COMP9313: Big Data Management

Lecturer: Xin Cao

Course web site: http://www.cse.unsw.edu.au/~cs9313/

Chapter 12: Revision and Exam

Revision of Chapters Required in Exam

Topic 1：MapReduce (Chapters 2-4)

Map and Reduce Functions

- Programmers specify two functions:
  - **map** $(k_1, v_1) \rightarrow$ list $[<k_2, v_2>]$
    - Map transforms the input into key-value pairs to process
  - **reduce** $(k_2, [v_2]) \rightarrow [<k_3, v_3>]$
    - Reduce aggregates the list of values for each key
    - All values with the same key are sent to the same reducer
- Optionally, also:
  - combine $(k_2, [v_2]) \rightarrow [<k_3, v_3>]$
    - Mini-reducers that run in memory after the map phase
    - Used as an optimization to reduce network traffic
  - partition (k_2, number of partitions) $\rightarrow$ 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…

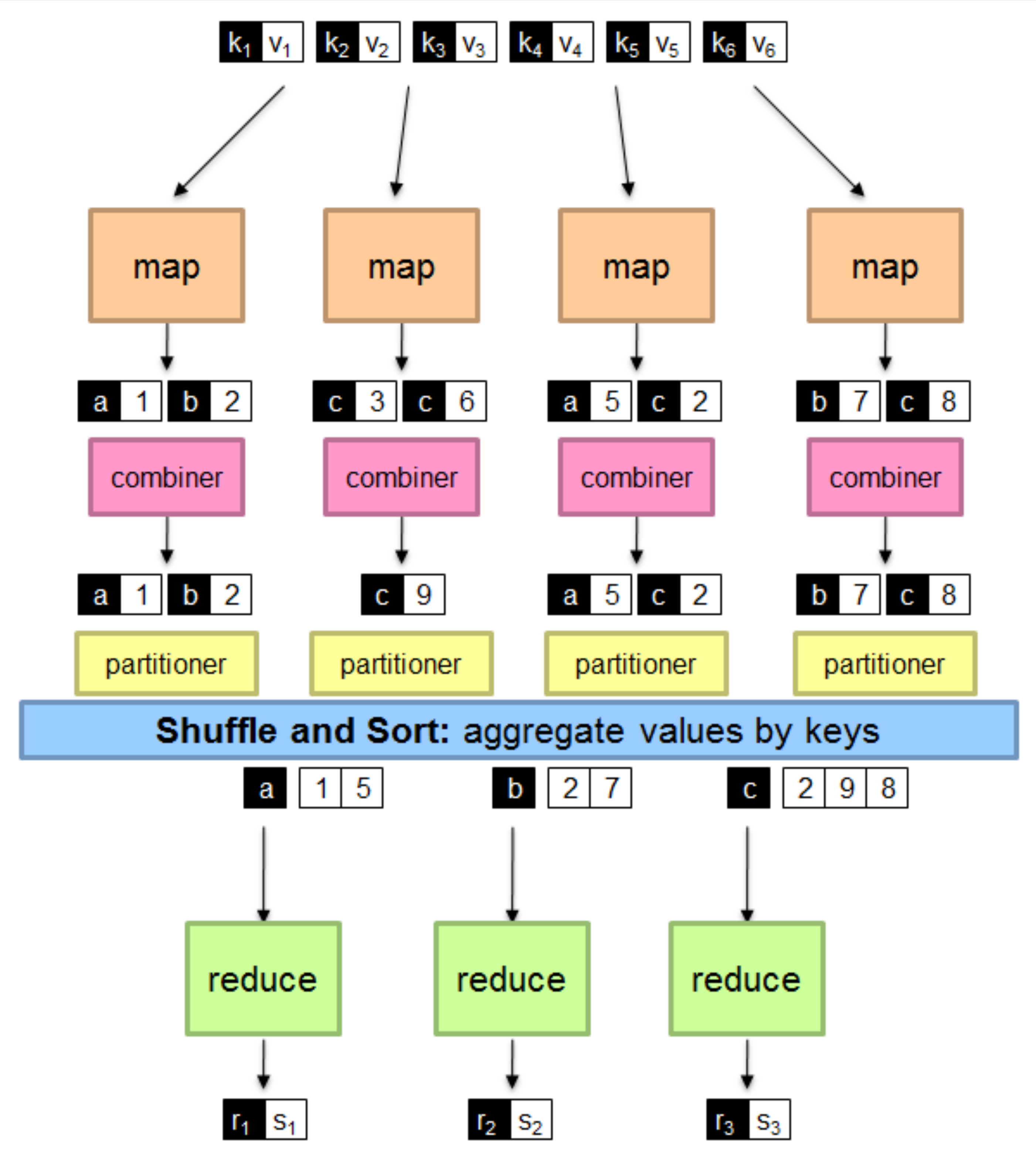
Combiners

- Often a Map task will produce many pairs of the form *(k,v_1), (k,v_2), …* 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

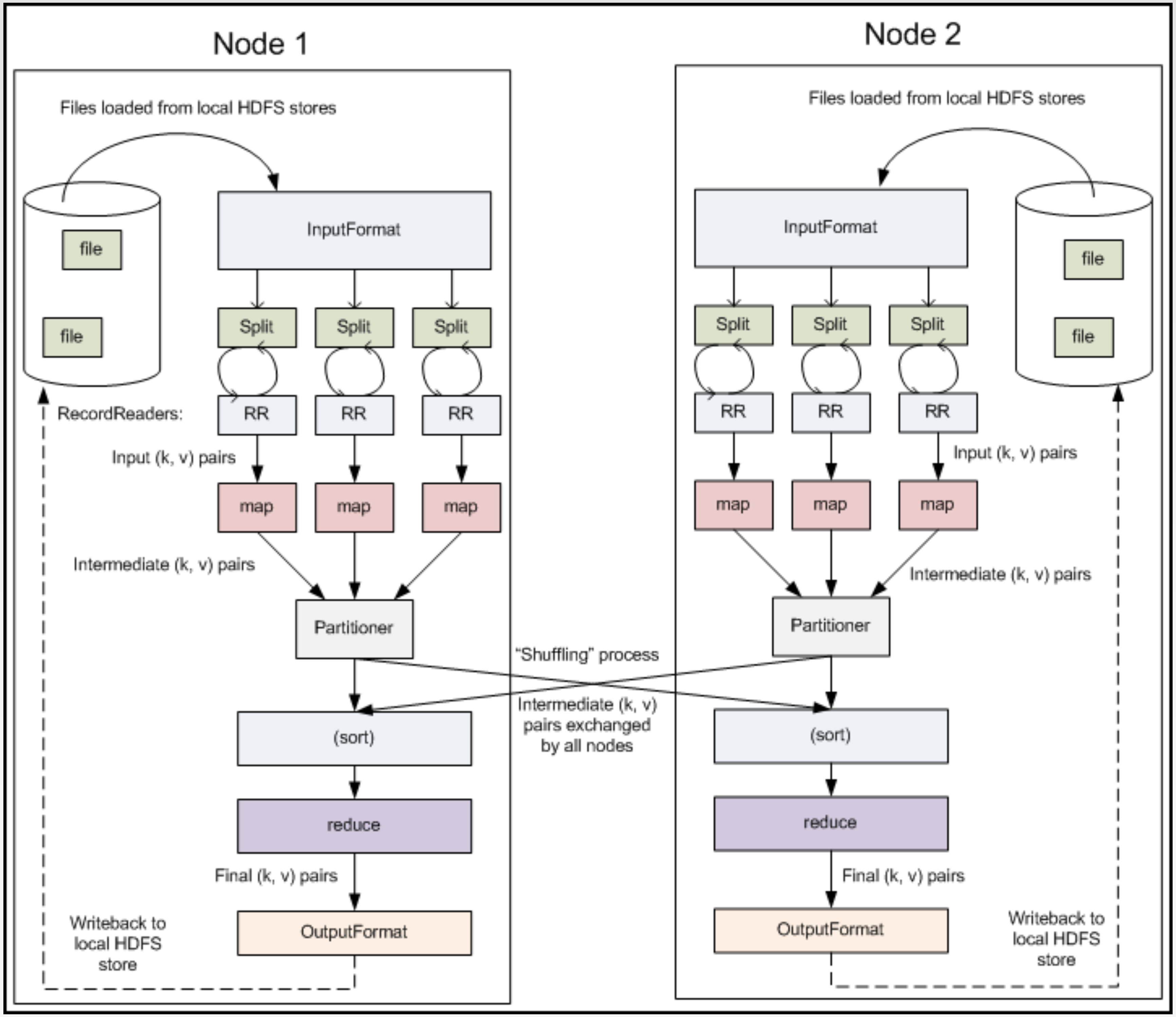
Partitioner

- 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

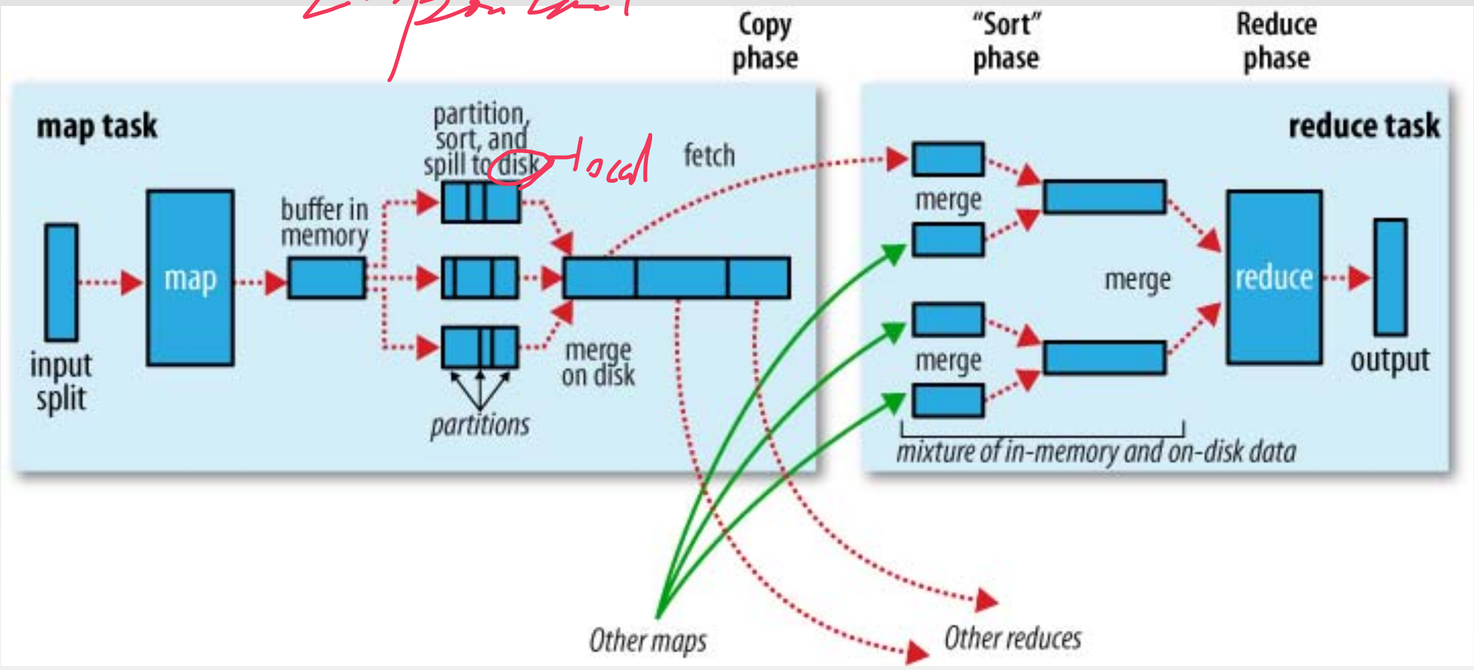


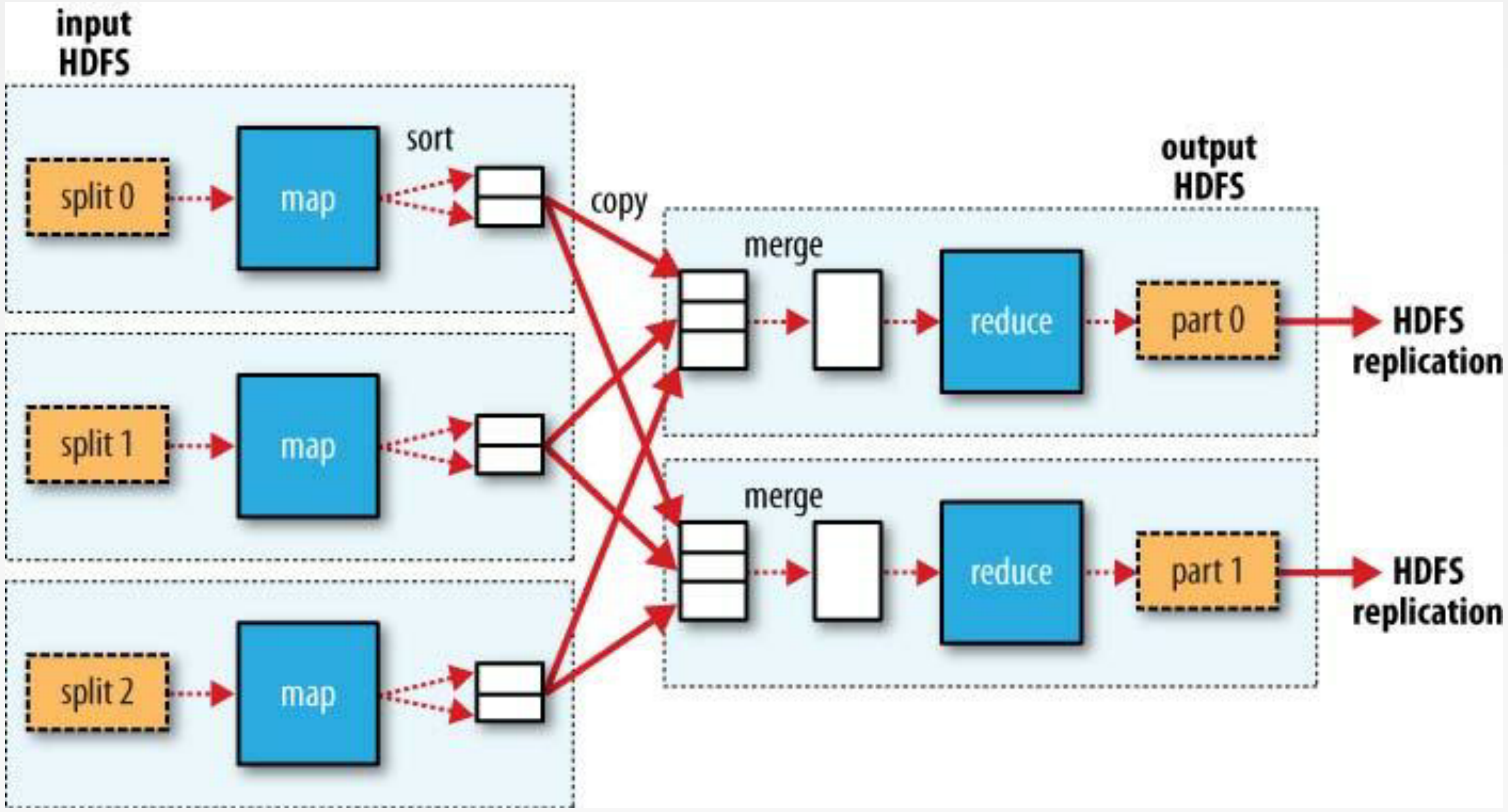
MapReduce Data Flow



MapReduce Data Flow

Important



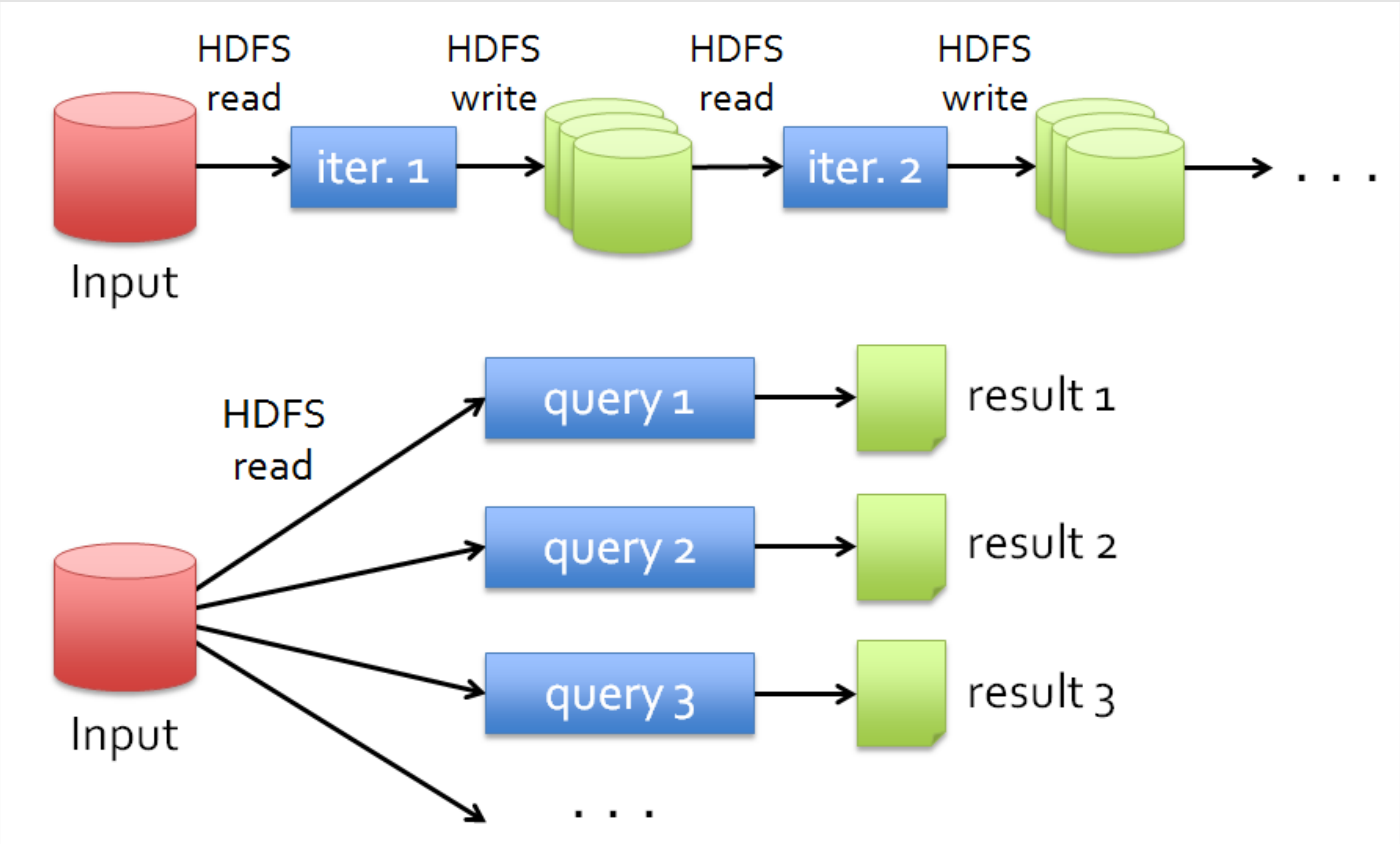


MapReduce Algorithm Design Patterns

- 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

Topic 2：Spark Core and GraphX (Chapters 6 and 7)

Data Sharing in MapReduce

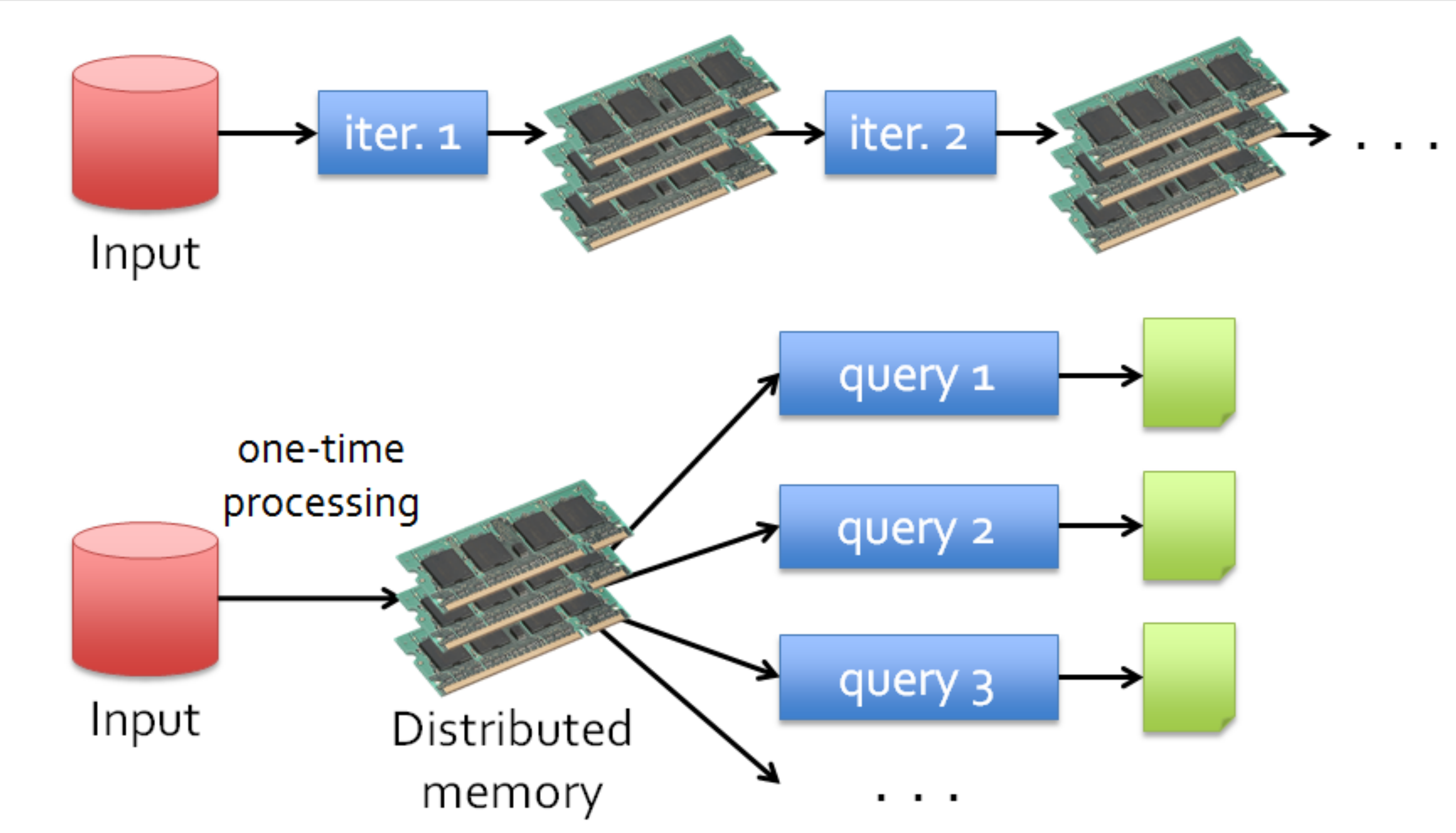


Slow due to replication, serialization, and disk IO

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

Efficient primitives for **data sharing**

Data Sharing in Spark Using RDD

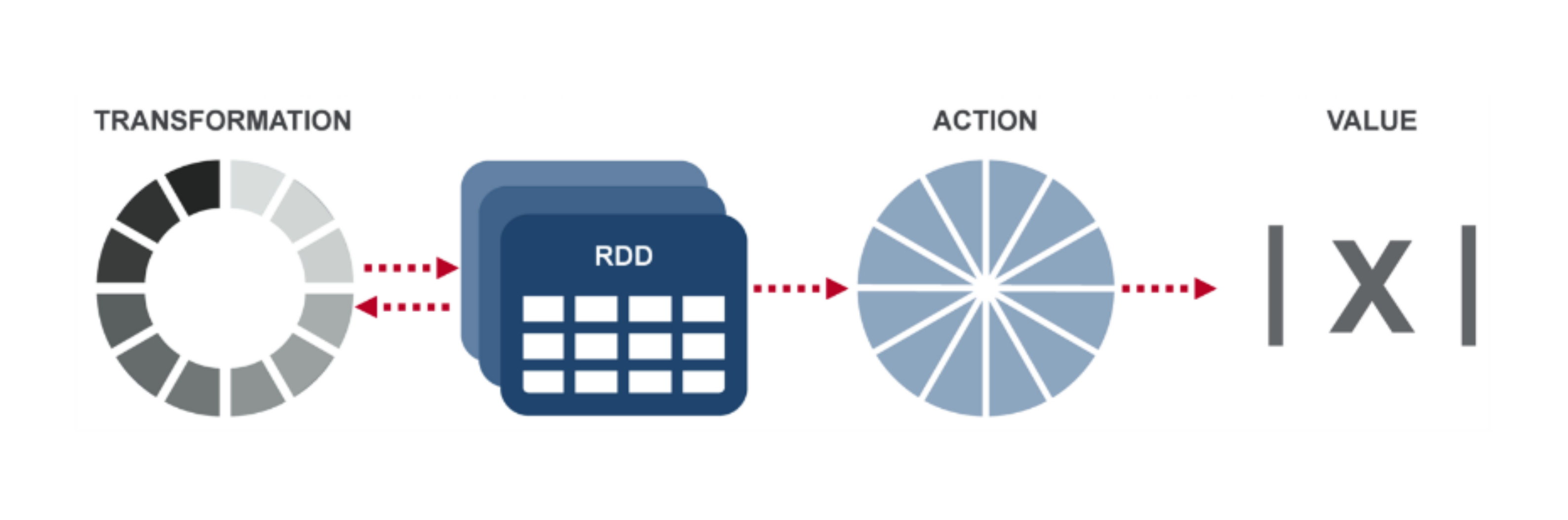


10-100× faster than network and disk

What is RDD

- Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. Matei Zaharia, et al. NSDI'12
  - 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 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.*

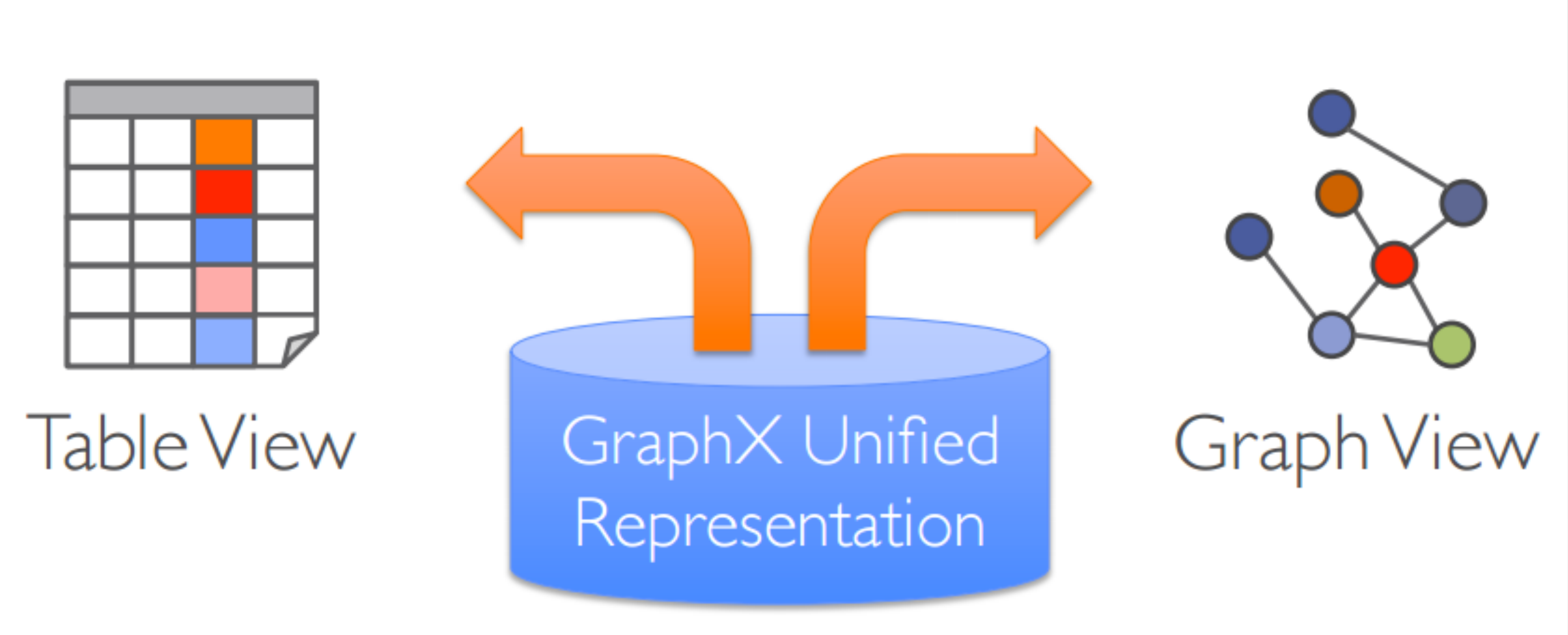
RDD Operations

required

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

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

like shortest Path

```
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).

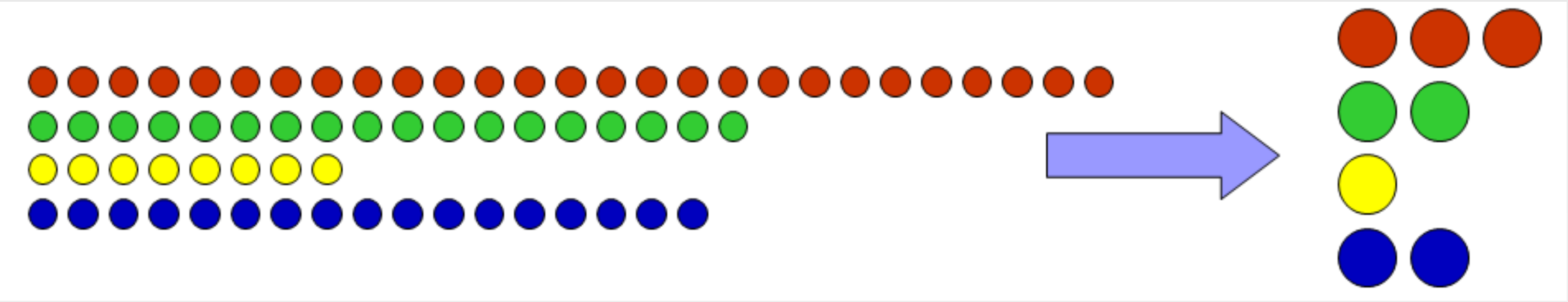
- The second argument list contains the user defined functions for receiving messages (the vertex program vprog), computing messages (sendMsg), and combining messages mergeMsg.

Topic 3：Mining Data Streams (Chapter 8)

- Types of queries one wants on answer on a data stream: (we'll learn these today)
  - Sampling data from a stream
    - Construct a random sample
  - Queries over sliding windows
    - Number of items of type x in the last *k* elements of the stream
  - Filtering a data stream
    - Select elements with property x from the stream

Sampling Data Streams

- Since we can not store the entire stream, one obvious approach is to store a **sample**



- Two different problems:
  - **(1)** Sample a **fixed proportion** of elements in the stream (say 1 in 10)
    - As the stream grows the sample also gets bigger
  - **(2)** Maintain a **random sample of fixed size** over a potentially infinite stream
    - As the stream grows, the sample is of fixed size
    - At any "time" ***t*** we would like a random sample of ***s*** elements
      - **What is the property of the sample we want to maintain?** For all time steps ***t***, each of ***t*** elements seen so far has equal probability of being sampled

Fixup: DGIM Algorithm

- **Idea:** Instead of summarizing fixed-length blocks, summarize blocks with specific number of **1s**:
  - Let the block ***sizes*** (number of **1s**) increase exponentially

- When there are few 1s in the window, block sizes stay small, so errors are small

10010101100010110101010101010110101010101011101010101110101000101110010

← ***N*** →

- Timestamps:
  - Each bit in the stream has a timestamp, starting from **1**, **2,** …
  - Record timestamps modulo ***N*** (**the window size**), so we can represent any **relevant** timestamp in $O(\log_2 N)$ bits
    - E.g., given the windows size 40 (***N***), timestamp 123 will be recorded as 3, and thus the encoding is on 3 rather than 123

Example: Updating Buckets

Current state of the stream:

10010101100010110101010101010101101010101010101110101010111010100010110010

Bit of value 1 arrives

0010101100010110101010101010101101010101010101110101010111010100010110010 1

Two white buckets get merged into a yellow bucket

00101011000101101010101010101011010101010101011101010101110101000101100101

Next bit 1 arrives, new orange white is created, then 0 comes, then 1:

0101100010110101010101010101011010101010101011101010101110101000101100101101

Buckets get merged…

0101100010110101010101010101011010101010101011101010101110101000101100101101

State of the bucket after merging

0101100010110101010101010101011010101010101011101010101110101000101100101101

Bloom Filter

- Consider: **|S| = *m*, |B| = *n***
- Use ***k*** independent hash functions $h_1, \ldots, h_k$
- **Initialization:**
  - Set **B** to all **0s**
  - Hash each element $s \in S$ using each hash function h_i, set $B[h_i(s)] = 1$ (for each $i = 1, \ldots, k$)
- **Run-time:**
  - When a stream element with key ***x*** arrives
    - If $B[h_i(x)] = 1$ for all $i = 1, \ldots, k$ then declare that ***x*** is in ***S***
      - That is, ***x*** hashes to a bucket set to **1** for every hash function $h_i(x)$
    - Otherwise discard the element ***x***

Bloom Filter Example

- Consider a Bloom filter of size m=10 and number of hash functions k=3. Let H(x) denote the result of the three hash functions.
- The 10-bit array is initialized as below

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

- Insert x_0 with $H(x_0) = \{1, 4, 9\}$

0	1	2	3	4	5	6	7	8	9
0	1	0	0	1	0	0	0	0	1

- Insert x_1 with $H(x_1) = \{4, 5, 8\}$

0	1	2	3	4	5	6	7	8	9
0	1	0	0	1	1	0	0	1	1

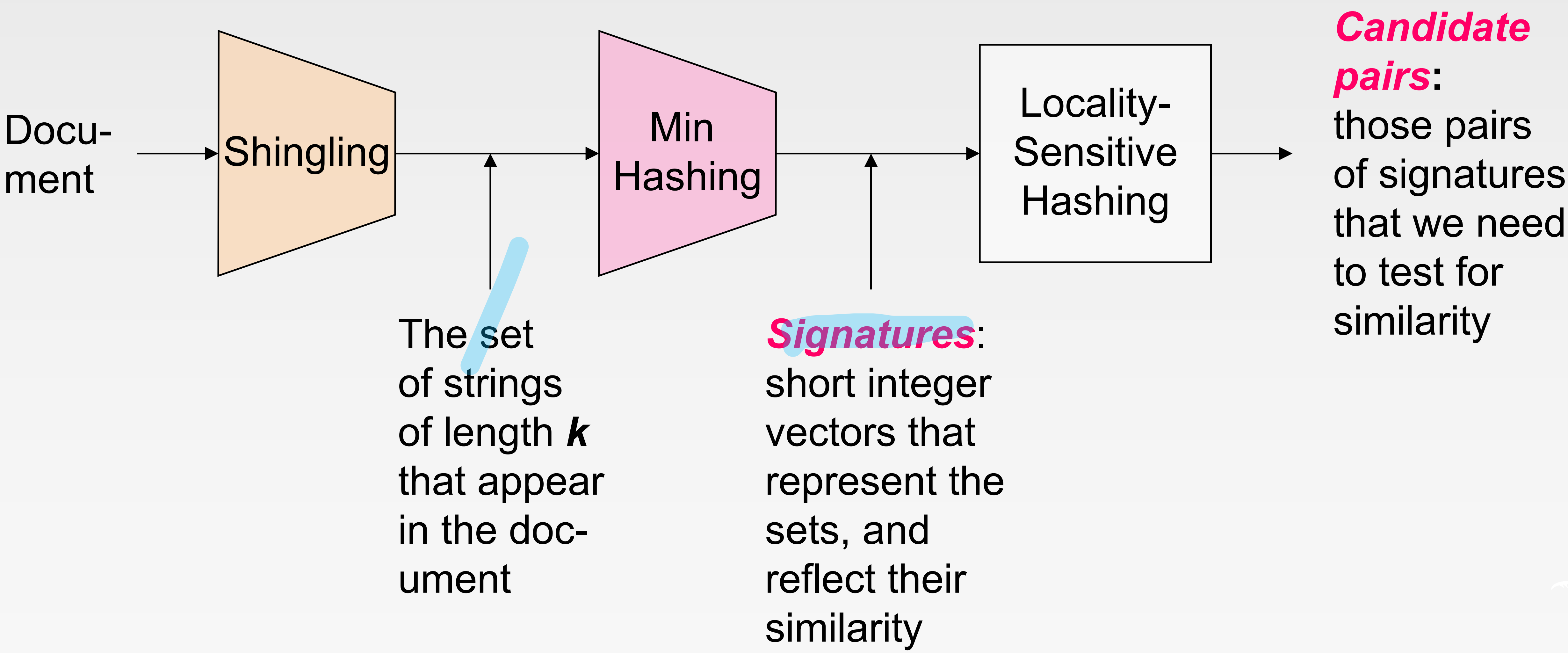
- Query y_0 with $H(y_0) = \{0, 4, 8\}$ => ???
- Query y_1 with $H(y_1) = \{1, 5, 8\}$ => ??? **False positive!**

check formula in previous slide

- Another Example: https://llimllib.github.io/bloomfilter-tutorial/

Topic 4：Finding Similar Items (Chapter 9)

- The Big Picture



Shingling

- A *k*-shingle (or *k*-gram) for a document is a sequence of *k* tokens that appears in the doc
  - Tokens can be characters, words or something else, depending on the application
  - Assume tokens = characters for examples

- **Example:** **k=2**; document $\mathbf{D_1}$ = abcab
  Set of 2-shingles: $\mathbf{S(D_1)}$ = {ab, bc, ca}

- Documents that are intuitively similar will have many shingles in common.
  - Example: k=3, “The dog which chased the cat” versus “The dog that chased the cat”.
    - Only 3-shingles replaced are g_w, _wh, whi, hic, ich, ch_, and h_c.

Min-Hash Signatures

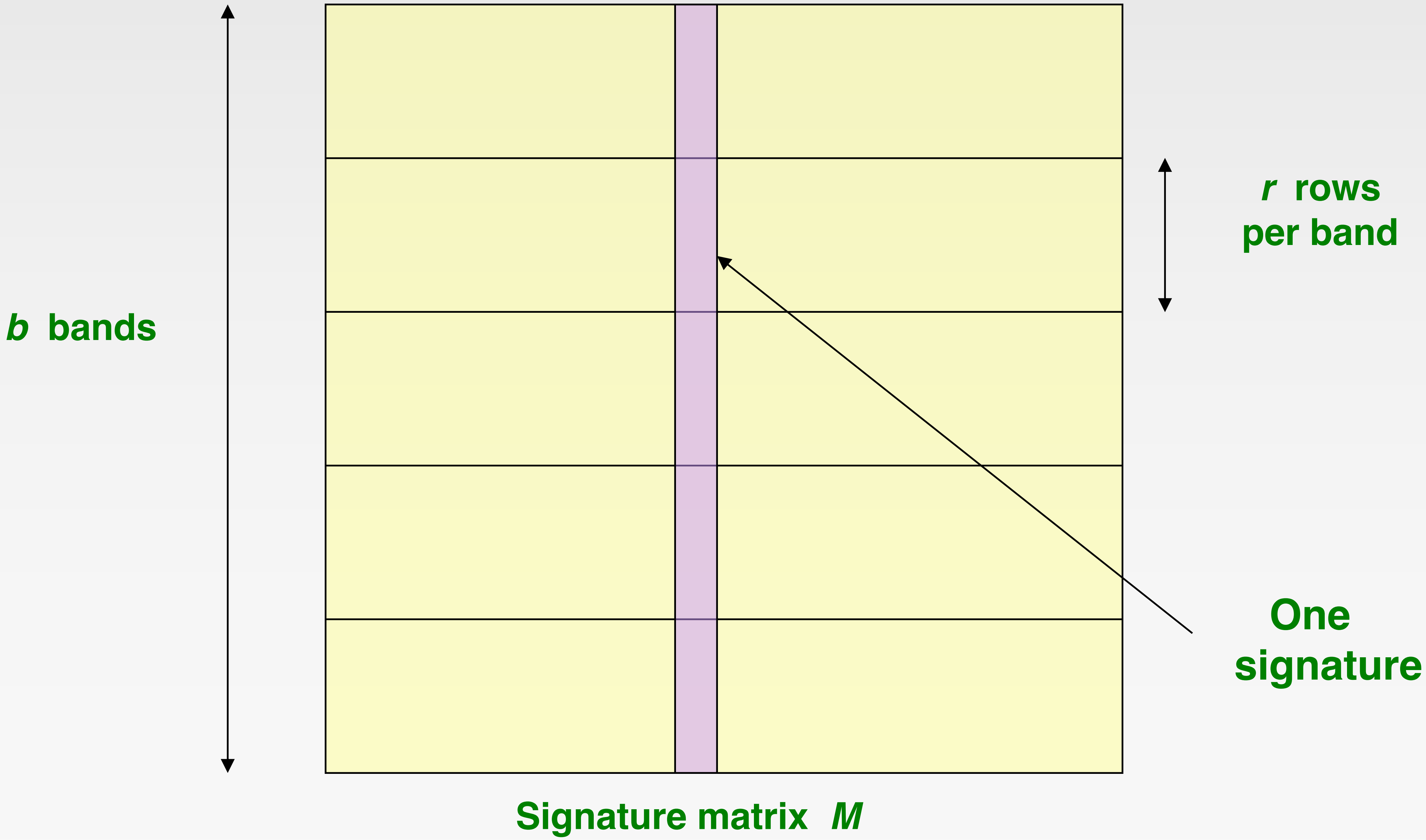
- **Pick K=100 random permutations of the rows**
- Think of ***sig*(C)** as a column vector
- **s*ig*(C)[i] =** according to the *i*-th permutation, the index of the first row that has a 1 in column *C*

$$sig(C)[i] = \min(\pi_i(C))$$

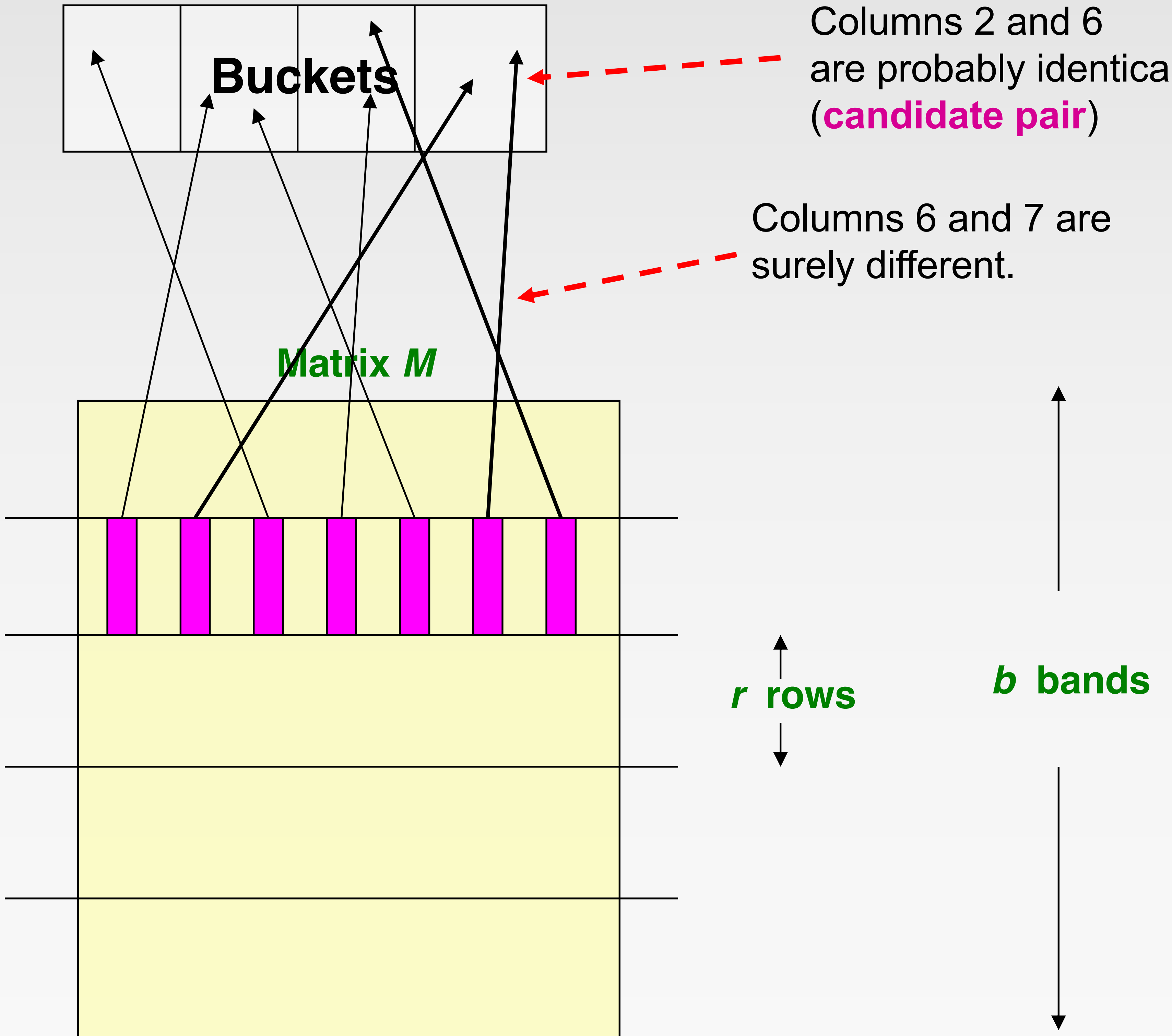
- **Note:** The sketch (signature) of document *C* is small **~100 bytes!**

- **We achieved our goal! We "compressed" long bit vectors into short signatures**

Partition *M* into *b* Bands



Hashing Bands



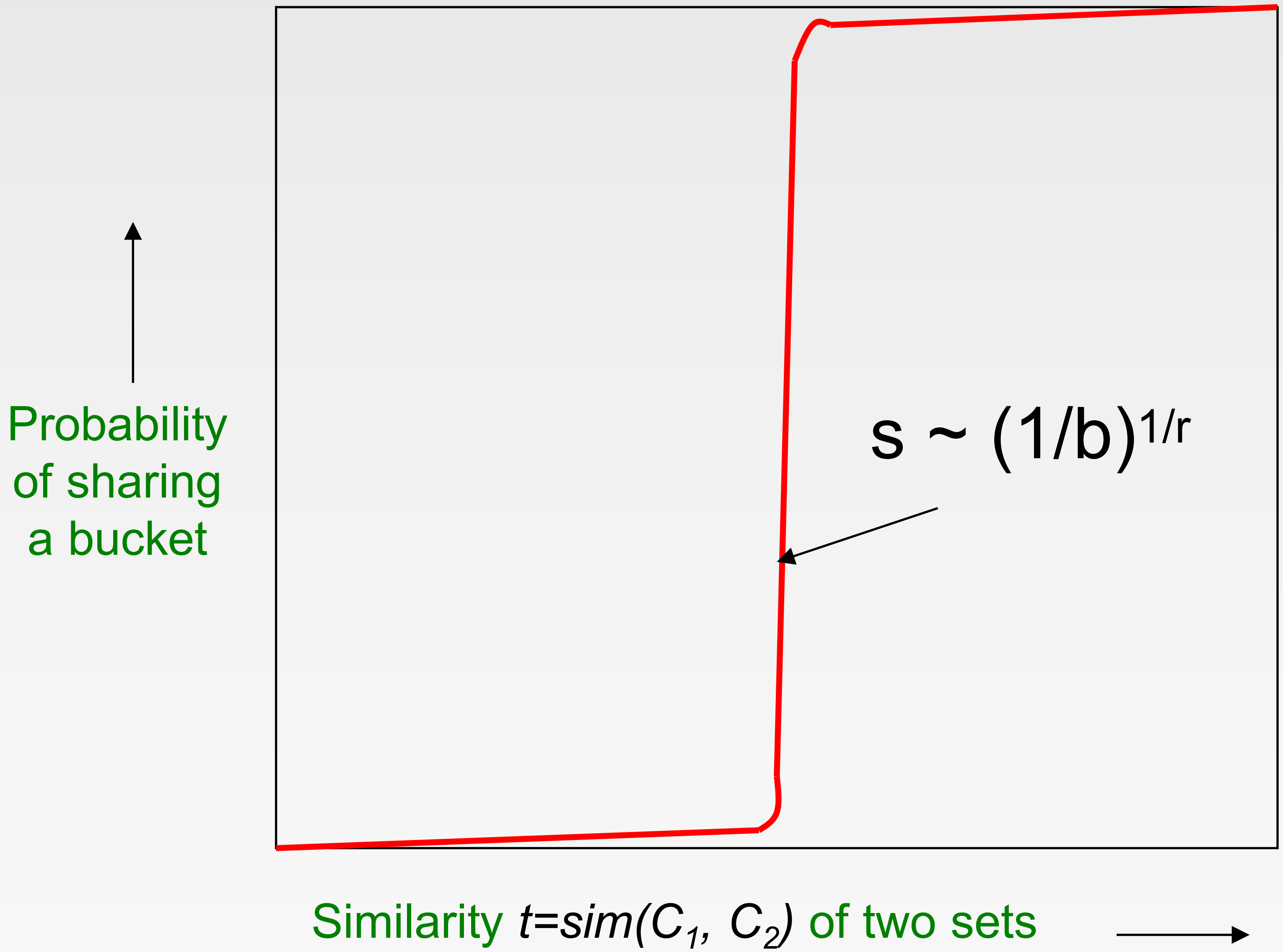
Columns 2 and 6 are probably identical (**candidate pair**)

Columns 6 and 7 are surely different.

b bands, *r* rows/band

- The probability that the minhash signatures for the documents agree in any one particular row of the signature matrix is t ($sim(C_1, C_2)$)
- Pick any band (r rows)
  - Prob. that all rows in band equal = t^r
  - Prob. that some row in band unequal = $1 - t^r$
- Prob. that no band identical = $(1 - t^r)^b$
- Prob. that at least 1 band identical = $1 - (1 - t^r)^b$

What *b* Bands of *r* Rows Gives You



At least one band identical

No bands identical

$$1 - (1 - t^r)^b$$

Some row of a band unequal

All rows of a band are equal

Topic 5：Recommender Systems (Chapter 11)

- Recommender systems
  - Content-based recommendation
  - Collaborative recommendation
    - User-user collaborative filtering
    - Item-item collaborative filtering
  - BellKor Recommender System (the idea)
    - ~~Matrix Factorization~~

	Avatar	LOTR	Matrix	Pirates
Alice	1		0.2	
Bob		0.5		0.3
Carol	0.2		1	
David				0.4

Final exam

- Final written exam (100 pts)
- Five questions in total on five topics
- Two hours
- Closed book exam
- If you are ill on the day of the exam, do not attend the exam – I will not accept any medical special consideration claims from people who already attempted the exam.

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

myExperience Survey

Thank you!