

UMD DATA605 - Big Data Systems

Storing and Computing Big Data

MapReduce Framework

(Apache) Hadoop

Algorithms

MapReduce vs DBs

Dr. GP Saggese

gsaggese@umd.edu

with thanks to:

Alan Sussman

Amol Deshpande

Authors of www.mmnds.org

UMD DATA605 - Big Data Systems

Storing and Computing Big Data

MapReduce Framework

(Apache) Hadoop

Algorithms

MapReduce vs DBs

Resources

- Silberschatz: Chap 10
- Seminal papers
 - Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung: [The Google File System](#), 2003
 - Jeffrey Dean and Sanjay Ghemawat: [MapReduce: Simplified Data Processing on Large Clusters](#), 2004

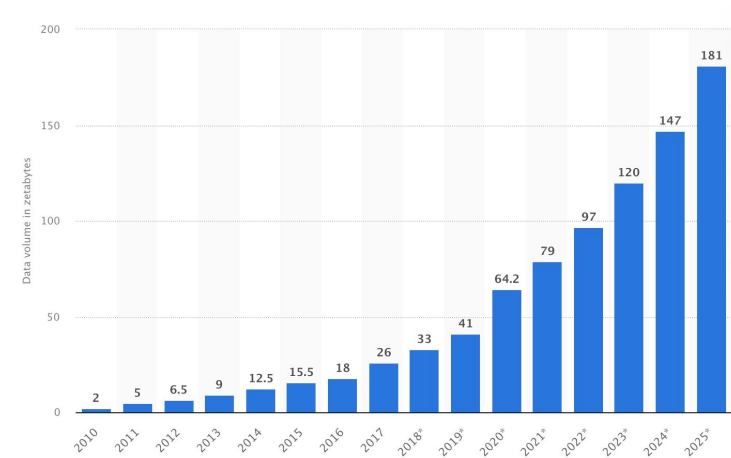
Big Data: Sources and Applications

- **Growth of World Wide Web in 1990s and 2000s**

- Storing and querying data much larger than enterprise data
- Extremely valuable data to target advertisements and marketing
 - Web server logs, web links
 - Social media
 - Data from mobile phone apps
 - Transaction data
 - Data from sensors / Internet of Things
 - Metadata from communication data

- **Big Data characteristics**

- Volume:
 - Amount of data to store and process is much larger than traditional DBs
 - Too big even for parallel DB systems with 10-100 machines
- Velocity
 - Store data at very high rate, due to rate of arrival
 - Data might be processed in real-time (e.g., streaming systems)
- Variety
 - Not all data is relational (e.g., semi-structured, textual, graph data)
- **Solution:** data is processed by systems with 10,000-100,000 machines



Volume of data in the world

Big Data: Sources and Applications

- **Web server logs**

- Record user interactions with web servers
- Billions of users click on thousands links per day → TB of data / day
- Contain important information to:
 - Decide what information (e.g., posts, news) to present to users to keep them engaged
 - E.g., what user has viewed, what other similar users has viewed
 - Understand visit patterns to make it easy for users to find information
 - Determine user preferences and trends to inform business decisions
 - Decide what advertisements to show to which users
 - Maximize benefit to the advertiser
 - Websites are paid for click-through or conversion

- **Click-through**

- A user clicks on an advertisement to get more information
- It is a measure of success in getting user attention / engagement

- **Conversion**

- When a user actually purchases the advertised product or service

Big Data: Storing and Computing

- **Two problems**

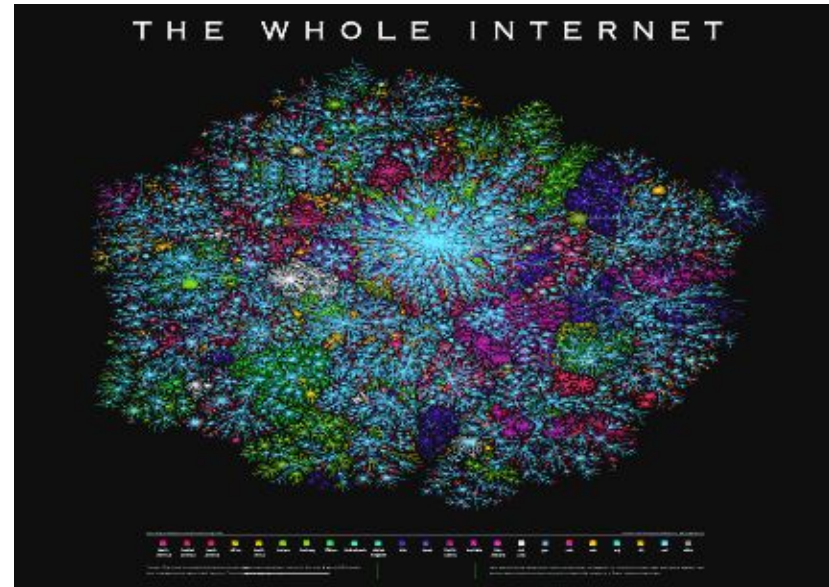
- Storing big data
- Computing big data

- **Need to be solved *together and efficiently***

- If one phase is slow → the entire system is slow

Processing the Web: Example

- The web has:
 - 20+ billion web pages
 - Total ~5m TBs = 5 ZB
 - ~1m hard drives to store the web
- One computer reads 300 MB/sec from disk
 - $5e6 * 1024 * 1024 * 8 / 300 / 3600 / 24 / 365$
= 4,433 years to read the web from one disk
- It takes even more to do something useful with the data!



Big Data: Storage Systems

- How to store big data?

1. Distributed file systems

- E.g., store large files like log files

2. Sharding across multiple DBs

- Partition records based on shard key across multiple systems

3. Parallel and distributed DBs

- Store data / perform queries across multiple machines
- Traditional relational DB interface

4. Key-value stores

- Data stored and retrieved based on a key
- Limitations on semantics, consistency, querying with respect to relational DBs
- E.g., NoSQL, Redis

1) Distributed File Systems

- **Distributed file system**

- = files stored across a number of machines, giving a single file-system view to clients
 - E.g., Google File System (GFS)
 - E.g., Hadoop File System (HDFS) based on GFS architecture
- Files are:
 - Broken into multiple blocks
 - Blocks are partitioned across multiple machines
 - Typically with some replication across machines

2) Sharding Across Multiple DBs

- **Sharding** = process of partitioning records across multiple DBs or machines
 - Shard keys
 - Aka partitioning keys / attributes
 - One or more attributes to partition the data
 - Range partition
 - Hash partition
- **Pros**
 - Scale beyond a centralized DB to handle more users, storage, processing speed
- **Cons**
 - Replication is often needed to deal with failures
 - Ensuring consistency is challenging
 - Relational DBs are not good at supporting constraints (e.g., foreign key) and transactions on multiple machines

3) Parallel and Distributed DBs

- **Parallel and distributed DBs**: store and process data running on multiple machines (aka "cluster")
- **Pros**
 - Programmer viewpoint
 - Traditional relational DB interface
 - Looks like a DB running on a single machine
 - Can run on 10s-100s of machines
 - Data is replicated for performance and reliability
 - Since failures are infrequent with 100s of machines, a query can be just restarted using a different machine
- **Cons**
 - Run a query incrementally requires a lot of complexity
 - Limit to the scalability

4) Key-value Stores

- **Problem**

- Many applications (e.g., web) store a very large number (billions or more) small records (few KBs to few MBs)
- File systems are not designed to store such a large number of files
- Relational DBs are not good at supporting constraints (e.g., foreign key) and transactions on multiple machines

- **Solution**

- Key-value stores / NoSQL systems
- Records are stored, updated, and retrieved based on a key
- Conceptually the operations are:
 - `put(key, value)` to store
 - `get(key)` to retrieve data

- **Pros**

- Partition data across multiple machines easily
- Support replication and consistency (no referential integrity)
- Balance workload and add more machines

- **Cons**

- Features are sacrificed to achieve scalability on large clusters
 - Declarative querying
 - Transactions
 - Retrieval based on non-key attributes

4) Parallel Key-value Stores

- **Parallel key-value stores**

- BigTable (Google)
- Apache HBase (open source version of BigTable)
- Dynamo, S3 (Amazon)
- Cassandra (Facebook)
- Azure cloud storage (Microsoft)
- Redis

- **Parallel document stores**

- Store data with certain format and execute simple queries
 - E.g., MongoDB accepts values in JSON format
- MongoDB cluster
 - Scale out on large data sizes and query / update loads with multiple machines
 - Partitioning is done based on the value of a specified attribute (partitioning attribute)
- Couchbase

- **In-memory caching systems**

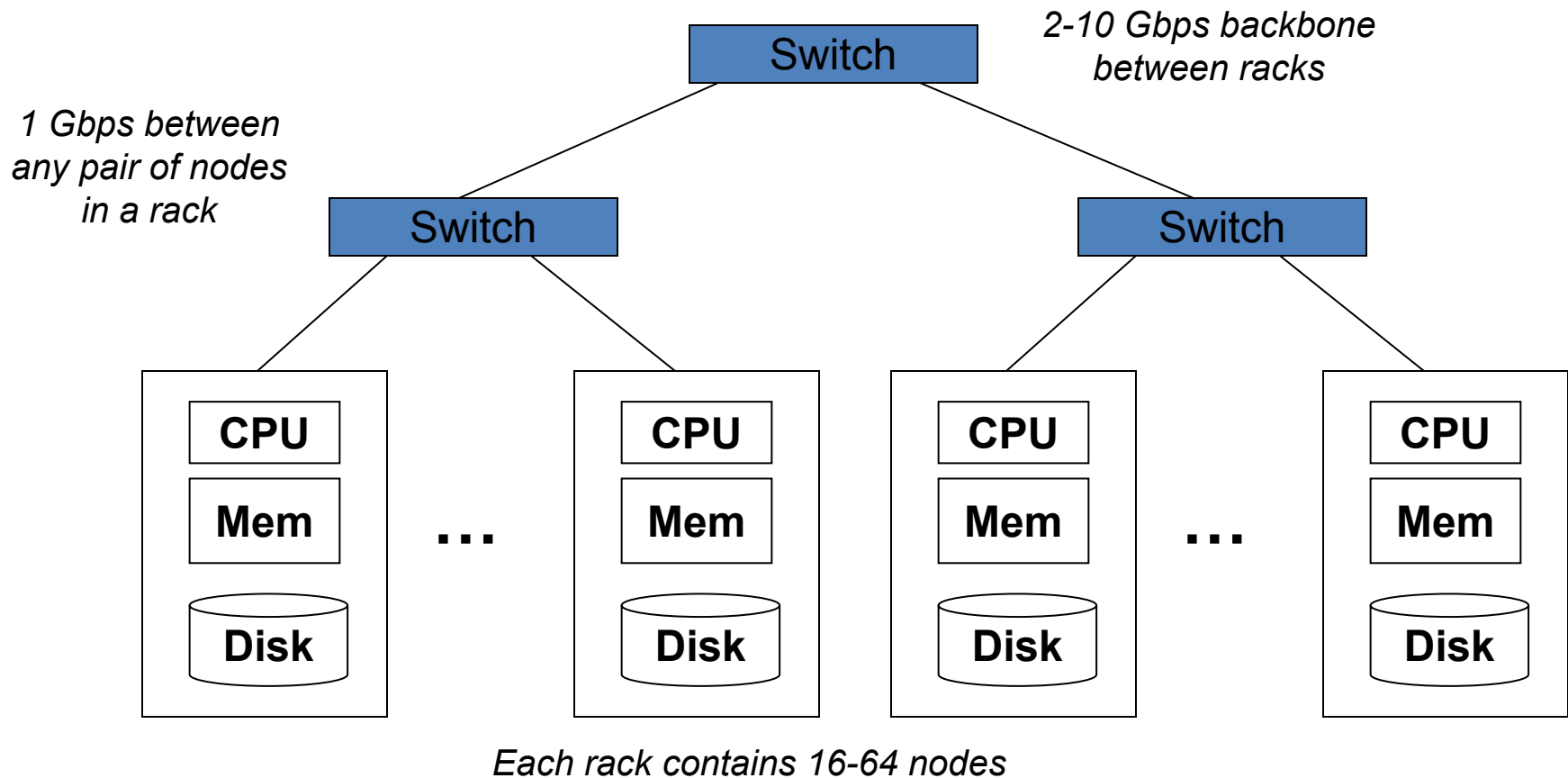
- Store some relations (or parts of relations) into an in-memory cache
- Replicated or partitioned across multiple machines
- E.g., memcached or Redis

MapReduce

- **How to process Big Data?**
- **Challenges**
 - How to distribute computation?
 - How can we make it easy to write distributed programs?
 - Distributed / parallel programming is hard
 - How to store data in a distributed system?
 - How to survive failures?
 - One server may stay up 3 years (1,000 days)
 - If you have 1,000 servers, expect to lose 1 / day
 - E.g., ~1M machines (Google in 2011) → 1,000 machines fail every day!
- **MapReduce**
 - Addresses these problems for certain kinds of computations
 - An elegant way to work with big data
 - Started as Google's data manipulation model
 - (But it wasn't an entirely new idea)

Cluster Architecture

- Today, a standard architecture for such problems has emerged:
 - Cluster of commodity Linux nodes
 - Commodity network (typically Ethernet) to connect them
 - In 2011 it was [guesstimated](#) that Google had 1M machines



Cluster Architecture: Network Bandwidth

- **Problem**

- Data is hosted on different machines in a cluster
- Copying data over a network takes time

- **Solutions**

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

- **MapReduce**

- Addresses both these problems
- Storage Infrastructure: distributed file system
 - Google: GFS
 - Hadoop: HDFS
- Programming model: MapReduce

Storage Infrastructure

- **Problem**

- How to store data *persistently* and *efficiently* when nodes can fail?

- **Typical data usage pattern**

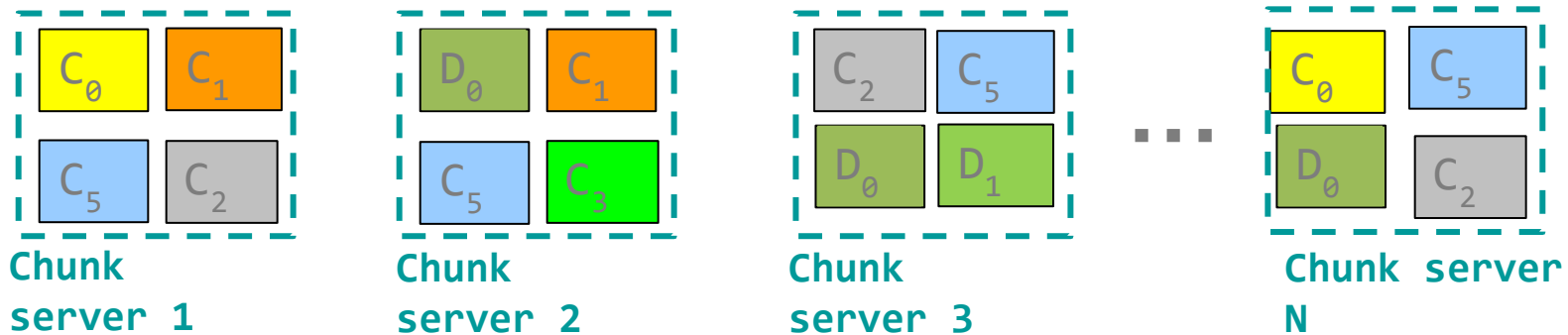
- Huge files (100s of GB to 1 TB)
- Reads and appends are common
- Data is rarely updated in place

- **Solution**

- Distributed file system
- Allow files to be stored across a number of machines
- Give a single file-system view to clients
- Files are:
 - Broken into multiple blocks
 - Partitioned across multiple machines
 - Typically with replication across machines

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



- Bring computation directly to the data
 - “Chunk servers” also serve as “compute servers”

Hadoop Distributed File System

- **NameNode**

- Store file / dir hierarchy
- Store metadata about files (e.g., where are stored)

- **DataNodes**

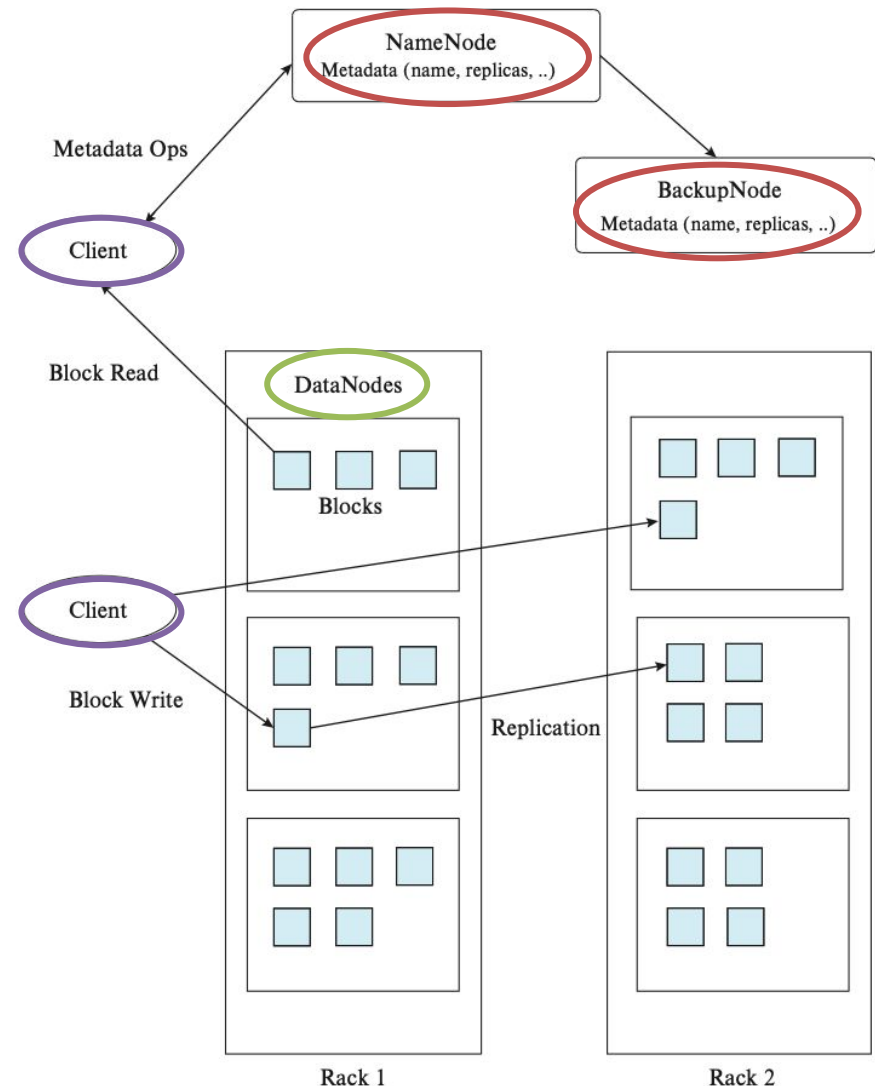
- Store data blocks
- File is split into contiguous 16-64MB blocks
- Each chunk is replicated (usually 2x or 3x)
- Try to keep replicas in different racks

- **Client**

- API (e.g., Python, Java) to library
- Mount HDFS on local filesystem

- **Library for file access**

- Read:
 - Talk to NameNode to find DataNode and pointer to block
 - Connect directly to DataNode to access data
- Write:
 - NameNode creates blocks
 - Assign blocks to several DataNodes
 - Client sends data to assigned DataNodes
 - DataNodes store data



MapReduce: Overview

- **MapReduce programming model**

- Inspired by functional programming (e.g., Lisp)
- Common pattern of parallel programming
- Algorithm
 - Given a large number of records to process
 - The same function `map()` is applied to each record
 - A form of aggregation `reduce()` is applied to the result of `map()`

- **Example**

- Goal: sum the length of all the tuples in a document
 - E.g., `[(), (a,) (a, b) (a, b, c)]`
- **map**(function, set of values)
 - Apply function to each value in the set (e.g., `len`)
`map(len, [(), (a), (a, b), (a, b, c)]) ⇒ (0 1 2 3)`
- **reduce**(function, set of values)
 - Combine all the values using a binary function (e.g., `add`)
`reduce(add, [1, 2, 3, 4, 5]) ⇒ 15`

MapReduce: Overview

- Structure of computation stays the same
 - **Read input**
 - Sequentially or in parallel
 - **Map**
 - Extract / compute something from records in the inputs
 - **Group by key**
 - Sort and shuffle
 - **Reduce**
 - Aggregate, summarize, filter, or transform
 - **Write the result**
- MapReduce framework (e.g., Hadoop, Spark) implements the general algorithm
- User specifies the map / reduce functions to solve the problem

MapReduce: Word Count

- **Word Count**

- “Hello world” of MapReduce
- We have a huge text file (so big you can’t keep it in memory)
- Count the number of times each distinct word appears in the file

- **Sample application**

- Analyze web server logs to find popular URLs

- **Linux solution**

- Example file from [https://en.wikipedia.org/wiki/Hot_Cross_Buns_\(song\)](https://en.wikipedia.org/wiki/Hot_Cross_Buns_(song))

```
> more doc.txt
```

```
One a penny, two a penny, hot cross buns.
```

```
> words(doc.txt) | sort | uniq -c
```

```
a 2
```

```
cross 1
```

```
...
```

- `words()` takes a file and outputs the words one per line
- This Unix pipeline is naturally parallelizable (in a MapReduce sense)

MapReduce: Word Count

Action

Read input

Map:

- Invoke `map()` on each input record
- Emit 0 or more output data items

Group by key:

- Gather all outputs from Map stage
- Collect outputs by keys

Reduce:

- Combine the list of outputs with same keys

Code

```
values = read(file_name)
```

```
def map(values):  
    # values: words in document  
    for word in values:  
        emit(word, 1)
```

```
def reduce(key, values):  
    # key: a word  
    # value: a list of counts  
    result = 0  
    for count in values:  
        result += count  
    emit(key, result)
```

Example

"One a penny, two a penny, hot cross buns."

Map:

```
[("one", 1), ("a", 1),  
 ("penny", 1), ("two", 1),  
 ("a", 1), ("penny", 1),  
 ("hot", 1), ("cross", 1),  
 ("buns", 1)]
```

Group by key:

```
[("a", [1, 1]),  
 ("buns", [1]),  
 ("cross", [1]),  
 ("hot", [1]),  
 ("one", [1]),  
 ("penny", [1, 1]),  
 ("two", [1])]
```

Reduce:

```
[("one", 1), ("a", 2),  
 ("penny", 2),  
 ("two", 1),  
 ("hot", 1),  
 ("cross", 1),  
 ("buns", 1)]
```

MapReduce: Word Count

Provided by the
programmer

Map:

Read input
Produce a set of
key-value pairs

Provided by the
programmer

Group by key:

Collect all pairs
with same key

Reduce:

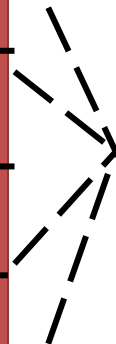
Collect all values
belonging to the
key and output

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

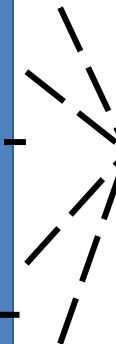
(The, 1)
(crew, 1)
(of, 1)
(the, 1)
(space, 1)
(shuttle, 1)
(Endeavor, 1)
(recently, 1)
....

(key, value)



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

(key, value)



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

(key, value)

MapReduce: Map Step

```
map(values: List):
```

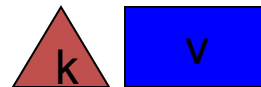
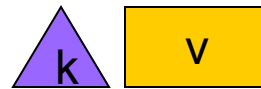
```
  # values: words in document
```

```
  for word in values:
```

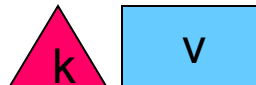
```
    emit(word, 1)
```

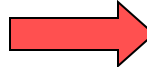
map needs to process all the values, but can output 0 or more tuples for each input

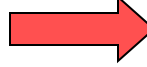
Input
key-value pairs



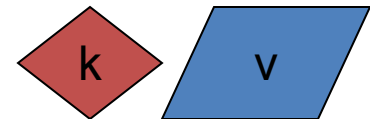
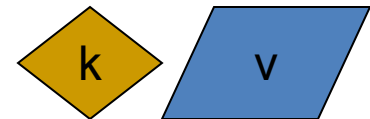
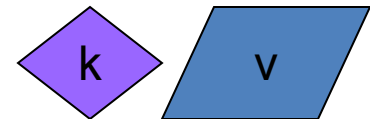
...



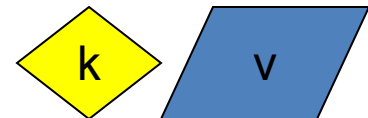
map


map


Intermediate
key-value pairs



...



MapReduce: Reduce Step

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

```
# key: a word
```

```
# value: an iterator over counts
```

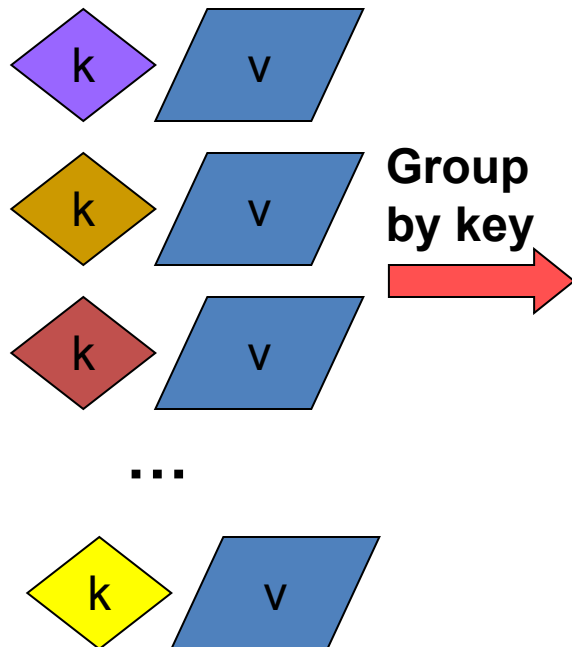
```
result = 0
```

```
for count in values:
```

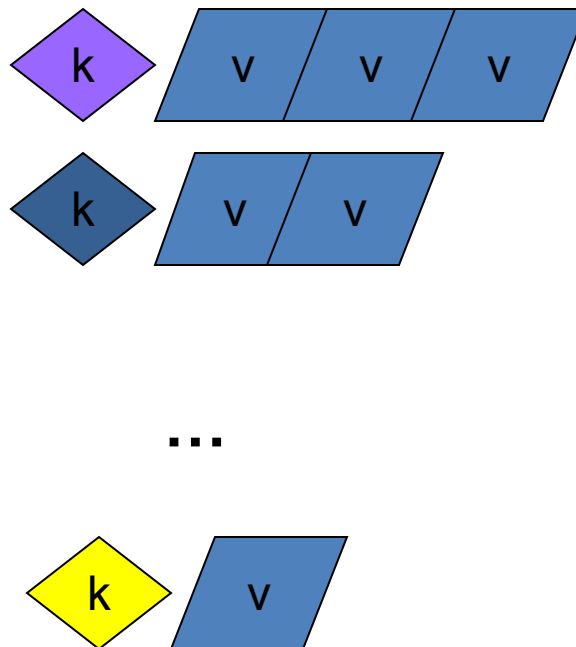
```
    result += count
```

```
emit(key, result)
```

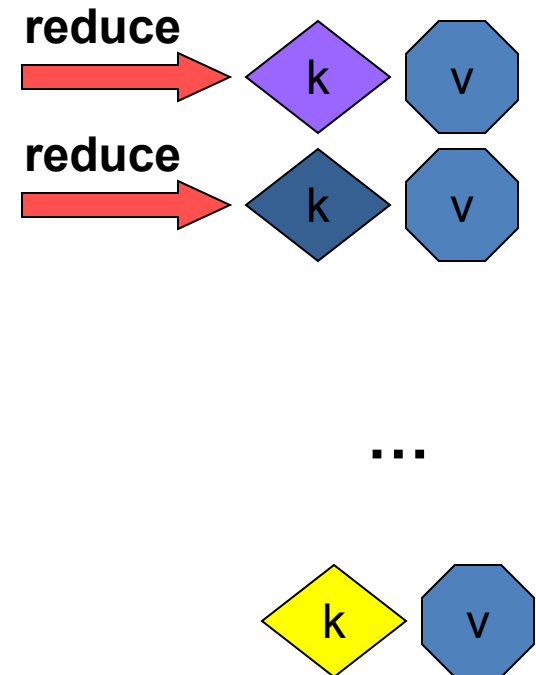
Intermediate
key-value pairs



Key-value groups



Output
key-value pairs



MapReduce: Interfaces

- Input: read a set of key-value pairs `List[Tuple[k, v]]`
- Programmer specifies two methods map and reduce

`Map(Tuple[k, v]) → List[Tuple[k, v]]`

- Take a key-value pair and outputs a set of key-value pairs
 - E.g., key is a file, value is the number of occurrences
 - “One a penny” → [(“One”, 1), (“a”, 1), (“penny”, 1)]
- There is one Map call for every (k, v) pair

`GroupBy(List[Tuple[k, v]]) → List[Tuple[k, List[v]]]`

- Group and optionally sorts all the records with the reduce key

`Reduce(List[Tuple[k, List[v]]]) → Tuple[k, v]`

- All values v' with same key k' are reduced together
- There is one Reduce call per unique key k'
- Output: write key-value pairs `List[Tuple[k, v]]`

MapReduce: Log Processing

- Log file recording access to a website with format

date, hour, filename

- **Goal:** find how many times each files is accessed during Feb 2013

- **Input**

- Read the file and split into lines

- **Map**

- Parse each line into the 3 fields
 - If the date is in the required interval
emit(dir_name, 1)

- **GroupBy**

- The reduce key is the filename
 - Accumulate all the (key, value) with the same filename

- **Reduce**

- Add the values for each list of (key, value) since they have the same filename
 - Output the number of access to each file

- **Output**

- Write results on disk separated by newline

After Input

```
...
2013/02/21 10:31:22.00EST /slide-dir/11.ppt
2013/02/21 10:43:12.00EST /slide-dir/12.ppt
2013/02/22 18:26:45.00EST /slide-dir/13.ppt
2013/02/22 18:26:48.00EST /exer-dir/2.pdf
2013/02/22 18:26:54.00EST /exer-dir/3.pdf
2013/02/22 20:53:29.00EST /slide-dir/12.ppt
...
```

After Map

```
[(`/slide-dir/11.ppt`, 1), ...]
```

After GroupBy

```
[(`/slide_dir/11.ppt`, 1), ...,
(`/slide-dir/12.ppt`, [1, 1]), ...]
```

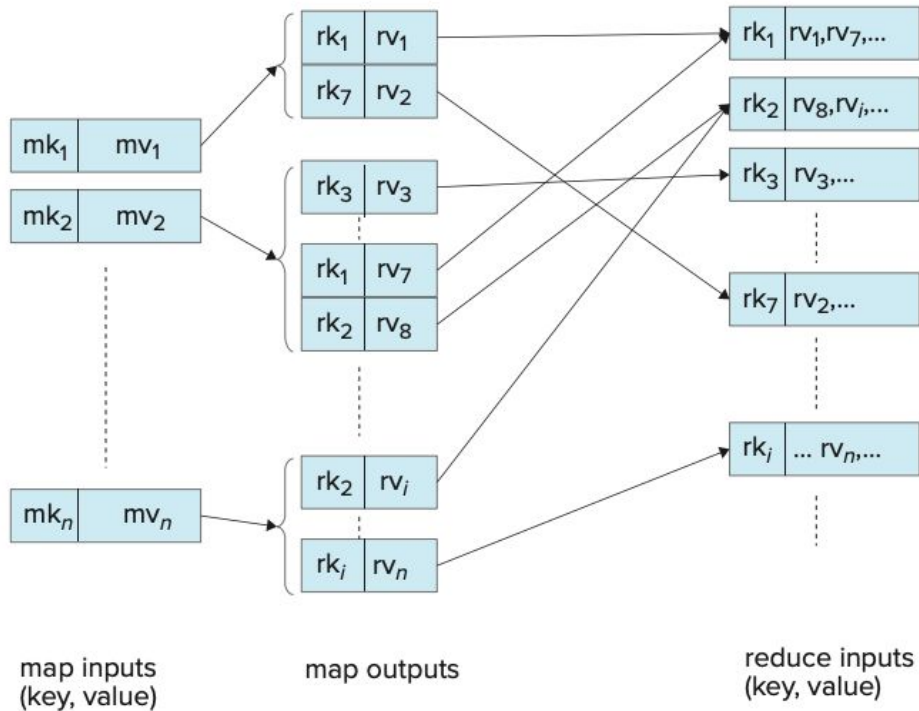
After Reduce

```
[(`/slide_dir/11.ppt`, 1), ...,
(`/slide-dir/12.ppt`, 2), ...]
```

Output

```
/slide_dir/11.ppt 1
...
/slide-dir/12.ppt 2
...
```

MapReduce: Data Flow



- **Input**

- **Map**

- mk_i = map keys
- mv_i = map input values

- **GroupBy**

- Shuffle / collect the data

- **Reduce**

- rk_i = reduce keys
- rv_i = reduce input values
- Reduce outputs are not shown

Input

Map

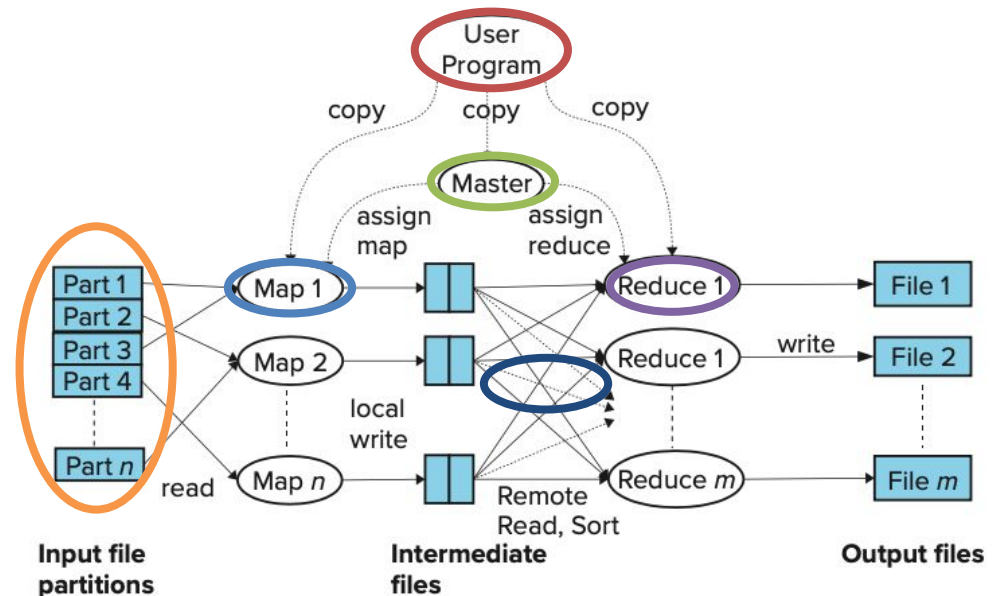
GroupBy

Reduce

- Focus is on MapReduce functionality / flow of the data to expose the parallelism

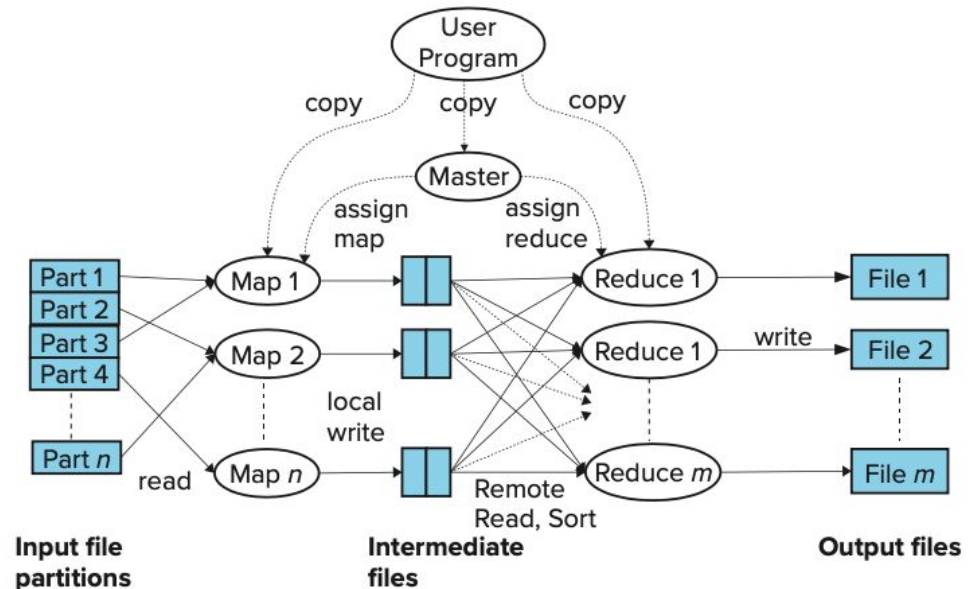
MapReduce: Parallel Data Flow

- So far focus on MapReduce functionality and flow of data
 - MapReduce enables parallel processing
- **User program** specifies map and reduce code
- **Input data** is partitioned across multiple machines (HDFS)
- **Master** node sends copies of the code to all computing nodes
- **Map**
 - n data chunks to process
 - Functions executed in parallel on multiple k machines
 - Each work on some part of the data
 - Output data from Map is saved on disk
- **GroupBy / Sort**
 - Output data from Map is sorted and partitioned based on reduce key
 - Different files are created for each Reduce task
- **Reduce**
 - Functions executed in parallel on multiple machines
 - Each work on some part of the data
 - Output data from Reduce is saved on disk
- **Write to disk**
 - All operations use HDFS as storage
 - Machines are reused for multiple computations (Map, GroupBy, Reduce) at different times



Master Node Responsibilities

- **Master node takes care of coordination**
 - Each task has status (idle, in-progress, completed)
 - Idle tasks get scheduled as workers become available
 - When a *Map task* completes, it sends the Master the location and sizes of its intermediate files
 - Master pushes this info to *Reduce tasks*
 - Reduce tasks become idle and can get scheduled
- **Master pings workers periodically to detect failures**



Dealing with Failures

- **Map worker failure**

- Failed map tasks are reset to idle (i.e., back in the queue for execution)
- Reduce workers are notified when task is rescheduled on another worker

- **Reduce worker failure**

- Only in-progress tasks are reset to idle
- Reduce task is restarted

- **Master failure**

- MapReduce task is aborted and client is notified

How many Map and Reduce jobs?

- M map tasks
- R reduce tasks
- N worker nodes
- Rule of thumb
 - $M \gg N$
 - Pros
 - Improve dynamic load balancing
 - Speed up recovery from worker failures
 - Cons
 - More communication between Master and Worker Nodes
 - Smaller files
 - $R > N$
 - Usually $R < M$
 - Output is spread across fewer files

Refinements: Backup Tasks

- **Problem**

- Slow workers significantly lengthen the job completion time
- Slow workers due to:
 - Older processor
 - Not enough RAM
 - Other jobs on the machine
 - Bad disks
 - OS thrashing / virtual memory hell

- **Solution**

- Near the end of Map / Reduce phase
 - Spawn backup copies of tasks
 - Whichever one finishes first “wins”

- **Result**

- Shorten job completion time

Refinement: Combiners

- **Problem**

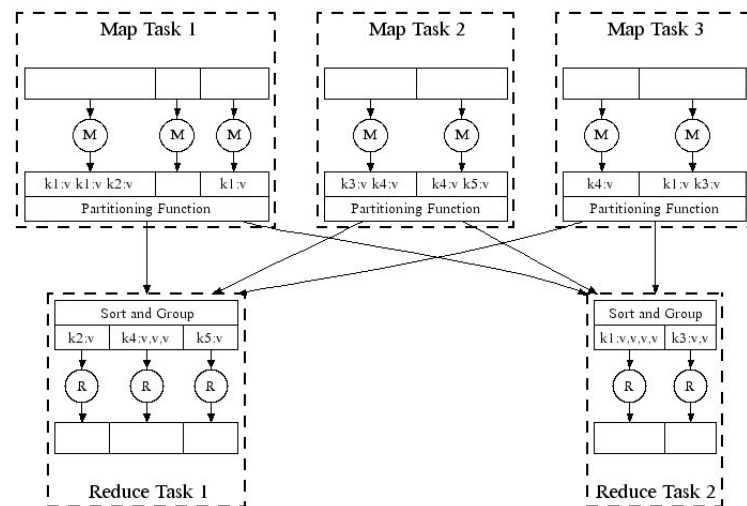
- Often a Map task produces many pairs for the same key k
 $[(k, v_1), (k, v_2), \dots]$
- E.g., common words in the word count example
- Increase complexity of the GroupBy stage

- **Solution**

- Pre-aggregate values in the Map with a Combine
 $[(k_1, (v_1, v_2, \dots)), k_2, (\dots)]$
- Combiner is usually the same as the Reduce function
- Works only if Reduce function is commutative and associative

- **Result**

- Better data locality
- Less shuffling and reordering
- Less network / disk traffic



Refinement: Partition Function

- **Problem**

- Sometimes user wants to control how keys get partitioned
- Inputs to Map tasks are created by contiguous splits of input file
- MapReduce uses a default partition function

$\text{hash}(\text{key}) \bmod R$

- Reduce needs to ensure that records with the same intermediate key end up at the same worker

- **Solution**

- Sometimes useful to override the hash function:
- E.g., $\text{hash}(\text{hostname}(\text{URL})) \bmod R$ ensures URLs from a host end up in the same output file

UMD DATA605 - Big Data Systems

MapReduce Framework

(Apache) Hadoop

Algorithms

MapReduce vs DBs

Implementations of MapReduce

- **Google**
 - Not available outside Google
- **Hadoop**
 - [Website](#)
 - An open-source implementation in Java
 - Uses HDFS for stable storage
 - Hadoop Wiki
 - [Introduction](#)
 - [Getting Started](#)
 - [Map/Reduce Overview](#)
- **Amazon Elastic MapReduce (EMR)**
 - [Website](#)
 - Hadoop MapReduce running on Amazon EC2
 - Can also run Spark, HBase, Hive, ...
- **Spark**
- **Dask**

MapReduce: Hadoop

- Hadoop is an open-source implementation of MapReduce



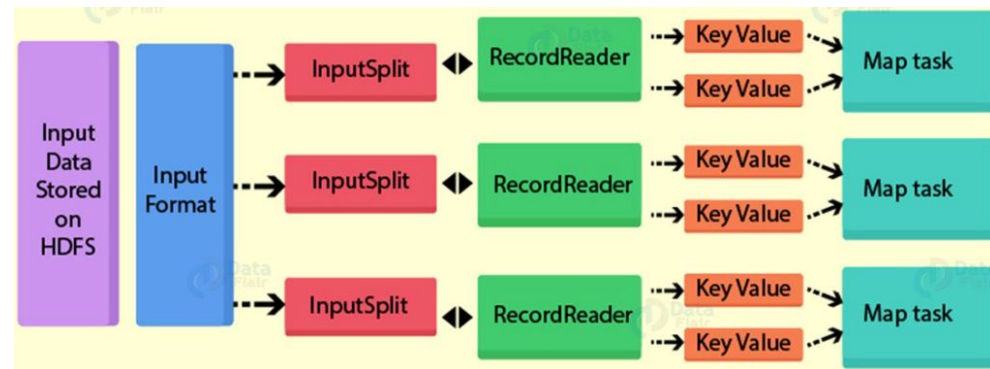
- Functionalities
 - Partition the input data (HDFS)
 - Input adapters
 - E.g., HBase, MongoDB, Cassandra, Amazon Dynamo
 - Schedule program's execution across a set of machines
 - Handle machine failures
 - Manage required inter-machine communication
 - Perform the GroupByKey step
 - Output adapters
 - E.g., Avro, ORC, Parquet
 - Schedule multiple MapReduce jobs

Data Flow

- Input, intermediate, final output are stored in a distributed file system (HDFS)
- Adapters to read / partition the data in chunks
- Scheduler tries to schedule map tasks “close” to physical storage location of input data
- Intermediate results (e.g., GroupBy) are stored on local FS of Map and Reduce workers
- Output is often input to another MapReduce task

Input Data

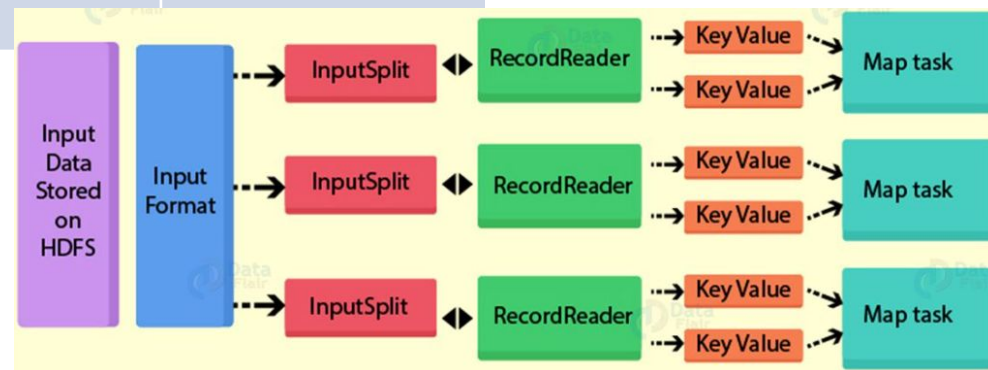
- **InputData** stores the data for a **MapTask** typically in a distributed file system (e.g., HDFS)
- The format of input data is arbitrary
 - Line-based log files
 - Binary files
 - Multi-line input records
 - Something else
 - E.g., an SQL database



InputFormat

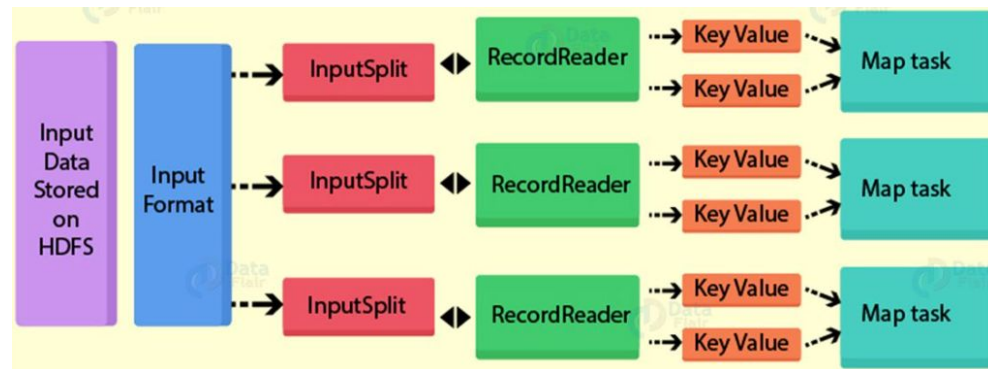
- **InputFormat** class reads and splits up the input files
 - Select the files that should be used for input
 - Defines the **InputSplits** that break a file
 - Provides a factory for **RecordReaders** objects that read the file

InputFormat	Description	Key	Value
TextInputFormat	Default format; reads lines of text files	The byte offset of the line	The line contents
KeyValueInputFormat	Parses lines into (K, V) pairs	Everything up to the first tab character	The remainder of the line
SequenceFileInputFormat	A Hadoop-specific high-performance binary format	User-defined	User-defined



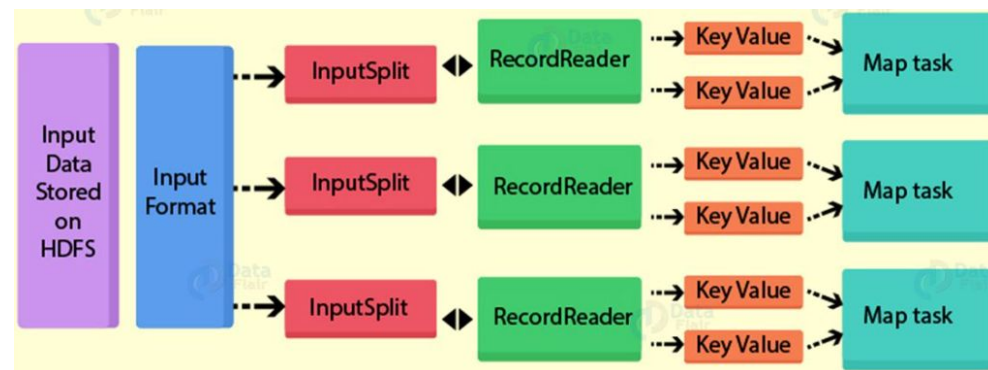
InputSplit

- **InputSplit** describes a unit of work that comprises a single **MapTask**
 - By default, the **InputFormat** breaks a file up into 64MB splits
- By dividing the file into splits
 - Each **MapTask** corresponds to a single input split
 - Several **MapTasks** to operate on a single file in parallel



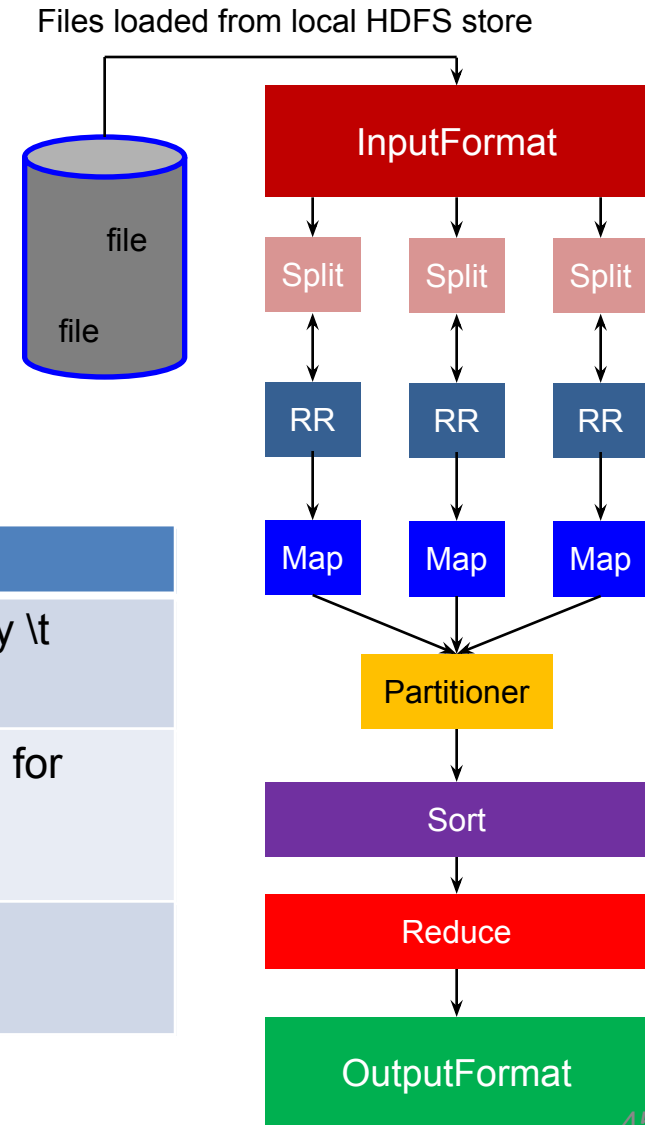
RecordReader

- The **InputSplit** defines a slice of work but does not describe how to access it
- The **RecordReader** class
 - Loads data from its source and converts it into **(K, V) pairs** suitable for reading by **MapTasks**
 - Is invoked repeatedly on the input until the entire **InputSplit** is consumed
 - Each invocation leads to a call of the map function defined by the programmer



OutputFormat

- The **OutputFormat** class
 - defines the way (K,V) pairs produced by **Reducers** are written to output files
 - write to files on the local disk or in HDFS in different formats



OutputFormat	Description
TextOutputFormat	Default; writes lines in "key \t value" format
SequenceFileOutputFormat	Writes binary files suitable for reading into subsequent MapReduce jobs
NullOutputFormat	Generates no output files

UMD DATA605 - Big Data Systems

MapReduce Framework

(Apache) Hadoop

Algorithms

MapReduce vs DBs

MapReduce: Applications

- Major classes of applications
 - Text tokenization, indexing, and search
 - Processing of large data structures
 - Data mining and machine learning
 - Link analysis and graph processing

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

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 = count running time of processes
- In this case, the big-O notation is not the most useful
 - Multiplicative constant matters
- Adding more machines is always an option
- Total cost tells what you pay in rent from your friendly neighborhood cloud provider
- Elapsed cost is wall-clock time using parallelism
- Either the I/O (communication) or processing (computation) cost dominates -> ignore one or the other

Example: Cost Measures

- **For a map-reduce algorithm:**
 - **Communication cost =**
 - input file size
 - $+ 2 \times$ (sum of the sizes of all files passed from Map processes to Reduce processes)
 - You need to write and read back the data
 - $+ \text{the sum of the output sizes of the Reduce processes}$
 - **Elapsed communication cost** is the sum of the largest input $+$ output for any map process, plus the same for any reduce process

Cost of Map-Reduce Join

- **Total communication cost**
 $= O(|R| + |S| + |R \bowtie S|)$
- **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

UMD DATA605 - Big Data Systems

MapReduce Framework

(Apache) Hadoop

Algorithms

MapReduce vs DBs

History

- Abstract ideas have been known before
 - See [MapReduce and Parallel DBMSs](#) by Stonebraker et al., 2010
 - Can be implemented using user-defined aggregates in PostgreSQL quite easily
 - Declarative design
 - User specifies what is to be done, not how many machines to use, etc...
- The strength of MapReduce comes from simplicity and ease of use
- No database system can come close to the performance of MapReduce infrastructure
- E.g., RDBMSs
 - Can't scale to that degree
 - Are not as fault-tolerant
 - Designed to support ACID
 - Databases were designed to support it
 - Most MapReduce applications don't care about ACID consistency

History

- MapReduce
 - Is very good at what it was designed for
 - If the application maps well to MapReduce, can achieve optimal theoretical speed-up
 - May not be ideal for more complex tasks
 - E.g., no notion of “query optimization”, e.g., operator order optimization
 - The sequence of MapReduce tasks makes it procedural within a single machine
- Joins are tricky to do
 - MapReduce assumes a single input
- Much work in recent years on extending the basic MapReduce functionality
 - E.g., Spark

Example: Join By Map-Reduce

- Compute the natural join $R(A, B) \bowtie S(B, C)$
- R and S are each stored in files as pairs (a, b) or (b, c)
- Use a hash function h from B -values to $1 \dots k$
- Map task
 - Transform an input tuple $R(a, b)$ into key-value pair $(b, (a, R))$
 - Each input tuple $S(b, c) \rightarrow (b, (c, S))$
- GroupBy task
 - Each key-value pair with key b is sent to Reduce task $h(b)$
 - Hadoop does this automatically; just tell it what k is
- Reduce task
 - Matches all the pairs $(b, (a, R))$ with all $(b, (c, S))$ to get (a, b, c)
 - Output (a, c)

R	A	B	\bowtie
	a_1	b_1	
	a_2	b_1	
	a_3	b_2	
	a_4	b_3	

S	B	C	=
	b_2	c_1	
	b_2	c_2	
	b_3	c_3	

A	C
a_3	c_1
a_3	c_2
a_4	c_3

Hadoop Ecosystem (aka Zoo)

- Pig
 - High-level data-flow language and execution framework for parallel computation
- HBase
 - Scalable, distributed database
 - Supports structured data storage for large tables (like Google BigTable)
- Cassandra
 - Scalable multi-master database with no single points of failure
- Hive
 - Data warehouse infrastructure
 - Provide data summarization and ad-hoc querying
- ZooKeeper
 - High-performance coordination service for distributed applications
- YARN, Kafka, Storm, Spark, Solr, ...

