Cloud Computing on Big Data with R

David Li

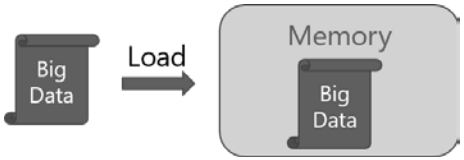
Is R so far so good?

- Pros
  - Programming language and software environment for statistical computing for small data size on single machine.
  - More than 11000 packages on CRAN, Github, Bioconductor, Bitbucket, etc.
  - Cross-platform, functional, graphical, etc.

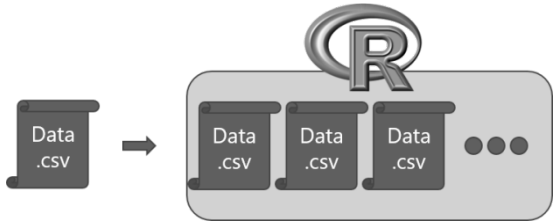
Is R so far so good?

- Cons
  - R is slow when data size is too big (how big is big?)
  - Your computer's hardware itself becomes the bottleneck.
  - Deficiency in execution mechanisms.

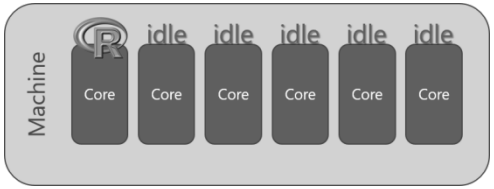
In-memory operation



Expensive data movement and duplication

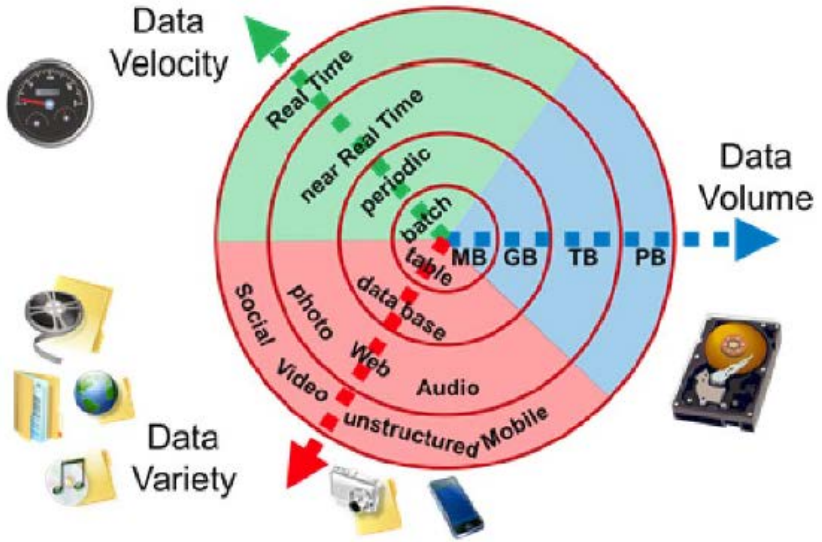


Lack of parallelism



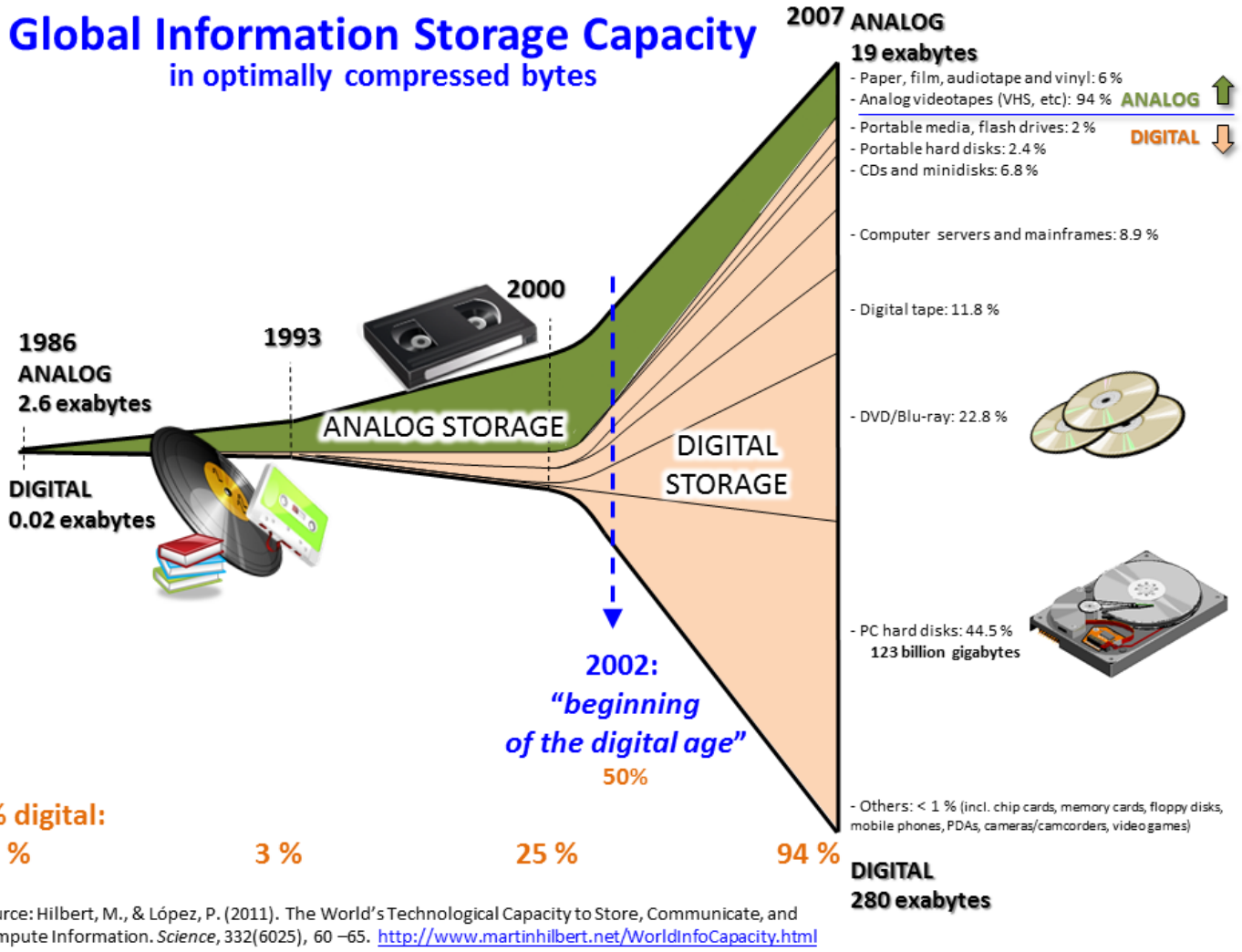
How big is big?

- If Excel couldn't open it, then it is big data (joking)
- But seriously, if your computer couldn't handle the data size, you need to consider cloud computing.



By Ender005 - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=49888192

Data sets grow very rapidly



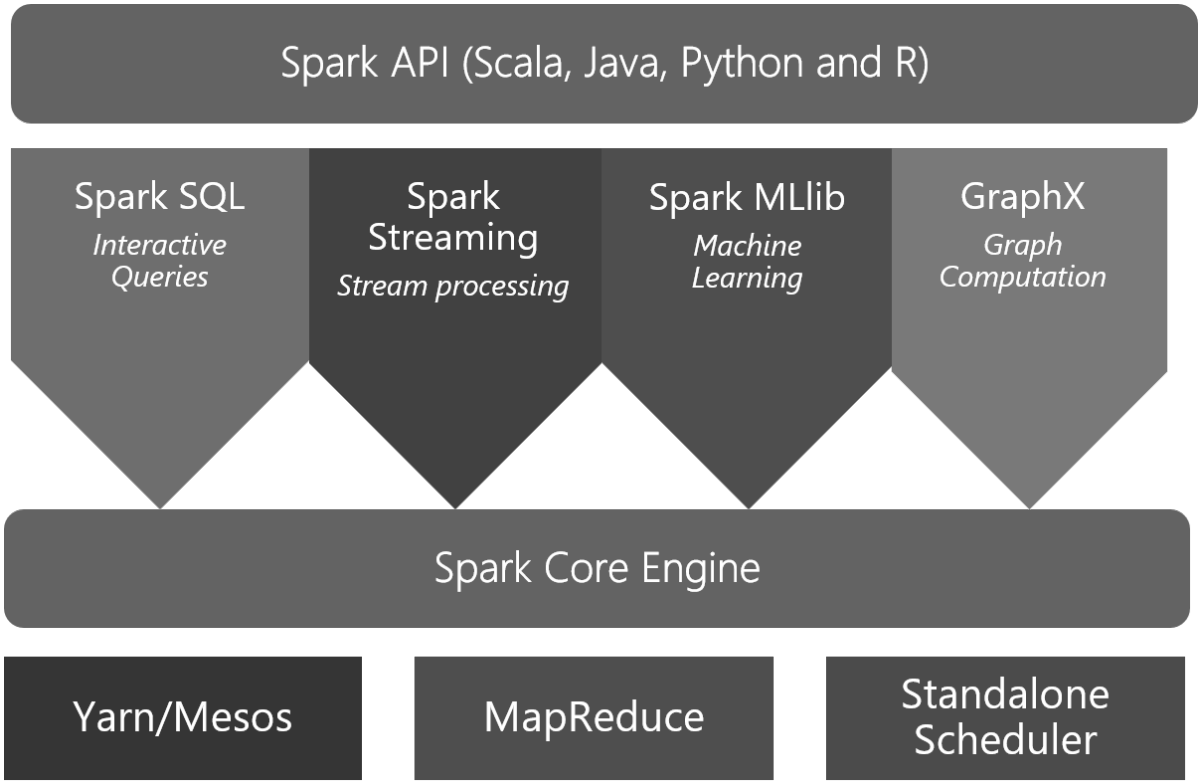
Source: Hilbert, M., & López, P. (2011). The World's Technological Capacity to Store, Communicate, and Compute Information. *Science*, 332(6025), 60 –65. http://www.martinhilbert.net/WorldInfoCapacity.html

Why cloud computing for R

- Benefits of Cloud computing
  - •Administration and operation of cloud resources.
  - •Computational efficiency of on-demand high-performance and distributed data analytics, “big data” in any size.
  - •Interactively prototype and develop data science or AI solutions.
  - •Remote interaction
  - •Deploy applications or services.
  - •Cost effectiveness.
  - •…

Cloud computing on Spark cluster

- Fast and general engine for large scale data processing.
- Java, Scala, Python, and R.
- Combine SQL, streaming, and complex analytics.
- Standalone, Mesos, and YARN as cluster manager.



R interfaces

- RevoScaleR
- Sparklyr
- SparkR

More about Spark

- Fast, expressive cluster computing system compatible with Apache Hadoop
  - Works with any Hadoop-supported storage system (HDFS, S3, Avro, …)
- Improves **efficiency** through: → Up to 100× faster
  - In-memory computing primitives
  - General computation graphs
- Improves **usability** through: → Often 2-10× less code
  - Rich APIs in Java, Scala, Python and R
  - Interactive shell

Key Ideas behind Spark

- **Functional programming.** Ship the data subset and function objects to each node and execute from there
- Hide the complexity of distributed computing. Expressive computing system, not limited to map-reduce model
- Work with distributed collections as you would with local ones with minimum code impact

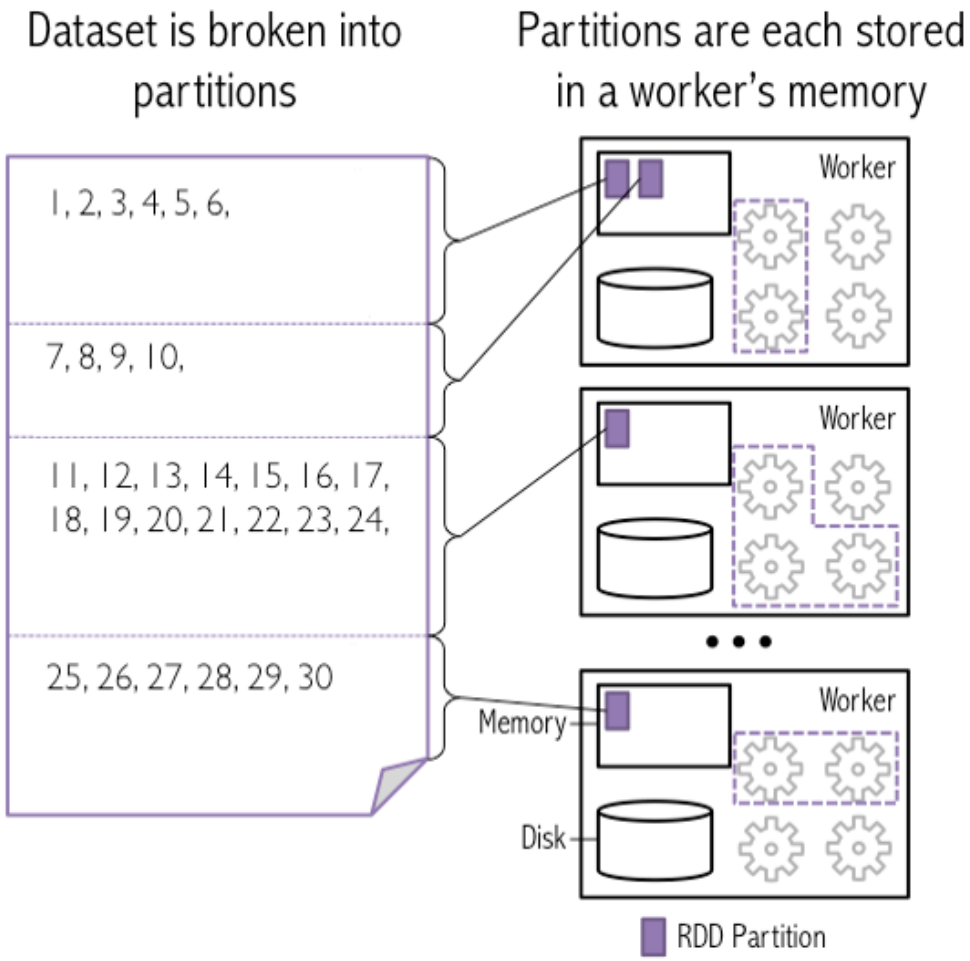
RDD abstraction

- Concept: resilient distributed datasets (RDDs)
  - Immutable, i.e., read-only
  - partitioned collections of objects
  - spread across a cluster
  - Built through parallel transformations (map, filter, etc)
  - Automatically rebuilt on failure
  - Controllable persistence (e.g. caching in RAM)
    - different storage levels available
    - fallback to disk possible

RDD operations

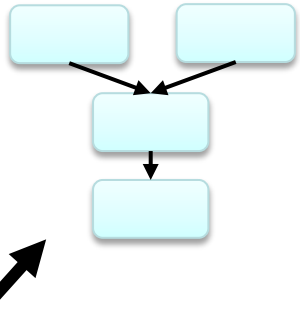
- *transformations* to build RDDs through deterministic operations on other RDDs
  - transformations include *map*, *filter*, *join*
  - lazy operation
- *actions* to return value or export data
  - actions include *count*, *collect*, *save*
  - triggers execution

RDD Partitions



From RDD to Worker

RDD Objects

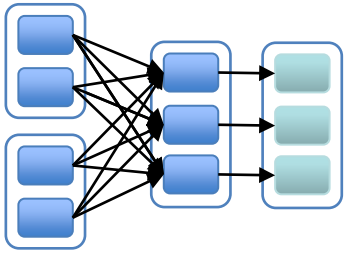


```
rdd1.join(rdd2)
    .groupBy(…)
    .filter(…)
```

build operator DAG

DAG

DAGScheduler

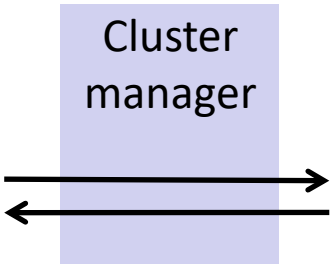


split graph into *stages* of tasks

submit each stage as ready

TaskSet

TaskScheduler

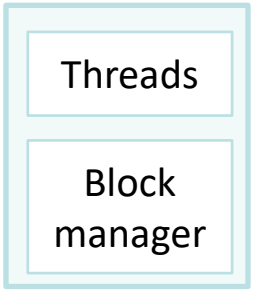


launch tasks via cluster manager

retry failed or straggling tasks

Task

Worker



execute tasks

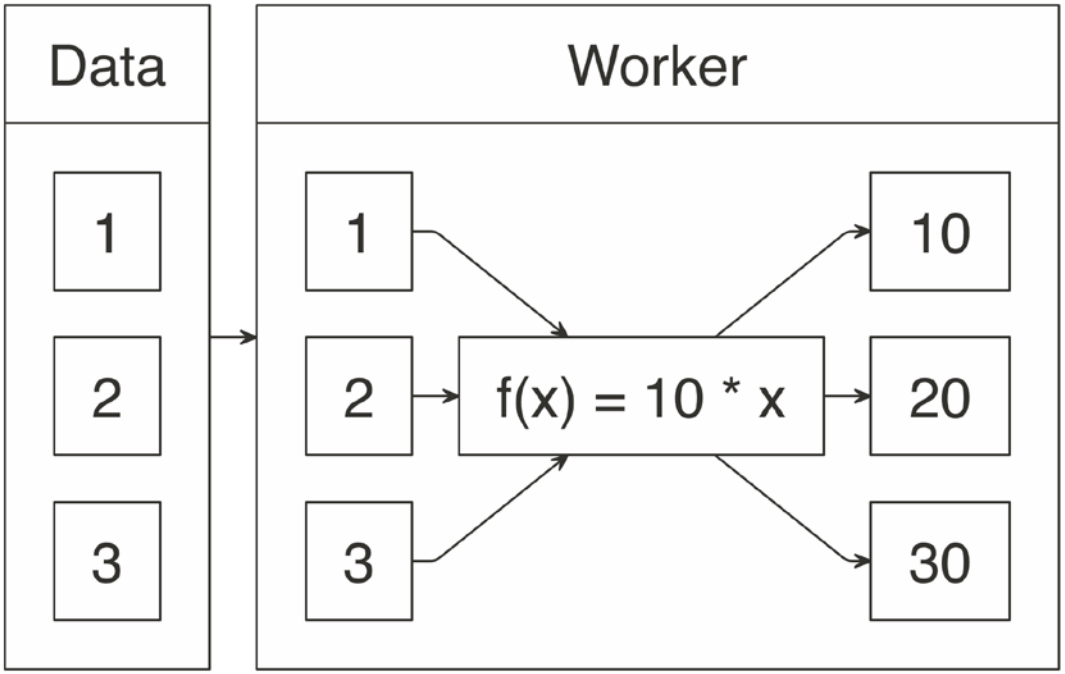
store and serve blocks

source: https://cwiki.apache.org/confluence/display/SPARK/Spark+Internals

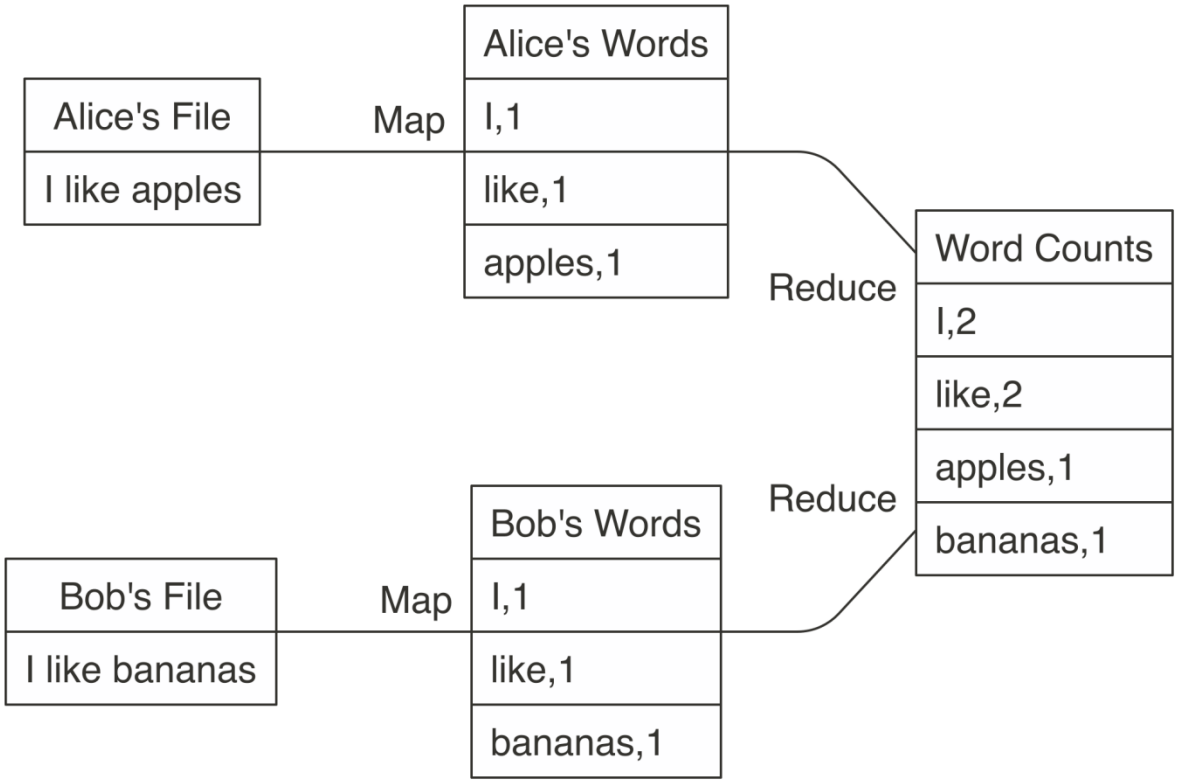
Perform operations across distributed datasets

- Two operations in MapReduce: map and reduce.
  - The *map operation* provides an arbitrary way to transform each dataset into a new dataset,
  - The *reduce operation* combines two dataset.
  - Both operations require custom computer code, but the MapReduce framework takes care of automatically executing them across many computers at once.

Map operation when multiplying by 10

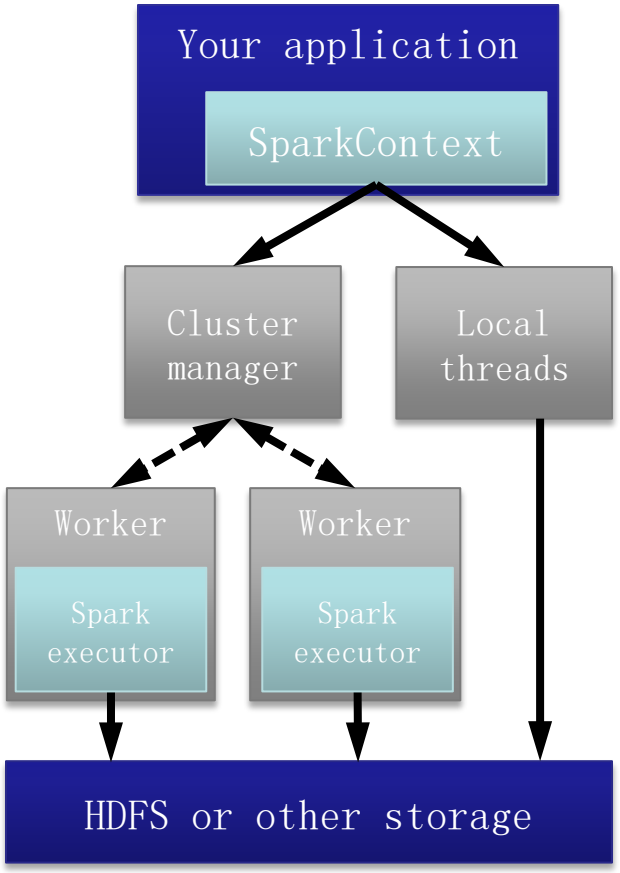


MapReduce example counting words across files



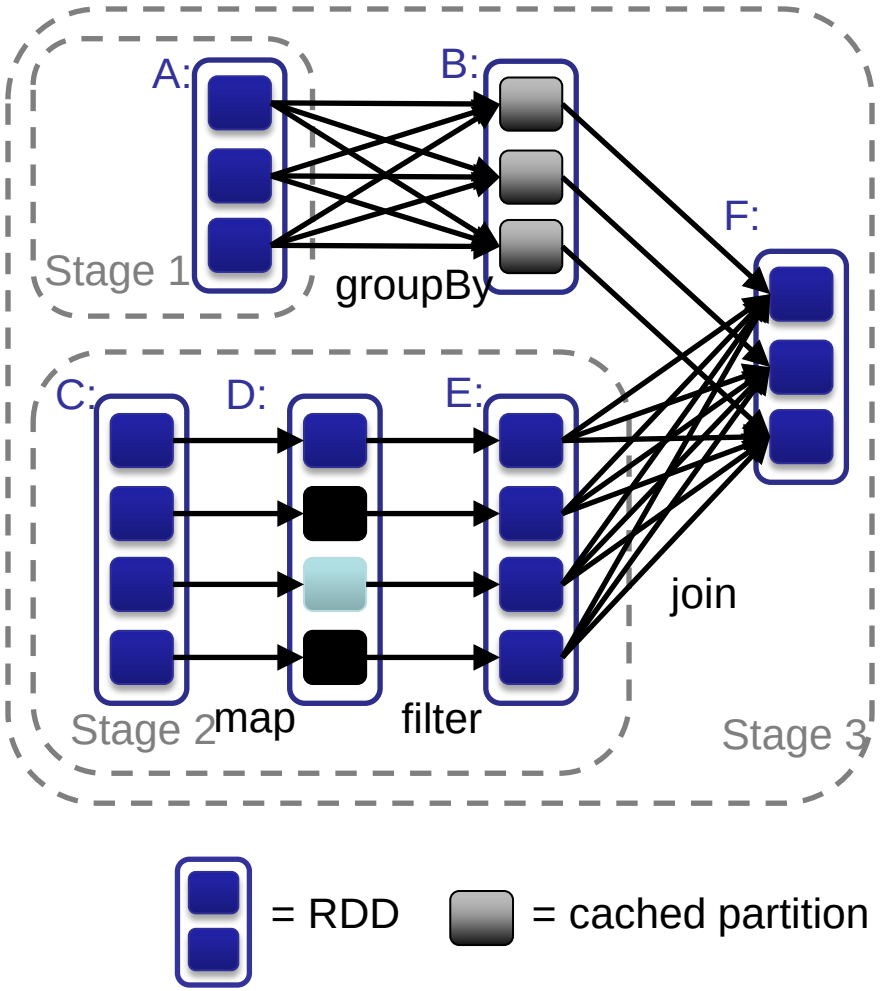
Software Cloud Architecture

- Spark runs as a library in your program (one instance per app)
- Runs tasks locally or on a cluster
  - Standalone deploy cluster, Mesos or YARN
- Accesses storage via Hadoop InputFormat API
  - Can use HBase, HDFS, S3, …



Task DAG and Scheduler

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



Hadoop Compatibility

- Spark can read/write to any storage system / format that has a plugin for Hadoop!
  - Examples: HDFS, S3, HBase, Cassandra, Avro, SequenceFile
  - Reuses Hadoop's InputFormat and OutputFormat APIs

R: Distributed Data Frame on Spark cluster

- sparklyr – R interface to Apache Spark with a complete backend of dplyr
  - Filter and aggregate Spark datasets and then bring them into R for analysis and visualization.
  - Use Spark's distributed machine learning library from R.
  - Create extensions to call Spark API and provide interfaces to Spark packages.
- SparkR – light-weight frontend to use Apache Spark in R
  - Operations like filtering, grouping, selecting, etc., and MLib model training.

```
# sparklyr for data manipulation.
sc <- spark_connect(master="local")

iris_tbl <- copy_to(sc, iris)
flights_tbl <- copy_to(sc, flights, "flights")

iris_preview <-
   dbGetQuery(sc, "SELECT * FROM iris LIMIT 10")
flights_tbl %>% filter(dep_delay == 2)
```

```
# SparkR for data manipulation
sparkR.session(master="local[*]",

sparkConfig=list(spark.driver.memory="2g")

df <- as.DataFrame(faithful)

head(select(df, df$eruptions))
head(select(df, "eruptions"))
```

First Step: SparkContext

- Main entry point to Spark functionality
- Created for you in Spark shells as variable **sc**
- In standalone programs, you'd make your own (see sample code in Jupyter notebook)

```
library(sparklyr)
sc <- spark_connect(master = "local")
```

Creating RDDs

- An RDD distributes copies of the same data across many machines, such that if one machine fails, others can complete the task—hence, the term “resilient.”
- Most sparklyr operations that retrieve a Spark DataFrame cache the results in memory
- copy_to() will provide a Spark DataFrame that is already cached in memory.
- As a Spark DataFrame, this object can be used in most sparklyr functions, including data analysis with dplyr or machine learning

```
cars<- copy_to(sc, mtcars)
spark_web(sc) # Open the spark web UI
```

Remote "dataframe" on Spark

- The dataframe "mtcars" was copied into Spark
- Returns a reference "cars" to the dataset in Spark
- The class of "cars" is not "data.frame"
- To print the reference "cars", Spark *collects* some of the records and displays.

```
[44]: cars
```

```
# Source: spark<mtcars> [?? x 11]
     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1  21        6  160    110  3.9   2.62  16.5     0     1     4     4
 2  21        6  160    110  3.9   2.88  17.0     0     1     4     4
 3  22.8      4  108     93  3.85  2.32  18.6     1     1     4     1
 4  21.4      6  258    110  3.08  3.22  19.4     1     0     3     1
 5  18.7      8  360    175  3.15  3.44  17.0     0     0     3     2
 6  18.1      6  225    105  2.76  3.46  20.2     1     0     3     1
 7  14.3      8  360    245  3.21  3.57  15.8     0     0     3     4
 8  24.4      4  147.    62  3.69  3.19  20       1     0     4     2
 9  22.8      4  141.    95  3.92  3.15  22.9     1     0     4     2
10  19.2      6  168.   123  3.92  3.44  18.3     1     0     4     4
# ... with more rows
```

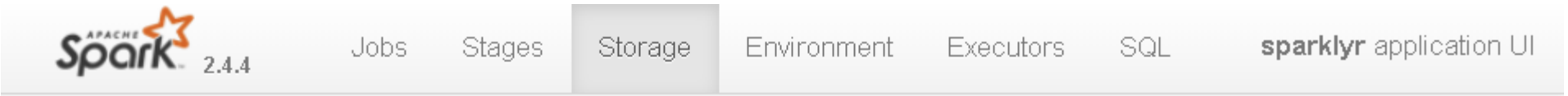
```
[45]: class(cars)
```

'tbl_spark' · 'tbl_sql' · 'tbl_lazy' · 'tbl'

```
[46]: typeof(cars)
```

'list'

Cached RDD in the Spark web interface



RDD Storage Info for In-memory table `mtcars`

Storage Level: Memory Deserialized 1x Replicated
Cached Partitions: 1
Total Partitions: 1
Memory Size: 3.9 KB
Disk Size: 0.0 B

Data Distribution on 1 Executors

Host	On Heap Memory Usage	Off Heap Memory Usage	Disk Usage
3dns.adobe.com:63188	3.9 KB (391.0 MB Remaining)	0.0 B (0.0 B Remaining)	0.0 B

1 Partitions

Block Name ▲	Storage Level	Size in Memory	Size on Disk	Executors
rdd_9_0	Memory Deserialized 1x Replicated	3.9 KB	0.0 B	3dns.adobe.com:63188

Use dplyr functions on remote data

- General practice
  - -Analyze remote data in Spark with dplyr as if the data is local
  - -Select a subset of columns to limit the return size
  - -Sample rows because “head” doesn’t make sense in distributed data
  - -collect() to retrieve all data from this data reference and return a local dataframe, which hands over to local R
  - DANGER: Know your data size before run collect()

```
select(cars, hp, mpg) %>%
sample_n(100) %>%
collect() %>%
plot()
```

Data I/O options on Spark

- Read dataset from a distributed storage system like HDFS, or AWS S3 bucket, etc

- Or read from local file system without creating local dataframe

```
cars <- spark_read_csv(sc, "cars.csv")
```

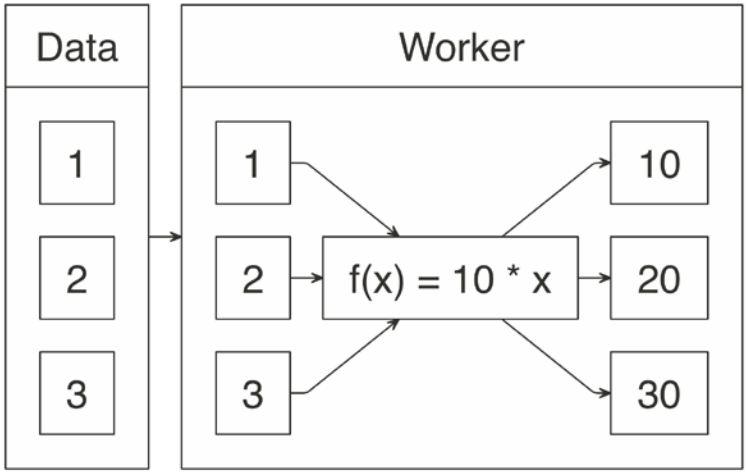
- Export remote dataframe as a local csv file

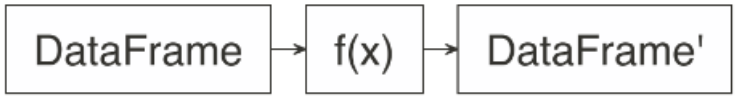
```
spark_write_csv(cars, "cars.csv")
```

DANGER: Know your data size before run it

Distributed R on Spark cluster

- Distributed dplyr on Spark is great
- How to distribute user-defined R functions?






```
copy_to(sc, data.frame(a=1:3)) %>%
    spark_apply(function(x)(x*10)) %>%
                collect()
```

A tibble: 3 × 1

a
<dbl>
10
20
30

```
f=function(x)(x*10)
f(data.frame(a=1:3))
```

A data.frame: 3 × 1

a
<dbl>
10
20
30

Compact definition of function

- The ~ operator is defined in the rlang package and provides a compact definition of an *anonymous* function

```
fx=rlang::as_function(~ 10 * .x)
```

```
fx(10)
```

100

```
fx(data.frame(x=1:3, y=4:6))
```

A data.frame: 3 × 2

x <dbl>	y <dbl>
10	40
20	50
30	60

```
f=function(x)(x*10)
```

```
f(10)
```

100

```
f(data.frame(x=1:3, y=4:6))
```

A data.frame: 3 × 2

x <dbl>	y <dbl>
10	40
20	50
30	60

Use compact function in Spark

- Rewrite the code

Data Partitioning on Spark

- The remote dataframe is not one piece. It consists of multiple pieces on multiple machines for parallel computing.
- Most Spark operations, including dplyr, simply work automatically without worrying about partitioning, as if the dataframe is local
- For distributed user-defined R function, it can be very tricky

```
copy_to(sc, data.frame(1:100), repartition =2) %>%
  spark_apply(~nrow(.x))
```

```
# Source: spark<?> [?? x 1]
  result
   <int>
1     50
2     50
```

- spark_apply() receives each partition and processes them separately, so the results are still in separate partitions
- Always keep in mind, it silently calls collect() to print result so that the result looks like one piece.

Aggregate partitions

- Repartition the results into one partition so that you can run another distributed R function on this single partition
- i.e., spark_apply does the “Map”, you manually do the “Reduce”

```
copy_to(sc, data.frame(1:100), repartition =2) %>%
  spark_apply(~nrow(.x)) %>%
  sdf_repartition(1) %>%
  spark_apply(~sum(.x))
```

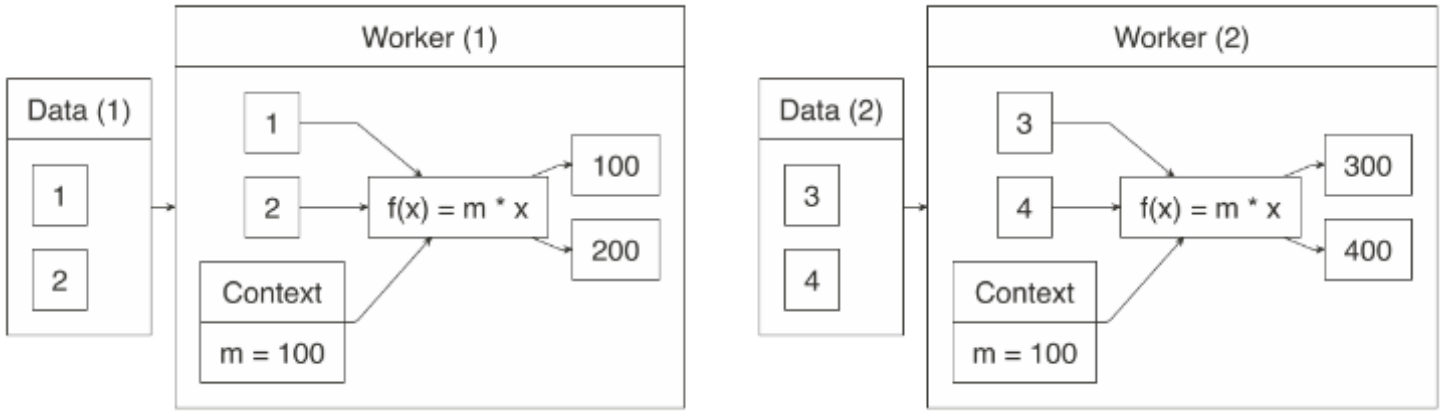
```
# Source: spark<?> [?? x 1]
  result
   <int>
1    100
```

Carry-on bag of your R function

- You may want to include auxiliary data together with your function to Spark
- Specify it as the ”context” parameter

```
copy_to(sc, data.frame(a=1:4), repartition=2) %>%
    spark_apply(function(x, context) x*context, context=100)
```

```
# Source: spark<?> [?? x 1]
      a
  <dbl>
1   100
2   200
3   300
4   400
```



Carry-on multiple items

- Since the *context* parameter is serialized as an R object, it can contain anything
- We know that list is a container, so..

```
copy_to(sc, data.frame(a=1:4), repartition=2) %>%
    spark_apply(function(x, context) x*context$a+context$b,
                context=list(a=100, b=5))
```

```
# Source: spark<?> [?? x 1]
      a
  <dbl>
1   105
2   205
3   305
4   405
```

- The values in context can be dynamic. May contain function objects as well. (again, functional programming)
- Functions with a context are also referred to as a *closure*.
    - In contrast, pure functions don't have context, i.e., don't depend on outside information

Release RDDs

- Data loaded in memory will be released when the R session terminates, either explicitly or implicitly, with a restart or disconnection
- To manually free up resources, you can use tbl_uncache():

```
tbl_uncache(sc, “cars")
```

Disconnect from Spark cluster

- After you are done processing data, you should disconnect by running the following

```
spark_disconnect(sc)
```

- Disconnect all active Spark connections

```
spark_disconnect_all()
```

How to Run It

- Local multicore: just a library in your program
- Virtual machines on cloud/AWS/Azure: scripts for launching a Spark cluster
- Private cluster: Mesos, YARN, Standalone Mode

Summary

- Spark offers a rich API to make data analytics *fast*: both fast to write and fast to run
- Achieves 100x speedups in real applications
- Details, tutorials, videos:
  - http://spark.apache.org/
  - https://therinspark.com/
  - https://spark.rstudio.com/