Introduction to Database

Chieh-Sen (Jason) Huang

Department of Applied Mathematics

National Sun Yat-sen University

Thank Y. Kang (MIS/NSYSU) for providing slides.

Topic Overview

- Data Analytics v.s. Data Analysis
- Introduction to Database
- Introduction to In-Database Analytics
- Introduction to Massive Parallel Processing Database
- Apache MADlib and R

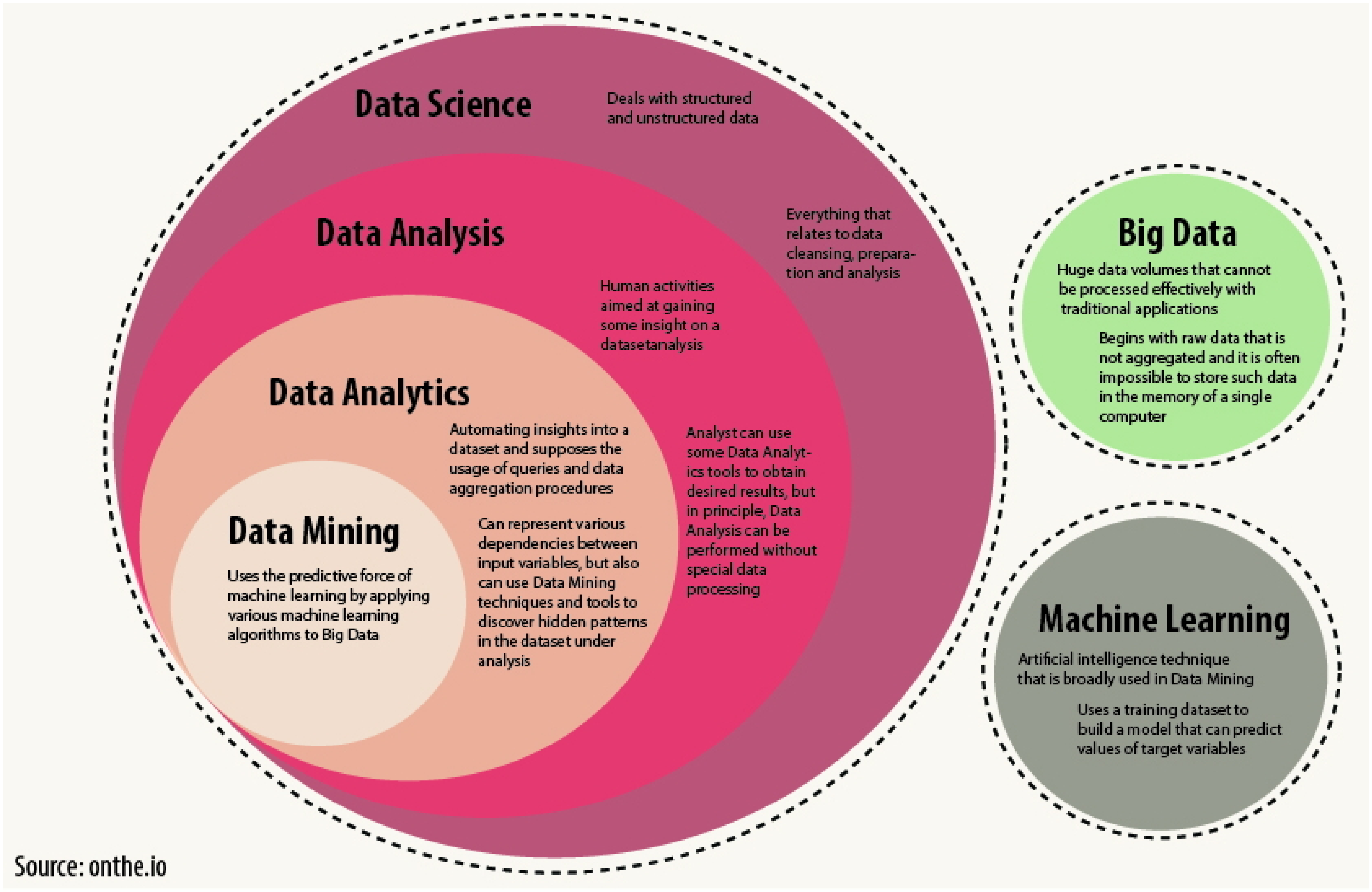
What is the difference between data analytics and analysis

- Analysis
  1. Analysis is separating out a whole into parts, study the parts individually and their relationships with one another.
  2. Analys is looks backwards over time, providing marketers with a historical view of what has happened.
  3. For example - if we have a whole data set and we are doing analysis on it means we pull sample data set from the whole data and then learn more about it and how it is related to the other samples.

- Analytics
  1. Analytics is the principle or logic that drives the analysis.
  2. Analytics look forward to model the future or predict a result.
  3. For example - As mentioned in above example when we pull the sample and do analysis on it, the techniques or logic we are using to analyse the sample is analytics.

- Therefore, we do analysis using analytics.

Data Science, Analysis, Analytics, Mining; Machine Learning; Big Data


Data Science
Deals with structured and unstructured data
Everything that relates to data cleansing, preparation and analysis
Data Analysis
Human activities aimed at gaining some insight on a datasetanalysis
Analyst can use some Data Analytics tools to obtain desired results, but in principle, Data Analysis can be performed without special data processing
Data Analytics
Automating insights into a dataset and supposes the usage of queries and data aggregation procedures
Can represent various dependencies between input variables, but also can use Data Mining techniques and tools to discover hidden patterns in the dataset under analysis
Data Mining
Uses the predictive force of machine learning by applying various machine learning algorithms to Big Data
Big Data
Huge data volumes that cannot be processed effectively with traditional applications
Begins with raw data that is not aggregated and it is often impossible to store such data in the memory of a single computer
Machine Learning
Artificial intelligence technique that is broadly used in Data Mining
Uses a training dataset to build a model that can predict values of target variables
Source: onthe.io

Definition of Database

- A database is a collection of information that is organized so that it can be easily accessed, managed and updated.

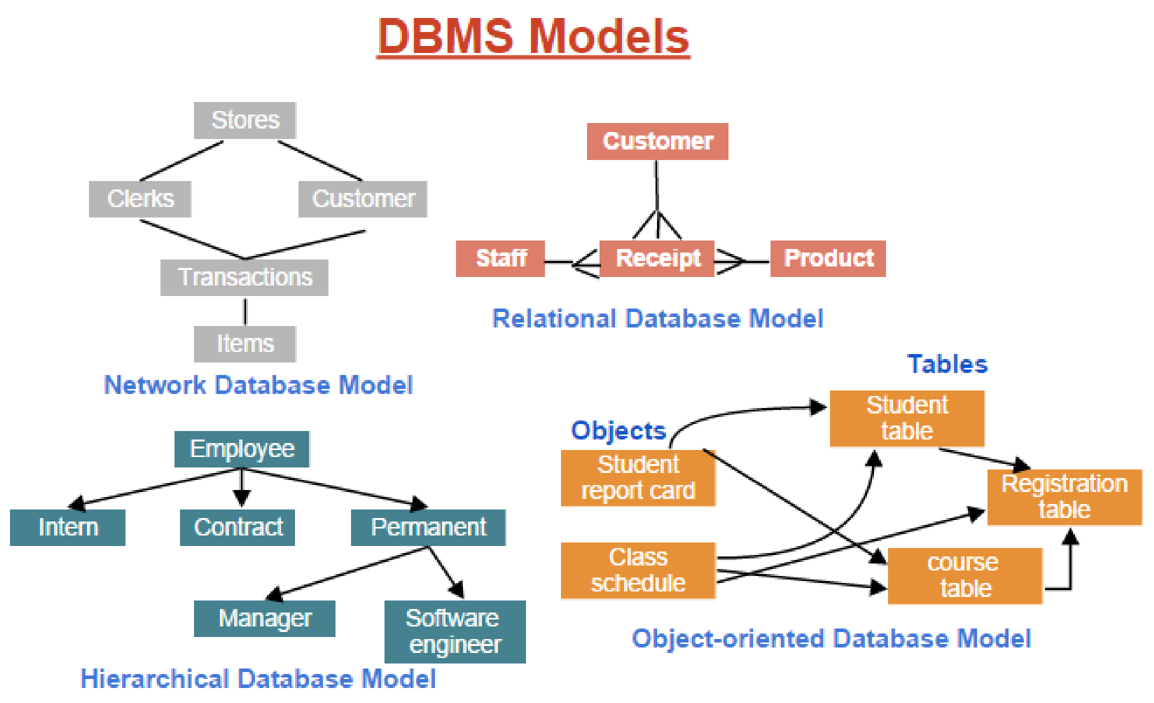
- Data is organized into rows, columns and tables, and it is indexed to make it easier to find relevant information. Data gets updated, expanded and deleted as new information is added. Databases process workloads to create and update themselves, querying the data they contain and running applications against it.

- A database manager provides users with the ability to control read/write access, specify report generation and analyze usage.

Evolution of Database

- Databases have evolved since their inception in the 1960s, beginning with hierarchical and network databases, through the 1980s with object-oriented databases, and today with SQL (Relational) and NoSQL (Not only SQL) databases and cloud databases.

  Database Management System (DBMS)



Hierarchical Database (1960s to Mid-1970s)

- Resemble an upside-down tree.
- Files are related in a parent-child manner, with each parent capable of relating to more than one child, but each child only being related to one parent.
- It represents one-to-many relationships (one parent has many children; for example, one company branch has many employees)
- It is the way most file systems work. It deals with unrelated files.
- Disadvantages: many-to-many relationships, accessing data requires to follow the entire path in the tree structure.

Network Database (1960s to Mid-1970s)

- The network database model was a progression from the hierarchical database model.
- Instead of only allowing each child to have one parent, this model allows each child to have multiple parents.
- It addresses the need to model more complex relationships such as the orders/parts many-to-many relationship.
- Disadvantages: It is more difficult to implement and maintain.

Relational Database (1970s to Mid-1980s)

- A relational database, invented by E.F. Codd at IBM in 1970, is a tabular database in which data is defined so that it can be reorganized and accessed in a number of different ways.
- Relational databases (also known as SQL databases) are made up of a set of tables with data that fits into a predefined category.
- Each table has at least one data category in a column, and each row has a certain data instance for the categories which are defined in the columns.
- The Structured Query Language (SQL) is the standard user and application program interface for a relational database. Relational databases are easy to extend, and a new data category can be added after the original database creation without requiring that you modify all the existing applications.

Relational Database (1970s to Mid-1980s)

- SQL statements are used both for interactive queries for information from a relational database and for gathering data for reports.

- For example, a typical business order entry database would include a table that described a customer with columns for name, address, phone number, and so forth. Another table would describe an order: product, customer, date, sales price, and so forth.

- A user of the database could obtain a view of the database that fitted the user's needs.

SQL (Structured Query Language)

- SQL (Structured Query Language) is a standardized programming language used for managing relational databases and performing various operations on the data in them.
- Initially created in the 1970s, SQL is regularly used by database administrators, as well as by developers writing data integration scripts and data analysts looking to set up and run analytical queries.
- The uses of SQL include modifying database table and index structures; adding, updating and deleting rows of data; and retrieving subsets of information from within a database for transaction processing and analytics applications.
- Commonly used SQL statements include select, add, insert, update, delete, create, alter and truncate.

SQL (Structured Query Language)

- An official SQL standard was adopted by the American National Standards Institute (ANSI) in 1986.
- Both proprietary and open source relational database management systems built around SQL are available for use by organizations. They include Microsoft SQL Server, Oracle Database, IBM DB2, SAP HANA, SAP Adaptive Server, MySQL (now owned by Oracle) and PostgreSQL.

Distributed and Cloud Database

- A distributed database is a database in which portions of the database are stored in multiple physical locations, and in which processing is dispersed or replicated among different points in a network.

- A cloud database is a database that has been optimized or built for a virtualized environment, either in a hybrid cloud, public cloud or private cloud. Cloud databases provide benefits such as the ability to pay for storage capacity and bandwidth on a per-use basis, and they provide scalability on demand, along with high availability.

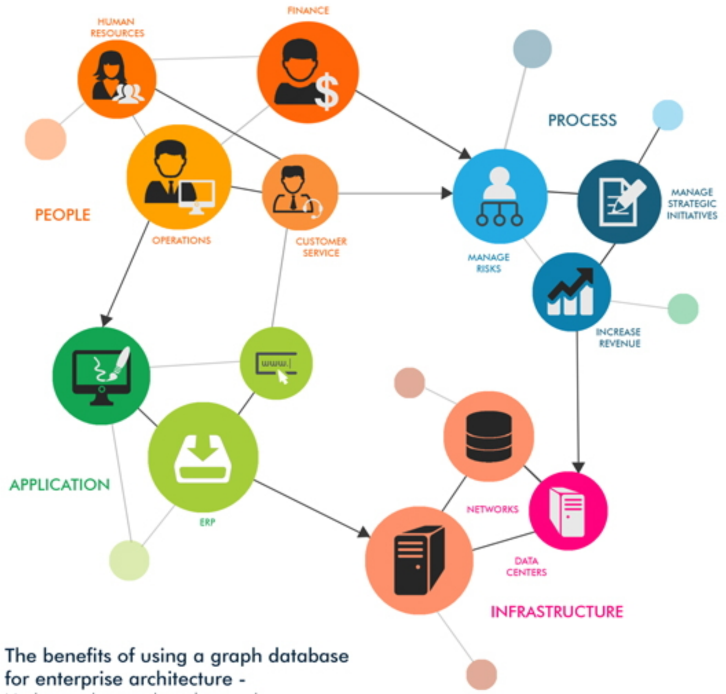
NoSQL, Object-oriented

- NoSQL databases are useful for large sets of distributed data.
- NoSQL databases are effective for big data performance issues that relational databases aren't built to solve. They are most effective when an organization must analyze large chunks of unstructured data or data that's stored across multiple virtual servers in the cloud.
- An object-oriented database is organized around objects rather than actions, and data rather than logic. For example, a multimedia record in a relational database can be a definable data object, as opposed to an alphanumeric value.

NoSQL, Graph Database

- A graph-oriented database, or graph database, is a type of NoSQL database that uses graph theory to store, map and query relationships. Graph databases are basically collections of nodes and edges, where each node represents an entity, and each edge represents a connection between nodes.

- Graph databases are growing in popularity for analyzing interconnections. For example, companies might use a graph database to mine data about customers from social media.



The benefits of using a graph database for enterprise architecture -
Understanding and analyzing dynamic connections between people, process and technology

Introduction to In-Database Analytics

- A database management system (DBMS) is a computer software application that interacts with the user, other applications, and the database itself to capture and analyze data.
- DBMS is a general term for a database with built in functions for data retrieval, data definition, and administration.
- A general-purpose DBMS is designed to allow the definition, creation, querying, update, and administration of databases.
- Well-known DBMSs include: **MySQL, PostgreSQL, MongoDB, MariaDB , Microsoft SQL Server, Oracle, Sybase, SAP HANA, MemSQL, IBM DB**.
- Under DBMS we have Relational data base management systems. These are platforms that stress the relationships as being important.

Introduction to In-Database Analytics(cont.)

- Computer scientists have been dealing massive data for years even before the advent of the buzzword "Big Data".
- 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 Expressionsthe 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.)

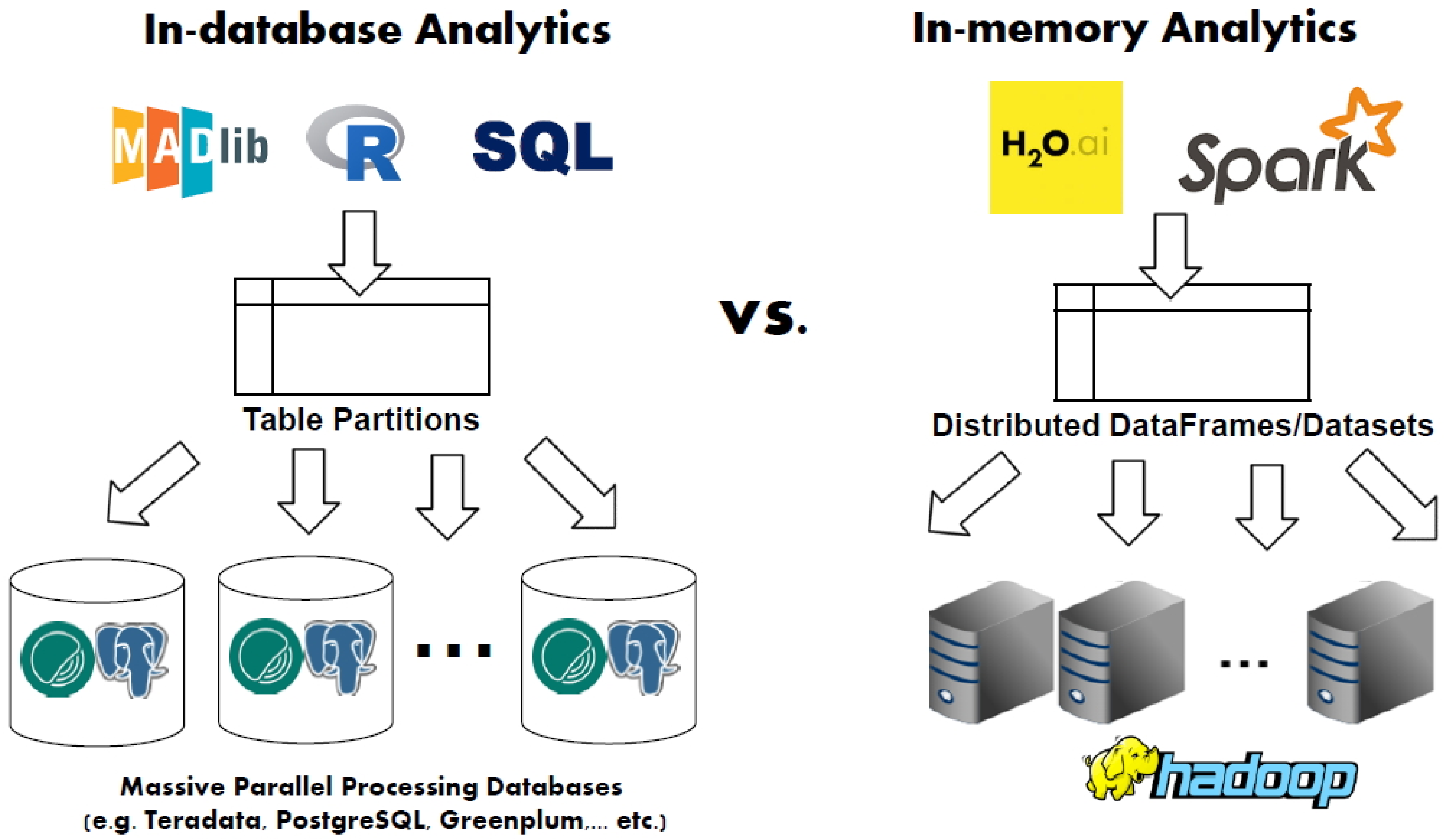
- Embedding analytics capabilities inside the DBMS reduces the overhead of moving data back and forth.
- Most statistical packages and data mining applications require datasets to be loaded & fit in RAM, which limits the possibility of analyzing big data.
- Processing data right in the database systems, 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.

Introduction to In-Database Analytics(cont.)

- 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


In-database Analytics
MADlib
R
SQL
Table Partitions
Massive Parallel Processing Databases
(e.g. Teradata, PostgreSQL, Greenplum,... etc.)
vs.
In-memory Analytics
H2O.ai
Spark
Distributed DataFrames/Datasets
hadoop

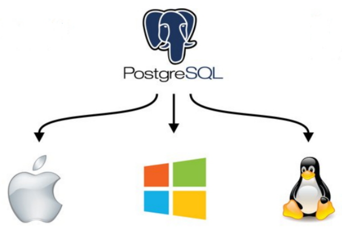
PostgreSQL for Big Data

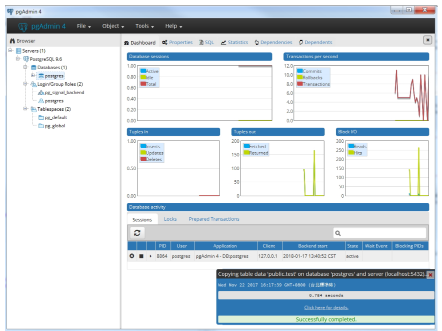
- PostgreSQL is a powerful open source object-relational database system with most up-to-date & sophisticated features/implementations of advanced database systems.

- It runs on all major operating systems, including Linux, UNIX (AIX, BSD, HP-UX, macOS, Solaris), and Windows.

- PostgreSQL runs stored procedures in more than a dozen programming languages, including Java, Perl, Python, Ruby, Tcl, C/C++, and its own PL/pgSQL, which is similar to Oracle's PL/SQL.

- PostgreSQL allows various languages both compiled and interpreted to interface with PostgreSQL. There are interfaces for Java (JDBC), ODBC, Perl, Python, Ruby, C, C++, PHP, Lisp, Scheme, and Qt.





			PID	User	Application	Client	Backend start	State	Wait Event	Blocking PIDs
			8864	postgres	pgAdmin 4 - DB:postgres	127.0.0.1	2018-01-17 13:40:52 CST	active		

Copying table data 'public.test' on database 'postgres' and server (localhost:5432)...

Wed Nov 22 2017 16:17:39 GMT+0800 (台北標準時)

0.784 seconds

Click here for details.

Successfully completed.

Introduction to Greenplum

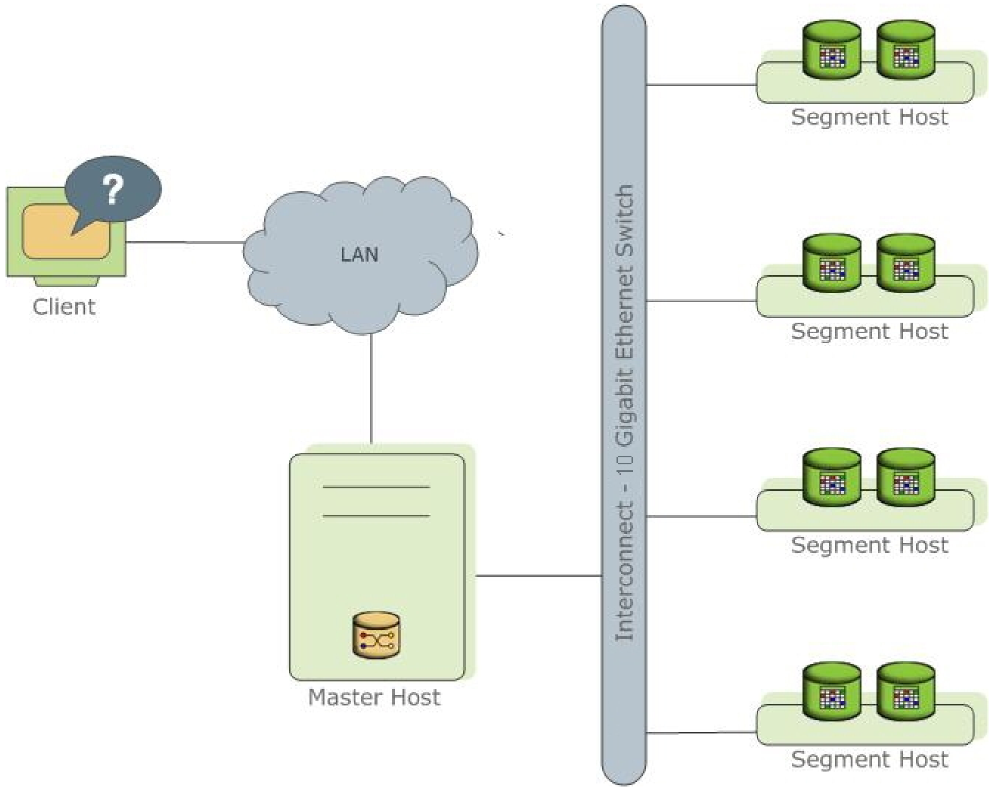
- Greenplum Database is an advanced open source data platform. It provides powerful and rapid analytics on **PETABYTE SCALE** data volumes.

- Greenplum Database, also a derivative of PostgreSQL, is a highly-scalable, shared-nothing **MASSIVELY PARALLEL PROCESSING** MPP database system able to handle storage and processing of massive data by distributing the load across several nodes.

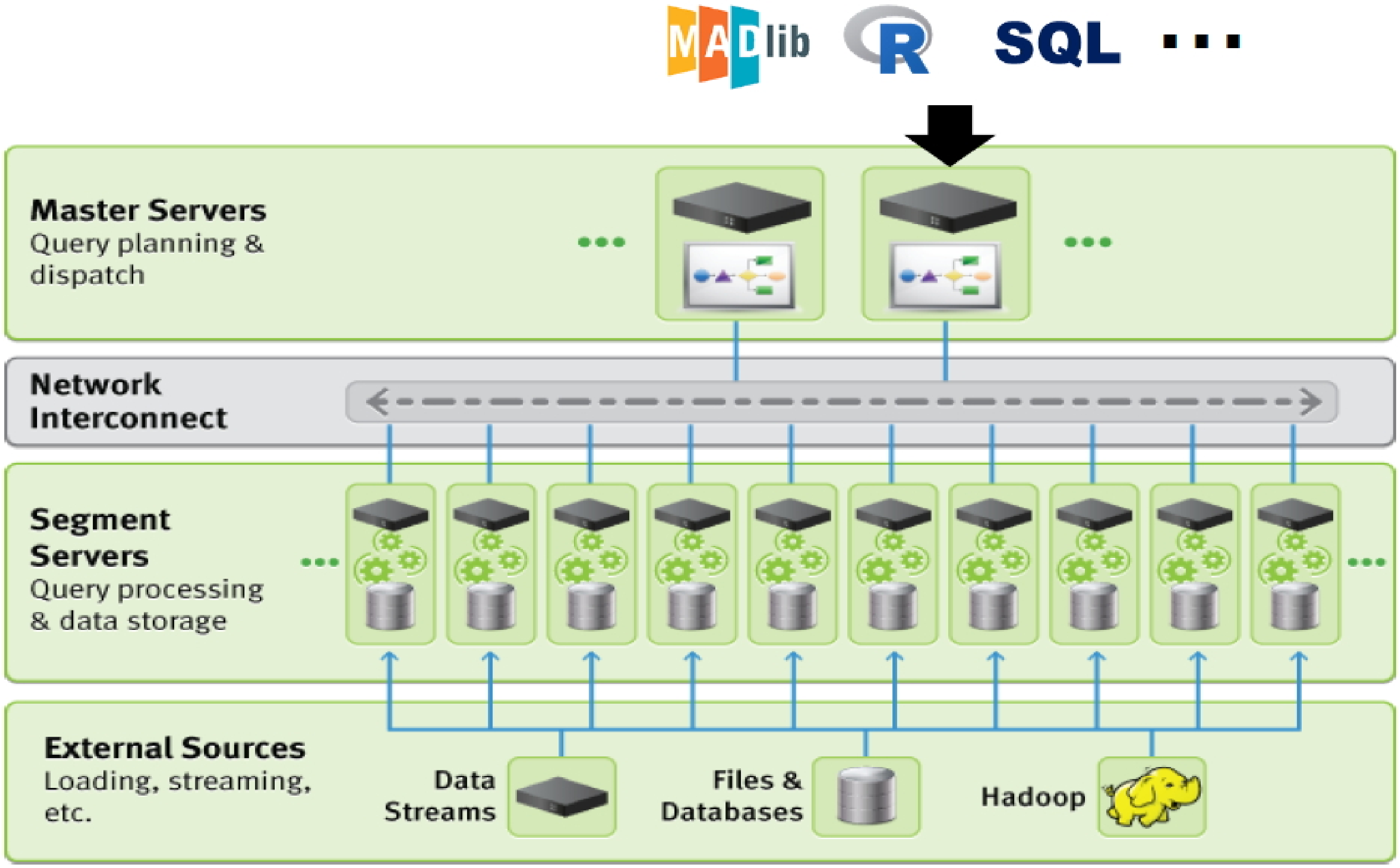
- Greenplum Database is capable with **ADVANCED MACHINE LEARNING**. Provided by Apache MADlib, a library for scalable in-database analytics extending the SQL capabilities of Greenplum Database through user-defined functions.

Introduction to Greenplum(cont.)

- 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.



Introduction to Greenplum(cont.)



A bit about table distribution strategy

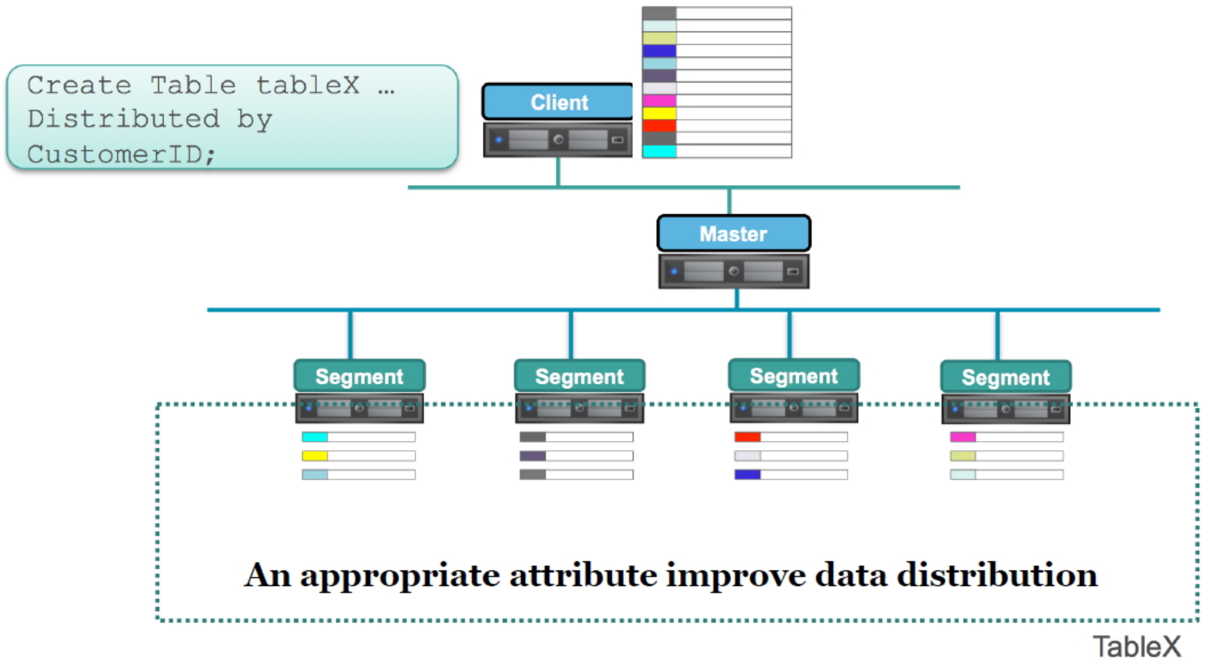
- Table distribution techniques are crucial to the workload balancing among all computing nodes in most distributed database systems.

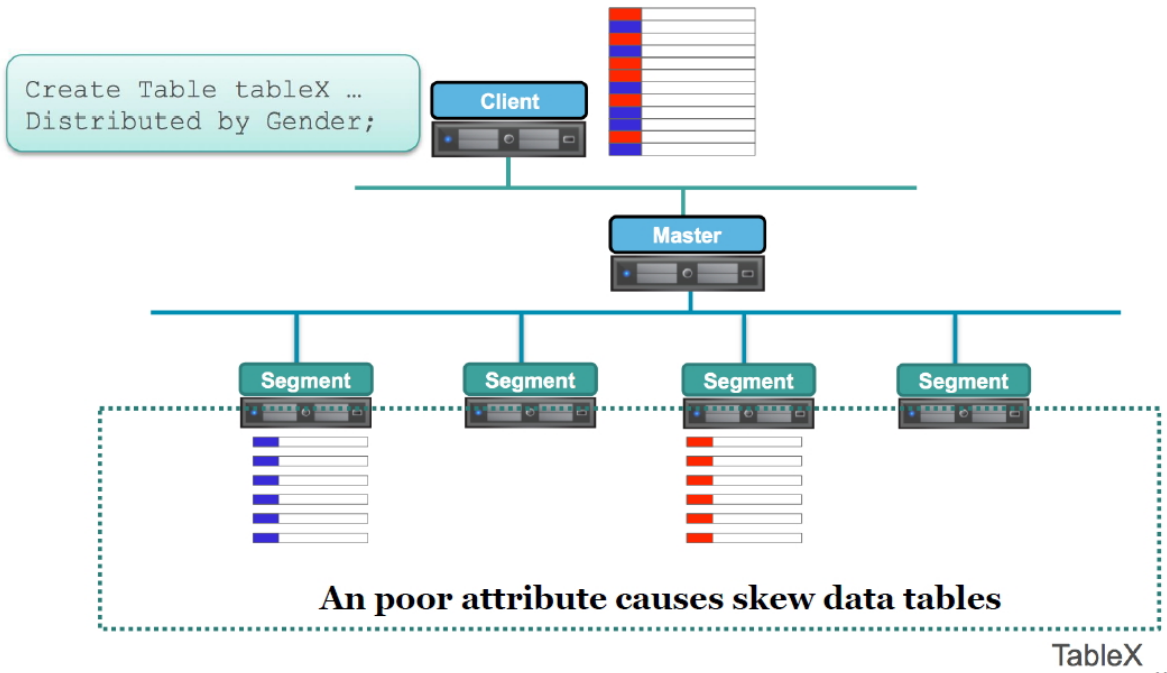
- In Greenplum, all database tables are distributed across all segments, and the performance of data processing & analytics depends heavily on the table distribution strategy.

- Greenplum supports two common distributions:

  1. 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.
  2. 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.)





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).

- BI encompasses a wide variety of tools, applications and methodologies that enable organizations to collect data from internal systems and external sources; prepare it for analysis; develop and run queries against that data; and create reports, dashboards and data visualizations to make the analytical results available to corporate decision–makers, as well as operational workers

MAD Skills(cont.)

- MAD Skills is the most popular new data analytics practice, where "MAD" stand for:
  - **Magnetic** **Agile** **Deep**

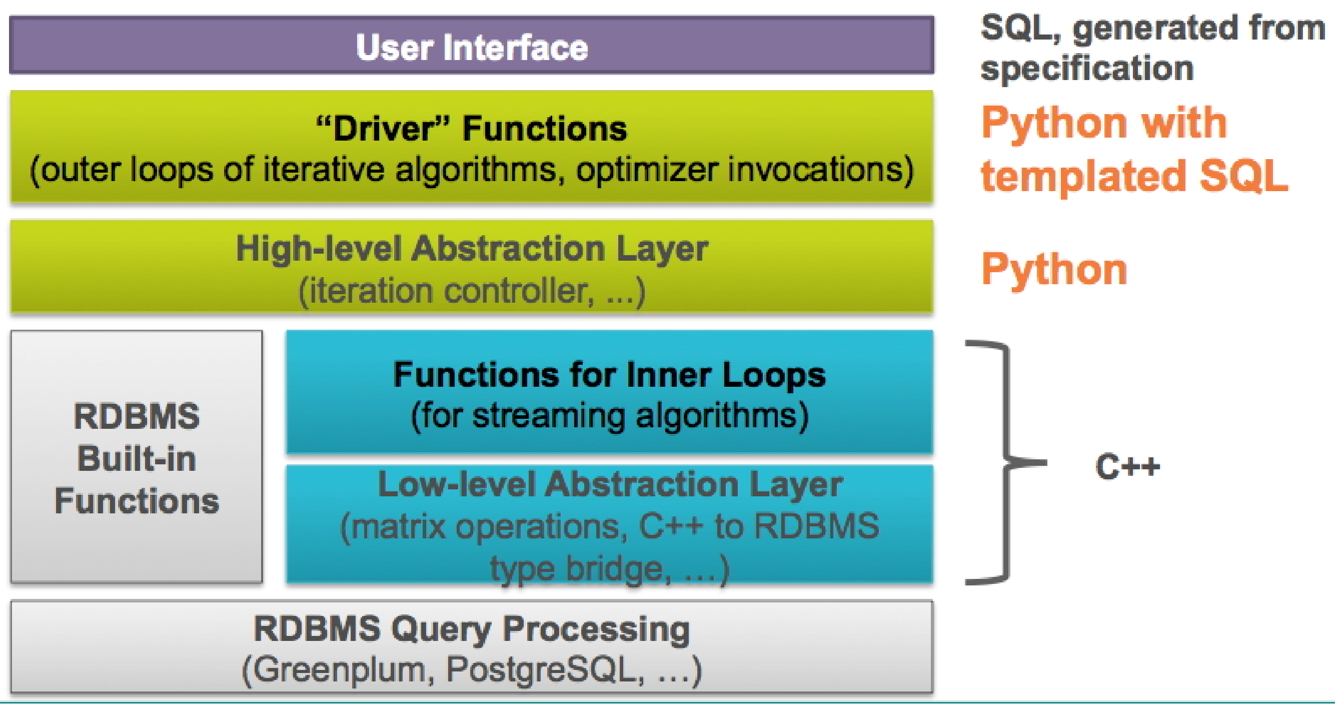
MAD Skills(cont.)

- **Magnetic**: Traditional data warehouse discourages the incorporation of data until they are carefully cleaned 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 incontinuous 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.
- Roll–up allows us to look at coarser, big picture data by dropping one or more dimensions or climbing up along the dimension hierarchies. A drill–down operation is the opposite of roll–up for more detailed data.

Apache MADlib

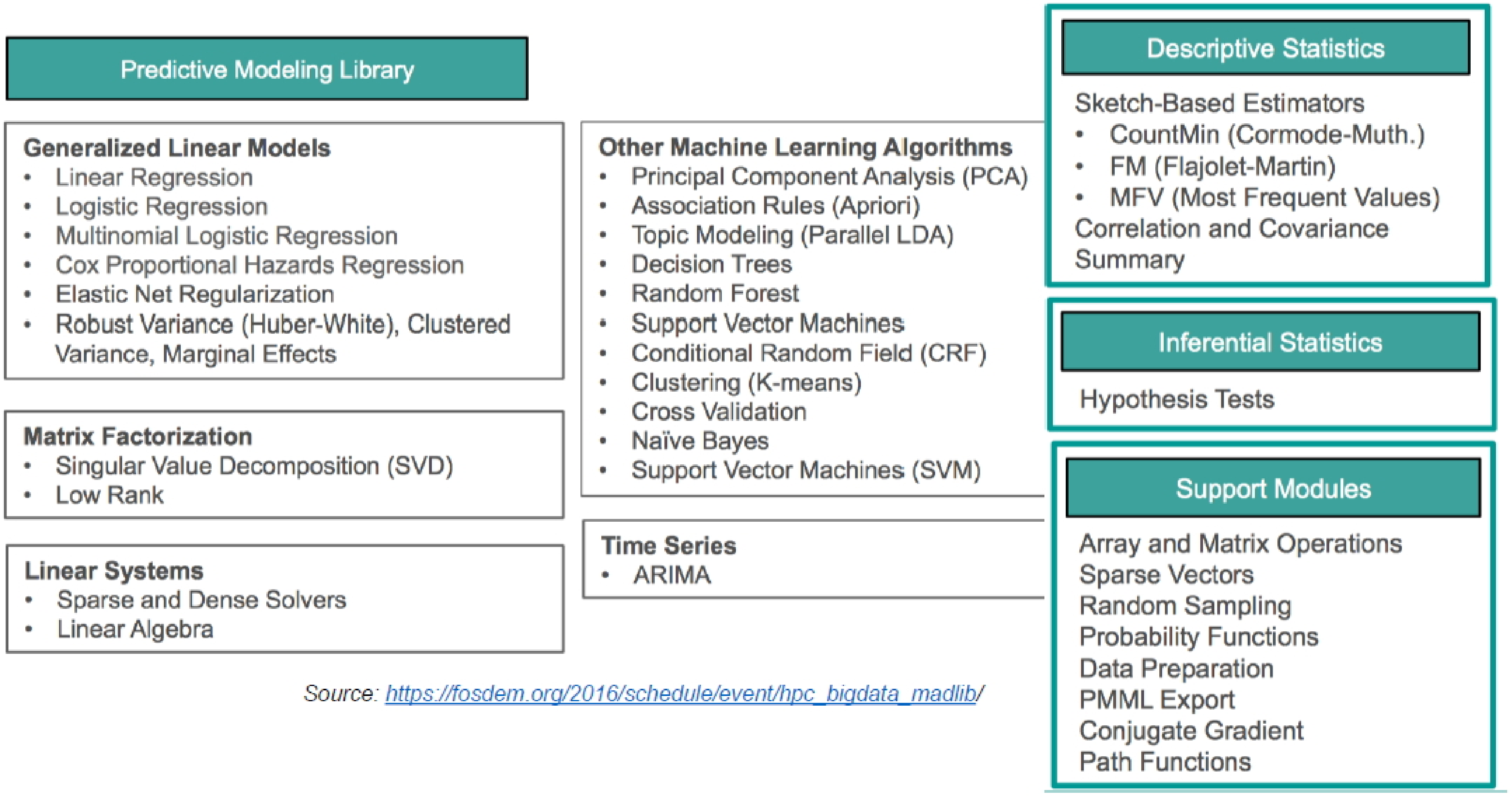
- The MADlib, an incubating project in Apache Software Foundation(originally developed by UC Berkeley and Greenplum), is the product followed by the publication of the MAD Skills related research.

- 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).


User Interface
SQL, generated from specification
"Driver" Functions
(outer loops of iterative algorithms, optimizer invocations)
Python with templated SQL
High-level Abstraction Layer
(iteration controller, ...)
Python
RDBMS Built-in Functions
Functions for Inner Loops
(for streaming algorithms)
Low-level Abstraction Layer
(matrix operations, C++ to RDBMS type bridge, ...)
C++
RDBMS Query Processing
(Greenplum, PostgreSQL, ...)

Apache MADlib

- Check http://madlib.incubator.apache.org/docs/latest/ for more information.



Predictive Modeling Library

Generalized Linear Models
- Linear Regression
- Logistic Regression
- Multinomial Logistic Regression
- Cox Proportional Hazards Regression
- Elastic Net Regularization
- Robust Variance (Huber-White), Clustered Variance, Marginal Effects

Matrix Factorization
- Singular Value Decomposition (SVD)
- Low Rank

Linear Systems
- Sparse and Dense Solvers
- Linear Algebra

Other Machine Learning Algorithms
- Principal Component Analysis (PCA)
- Association Rules (Apriori)
- Topic Modeling (Parallel LDA)
- Decision Trees
- Random Forest
- Support Vector Machines
- Conditional Random Field (CRF)
- Clustering (K-means)
- Cross Validation
- Naïve Bayes
- Support Vector Machines (SVM)

Time Series
- ARIMA

Descriptive Statistics

Sketch-Based Estimators
- CountMin (Cormode-Muth.)
- FM (Flajolet-Martin)
- MFV (Most Frequent Values)

Correlation and Covariance

Summary

Inferential Statistics

Hypothesis Tests

Support Modules

Array and Matrix Operations
Sparse Vectors
Random Sampling
Probability Functions
Data Preparation
PMML Export
Conjugate Gradient
Path Functions

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

PostgreSQL

- Login to the system

```
fisher/home/huangcs 2: psql -h 192.168.30.2 -d postgres -U huangcs
Password for user huangcs:
Welcome to psql 8.2.11 (server 8.2.15), the PostgreSQL interactive
terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help with psql commands
       \g or terminate with semicolon to execute query
       \q to quit

#/usr/local/greenplum-db-4.3.16.0/bin/psql
```

- Change password

```
postgres=> \password
Enter new password:
Enter it again:
postgres=>
```

- Tutorial:: www.tutorialspoint.com/postgresql/index.htm

PostgreSQL

- List all databases

```
postgres=> \list
                        List of databases
   Name     |    Owner     | Encoding |   Access privileges
-----------+-------------+-----------+------------------------
 ipass      | gpadmin     | UTF8      | =Tc/gpadmin          +
            |             |           | gpadmin=CTc/gpadmin  +
            |             |           | m052040032=CTc/gpadmin+
            |             |           | d052040002=CTc/gpadmin
 kmrt       | gpadmin     | UTF8      |
 postgres   | gpadmin     | UTF8      |
 template0  | gpadmin     | UTF8      | =c/gpadmin           +
            |             |           | gpadmin=CTc/gpadmin
 template1  | gpadmin     | UTF8      | =c/gpadmin           +
            |             |           | gpadmin=CTc/gpadmin
(5 rows)
```

- =# means “super user”.

```
postgres=#
```

PostgreSQL

- Create a database

```
postgres=>CREATE DATABASE test;
postgres=> \l
                     List of databases
   Name     |   Owner     | Encoding |   Access privileges
------------+-------------+----------+------------------------
 ipass      | gpadmin     | UTF8     | =Tc/gpadmin           +
            |             |          | gpadmin=CTc/gpadmin   +
            |             |          | m052040032=CTc/gpadmin+
            |             |          | d052040002=CTc/gpadmin
 kmrt       | gpadmin     | UTF8     |
 postgres   | gpadmin     | UTF8     |
 template0  | gpadmin     | UTF8     | =c/gpadmin            +
            |             |          | gpadmin=CTc/gpadmin
 template1  | gpadmin     | UTF8     | =c/gpadmin            +
            |             |          | gpadmin=CTc/gpadmin
 test       | m052040032  | UTF8     |
(6 rows)
```

- Change database

```
postgres=> \connect ipass
You are now connected to database "ipass" as user "huangcs".
ipass=>
```

PostgreSQL

- View size of all databases

```
ipass=> SELECT pg_size_pretty(SUM(pg_database_size(datname))::bigint ) As human_size
ipass->  ,SUM(pg_database_size(datname) )::bigint  As raw_size
ipass->  ,pg_size_pretty( (SUM(pg_database_size(datname))
ipass->    - pg_database_size(current_database()))::bigint ) As h_without_current
ipass->  FROM pg_database;

 human_size |   raw_size    | h_without_current
------------+---------------+--------------------
 130 GB     | 139754762996 | 128 GB
(1 row)
```

- View size of one databases

```
ipass=> SELEct PG_size_pretty( pg_database_size('ipass') ) As human_size
    , pg_database_size('ipass') As raw_size;
 human_size |  raw_size
------------+------------
 320 MB     | 2709587834
(1 row)
```

PostgreSQL

- Create table; create a DEPARTMENT table with ID as primary key and NOT NULL are the constraints showing that these fields cannot be NULL while creating records in this table.

```
ipass=> CREATE TABLE DEPARTMENT(
ipass-> ID INT PRIMARY KEY      NOT NULL,
ipass-> DEPT           CHAR(50) NOT NULL,
ipass-> EMP_ID         INT      NOT NULL);
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index
"department_pkey" for table "department" CREATE TABLE
ipass=> \d department
```

```
     Table "public.department"
 Column |     Type      | Modifiers
--------+---------------+-----------
 id     | integer       | not null
 dept   | character(50) | not null
 emp_id | integer       | not null
Indexes:
    "department_pkey" PRIMARY KEY, btree (id)
Distributed by: (id)
```

PostgreSQL

- Insert a row into table

```
ipass=>INSERT INTO DEPARTMENT (id, dept, emp_id)
ipass->VALUES (1, 'Math',2);
INSERT 0 1
ipass=>INSERT INTO DEPARTMENT (id, dept, emp_id)
ipass->VALUES (2, 'Chem',3);
INSERT 0 1
ipass=>INSERT INTO DEPARTMENT (id, dept, emp_id)
ipass->VALUES (3, 'Phy',1);
INSERT 0 1
ipass=> SELECT * FROM DEPARTMENT;
 id |                    dept                    | emp_id
----+--------------------------------------------+---------
  3 | Phy                                        |       1
  2 | Chem                                       |       3
  1 | Math                                       |       2
(3 rows)
```

```
INSERT 0  OID (Object Identifers)
       1  "1" stands for the number of the records added
```

PostgreSQL

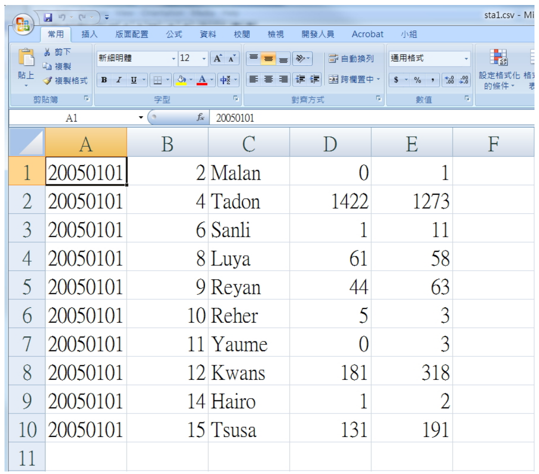
- SELECT statement is used to fetch the data from a database table

```
ipass=>SELECT id, dept FROM DEPARTMENT;
 id |                    dept
----+----------------------------------------------
  4 | Bio
  2 | Chem
  3 | Phy
  1 | Math
(4 rows)

ipass=# DELETE FROM DEPARTMENT WHERE ID = 3;
DELETE 1
ipass=# SELECT * FROM DEPARTMENT;
 id |                    dept                    | emp_id
----+--------------------------------------------+--------
  1 | Math                                       |      2
  4 | Bio                                        |      4
  2 | Chem                                       |      3
(3 rows)
```

PostgreSQL

- Load a table from a CSV file (sta1.csv), need to remove the table of content first from the csv file.

```
ipass=> CREATE TABLE station(yyyymmdd  char(8)
ipass-> ,stationcode  integer ,stationname  varchar(5)
ipass-> ,input   integer, output   integer);

ipass=# COPY station FROM '/home/huangcs/Postgres/sta1.csv' delimiter ',' csv header;
COPY 9
ipass=# SELECT * FROM station;
 yyyymmdd | stationcode | stationname | input | output
----------+-------------+-------------+-------+--------
 20050101 |           4 | Tadon       |  1422 |   1273
 20050101 |           6 | Sanli       |     1 |     11
 20050101 |           8 | Luya        |    61 |     58
 20050101 |           9 | Reyan       |    44 |     63
 20050101 |          10 | Reher       |     5 |      3
 20050101 |          11 | Yaume       |     0 |      3
 20050101 |          12 | Kwans       |   181 |    318
 20050101 |          14 | Hairo       |     1 |      2
 20050101 |          15 | Tsusa       |   131 |    191
(9 rows)
```



	A	B	C	D	E	F
1	20050101	2	Malan	0	1	
2	20050101	4	Tadon	1422	1273	
3	20050101	6	Sanli	1	11	
4	20050101	8	Luya	61	58	
5	20050101	9	Reyan	44	63	
6	20050101	10	Reher	5	3	
7	20050101	11	Yaume	0	3	
8	20050101	12	Kwans	181	318	
9	20050101	14	Hairo	1	2	
10	20050101	15	Tsusa	131	191	
11						

PostgreSQL

- List the sizes of all tables in the database

```
ipass=> SELECT relname as "Table",
ipass-> pg_size_pretty(pg_total_relation_size(relid)) As "Size",
ipass-> pg_size_pretty(pg_total_relation_size(relid) - pg_relation_size(relid))
ipass-> as "External Size"
ipass-> FROM pg_catalog.pg_statio_user_tables ORDER BY
ipass-> pg_total_relation_size(relid) DESC;

                         Table                         |  Size    | External Size
-------------------------------------------------------+----------+---------------
 weather                                               | 118 MB   | 0 bytes
 weathers                                              | 59 MB    | 0 bytes
 p2_diamonds                                           | 7200 kB | 1056 kB
 p_diamonds                                            | 7200 kB | 1056 kB
 db_diamonds_few_vars                                  | 5152 kB | 1056 kB
 dt_golf                                               | 1568 kB | 1056 kB
 department                                            | 1248 kB | 1152 kB
 train_output_group                                    | 1120 kB | 1056 kB
 diamonds_lm_summary                                   | 1088 kB | 1056 kB
 diamonds_lm1                                          | 1088 kB | 1056 kB
 diamonds_lm                                           | 1088 kB | 1056 kB
```

- List the sizes of one table

```
ipass=# SELECT 'weathers' as "Table"
ipass-> ,pg_size_pretty(pg_total_relation_size('weathers')) As "Size";
  Table    | Size
----------+-------
 weathers | 59 MB
(1 row)
```

PostgreSQL

- A schema is a named collection of tables.
- It organizes database objects into logical groups to make them more manageable.
- Schemas are analogous to directories at the operating system level, except that schemas cannot be nested.

```
ipass=# create schema jason_schema;
CREATE SCHEMA
ipass=# \dn
                   List of schemas
                    Name                    |  Owner
--------------------------------------------+---------
 gp_toolkit                                 | gpadmin
 information_schema                         | gpadmin
 jason_schema                               | huangcs
 madlib                                     | gpadmin
 madlib_installcheck_recursive_partitioning | gpadmin
 pg_aoseg                                   | gpadmin
 pg_bitmapindex                             | gpadmin
 pg_catalog                                 | gpadmin
 pg_toast                                   | gpadmin
 public                                     | gpadmin
(10 rows)
```

PostgreSQL

- Show search path, set search path

```
ipass=# show search_path;
  search_path
----------------
 "$user",public
(1 row)

ipass=# set search_path="jason_schema";
SET
ipass=# show search_path;
 search_path
--------------
 jason_schema
(1 row)
```

PostgreSQL

- We could create “the same” table “department”, but now, it is under jason_schema

```
ipass=# \dt
                      List of relations
    Schema     |    Name     | Type  |  Owner   | Storage
---------------+-------------+-------+----------+---------
 jason_schema | company     | table | huangcs | heap
 jason_schema | department | table | huangcs | heap
(2 rows)

ipass=# INSERT INTO DEPARTMENT (id, dept, emp_id) VALUES (1, 'Math',2);
INSERT 0 1
ipass=# SELECT * FROM DEPARTMENT;
 id |                        dept                        | emp_id
----+----------------------------------------------------+--------
  1 | Math                                               |      2
(1 row)

ipass=# SELECT * FROM public.DEPARTMENT;
 id |                        dept                        | emp_id
----+----------------------------------------------------+--------
  4 | Bio                                                |      4
  2 | Chem                                               |      3
  1 | Math                                               |      2
(3 rows)
```

PostgreSQL

- Table schema is the structure of the table that defines the objects in the table.

- We create a new table using existing table schema and data.

```
ipass=# create table department_new as select * from department;
SELECT 1
ipass=# \dt
                     List of relations
    Schema     |      Name      | Type  |  Owner   | Storage
--------------+----------------+-------+----------+---------
 jason_schema | company        | table | huangcs | heap
 jason_schema | department     | table | huangcs | heap
 jason_schema | department_new | table | huangcs | heap
(3 rows)

ipass=# SELECT * FROM department_new;
 id |                         dept                         | emp_id
----+------------------------------------------------------+--------
  1 | Math                                                 |      2
(1 row)
```

- Drop Database, Drop Table and more .... see https://www.tutorialspoint.com/postgresql/index.htm

PostgreSQL

- Linear regression using MADlib. (Table diamonds has been uploaded.)

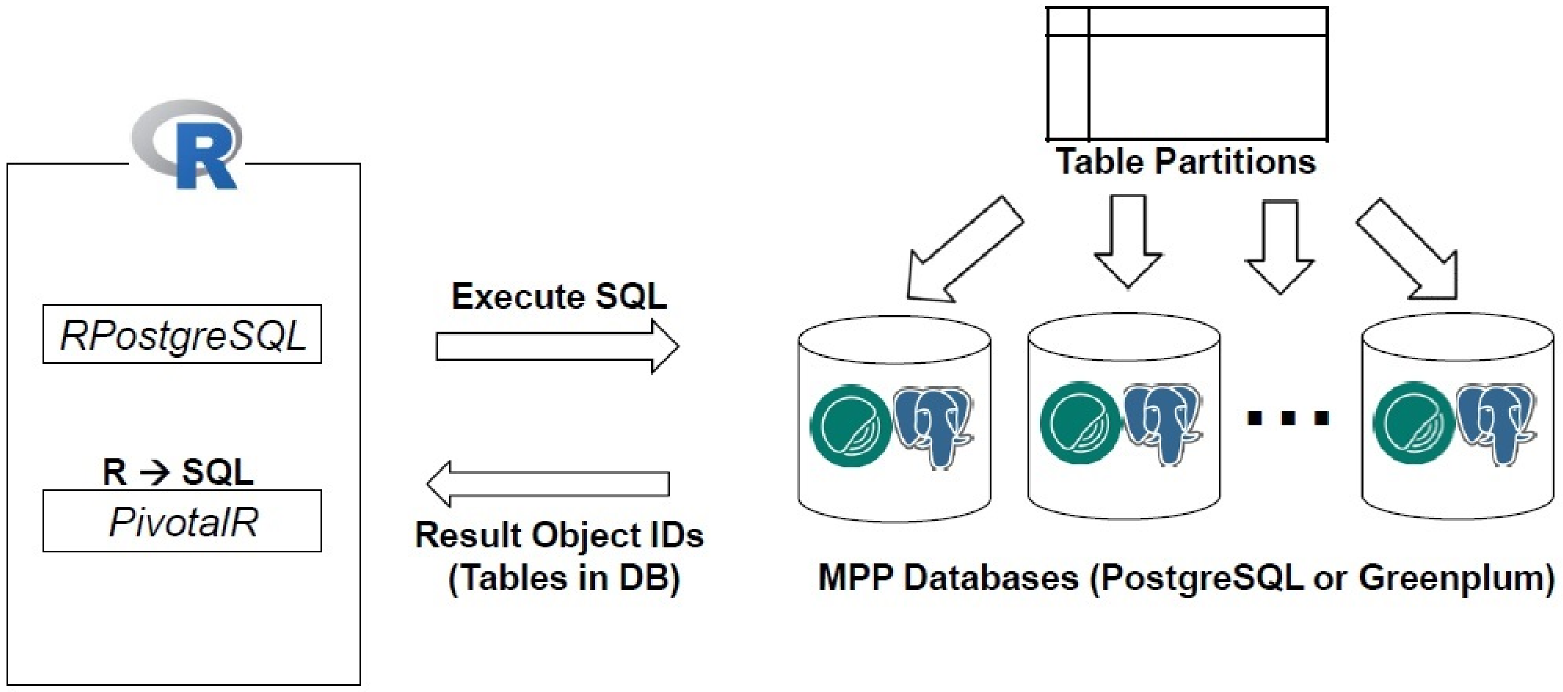
```
ipass=# select * from diamonds;
 row.names | carat |    cut    | color | clarity | depth | table | price |   x   |   y   |  z
-----------+-------+-----------+-------+---------+-------+-------+-------+-------+-------+------
 25        |  0.31 | Very Good | J     | SI1     |  58.1 |    62 |   353 |  4.44 |  4.47 | 2.59
 43        |  0.26 | Good      | D     | VS2     |  65.2 |    56 |   403 |  3.99 |  4.02 | 2.61
.....
ipass=# SELECT madlib.linregr_train('diamonds','diamonds_lm', 'price', 'ARRAY[1, carat, x]');
 linregr_train
---------------

 (1 row)
ipass=# \dt
                                    List of relations
   Schema     |                          Name                          | Type  |  Owner  | Storage
--------------+--------------------------------------------------------+-------+---------+---------
 jason_schema | diamonds                                               | table | huangcs | heap
 jason_schema | diamonds_lm                                            | table | huangcs | heap
 jason_schema | diamonds_lm_summary                                    | table | huangcs | heap
 pg_temp_117  | __madlib_temp_73474105_1522229665_22873576__           | table | huangcs | heap
(3 rows)

ipass=# select * from diamonds_lm;
        coef              |  r2  |   std_err          |  t_stats           |  p_values  |  ......
--------------------------+------+-----------------------------------------------------+------
 {1737.9,10125.9,-1026.8} | 0.85 | {103.6,62.5,26.4} | {16.7,161.8,-38.8} |{5.6e-63,0,0}| .......
(1 row)
ipass=# select * from diamonds_lm_summary;
 method  | source_table |  out_table  | dependent_varname | independent_varname | num_rows_.....
---------+--------------+-------------+-------------------+---------------------+----------+...
 linregr | diamonds     | diamonds_lm | price             | ARRAY[1, carat, x]  |    53940 |..
(1 row)
```

Apache MADlib and R

- We will use 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.
- 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.

The Connection Between R and PostgreSQL or Greenplum

- RPostgreSQL and PivotalR are used to connect to massively parallel processing (MPP) database


RPostgreSQL
R → SQL
PivotalR
Execute SQL
Result Object IDs (Tables in DB)
Table Partitions
MPP Databases (PostgreSQL or Greenplum)

Hands-on RPostgreSQL

- Build the connection

```
library(DBI); library(RPostgreSQL); library(ggplot2)
diamonds = as.data.frame(diamonds) # Dataset ggplot2::diamonds
## Using RPostgreSQL/DBI
# Create a PostgreSQL DB connection
>pgConn= dbConnect(RPostgreSQL::PostgreSQL(), host = "192.168.30.2"
          , dbname= "ipass", user="yourID", password="XXXXX")
```

- List all tables in the database

```
>dbListTables(pgConn)# List all tables in the database
 [1] "__gp_localid"
 [2] "__gp_masterid"
 [3] "__gp_log_segment_ext"
 [4] "__gp_log_master_ext"
 [5] "gp_disk_free"
 [6] "migrationhistory"
........
```

Hands-on RPostgreSQL

- Create table "diamonds". Delete if it exists.

```
# Create table "diamonds". Delete if it exists.
>ifelse(dbExistsTable(pgConn, "diamonds"), dbRemoveTable(pgConn,
       "diamonds"), dbWriteTable(pgConn, "diamonds", diamonds))
>dbReadTable(pgConn, "diamonds") # Read existing table
      carat       cut color clarity depth table price     x     y     z
15     0.20   Premium     E     SI2  60.2  62.0   345  3.79  3.75  2.27
64     0.30     Ideal     D     SI1  62.1  56.0   552  4.30  4.33  2.68
101    0.75 Very Good     D     SI1  63.2  56.0  2760  5.80  5.75  3.65
...........

>dbListFields(pgConn,"diamonds") # Get Column names
[1] "row.names" "carat"     "cut"       "color"     "clarity"   "depth"     "table"     "price"
[9] "x"         "y"         "z"
```

- Run SQL statements

```
>rs= dbSendQuery(pgConn, statement = "SELECT count(*) FROM diamonds");dbFetch(rs);
  count
1 53940
# Clear by using dbClearResult(rs); rm(rs);
# dbGetQuery comes with a default implementation that calls dbSendQuery,
# then dbFetch, ensuring that the result is always free-d by dbClearResult
> dbGetQuery(pgConn, "SELECT count(*) FROM diamonds")
  count
1 53940
```

Hands-on RPostgreSQL

- Table summary statistics

```
dbSendQuery(pgConn, "SELECT * FROM madlib.summary('diamonds',
            'diamonds_summary')")
dia_sum= dbReadTable(pgConn, "diamonds_summary"); dia_sum
dbRemoveTable(pgConn,"diamonds_summary") # remove table
# Correlation matrix
dbSendQuery(pgConn, "SELECT * from madlib.correlation( 'diamonds',
            'diamonds_pearson', 'price, depth, carat')")
# 2 tables are generated
>dbReadTable(pgConn, "diamonds_pearson")
  column_position variable      price      depth carat
1               2    depth -0.0106474 1.00000000    NA
2               1    price  1.0000000         NA    NA
3               3    carat  0.9215913 0.02822431     1
>dbReadTable(pgConn, "diamonds_pearson_summary")
       method   source     output_table        column_names
1 Correlation diamonds diamonds_pearson {price,depth,carat}
          mean_vector total_rows_processed total_rows_skipped
1 {3932.7,61.7,0.797}                53940                  0
```

Hands-on RPostgreSQL

- Linear Regression

```
Call:
lm(formula = price ~ carat + depth, data = diamonds)
Coefficients:
(Intercept)          carat          depth
     4045.3         7765.1         -102.2

>dbSendQuery(pgConn, "select madlib.linregr_train('diamonds',
              'diamonds_lm','price', 'ARRAY[1, carat, depth]')")
>dbReadTable(pgConn, "diamonds_lm")
                                               coef        r2
1 {4045.33318266443,7765.14066376722,-102.16532215902} 0.8506755
```

- Remove tables if we don't need them

```
Map(function(tb) dbRemoveTable(pgConn, tb),
    c("diamonds_pearson", "diamonds_pearson_summary") )
```

Using PivotalR

- Using SQL code may be a pain in the neck. PivotalR connects to the database through the RPostgreSQL.
- PivotalR's functions and syntax for the manipulation of tables are very similar to those for data.frame manipulation defined natively in R.
- PivotalR utilizes the full power of parallel computation and distributive storage, and thus gives the normal R user access to big data.
- Therefore PivotalR benefits R users by providing an easy-to-access interface for manipulating data tables stored in PostgreSQL database system and a combination of the powers of R and MADlib.
- The user does not need to worry about the restriction of memory size even if the data size is very big, because PivotalR minimizes the amount of data transferred between the database and R. The user manipulates the data from R but the data itself stays in the database.

Apache MADlib and PivotalR

- PivotalR

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

- SQL

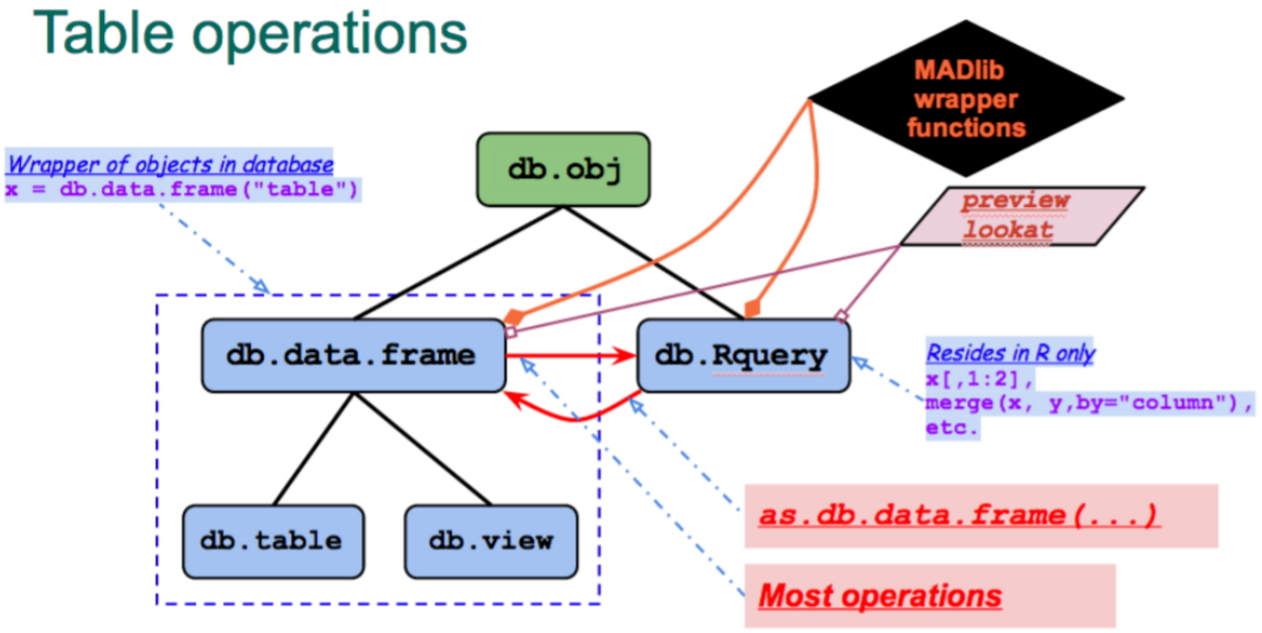
```
SELECT madlib.linregr_train('diamonds', 'diamonds_lm',
'price', 'ARRAY[1, carat, depth]');
```

Work Flow of PivotalR

- First, the user uses "db.connect" to connect to a database.
- Then, "db.data.frame" can be used to create a wrapper for a data table in the database.
- Minimal information about the table is kept in the wrapper, and no data is loaded into the memory. During this data preparation step, no data is loaded.
- The user can easily operate on the "db.data.frame" wrapper object and any operation creates a "db.Rquery" object, which is just a wrapper for a series of operations and contains a SQL query string.
- In the next step, the user can call "lookat" function to view a sample of the operation result or call one of the MADlib wrapper functions to execute a machine learning algorithm.
- Usually the computation result is small and can be loaded into memory for further processing

Using PivotalR

- 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 in R) until it is actually loaded & saved into the database as a DataFrame(db.data.frame).
- The MADlib wrappers can be applied onto both "db.data.frame" and "db.Rquery" objects.
- https://cran.r-project.org/web/packages/PivotalR/vignettes/pivotalr.pdf



Using PivotalR

- Create a PivoteR DB connection

```
dbc= db.connect(host="192.168.30.2",dbname= "ipass", user="myid",
   password="XXXXX", default.schema= "public, madlib", verbose=T)
Created a connection to database with ID 1
> db.list()

Database Connection Info

## -------------------------------
[Connection ID 1]
Host       :    192.168.30.2
User       :    huangcs
Database   :    ipass
DBMS       :    Greenplum 4
MADlib     :    installed in schema madlib
# List all objects/tables of a connection in
# default schema "jason_schema"
> db.objects("jason_schema.", conn.id = dbc)

[1] "jason_schema.company"          "jason_schema.department"
[3] "jason_schema.department_new"   "jason_schema.diamonds"
[5] "jason_schema.diamonds_lm"      "jason_schema.diamonds_lm_summary"
```

Using PivotalR

- Upload a table to Database

```
>as.db.data.frame(diamonds, "p2_diamonds", dbc, verbose = T)
Counting and caching the data table dimension...0.024 sec...done.
An R object pointing to "p2_diamonds" in connection 2 is created !

The data in the data.frame diamonds is stored into "p2_diamonds" in
database test on 192.168.30.2 !
```

- Link to the table "p2_diamonds" in the database

```
> db_diamonds= db.data.frame("p2_diamonds", dbc, verbose = T)
Counting and caching the data table dimension...1.735 sec...done.
An R object pointing to "p2__diamonds" in connection 2 is created !
```

Using PivotalR

- db_diammonds is just an object pointing to “diamonds” in the database!
- The size is quite small comparing to the real data size.

```
>object.size(db_diamonds)
5448 bytes

#Find the actual size in database
test=# SELECT 'diamonds' as "Table", pg_size_pretty
test-# (pg_total_relation_size('p2_diamonds')) As "Size";
  Table    |  Size
----------+----------
 diamonds | 7200 kB
(1 row)

>dim(db_diamonds) # dimensions
[1] 53940    11
>names(db_diamonds) # column names
[1] "row.names" "carat"     "cut"       "color"     "clarity"
[6] "depth"     "table"     "price"    "x"         "y"         "z"
```

Using PivotalR

```
>summary(db_diamonds) # summary statistics
For all data:

 carat
 column_number        : 1
 data_type            : float8
 row_count            : 53940
 distinct_values      : 273
 missing_values       : 0
 blank_values         : NA
 fraction_missing     : 0
 fraction_blank       : NA
 mean                 : 0.797939747867998
 variance             : 0.224686659822778
 min                  : 0.2
 max                  : 5.01
 first_quartile       : 0.4
 median               : 0.7
 third_quartile       : 1.04
 most_frequent_values: 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3
 mfv_frequencies      : 2604, 2604, 2604, 2604, 2604, 2604, 2604, 2604, 2604, 2604
------------------------------------------------
For all data:

 cut
 column_number        : 2
 data_type            : text
 row_count            : 53940
 distinct_values      : 5
 missing_values       : 0
 blank_values         : 0
..........................
```

Using PivotalR

- Split-Apply-Combine

```
test=# SELECT cut,AVG(price) FROM diamonds GROUP BY cut;
    cut     |        avg
-----------+------------------
 Fair      | 4358.75776397516
 Very Good | 3981.75989074657
 Ideal     |  3457.5419702102
 Good      | 3928.86445169181
 Premium   | 4584.25770429991
(5 rows)
```

- mean_price_by_cut is just an db.Rquery object, no data is loaded in memory. lookat (or a shorthand lk) to execute the operation and load part or all of the results into memory to view.

```
# by is a R command.
> mean_price_by_cut= by(db_diamonds$price, db_diamonds$cut, mean)
> lk(mean_price_by_cut)
        cut price_avg
1      Fair  4358.758
2 Very Good  3981.760
3     Ideal  3457.542
4   Premium  4584.258
5      Good  3928.864

# check the actual content of "mean_price_by_cut"
> content(mean_price_by_cut)
[1] "select \"cut\" as \"cut\", avg(\"price\") as \"price_avg\"
     from \"diamonds\" group by (\"cut\")"
```

Using PivotalR

- 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_varsis a db.Rquery before we call as.db.data.frame()

test=# \dt
                   List of relations
 Schema |      Name       | Type  |   Owner    | Storage
--------+----------------+-------+------------+---------
 public | diamonds        | table | m052040032 | heap
(1 rows)

# as.db.data.frame executes the SQL query and save the result into a new table.
>db_diamonds_few_vars= as.db.data.frame(db_diamonds_few_vars,
                                       "db_diamonds_few_vars", verbose= T)

Counting and caching the data table dimension ... 0.025 sec ... done.
An R object pointing to "db_diamonds_few_vars" in connection 1 is created !

The data created by db_diamonds_few_vars is stored into
"db_diamonds_few_vars" in database test on 192.168.30.2 !

test=# \dt
                      List of relations
 Schema |         Name          | Type  |   Owner    | Storage
--------+-----------------------+-------+------------+---------
 public | db_diamonds_few_vars | table | huangcs    | heap
 public | diamonds              | table | m052040032 | heap
(2 rows)
```

Using PivotalR

```
>lk(db_diamonds_few_vars, 5) # Get the first 5 observations
  carat       cut clarity depth price
1  0.90      Fair     SI2  65.3  2834
2  0.93   Premium     SI2  61.4  3019
3  0.73 Very Good     VS1  62.3  3226
4  0.90 Very Good     VS2  61.4  3445
5  1.00      Good     SI2  63.4  3676
```

- Create 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

[1] "select \"price\" as \"price\", \"carat\" as \"carat\",
                    ln(\"price\") as \"logprice\" from \"diamonds\""

# 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= 5) # Take a look

Counting and caching the data table dimension ... 0.027 sec ... done.
An R object pointing to "db_diamonds_price_clarity" in connection 1 is created !
The data created by db_diamonds_price_clarity is stored into
          "db_diamonds_price_clarity" in database test on 192.168.30.2 !

   price carat logprice
1   2813  0.91 7.942007
2   2968  0.75 7.995644
3   3014  0.78 8.011023
4   3220  0.75 8.077137
5   3442  0.53 8.143808
```

Using PivotalR

- Remove the table and Clean all temporary MADlib tables

```
>delete(db_diamonds_few_vars)
[1] TRUE
>clean.madlib.temp()
```

Linear Models with MADlib

```
>db_diamonds = db.data.frame("diamonds", dbc)
# fitting Gaussian linear model
>price_db_lm = madlib.lm(price ~ carat + depth, data = db_diamonds)
>price_db_lm

MADlib Linear Regression Result

Call:
madlib.lm(formula = price ~ carat + depth, data = db_diamonds)

---------------------------------------

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 4045.333    286.205   14.13   <2e-16 ***
carat       7765.141     14.009  554.28   <2e-16 ***
depth       -102.165      4.635  -22.04   <2e-16 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
R-squared: 0.8506755
Condition Number: 2664.081
```

Linear Models with MADlib

```
# Using "|" to fit models conditioned on the values of variable(s)
>price_db_by_cut_lm= madlib.lm(price ~ carat + depth |cut, data = db_diamonds)
>price_db_by_cut_lm

MADlib Linear Regression Result
Call:
madlib.lm(formula = price ~ carat + depth | cut, data = db_diamonds)
The data is divided into 5 groups
---------------------------------------
Group 1 when
cut: Very Good

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 2507.868    603.324   4.157 3.25e-05 ***
carat       7941.680     29.283 271.200  < 2e-16 ***
depth        -79.752      9.759  -8.172 3.33e-16 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
R-squared: 0.8589421
Condition Number: 2774.731
---------------------------------------
Group 2 when
cut: Ideal

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   704.53     812.14   0.867 0.385684
carat        8194.70      21.85 375.005  < 2e-16 ***
depth         -48.72      13.16  -3.701 0.000215 ***
...............
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
R-squared: 0.8556349
Condition Number: 3243.104
```

k-fold Cross-Validation

```
# Getting 10-fold CV RMSEs
#adding an indexing column to diamonds
>diamonds$row_names = seq(1:nrow(diamonds))
>db_diamonds = as.db.data.frame(diamonds, table.name = "diamonds")

Counting and caching the data table dimension ... 0.026 sec ... done.
An R object pointing to "diamonds" in connection 1 is created !

The data in the data.frame diamonds is stored into "diamonds" in
database ipass on 192.168.30.2 !

# 70% as the training set #This "sample" is a function in PivotalR package
>db_diamonds_train = sample(db_diamonds, 0.7 * nrow(db_diamonds))
# Create testing dataset
# The training dataset is temporarily keeping under
# pg_temp_52.madlib_temp_29c0fd84_0c7b_b1beb8_a80ec2643b71
# we can see this via
#> noquote(db_diamonds_train@.name)
#[1] pg_temp_52
#[2] "madlib_temp_29c0fd84_0c7b_b1beb8_a80ec2643b71"

# We can view them by
# select * from pg_temp_52.madlib_temp_29c0fd84_0c7b_b1beb8_a80ec2643b71
# This command results in a new table "diamonds_test"
>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")

# SQL commands : select * into diamonds_test from diamonds where row_names not in
(select row_names from pg_temp_52."madlib_temp_29c0fd84_0c7b_b1beb8_a80ec2643b71" );
```

k-fold Cross-Validation

```
# R cross-validation commands
# train = function madlib.lm(...)
# predict = predict ?
# metrics = function(pred, d) i.e. RMSE
# data = db_diamonds_train, a db.obj
> generic.cv( function(d) madlib.lm(price ~ carat + depth, data = d),
              predict, function(pred, d) lookat(sqrt(mean((d$price - pred)^2))),
              data = db_diamonds_train, k = 10)

Computation in-database ...
Cutting the data row-wise into 10 pieces ...
Running on fold 1 now ...
Running on fold 2 now ...
Running on fold 3 now ...
Running on fold 4 now ...
Running on fold 5 now ...
Running on fold 6 now ...
Running on fold 7 now ...
Running on fold 8 now ...
Running on fold 9 now ...
Running on fold 10 now ...
Cleaning up ...
Done.
$err
[1] 1546.271
$err.std
[1] 33.95941

# We could redo this with log(price) as our y values.
# Which one is better ? I have no idea.
$err
[1] 0.3966753
$err.std
[1] 0.009013812
```

Array & Matrix Operations

```
# 1000 normal random numbers as example
# Add serial ID/names numbers for easy operations on db.data.frame
# need run this command that table can put into DB

data <- data.frame(matrix(rnorm(1000,0,1), ncol = 5))
row_names <- as.data.frame(seq(1:nrow(data)))
names(row_names)="row_names"
data=cbind(row_names,data)
db_nrand_1000 <- as.db.data.frame(data,"db_nrand_1000")

>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")
The data created by cbind(db_nrand_1000[, 1], db.array(db_nrand_1000[,
2:6])) is stored into "db_nrand_1000_ary" in database ipass on
192.168.30.2 !

>lk(db_nrand_1000_ary, array = F, nrows= 5)
  row_names                                       agg_opr
1        88     {-0.0643, 1.6161, 0.7727, 1.4015, 1.1956}
2       154     {-1.7860, 0.0760, 1.4270, 1.8382,-0.7072}
3       112     {-0.0153, 0.8870, 1.4937,-0.6377,-0.8657}
4        77      {0.0533, 1.2428,-1.0834,-0.6153,-1.6791}
5       132      {1.3993, 0.8443,-1.0956,-0.3216, 0.7580}

# 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)
> stat_ary
  row_names  array_sum array_mean array_stddev
1        88  4.9217065  0.9843413    0.6637922
2       154  0.8482468  0.1696494    1.4973177
3       112  0.8620132  0.1724026    1.0032788
4        77 -2.0814534 -0.4162907    1.1239833
5       132  1.5844228  0.3168846    1.0059264
```

Array & Matrix Operations

```
# 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= 5)

  row_names     norm
1        31 1.394059
2        63 1.655406
3        95 2.903345
4       191 1.663572
5       127 2.841238
```

An Example of Loading Big Data Using PivotalR

```
library(PivotalR); library(DBI);library(data.table)
#Link and create vitrual director to /home/stock
#ln -s /home/stock ./stock

#set "path" string
path <- setwd("/home/stock/mrk_re/2008")
#list all csv files in path directory
all.excel <- list.files(path=path,pattern = "*.csv")

#fread read the first file in all.excel
data <- fread(all.excel[1],header =T,sep=",")

# Build connection
dbc = db.connect(host="192.168.30.2", user = "huangcs",
                 password = "XXXXXX", dbname = "test")
# Putting data to PostgreDB with table name stock_try
as.db.data.frame(data,"stock_try")
```

An Example of Loading Big Data Using PivotalR

```
#SQL: copy stock_try FROM '/home/stock/mrk_re/2008/all.excel[i]
#   ' DELIMITER ',' csv header

for(i in 2:length(all.excel)){
  db.q(paste0("copy stock_try FROM '/home/stock/mrk_re/2008/",
               all.excel[i],"' DELIMITER ',' csv header"))
}


# Now stock_try is the full data are ready for use
# Create db_stock_try, a db.data.frame wrapper object
db_stock_try <- db.data.frame("stock_try",dbc,verbose = T)
# SELECT 'stock_try' as "Table" ,
# pg_size_pretty(pg_total_relation_size('stock_try')) As "Size";
>db.q("select pg_size_pretty(pg_relation_size('stock_try'))"
       ,conn.id=dbc)

select pg_size_pretty(pg_relation_size('stock_try'))
  pg_size_pretty
1           65 GB

>dim(db_stock_try)
[1] 123344822       127
```

An Example of Loading Big Data Using PivotalR

```
#extract two columns, "Ask_Price1","Ask_Size1"
>db_stock_t<- db_stock_try[,c("Ask_Price1","Ask_Size1")]
>as.db.data.frame(db_stock_t,"stock_t")

The data created by db_stock_t is stored into "stock_t" in
database test on 192.168.30.2 !

Table          :     "stock_t"
Database       :     test
Host           :     192.168.30.2
Connection     :     1

>db.q("select pg_size_pretty(pg_relation_size('stock_t'))",
       conn.id=dbc)
select pg_size_pretty(pg_relation_size('stock_t'))

  pg_size_pretty
1         4241 MB
```

An Example of Loading Big Data Using PivotalR

```
# example 1 #
# clock 1
timestart1 <- Sys.time()
# List Descendingly the first 10 Ask_Price1 and the average of
# Ask_Size1 of the same Ask_Price1
db.q("select \"Ask_Price1\", avg(\"Ask_Size1\")from stock_try group
      by \"Ask_Price1\"order by \"Ask_Price1\" DESC limit 10",conn.
timeend1 <- Sys.time()
# clock 2
runningtime1 <- timeend1-timestart1
print(runningtime1)

require(dplyr)
# Extract two columns and load them to R-studio
# This takes a while, since it is actally loaded to local memory
A=db.q("select \"Ask_Price1\",\"Ask_Size1\" from stock_try ",
        nrows = "all",conn.id=dbc)
#object.size(A): about 986 M
timestart2 <- Sys.time()
B=A%>%group_by(Ask_Price1)%>%summarise(Ask_Size_mean=
  mean(Ask_Size1))%>%arrange(desc(Ask_Price1))
timeend2 <- Sys.time()
runningtime2 <- timeend2-timestart2
print(runningtime2)
```

An Example of Loading Big Data Using PivotalR

```
# example 2 #
timestart3 <- Sys.time()
db.q("select \"Order_ID\",avg(\"Ask_Size1\") from stock_try group
      by \"Order_ID\" order by \"Order_ID\" ", conn.id=dbc)
timeend3 <- Sys.time()
runningtime3 <- timeend3-timestart3
print(runningtime3)


A=db.q("select \"Order_ID\",\"Ask_Size1\" from stock_try ",
        nrows = "all", conn.id=dbc)
timeend4 <- Sys.time()
runningtime4 <- timeend4-timestart4
print(runningtime4)
require(dplyr)

timestart4 <- Sys.time()
B=A%>%group_by(Order_ID) %>% summarise(Ask_Size_mean=
  mean(Ask_Size1))%>%arrange(Order_ID)
object.size(B)
timeend4 <- Sys.time()
runningtime4 <- timeend4-timestart4
print(runningtime4)
```

Homeworks:

- 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 Hitters.

Goodbye Postgre

I'll be back.

"We" will work on SparklyR.