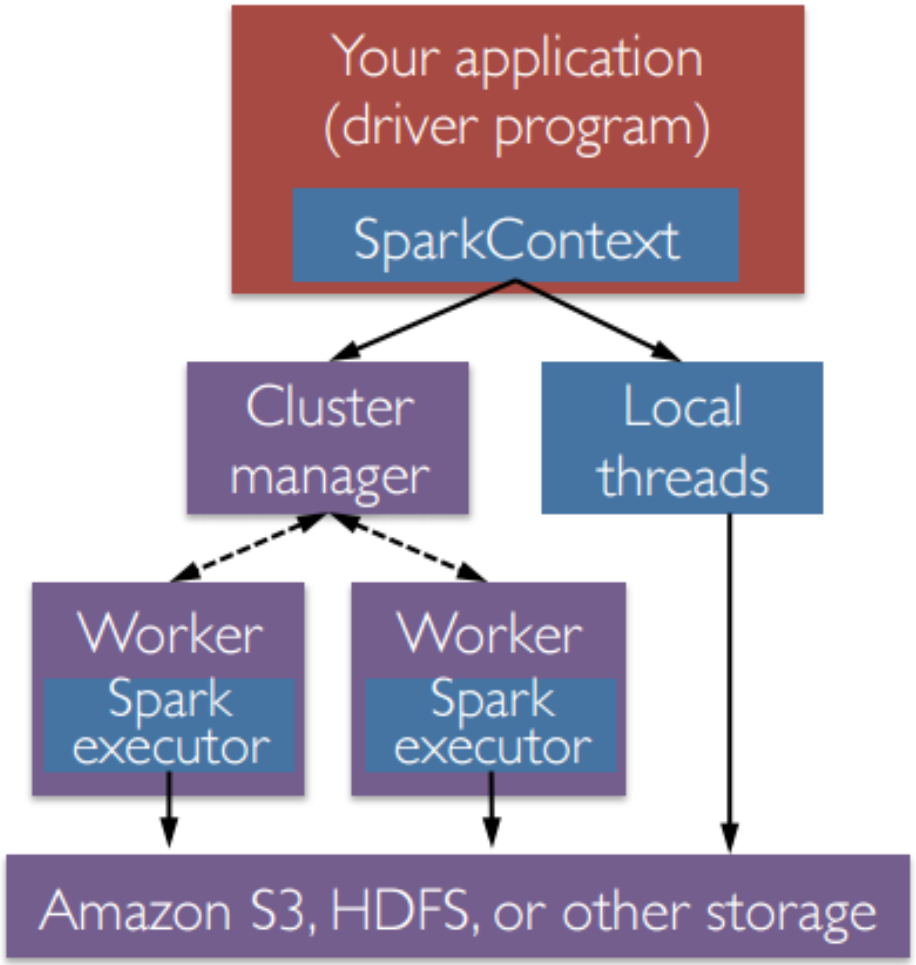
→ Often 5 × less code

- **Spark is not**
  - a modified version of Hadoop
  - dependent on Hadoop because it has its own cluster management
  - Spark uses Hadoop for storage purpose only

Spark Ideas

- Expressive computing system, not limited to map-reduce model
- Facilitate system memory
  - avoid saving intermediate results to disk
  - cache data for repetitive queries (e.g. for machine learning)
- Layer an **in-memory system** on top of Hadoop.
- Achieve **fault-tolerance** by re-execution instead of replication

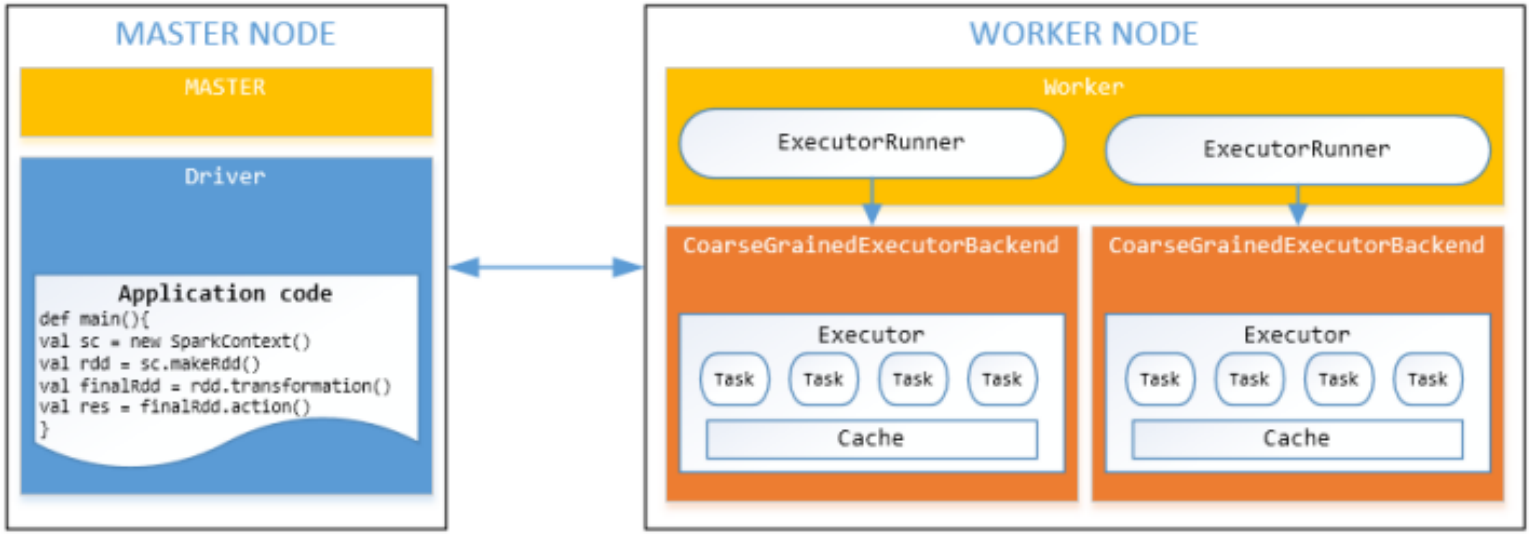
Spark Workflow



- A Spark program first creates a SparkContext object
  - Tells Spark how and where to access a cluster
  - Connect to several types of **cluster managers** (e.g., YARN, Mesos, or its own manager)
- Cluster manager:
  - Allocate resources across applications
- Spark executor:
  - Run computations
  - Access data storage

Worker Nodes and Executors

- Worker nodes are machines that run executors
  - Host one or multiple Workers
  - One JVM (1 process) per Worker
  - Each Worker can spawn one or more Executors
- Executors run tasks
  - Run in child JVM (1 process)
  - Execute one or more task using threads in a ThreadPool



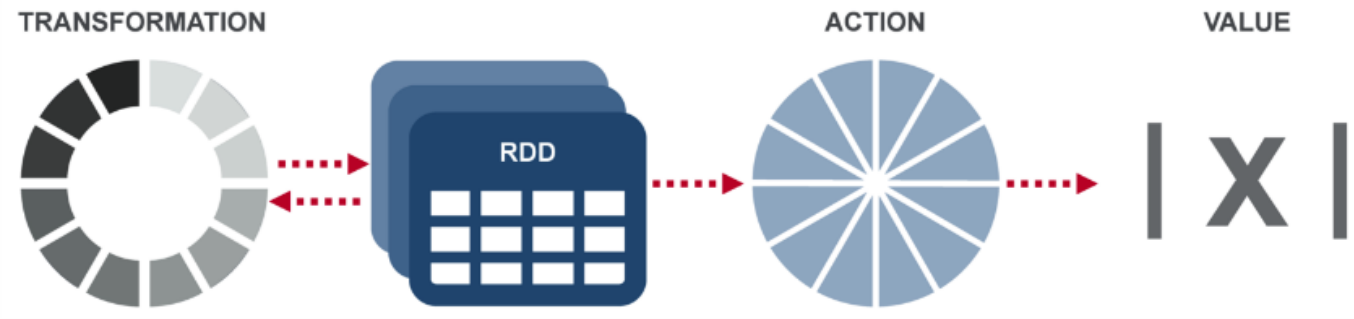
What is RDD

- **Resilient Distributed Datasets**: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.
  - RDD is a **distributed** memory abstraction that lets programmers perform **in-memory** computations on large clusters in a **fault-tolerant** manner.
- **Resilient**
  - Fault-tolerant, is able to recompute missing or damaged partitions due to node failures.
- **Distributed**
  - Data residing on multiple nodes in a cluster.
- **Dataset**
  - A collection of partitioned elements, e.g. tuples or other objects (that represent records of the data you work with).
- RDD is the primary data abstraction in Apache Spark and the core of Spark. It enables operations on collection of elements in parallel.

RDD Traits

- **In-Memory**, i.e. data inside RDD is stored in memory as much (size) and long (time) as possible.
- **Immutable** or **Read-Only**, i.e. it does not change once created and can only be transformed using transformations to new RDDs.
- **Lazy evaluated**, i.e. the data inside RDD is not available or transformed until an action is executed that triggers the execution.
- **Cacheable**, i.e. you can hold all the data in a persistent "storage" like memory (default and the most preferred) or disk (the least preferred due to access speed).
- **Parallel**, i.e. process data in parallel.
- **Typed**, i.e. values in a RDD have types, e.g. RDD[Long] or RDD[(Int, String)].
- **Partitioned**, i.e. the data inside a RDD is partitioned (split into partitions) and then distributed across nodes in a cluster (one partition per JVM that may or may not correspond to a single node).

RDD Operations



- **Transformation:** returns a new RDD.
  - Nothing gets evaluated when you call a Transformation function, it just takes an RDD and return a new RDD.
  - Transformation functions include *map, filter, flatMap, groupByKey, reduceByKey, aggregateByKey, filter, join, etc*.
- **Action:** evaluates and returns a new value.
  - When an Action function is called on a RDD object, all the data processing queries are computed at that time and the result value is returned.
  - Action operations include *reduce, collect, count, first, take, countByKey, foreach, saveAsTextFile, etc*.

Spark Transformations

- Create new datasets from an existing one
- Use lazy evaluation: results not computed right away – instead *Spark remembers set of transformations applied to base dataset*
  - Spark optimizes the required calculations
  - Spark recovers from failures
- Some transformation functions

Transformation	Description
`map(`*`func`*`)`	return a new distributed dataset formed by passing each element of the source through a function *func*
`filter(`*`func`*`)`	return a new dataset formed by selecting those elements of the source on which *func* returns true
`distinct([`*`numTasks`*`]))`	return a new dataset that contains the distinct elements of the source dataset
`flatMap(`*`func`*`)`	similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item)

Spark Actions

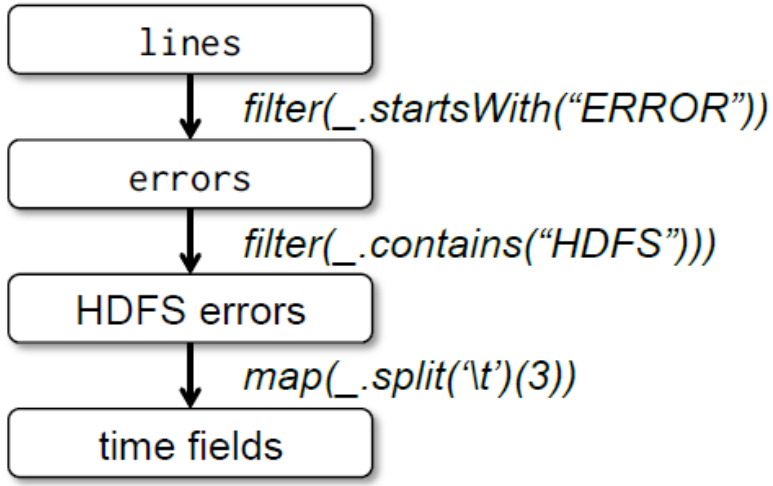
- Cause Spark to execute recipe to transform source
- Mechanism for getting results out of Spark
- Some action functions

Action	Description
`reduce(`*`func`*`)`	aggregate dataset's elements using function *func*. *func* takes two arguments and returns one, and is commutative and associative so that it can be computed correctly in parallel
`take(`*`n`*`)`	return an array with the first *n* elements
`collect()`	return all the elements as an array WARNING: make sure will fit in driver program
`takeOrdered(`*`n, key=func`*`)`	return n elements ordered in ascending order or as specified by the optional key function

- Example: words.collect().foreach(println)

Example

- Web service is experiencing errors and an operators want to search terabytes of logs in the Hadoop file system to find the cause.

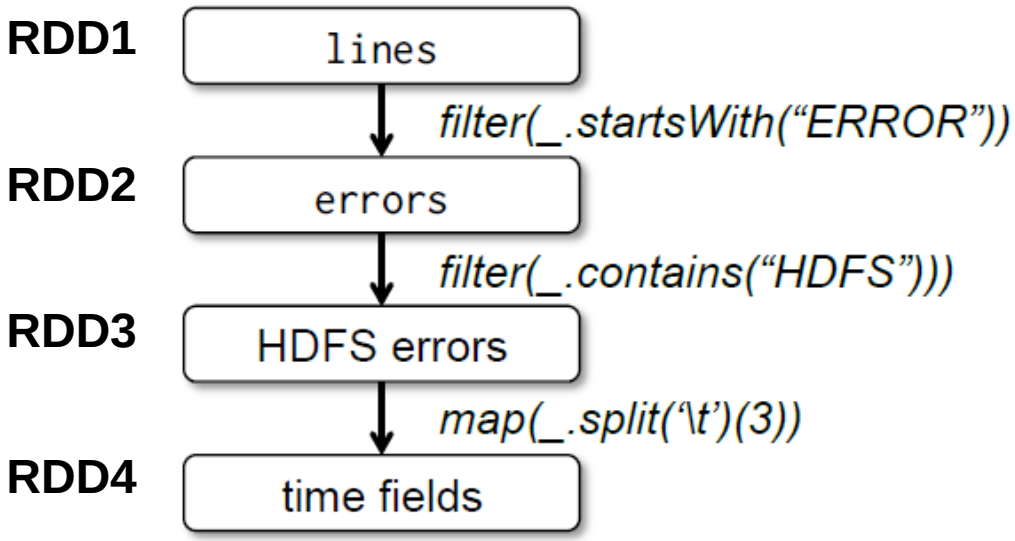



```
//base RDD
val lines = sc.textFile("hdfs://…")
//Transformed RDD
val errors =
lines.filter(_.startsWith("Error"))
errors.persist()
errors.count()
errors.filter(_.contains("HDFS"))
        .map(_.split('\t')(3))
        .collect()
```

- Line1: RDD backed by an HDFS file (base RDD lines not loaded in memory)
- Line3: Asks for errors to persist in memory (errors are in RAM)

Lineage Graph

- RDDs keep track of lineage
- RDD has enough information about how it was derived from to compute its partitions from data in stable storage.



- Example:
  - If a partition of errors is lost, Spark rebuilds it by applying a filter on only the corresponding partition of lines.
  - Partitions can be recomputed in parallel on different nodes, without having to roll back the whole program.

Spark Key-Value RDDs

- Similar to Map Reduce, Spark supports Key-Value pairs
- Each element of a *Pair RDD* is a pair tuple
- Some Key-Value transformation functions:

Key-Value Transformation	Description
`reduceByKey(`*`func`*`)`	return a new distributed dataset of (K,V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) ➔ V
`sortByKey()`	return a new dataset (K,V) pairs sorted by keys in ascending order
`groupByKey()`	return a new dataset of (K, Iterable<V>) pairs

More Examples on Pair RDD

- Create a pair RDD from existing RDDs

```
val pairs = sc.parallelize( List( (“This”, 2), (“is”, 3), (“Spark”, 5), (“is”, 3) ) )
pairs.collect().foreach(println)
```

Output?

- reduceByKey() function: reduce key-value pairs by key using give *func*

```
val pair1 = pairs.reduceByKey((x,y) => x + y)
pairs1.collect().foreach(println)
```

Output?

- mapValues() function: work on values only

```
val pair2 = pairs.mapValues( x => x -1 )
pairs2.collect().foreach(println)
```

Output?

- groupByKey() function: When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs

```
pairs.groupByKey().collect().foreach(println)
```

map vs. flatMap

- Sample input file:

```
comp9313@comp9313-VirtualBox:~$ hdfs dfs -cat inputfile
This is a short sentence.
This is a second sentence.
```

```
scala> val inputfile = sc.textFile("inputfile")
inputfile: org.apache.spark.rdd.RDD[String] = inputfile MapPartitionsRDD[1] at t
extFile at <console>:24
```

- map: Return a new distributed dataset formed by passing each element of the source through a function *func*.

```
scala> inputfile.map(x => x.split(" ")).collect()
res3: Array[Array[String]] = Array(Array(This, is, a, short, sentence.), Array(T
his, is, a, second, sentence.))
```

- flatMap: Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).

```
scala> inputfile.flatMap(x => x.split(" ")).collect()
res4: Array[String] = Array(This, is, a, short, sentence., This, is, a, second,
sentence.)
```

RDD Operations

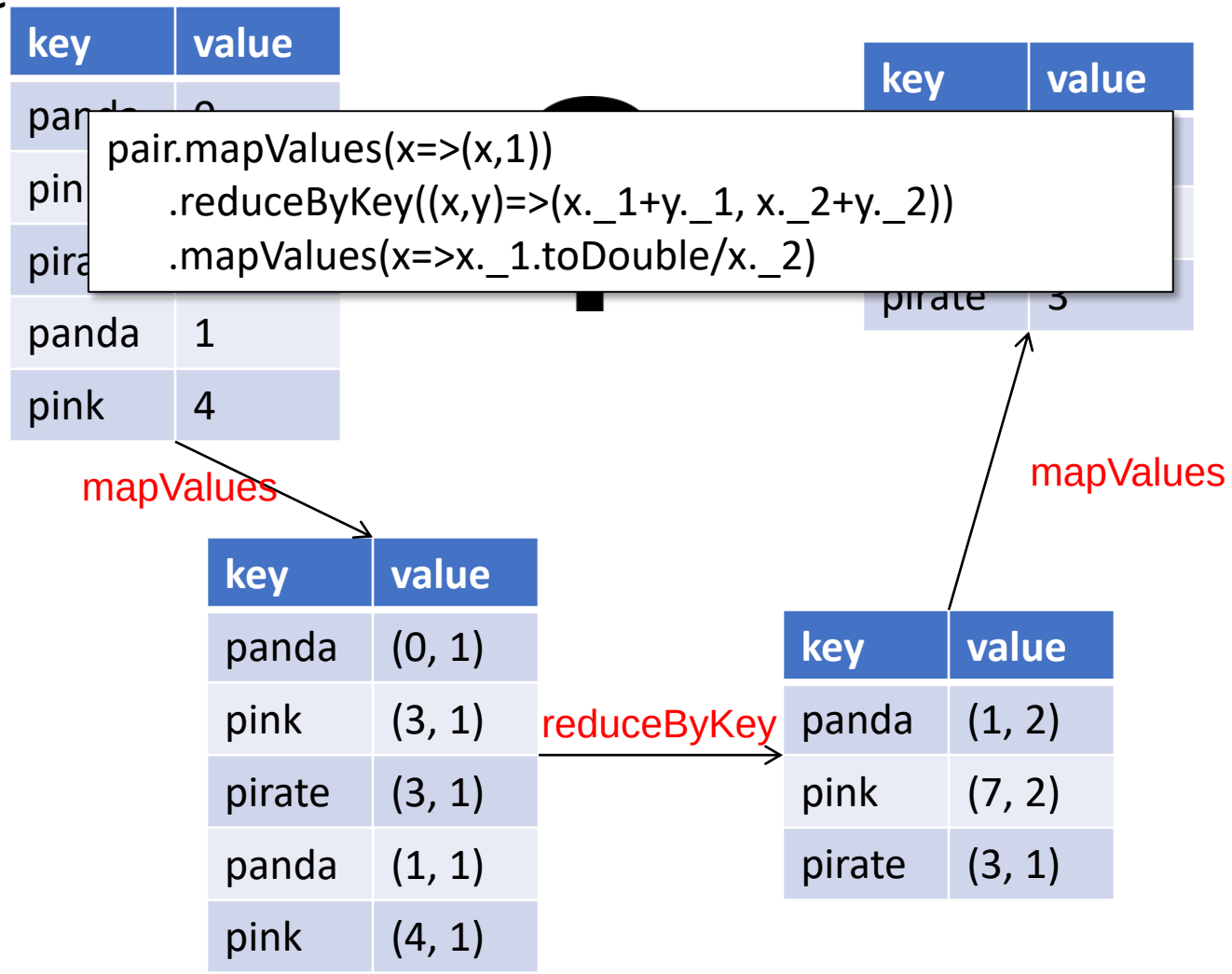
Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(*f* : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

Topic 2: Spark Core and GraphX

- Part A: Spark concepts
- ***Part B: Show output of the given code***
- Part C: Spark algorithm design
  - Spark Core
  - Spark GraphX

Practice

- Problem 1: Given a pair RDD of type [(String, Int)], compute the per-key average



Practice

- Problem 2: Given the data in format of key-value pairs <Int, Int>, find the maximum value for each key across all values associated with that key.

```
val pairs = sc. Parallelize(List((1, 2), (3, 4),… …))

val resMax = pairs.groupByKey().mapValues(x=>x.max)

val resMax = pairs.reduceByKey( (a, b) => if(a > b) a else b )

resMax.foreach(x => println(x._1, x._2))
```

Practice

- Problem 3: Given a collection of documents, compute the average length of words starting with each letter.

```
val textFile = sc.textFile(inputFile)
val words = textFile.flatMap(_.split(“ “).toLowerCase)

val counts = words.filter(x=> x.length >=1 && x.charAt(0)<=’z’ &&
            x.charAt(0)>=’a’).map(x=>(x.charAt(0), (x.length, 1)))

val avgLen = counts.reduceByKey((a, b)=>(a._1+b._1,
a._2+b._2)).foreach(x=>(x._1,          x._2._1.toDouble/x._2._2))

avgLen.foreach(x => println(x._1, x._2))
```

Question 2 Spark

- Write down the output

a) val lines = sc.parallelize(List("hello world", "this is a scala program", "to create a pair RDD", "in spark"))

```
val pairs = lines.map(x => (x.split(" ")(0), x))
pairs.filter {case (key, value) => key.length <3}.foreach(println)
```

Output: (“to”, “to create a pair RDD”) (“in”, “in spark”)

b) val pairs = sc.parallelize(List((1, 2), (3, 4), (3, 9), (4,2)))

```
val pairs1 = pairs.mapValues(x=>(x, 1)).reduceByKey((x,y) => (x._1 + y._1, x._2+y._2)).mapValues(x=>x._2/x._1)
pairs1.foreach(println)
```

Output: (1, 0) (3, 0) (4, 0) (because no “.toDouble” used)

Question 2 Spark

- Given a large text file, your task is to find out the top-k most frequent co-occurring term pairs. The co-occurrence of (w, u) is defined as: u and w appear in the same line (this also means that (w, u) and (u, w) are treated equally). Your Spark program should generate a list of ***k*** key-value pairs ranked in descending order according to the frequencies, where the keys are the pair of terms and the values are the co-occurring frequencies (**Hint:** you need to define a function which takes an array of terms as input and generate all possible pairs).

```
val textFile = sc.textFile(inputFile)
val words = textFile.map(_.split(" ").toLowerCase)

// fill your code here, and store the result in a pair RDD topk

topk.foreach(x => println(x._1, x._2))
```

Solution

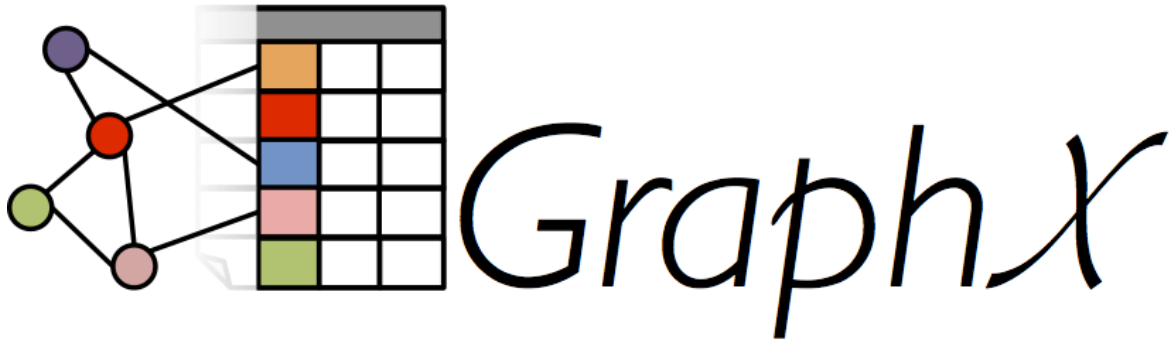
```
def pairGen(wordArray: Array[String]) : ArrayBuffer[(String, Int)] = {
  val abuf = new ArrayBuffer[(String, Int)]

  for(i <- 0 to wordArray.length -1){
   val term1 = wordArray(i)
   if(term1.length()>0){
    for(j <- i+1 to wordArray.length - 1){
     val term2 = wordArray(j)
     if(term2.length()>0){
      if(term1 < term2){abuf.+=:(term1 + "," + term2, 1)}
      else {abuf.+=:(term2 + "," + term1, 1)}
     }
    }
   }
  }
  return abuf
}


val textFile = sc.textFile(inputFile)
val words = textFile.map(_.split(" ").toLowerCase)

val pairs = words.flatMap(x => pairGen(x)).reduceByKey(_+_)
val topk = pairs.map(_.swap).sortByKey(false).take(k).map(_.swap)

topk.foreach(x => println(x._1, x._2))
```

Topic 2: Spark Core and GraphX

- Part A: Spark concepts
- Part B: Show output of the given code
- ***Part C: Spark algorithm design***
  - Spark Core
  - Spark GraphX

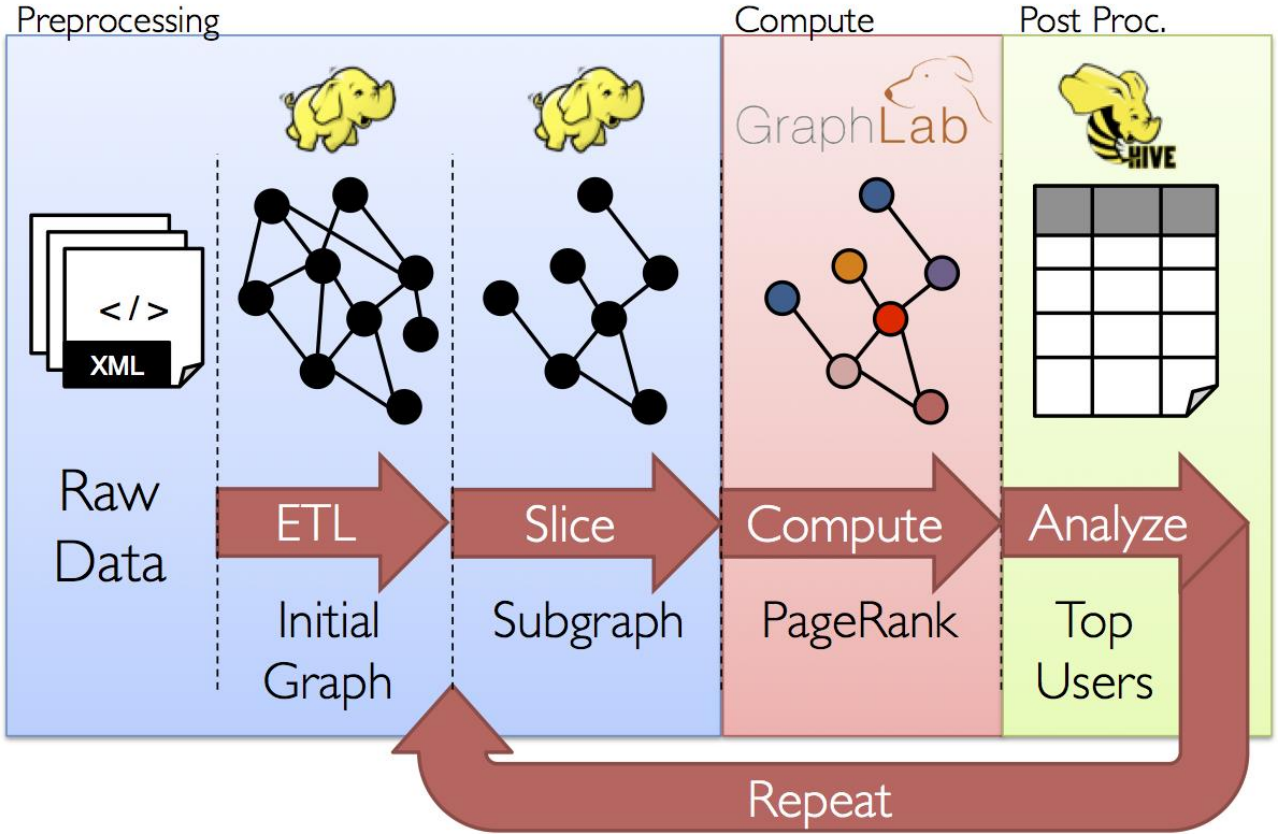
Spark GraphX

- **GraphX** is Apache Spark's API for graphs and graph-parallel computation.
- At a high level, GraphX extends the Spark RDD by introducing a new Graph abstraction: **a directed multigraph with properties attached to each vertex and edge**
- To support graph computation, GraphX exposes a set of fundamental operators (e.g., subgraph, joinVertices) as well as an optimized variant of the Pregel API
- GraphX includes a growing collection of graph algorithms and builders to simplify graph analytics tasks.



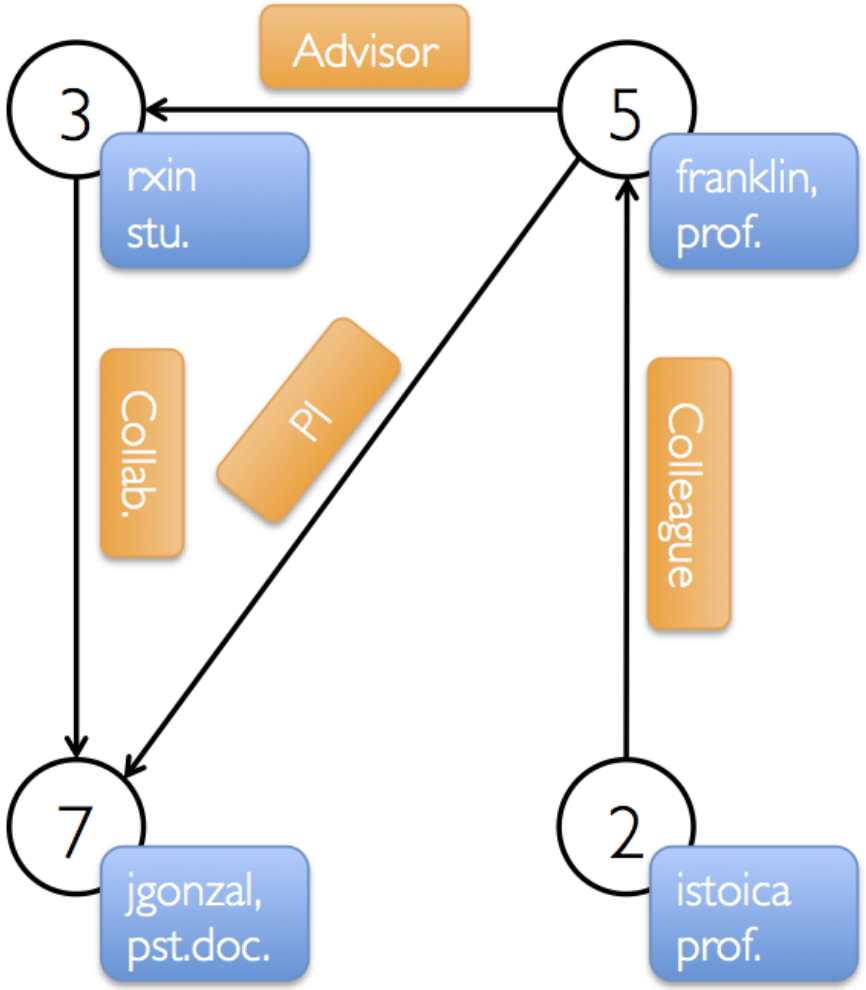
GraphX Motivation

- The goal of the GraphX project is to unify graph-parallel and data-parallel computation in one system with a single composable API.
- The GraphX API enables users to view data both as graphs and as collections (i.e., RDDs) without data movement or duplication.



View a Graph as a Table

Property Graph



Vertex Table

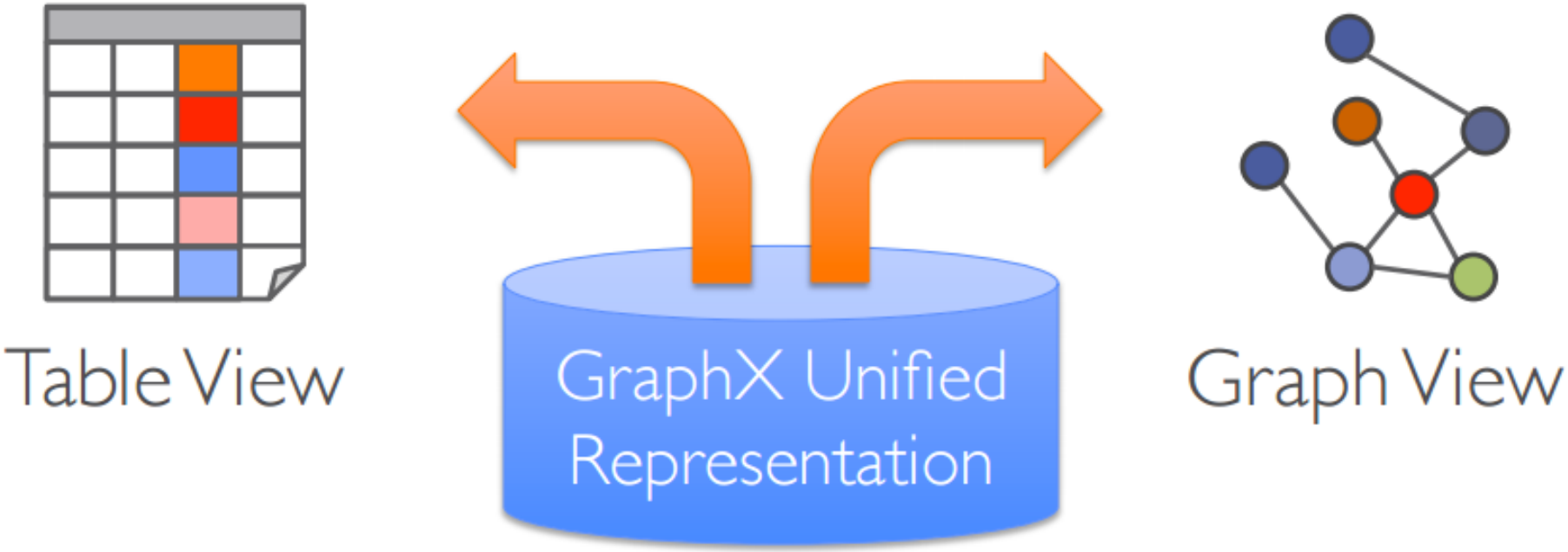
Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

GraphX Motivation

- Tables and Graphs are composable views of the same physical data



- Each view has its own operators that exploit the semantics of the view to achieve efficient execution

Pregel Operators

```
def pregel[A]
    (initialMsg: A,
     maxIter: Int = Int.MaxValue,
     activeDir: EdgeDirection = EdgeDirection.Out)
    (vprog: (VertexId, VD, A) => VD,
     sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId, A)],
     mergeMsg: (A, A) => A)
  : Graph[VD, ED] = {
            … …
   }
```

- The first argument list contains configuration parameters including the initial message, the maximum number of iterations, and the edge direction in which to send messages (by default along out edges).

- The second argument list contains the user defined functions for receiving messages (the vertex program vprog), computing messages (sendMsg), and combining messages mergeMsg.