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.

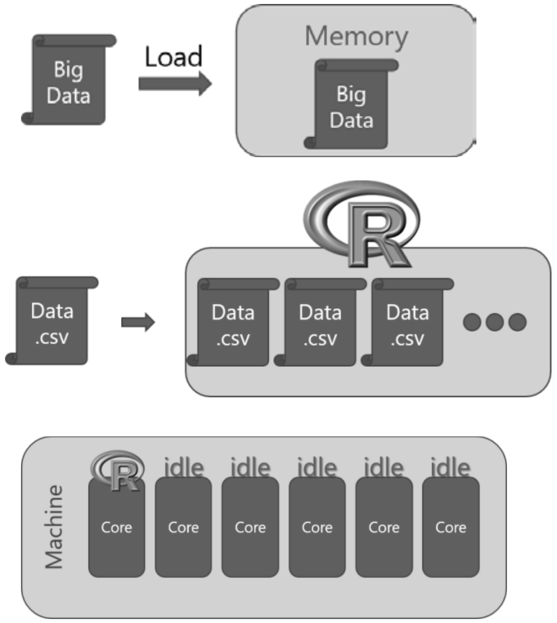
Is R so far so good?

- 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

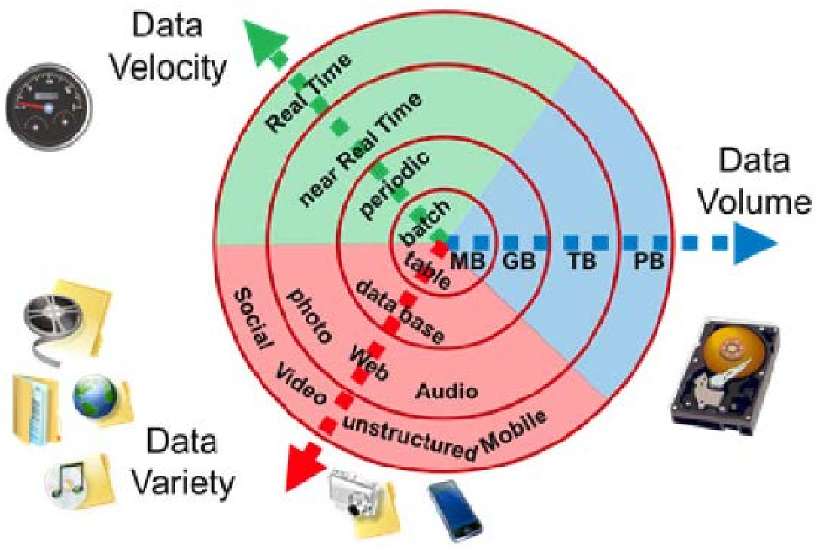
Lack of parallelism



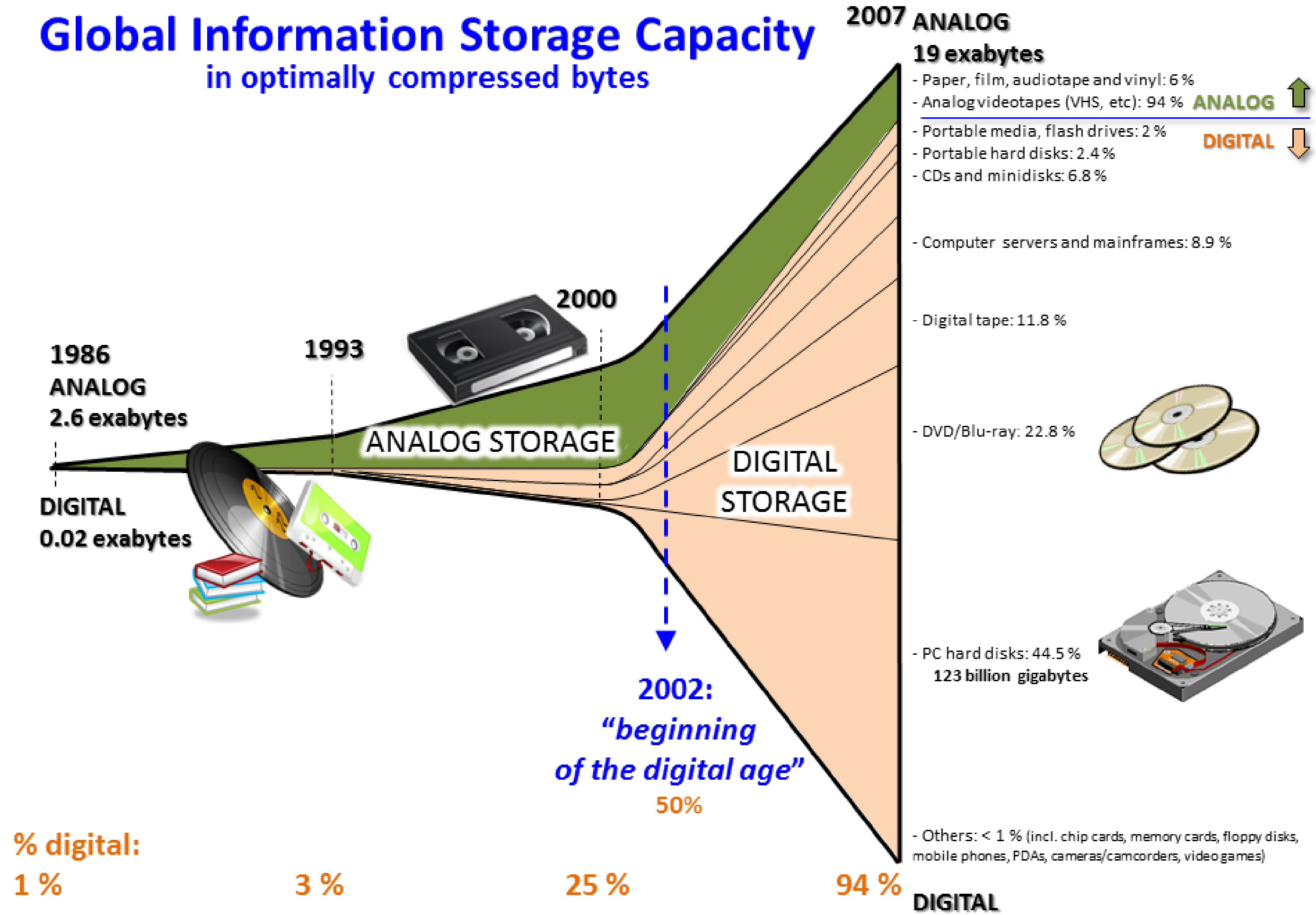
How big is big?

-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.



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

By Myworkforwiki - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=29452425

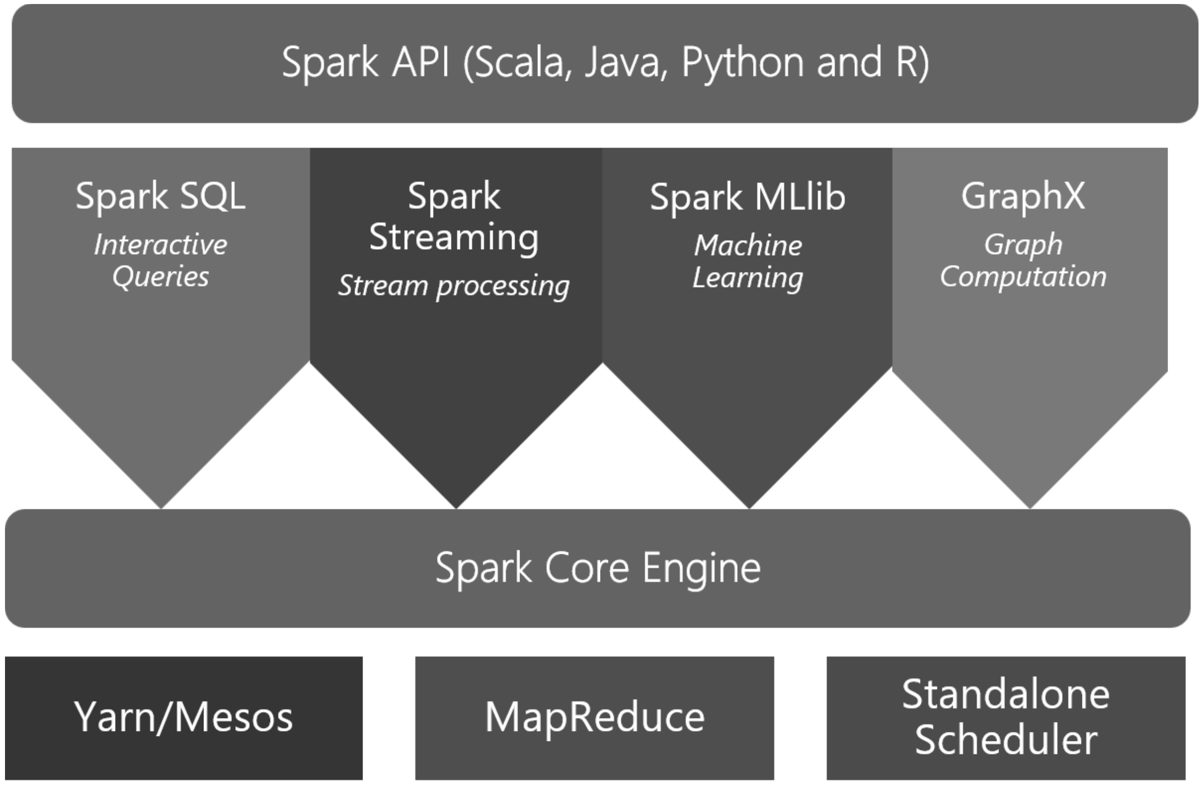
Why cloud computing for R

- 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.
  - •…

code R
THANKS TO CLOUD TECHNOLOGY, NIGEL CAN NOT ONLY ~~WORK~~ FROM HOME, BUT HE CAN ~~WORK~~ FROM OTHER PEOPLE'S HOMES AS WELL.

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: $\longrightarrow$ Up to 100 × faster
  - In-memory computing primitives
  - General computation graphs
- Improves **usability** through: $\longrightarrow$ 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

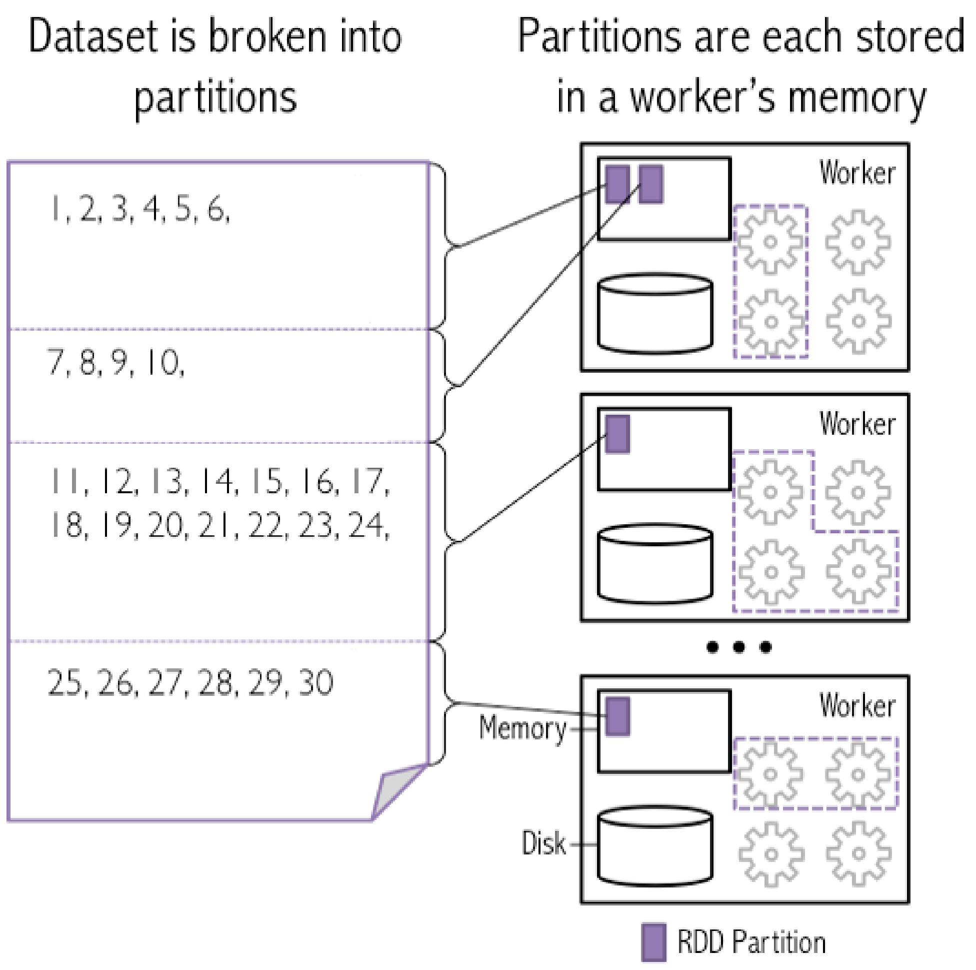
RDD abstraction

- 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

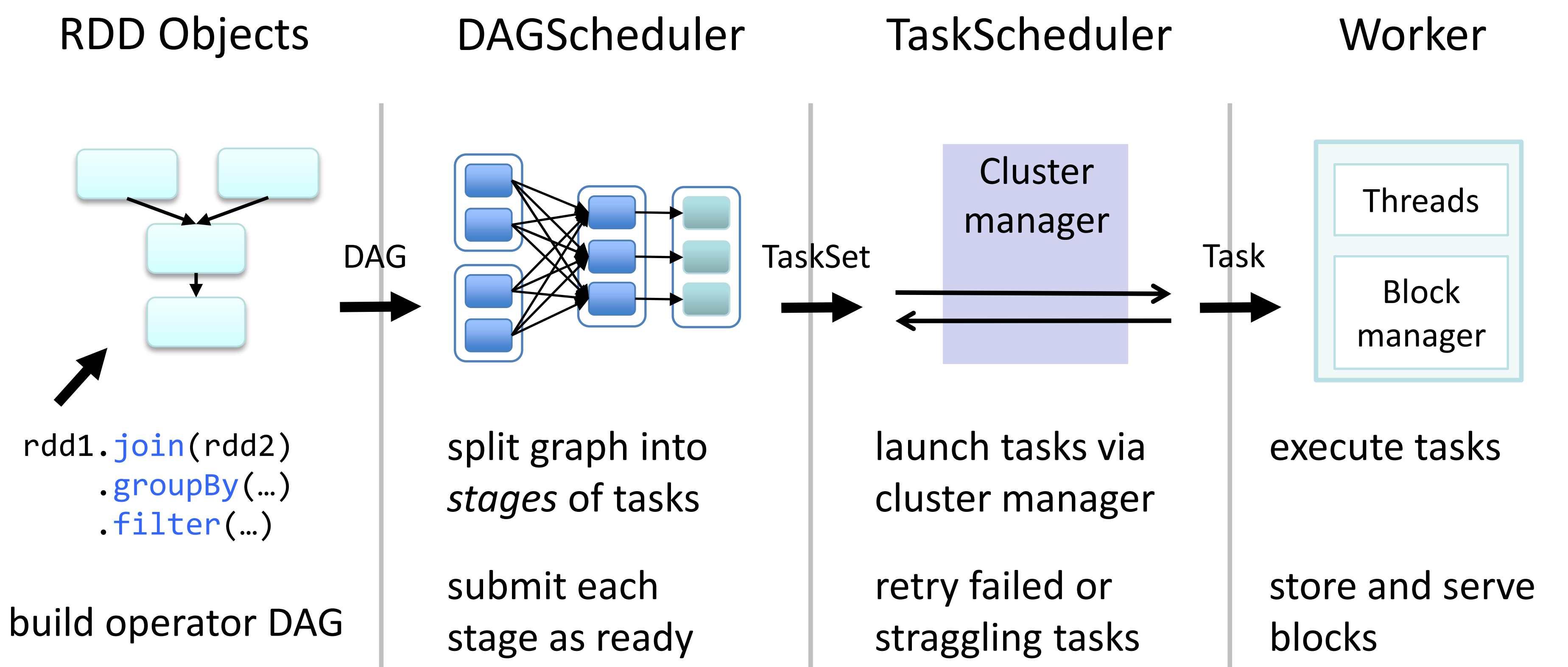
RDD operations

- *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



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

build operator DAG

split graph into *stages* of tasks

submit each stage as ready

launch tasks via cluster manager

retry failed or straggling tasks

execute tasks

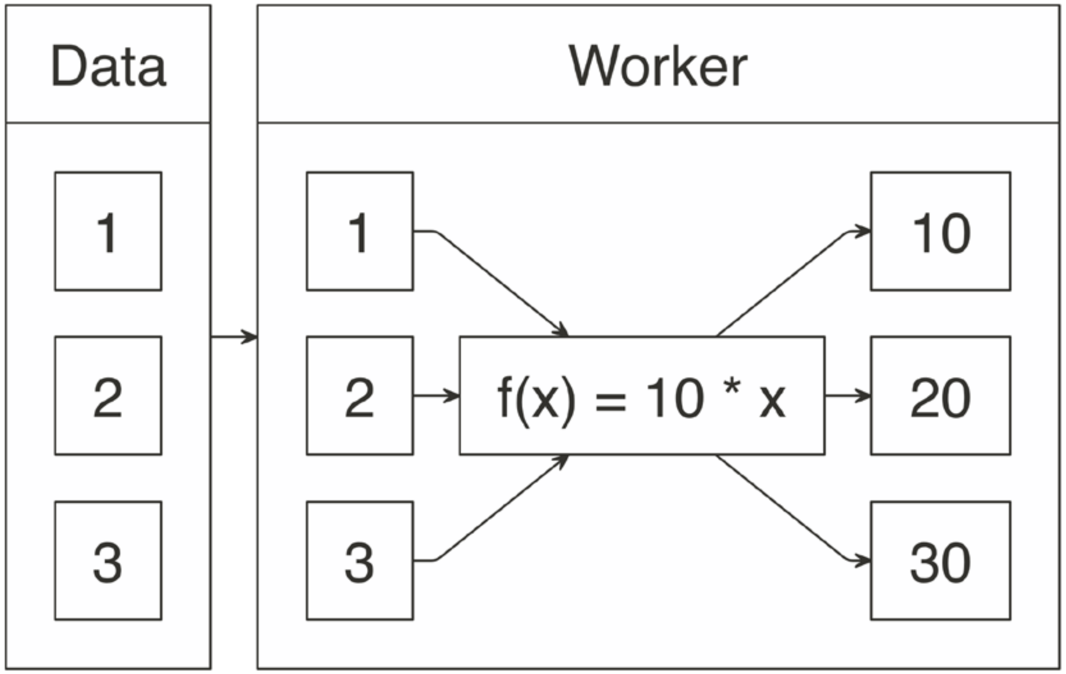
store and serve blocks

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

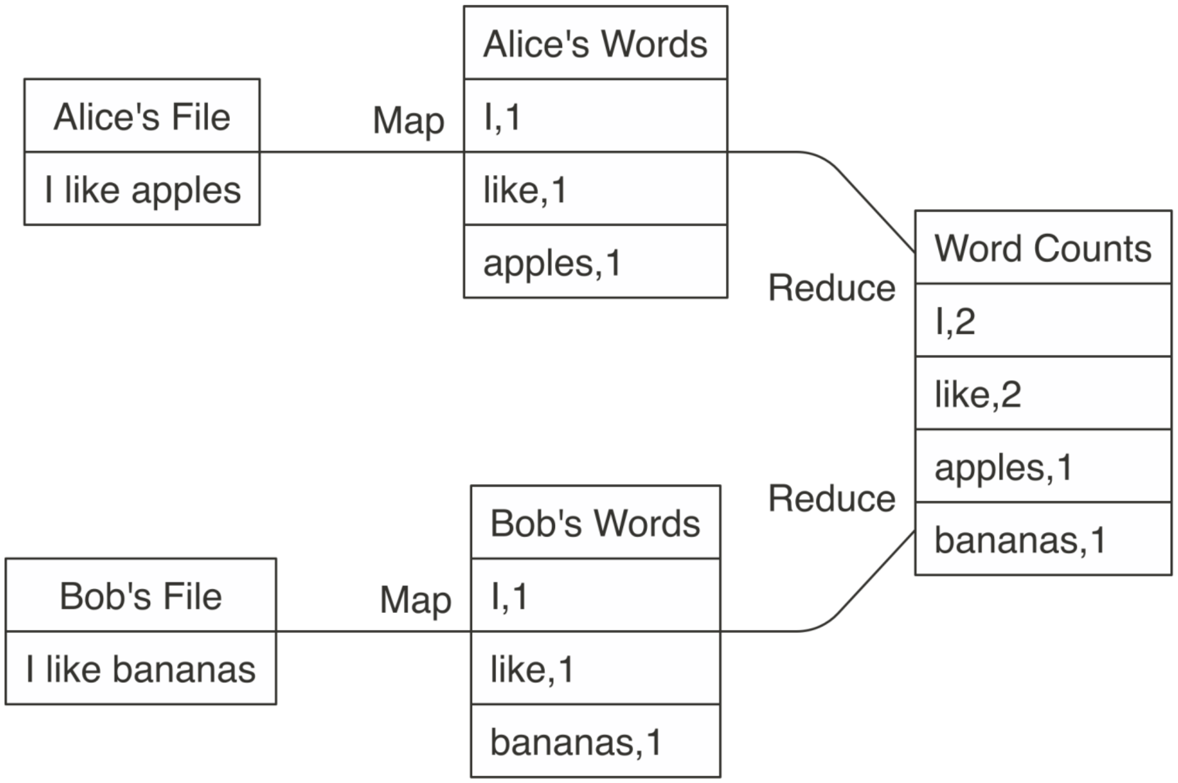
Perform operations across distributed datasets

- 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

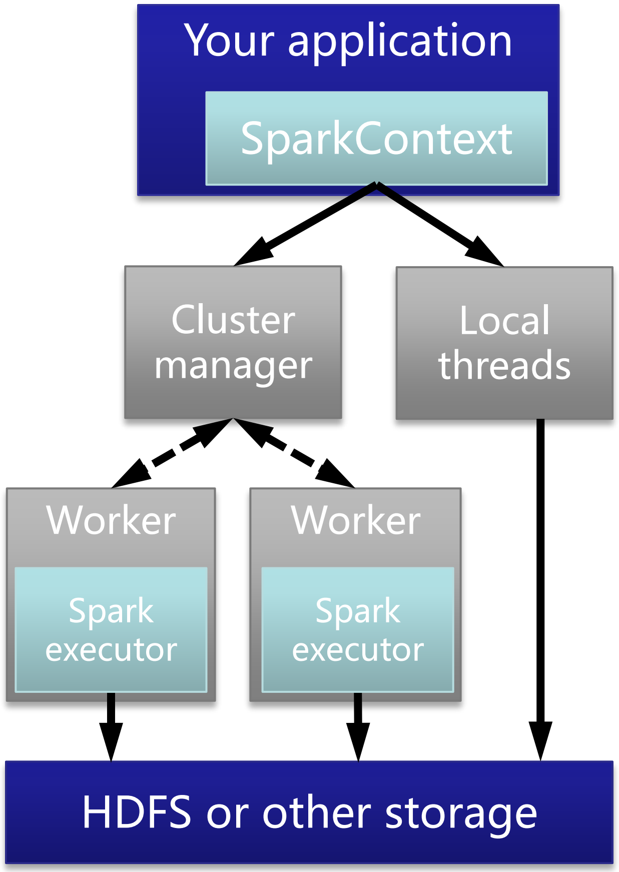


MapReduce example counting words across files



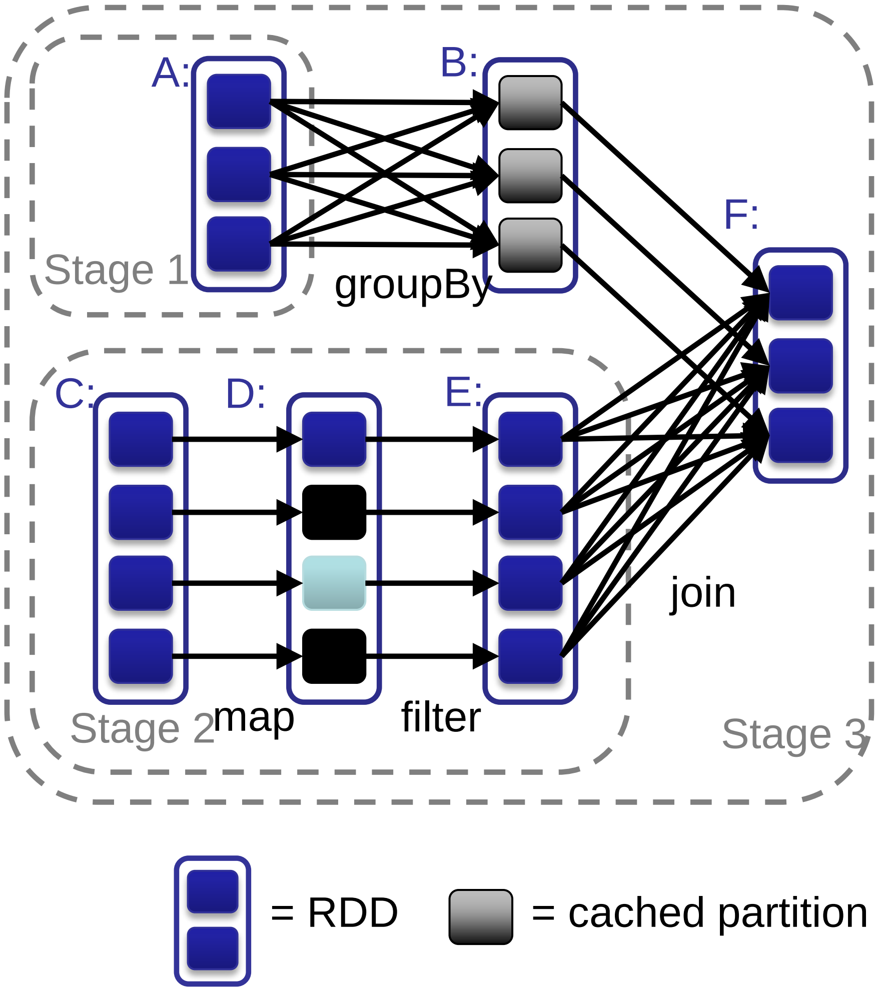
Software Cloud Architecture

- 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")
```

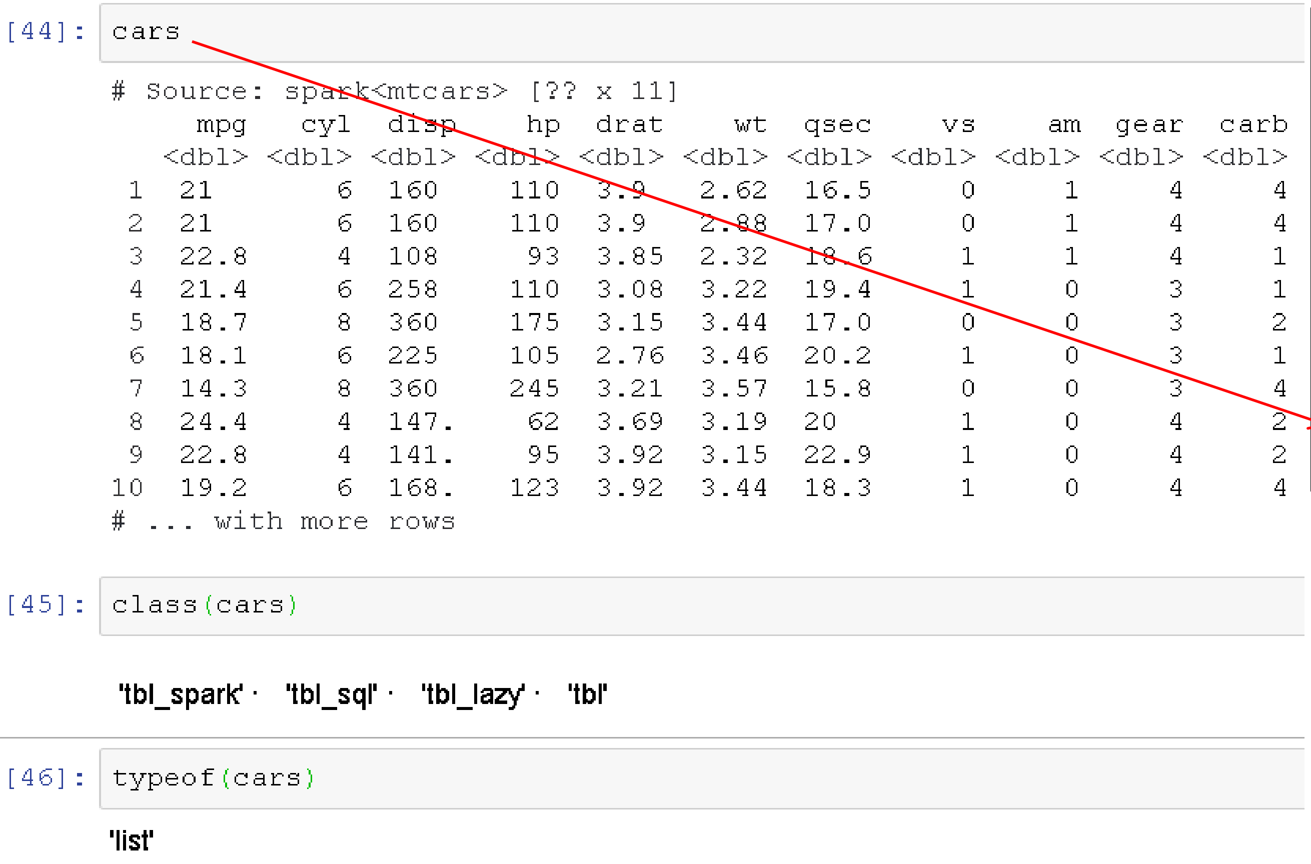
Creating RDDs

- 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

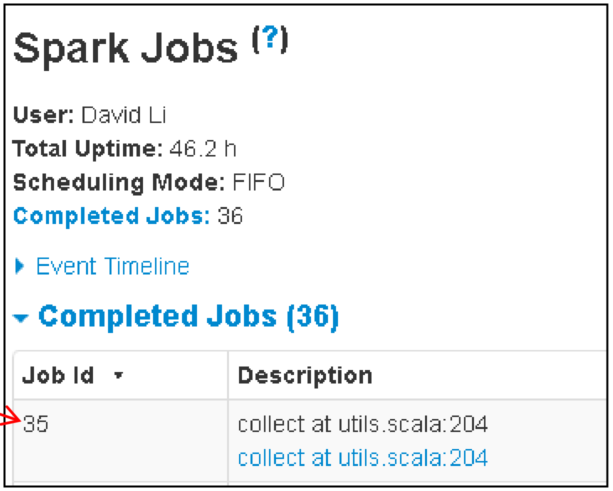
```
cars<- copy_to(sc, mtcars)
spark_web(sc) # Open the spark web UI
```

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'
```



Spark Jobs (?)

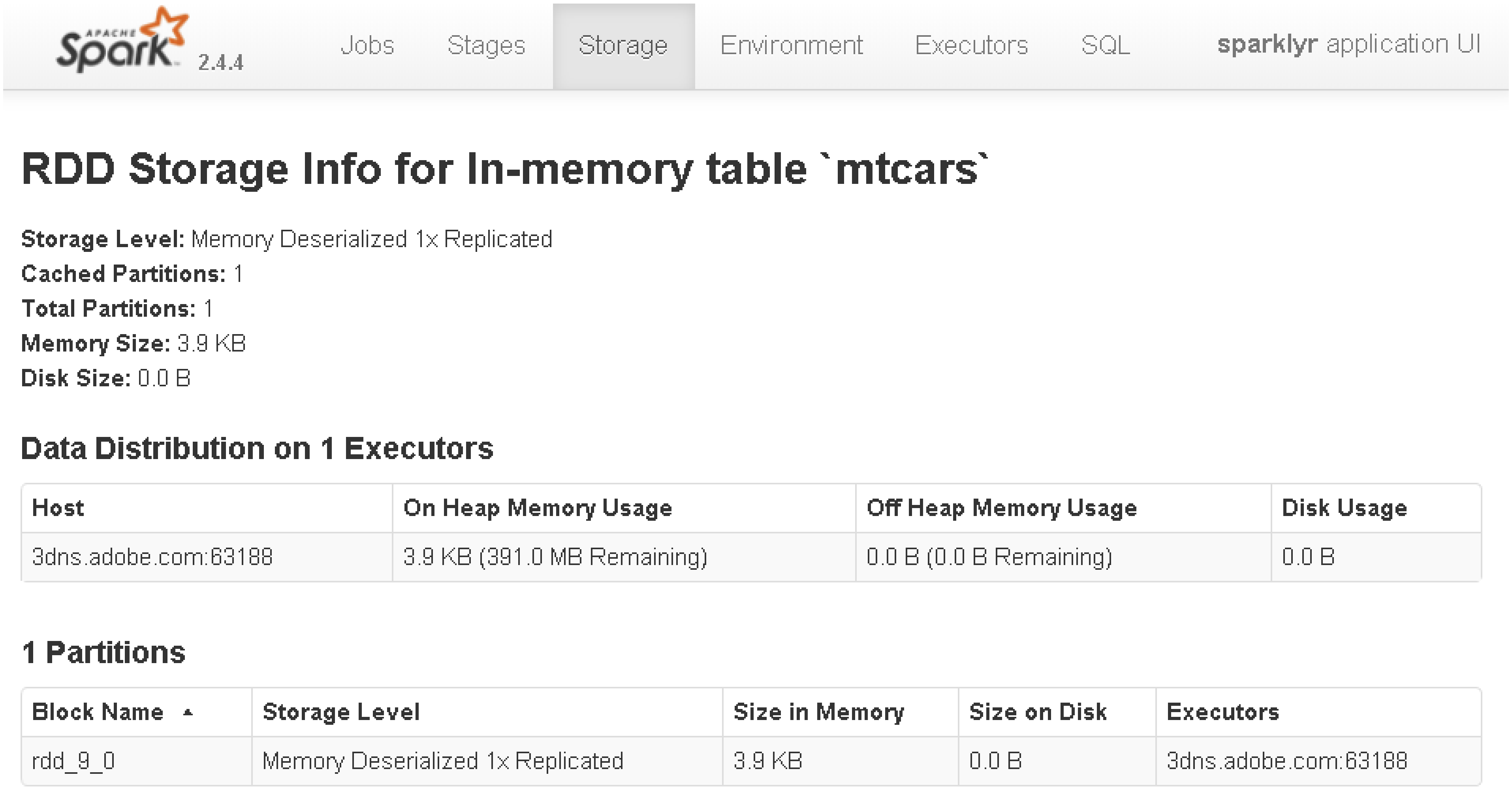
User: David Li
Total Uptime: 46.2 h
Scheduling Mode: FIFO
Completed Jobs: 36

▸ Event Timeline

▾ **Completed Jobs (36)**

Job Id ▾	Description
35	collect at utils.scala:204 collect at utils.scala:204

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")
```

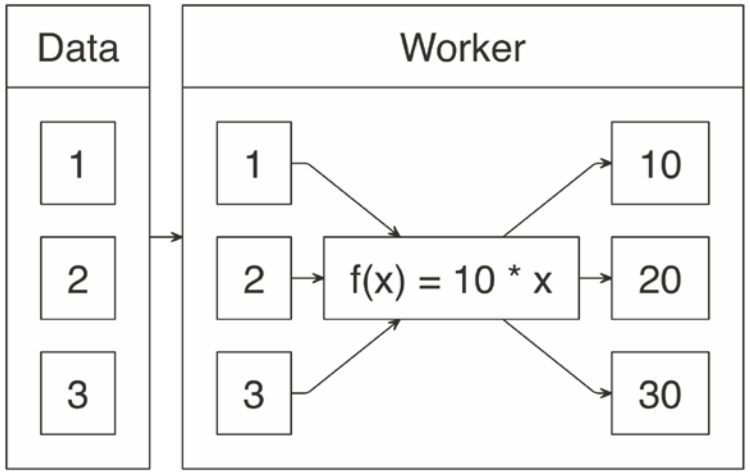
•Export remote dataframe as a local csv file

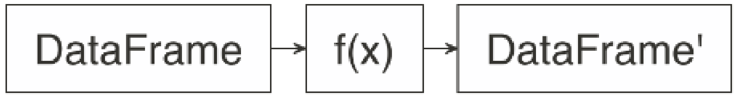
```
spark_write_csv(cars, "cars.csv")
```

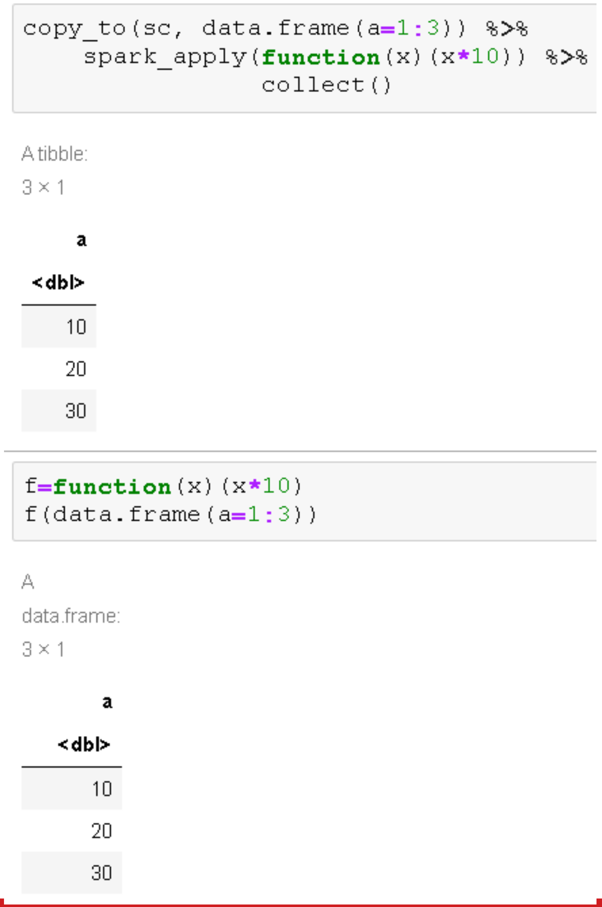
DANGER: Know your data size before run it

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

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

```
f(10)
```

```
100
```

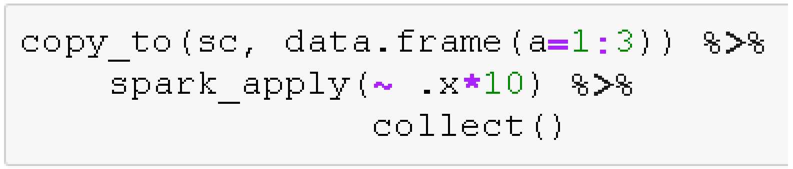
```
f(data.frame(x=1:3, y=4:6))
```

A data.frame: 3 × 2

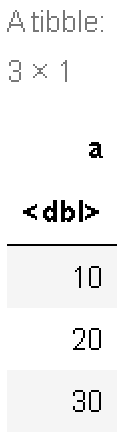
x	y
<dbl>	<dbl>
10	40
20	50
30	60

Use compact function in Spark

- Rewrite the code



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



```
A tibble:
3 × 1
    a
<dbl>
   10
   20
   30
```

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”

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

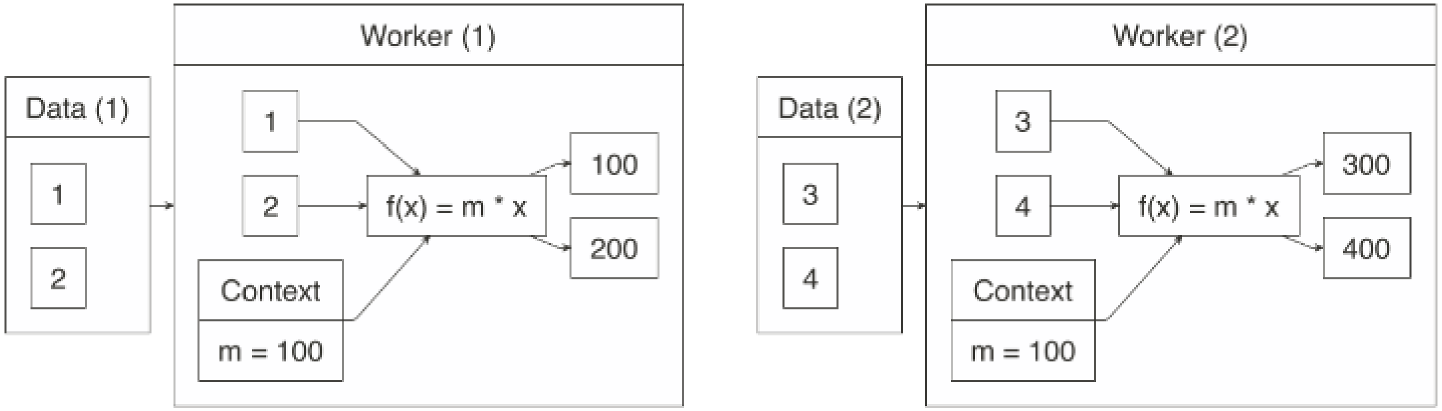
```
# Source: spark<?> [?? x 1]
  result
   <int>
1    100
```

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/

Graph Theory and Analytics in R

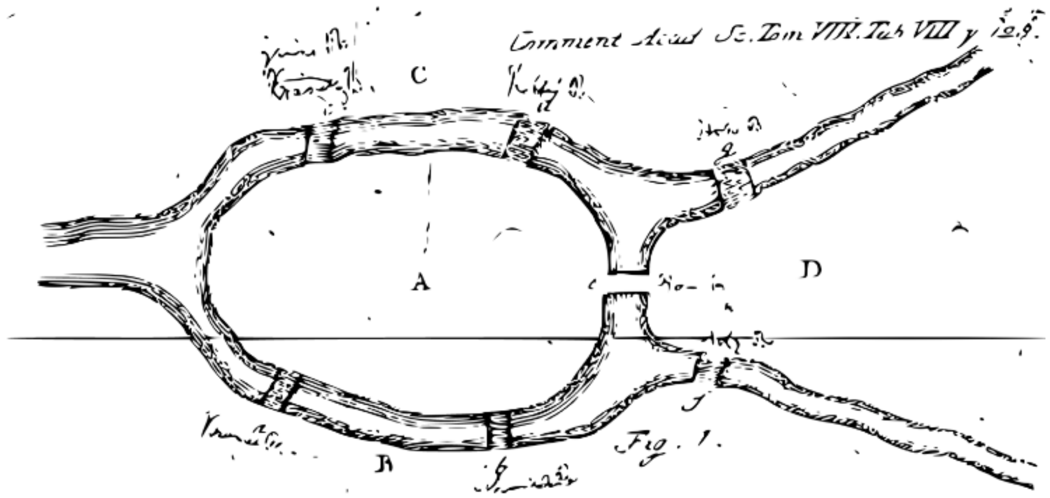
David Li

The goals of today

- Introduction to Graph Theory
    - Vertex
    - Edge
    - Degree
    - Path
- Graph Analytics in R with Spark
    - Create graphframes
    - Run algorithms to get answers
- Apply the graph analytics in real word problems
    - Social media network analysis
        - Facebook/WeChat/Line/Linkedin/YouTube
    - Investment banking analysis
        - Peer selection/Pitching/M&A

Introduction to Graph Theory

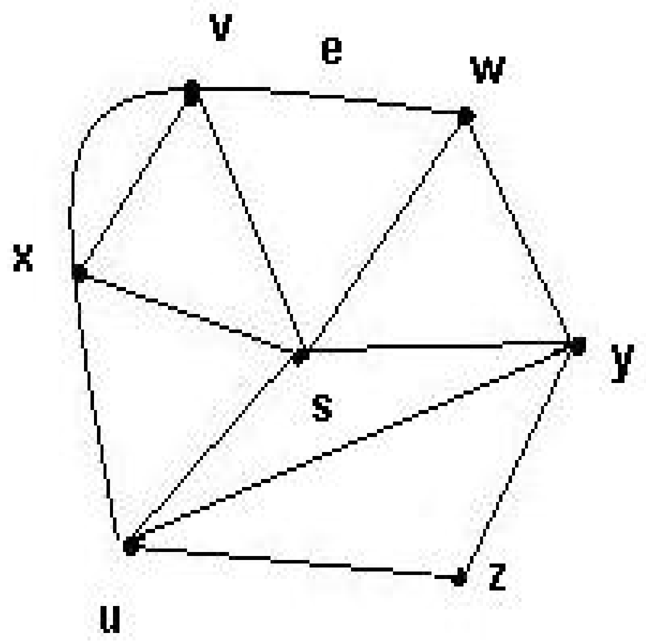
The Seven Bridges of Königsberg



- 1736: **Leonhard Euler**
  - Basel, 1707-St. Petersburg, 1786
  - He wrote *A solution to a problem concerning the geometry of a place.* First paper in graph theory.
- Problem of the Königsberg bridges:
  - Starting and ending at the same point, is it possible to cross all seven bridges just once and return to the starting point?

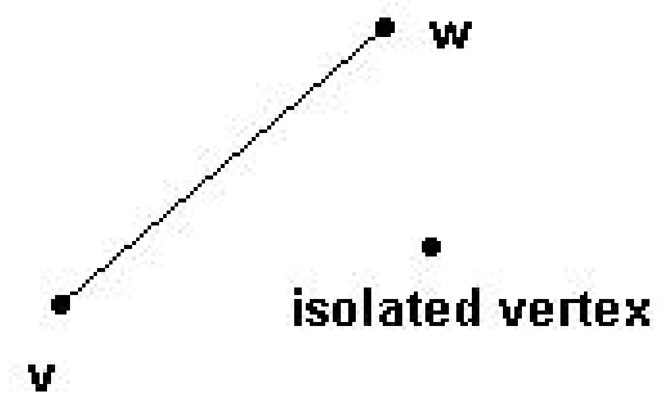
Basic Concepts of Graph

- What is a graph G?
- It is a pair G = (V, E), where
  - V = V(G) = set of vertices
  - E = E(G) = set of edges
- **Example:**
  - V = {s, u, v, w, x, y, z}
  - E = {(x,s), $(x,v)_1$, $(x,v)_2$, (x,u), (v,w), (s,v), (s,u), (s,w), (s,y), (w,y), (u,y), (u,z),(y,z)}



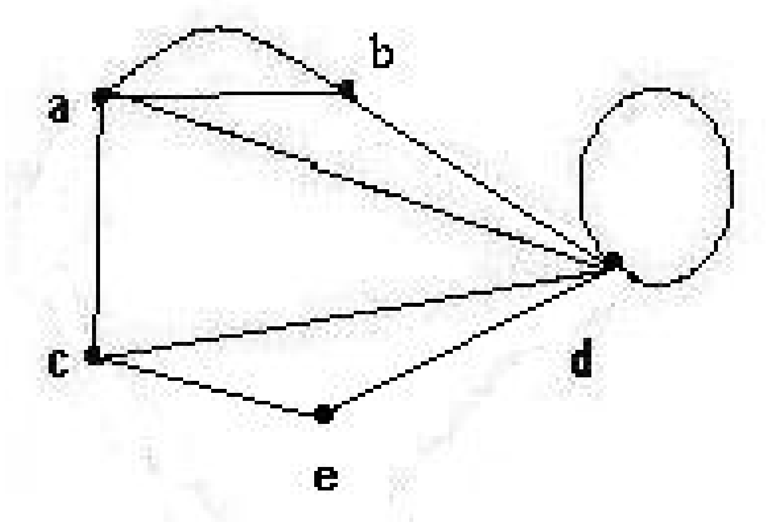
Edges

- An edge may be labeled by a pair of vertices, for instance e = (v,w).
- e is said to be *incident* on v and w.
- Isolated vertex = a vertex without incident edges.



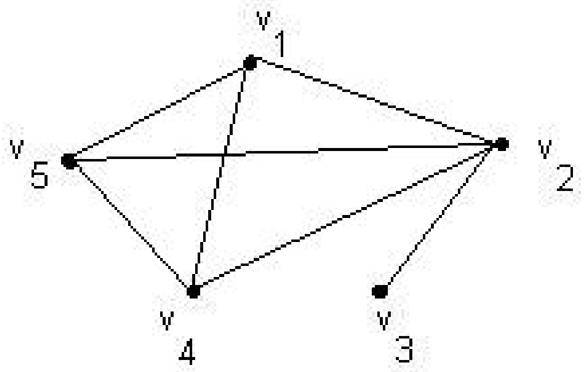
Special edges

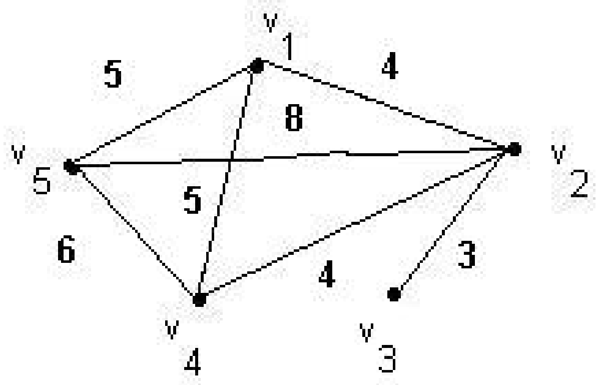
- Parallel edges
  - Two or more edges joining a pair of vertices
    - in the example, **a** and **b** are joined by two parallel edges
- Loops
  - An edge that starts and ends at the same vertex
    - In the example, vertex **d** has a loop



Special graphs

- Simple graph
  - A graph without loops or parallel edges.
- Weighted graph
  - A graph where each edge is assigned a numerical label or “weight”.
  - Examples:
    - Shipping price between cities,
    - ping round trip time between IPs,
    - exchange rate between currencies

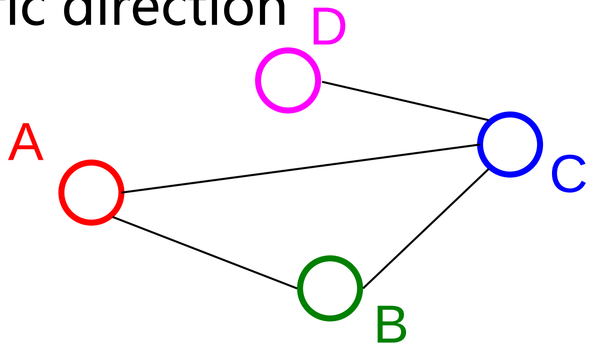




Undirected Graphs and Degree

In undirected graphs, edges have no specific direction

- Edges are always "two-way "
  - Such as "friends" relationship



Thus, $(u, v) \in E$ implies $(v, u) \in E$.

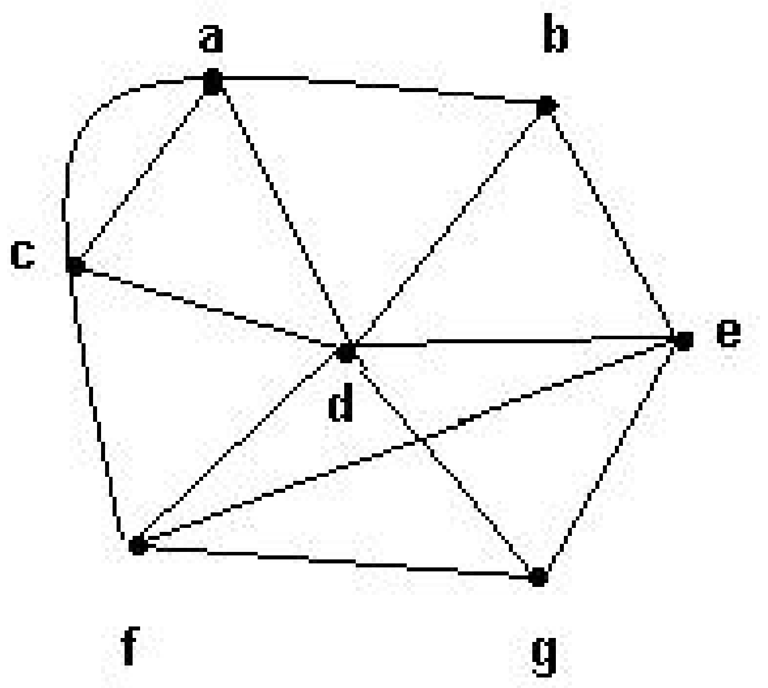
- Only one of these edges needs to be in the set
- The other is implicit, so normalize how you check for it

Degree of a vertex: number of edges containing that vertex

- Put another way: the number of adjacent vertices

Degree of a vertex in undirected graphs

- The *degree* of a vertex v, denoted by δ(v), is the number of edges incident on v
- Example:
  - δ(a) = 4, δ(b) = 3,
  - δ(c) = 4, δ(d) = 6,
  - δ(e) = 4, δ(f) = 4,
  - δ(g) = 3.



The Properties of Degree in undirected graphs

- The degree of a vertex deg(v) is a number of edges that have vertex v as an endpoint. Loop edge gives vertex a degree of 2
- In any graph the sum of degrees of all vertices equals twice the number of edges
- The total degree of a graph is even
- In any graph there are even number of vertices of odd degree

Sum of the degrees of a graph

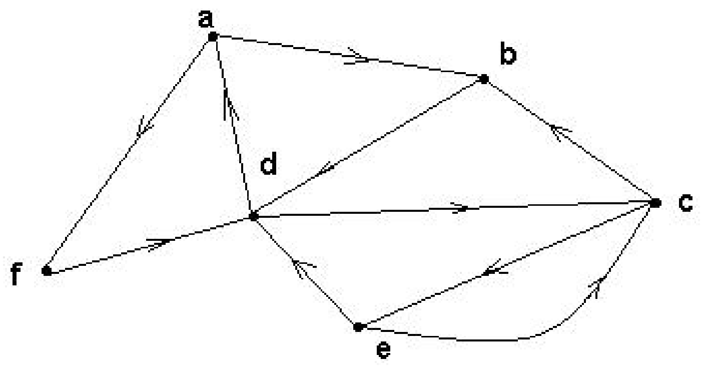
If G is a graph with m edges and n vertices $v_1, v_2, \ldots, v_n$, then

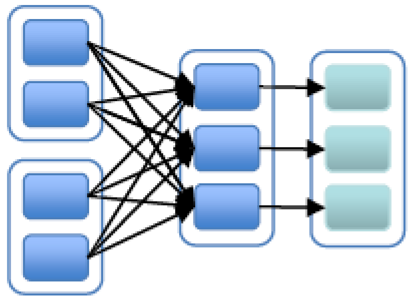
$$\sum_{i=1}^{n} \delta(v_i) = 2m$$

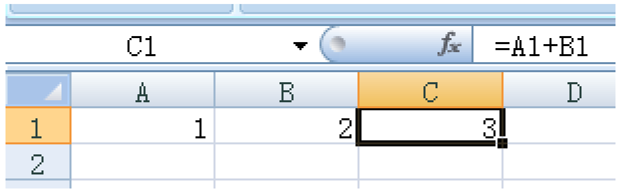
In particular, the sum of the degrees of all the vertices of a graph is even.

Directed graphs (digraphs)

- G is a *directed graph* or *digraph*
  - if each edge has been associated with an ordered pair of vertices, i.e. each edge has a direction
  - Examples:
    - Spark tasks dependency graph
    - Excel Spreadsheet formulas
    - Family trees



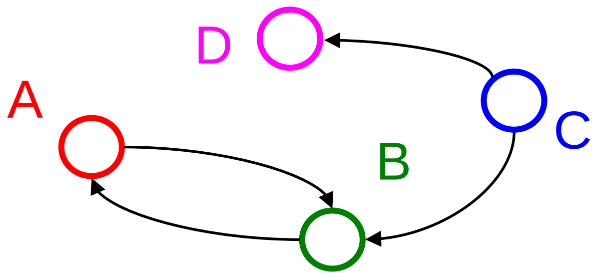




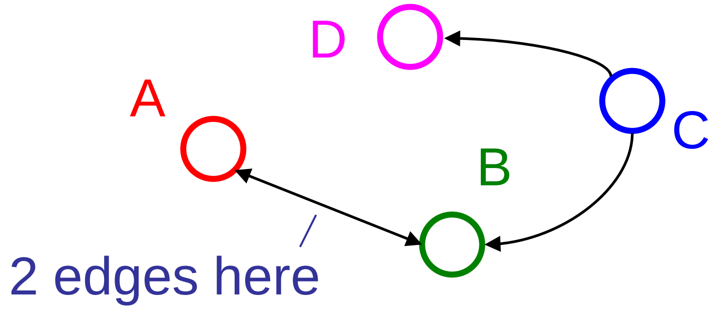
C1 =A1+B1

	A	B	C	D
1	1	2	3	
2				

Directed Graphs



or



In directed graphs (or digraphs), edges have direction

Thus, $(u, v) \in E$ does not imply $(v, u) \in E$.

- –such as "follow" in Facebook
- –Or "love"

Directed Graphs

Let $(u, v) \in E$ mean $u \rightarrow v$

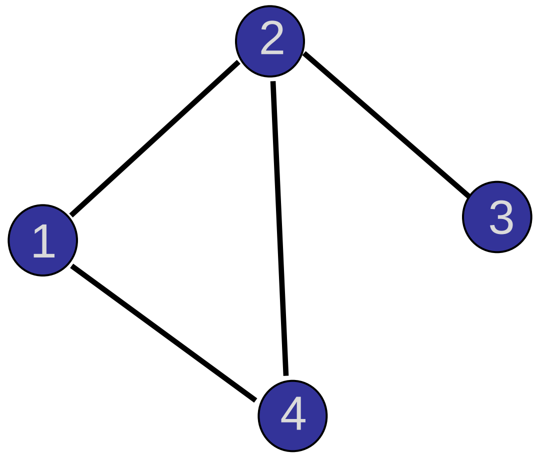
- Call u the source and v the destination

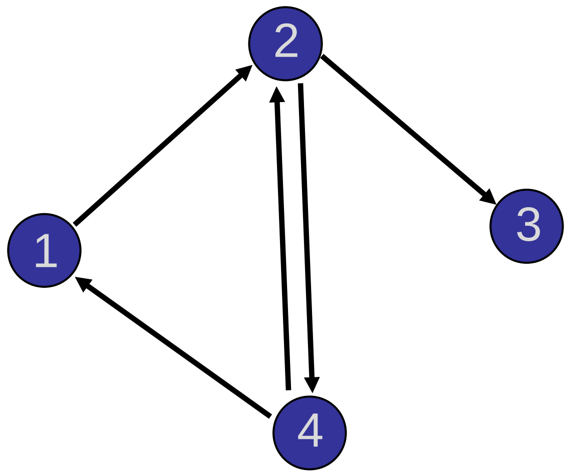
Degree in Directed Graphs

- In-Degree of a vertex: number of in-bound edges (edges where the vertex is the destination)
- Out-Degree of a vertex: number of out-bound edges (edges where the vertex is the source)

Degree in undirected graph and directed graph

- Degree of a vertex in an undirected graph is the number of edges incident on it.
- In a directed graph, the out degree of a vertex is the number of edges leaving it and the in degree is the number of edges entering it





The *degree* of vertex 2 is 3

The *in degree* of vertex 2 is 2 and the *in degree* of vertex 4 is 1

What does "Degree" mean in real world?

- Social media network
  - Popularity
  - Influence
  - Small-world effect
  - Marketing opportunities
- Search engine
  - Result ranking
  - Accuracy
- Investment Banking
  - Market analysis
  - Peer company analysis

Paths and Circuits

- A walk in a graph is an alternating sequence of adjacent vertices and edges
- A path is a walk that does not contain a repeated edge
- Simple path is a path that does not contain a repeated vertex
- A closed walk is a walk that starts and ends at the same vertex
- A circuit is a closed walk that does not contain a repeated edge
- A simple circuit is a circuit which does not have a repeated vertex except for the first and last

Shortest-path algorithm

- We say "a path exists from v_0 to v_n" if there is a list of vertices $[v_0, v_1, \ldots, v_n]$ such that $(v_i, v_{i+1}) \in E$ for all $0 \le i < n$.

- Shortest path
  - The shortest path length between two vertices *i* and *j* is the number of edges comprising the shortest path (or a shortest path) between *i* and *j*.

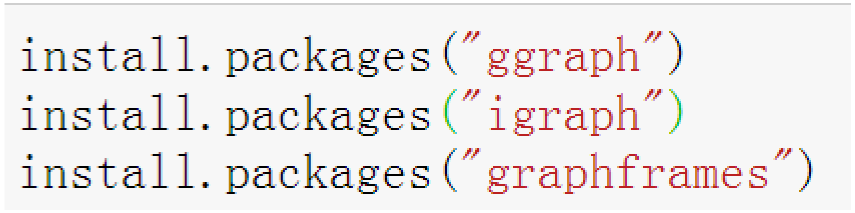
- Due to Edsger W. Dijkstra, Dutch computer scientist born in 1930
- Dijkstra's algorithm finds the length of the shortest path from a single vertex to any other vertex in a connected weighted graph.
- For a simple, connected, weighted graph with n vertices, Dijkstra's algorithms has worst-case run time $\Theta(n^2)$.

Shortest-path in real world

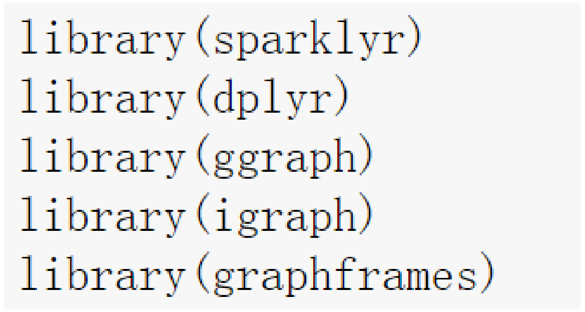
- Flight ticket with minimum stops
- Amazon Logistics
- GPS direction
- The flow of data in a network
- Social engineering

Graph Analytics in R with Spark

Install libraries



```r
install.packages("ggraph")
install.packages("igraph")
install.packages("graphframes")
```

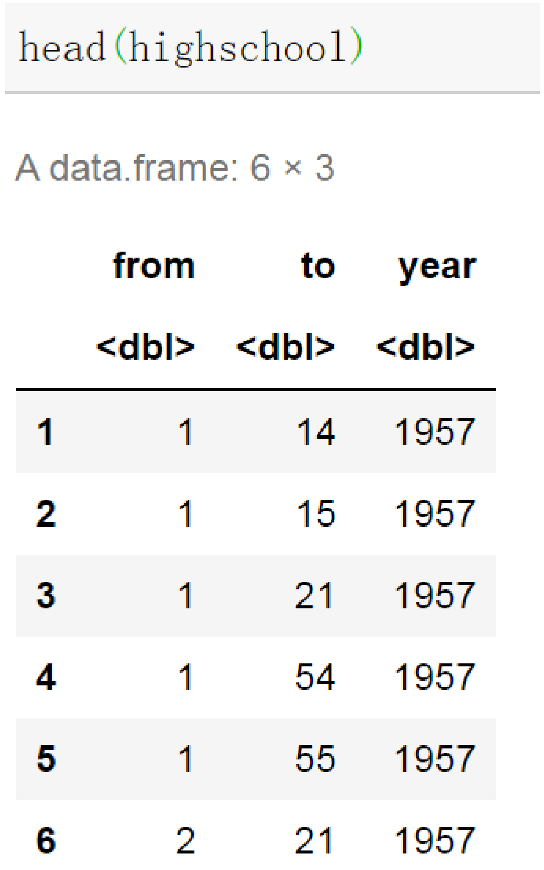


```r
library(sparklyr)
library(dplyr)
library(ggraph)
library(igraph)
library(graphframes)
```

- we use the highschool dataset from the ggraph package, which tracks friendship among high school boys.
- igraph is a library collection for creating and manipulating graphs and analyzing networks
- graphframes is a package for 'Apache Spark' that provides a DataFrame-based API for working with graphs.

- *Assume you already have sparklyr and dplyr*

Start from dataframe



```
head(highschool)
```

A data.frame: 6 × 3

	from	to	year
	<dbl>	<dbl>	<dbl>
1	1	14	1957
2	1	15	1957
3	1	21	1957
4	1	54	1957
5	1	55	1957
6	2	21	1957

- The vertices are the students and the edges describe pairs of students who happen to be friends in a particular year
- “from” and “to” are the student IDs
- “year” indicates the year when the two students were friends
- “friends” relationship is undirected edge, so we will build an undirected graph
- “friends” in our example doesn’t indicate weight, so the graph will be a simple undirected graph

The questions we want to answer

- From the dataframe, we can write R code to answer some basic questions
  - Who was the popular star in a particular year?
    - *in graph theory, find the vertices with maximum degree in a particular year*
  - How friendly was the atmosphere in a particular year?
    - *in graph theory, find the average degree in a particular year*

- The R code will have loops through the rows
  - slow
  - hard to reuse on other problems

Why need Spark?

•When the high school dataset is small, it can be processed in non-trivial R code (what's the time complexity?)

•Medium-size graph datasets can be very difficult to process

•Spark is well suited here to distribute the computation across a cluster of machines

•Spark supports processing graphs through the graphframes extension, which in turn uses the GraphX Spark component.

•GraphX is Apache Spark's API for graphs and graph-parallel computation. It's comparable in performance to the fastest specialized graph-processing systems and provides a growing library of graph algorithms.

Distributed dataframe in Spark

```
sc=spark_connect(master = "local")
highschool_tbl <- copy_to(sc, highschool, "highschool", overwrite = TRUE)
```

```
highschool_tbl
```

```
# Source: spark<highschool> [?? x 3]
    from    to  year
   <dbl> <dbl> <dbl>
 1     1    14  1957
 2     1    15  1957
 3     1    21  1957
 4     1    54  1957
 5     1    55  1957
 6     2    21  1957
 7     2    22  1957
 8     3     9  1957
 9     3    15  1957
10     4     5  1957
# ... with more rows
```

Same data, now distributed in Spark

Prepare data for graph

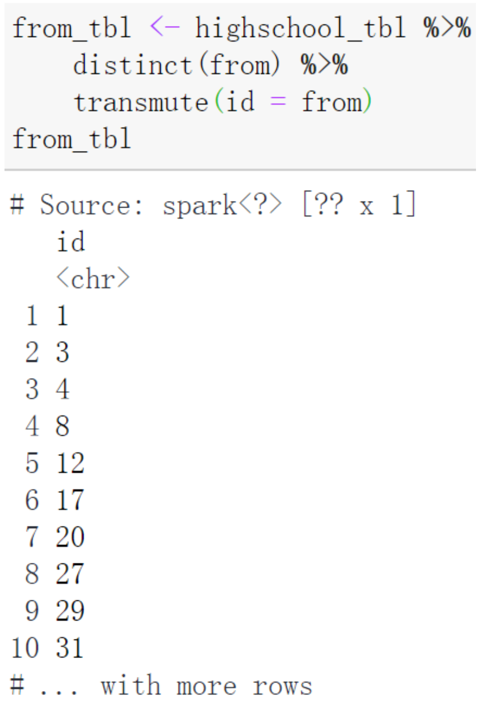
```r
highschool_tbl <- copy_to(sc,
                          highschool,
                          "highschool",
                          overwrite = TRUE) %>%
  filter(year == 1957) %>%
  transmute(from = as.character(as.integer(from)),
            to = as.character(as.integer(to)))
```

```r
highschool_tbl
```

```
# Source: spark<?> [?? x 2]
   from  to
   <chr> <chr>
 1 1     14
 2 1     15
 3 1     21
 4 1     54
 5 1     55
 6 2     21
 7 2     22
 8 3     9
 9 3     15
10 4     5
# ... with more rows
```

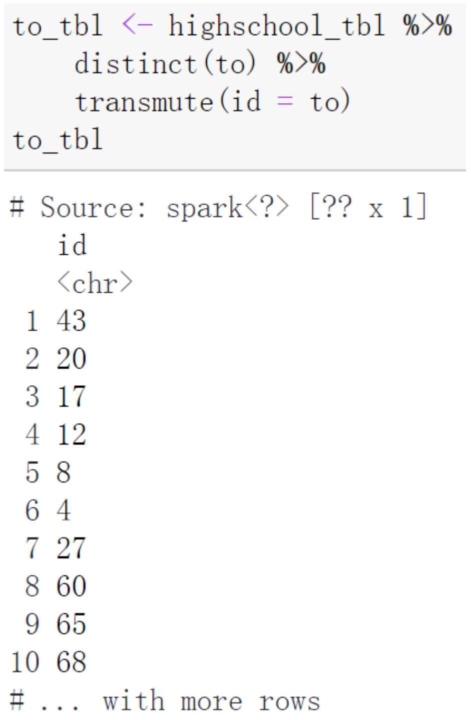
- We are only interested in the year of 1957
- We only need the “from” and “to” to build the simple undirected graph

Building graph step 1: vertices



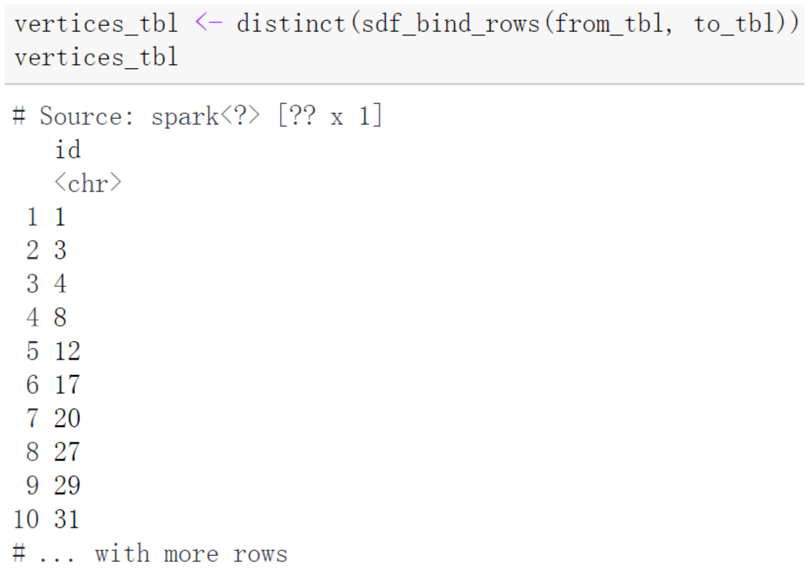
```
from_tbl <- highschool_tbl %>%
    distinct(from) %>%
    transmute(id = from)
from_tbl
```

```
# Source: spark<?> [?? x 1]
   id
   <chr>
 1 1
 2 3
 3 4
 4 8
 5 12
 6 17
 7 20
 8 27
 9 29
10 31
# ... with more rows
```



```
to_tbl <- highschool_tbl %>%
    distinct(to) %>%
    transmute(id = to)
to_tbl
```

```
# Source: spark<?> [?? x 1]
   id
   <chr>
 1 43
 2 20
 3 17
 4 12
 5 8
 6 4
 7 27
 8 60
 9 65
10 68
# ... with more rows
```



```
vertices_tbl <- distinct(sdf_bind_rows(from_tbl, to_tbl))
vertices_tbl
```

```
# Source: spark<?> [?? x 1]
   id
   <chr>
 1 1
 2 3
 3 4
 4 8
 5 12
 6 17
 7 20
 8 27
 9 29
10 31
# ... with more rows
```

- Collect all unique IDs from “from” and “to” columns.
- They are vertices in our graph.
- All functions above run on Spark, so the performance will be good on big data

Building graph step 2: edges

```
edges_tbl <- highschool_tbl %>%
    transmute(src = from, dst = to)
edges_tbl
```

```
# Source: spark<?> [?? x 2]
   src   dst
   <chr> <chr>
 1 1     14
 2 1     15
 3 1     21
 4 1     54
 5 1     55
 6 2     21
 7 2     22
 8 3     9
 9 3     15
10 4     5
# ... with more rows
```

- Just rename the columns to “src” and “dst”
- Each row represents an edge
- Undirected, so it doesn’t matter which column is called “src”.

Building graph step 3: graph

```
graph <- gf_graphframe(vertices_tbl, edges_tbl)
graph
```

```
GraphFrame
Vertices:
  Database: spark_connection
  $ id <chr> "1", "3", "4", "8", "12", "17", "20", "27", "29", "31", "34", "3...
Edges:
  Database: spark_connection
  $ src <chr> "1", "1", "1", "1", "1", "2", "2", "3", "3", "4", "4", "4", "4"...
  $ dst <chr> "14", "15", "21", "54", "55", "21", "22", "9", "15", "5", "18",...
```

- Use vertices and edges to build a graph
- graphframe is stored in a distributed mode on Spark cluster machines

Ready to answer questions

- Who was the popular star in a particular year?
  - *in graph theory, find the vertices with maximum degree in a particular year*

```
gf_degrees(graph)
```

```
# Source: spark<?> [?? x 2]
   id     degree
   <chr>   <int>
 1 1           5
 2 3           2
 3 4           5
 4 43         12
 5 20         11
 6 17          5
 7 8           3
 8 12          5
 9 27          4
10 60          7
# ... with more rows
```

```
gf_degrees(graph) %>% arrange(desc(degree))
```

```
# Source:      spark<?> [?? x 2]
# Ordered by: desc(degree)
   id     degree
   <chr>   <int>
 1 71         16
 2 21         14
 3 22         14
 4 70         13
 5 54         13
 6 69         12
 7 43         12
 8 66         12
 9 52         11
10 67         11
# ... with more rows
```

Ready to answer questions

- How friendly was the atmosphere in a particular year?
  - *in graph theory, find the average degree in a particular year*

In the year of 1957

```
gf_degrees(graph) %>% summarise(friends = mean(degree, na.rm = TRUE))
```

```
# Source: spark<?> [?? x 1]
  friends
    <dbl>
1    6.94
```

In the year of 1958

```
gf_degrees(graph) %>% summarise(friends = mean(degree, na.rm = TRUE))
```

```
# Source: spark<?> [?? x 1]
  friends
    <dbl>
1    7.62
```

It shows the class of 1958 was more friendly than 1957!

Shortest-path question

- what are everybody's shortest distances from one student in a particular year?
    - *in graph theory, find the shortest-path from every vertex to a particular vertex in a particular year*

```
gf_shortest_paths(graph, 14) %>%
  filter(size(distances) > 0) %>%
  mutate(distance = explode(map_values(distances))) %>%
  select(id, distance)
```

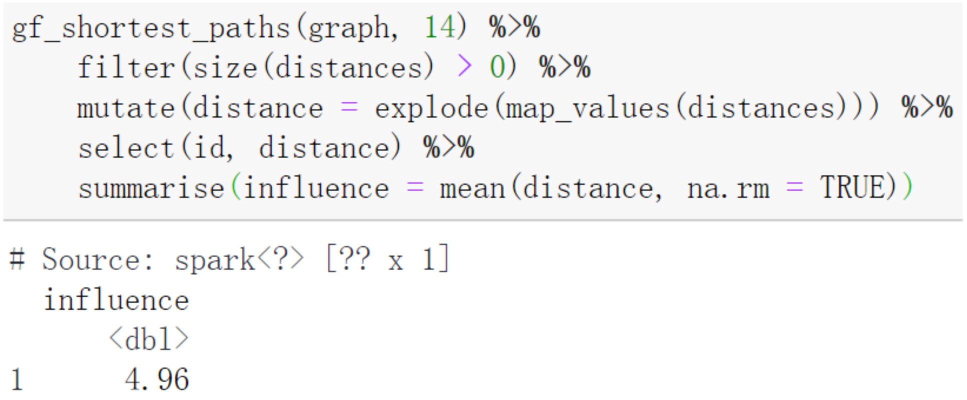
```
# Source: spark<?> [?? x 2]
   id     distance
   <chr>     <int>
 1 17            2
 2 29            4
 3 34            6
 4 8             1
 5 13            3
 6 14            0
 7 22            6
 8 24            8
 9 28            6
10 38            7
# ... with more rows
```

```
filter(edges_tbl, dst==14)
```

```
# Source: spark<?> [?? x 2]
  src   dst
  <chr> <chr>
1 1     14
2 8     14
```

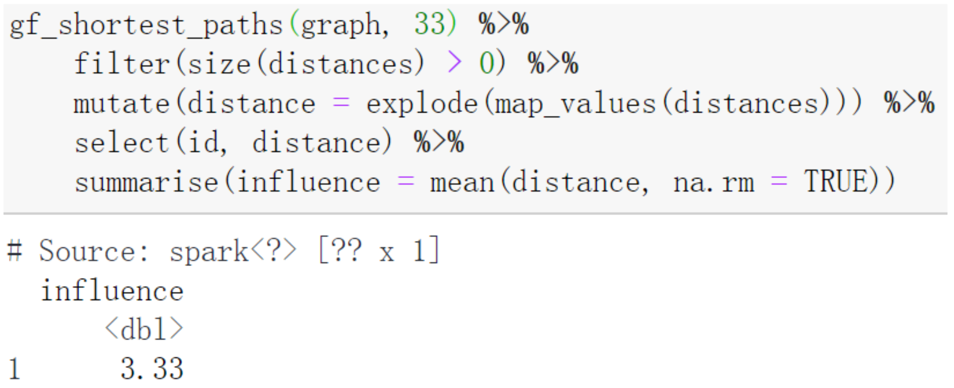
Shortest-path question

- Who has more influence in his friends circle?
  - *in graph theory, find the <u>average shortest-path</u> to a vertex in a particular year*



```
gf_shortest_paths(graph, 14) %>%
    filter(size(distances) > 0) %>%
    mutate(distance = explode(map_values(distances))) %>%
    select(id, distance) %>%
    summarise(influence = mean(distance, na.rm = TRUE))
```

```
# Source: spark<?> [?? x 1]
  influence
      <dbl>
1      4.96
```



```
gf_shortest_paths(graph, 33) %>%
    filter(size(distances) > 0) %>%
    mutate(distance = explode(map_values(distances))) %>%
    select(id, distance) %>%
    summarise(influence = mean(distance, na.rm = TRUE))
```

```
# Source: spark<?> [?? x 1]
  influence
      <dbl>
1      3.33
```

It shows that in the class of 1957, the student 33 has more influence (less average distance) than student 14 in their friends circle.

If I launch a marketing campaign, which student would be my target?

Summary

• Graphs are a powerful representation and have been studied deeply
• Many real world problems are fundamentally graph problems.
• R with Spark provides a high performance framework to solve all these types of problems.

•Read more
    •https://therinspark.com/extensions.html#graphs
    •https://spark.rstudio.com/graphframes/
    •https://github.com/rstudio/graphframes

•Exercise of Today
    •Follow the jupyter notebook to install libs and run the sample code