

Big Data Analytics

School of Computer Science and Engineering

Fairleigh Dickinson University – Vancouver

Ali Safari



Data contains value and knowledge

Data Mining, Data Science, ...

- But to extract the knowledge data needs to be
 - Stored (systems)
 - Managed (databases)
 - And ANALYZED ← this class

**Data Mining ≈ Predictive Analytics ≈
Data Science ≈ Machine Learning ≈
Data-Centric AI**

What This Course Is About

- Extraction of actionable information from (usually) very large datasets, is the subject of extreme hype, fear, and interest
- It's not all about machine learning
- But most of it is!
- Emphasis in this course is on algorithms that scale
 - Parallelization often essential

Data Mining/ML Pipeline

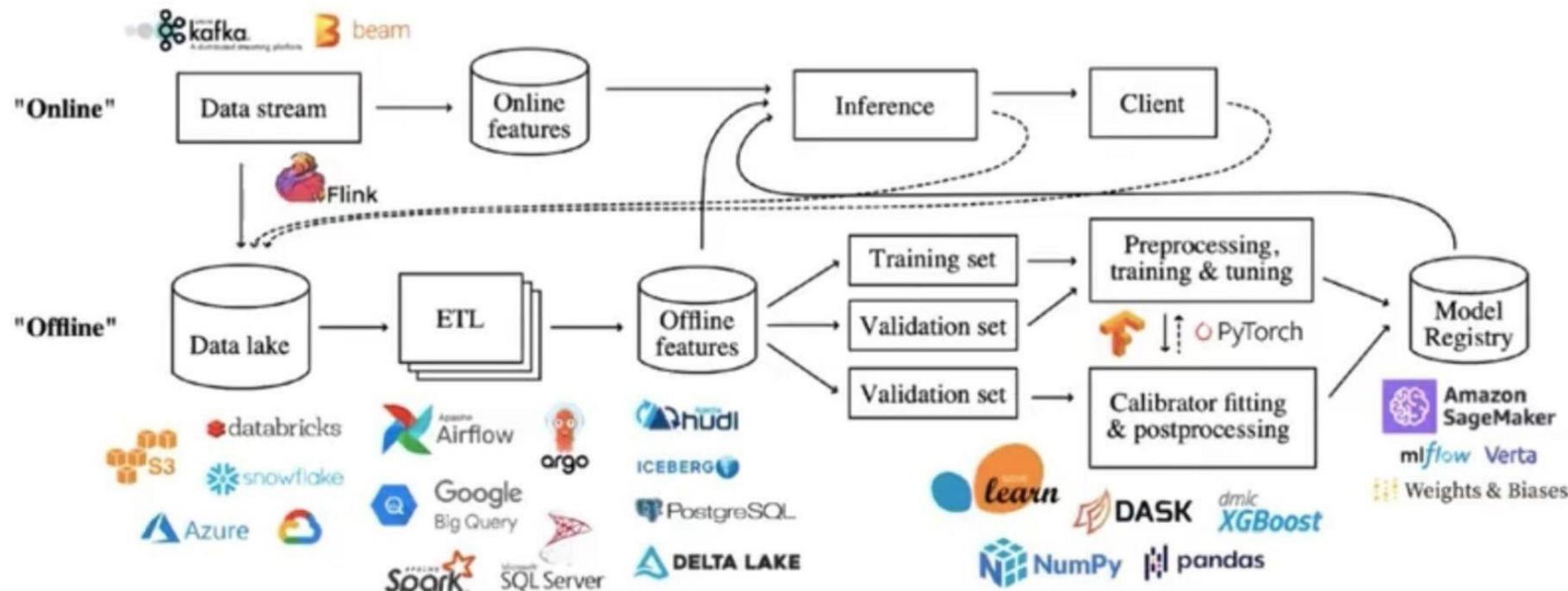


Figure 1: High-level architecture of a generic end-to-end machine learning pipeline. Logos represent a sample of tools used to construct components of the pipeline, illustrating heterogeneity in the tool stack. *Shankar et al. 2021*

What is Data Mining?

- Given lots of data
- Discover patterns and models that are:

Valid: hold on new data with some certainty

Useful: should be possible to act on the item

Unexpected: non-obvious to the system

Understandable: humans should be able to interpret the pattern

Data Mining Methods

■ Descriptive methods

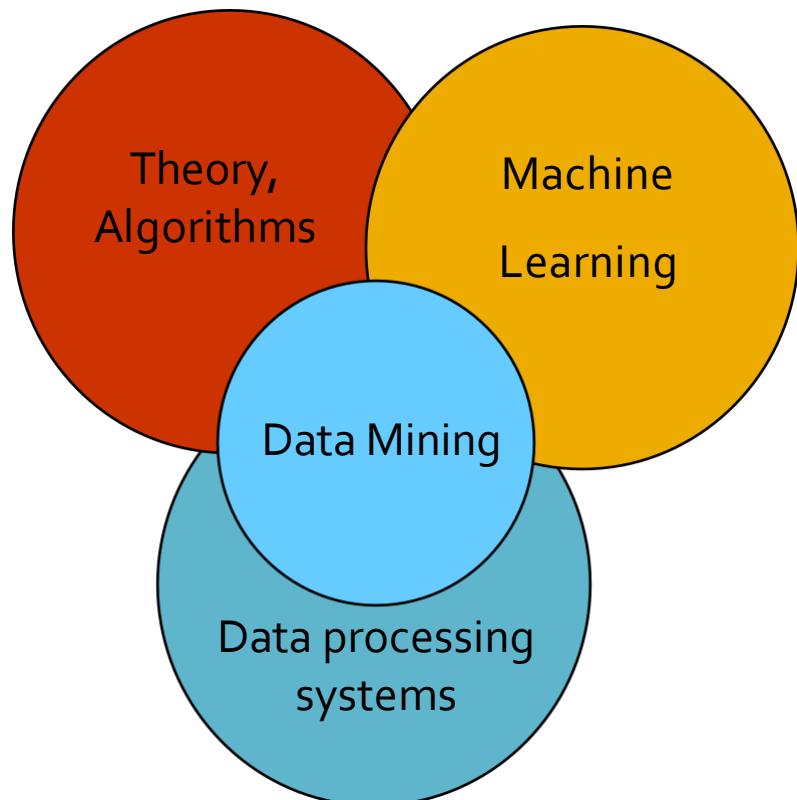
- Find human-interpretable patterns that describe the data
 - Example: Clustering

■ Predictive methods

- Use some variables to predict unknown or future values of other variables
 - Example: Recommender systems

This Class

- This combines best of machine learning, statistics, artificial intelligence, databases but more stress on
 - Scalability (big data)
 - Algorithms
 - Computing architectures
 - Automation for handling large data



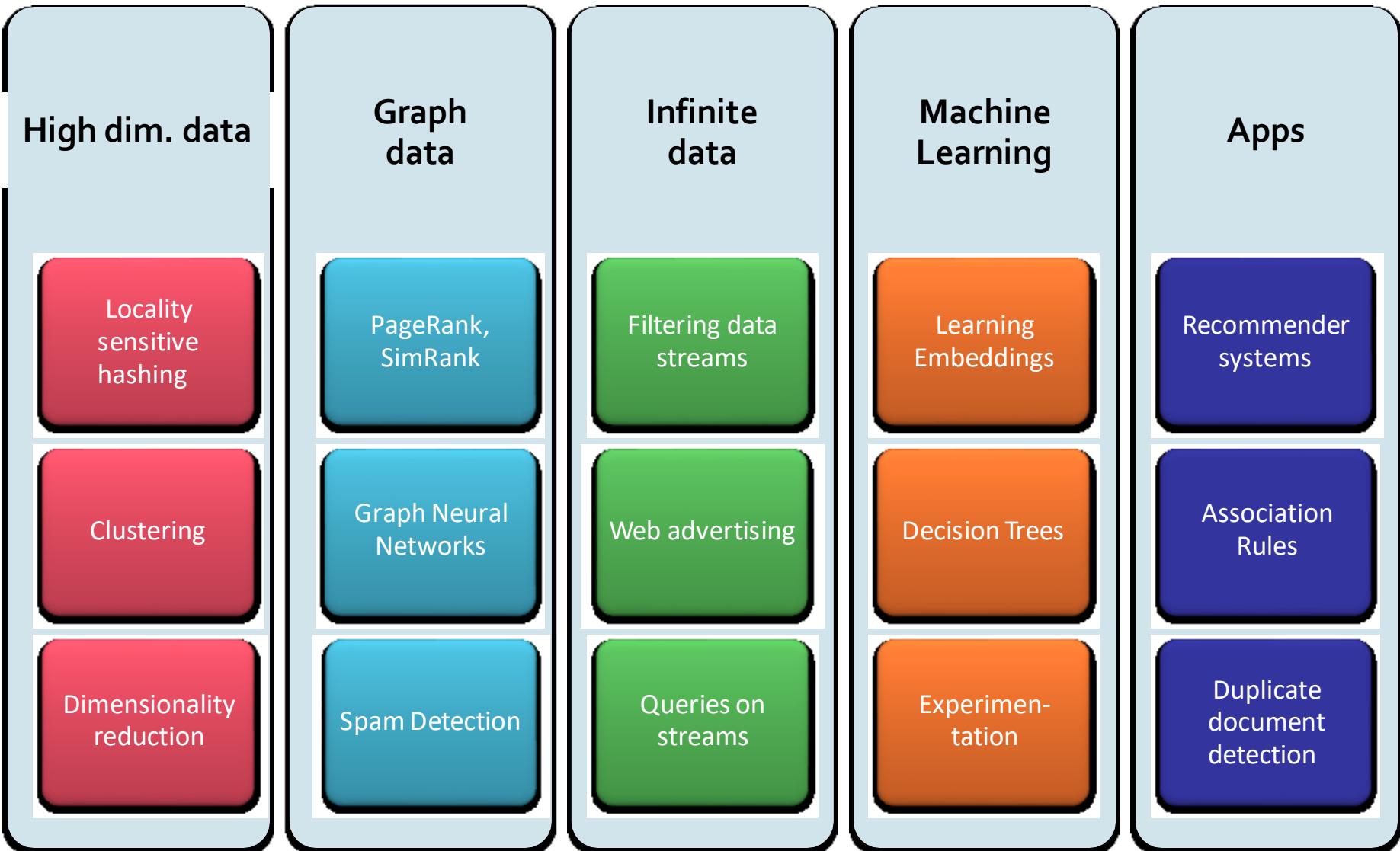
What will we learn?

- We will learn to **mine different types of data:**
 - Data is high dimensional
 - Data is a graph
 - Data is infinite/never-ending
 - Data is labeled
- We will learn to **use different models of computation:**
 - MapReduce
 - Streams and online algorithms
 - Single machine in-memory

What will we learn?

- We will learn to **solve real-world problems**:
 - Recommender systems
 - Market Basket Analysis
 - Spam detection
 - Data filtering
- We will learn **various “tools”**:
 - Linear algebra (SVD, Rec. Sys., Communities)
 - Optimization (stochastic gradient descent)
 - Dynamic programming (frequent itemsets)
 - Hashing (LSH, Bloom filters)

How the Class Fits Together





How do you want that data?

Distributed Computing for Data Mining



Large-scale Computing

- Large-scale computing for data mining problems on commodity hardware
- Challenges:
 - How do you distribute computation?
 - How can we make it easy to write distributed programs?
 - Machines fail:
 - One server may stay up 3 years (1,000 days)
 - If you have 1,000 servers, expect to lose 1/day
 - With 1M machines 1,000 machines fail every day!

An Idea and a Solution

- **Issue:**

Copying data over a network takes time

- **Idea:**

- Bring computation to data
 - Store files multiple times for reliability

- **Spark/Hadoop address these problems**

- **Storage Infrastructure – File system**

- Google: GFS. Hadoop: HDFS

- **Programming model**

- MapReduce
 - Spark

Storage Infrastructure

- **Problem:**
 - If nodes fail, how to store data persistently?
- **Answer:**
 - **Distributed File System**
 - Provides global file namespace
- **Typical usage pattern:**
 - Huge files (100s of GB to TB)
 - Data is rarely updated in place
 - Reads and appends are common

Distributed File System

- **Chunk servers**

- File is split into contiguous chunks
- Typically each chunk is 16-64MB
- Each chunk replicated (usually 2x or 3x)
- Try to keep replicas in different racks

- **Master node**

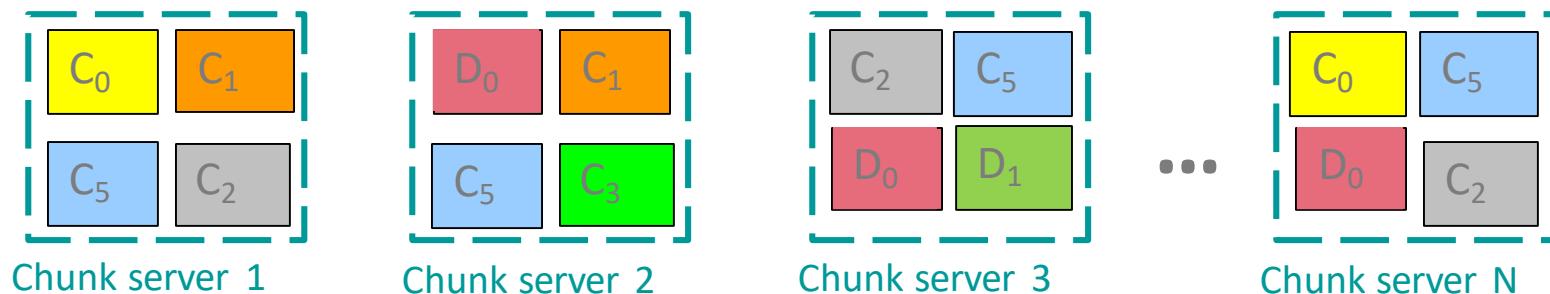
- a.k.a. Name Node in Hadoop's HDFS
- Stores metadata about where files are stored
- Master nodes are typically more robust to hardware failure and run critical cluster services.

- **Client library for file access**

- Talks to master to find chunk servers
- Connects directly to chunk servers to access data

Distributed File System

- Reliable distributed file system
- Data kept in “chunks” spread across machines
- Each chunk **replicated** on different machines
 - Seamless recovery from disk or machine failure



Notation: C₂... chunk no. 2 of file C

Bring computation directly to the data!

Chunk servers also serve as compute servers

MapReduce: Early Distributed Computing Programming Model

Programming Model: MapReduce

- MapReduce is a **style of programming** designed for:
 1. Easy parallel programming
 2. Invisible management of hardware and software failures
 3. Easy management of very-large-scale data
- It has several **implementations**, including Hadoop, Spark (used in this class), Flink, and the original Google implementation just called “MapReduce”

MapReduce: Overview

- Sequentially read a lot of data
- **Map:**
 - Extract something you care about
- **Group by key:** Sort and Shuffle
- **Reduce:**
 - Aggregate, summarize, filter or transform
- Write the result

Outline stays the same, **Map** and **Reduce** change to fit the problem

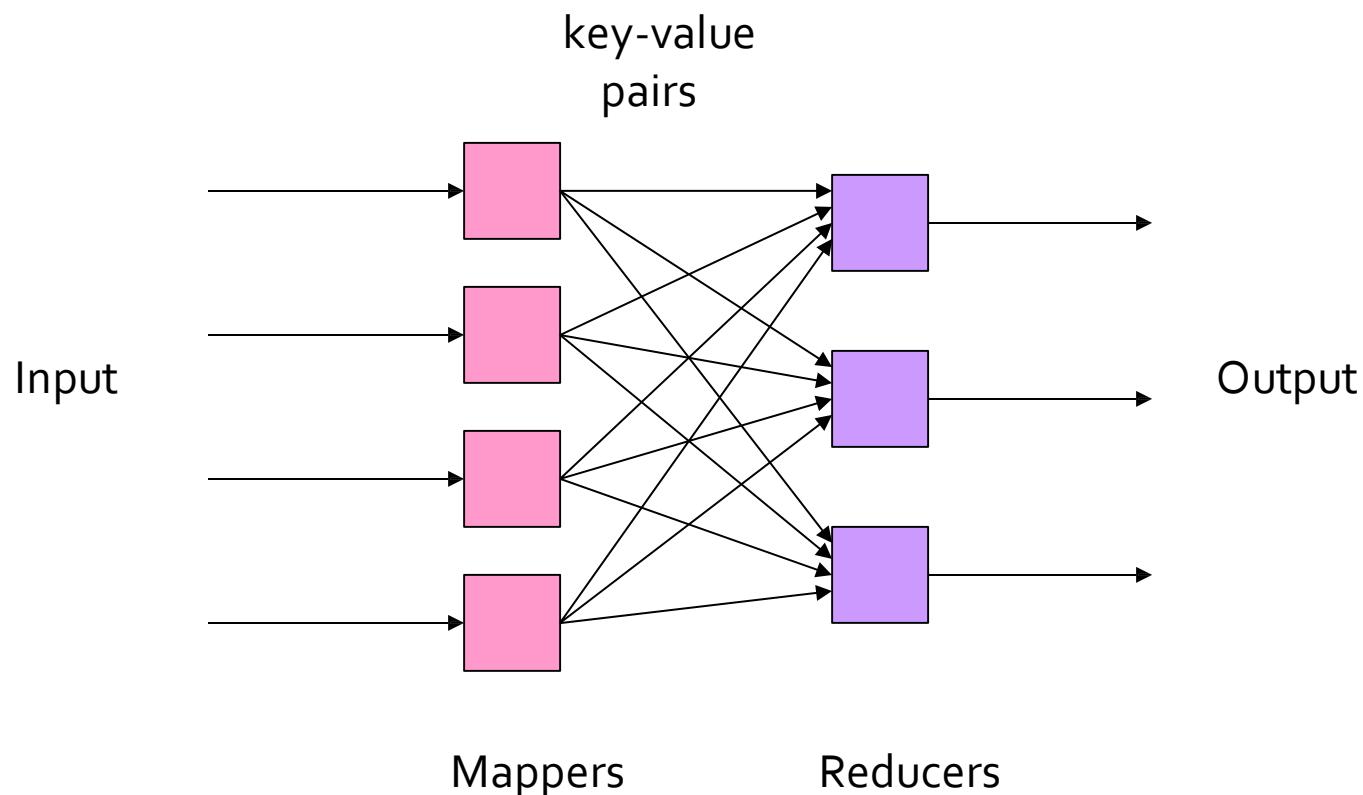
MapReduce: Overview

3 steps of MapReduce

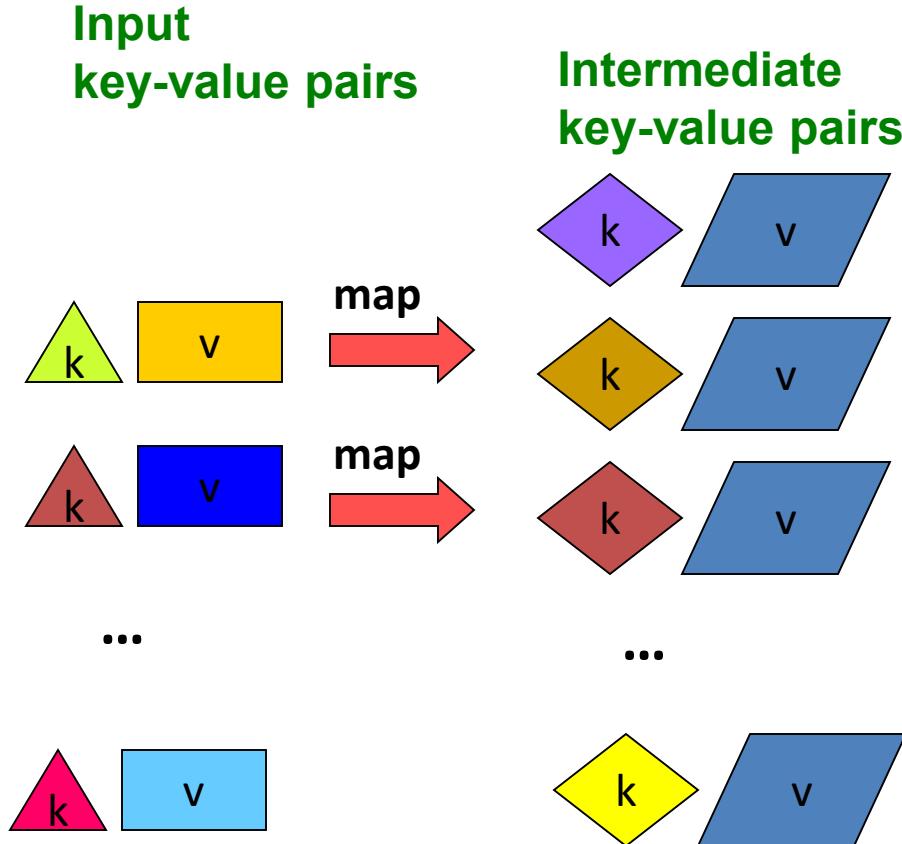
- **Map:**
 - Apply a user-written *Map function* to each input element
 - *Mapper* applies the Map function to a single element
 - Many mappers grouped in a *Map task* (the unit of parallelism)
 - The output of the Map function is a set of 0, 1, or more *key-value pairs*.
- **Group by key:** Sort and shuffle
 - System sorts all the key-value pairs by key, and outputs key-(list of values) pairs
- **Reduce:**
 - User-written *Reduce function* is applied to each key-(list of values)

Outline stays the same, **Map** and **Reduce** change to fit the problem

MapReduce Pattern

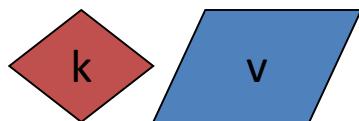
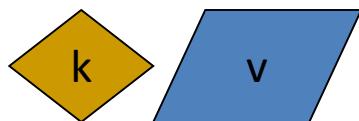
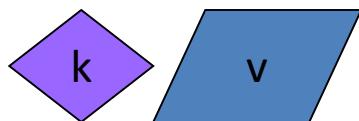


MapReduce: The Map Step

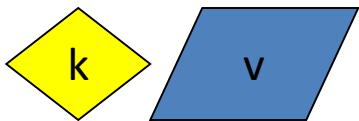


MapReduce: The Reduce Step

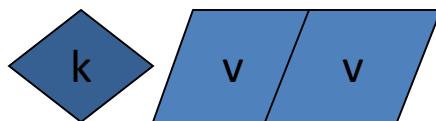
Intermediate key-value pairs



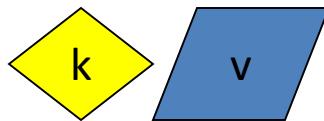
...



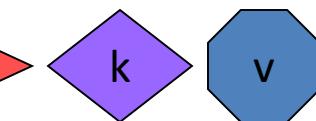
Key-value groups



...

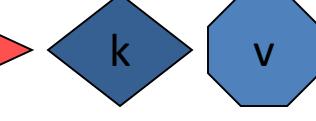


Output key-value pairs

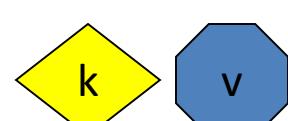


reduce

reduce



...



Example: Word Counting

Example MapReduce task:

- We have a huge text document
- Count the number of times each distinct word appears in the file
- **Many applications of this:**
 - Analyze web server logs to find popular URLs
 - Statistical machine translation:
 - Need to count number of times every 5-word sequence occurs in a large corpus of documents

Task: Word Count

Case 1:

- File too large for memory, but all <word, count> pairs fit in memory

Case 2:

- Count occurrences of words:
 - `words (doc.txt) | sort | uniq -c`
 - where `words` takes a file and outputs the words in it, one per a line
- Case 2 captures the essence of MapReduce
 - Great thing is that it is naturally parallelizable

More Specifically

- **Input:** a set of key-value pairs
- Programmer specifies two methods:
 - **Map(k, v) $\rightarrow <k', v'>^*$**
 - Takes a key-value pair and outputs a set of key-value pairs
 - E.g., key is the filename, value is a single line in the file
 - There is one Map call for every (k, v) pair
 - **Reduce(k' , $<v'>^*$) $\rightarrow <k', v''>^*$**
 - All values v' with same key k' are reduced together and processed in v' order
 - There is one Reduce function call per unique key k'

MapReduce: Word Counting

Data can be partitioned and processed in parallel

Provided by the programmer

MAP:

Read input and produces a set of key-value pairs

Group by key:

Collect all pairs with same key

Provided by the programmer

Reduce:

Collect all values belonging to the key and output

The crew of the space shuttle Endeavor recently returned to Earth as ambassadors, harbingers of a new era of space exploration. Scientists at NASA are saying that the recent assembly of the Dextre bot is the first step in a long term space based man/machine partnership. "The work we're doing now -- the robotics we're doing -- is what we're going to need

Big document

(The, 1)
(crew, 1)
(of, 1)
(the, 1)
(space, 1)
(shuttle, 1)
(Endeavor, 1)
(recently, 1)
....

(key, value)

(crew, 1)
(crew, 1)
(space, 1)
(the, 1)
(the, 1)
(the, 1)
(shuttle, 1)
(recently, 1)
....

(key, value)

(crew, 2)
(space, 1)
(the, 3)
(shuttle, 1)
(recently, 1)
....

(key, value)

Only sequential reads

Word Count Using MapReduce

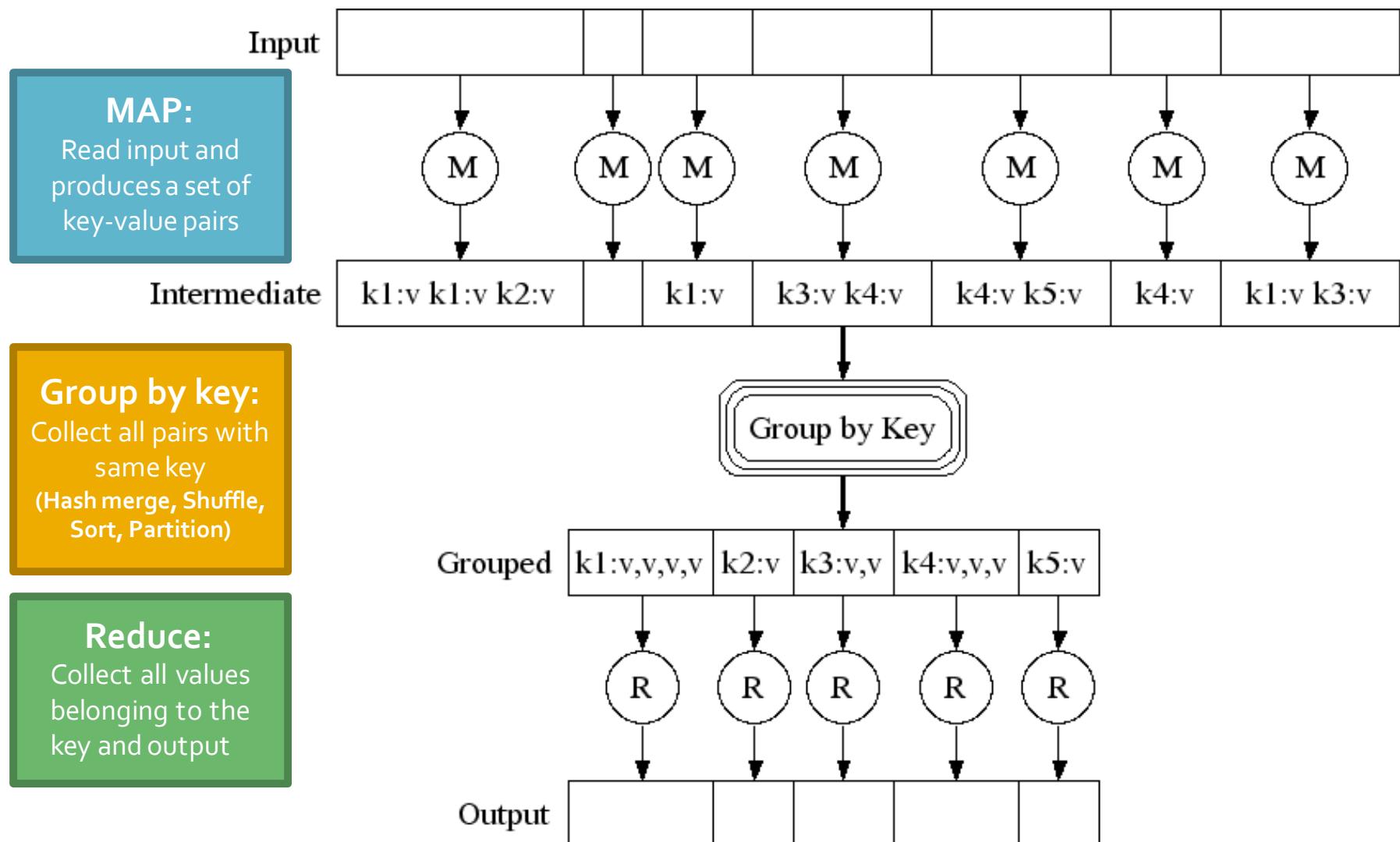
```
map(key, value) :
```

```
# key: document name; value: text of the document
for each word w in value:
    emit(w, 1)
```

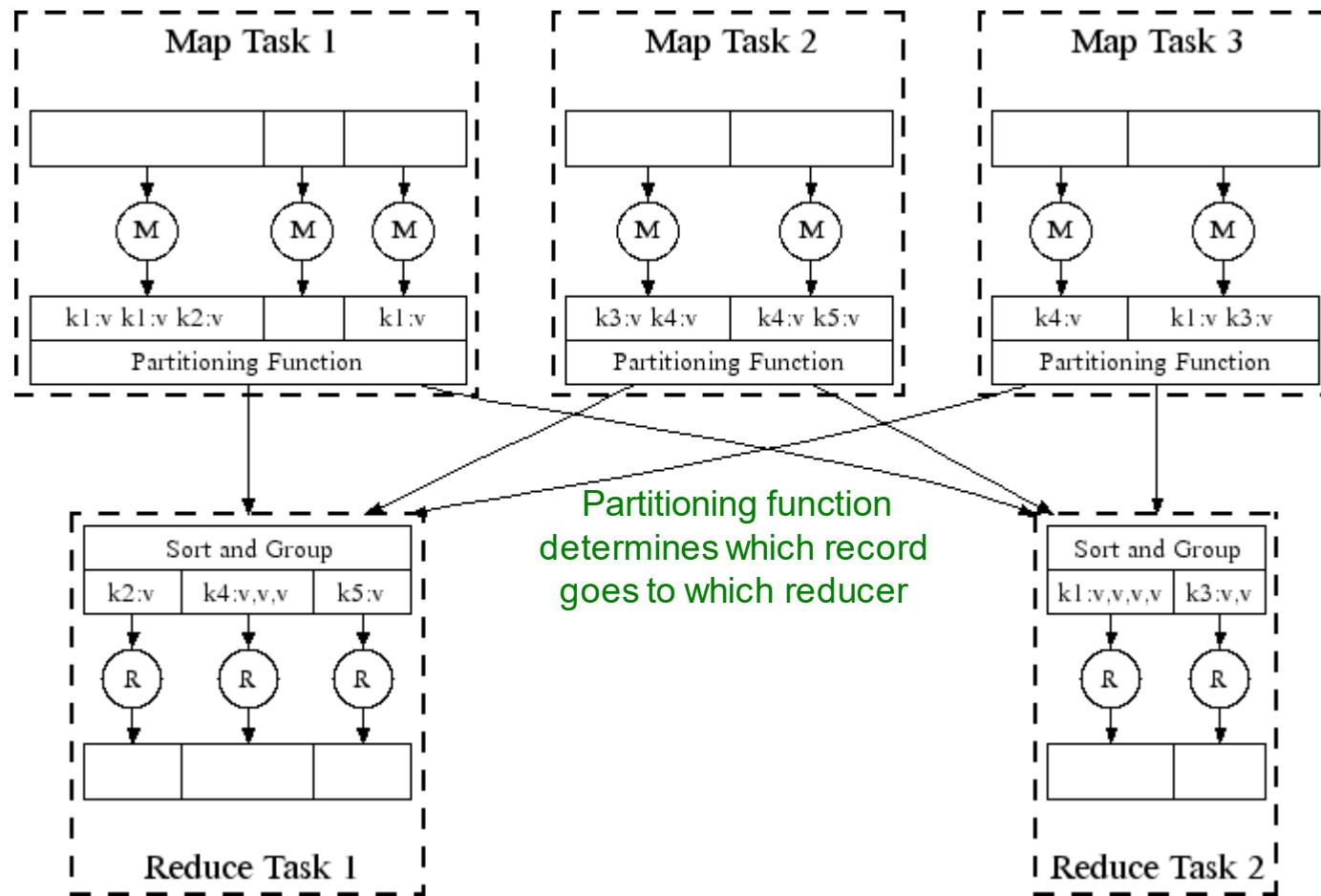
```
reduce(key, values) :
```

```
# key: a word; value: an iterator over counts
    result = 0
    for each count v in values:
        result += v
    emit(key, result)
```

Map-Reduce: A Diagram



Map-Reduce: In Parallel



Phases of Map-Reduced are distributed with many tasks
doing the work in parallel

MapReduce: Environment

MapReduce environment takes care of:

- Partitioning the input data
- Scheduling the program's execution across a set of machines
- Performing the group by key step
 - In practice this is the bottleneck
- Handling machine failures
- Managing required inter-machine communication

Dealing with Failures

■ Map worker failure

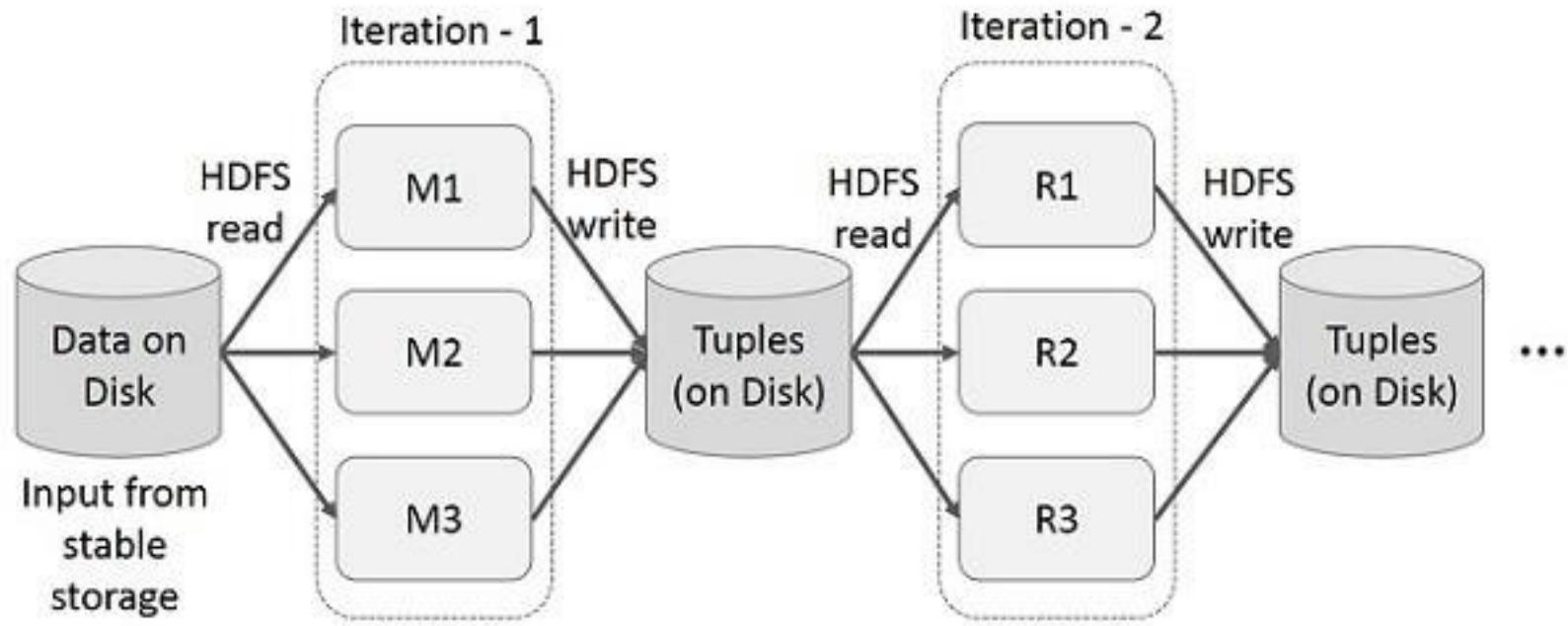
- Map tasks completed or in-progress at worker are reset to idle and rescheduled
- Reduce workers are notified when map task is rescheduled on another worker

■ Reduce worker failure

- Only in-progress tasks are reset to idle and the reduce task is restarted

Spark: Extends MapReduce

Problems with MapReduce



MapReduce incurs substantial overheads due to data replication, disk I/O, and serialization

- Outputs of mappers **M** are saved on the disk, sorted, and then read again by reducers **R** (HDFS read, HDFS write)

Problems with MapReduce

- **Two major limitations of MapReduce:**
 - Difficulty of programming directly in MapReduce
 - Many big data problems/algorithms aren't easily described as map-reduce
 - Performance bottlenecks, or batch not fitting the use cases
 - Saving to disk is typically much slower than in-memory work
- **In short, MapReduce doesn't compose well for large applications**
 - Many times, one needs to chain multiple map-reduce steps.

Data-Flow Systems

- **MapReduce uses two “ranks” of tasks:**
One for **Map** the second for **Reduce**
 - Data flows from the first rank to the second
- **Data-Flow Systems generalize this in two ways:**
 1. Allow any number of tasks/ranks
 2. Allow functions other than Map and Reduce
 - If data flow is in one direction only (DAG=directed acyclic graph), we can have the blocking property and allow recovery of tasks rather than whole jobs

Data Analytics Software Stack

Spark Streaming
Stream processing

GraphX
Graph computation

MLlib
User-friendly machine learning

SparkSQL
SQL API

Hive

Storm

MPI

Spark

Fast memory-optimized execution engine (Python/Java/Scala APIs)

Hadoop MR

Tachyon *Distributed Memory-Centric Storage System*

Hadoop Distributed File System (HDFS)

Mesos *Cluster resource manager, multi-tenancy*

Spark: Most Popular Data-Flow System

- Expressive computing system, not limited to the map-reduce model
- Additions to MapReduce model:
 - Fast data sharing
 - Avoids saving intermediate results to disk
 - Caches data for repetitive queries (e.g. for machine learning)
 - General execution graphs (DAGs=directed acyclic graph)
 - Richer functions than just map and reduce
- Compatible with Hadoop

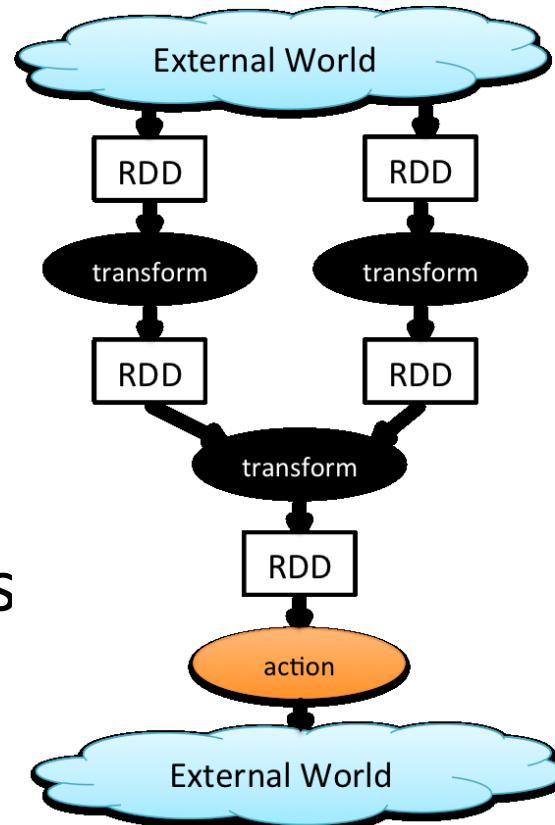
Spark: Overview

- **Key construct/idea:** Resilient Distributed Dataset (RDD)
- **Higher-level APIs:** DataFrames & DataSets
 - Introduced in more recent versions of Spark
 - Different APIs for aggregate data, which allowed to introduce SQL support

Spark: RDD

Key concept: *Resilient Distributed Dataset* (RDD)

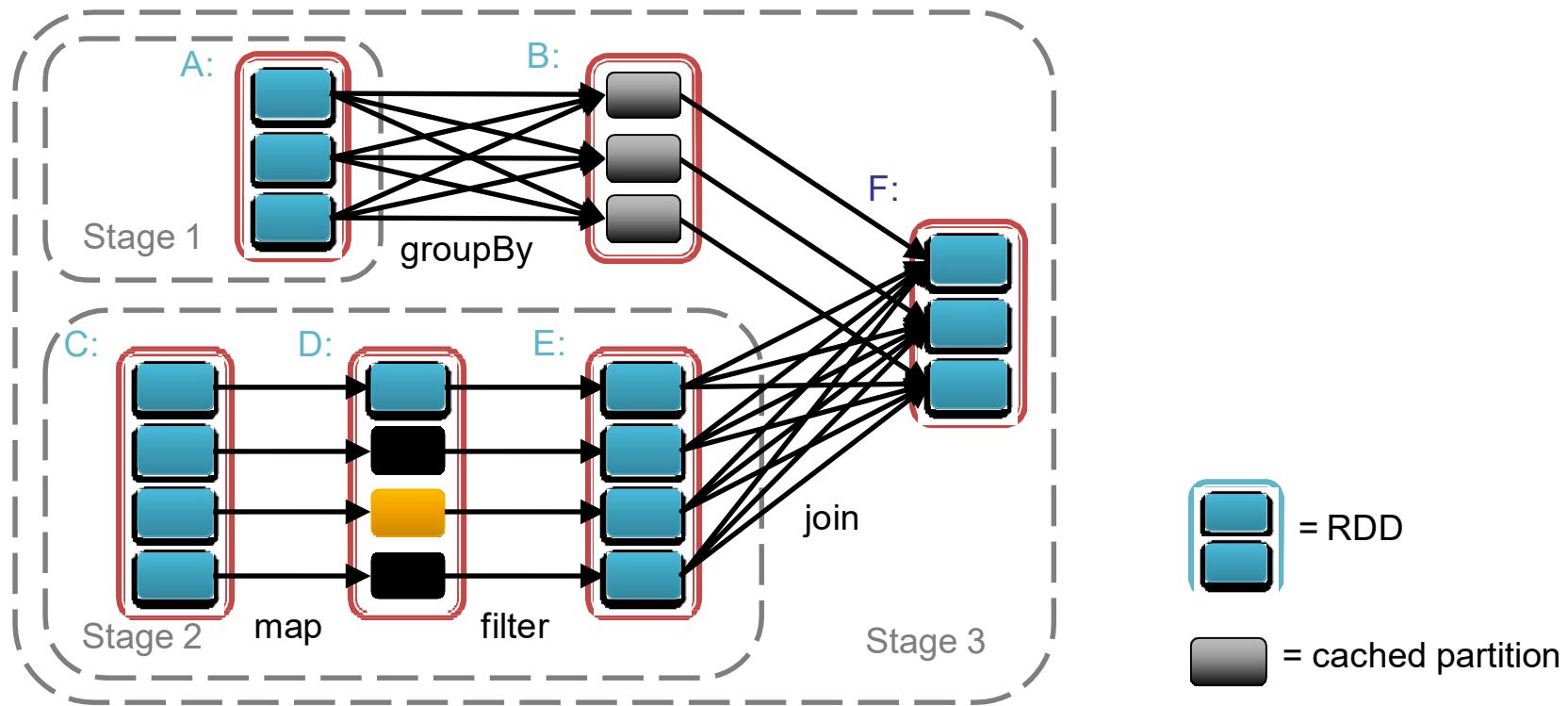
- Partitioned collection of records
 - Generalizes (key-value) pairs
- Spread across the cluster, Read-only
- Caching dataset in memory
 - Fallback to disk possible
- **RDDs** can be created from Hadoop, or by transforming other RDDs (you can stack RDDs)
- **RDDs** are best suited for applications that apply the same operation to all elements of a dataset



Spark RDD Operations

- **Transformations** build RDDs through deterministic operations on other RDDs:
 - Transformations include *map, filter, join, union, intersection, distinct*
 - **Lazy evaluation:** Nothing computed until an action requires it
- **Actions** to return value or export data
 - Actions include *count, collect, reduce, save*
 - Actions can be applied to RDDs; actions force calculations and return values

Task Scheduler: General DAGs



- Supports general directed acyclic task graphs
- Pipelines functions where possible
- Cache-aware data reuse & locality
- Partitioning-aware to avoid shuffles

Higher-Level API: DataFrame & Dataset

- **DataFrame:**

- Unlike an RDD, data organized into named columns, e.g. a **table in a relational database**.
- Imposes a structure onto a distributed collection of data, allowing higher-level abstraction

- **Dataset:**

- Extension of DataFrame API which provides **type-safe, object-oriented programming interface** (compile-time error detection)

Both built on Spark SQL engine. Both can be converted back to an RDD.

Useful Libraries for Spark

- Spark SQL
 - **scalable processing of relational data**
- Spark Streaming
 - **stream processing of live datastreams**
- MLlib
 - **scalable machine learning**
- GraphX
 - **graph manipulation**
 - Extends Spark RDD with a Graph abstraction: a directed multigraph with properties attached to each vertex and edge

Spark vs. Hadoop MapReduce

- Performance: **Spark is normally faster** but **with caveats**
 - Spark can process data in-memory; Hadoop MapReduce persists back to the disk after a map or reduce action
 - Spark generally outperforms MapReduce, but it **often needs lots of memory to perform well**; if there are other resource-demanding services or can't fit in memory, Spark degrades
 - MapReduce easily runs alongside other services with minor performance differences, & works well with the 1-pass jobs it was designed for
- Ease of use: **Spark is easier to program** (higher-level APIs)
- Data processing: **Spark more general**

Problems Suited for MapReduce

Example: Host size

- Suppose we have a large web corpus
- Look at the metadata file
 - Lines of the form: (URL, size, date, ...)
- For each host, find the total number of bytes
 - That is, the sum of the page sizes for all URLs from that particular host
- Other examples:
 - Link analysis and graph processing
 - Machine Learning algorithms

Example: Language Model

- **Statistical machine translation:**
 - Need to count number of times every 5-word sequence occurs in a large corpus of documents
- **Very easy with MapReduce:**
 - **Map:**
 - Extract (5-word sequence, count) from document
 - **Reduce:**
 - Combine the counts

Example: Join By Map-Reduce

- Compute the natural join $R(A,B) \bowtie S(B,C)$
- R and S are each stored in files
- Tuples are pairs (a,b) or (b,c)

A	B
a ₁	b ₁
a ₂	b ₁
a ₃	b ₂
a ₄	b ₃

R



B	C
b ₂	c ₁
b ₂	c ₂
b ₃	c ₃

S



A	C
a ₃	c ₁
a ₃	c ₂
a ₄	c ₃

Map-Reduce Join

- Use a hash function h from B-values to $1\dots k$
- A Map process turns:
 - Each input tuple $R(a,b)$ into key-value pair $(b,(a,R))$
 - Each input tuple $S(b,c)$ into $(b,(c,S))$
- Map processes send each key-value pair with key b to Reduce process $h(b)$
 - Hadoop does this automatically; just tell it what k is.
- Each Reduce process matches all the pairs $(b,(a,R))$ with all $(b,(c,S))$ and outputs (a,b,c) .

Problems NOT suitable for MapReduce

- **MapReduce is great for:**
 - Problems that require sequential data access
 - Large batch jobs (**not** interactive, real-time)
- **MapReduce is inefficient for problems where random (or irregular) access to data required:**
 - Graphs
 - Interdependent data
 - Machine learning
 - Comparisons of many pairs of items

Cost Measures for Algorithms

- In MapReduce we quantify the cost of an algorithm using
 1. *Communication cost* = total I/O of all processes
 2. *Elapsed communication cost* = max of I/O along any path
 3. (*Elapsed*) *computation cost* analogous, but count only running time of processes

Note that here the big-O notation is not the most useful
(adding more machines is always an option)

Example: Cost Measures

- For a map-reduce algorithm:
 - **Communication cost** = input file size + 2^* (sum of the sizes of all files passed from Map processes to Reduce processes) + the sum of the output sizes of the Reduce processes.
 - Q: Why is there a factor 2^* ?
 - **Elapsed communication cost** is the sum of the largest input + output for any map process, plus the same for any reduce process

What Cost Measures Mean

- Either the I/O (communication) or processing (computation) cost dominates
 - Ignore one or the other
- Total cost tells what you pay in rent from your friendly neighborhood cloud
- Elapsed cost is wall-clock time using parallelism

Cost of Map-Reduce Join of R and S

- **Total communication cost of joining R and S:**
 $= O(|R| + |S| + |R \bowtie S|)$
- **Elapsed communication cost = $O(s)$**
 - We're going to pick k and the number of Map processes so that the I/O limit s is respected
 - We put a limit s on the amount of input or output that any one process can have. **s could be:**
 - What fits in main memory
 - What fits on local disk
- With proper indexes, computation cost is linear in the input + output size
 - So, computation cost is like communication cost