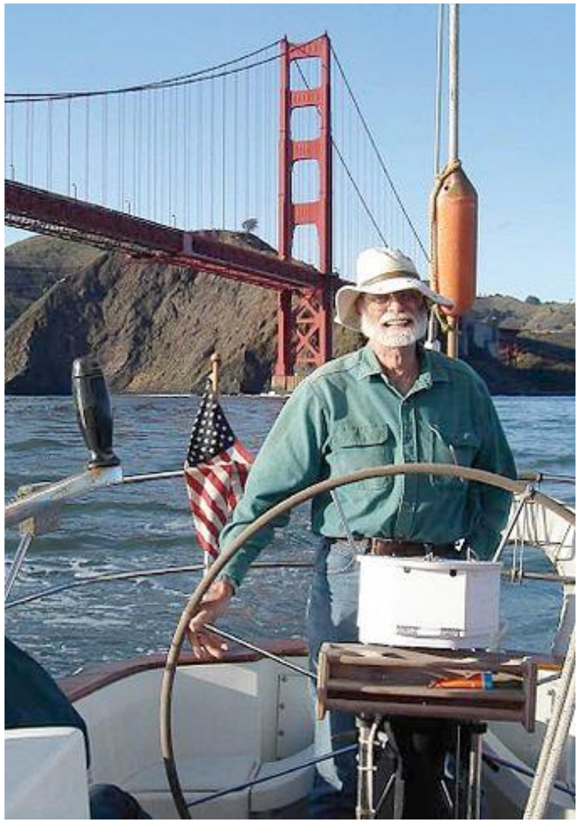
MIS 573:

Practical Big Data Analytics

Fast Scalable Data Analytics using R, H2O, and Spark

Yihuang K. Kang



"Memory is the new disk, and disk is the new tape."

[Jim Gray]

Toward Fast Scalable Data Analytics

- The explosion of data growth has called for the need for scalable and (nearly) real-time data analysis. Indeed, with help of R and its comprehensive package repositories, we can virtually do any kinds of data analysis tasks. However, we are still experiencing high latency when dealing with large datasets or computation-intensive analysis tasks.

- Practitioner and researchers have been proposing new data processing framework (e.g. Apache Hadoop) that incorporates and coordinates a cluster of computers to solve such problems. Most of them are featured with high availability and fault-tolerance data management. Nevertheless, those centralized-managed systems, cumbersome cluster installation, and bath data processing have hindered them from adoption of fast scalable data analysis applications.

Introduction to H2O

- H2O framework developed by *H2O.ai* is the product aimed at solving such problems. Its cross-platform (on JVMs) and peer-to-peer computing architecture allows us to easily set up a flexible & running-on-demand computing cluster with any sizes. Its built-in distributed data processing and machine learning libraries can handle billions of data rows in-memory, even with a small cluster.

- A good news for R users is that it is originally designed for running R programs. The R package *h2o* (can be found on CRAN) provides functions that support R's data manipulation and analysis syntax. R users can easily write scripts to control remote distributed data frames on H2O cluster.

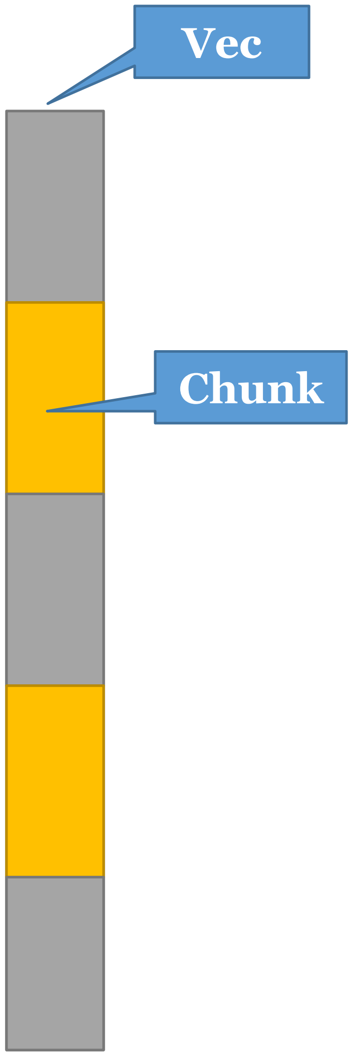
- Another feature is that we can use H2O REST APIs to get access to all the capabilities of H2O from external programs or scripts written in other languages (e.g. Python, Java, etc.), via JSON over HTTP. And the built-in *H2O Flow* web interface make it even easier for non-engineers to perform data analysis in workflow manner.

H2O Objects

- H2O uses a *distributed key-value store* (DKV) that contains pointers to the various objects of the H2O ecosystem. The DKV is a kind of container that encapsulates the following shared objects.

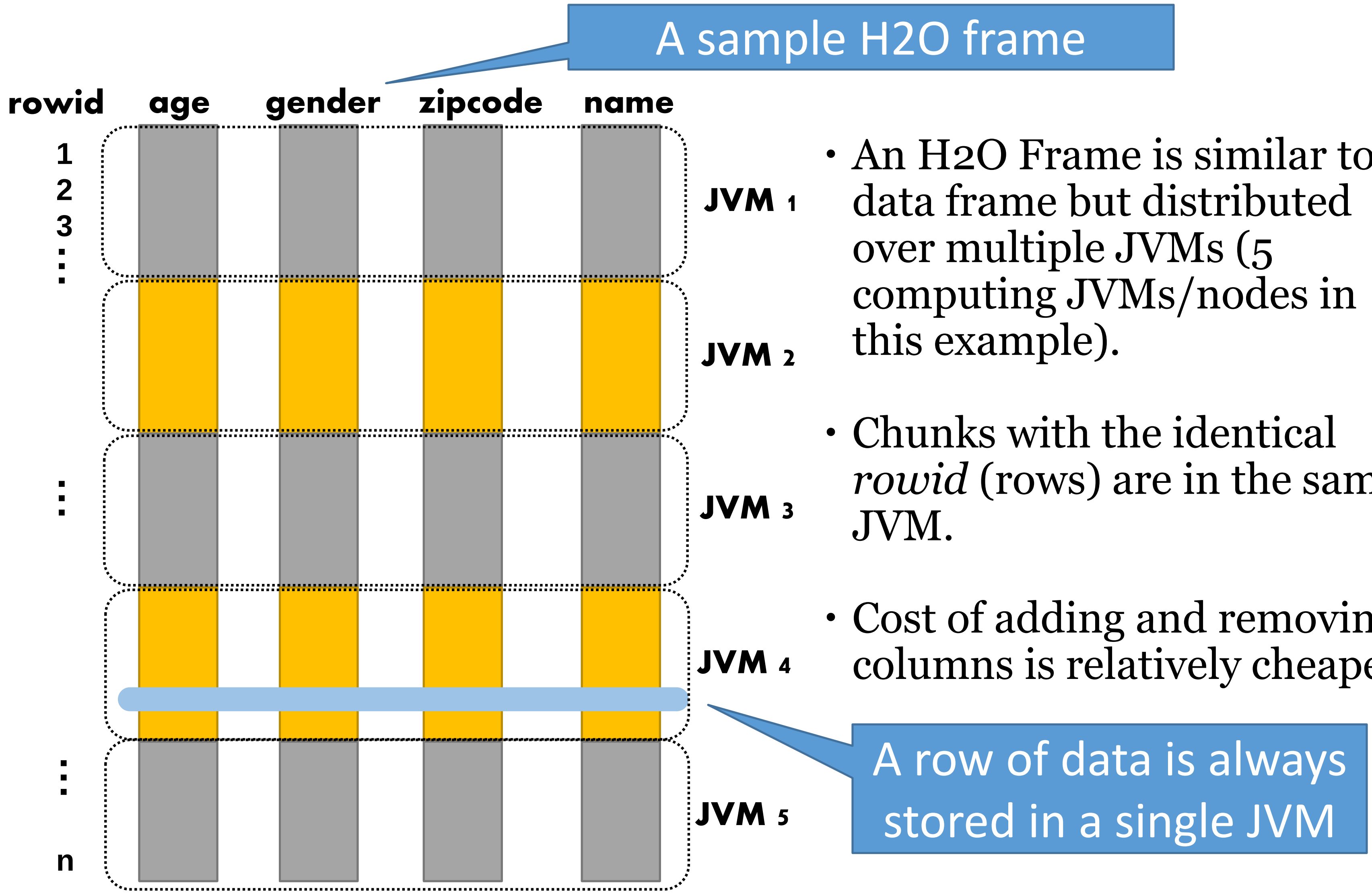
Primitive	The smallest object that contains data in a Java primitive data type–a byte, char, long or double.
Chunk	A *Chunk* holds a fraction of the data (Vec). It is a 1D array of elements.
Vec	A *Vec* is a collection of Chunk objects. It may be divided into parts distributed in a cluster.
Frame	A 2D collection of *Vec* objects.
ModelMetrics	A collection of metrics for a given category of model.
Model	An immutable object having predictive and metrics methods.
Job	A non-blocking task that performs a finite amount of work.
Key	An entry in the DKV that maps to an object in H2O.

H2O Objects(cont.)



- An H2O *Vec* (vector) is a compressed column distributed over multiple JVMs. It may contain billions of Java primitives in the same data types.
- In H2O Frame, a Vec is split into *Chunks*. Each Chunk may contain hundreds or thousands of elements.
- A Chunk is the minimum unit of parallel processing. It supports fast random access and on-the-fly data decompression.

H2O Objects(cont.)



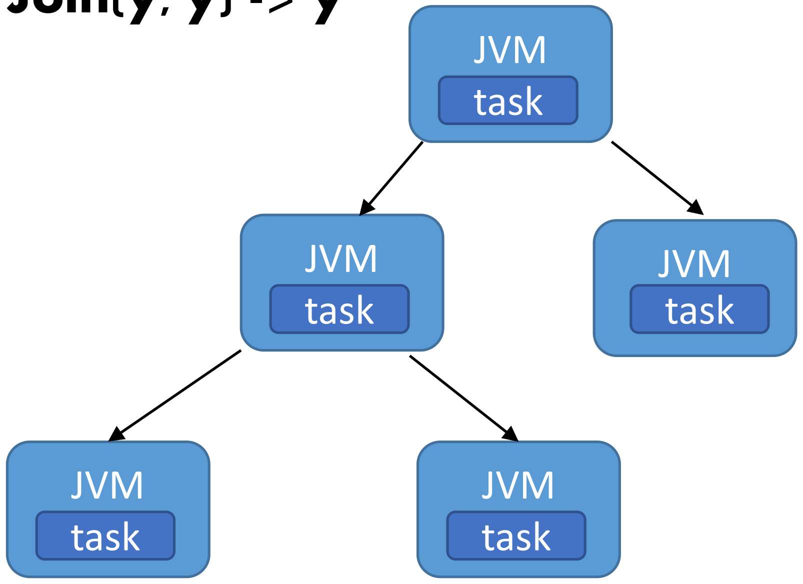
- An H2O Frame is similar to R data frame but distributed over multiple JVMs (5 computing JVMs/nodes in this example).
- Chunks with the identical *rowid* (rows) are in the same JVM.
- Cost of adding and removing columns is relatively cheaper.

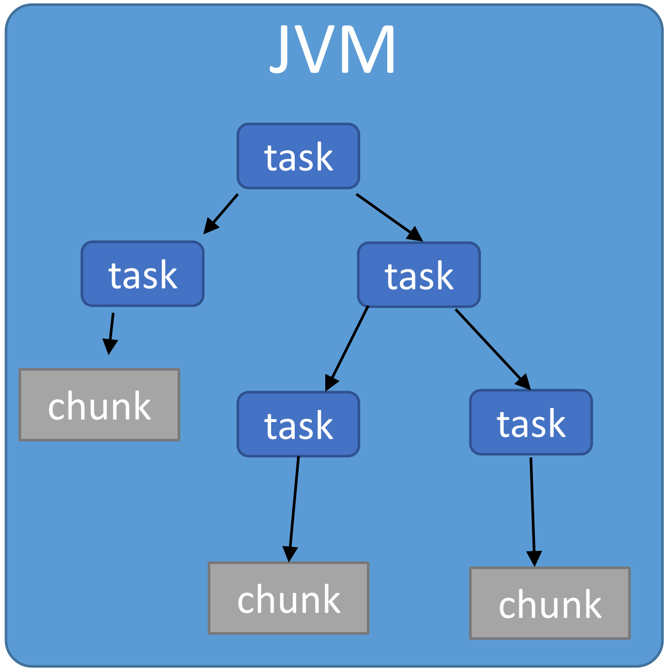
H2O Fork/Join

- H2O has its own data processing model, Java *Fork/Join framework,* which is similar to *MapReduce* in Apache Hadoop. *Fork*() parallelizes and splits tasks in tree pattern, whereas *Join*() reduces results at each inner node and returns them when all subtasks are done.

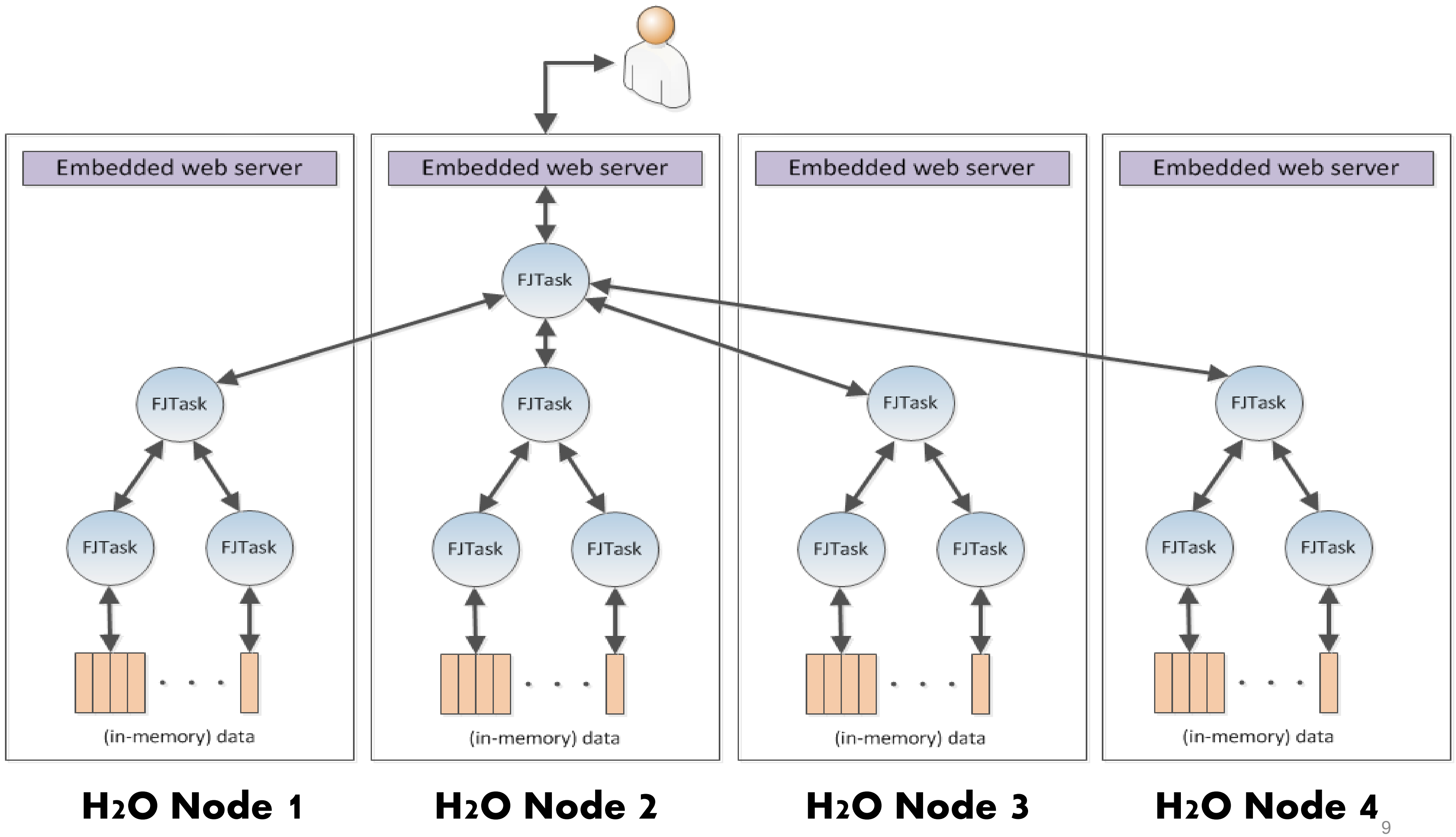
Fork(x) -> y

Join(y, y) -> y

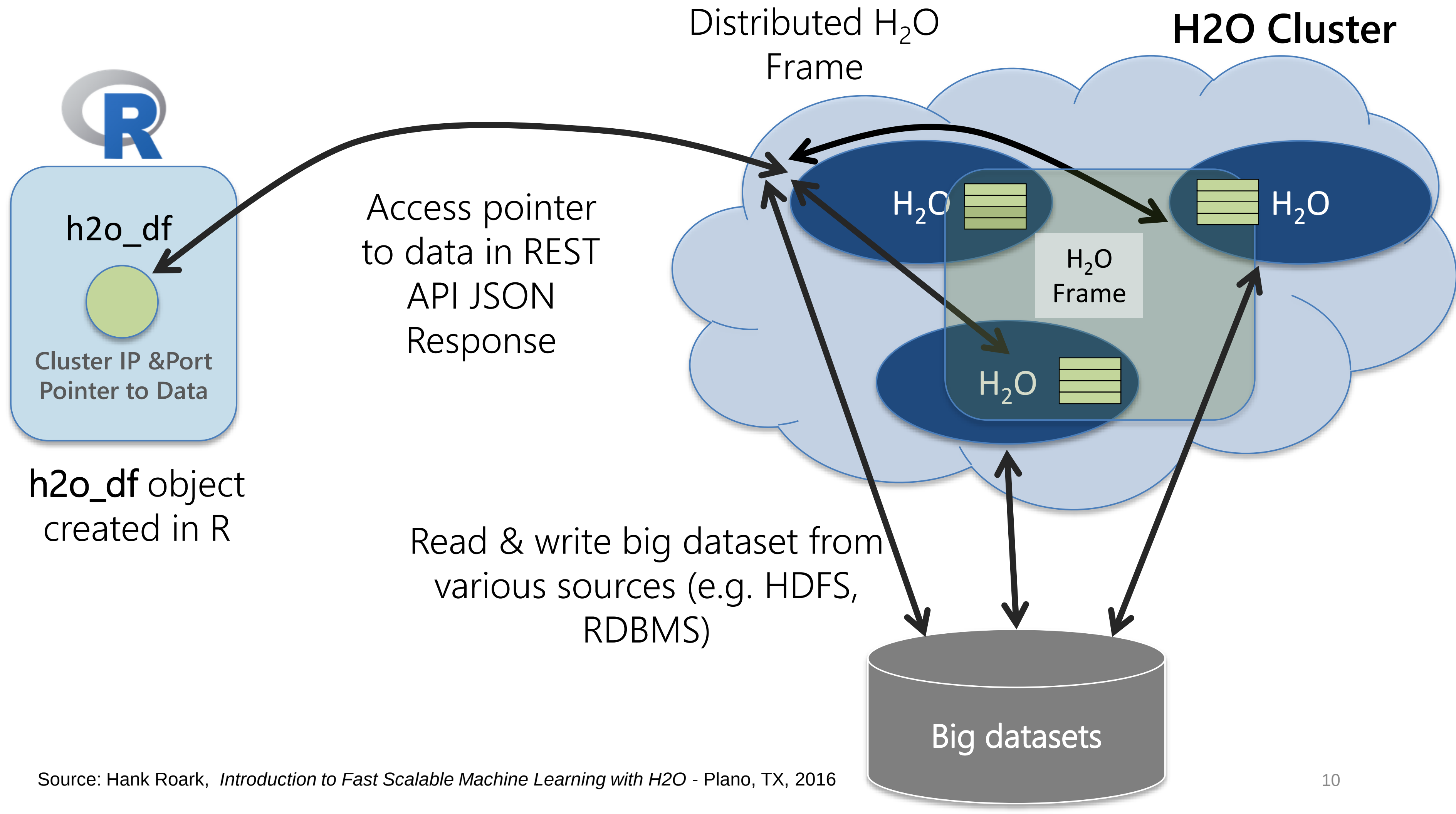




H2O Fork/Join(cont.)



Accessing H2O distributed data frame


Distributed H_2O Frame
H2O Cluster
h2o_df
Cluster IP &Port
Pointer to Data
h2o_df object created in R
Access pointer to data in REST API JSON Response
H_2O
H_2O
H_2O Frame
H_2O
Read & write big dataset from various sources (e.g. HDFS, RDBMS)
Big datasets

Source: Hank Roark, *Introduction to Fast Scalable Machine Learning with H2O* - Plano, TX, 2016

Hands on H2OFrame operations

- We can easily get access to an H2O cluster, load data from any sources, and create H2O data frames in R using *H2O REST APIs*.

```
# Load H2O
library(h2o)
# Create a connection to an existing H2O cluster
remoteH2O = h2o.init(ip="localhost", port = 54321,
  startH2O = F, nthreads = -1)
# Or you can create a local 1-node H2O cluster with limited memory
# NOTE: DO NOT run the following code on the CM server
##  localH2O = h2o.init(ip = "localhost", port = 54321,
##  startH2O = T, max_mem_size = "2G", nthreads = -1)
# Shutdown H2O cluster
## h2o.shutdown()
# Checking the cluster information
h2o.clusterInfo()
# Check out overall network connection speed.
# NOTE: It's for testing purpose. DO NOT run the following
# code on the NSYSU CM server
## h2o.networkTest()
```

Hands on H2OFrame operations(cont.)

```
# Import remote IRIS data into H2O as an H2OFrame
iris.hex = h2o.importFile("https://archive.ics.uci.edu/ml/machine-
learning-databases/iris/iris.data",
                         destination_frame = "iris.hex",sep = ",")
iris.hex
# iris.hex is actually a pointer to the
# H2OFrame in remote H2O cluster
class(iris.hex); object.size(iris.hex); str(iris.hex)

# Get more info. about "iris.hex" H2OFrame via H2O REST APIs
library(jsonlite)
# Basic frame info.
iris.hex.info = fromJSON("http://localhost:54321/3/Frames")
# What is the URL for the frame?
iris.hex.info$frames$frame_id
# More detail about "iris.hex"
iris.hex.detail =
   fromJSON("http://localhost:54321/4/Frames/iris.hex")
# Summary of "iris.hex" distribution
iris.hex.detail$frames$distribution_summary
```

Hands on H2OFrame operations(cont.)

- We can see that the input/output of the REST APIs are in JSONs, which requires additional data processing and interpretations. Unfortunately, until H2O R 3.8.2.3, some of H2O cluster & H2OFrame operations could only be done via REST APIs. However, there does have some R native functions that could make your life easier.

```
# Get info. of an H2OFrame, similar to functions in base R
str(iris.hex); nrow(iris.hex); summary(iris.hex, exact_quantiles = T)

# Or, just write a function that provides readable distribution info.
getH2OFrameDistInfo = function(h2oFrameName, host = "127.0.0.1",
 port = 54321){
  library(jsonlite)
  frameInfoJSON = fromJSON(paste("http://", host,":", port,
     "/4/Frames/", h2oFrameName, sep = ""))
  hexDistInfo =
   as.matrix(t(frameInfoJSON$frames$distribution_summary$data[[1]]))
  colnames(hexDistInfo) =
    frameInfoJSON$frames$distribution_summary$columns[[1]]$description
  return(hexDistInfo)
}
getH2OFrameDistInfo(h2oFrameName = "iris.hex")
```

Hands on H2OFrame operations(cont.)

```
# Another dataset, airlines Delays dataset, 31 columns
allyears2k.hex = h2o.importFile("https://s3.amazonaws.com/h2o-airlines-
unpacked/allyears2k.csv", header = T, destination_frame =
   "allyears2k.hex", sep = ",", na.strings = "NA")
getH2OFrameDistInfo("allyears2k.hex")

colnames(allyears2k.hex) # column names
# List all existing H2O objects (keys)
h2o.ls()

# Only 1-way frequency table is supported by h2o.table()
h2o.table(allyears2k.hex$IsDepDelayed)

# It creates crosstab in flat way
flatCrosstab = h2o.tabulate(allyears2k.hex, "IsDepDelayed",
   "IsArrDelayed" )

# Get our familiar "crosstab". Note that "as.data.frame()" save
# H2OFrame as local R Data Frame
xtabs(counts ~ isdepdelayed + isarrdelayed,
   as.data.frame(flatCrosstab$count_table))
```

Hands on H2OFrame operations(cont.)

- Note that *H2OFrame operations are done remotely by H2O built-in functions*, so passing our R functions (including user-defined functions) as the input of the H2O functions may not work. For example, our anonymous functions cannot actually "*return*()" a value to the H2O cluster. Instead, H2O functions (e.g. *h2o.ddply*()) simply take the result of last expression in the input R function as the output.

```
# Split-Apply-Combine
# ERROR: H2O doesn't recognize both function "return()" and "class()"
h2o.ddply(allyears2k.hex, .variables = "Year",
   FUN = function(df){ return(class(df)) } )
Error in .process.stmnt(stmnt, formalz, envs) :
  Don't know what to do with statement: class df

# Get average distance by "Year" x "IsDepDelayed"
# Note that we can only access columns of the intermediate
# H2OFrames by column numbers.
meanDelayedDistance = h2o.ddply(allyears2k.hex, .variables = c("Year",
   "IsDepDelayed"), FUN = function(df) {mean(df[, 19])} )
# Print all rows
head(meanDelayedDistance, n = nrow(meanDelayedDistance))
```

Hands on H2OFrame operations(cont.)

- Every operation that outputs an H2OFrame would create a temporary pointer (H2O key) to a new H2OFrame, but the frame is not necessarily instantiated—a *lazy evaluation* strategy of H2O. One major reason of doing it is that it could improve the performance by avoiding unnecessary calculations and memory consumption.

```
# Checking all keys again in H2O
h2o.ls()
# A column vector of an H2OFrame is still a (1-D) H2OFrame
class(allyears2k.hex$IsDepDelayed)
# There is a key for the column, but the column doesn't actually exist!
h2o.getId(allyears2k.hex$IsDepDelayed)

# Instantiate (create a copy of) the column and
# assign a new key for it
h2o.assign(allyears2k.hex$IsDepDelayed, "IsDepDelayed.hex")
h2o.ls(); getH2OFrameDistInfo("IsDepDelayed.hex")

# You may remove everything in H2O by running
# h2o.removeAll()
```

Hands on H2OFrame operations(cont.)

- H2O R only supports some basic R data frame-like operations, so the best strategy of working with H2O is to consider H2OFrame as a pointer to an external object before combining existing R functions with H2O functions.

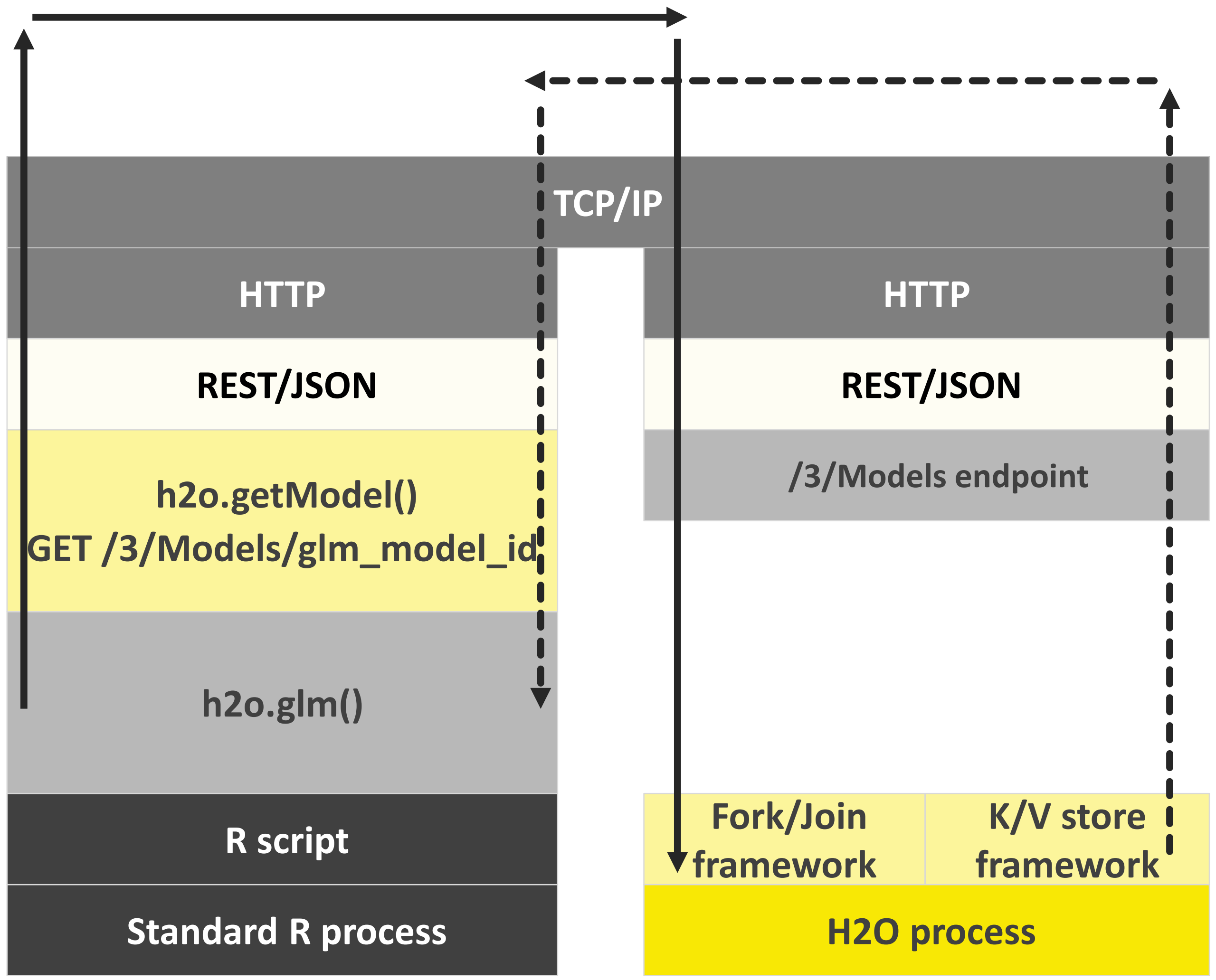
```
# H2OFrame R environment is just a pointer,
# so it can be enclosed in an R list
listOfH2OFrame = list(allyears2k.hex$Year, allyears2k.hex$IsDepDelayed)
# Frequency tables of two vectors/columns. Note that the output of H2O
# functions are still H2OFrames (pointers in R)
freqTbl = Map(function(v) h2o.table(v), listOfH2OFrame)
# What are the keys/IDs for both H2OFrames (frequency tables)?
freqTblKeys = Map(h2o.getId, freqTbl)
# Let's convert both tables into "local" R data frames
freqTblLocal = Map(function(h2oKey) as.data.frame(h2o.getFrame(h2oKey)),
   freqTblKeys)

# Another example to remove those columns with 50%+ NAs.
Map(function(vec){
    if(sum(is.na(vec)) > (nrow(allyears2k.hex) / 2))
      allyears2k.hex[, colnames(vec) ] <<- NULL
}, allyears2k.hex)
```

H2O model fitting via R

- H2O supports most of popular & advanced machine learning algorithms, e.g. Generalized Linear Models, Random Forest, Principal Components Analysis, and even some of Deep Learning algorithms. Fitting models using H2O is as smooth as what we're doing in local R.

```r
# Fitting GLM with diamonds data
library(ggplot2); diamonds_local = as.data.frame(diamonds)
diamonds_local$price = log(diamonds_local$price);
colnames(diamonds_local)[7] = "logPrice"
# Unfortunately, H2O doesn't recognize "ordered factors"
diamonds_local = data.frame(Map(function(x) if(is.factor(x))
   as.character(x) else x , diamonds_local ), stringsAsFactors = F)
diamonds.hex = as.h2o(diamonds_local, "diamonds.hex")
getH2OFrameDistInfo("diamonds.hex") # Get H2OFrame info.
# Convert string into h2o factors
Map(function(vecName) diamonds.hex[,vecName] <<-
   h2o::as.factor(diamonds.hex[,vecName]), list("cut", "clarity", "color"))
# Differences between R glm() and H2O h2o.glm()
dm_glm_local <- glm(logPrice ~ ., family = "gaussian",
   data = diamonds_local)
dm_glm_h2o <- h2o.glm(x = (1:10)[-7], y = "logPrice", family = "gaussian",
   training_frame = diamonds.hex, model_id = "dm_glm_h2o")
```

H2O model fitting via R(cont.)



Source: Hank Roark, *Introduction to Fast Scalable Machine Learning with H2O* - Plano, TX, 2016

H2O model fitting via R(cont.)

- So, where exactly is my H2OModels? As discussed previously, we can do almost all H2O cluster operations via H2O REST APIs.

```r
# Get all my H2OModels using REST APIs, where
# HTTP GET "/4/Models" returns all Models
# HTTP GET "/4/Models/myModelID" returns model "myModelID" info. in JSON
AllH2OModelInfo = fromJSON("http://localhost:54321/4/Models")
AllH2OModelInfo$models$model_id
# Previous model with ID "dm_glm_h2o"
dm_glm_h2o_info = fromJSON("http://localhost:54321/4/Models/dm_glm_h2o")
dm_glm_h2o_info$models
```

- However, it looks a bit awkward. A few H2O R functions may help.

```r
# We can always get our H2OModel back as long as we have the ID
# and the model is still in the cluster.
h2o.ls();  dm_glm_h2o_copy = h2o.getModel("dm_glm_h2o")
# Also, H2OModels can be saved as binary objects in local storage
# and load it back to H2O cluster later as needed.
h2o.saveModel(dm_glm_h2o_copy, "./")
h2o.rm(dm_glm_h2o) # Remove the H2O object by the key "dm_glm_h2o"
# Load the model back to the cluster
dm_glm_h2o = h2o.loadModel("./dm_glm_h2o")
```

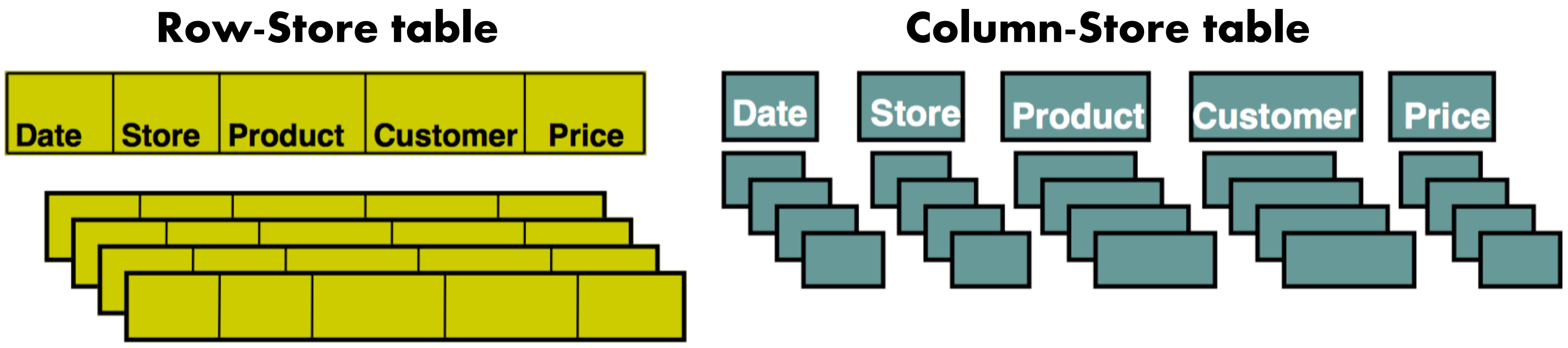
Useful things to know about H2O

- H2O framework with its machine learning library provides us a friendly interface to process & analyze massive datasets. However, there are things to know before introducing it to your analytics platform.
  - *H2O may not work when dataset is much bigger than total cluster memory*. H2O is fast simply because it does almost everything (compressing, partitioning, and distributing datasets across all nodes) in-memory, with the cost of less generic data frames (compared to other framework like Spark RDDs) and the benefit of highly-compressed data frames.
  - *H2O does not "persist"*. For performance reasons, H2O uses all available memory to process its own data frames. As there is no explicit caching process in H2O, we should always save the results from H2O cluster before shutting it down.
  - *H2O does not support high availability (HA)*. As you may notice, H2OFrames are distributed row-wisely and a row is always stored in one node. When any node is down or unavailable, everything in the cluster may become inaccessible.

A bit about column-store data table

- In recent decades, data management vendors have been implementing columnar storage format, where data is stored in columns instead of rows, into their product to speed up read-mostly or read-intensively massive data processing tasks.

- One major advantage of using column-store format is that, as most of queries on tables only deal with a small subset of fields, not the entire rows, it can save large amount of I/O time when accessing data tables.

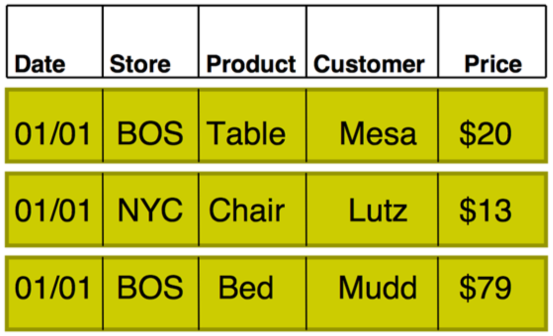


Source: Stavros Harizopoulos, Daniel Abadi, Peter Boncz, " *Column-Oriented Database Systems*", VLDB Tutorial, 2009

A bit about column-store data table(cont.)

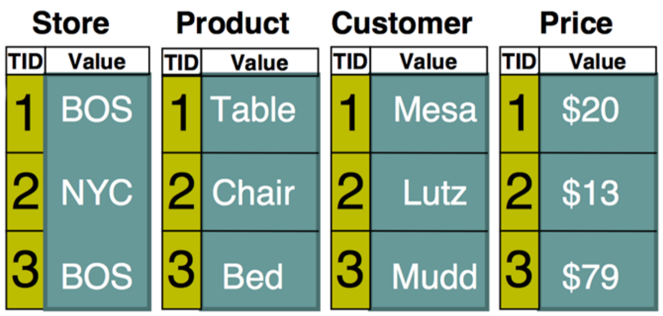
- Row-store table is easy to add/modify records but may read unnecessary data (columns). Column-store table, however, imposes more overhead on writing records but only need to read required columns.
- Another advantage of using column-store is that data stored in columns is more compressible (which as improves I/O performance), because of its low entropy data values (high data value locality). For example, the same value in below "Date" column can be easily run-length encoded, as the attribute has the same data domain.

Row-Store table



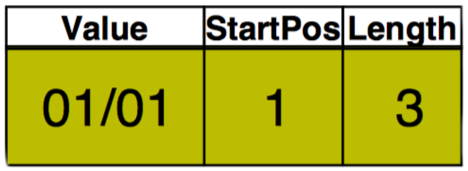
Date	Store	Product	Customer	Price
01/01	BOS	Table	Mesa	$20
01/01	NYC	Chair	Lutz	$13
01/01	BOS	Bed	Mudd	$79

Column-Store table



Store		Product		Customer		Price	
TID	Value	TID	Value	TID	Value	TID	Value
1	BOS	1	Table	1	Mesa	1	$20
2	NYC	2	Chair	2	Lutz	2	$13
3	BOS	3	Bed	3	Mudd	3	$79

Run-length encoded "Date"

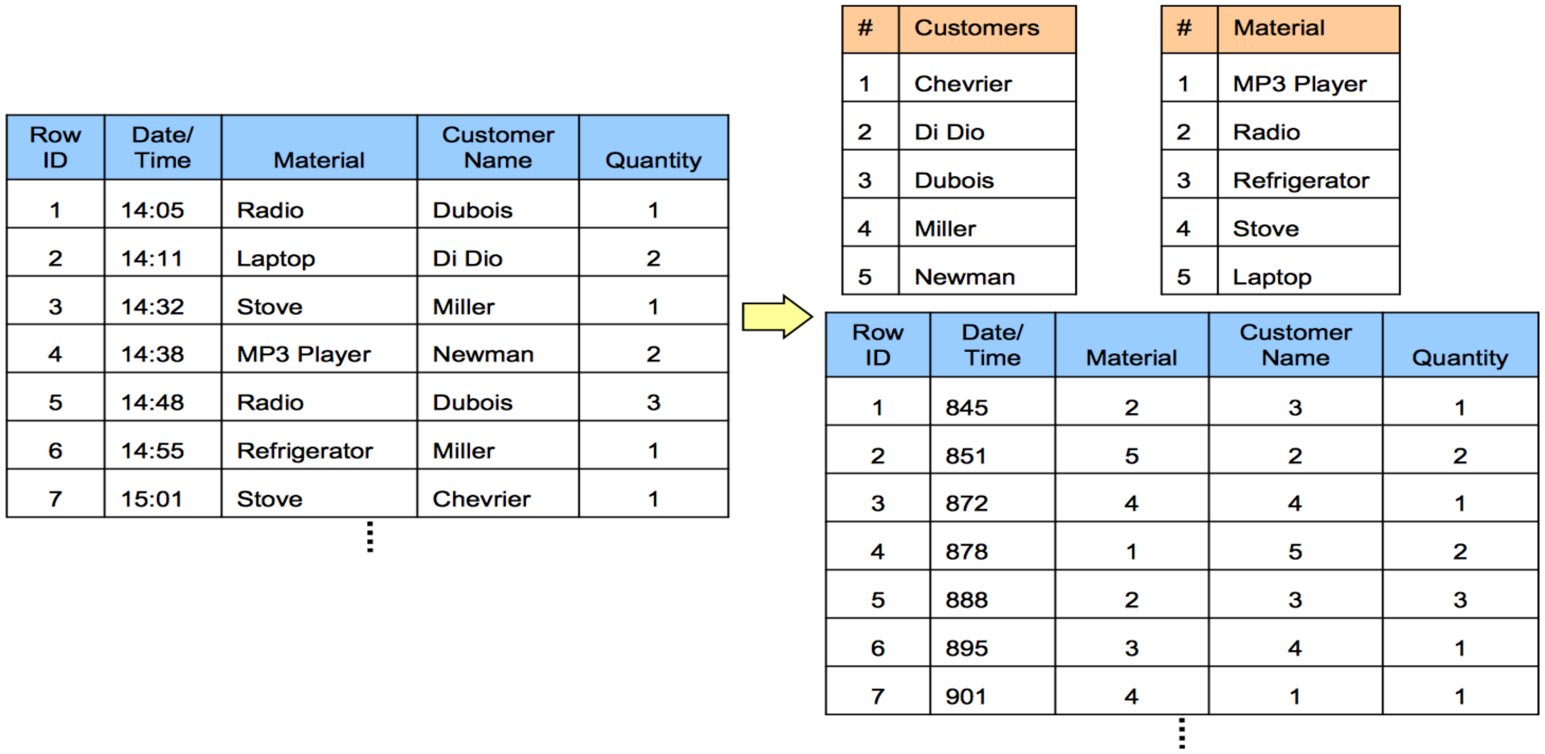


Value	StartPos	Length
01/01	1	3

Source: Stavros Harizopoulos, Daniel Abadi, Peter Boncz, " *Column-Oriented Database Systems*", VLDB Tutorial, 2009

A bit about column-store data table(cont.)

- Here is another example about *dictionary compression* of column-store data. The dictionaries are here used to represent text as integer numbers. As the column is all in integer, the database can compress the dataset significantly and later reduce data movement.



Row ID	Date/ Time	Material	Customer Name	Quantity
1	14:05	Radio	Dubois	1
2	14:11	Laptop	Di Dio	2
3	14:32	Stove	Miller	1
4	14:38	MP3 Player	Newman	2
5	14:48	Radio	Dubois	3
6	14:55	Refrigerator	Miller	1
7	15:01	Stove	Chevrier	1
		⋮		

#	Customers
1	Chevrier
2	Di Dio
3	Dubois
4	Miller
5	Newman

#	Material
1	MP3 Player
2	Radio
3	Refrigerator
4	Stove
5	Laptop

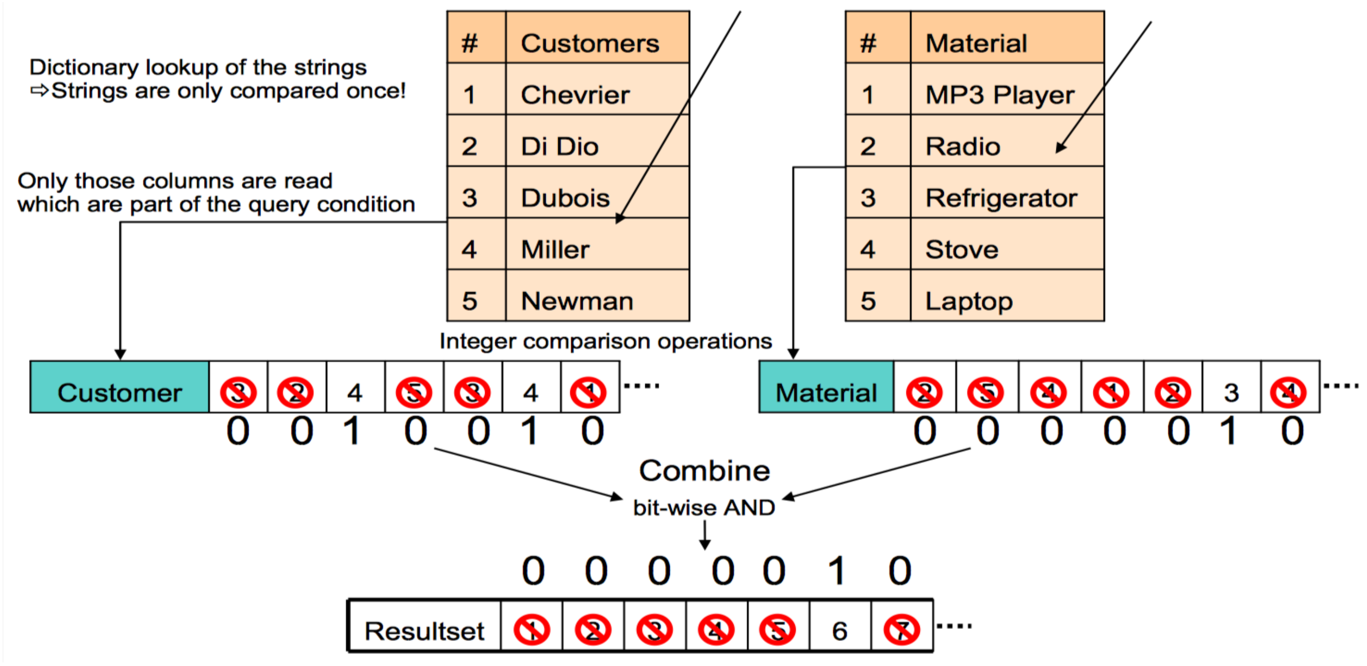
Row ID	Date/ Time	Material	Customer Name	Quantity
1	845	2	3	1
2	851	5	2	2
3	872	4	4	1
4	878	1	5	2
5	888	2	3	3
6	895	3	4	1
7	901	4	1	1
			⋮	

Source: "In-memory Computing with SAP HANA on Lenovo X6 Systems", Lenovo Press

A bit about column-store data table(cont.)

- The columnar data with dictionary compression can also facilitate query executions. Below query shows the columns are scanned for values that match the query condition. The other columns are not required for the selection process and the selection marks can be represented as bitmaps, which allows for efficient Boolean operations to extract records with matched Row IDs.

select * from table where Customers = 'Miller' and Material = 'Refrigerator'



#	Customers
1	Chevrier
2	Di Dio
3	Dubois
4	Miller
5	Newman

#	Material
1	MP3 Player
2	Radio
3	Refrigerator
4	Stove
5	Laptop

Dictionary lookup of the strings
⇨Strings are only compared once!
Only those columns are read which are part of the query condition
Integer comparison operations
Customer | ~~3~~ | ~~2~~ | 4 | ~~5~~ | ~~3~~ | 4 | ~~1~~ ....
0 0 1 0 0 1 0
Material | ~~2~~ | ~~5~~ | ~~4~~ | ~~1~~ | ~~2~~ | 3 | ~~4~~ ....
0 0 0 0 0 1 0
Combine
bit-wise AND
0 0 0 0 0 1 0
Resultset | ~~1~~ | ~~2~~ | ~~3~~ | ~~4~~ | ~~5~~ | 6 | ~~7~~ ....

Source: "In-memory Computing with SAP HANA on Lenovo X6 Systems", Lenovo Press

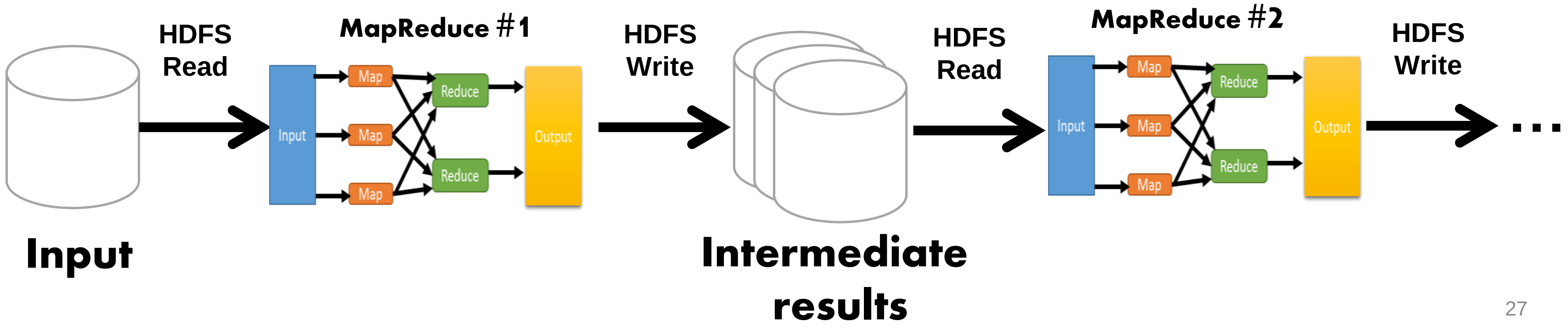
"If everything seems under control,
you're not going fast enough."

Mario Andretti

Going beyond Hadoop & MapReduce

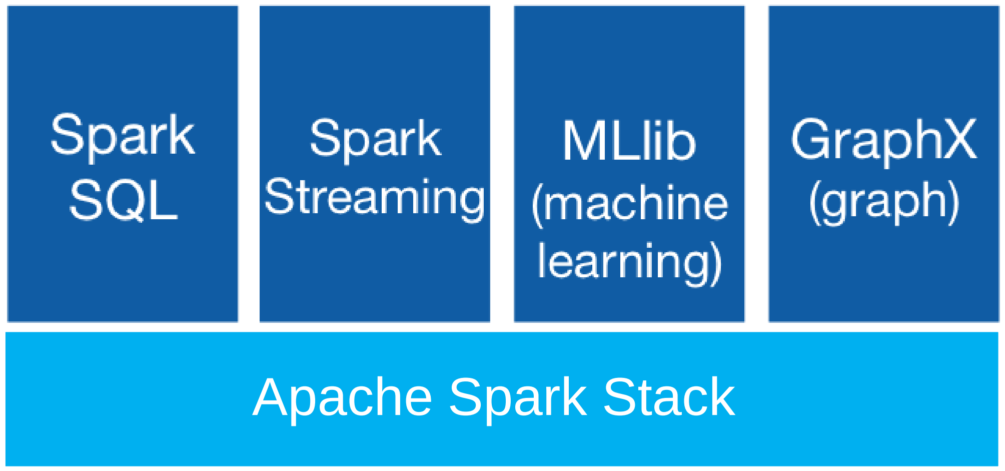
- Hadoop and MapReduce have been very successful in recent decade due to its easiness of use, linear scalability, and high availability. However, data analytics processes, such as feature engineering, often involves iterative and interactive data pipelining/sharing.

- Data reuse in Hadoop MapReduce is slow simply because it requires a series of replications, serializations, and disk I/O—chains of MapReduce tasks. The lack of data abstraction and data sharing mechanisms make Hadoop and MapReduce unable to leverage distributed memory of multiple computing nodes in emerging data applications that reuse intermediate analysis results.



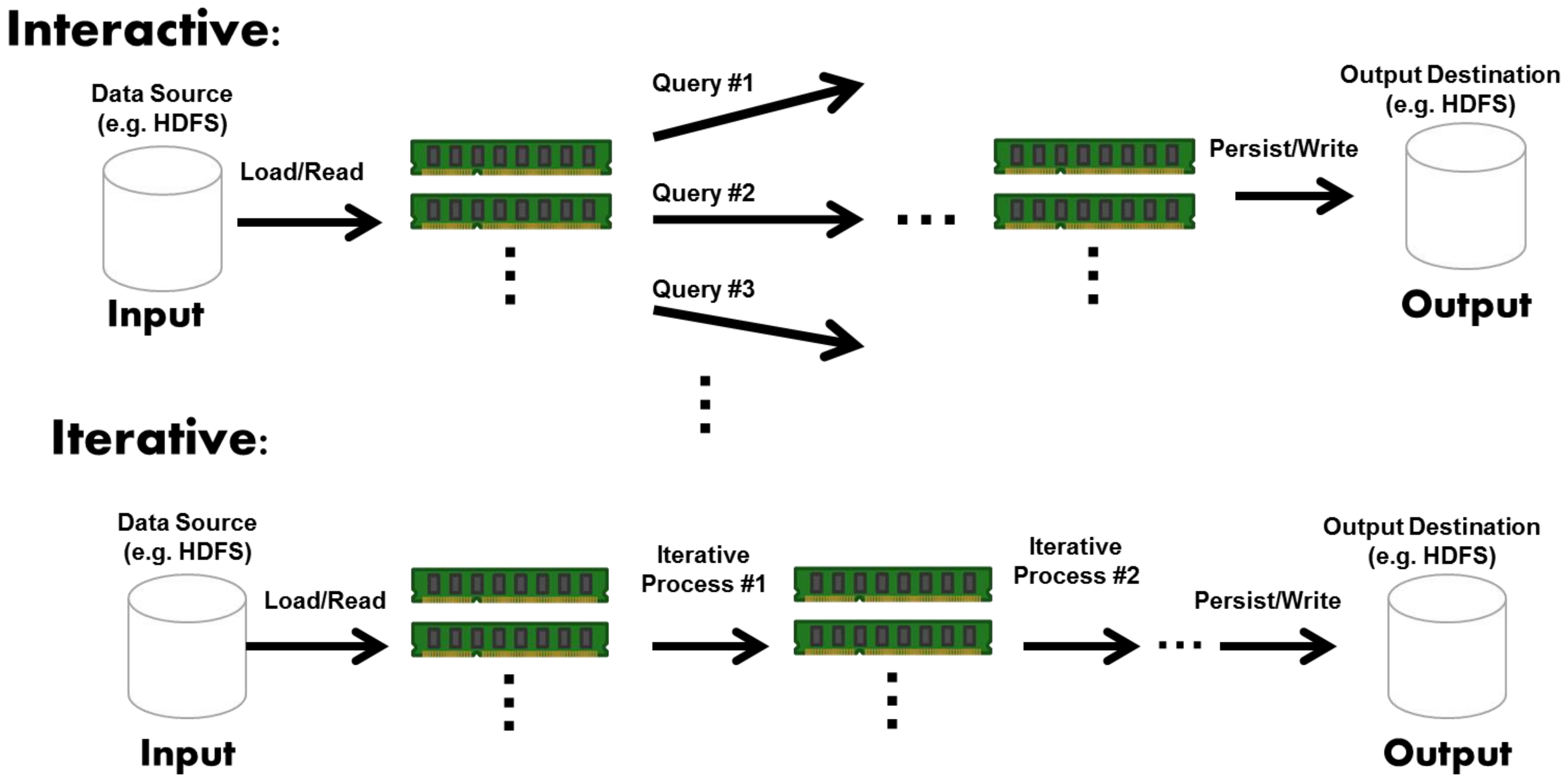
Introduction to Apache Spark

- Apache Spark, an open-source cluster computing framework, is developed in 2009 at UC Berkeley AMPLab to solve the problems. Spark provides an unified engine with a stack of libraries that allow for complex analytics, including batch, streaming, interactive, and graphical computing. It is not proposed to replace but considered a generalization of Hadoop & MapReduce. Now, Spark is one of the largest OSS communities in big data analytics, and its applications range from business, finance, healthcare, and other scientific computing.



Need for Speed

- Growing main memory capacity has fueled the development of in-memory big data analytics. Unlike Hadoop MapReduce, Spark supports interactive and iterative data processing by eliminating disk I/O bottleneck.



Resilient Distributed Datasets

- The core of Spark is the *Resilient Distributed Dataset* (RDD). Similar to H2OFrame, it is an immutable distributed collection of data, which can be partitioned across a cluster of machines. It facilitates two types of data operations: *transformation* and *action*.

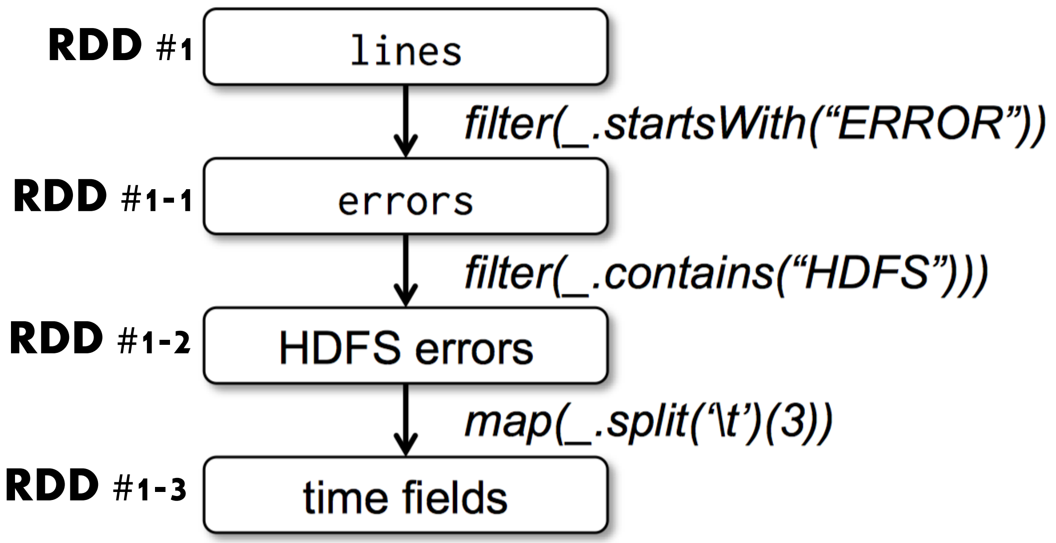
- A *transformation* is an dataset operation, such as filter() and map() on an RDD that create another RDD. An *action*, on the other hand, is a computation on RDDs, such as count() and collect(). An *action* actually triggers a creation/computation of RDDs, returns a value back to the Master node, or writes RDDs to a persistent storage system.

- Like dataset operations on *dplyr tibble*, *transformations* are *lazily evaluated*, in that they do not run until an *action* performs it. Spark Master nodes remembers all the transformations applied to an RDD, a *lineage* of RDDs.

Resilient Distributed Datasets(cont.)

- Spark Master nodes keeps lineages of RDDs. If a partition/RDD is lost (say a slave machine goes down), the partition can easily be reconstructed on the other machines in the cluster—the “resilient” and "fault-tolerant" features of the Spark RDD.

- Also, all the intermediate RDDs are "cacheable", which means we can always save/write results of transformations and actions to external data storage systems.



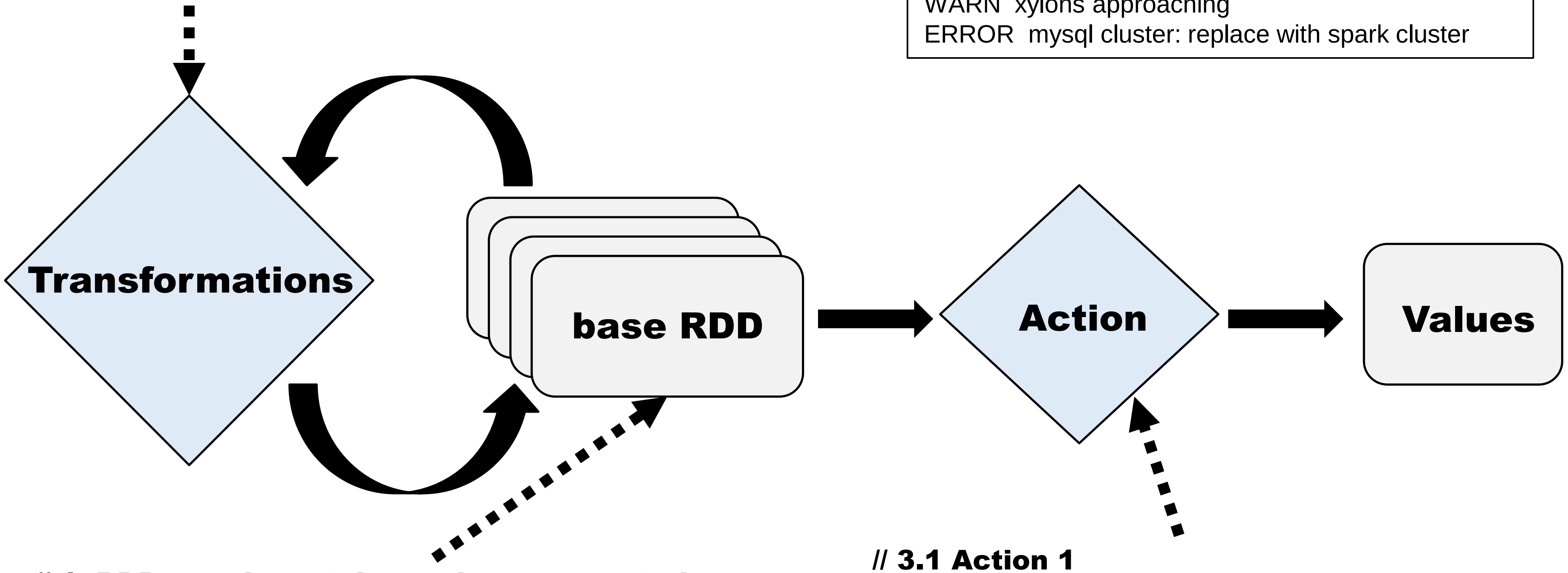
Lineage of Resilient Distributed Datasets

Source: https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf

RDD Operations—Log Mining Example in Scala

// 2. transforming RDDs
```
val errors = lines.filter(_.startsWith("ERROR"))!
val messages = errors.map(_.split("\t")).map(r => r(1))
messages.cache()
```

Log.txt
```
ERROR  php: dying for unknown reasons
WARN  dave, are you angry at me?
ERROR  did mysql just barf?
WARN  xylons approaching
ERROR  mysql cluster: replace with spark cluster
```



// 1. RDDs are immutable and lazy evaluated
```
val lines = sc.textFile("hdfs://tmp/Log.txt")
```

// 3.1 Action 1
```
messages.filter(_.contains("mysql")).count()
```
// 3.2 Action 2
```
messages.filter(_.contains("php")).count()
```

"Log.txt" source: https://gist.github.com/ceteri/8ae5b9509a08c08a1132

Introduction to Sparklyr

- *Sparklyr*, developed by RStudio, is an R interface for Apache Spark. It provides *dplyr* and *DBI* backends with wrapper functions to Spark distributed Machine Learning library(MLlib) that allows us to use familiar R syntax to manipulate and analyze massive datasets (or RDDs) on Spark.

- We can just download & install *sparklyr* from CRAN to get started. Notice that many functions in *sparklyr* is still under experimental development. Remember to check it out on sparklyr page frequently for the latest development, and install the up-to-date version(s) from Github for new functions. You may also have found that, after installation of *sparklyr*, the latest release of RStudio Desktop/Server IDE includes a friendly "Spark" integrated tab, which provides interface to build connections to local or remote Spark cluster.

```r
# install from CRAN
install.packages("sparklyr")
# install from GitHub
devtools::install_github("rstudio/sparklyr")
```

Sparklyr in Action

- *Sparklyr* provide a set of different versions of Spark & Hadoop bundles for local development and debugging purposes. A good practice of working with massive data is to always take a small sample of your big dataset and make sure your code can run on local Spark before actually load large amount of data into Spark cluster.

```r
library(sparklyr); library(DBI); library(dplyr); library(ggplot2)
spark_available_versions() # Check available Spark version
# Install a local Spark for development/testing purpose
spark_install(version = "2.0.2", hadoop_version = "2.7")
spark_installed_versions() # Check what version(s) has been installed
# Remove the installed spark if you'd like:
# spark_uninstall(version = "2.0.2",  hadoop_version = "2.7")
# Spark connection configuration
conf = spark_config(); conf$spark.ui.port = 4040 # default port
conf$spark.executor.memory = "4G" # memory per working node
conf$spark.yarn.executor.memoryOverhead = "1024"
# Create a connection to local Spark
# You may check out http://localhost:4040/ for Spark UI
sc = spark_connect(master = "local", version = "2.0.2",
   app_name = "local_app", config = conf)
spark_version(sc) # Check Spark running version
```

Sparklyr in Action(cont.)

- Just like the way we create *dplyr tibbles*, tibbles to Spark DataFrames (SDFs) can be generated by importing files from various data sources (e.g. HDFS and local FS) or by assigning existing RDDs in Spark to local tibble pointers.

```
# Sys.setenv("HADOOP_CMD" = "/home/hadoop/hadoop/bin/hadoop");
# Sys.setenv("HADOOP_STREAMING"=
#   "/home/hadoop/hadoop/share/hadoop/tools/lib/hadoop-streaming-
#   2.7.2.jar")
# library(rhdfs); hdfs.init(); hdfs.ls("/")
# mySDF = spark_read_csv(sc, name = "myfile",
#   path = hdfs:///home/myfolder/my_file_path.csv", header = T)
# Move R dataframe to Spark. Movies dataset as the example.
movies_sdf = copy_to(sc, df = ggplot2movies::movies, name = "movies",
overwrite = T)
head(movies_sdf) # first 10 observations
# List all available Spark data frames in Spark memory
dbListTables(sc)
# Check Spark log file for monitoring purposes
spark_log(sc, n = 20)
```

Sparklyr in Action(cont.)

```
# "Explain" complex query plan before you actually hit "run"
movies_sdf %>% select(title, rating) %>% explain()
# Or, if we just need SQL translations
movies_sdf %>% select(title, rating) %>% sql_render()
# The good, the bad, and the ugly(very long movie title?)
movies_sdf %>% select(title, length, rating ) %>%
  filter(rating > 9 | rating < 2 | nchar(title) > 80) %>%
  arrange(desc(nchar(title)))
# Give me "Star Trek"!
movies_sdf %>% select(title, rating, budget) %>%
  filter(title %like% "%Star Trek%")
# Movie title with r1-r5, the select helpers are also supported.
# type ?select_helpers for more information
movies_sdf %>% select(title, num_range(prefix="r", range = 1:5))
# title with any variable names that contains numbers
movies_sdf %>% select(title, matches("[[:digit:]]"))
# How many long/short movies?
movies_sdf_longShort =
  movies_sdf %>% select(title, rating, length) %>%
  mutate(longShort = ifelse(length > 130, "long", "short")) %>%
  group_by(longShort) %>% summarise(count = n())
```

Sparklyr in Action(cont.)

- Notice that previous SDF operations did not actually create named SDFs/RDDs in Spark, as most of them are *transformations* and thus lazily evaluated by Spark. Surely, they are also *cacheable* when using *sparklyr*.

```
# Create the temporary table "movies_sdf_longShort"
compute(movies_sdf_longShort, "movies_longShort_ct")
dbListTables(sc) # It's now in Spark as a temporary RDD
# You can actually create (cache) a Spark DataFrame in memory.
# It may improve performance. Remember to check your Spark UI/Tab.
tbl_cache(sc, "movies_longShort_ct")

# Create an SDF pointer to existing RDD
movies_longShort_ct_sdf = tbl(sc, from = "movies_longShort_ct")
# For small SDFs, we may save the result as local R data frames
movies_longShort_ct_df = collect(movies_longShort_ct_sdf)

# We can surely save or remove RDDs
# spark_write_csv(movies_longShort_ct_sdf, path =
#   "hdfs:///home/myfolder/movies_longShort_ct.csv")
rm(movies_longShort_ct_sdf)
tbl_uncache(sc, "movies_longShort_ct")
```

A bit about Spark SQL in Sparklyr

- Sparklyr supports Spark SQL via functions in DBI. However, SQL via DBI is not intended to be used to create new SDFs/RDDs. It should only be used in query purposes.

```
# Save query result as local R data frame
movies_10obs = dbGetQuery(sc, "select * from movies limit 10")

# Or a query result in Spark
rs = dbSendQuery(sc, "select title, rating from movies limit 100")
dbFetch(rs, n = 5)
# Remove objects
dbClearResult(rs); rm(rs)

# Use dbWriteTable() to create a new RDD, diamonds with "log of Price"
dbWriteTable(sc, "diamonds", mutate(diamonds,
  logPrice = log(price) ))

# Create a tibble pointing to the cached RDD
diamonds_sdf = tbl(sc, "diamonds")
tbl_cache(sc, "diamonds"); diamonds_sdf
```

Hands on Spark MLlib

- Although some up-to-date functions are not available, *sparklyr* provides wrapper functions that allows us to access the power of Spark MLlib.

```
# 70% as the training set
diamonds_train_sdf =  sdf_sample(diamonds_sdf, 0.7, seed = 1 )
nrow(diamonds_train_sdf)
# 30% as the testing set
diamonds_test_sdf = setdiff(diamonds_sdf, diamonds_train_sdf)
nrow(diamonds_test_sdf)
# Or we can just use sdf_partition() to make your life easier
diamonds_train_test = sdf_partition(diamonds_sdf, training = 0.7,
   test = 0.3, seed =1)
# Actually create RDDs to faciliate model fitting
compute(diamonds_train_sdf, "diamonds_train", temporary = F)
tbl_cache(sc, "diamonds_train", force = T)
compute(diamonds_test_sdf, "diamonds_test", temporary = F)
tbl_cache(sc, "diamonds_test", force = T)
# General linear model
diamonds_lm = ml_linear_regression( logPrice ~ carat + cut + clarity +
  x + y + z, data = diamonds_train_sdf  )
summary(diamonds_lm)
```

Hands on Spark MLlib(cont.)

```
# We can also save model to local filesystem.
# It is actually a folder instead of file
ml_save(diamonds_lm, file = "./diamonds_spark_lm.RData")
rm(diamonds_lm) # delete the model
# Then load it back later.
# Unfortunately, until sparklyr 0.55, only model coefficients
# can be serialized.
# The reloaded model can only be used for prediction
diamonds_lm = ml_load(sc, "./diamonds_spark_lm.RData")

# Predicted response/outcome column is called "prediction"
diamonds_lm_testpred = sdf_predict(diamonds_lm,
  newdata = diamonds_test_sdf)
diamonds_lm_testpred %>% select(actualLogPrice = logPrice,
  predictedLogPrice = prediction) %>% head()

# Compute RMSE
diamonds_lm_testpred %>%
  transmute(MSE  = mean((logPrice - prediction)^2) ) %>%
  head(1) %>%
  transmute(RMSE = round(sqrt(MSE), 4) ) %>% collect()
```