MIS 573:

Practical Big Data Analytics

In-Database Analytics using R, MADlib, and MPP Databases

Yihuang K. Kang

"...if you are looking for a career where your services will be in high demand, you should find something where you provide a scarce, complementary service to something that is getting ubiquitous and cheap...

So what's getting ubiquitous and cheap? Data. And what is complementary to data? Analysis."

Hal R. Varian

Introduction to In-Database Analytics

- Computer scientists have been dealing massive data for years even before the advent of the buzzword "Big Data". A few decades ago when "unstructured data" did not get that much attention, traditional OLTP RDBMS along with OLAP applications were often used in processing of massive datasets.

- To analyze massive structured data, many traditional DBMS and data warehousing system vendors (e.g. Oracle, IBM, Teradata, Microsoft, and even Amazon!) have been expanding the functionalities of their data processing tools & techniques, such as extensions of SQL, procedural languages, UDFs, and Multidimensional Expressions—the dawn of *In-Database Analytics*.

- These extensions of data mining/analytics applications are often implemented as part of these vendors' data warehousing systems. It allows for loading existing or user-defined analytics library into database process spaces so as to have access to database systems' memory management and parallel data processing capabilities.

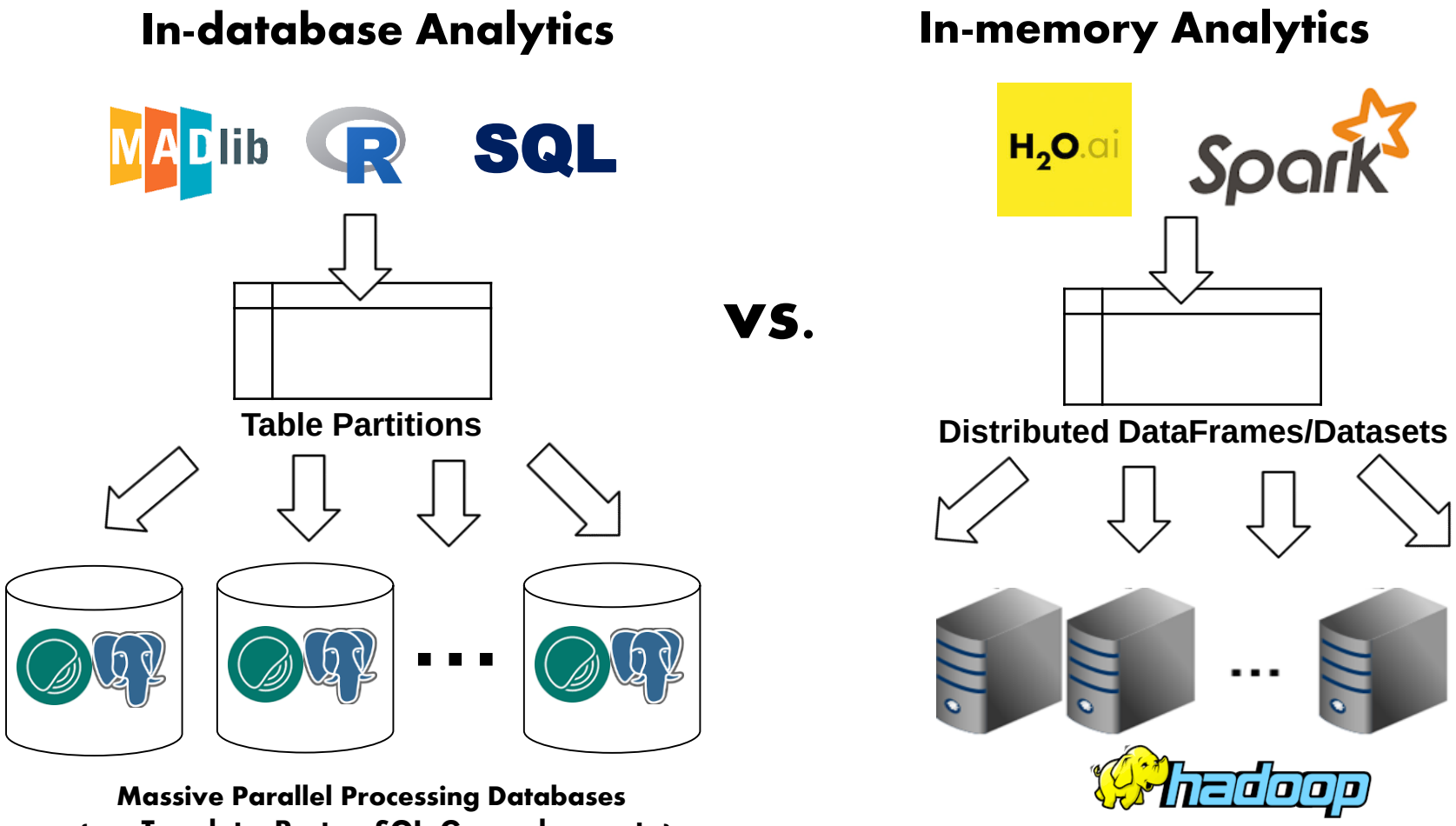
Introduction to In-Database Analytics(cont.)

- Embedding analytics capabilities inside the database management systems reduces the overhead of moving data back and forth. Most statistical packages and data mining applications require datasets to be loaded & fit in RAM, which limits the possibility of analyzing big data. Processing data right in the database systems, however, does not have such restrictions. It means that you can directly fit models from a huge dataset in one or multiple database server(s) without moving them into the memory of your local machines.

- Also, under such background, a concept of *Massive Parallel Processing* (MPP) database systems was therefore proposed, which aims at providing high query performance and platform scalability by partitioning data across multiple database servers/nodes without sharing processing resources—each server processes its part of data locally using its own memory and processor(s).

- Such MPP database systems with in-database analytics facility is now considered the other side of the big data analytics—another ecosystem that allows users to employ existing database systems to deal with big data, especially structured and semi-structured data, without bothering with new distributed computing systems (e.g. Apache Hadoop framework).

Two ecosystems of scalable data analytics



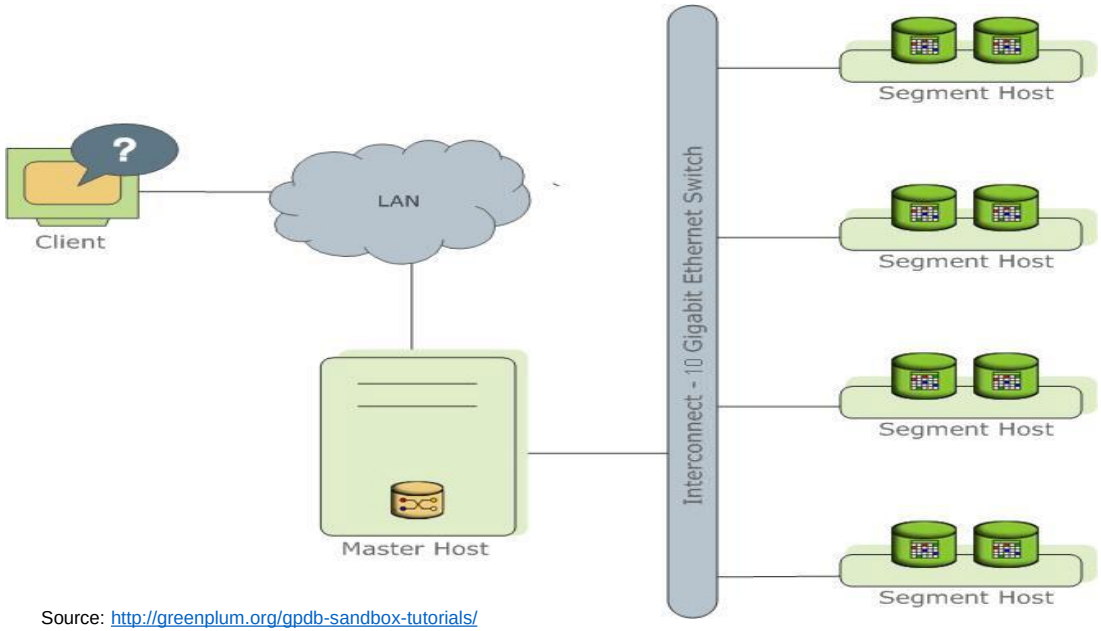
PostgreSQL for Big Data

- PostgreSQL is a powerful open source object-relational database system with most up-to-date & sophisticated features/implementations of advanced database systems. It has been widely used in many mission-critical applications in the world. Before introducing PostgreSQL and its derivatives (e.g. Amazon Redshift and HP Vertica) into your big data applications, however, some limitations you should be aware first.

Limitation	Value
Maximum Database Size	Unlimited
Maximum Table Size	32 TB
Maximum Row Size	1.6 TB
Maximum Field Size	1 GB
Maximum Rows per Table	Unlimited (2^48 for Greenplum 4)
Maximum Columns per Table	**1600** (use data type "Array" could solve this problem)
Maximum Indexes per Table	Unlimited
JSON data type support	Since version 9+

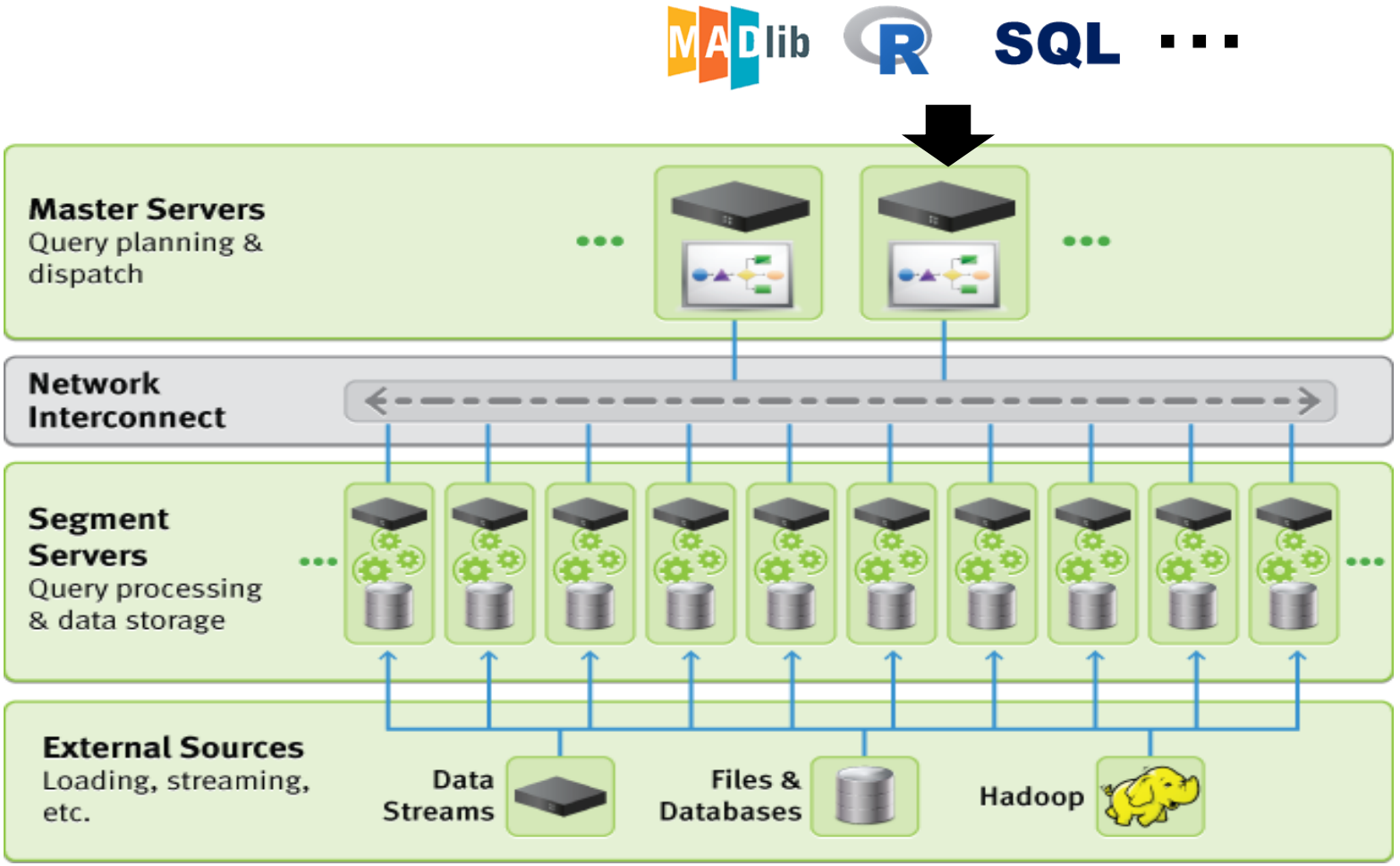
Introduction to Greenplum

- Greenplum Database, also a derivative of PostgreSQL, is a highly-scalable, shared-nothing MPP database system able to handle storage and processing of massive data by distributing the load across several nodes. A database in Greenplum is actually an array of individual PostgreSQL databases working together to present a single database image. The *master* is the entry point to the Greenplum. It is the database instance where clients connect and submit SQL statements. The master coordinates the work with the other database instances in the system, the *segments*, which handle data processing and storage.



Source: http://greenplum.org/gpdb-sandbox-tutorials/

Introduction to Greenplum (cont.)



Source: http://greenplum.org/gpdb-sandbox-tutorials/

A bit about table distribution strategy

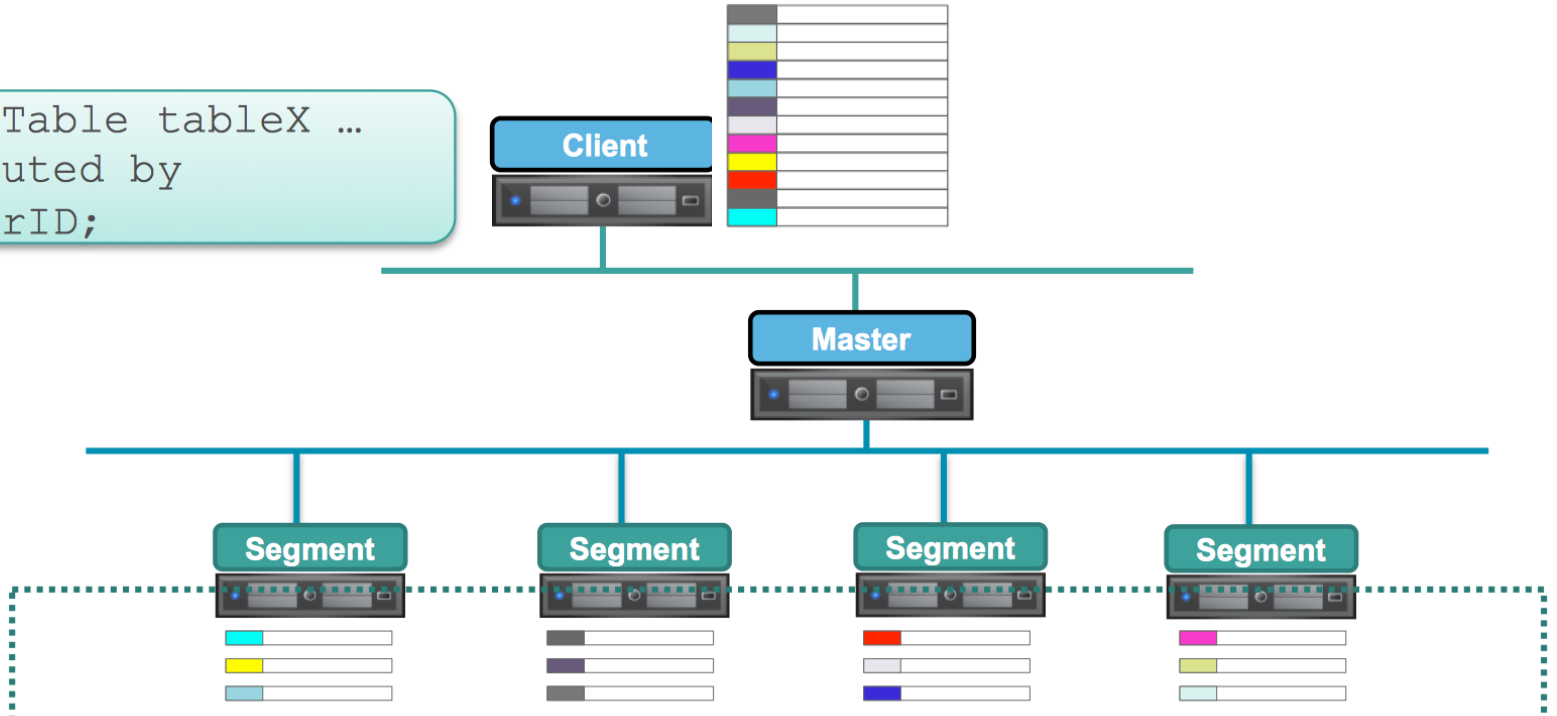
- Table distribution techniques are crucial to the workload balancing among all computing nodes in most distributed database systems. In Greenplum, all database tables are distributed across all segments, and the performance of data processing & analytics depends heavily on the table distribution strategy. Greenplum supports two common distributions:

  - *Hash Distribution* (by default), where one or more columns is used as distribution key and data rows/tuple are hashed into "buckets" for each segment. Often, unique keys (e.g. primary key attribute(s)) could assure even data distribution.

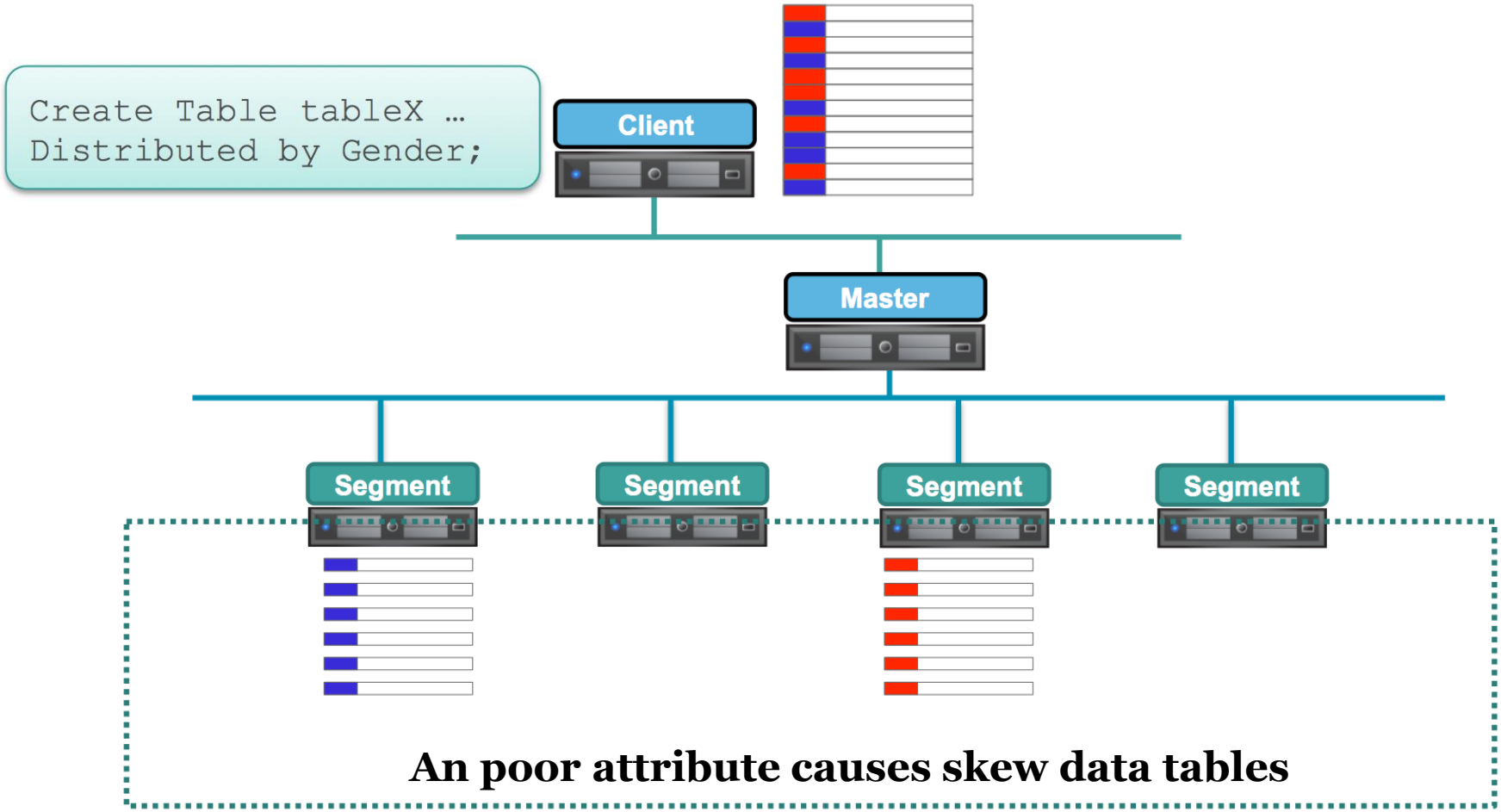
  - *Random Distribution*, where round-robin distribution is used to evenly assign data rows to segments. It may be less efficient than hash distribution.

A bit about table distribution strategy(cont.)

```
Create Table tableX …
Distributed by
CustomerID;
```



A bit about table distribution strategy(cont.)



MAD Skills

- In the recent years, due to the possibility of massive data acquisition along with cheaper data processing and storage, researchers have been proposing new data management and analysis practices different from those of traditional *Business Intelligence* (BI). *MAD Skills* is the most popular new data analytics practice, where "MAD" stand for:

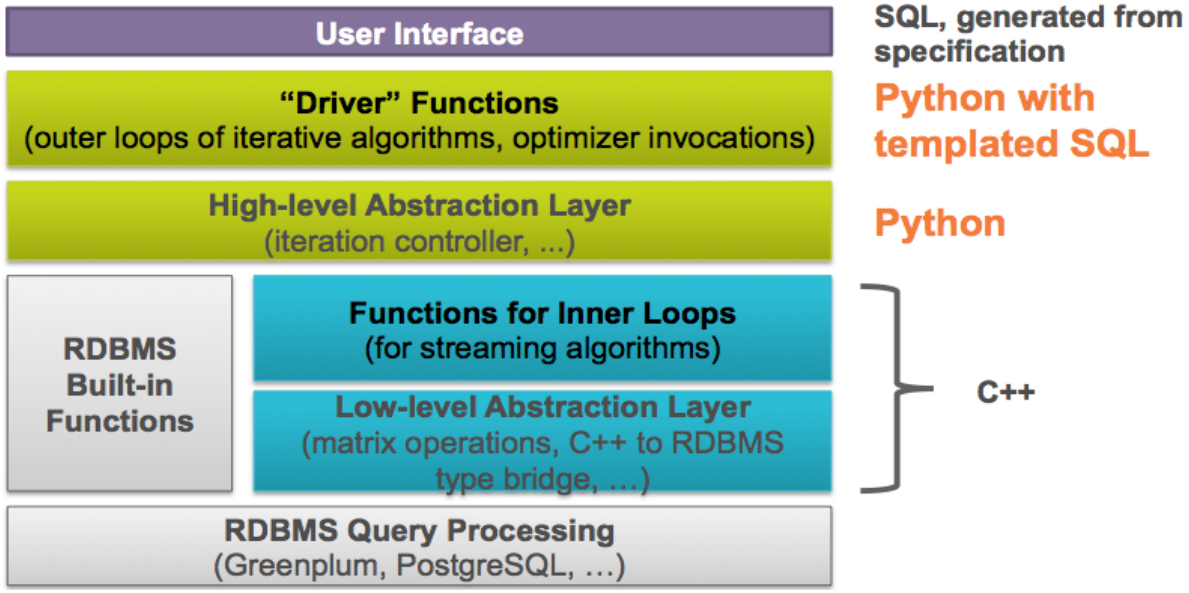
Magnetic: Traditional data warehouse discourages the incorporation of data until they are carefully cleansed and integrated. Today, a data warehouse can keep pace only by being “magnetic"—attracting all the data sources that crop up within an organization regardless of data quality.

Agile: A modern data warehouse must allow analysts to easily ingest, digest, produce, and adapt data at a rapid pace. This requires a database system whose physical/logical contents and structures can be in continuous rapid evolution.

Deep: We're going "deep", far away beyond the "rollups" and "drilldowns" of traditional BI. Data warehousing should serve both as a data repository and as a sophisticated algorithmic runtime engine.

Apache MADlib

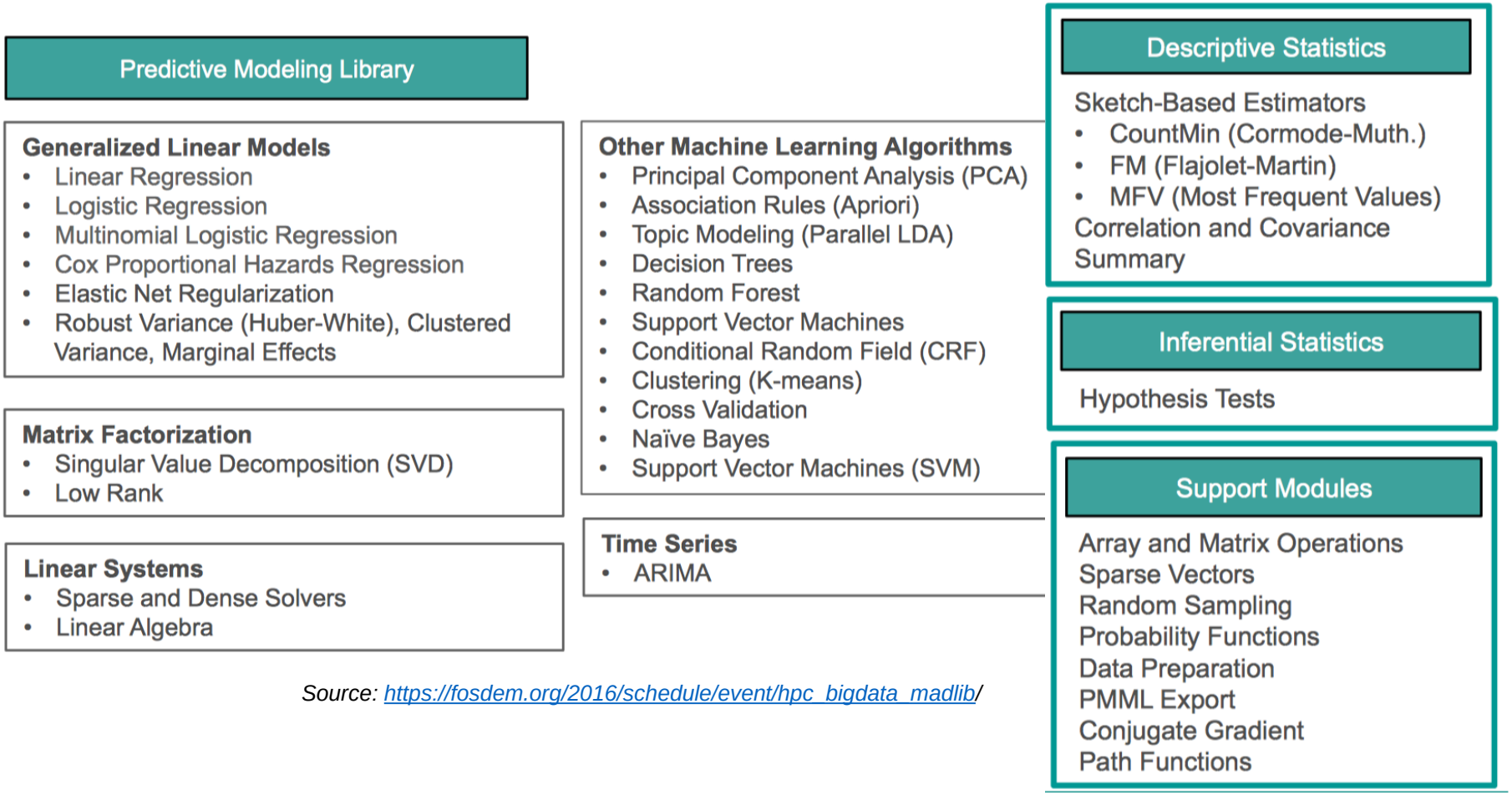
- The *MADlib*, an incubating project in Apache Software Foundation (originally developed by UC Berkeley and Greenplum), is the product followed by the publication of the MAD Skills related research. It provides SQL-based, open source library of distributed in-database machine learning algorithms that runs at scale within a database engine (currently support PostgreSQL and Greenplum Database).



Source: https://cwiki.apache.org/confluence/display/MADLIB/Architecture

Apache MADlib(cont.)

- Remember to check the latest *MADlib* library for more information!



Source: https://fosdem.org/2016/schedule/event/hpc_bigdata_madlib/

Apache MADlib and R

- In this unit, we will be using package *RPostgreSQL* and *PivotalR* to get access to the power of *MADlib*. *RPostgreSQL* is an R interface that allows us to manipulate tables/views and execute SQL on PostgreSQL or Greenplum database servers. *PivotalR* is, however, a set of wrapper functions of *MADlib* for R users who are not familiar with SQL but would like to use *MADlib*. Below is an example that shows how to use *PivotalR* functions and SQL to fit a general linear model.

PivotalR

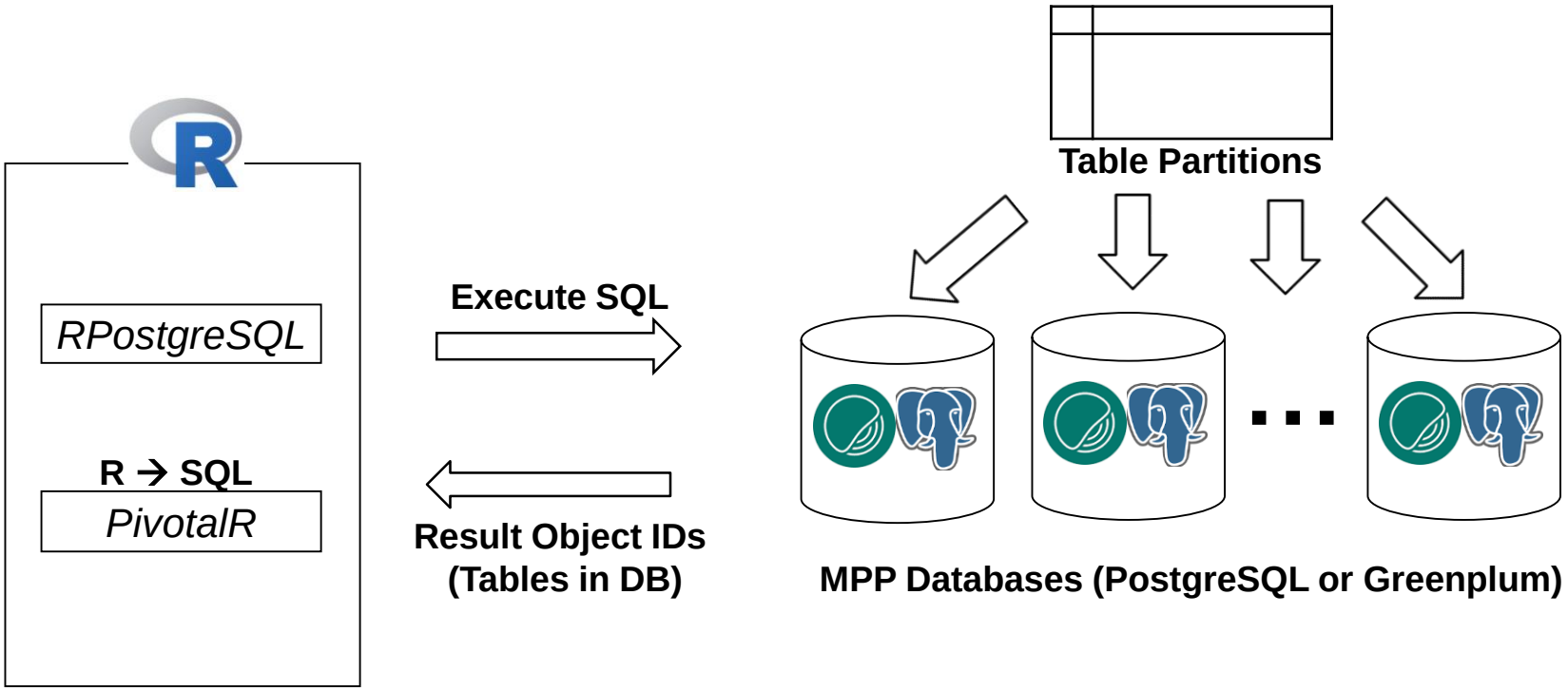
```
diamonds_dbdf = db.data.frame("diamonds")
diamonds_lm =
  madlib.lm( price ~ carat + x,
             data = diamonds_dbdf )
```

SQL

```
SELECT madlib.linregr_train('diamonds',
   'diamonds_lm', 'price',
   'ARRAY[1, carat, x]');
```

Apache MADlib and R(cont.)

- The *PivotalR* is actually a "translator" that helps convert your R code into SQL code, call interface functions in *RPostgreSQL,* and then send the SQL code to the backend PostgreSQL or Greenplum database cluster.



Hands-on MADlib

- Let's create R SQL pass-through queries that call MADlib functions.

```
library(PivotalR); library(DBI); library(RPostgreSQL); library(ggplot2)
diamonds = as.data.frame(diamonds) # Dataset ggplot2::diamonds
## Using RPostgreSQL/DBI
# Create a PostgreSQL DB connection
pgConn = dbConnect("PostgreSQL", dbname = "mydb", user="myname",
  password="mypass")
dbListTables(pgConn) # List all tables in the database
# Create table "diamonds". Delete if it exists.
ifelse(dbExistsTable(pgConn, "diamonds"), dbRemoveTable(pgConn,
   "diamonds"), dbWriteTable(pgConn, "diamonds", diamonds))
dbReadTable(pgConn, "diamonds") # Read existing table
dbListFields(pgConn,"diamonds") # Get Column names
# Run SQL statements by creating a resultset object
rs = dbSendQuery(pgConn, statement = "SELECT count(*) FROM diamonds");
fetch(rs);
dbColumnInfo(rs) # column info.
dbClearResult(rs); rm(rs) # Clear resultset
# Or we can call
dbGetQuery(pgConn, "SELECT count(*) FROM diamonds")
```

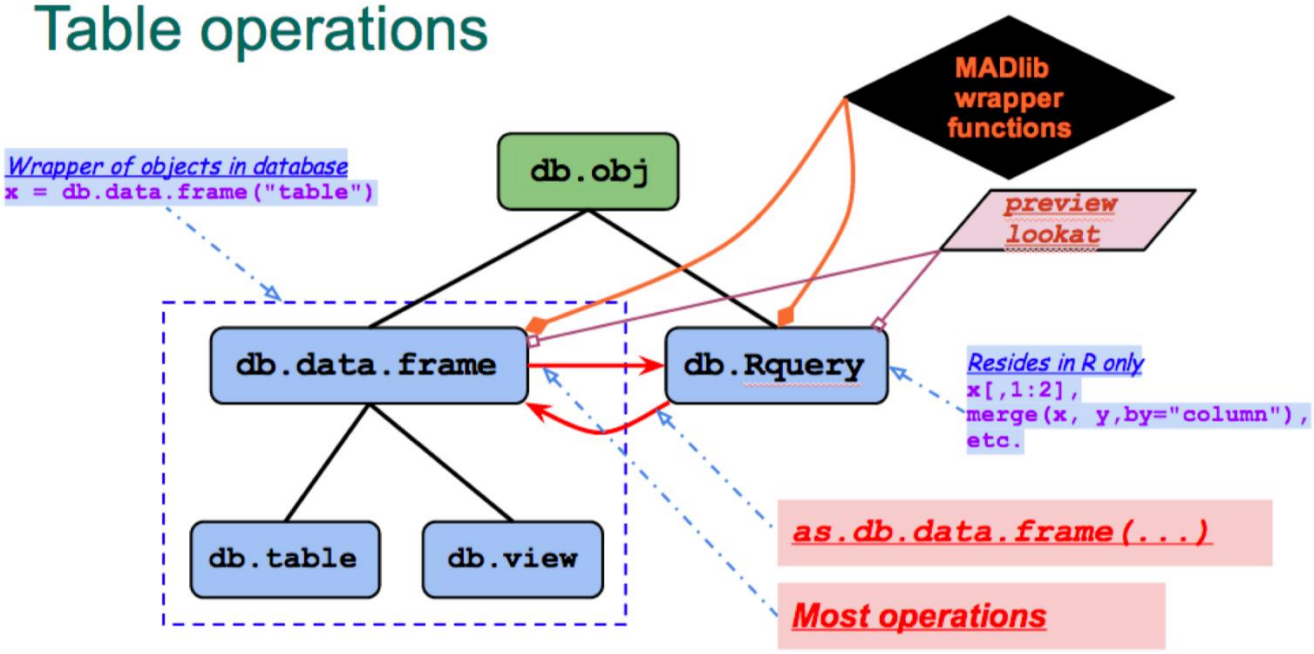
Hands-on MADlib(cont.)

```
# Table summary statistics
dbSendQuery(pgConn, "SELECT * FROM madlib.summary('diamonds',
   'diamonds_summmary')")
dia_sum = dbReadTable(pgConn, "diamonds_summmary"); dia_sum
dbRemoveTable(pgConn,"diamonds_summmary") # remove table
# Correlation matrix
dbSendQuery(pgConn, "SELECT * from madlib.correlation( 'diamonds',
   'diamonds_pearson', 'price, depth, carat')")
# 2 tables are generated
dbReadTable(pgConn, "diamonds_pearson")
dbReadTable(pgConn, "diamonds_pearson_summary")
 column_position variable      price      carat depth
1               1    price  1.0000000         NA    NA
2               2    carat  0.9215913 1.00000000    NA
3               3    depth -0.0106474 0.02822431     1

# Remove tables if we don't need them
Map(function(tb) dbRemoveTable(pgConn, tb),
    c("diamonds_pearson", "diamonds_pearson_summary") )
# Linear Regression. Equivalent to lm(price ~ carat + depth, diamonds)
dbSendQuery(pgConn, "select madlib.linregr_train('diamonds',
   'diamonds_lm','price', 'ARRAY[1, carat, depth]')")
dbReadTable(pgConn, "diamonds_lm")
```

Using *PivotalR*

- You may have felt that using SQL code is a bit awkward. For those of you who are familiar with R, using package *PivotalR* would make your life easier. In *PivotalR*, a table object or its derivative is a subclass of *db.obj*. And all table operations would create a temporary format of table—*transformation* (*db.Rquery*) until it is actually loaded & saved into the database as a *DataFrame* (*db.data.frame*).



Source: https://cran.r-project.org/web/packages/PivotalR/vignettes/pivotalr.pdf

Using *PivotalR*(cont.)

- Let's give it a try. Here we use the *diamonds* again as the example.

```
# Create a PivotalR DB connection
dbc = db.connect(dbname = "mydb", user="myname",
     password="mypass", default.schema = "public, madlib", verbose=T)
db.list() # List all current PivotalR DB connections
# List all objects/tables of a connection in default schema "public"
db.objects("public.", conn.id = dbc)
# Link to the table "diamonds" in the database
db_diamonds = db.data.frame("diamonds", dbc, verbose = T)
object.size(db_diamonds)
# db_diamonds is just an object pointing to "diamonds" in the database!
dim(db_diamonds) # dimensions
summary(db_diamonds) # summary statistics
names(db_diamonds) # column names
# Split-Apply-Combine. Unfortunately, only works on SQL aggregation functions
mean_price_by_cut = by(db_diamonds$price, db_diamonds$cut, mean)
# mean_price_by_cut is just an db.Rquery object,
# no data is loaded in memory
# check the actual content of "mean_price_by_cut"
content(mean_price_by_cut)
# Actually execute SQL query and get the result
lk(mean_price_by_cut)
```

Using *PivotalR* (cont.)

- **Table Operations**

```
# Let's say we only need a few variables
db_diamonds_few_vars = db_diamonds[,c("carat", "cut","clarity",
   "depth", "price")]
# db_diamonds_few_vars is a db.Rquery before we call as.db.data.frame()
db_diamonds_few_vars = as.db.data.frame(db_diamonds_few_vars,
   "db_diamonds_few_vars", verbose = T)
lk(db_diamonds_few_vars, 10) # Get the first 10 observations
# Let's try creating a new table
db_diamonds_price_clarity = db_diamonds[, c("price", "carat")]
# A new column "log of price"
db_diamonds_price_clarity$logprice =
   log(db_diamonds_price_clarity$price)
content(db_diamonds_price_clarity) # The actual SQL code
# Save as a real table in the database
db_diamonds_price_clarity=as.db.data.frame(db_diamonds_price_clarity,
   "db_diamonds_price_clarity")
lk(db_diamonds_price_clarity, nrows = 10) # Take a look
# Remove the table if you'd like
delete(db_diamonds_xy)
# Clean all temporary MADlib tables
clean.madlib.temp()
```

Try It!

- Load *Hitters* and *Default* datasets in package *ISLR*. Save them as tables in your PostgreSQL database.

- Use functions in *PivotalR* to count the numbers of *defaults* by *student* from your table *Default*.

- Use functions in *PivotalR* to create a new table that contains log of *Salary*, *Years* and *Hits* from your table *Default*.

"...every company has big data in its future and every company will eventually be in the data business..."

Thomas H. Davenport

Array & Matrix Operations

- As all computations on objects of *db.obj* are within the databases, we can easily manipulate huge datasets, even those too large to fit in memory (e.g. large array/matrix operations).

```
pgConn = dbConnect("PostgreSQL", dbname = "mydb", user="myname",
   password="mypas")
# 1000 normal random numbers as example
# Add serial ID/names numbers for easy operations on db.data.frame
db_nrand_1000 = as.db.data.frame(data.frame(matrix(rnorm(1000,0,1),
   ncol=5)),"db_nrand_1000",add.row.names = T, key = "row_names")
lk(db_nrand_1000) # Take a look the result
# In MADlib, matrices are often implemented/represented as
# a collection of 1-D arrays.
db_nrand_1000_ary = as.db.data.frame( cbind(db_nrand_1000[,1],
   db.array(db_nrand_1000[,2:6]) ), table.name = "db_nrand_1000_ary")
# Take a look at the first 5 row of the data in array form
lk(db_nrand_1000_ary, array = F, nrows = 5)
# Use db.q() to send SQL query to the database.
# Compute simple statistics
stat_ary = db.q("select row_names, madlib.array_sum(agg_opr),
   madlib.array_mean(agg_opr), madlib.array_stddev(agg_opr) from
   db_nrand_1000_ary", nrows = 5)
```

Array & Matrix Operations(cont.)

```
# Get Euclidean norm. Then save as a new table in database
db.q("select row_names, sqrt(madlib.array_dot(agg_opr, agg_opr)) as
   norm into db_nrand_1000_ary_norm from db_nrand_1000_ary")
db_nrand_1000_ary_norm = db.data.frame("db_nrand_1000_ary_norm")
lk(db_nrand_1000_ary_norm, nrows = 20)
# Matrix Transposition: "200 x 5" to "5 x 200"
dim(db_nrand_1000_ary) # 200 x 2, 2nd column is an array data type
# Specify key and array column, output to table "db_nrand_1000_ary_t"
db.q("select madlib.matrix_trans('db_nrand_1000_ary', 'row=row_names,
   val=agg_opr', 'db_nrand_1000_ary_t')")
db_nrand_1000_ary_t = db.data.frame("db_nrand_1000_ary_t")
dim(db_nrand_1000_ary_t) # it's now 5 x 2
lk(db_nrand_1000_ary_t,nrows = 1,array = F)
# Singular Value Decomposition. Syntax:
# svd( src_table,output_prefix, row_id, n_vecs, n_iter);
db.q("select madlib.svd('db_nrand_1000_ary', 'db_nrand_100_svd',
'row_names', 3)")
allTables = dbListTables(pgConn)
svd = allTables[grep("svd", allTables)] # tables with the prefix
Map(function(x) db.q(paste("select * from ", x, " order by row_id")) ,
   svd_tables )
```

Linear Models

- *MADlib* also supports fitting general linear model in the databases.

```
dbc = db.connect(dbname = "mydb", user="myname",
                 password="mypass", default.schema = "public, madlib")
db_diamonds = db.data.frame("diamonds")
# fitting Gaussian linear model
price_db_lm = madlib.lm(log(price) ~ carat + clarity +  x + y + z,
   data = db_diamonds)
price_db_lm
...
Estimate Std. Error t value Pr(>|t|)
(Intercept)   0.717382   0.017683  40.570  < 2e-16 ***
carat        -0.680530   0.009399 -72.405  < 2e-16 ***
clarity.IF    1.022835   0.010025 102.025  < 2e-16 ***
...
price_db_lm$model@.name # model info. is saved in this temporary table
# Using "|" to fit models conditioned on the values of variable(s)
price_db_by_cut_lm = madlib.lm(log(price) ~ carat +  x + y + z | cut,
   data = db_diamonds)
price_db_by_cut_lm
```

k-fold Cross-Validation

- A common application is to evaluate your model performance based on cross-validation error results. *PivotalR/MADlib* provides a generic function for all its supervised learning algorithms.

```
# Getting 10-fold CV RMSEs
set.seed(1)  # training and testing datasets
dbRemoveTable(pgConn, "diamonds")
db_diamonds = as.db.data.frame(diamonds, table.name = "diamonds",
  add.row.names = T) # Add row names as the unique IDs
# 70% as the training set
db_diamonds_train = sample(db_diamonds, 0.7 * nrow(db_diamonds))
# Create testing dataset
db.q(paste("select * into diamonds_test from diamonds
   where row_names not in (select row_names from",
   noquote(paste(db_diamonds_train@.name, collapse = ".")), ")" ))
db_diamonds_test = db.data.frame("diamonds_test")

generic.cv( function(d) madlib.lm(log(price) ~ carat + x + y + z,
   data = d), predict, function(pred, d)
              lookat(sqrt(mean((log(d$price) - pred)^2))),
            data = db_diamonds_train, k = 10)
```

Logistic Regression

- Just like *glm*() we discussed, *madlib.glm*() allows you to create generalized linear models with support for multiple family distributions and link functions. Consider a logit model for *ISLR::Default* data as example.

```
# Logistic regression
library(ISLR); data("Default"); set.seed(1)
train_idx = sample(1:nrow(Default), 0.7 * nrow(Default));
test_idx = setdiff(1:nrow(Default), train_idx)
db_defaults_train = as.db.data.frame(Default[train_idx,],
   "db_defaults_train", add.row.names = T, key = "row_names" )
db_defaults_test = as.db.data.frame(Default[test_idx,],
   "db_defaults_test", add.row.names = T, key = "row_names" )

# Learn logit model from training data
Default_logit = madlib.glm(default ~ student + balance,
   data = db_Defaults_train, family = binomial); Default_logit
Default_logit$odds_ratios # Odd ratios

# Compute predicted probabilities
db_Defaults_test_pred = as.db.data.frame(predict(Default_logit,
   db_Defaults_test, type="prob"), "db_Defaults_test_pred")
lk(db_Defaults_test_pred) # Compute predicted probabilities
```

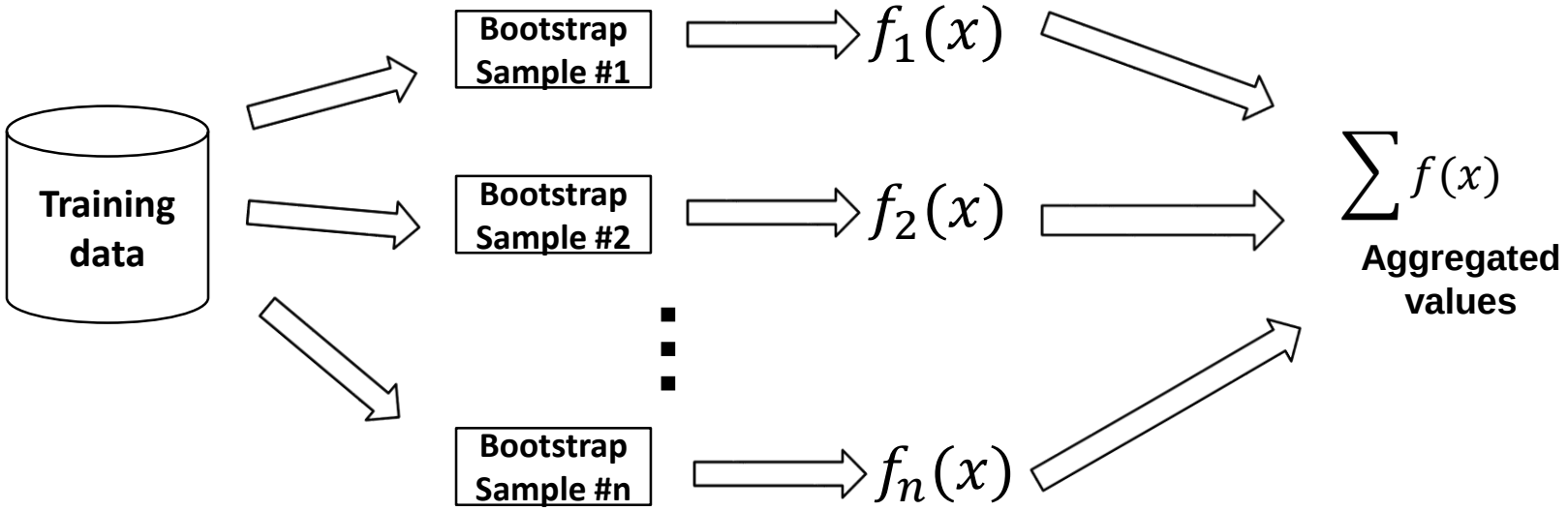
Classification and Regression Tree

- Although not as complete as package *rpart*, *PivotalR*/*MADlib* does have some similar functions in *rpart*. But bad news is, until *MADlib* 1.8, we cannot post-prune a tree and get information about cost-complexity parameters.

```
# Classification tree for "default" data
# Note that we must specify the primary key attribute as "id"
Default_CT = madlib.rpart(default ~ student + balance,
   id = "row_names", data = db_defaults_train,
   parms = list(split='gini'), control = list(cp=0))
Default_CT
# Plot the tree
plot(Default_CT); text(Default_CT)
# Performance evaluation on testing data
db_defaults_test_pred = as.db.data.frame(predict(Default_CT,
   db_defaults_test), "db_defaults_test_pred")
# Confusion Matrix
db.q("select p.estimated_default, a.default as act_default, count(*)
     from db_defaults_test_pred p, db_defaults_test a
     where p.row_names = a.row_names
     group by estimated_default, act_default ")
```

Bootstrap Aggregating

- *Bootstrap Aggregating* (a.k.a *Bagging*) is a technique that improves prediction accuracy of learning algorithms (learners) by averaging resulting predictions so as to reduce prediction variance. It is particularly helpful to *unstable* learners (e.g. decision trees). By "unstable" here, we mean those techniques sensitive to small changes of training dataset(e.g. adding/removing features or observations).



Bootstrap Aggregating(cont.)

- *PivotalR* provides a generic function, *generic.bagging*(), that allows you to create bagging learners and save them in the database. Let's consider bagging linear model as the example.

```
set.seed(1);  # Create tables in PostgreSQL
train_idx = sample(1:nrow(diamonds), 0.7 * nrow(diamonds))
test_idx = setdiff(1:nrow(diamonds), train_idx)
db_diamonds_train = as.db.data.frame(diamonds[train_idx, ],
   "db_diamonds_train", add.row.names=T, key="row_names" )
db_diamonds_test = as.db.data.frame(diamonds[test_idx, ],
   "db_diamonds_test", add.row.names=T, key="row_names" )
# Bagging linear model
bagLM = generic.bagging(function(d){
     madlib.lm(log(price) ~ carat + clarity +  x + y + z, data = d)},
   data = db_diamonds_train, nbags = 10, fraction = 0.5)
# One single linear model
simpleLM = madlib.lm(log(price)~carat + clarity + x + y + z,
 data = db_diamonds_train)
# RMSEs for bagging LM and one LM
bigLMPred = predict(bagLM, db_diamonds_test)
lookat(sqrt(mean((log(db_diamonds_test$price) - bigLMPred)^2)))
simpleLMPred = predict(simpleLM, db_diamonds_test)
lookat(sqrt(mean((log(db_diamonds_test$price) - simpleLMPred)^2)))
```

Random Forest

- *Random Forest* might be your first choice of modern learning algorithms when you deal with classification and regression problems. It is no doubt that you will find it in most machine learning libraries. However, *MADlib* only support part of the features in the R package *randomForest*.

```
# Building a random forest
diamonds_RF = madlib.randomForest(log(price) ~ carat +clarity + x+y+z,
   data = db_diamonds_train, id = "row_names",
   ntree = 20, importance = T)
diamonds_RF$importance # Variable Importance
# Predicted results
diamonds_RF_pred = as.db.data.frame(predict(diamonds_RF,
db_diamonds_test), "diamonds_RF_pred")
# Unfortunately, the predicted result IDs doesn't
# match the ones in the database. Sort by the row_names first
act_RF = lk(db_diamonds_test[,c("row_names", "price")], "all")
act_RF = act_RF[order(act_RF[,1]),]
pred_RF = lk(diamonds_RF_pred, "all")
pred_RF = pred_RF[order(pred_RF[,1]),]
# RMSE
sqrt(mean(( log(act_RF$price) - pred_RF[,2])^2 ) )
```

Try It!

- Use bagging function in *PivotalR* to create bagging regression tree for previous *diamonds* data. Does your bagging trees outperform the bagging linear models? Why or why not?

- Use bagging function in *PivotalR* to build your own RandomForest algorithm. Apply it to previous *diamonds* data and compare RMSEs. Does your own RandomForest outperform the built-in one in *MADlib*? Why or why not?

Final Comment

- For applications that involve analyzing massive & out-of-memory (e.g. bigger than total memory of a cluster of machines) datasets, using such in-database computing with MPP databases may be a good start.

- *Apache MADlib* is a great in-database machine learning algorithm library that allows us to use plain SQL to parallelize computations, manipulate huge datasets, and fit models within MPP databases. The *MADlib* + *PivotalR* is a powerful marriage that makes *MADlib* much more easier for non-SQL users.

- However, *MADlib* and *PivotalR* might be considered a immature big data analytics solution for now, as many features of modern machine learning algorithms are still missing.