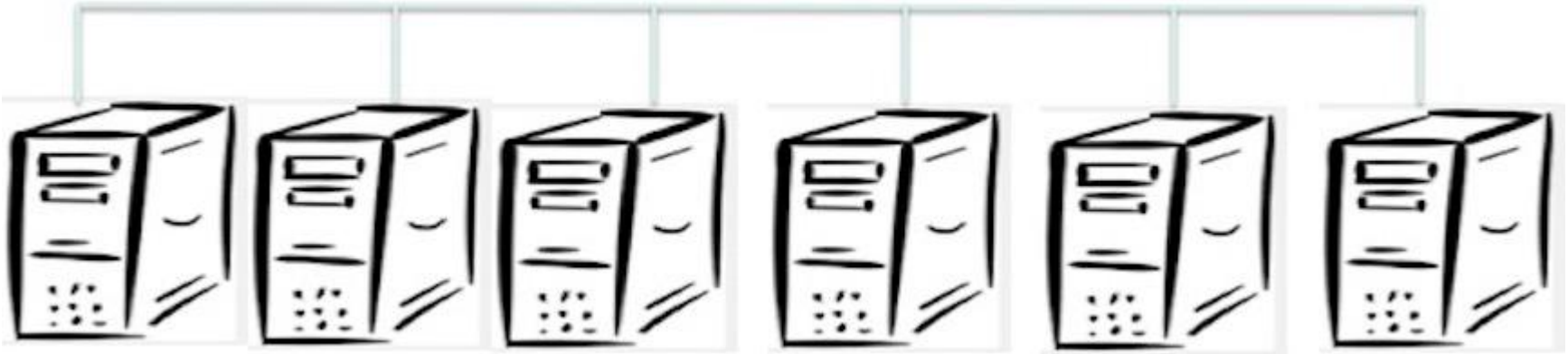
Greenplum

Petabyte Scale Data Warehousing

Introduction

Little History

Massively Parallel Processing



Greenplum Architecture

Master Servers

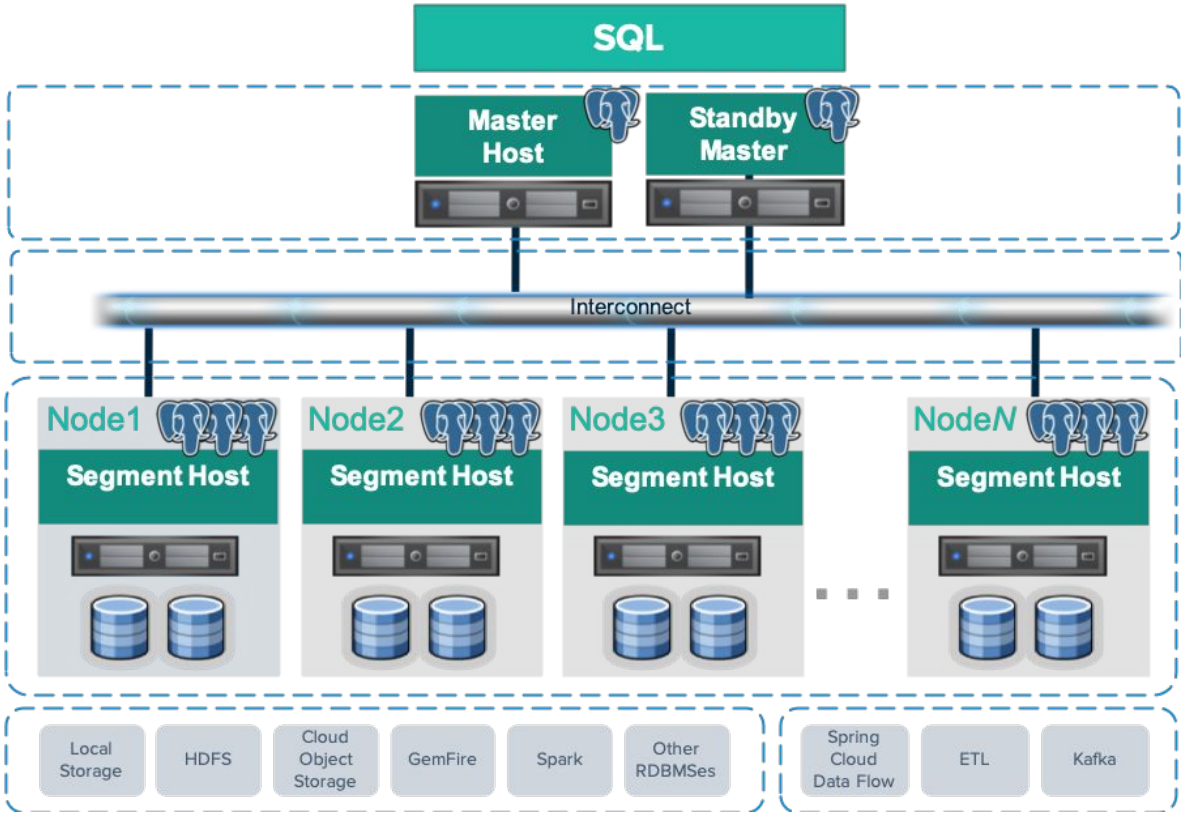
Query planning and dispatch

Interconnect

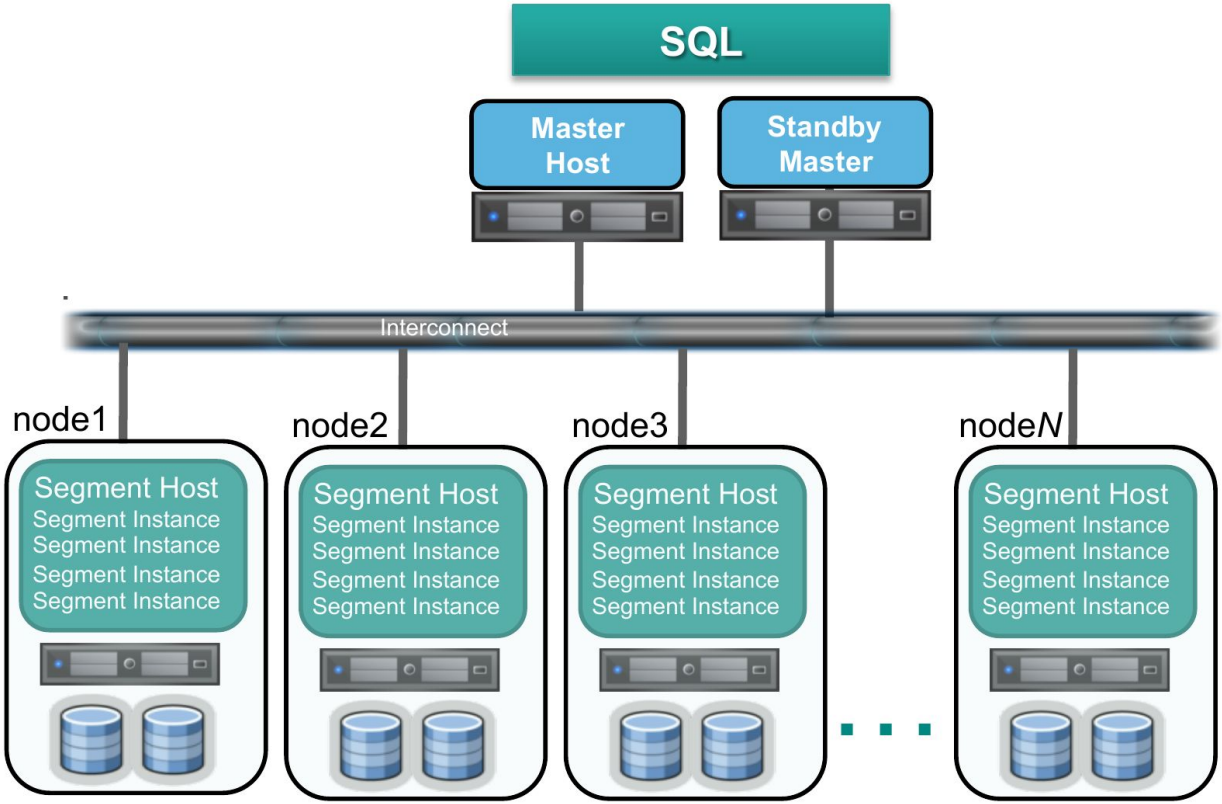
Segment Servers

Query processing and data storage

External Sources and Pipelines

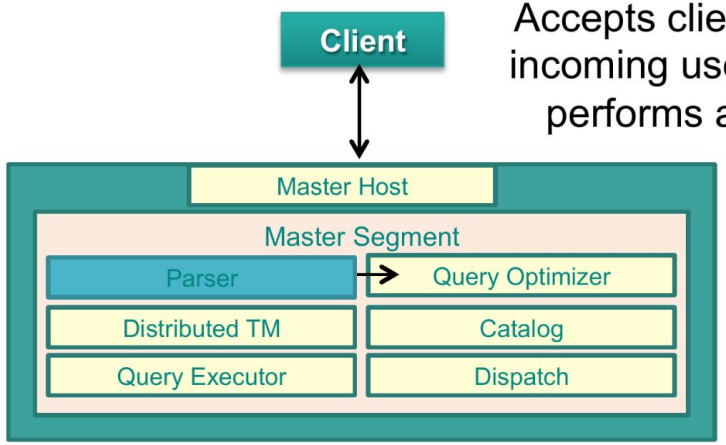


Greenplum Architecture



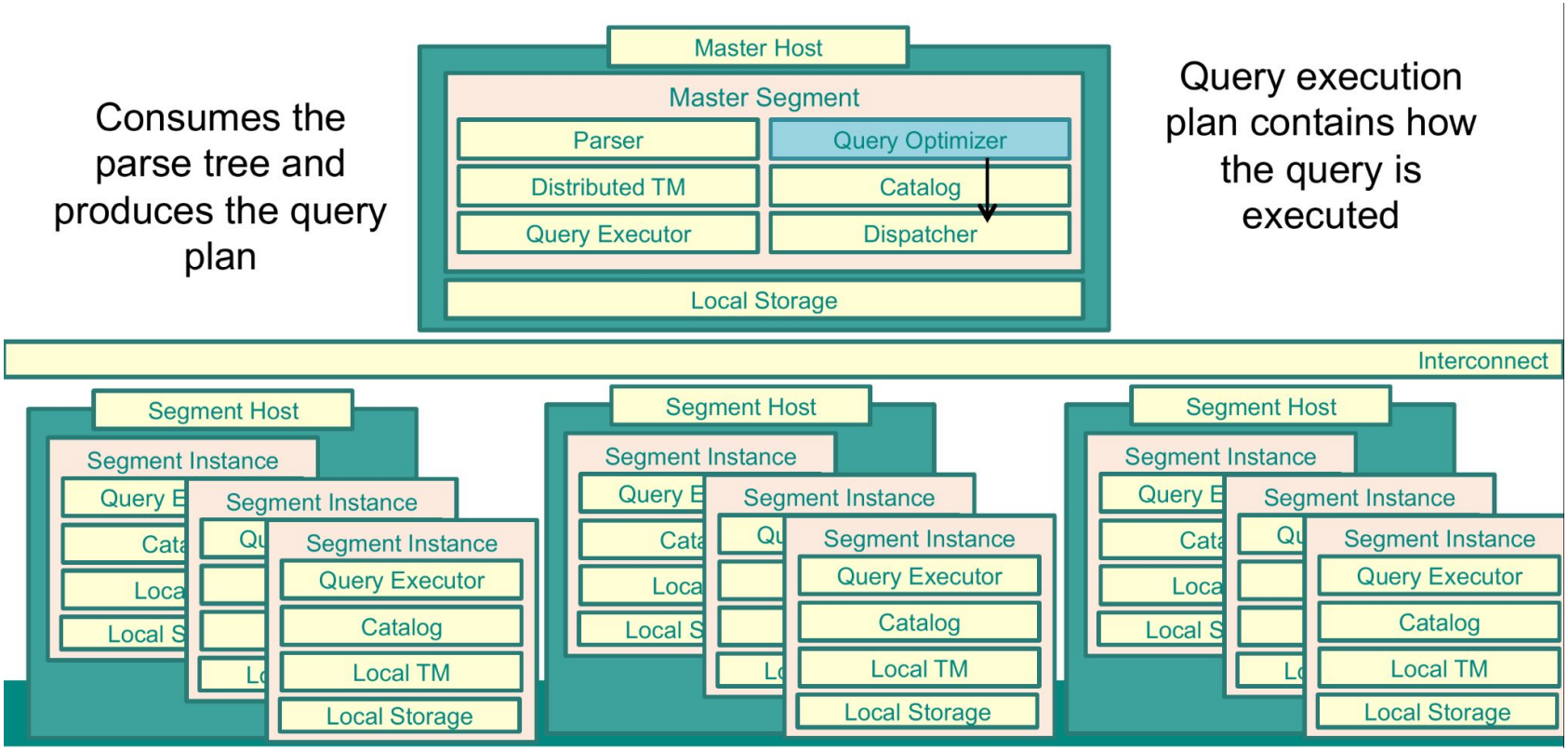
Master Host

Parser enforces syntax, semantics and produces a parse tree

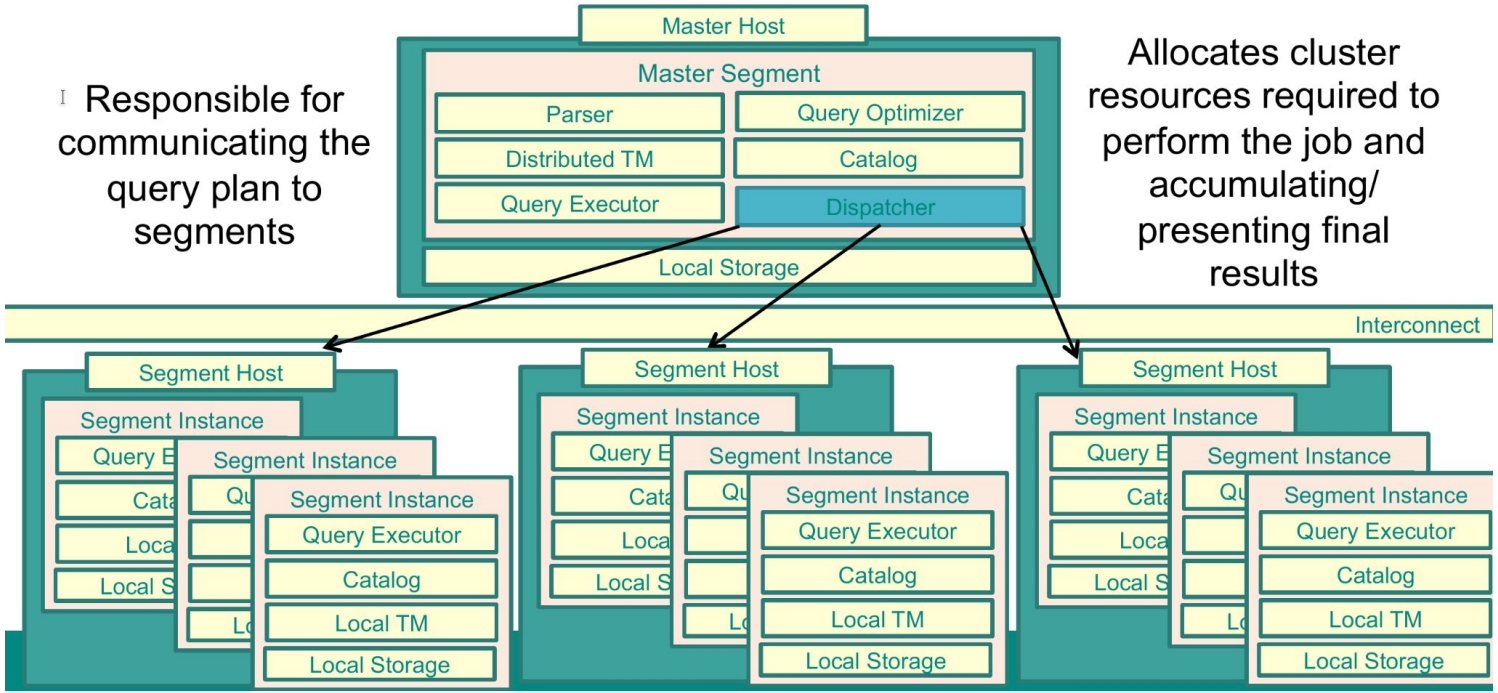


Accepts client connections, incoming user requests and performs authentication

Pivotal Query Optimizer

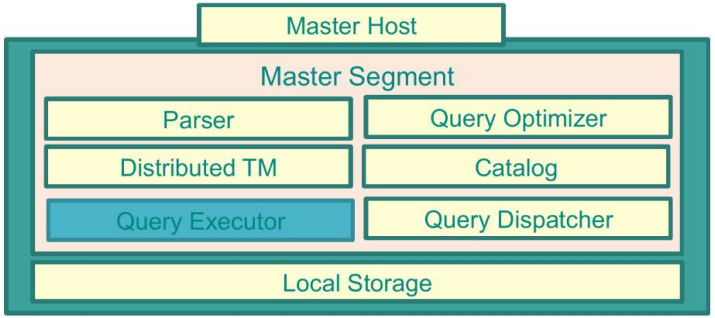


Query Dispatcher

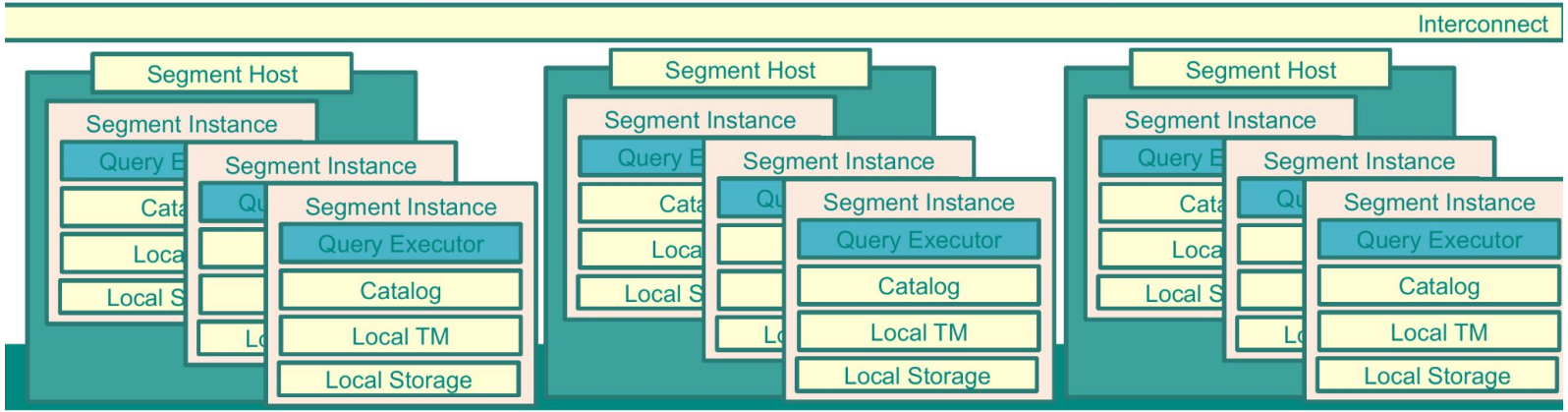


Query Executor

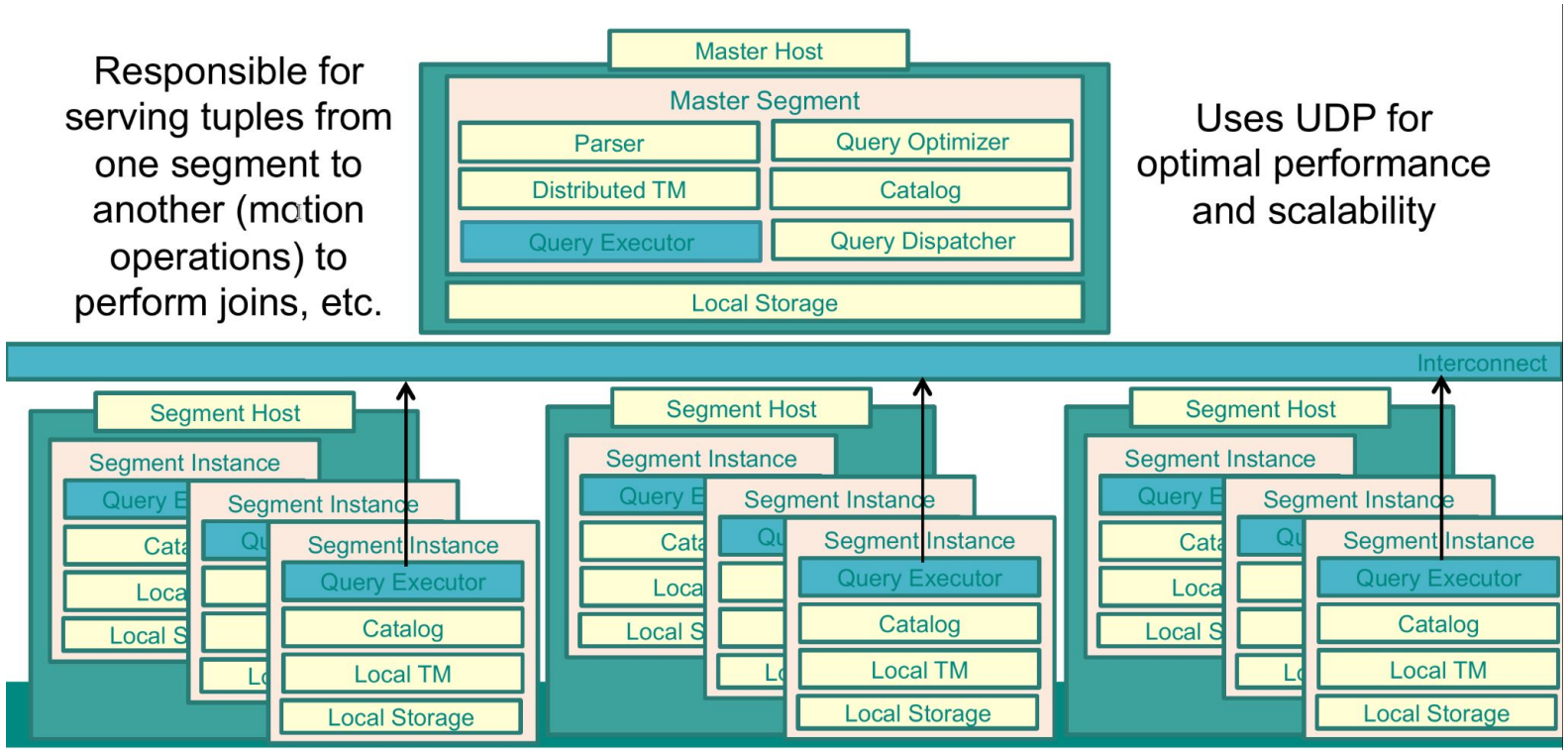
Responsible for executing the steps in the plan (e.g. open file, iterate over tuples)



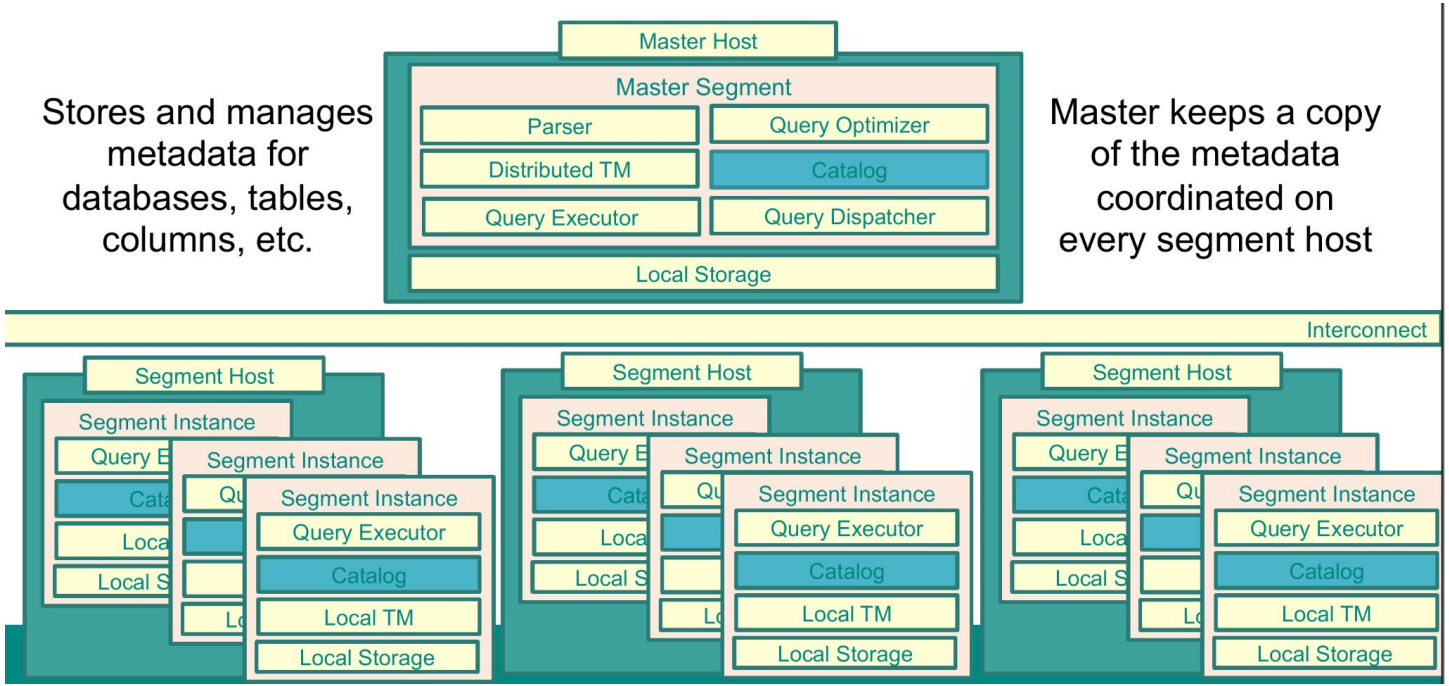
Communicates its intermediate results to other executor processes



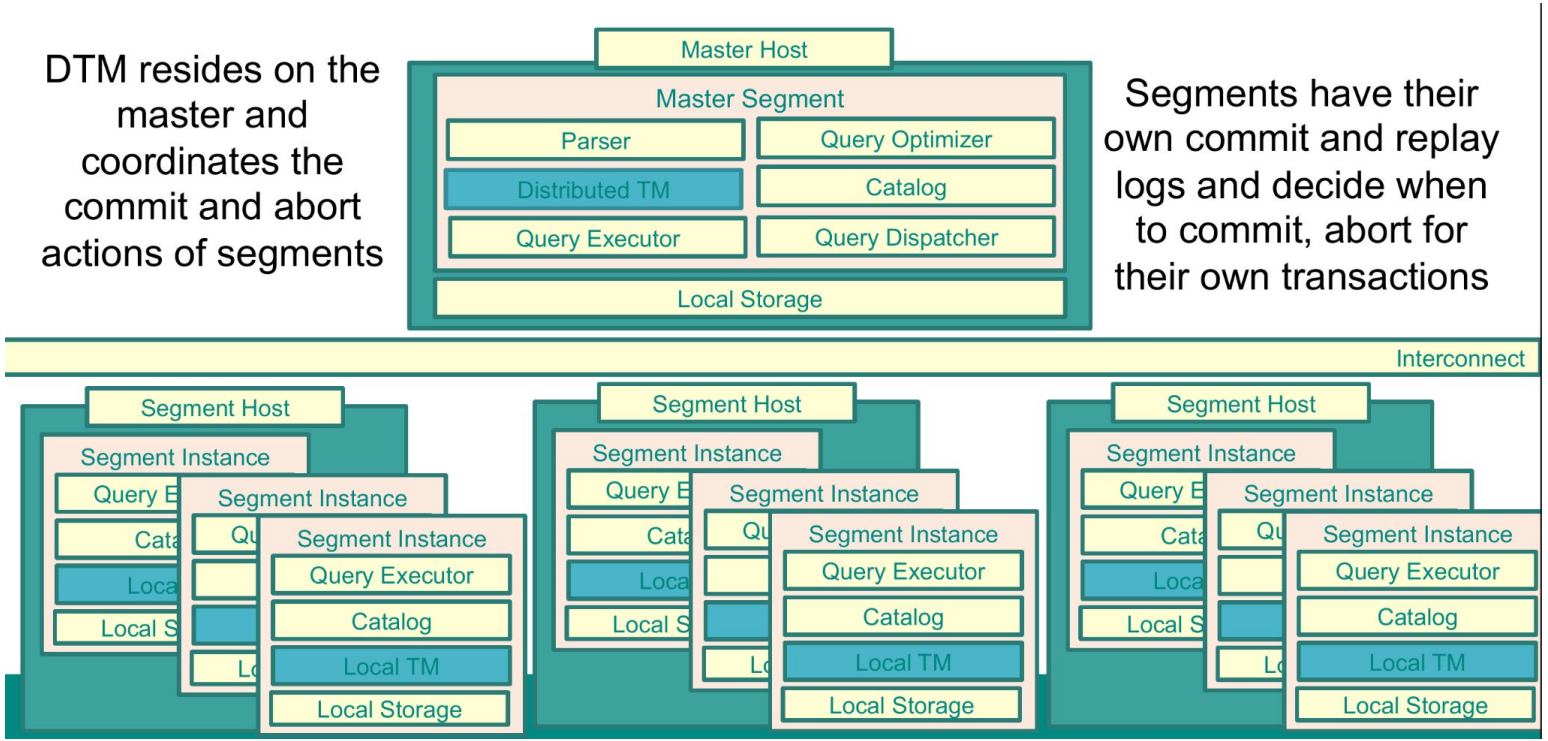
Interconnect



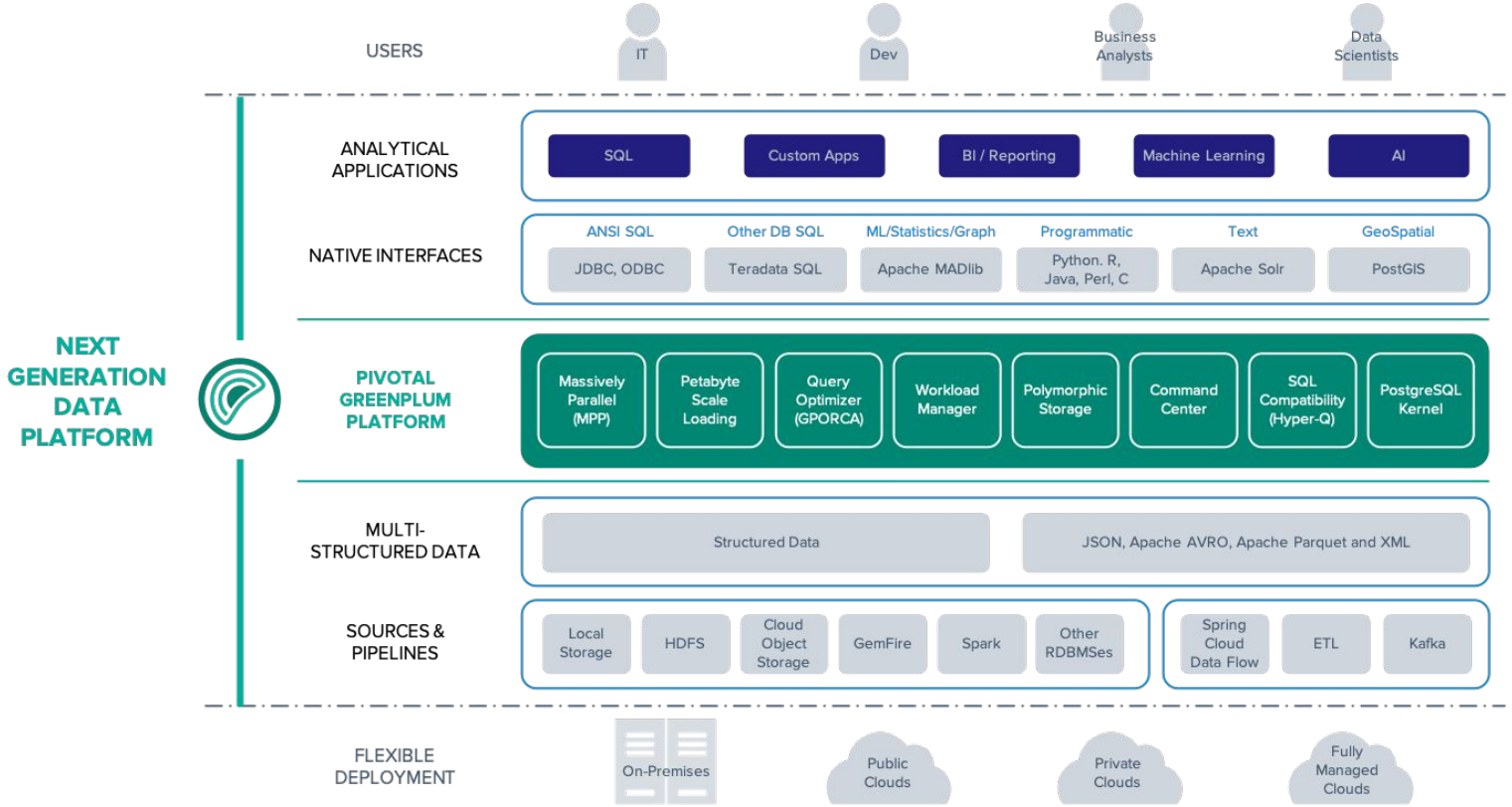
System Catalog



Distributed Transaction Management



Greenplum Environment



Greenplum Deployment

Infrastructure-Agnostic

Bare-Metal

Private Cloud

vmware vSphere

openstack.

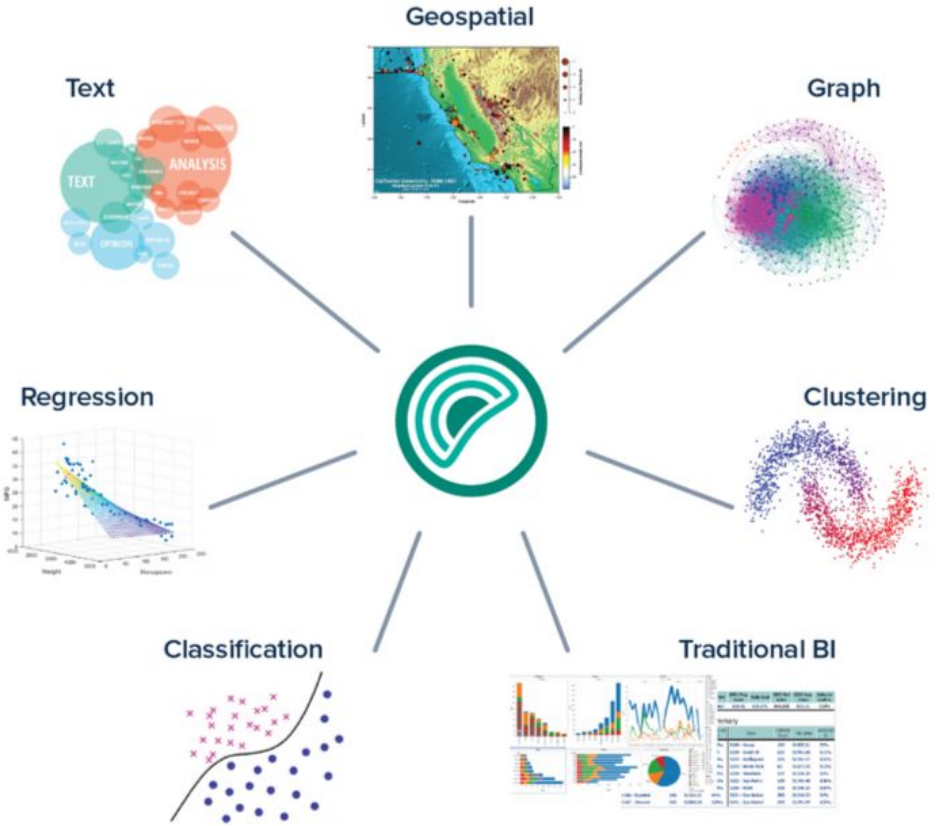
Public Cloud

amazon web services

Microsoft Azure

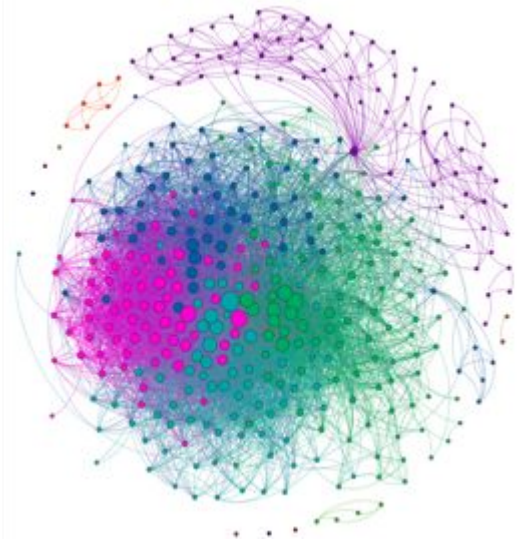
Google Cloud Platform

Greenplum - Integrated Analytics



Greenplum Graph - Integrated Analytics

- Designed for very large graphs (billions of vertices/edges
- No need to move and transform data for an external graph engine
- Support for most popular graph algorithms
- Familiar SQL interface



```
SELECT madlib.pagerank (
   'vertex',              -- Vertex table
   'id',                   -- Vertix id column
   'edge'                 -- Edge table
   'src=src, dest=dest', -- Edge arguments
   'pagerank out',        -- Output table of PageRank
   NULL,                  -- Default damping factor
(0.85)
   NULL,                  -- Default max iters (100)
   0.00000001,            -- Threshold
   'user_id'              -- Grouping column name
 );
```

Vertex Table

Vertex	Vertex Params	...
0	...	
1	...	
2	...	
3	...	

Edge Table

Source Vertex	Dest Vertex	Edge Weight	Edge Params	...
0	3	1.0	...	
1	0	5.0	...	
1	2	3.0	...	
2	3	8.0	...	
3	0	3.0	...	
3	1	2.0	...	

Greenplum Text - Integrated Analytics

Use Cases

- Communications compliance and monitoring
- Customer sentiment analysis
- Document search and query
- Social media processing, etc

GPText

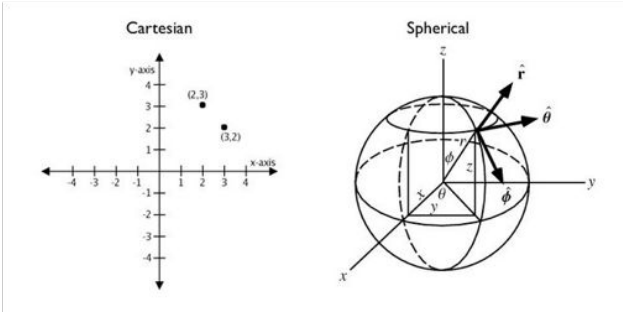
- Leverages Apache Solr and Greenplum
- Python and Java integration for Natural Language Processing (NLP)
- Apache MADlib integration for machine learning on textual data

Greenplum Geospatial - Integrated Analytics

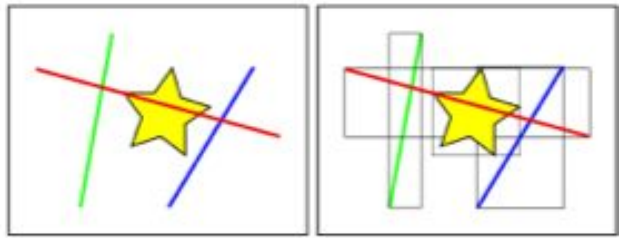
- Points, Lines, Polygons, Perimeter, Area
- Intersection, Contains, Distance, Longitude/Latitude

PostGIS
Geospatial Objects for PostgreSQL

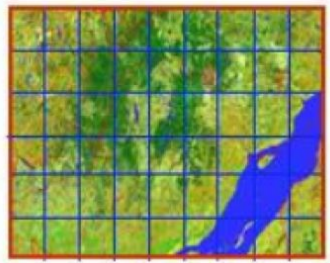
Round earth calculations



Spatial indexes &
bounding boxes

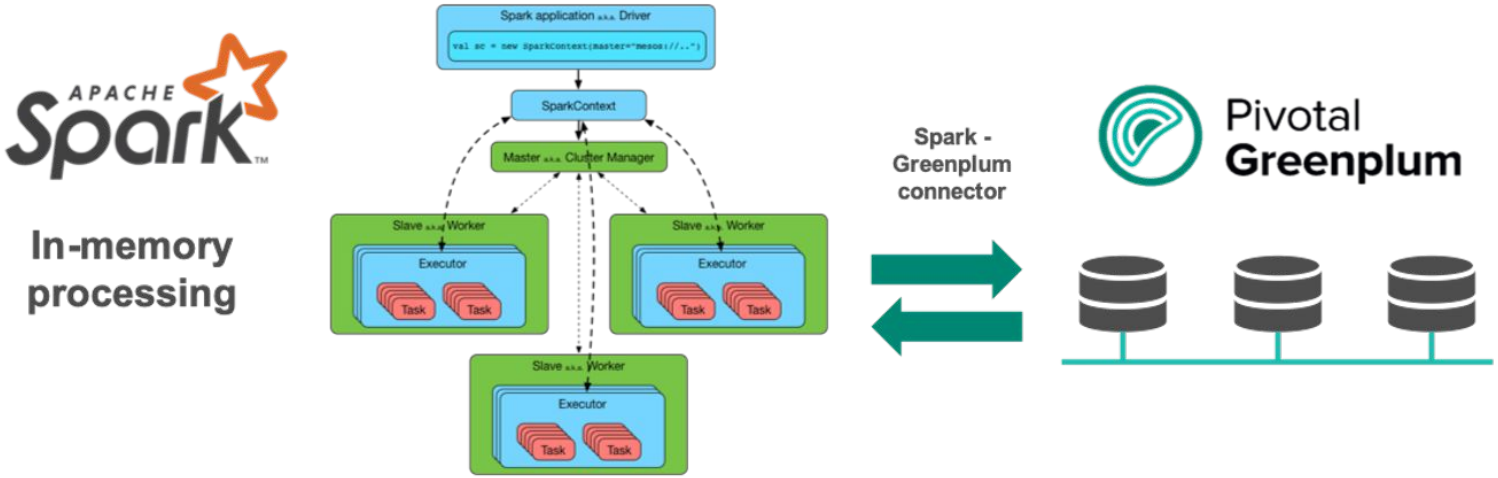


Raster support



Python & R Libraries - Integrated Analytics

Spark - Integrated Analytics



- Provide Data Access to Greenplum Data
- Leverage SPARK Skill Set of Data Scientists
- Leverage off-cluster compute resources to do computations
- Push result sets back into Greenplum for storage

Language Agnostic

Interfaces

- User Defined Types
- User Defined Functions
- User Defined Aggregates

PL/Container - Python/R Containerization

- Deploy Custom R & Python Developer Environment(s) To Cluster
- Execute Functions in Isolated Secure Containers
- Deploy code and functions as non super-user
- Package any custom Python and R modules in the deployment
- Pre-configured for Data Science or customized images by users
- Multiple developer environments on same cluster

Greenplum - SQL Containerization

- Resource isolation for multi-tenancy and mixed analytical SQL workloads
- Enhances stability and manageability
- Leverages Linux Cgroups

Open Source Agility

- 100% based on PostgreSQL innovation
  - the most advanced analytical database for PostgreSQL users
- Agile development
- A new Pivotal Greenplum release every month

PostgreSQL

Data Distribution

CREATE TABLE Definition

- One of the most important aspects of GP!
- Every table has a distribution method
- `DISTRIBUTED BY (column)`
  - Uses a hash distribution
- `DISTRIBUTED RANDOMLY`
  - Uses a random distribution which is not guaranteed to provide a perfectly even distribution
- Explicitly define a column or random distribution for all tables
  - Do not use the default

DISTRIBUTED BY (column_name)

```
CREATE TABLE foo (
     id integer,
     size float8)
DISTRIBUTED BY (id);
```

- Use a single column that will distribute data across all segments evenly
- For large tables significant performance gains can be obtained with local joins (co-located joins)
    - For tables commonly joined together, distribute on the same column
- Co-located join is performed within the segment
    - Segment operates independently of other segments
- Co-located join eliminates or minimizes motion operations
    - Broadcast motion or Redistribute motion

DISTRIBUTED BY (column_name)

Things to avoid

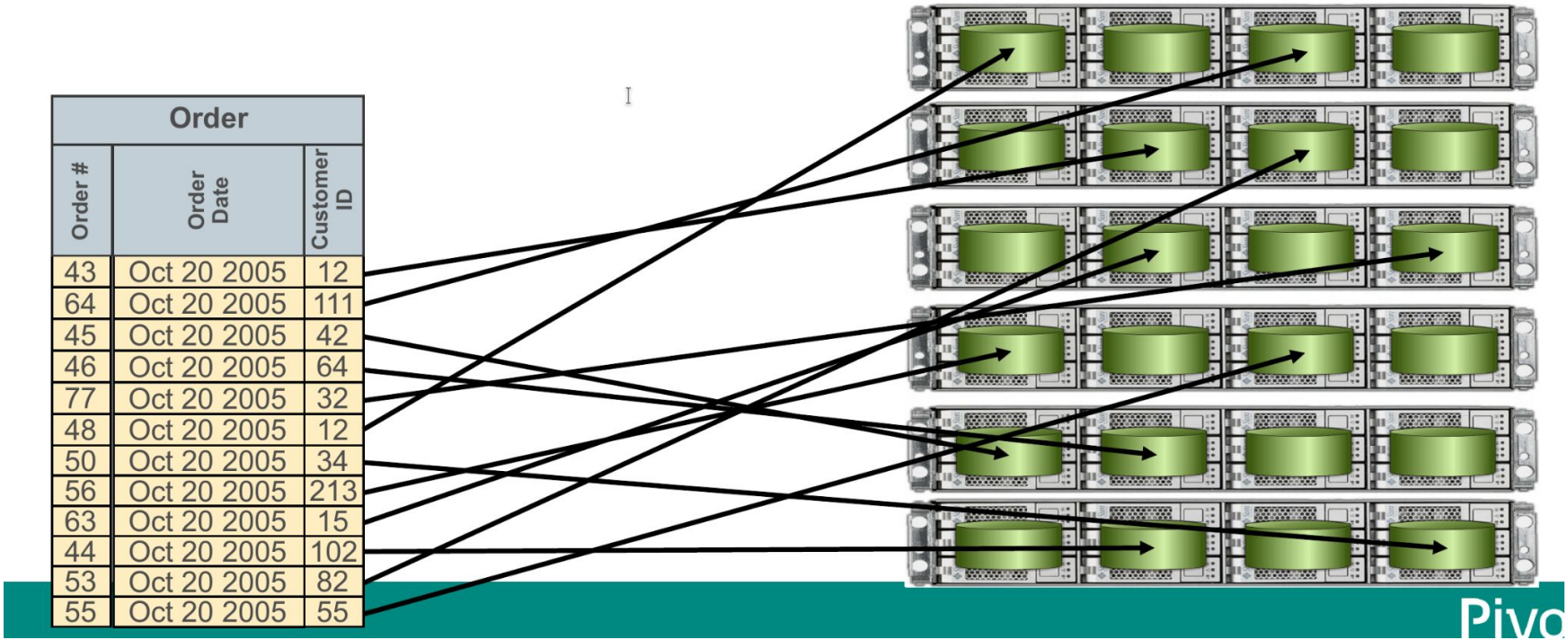
- Do not distribute on columns that will be used in the WHERE clause of a query
- Do not distribute on DATE or TIMESTAMP
- Never distribute and partition a table on the same column

DISTRIBUTED RANDOMLY

```
CREATE TABLE foo (
     id integer,
     size float8)
DISTRIBUTED RANDOMLY);
```

- Uses a random algorithm
  - Distributes data across all segments
  - Minimal data skew but not guaranteed to have a perfectly even distribution
- Any query that joins to a randomly distributed table will require a motion operation
  - Broadcast motion or
  - Redistribute motion

Data Distribution: The Key to Parallelism



Order		
Order #	Order Date	Customer ID
43	Oct 20 2005	12
64	Oct 20 2005	111
45	Oct 20 2005	42
46	Oct 20 2005	64
77	Oct 20 2005	32
48	Oct 20 2005	12
50	Oct 20 2005	34
56	Oct 20 2005	213
63	Oct 20 2005	15
44	Oct 20 2005	102
53	Oct 20 2005	82
55	Oct 20 2005	55

ALTERing Distribution

- Distribution is done at table creation time but can be altered.
  - `ALTER TABLE sales SET DISTRIBUTED BY (customer_id);`
- When you change the hash distribution of a table, data is automatically redistributed.
- Changing the distribution policy to RANDOM does not cause the data to be redistributed:
  - `ALTER TABLE sales SET DISTRIBUTED RANDOMLY;`

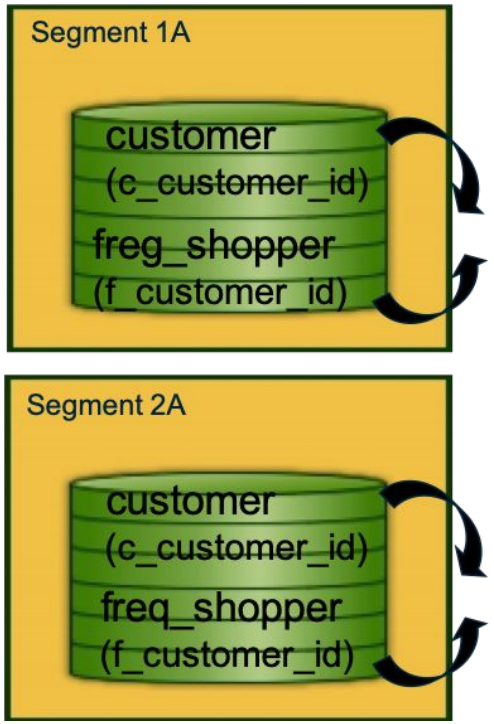
Best Practice: Create a new table with the new distribution and then swap

```
CREATE TABLE new_foo as select * from foo
distributed randomly;
DROP TABLE foo;
ALTER TABLE new_foo RENAME to foo;
```

Data and Computational Skew

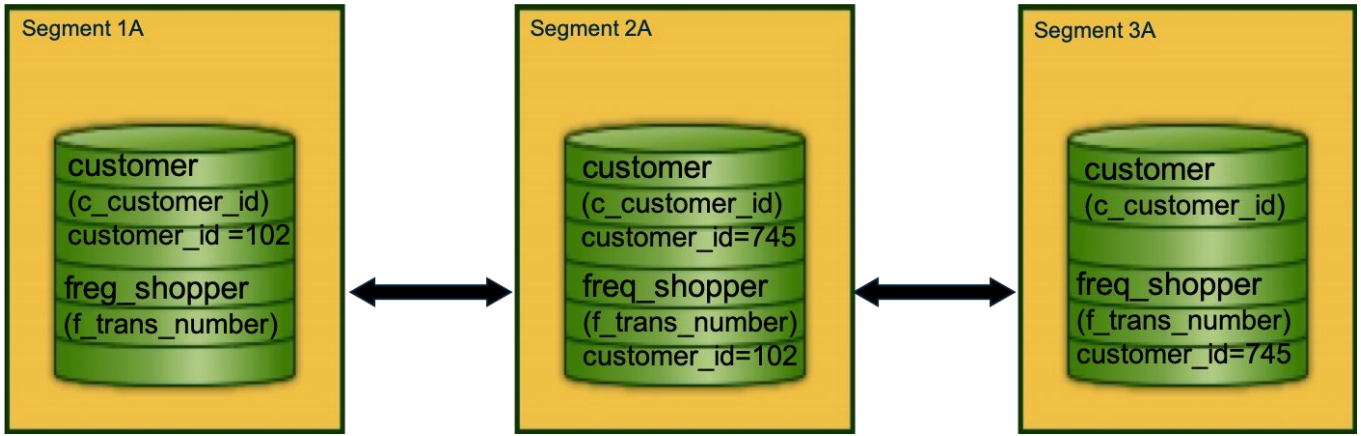
- Select a distribution key with **unique values and high cardinality** that will not result in data skew
  - Do not distribute on **boolean keys and keys with low cardinality**
    - The system distributes rows with the **same hash value** to the **same segment instance** therefore resulting in the **data being located on only a few segments**
- Select a distribution key that will not result in **computational skew** (in flight when a query is executing)
  - Operations on columns that have **low cardinality** or **non-uniform distribution**

Co-Located Joins



Distribute on the same key used in the join to obtain **local joins**

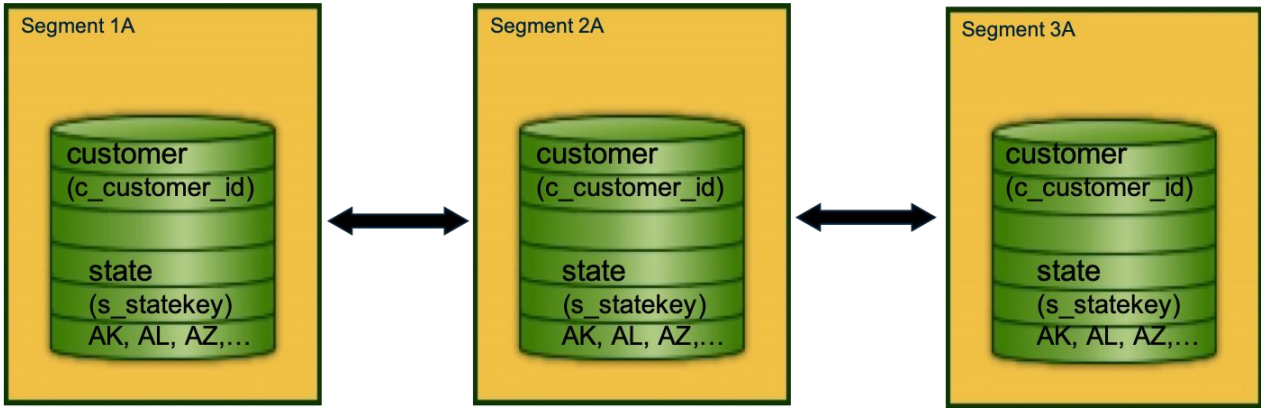
Redistribution Motion




```
WHERE customer.c_customer_id = freq_shopper.f_customer_id
```

freq_shopper table is dynamically redistributed on f_customer_id

Broadcast Motion




```
WHERE customer.c_statekey = freq_shopper.f_statekey
```

state table is dynamically broadcast to all segments

Join Keys - Data Types Matter

```
customer (c_customer_id)          745::int
freq_shopper (f_customer_id)      745::varchar(10)
```

- Values might appear the same, however they are
  - stored differently
  - hashed differently
- Resulting in:
  - like rows being stored on different segments
  - a distribution motion before the tables can be joined

Best Practice: For distribution (join) keys, use the same data type. This is important particularly for large tables that are joined.

Checking Data Skey

```
SELECT COUNT(*), gp_segment_id FROM <table-name>
GROUP BY gp_segment_id;
```

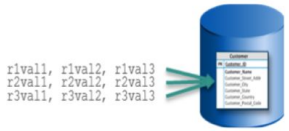
```
SELECT '<table-name>' as "Table Name",
       max(c) as "Max Seg Rows",
       min(c) as "Min Seg Rows",
       (max(c) - min(c))*100.0/max(c)
          as "% Difference Between Max & Min"
FROM (SELECT count(*) c, gp_segment_id
      FROM <table-name> GROUP BY 2) as a;
```

Best Practice:
Check for data skew after initial load and subsequent loads

Example

Data Loading

Traditional and Greenplum Specific



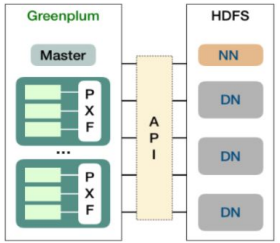
SQL INSERT

SQL COPY

External Tables

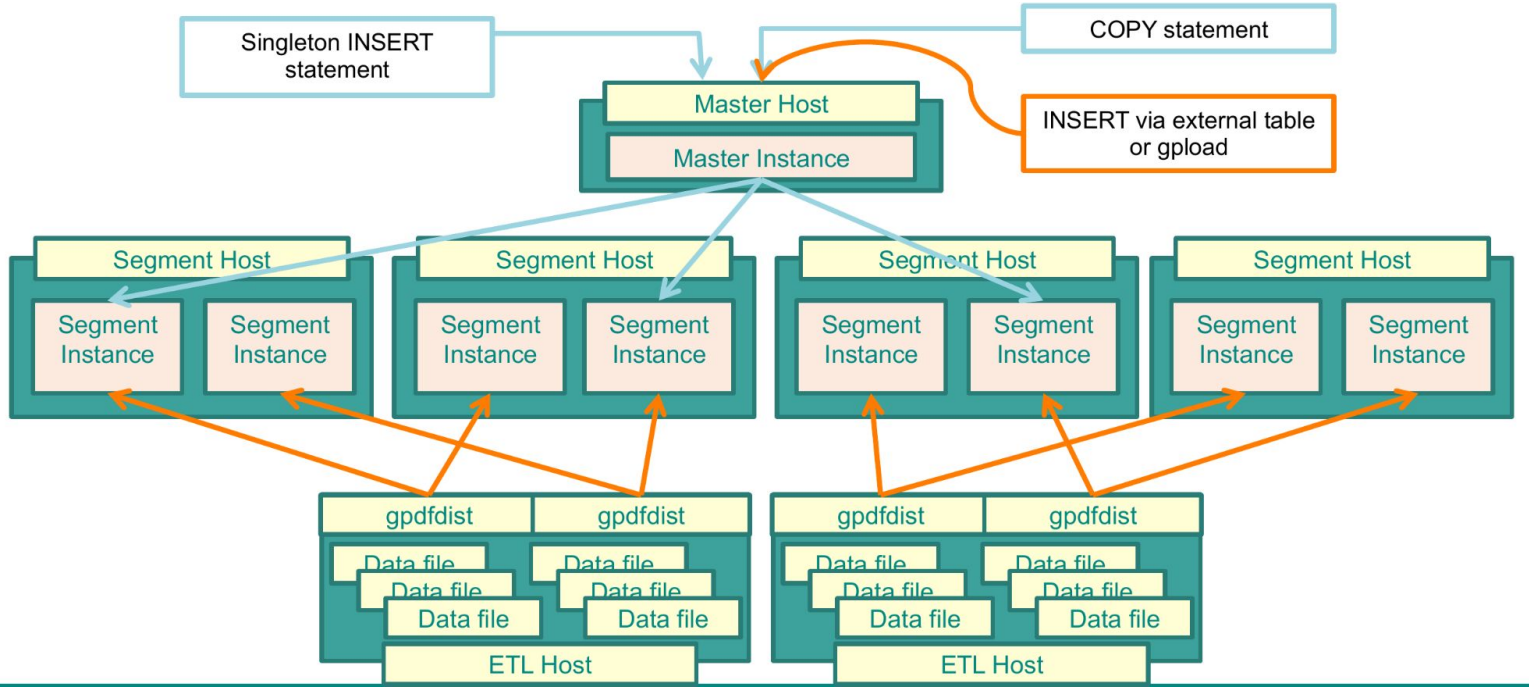
gpfdist/gpload

gphdfs



PXF

Load Architectures



COPY Command

SQL COPY command

- Is a PostgreSQL command
- Loads all rows in one command and is not parallel
- Loads data from a file or from standard input
- Support error handling similar to external tables

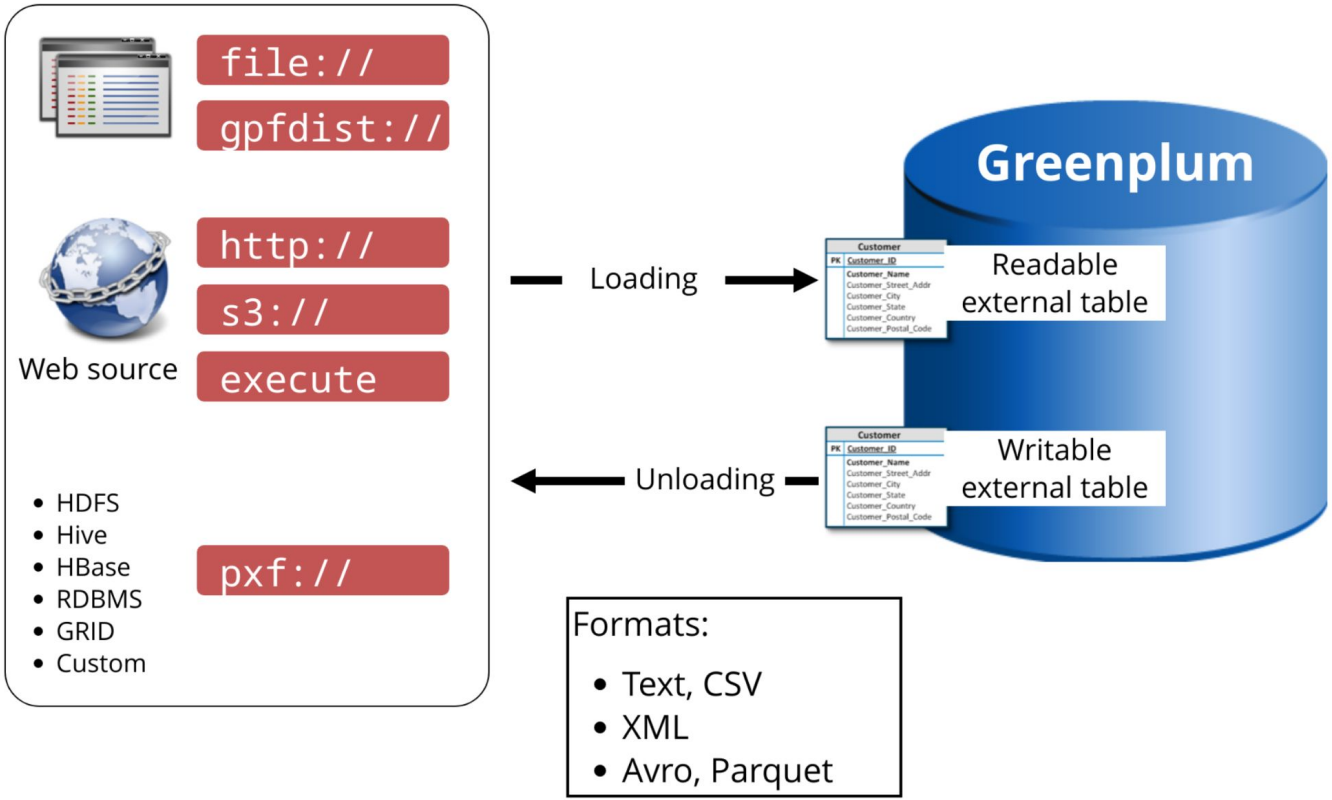
```
COPY mytable FROM '/data/myfile.csv'
WITH CSV HEADER;
```

Loading from External Tables

Read-only External Tables:

- Leverage parallel processing power of segments
- Can be accessed with SELECT statements
- Access data outside the Greenplum database
- Commonly used for ETL/ELT and data loading

External Table Types



File-based External Tables

When creating file-based external tables:

- Specify up to as many URIs as you have segments in the LOCATION clause
- Each URI points to an external data file or data source
- URIs do not need to exist prior to defining the external table
- The URI must exist when the data is queried

File-based External Tables

```
CREATE EXTERNAL TABLE ext_expenses (name text, date date, amount float4, category text,
description text)

LOCATION (

'file://seghost1/dbfast/external/expenses1.csv',

'file://seghost1/dbfast/external/expenses2.csv',

'file://seghost2/dbfast/external/expenses3.csv',

'file://seghost2/dbfast/external/expenses4.csv',

'file://seghost3/dbfast/external/expenses5.csv',

'file://seghost3/dbfast/external/expenses6.csv',

)

FORMAT 'CSV' ( HEADER );
```

Protocol can be file, gpfdist, gpfdists, or gphdfs

You can define as many URIs as you have segments

Format can be CSV, TEXT, XML, or custom

gpfdist and gpload - Parallel File Distribution Program

gpfdist

- Is a C program that uses HTTP
- Can be run on an external server
- Distributes data at 200 MB/s per gpfdist
- Provides full parallelism for best performance

gpload

- Interfaces with and invokes gpfdist
- Creates an external table definition
- Executes INSERT, UPDATE, or MERGE to load data
- Uses YAML config file

gpfdist example - Parallel File Distribution Program

```
gpfdist -d /var/load_files/expenses1 -p 8081 >> gpfdist.log 2>&1 &

gpfdist -d /var/load_files/expenses2 -p 8082 >> gpfdist.log 2>&1 &
```

```
CREATE EXTERNAL TABLE ext_expenses

   (name text, date date,  amount float4, description text)

   LOCATION (

        'gpfdist://etlhost:8081/*',

        'gpfdist://etlhost:8082/*')

   FORMAT 'TEXT' (DELIMITER '|')

   ENCODING 'UTF-8'

   LOG ERRORS

   SEGMENT REJECT LIMIT 10000 ROWS ;
```

```
INSERT INTO expenses (SELECT * FROM ext_expenses);
```

Gpload - Control File Example

```
VERSION: 1.0.0.1
DATABASE: faa2
USER: gpadmin
HOST: smdw
PORT: 5432
GPLOAD:
  INPUT:
    - SOURCE:
        LOCAL_HOSTNAME:
            - mdw
        PORT: 8081
        FILE:
            - /rawdata/FAAData/On_Time_On_Time_Performance_*.csv
    - FORMAT: csv
    - DELIMITER: ','
    - ESCAPE: '"'
    - QUOTE: '"'
    - HEADER: true
    - ERROR_LIMIT: 100
  OUTPUT:
    - TABLE: faadata.factontimeperformance
    - MODE: insert
  PRELOAD:
    - TRUNCATE: true
    - REUSE_TABLES: false
  SQL:
    - BEFORE: "INSERT INTO audit VALUES('start', current_timestamp)"
    - AFTER: "INSERT INTO audit VALUES('end', current_timestamp)"
```

Greenplum connection information

gpfdist config info

Pre-existing table, data entry mode

Pre and post SQL

gpload Invocation Syntax

```
gpload -f control_file
    [-l <log file>]
    [-h <hostname>]  [-p <port>]
    [-U <username>]  [-d <database>]
    [-W] [-v | -V] [-q] [-D]

gpload -? | --version
```

```
$ gpload -f load_faadata.yaml
2012-01-17 09:05:29|INFO|gpload session started 2012-01-17
09:05:29
2012-01-17 09:05:29|INFO|started gpfdist -p 8081 -P 8082 -f
"/rawdata/FAADOn_Time_Performance_*.csv" -t 30
2012-01-17 09:11:23|INFO|running time: 353.36 seconds
2012-01-17 09:11:23|INFO|rows Inserted          = 20860045
2012-01-17 09:11:23|INFO|rows Updated           = 0
2012-01-17 09:11:23|INFO|data formatting errors = 0
2012-01-17 09:11:23|INFO|gpload succeeded
```

External Web Table

Loading from a set of URLs

```
CREATE EXTERNAL WEB TABLE ext_expenses (name text, exp_date date,
                                        amount float4, desc text)
LOCATION ( 'http://xyz.company.com/expenses/sales/expenses.csv',
           'http://xyz.company.com/expenses/finance/expenses.csv'
           'http://xyz.company.com/expenses/ops/expenses.csv')
FORMAT 'CSV' ( HEADER );
```

Loading from a script

```
CREATE EXTERNAL WEB TABLE log_output (linenum int,
       message text)
EXECUTE '/var/load_scripts/get_log_data.sh'
ON HOST FORMAT 'TEXT' (DELIMITER '|');
```

Loading from a command

```
CREATE EXTERNAL WEB TABLE du_space (storage text)
EXECUTE 'df -k' ON ALL FORMAT 'TEXT';
```

External Table Error Handling

- Incorrectly formatted rows are rejected:
  - Missing or extra attributes
  - Columns of the wrong data type
  - Invalid client encoding sequence
- CONSTRAINT errors (NOT NULL, CHECK, UNIQUE) are handled as "all or nothing" - not single row isolation
- Format of error handling clause:

  [LOG ERRORS] SEGMENT REJECT LIMIT count [ROWS | PERCENT]

```
-- To view bad/rejected rows
SELECT  *  FROM gp_read_error_log ('table_name');
```

External Table Planner Statistics

Query planning of complex queries on external tables is not optimal

- Data resides outside the database
- Statistics are not captured for external table data
- Data from external tables are not meant for frequent or ad-hoc access

Performance Tips

- Drop indexes before loading and recreate after the load
- Use gpfdist to load and unload large amounts of data
- Spread the data evenly across as many ETL nodes as possible
- Split very large data files into equal parts and spread the data across as many file systems as possible
- Run two gpfdist instances per file system
- Run gpfdist on as many network interfaces as possible

Greenplum vs Postgres

OLTP versus OLAP

PostgreSQL is for OLTP (Online Transaction Processing)

- Have your hot data in memory
- Use any means necessary to rapidly identify that one dataset you want

Greenplum is for OLAP (Online Analytical Processing)

- Scan all your data, and again
- Calculate results across all your data
- No way this fits into memory - more hardware required

Scalability

PostgreSQL

- Scales well on a single machine
- Using multiple machines involves primary + secondary setups, failover, replication, etc
- Important every machine always holds a full copy (except when sharding)

Greenplum

- Purpose built to scale across many machines
- Comes with replication built-in
- Somewhat slower on OLTP workloads

Query Planner

PostgreSQL

- Grown over time (30 years)
- Complex code, hard to debug
- Not many people fully understand it

Greenplum with GPORCA

- Pluggable query planner, written from scratch in C++
- “Powers” both SQL and NoSQL workloads:
- SQL: Greenplum Database
- NoSQL: SQL-on-Hadoop for Apache HAWQ
- Comes with it’s own test suite (stand-alone)
- Debugging tool allows to extract query information from problematic queries

Partitioning

PostgreSQL

- Included only in last versions
- Basic Runtime partition pruning, otherwise partitions are only pruned during planning time
- Partitions are always scanned when they can’t be included by static values

Greenplum

- Has had partitioning for over a decade now
- Allows partitions and subpartitions (two levels)
- Partitioning is on top of sharding
- GPDB planner can do partition pruning during runtime

Resource Queues

PostgreSQL

- No resource management (usually not needed)

Greenplum

- Resources like CPU, Queries and Memory can be managed on a per-user basis
- Queries can be limited, or aborted, or queued

Pipelining

PostgreSQL

- Has Background Workers now
- Still no complete parallel execution

Greenplum

- Pipelines flow data from one processing part to the next
- Makes efficient use of CPU resources as more cores can be used in parallel

Parallel Loading and Unloading

PostgreSQL

- Usually: COPY (single process, single backend)
- gpLoader speeds up data loading
  - But is still one process

Greenplum

- Can load data using external tables, in parallel on all segments
- Scales out

Access to external resources

PostgreSQL

- Uses SQL/MED
- Requires additional plugins to read/write different formats, or connect resources

Greenplum

- Uses External Tables to access resources
- Scales out on all segments
- Can load data in parallel
- Can execute processes in parallel
- gp_toolkit is using scripts/tools on segments to provide cluster information
- Data can be imported/exported in many formats and can include transformations

Hadoop

PostgreSQL

- ???

Greenplum

- External Tables and the PXF connector allow access to Hadoop systems

Table Compression

PostgreSQL

- ???

Greenplum

Tables (rows or columns) can be compressed using different algorithms

- zlib
- quicklz
- RLE
- Delta

Column Orientation

PostgreSQL

???

Greenplum

- Tables can be row-oriented or column-oriented
- Row-oriented: traditional HEAP table
- Column-oriented: every column is stored in one file
  - Accessing a few columns does not read the entire row
  - Every column can be compressed, even using different algorithms

Text Search

PostgreSQL

- Basic Full Text Search included
- A bit complicated for different languages

Greenplum

GPText

- Integrates Apache Solr Cloud with Greenplum
- Runs in parallel on the segments, no single instance

Expansion

PostgreSQL

- Expansion beyond one system - includes replication, connection pooler, and HA setup
- Does not improve the overall performance (one data copy per system)

Greenplum

- Just add more segments and re-distribute the database

MADlib - Machine Learning at Scale

PostgreSQL

- Works with PostgreSQL

Greenplum

- Works with Greenplum
- Originally designed for Greenplum
- Scaling out will improve MADlib performance

Foreign Keys

PostgreSQL

- Works as expected

Greenplum

FKs are not enforced

- Because of the distributed nature, FK enforcement requires a scan across the entire database, for every single row

Partitioning

Partitioning in Greenplum

- A mechanism for use in physical database design
- Increase the available options for improving the performance of a certain class of queries
- Propagate data to the child tables
- GP partitioning is different than PostgreSQL partitioning
    - https://www.postgresql.org/docs/10/static/ddl-partitioning.html
    - https://www.postgresql.org/docs/9.1/static/ddl-partitioning.html

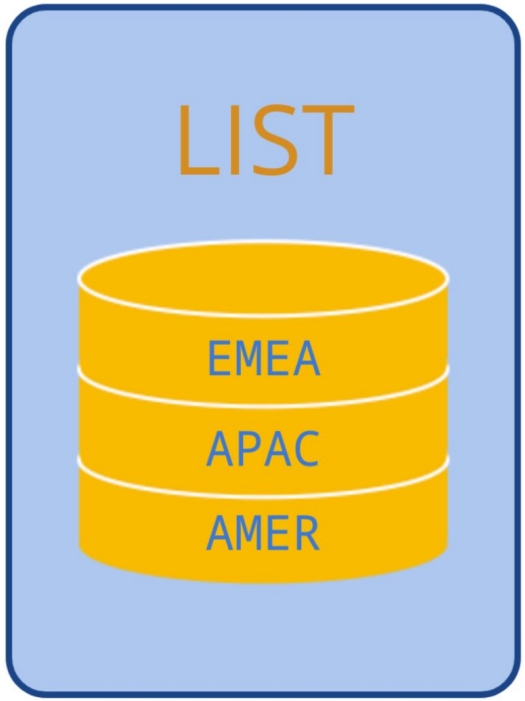
Why Partition a Table?

- To more efficiently query a subset of a large data set
  - decrease query times by avoiding full table scans
- To avoid the overhead and maintenance costs of an index
- To handle increasing volumes of data that the average query doesn't need
- Allow near instantaneous dropping of older data and simple addition of newer data
- Supports a rolling N periods methodology for archiving data

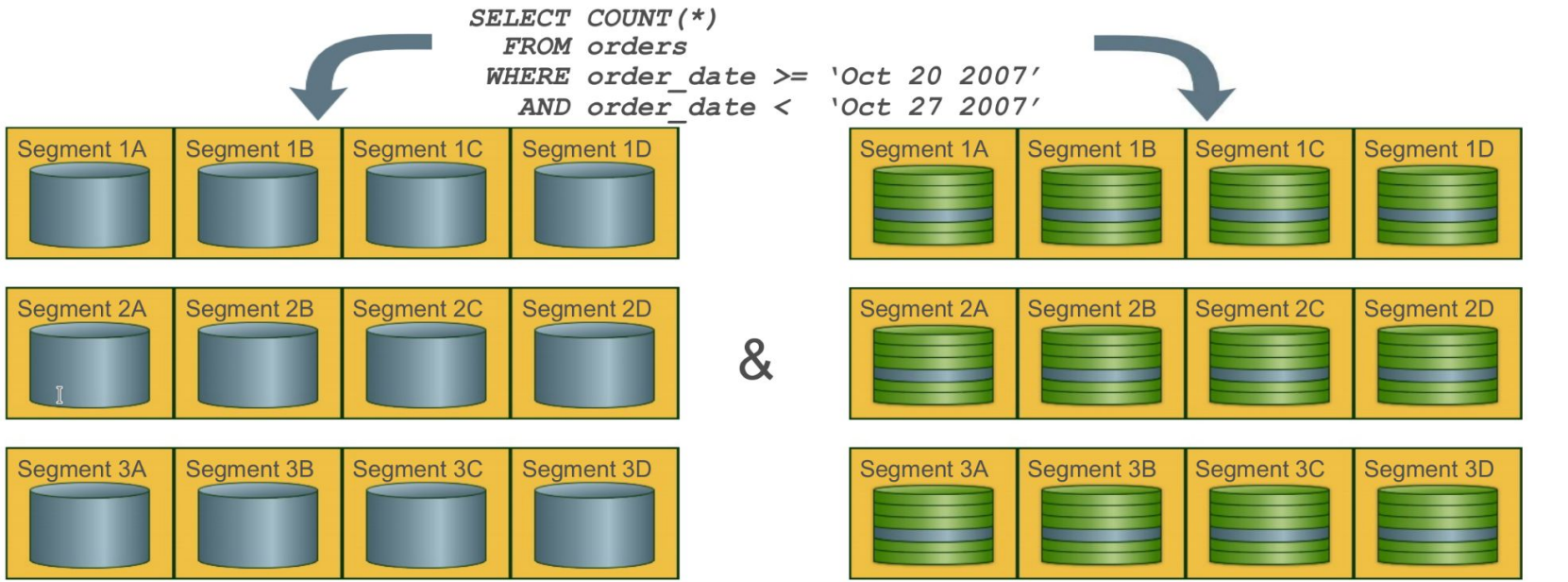
Why Partition a Table?

- Partition elimination/pruning
- Only scan relevant data
- Partition types are RANGE and LIST
- Distribution allocates rows to segments. In each segment, partitioning separates the rows into distinct files.

Partitioning Methods



Distribution & Partitioning



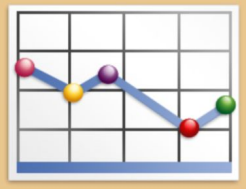
Evenly distribute *orders* data across all segments

Only scans the relevant *order* partitions

When do you partition?

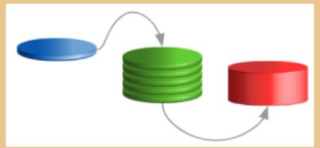
Large fact table

Performance issues



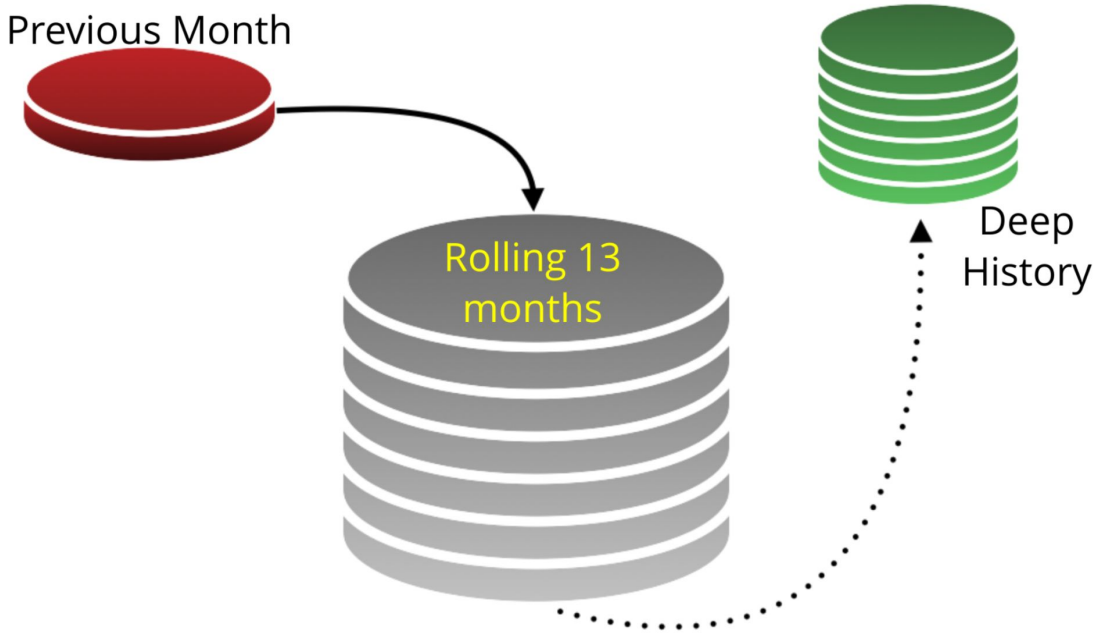
Data divides evenly

Identifiable access patterns



Maintaining rolling data

Historical Data Maintenance



Partitioned Table - DATE RANGE

```
CREATE TABLE sales(
    id int,
    sales_date date,
    amt decimal(10,2)
)
DISTRIBUTED BY (id)
PARTITION BY RANGE (sales_date)
( PARTITION Jan18 START (date '2018-01-01') INCLUSIVE ,
  PARTITION Feb18 START (date '2018-02-01') INCLUSIVE ,
  PARTITION Mar18 START (date '2018-03-01') INCLUSIVE ,
  …
  PARTITION Dec18 START (date '2018-12-01') INCLUSIVE
  END (date '2019-01-01') EXCLUSIVE );
```

Partitioned Table - DATE RANGE

```
CREATE TABLE sales(
    id int,
    sales_date date,
    amt decimal(10,2)
)
DISTRIBUTED BY (id)
PARTITION BY RANGE (sales_date)
( PARTITION pre    END   ('2000-01-01'::date),
  PARTITION yearly START ('2000-01-01'::date)
                 END   ('2020-01-01')
                 EVERY ('1 year'::INTERVAL),
  PARTITION post   START ('2020-01-01'::date)
);
```

Partition Table - NUMERIC RANGE

```
CREATE TABLE ranking(
    id      int,
    rank    int,
    yr      int,
    gender char(1)
)
DISTRIBUTED BY (id)
PARTITION BY RANGE (yr)
( START (2011)
  END   (2021)
  EVERY 1,
  DEFAULT PARTITION extra
);
```

Partition Table - LIST

```
CREATE TABLE widgits_staging (
    id int,
    w_name varchar,
    ...,
    state char(2)
)
DISTRIBUTED BY (id)
PARTITION BY LIST (state)
( PARTITION NE   VALUES ('ME','NH','VT','MA','RI','CT'),
  PARTITION MidA VALUES ('NY','NJ','PA','DE','MD','DC'),
   ...
  PARTITION West VALUES ('CA','OR','WA','AK','HI'),
  DEFAULT PARTITION
);
```

Partitioning an Existing Table

```
orders TABLE:

order_id    int,
order_date  date,
order_amt   numeric(8,2),
...
```

```
1 CREATE TABLE orders_partitioned (
2     LIKE orders
3 )
4 DISTRIBUTED BY (order_num)
5 PARTITION BY RANGE (order_date)
6 ( START ('01/01/2000'::date)
7   END   ('12/31/2025'::date) INCLUSIVE
8   EVERY 1 month
9 );
```

```
INSERT INTO orders_partitioned SELECT * FROM orders;

ALTER TABLE orders RENAME to orders_old;
ALTER TABLE orders_partitioned RENAME to orders;

/* verify data transfer and drop orders_old table */

GRANT ... TO orders;
...
```

Partition Maintenance

- Add, Rename, Truncate, Remove
- Exchange

```
1 /* Table 'sales_cm' contains sales data for the current month */
2 /* At the end of the month, add it to sales_history table */
3 ALTER TABLE sales_cm RENAME to jan18;
4 CREATE TABLE sales_cm (like sales_history) DISTRIBUTED BY ...;
5
6 ALTER TABLE sales_history EXCHANGE PARTITION FOR ('2018-01-01') WITH TABLE jan18;
```

- Split existing partition

```
1 ALTER TABLE sales_history SPLIT PARTITION FOR ('2018-01-01')
2 AT ('2018-01-16')
3 INTO (PARTITION jan181to15, PARTITION jan1816to31);
```

Partition Table Loading

- Top level, parent tables are empty
- Data is loaded into child partitions
- COPY or INSERT automatically load data to the correct partition
- Load a staging table and swap the table in place of an existing partition

Partition Table Summary

- Never distribute and partition on the same column!
- Do not use multi-level partitioning (sub-partitions) unless absolutely necessary
- Use partitioning on large tables to improve performance
- Used in addition to, not in place of, distribution
- Use if the table can be divided into roughly equal parts based on the partitioning key
- RANGE partitioning on DATE
- No overlapping ranges or duplicate values
- Partitioning benefits filtering (WHERE clause)
- Avoid DEFAULT PARTITION if possible