Greenplum Fundamentals

Module 1

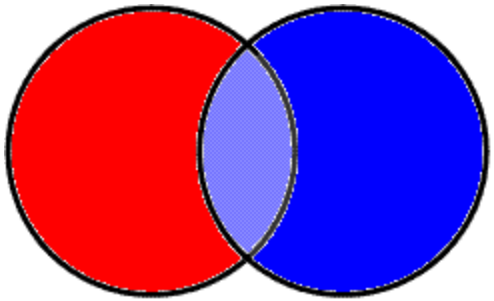
Greenplum
Concepts, Features & Benefits

Module 4

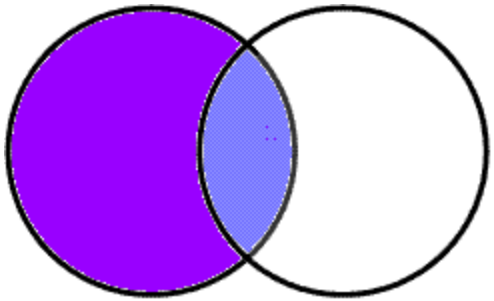
Joining Tables Types & Methods

Greenplum

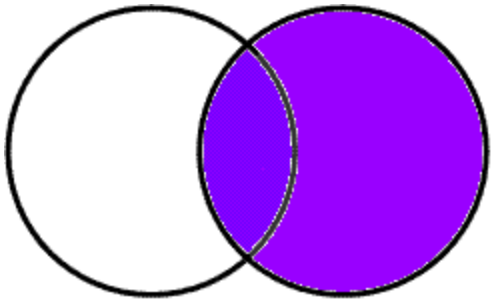
Join Types



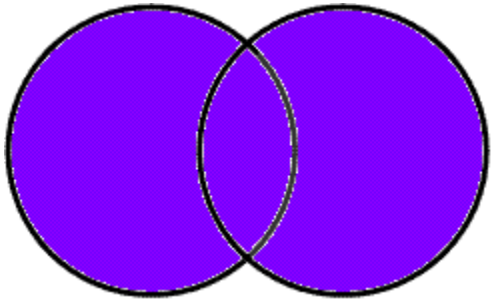
Inner Join



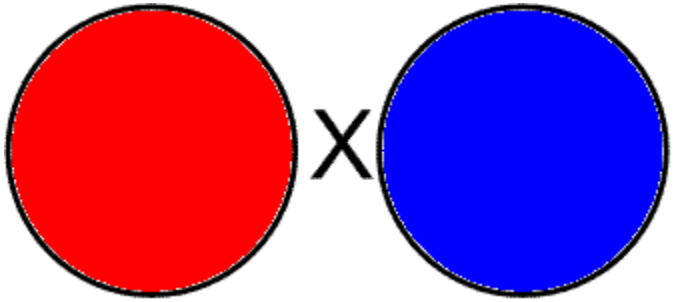
Left Outer Join



Right Outer Join



Full Outer Join



Cross Join

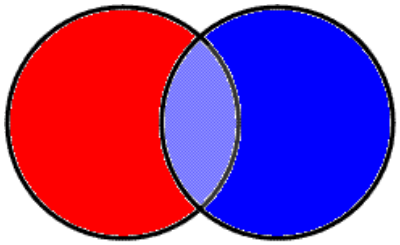
Greenplum

Inner Join

The syntax for an INNER JOIN **is simple!**
Also known as a SIMPLE JOIN **or** EQUI-JOIN**.**

```
SELECT
          EMP.EMPLOYEE_ID
        , EMP.EMPLOYEE_NAME
        , DPT.DEPT_NAME
FROM
          EMPLOYEES      EMP
        , DEPARTMENTS    DPT
WHERE
          EMP.DPT_ID = DPT.DPT_ID;
```

This query only retrieves rows based on the following join condition:
a corresponding row must exist in each table,
thus the term "INNER JOIN".



Greenplum

Left OUTER Join

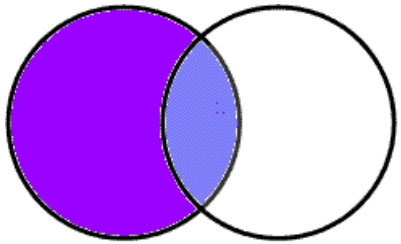
LEFT OUTER JOIN **returns all rows from the "left" table, and only those rows from the "right" table that match the left one.**

This is handy when there may be missing values in the "left" table, but you want to get a description or other information from the "right" table.

```
SELECT
        t.transid
        , c.custname
FROM
        facts.transaction t
        LEFT OUTER JOIN dimensions.customer c
        ON c.customerid = t.customerid;
```

You may also abbreviate LEFT OUTER JOIN to LEFT JOIN.

```
…
FROM
        facts.transaction t
        LEFT JOIN dimensions.customer c
        ON c.customerid = t.customerid;
```



Greenplum

Right OUTER Join

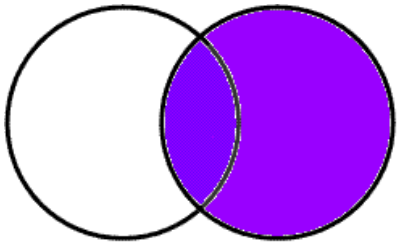
RIGHT OUTER JOIN is the inverse of the LEFT OUTER JOIN.

There is no difference in the functionality of these two. Just be careful when designating the left and right tables!

```
SELECT
        t.transid
        , c.custname
FROM
        dimensions.customer c
        RIGHT OUTER JOIN facts.transaction t
        ON c.customerid = t.customerid;
```

You may also abbreviate RIGHT OUTER JOIN to RIGHT JOIN.

```
…
FROM    dimensions.customer c
        RIGHT JOIN facts.transaction t
        ON c.customerid = t.customerid;
```



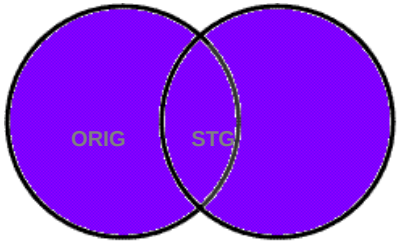
Greenplum

Full OUTER Join

This interesting join is used when you want to retrieve every row, whether there is a match or not based on the join criteria! When would you use this? An example is detecting “true changes” for a table.

```
SELECT
        COALESCE( ORIG.transid, STG.transid) AS transid
      , CASE
          WHEN ORIG.transid IS NULL AND STG.transid IS NOT NULL  THEN  'INSERT'
           WHEN ORIG.transid IS NOT NULL AND STG.transid IS NOT NULL  THEN
'UPDATE'
           WHEN ORIG.transid IS NOT NULL  AND STG.transid IS NULL THEN  'DELETE’
          ELSE 'UNKNOWN’
        END AS Process_type
FROM
        facts.transaction                          ORIG
        FULL OUTER JOIN staging.transaction_w      STG
        ON STG.transid = ORIG.transid;
```

The COALESCE function insures that we have a transaction id to work with in subsequent processing.



Greenplum

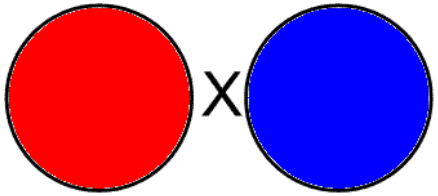
CROSS JOIN (Cartesian Product)

We see these joins, generally, when they are unintentionally created!

- **Every row in the “left” table is joined with every row in the “right” table!**
- **So, if you have a billion row table and CROSS JOIN it to a 100 row table, your answer set will have 100 billion rows!**
- **There are uses for deliberate CROSS JOINs and will be shown later.**

```
SELECT
      t.transid
    , t.transdate
    , COALESCE( c.custname, ’Unknown Customer’) AS custname
FROM
      facts.transaction t
    , dimensions.customer c;
```

Q: What is wrong with this query?
Why will it perform a CROSS JOIN?



Greenplum

Module 5

PostgreSQL Database Essentials

Greenplum

PostgreSQL Schemas – the Search Path

PostgreSQL will search for tables that are named without qualification (schema name not included) by use of the *search_path* variable.

You can change the search_path parameter using `SET` command:

```
SET search_path TO myschema,public;


SHOW search_path;
search_path
--------------
myschema,public
```

Greenplum

PostgreSQL – Roles

- Database roles are conceptually completely separate from operating system users.
- Database roles are global across all databases on the same cluster installation.
- Database user is a role with a LOGIN privilege.
- It is frequently convenient to group users together to ease management of privileges. That way, privileges can be granted to, or revoked from, one group instead of may users separately.

Greenplum

PostgreSQL – Roles and Privileges

- When an object is created, it is assigned an owner**. The default owner is the role that executed the creation statement.**
- **For most kinds of objects, the** initial state **is that** only the owner (or a superuser) can do anything **with it.**
- **To allow** other roles **to use it,** privileges must be granted**.**
- **There are several different kinds of privileges:**

```
SELECT      Read only, unable to create new data or modify
INSERT      Permission to add rows to the object
UPDATE      Permission to make changes to existing data
DELETE      Permission to remove data from the object
CREATE      Permission to create objects
CONNECT     Permission to connect (applies to a database)
EXECUTE     Permission to execute functions / procedures
USAGE       Permission to use objects (applies to a schema)
```

Greenplum

PostgreSQL – Users

- **In PostgreSQL** a user is a role **with login privileges.**
- Privileges on database objects **should be granted to a** group role **rather than a user role.**
- **Users may have** special (system) privileges **not associated with database objects.**

  `SUPERUSER`

  `PASSWORD`

  `CREATEDB`

  `CREATEROLE`

Tip: It is good practice to create a role that has the CREATEDB and CREATEROLE privileges, but is not a superuser, and then use this role for all routine management of databases and roles. This approach avoids the dangers of operating as a superuser for tasks that do not really require it.

Greenplum

PostgreSQL Table Basics - Constraints

- PRIMARY KEY **constraint** insures row uniqueness**.**

  A primary key indicates that a column or group of columns can be used as a unique identifier for rows in the table.

  Technically, a primary key constraint is simply a combination of a `UNIQUE` constraint and a `NOT NULL` constraint.

- FOREIGN KEY **constraint specifies that values in a column (or a group of columns) of a given table must match the values appearing in another table. * NOT SUPPORTED IN GREENPLUM**

Greenplum

PostgreSQL DDL – CREATE TABLE

- **Supports standard ANSI `CREATE TABLE` syntax:**

```
CREATE TABLE dimensions.store
(
  storeId      SMALLINT     NOT NULL,
  storeName    VARCHAR(40)  NOT NULL,
  address      VARCHAR(50)  NOT NULL,
  city         VARCHAR(40)  NOT NULL,
  state        CHAR(2)      NOT NULL,
  zipcode      CHAR(8)      NOT NULL
);
```

Greenplum

PostgreSQL DDL – MODIFY TABLE

- **You can modify certain attributes of a table:**
  - **Rename columns**
  - **Rename tables**
  - **Add/Remove columns**
    - `ALTER TABLE product ADD COLUMN description text;`
  - **Add/Remove constraints**
    - `ALTER TABLE product ALTER COLUMN prod_no SET NOT NULL;`
  - **Change default values**
    - `ALTER TABLE product ALTER COLUMN prod_no SET DEFAULT -999;`
  - **Change column data types**
    - `ALTER TABLE products ALTER COLUMN price TYPE numeric(10,2);`

Greenplum

PostgreSQL DDL – DROP TABLE

- **Table can be removed from the database with the `DROP TABLE` statement:**

```
DROP TABLE dimensions.customer_old;
```

- **If the table you are dropping does not exists, you can avoid error messages if you add "`IF EXISTS`" to the `DROP TABLE` statement:**

```
DROP TABLE IF EXISTS dimensions.customer_old;
```

Greenplum

PostgreSQL Column Basics - Constraints

- **A column can be assigned** a DEFAULT value**.**
- CHECK **constraint is the most generic constraint type. It allows you to specify that the value in a certain column must satisfy a Boolean expression. EXAMPLE: The column *price* must be greater than zero:**

```
price numeric CHECK (price > 0),
```

- NOT NULL **constraint simply specifies that a column must not assume the null value. EXAMPLE: The column *transactionid* may not be NULL:**

```
transactionid NOT NULL,
```

- UNIQUE **constraints ensure that the data contained in a column or a group of columns is unique with respect to all the rows in the table.**

Greenplum

PostgreSQL Indexes

- **PostgreSQL provides** several index types**:**
  - **B-tree**
  - **Hash**
  - **GiST (Generalized Search Tree)**
  - **GIN (Generalized Inverted Index)**
- **Each index type uses a different algorithm that is best suited to different types of queries.**
- By default**, CREATE INDEX command creates a** B-tree index**.**

Greenplum

Constants in PostgreSQL

- **There are three kinds of** implicitly-typed constants **in PostgreSQL:**
  - **Strings**
  - **Bit strings**
  - **Numbers**

- **Constants can also be specified with** explicit types**, which can enable** more accurate representation and more efficient **handling by the system.**

Casting Data Types in PostgreSQL

- A constant of an arbitrary type can be entered using any one of the following explicit notations:

```
type 'string '                  REAL '2.117902'
'string '::type                 '2.117902'::REAL
CAST ( 'string ' AS type )      CAST('2.117902' AS REAL)
```

  The result is a constant of the indicated type.

- If there is no ambiguity the explicit type cast can be omitted (for example, when constant it is assigned directly to a table column), in which case it is automatically converted to the column type.
- The string constant can be written using either regular SQL notation or dollar-quoting.

PostgreSQL Comments

- Standard SQL Comments

```
-- This is a comment
```

- “C” style Comments

```
/* This is a comment block
    that ends with the */
```

Greenplum

PostgreSQL Data Types

Name[1]	Alias	Size	Range	Description
bigint	int8	8 bytes	-9223372036854775808 to 9223372036854775807	large range integer
bigserial	serial8	8 bytes	1 to 9223372036854775807	large autoincrementing integer
bit [(n)]		*n* bits	bit string constant	fixed-length bit string
bit varying [(n)]	varbit	actual number of bits	bit string constant	variable-length bit string
boolean	bool	1 byte	true/false, t/f, yes/no, y/n, 1/0	logical boolean (true/false)
box		32 bytes	((x1,y1),(x2,y2))	rectangular box in the plane - not allowed in distribution key columns.
bytea		1 byte + *binary string*	sequence of octets	variable-length binary string
character [(n)]	char [(n)]	1 byte + *n*	strings up to *n* characters in length	fixed-length, blank padded
character varying [(n)]	varchar [(n)]	1 byte + *string size*	strings up to *n* characters in length	variable-length with limit
cidr		12 or 24 bytes		IPv4 and IPv6 networks
circle		24 bytes	<(x,y),r> (center and radius)	circle in the plane - not allowed in distribution key columns.

Greenplum

PostgreSQL Data Types

Name[1]	Alias	Size	Range	Description
date		4 bytes	4713 BC - 5874897 AD	calendar date (year, month, day)
decimal [(p, s)]	numeric [(p, s)]	variable	no limit	user-specified precision, exact
double precision	float8	8 bytes	15 decimal digits precision	variable-precision, inexact
inet		12 or 24 bytes		IPv4 and IPv6 hosts and networks
integer	int, int4	4 bytes	-2147483648 to +2147483647	usual choice for integer
interval [(p)]		12 bytes	-178000000 years - 178000000 years	time span
line		32 bytes	((x1,y1),(x2,y2))	infinite line in the plane - not allowed in distribution key columns.
lseg		32 bytes	((x1,y1),(x2,y2))	line segment in the plane - not allowed in distribution key columns.
macaddr		6 bytes		MAC addresses
money		4 bytes	-21474836.48 to +21474836.47	currency amount
path		16+16n bytes	[(x1,y1),...]	geometric path in the plane - not allowed in distribution key columns.
point		16 bytes	(x,y)	geometric point in the plane - not allowed in distribution key columns.
polygon		40+16n bytes	((x1,y1),...)	closed geometric path in the plane - not allowed in distribution key columns.
real	float4	4 bytes	6 decimal digits precision	variable-precision, inexact

Greenplum

PostgreSQL Data Types

Name[1]	Alias	Size	Range	Description
serial	serial4	4 bytes	1 to 2147483647	autoincrementing integer
smallint	int2	2 bytes	-32768 to +32767	small range integer
text		1 byte + *string size*	strings of any length	variable unlimited length
time [(p)] [without time zone]		8 bytes	00:00:00 - 24:00:00	time of day only
time [(p)] with time zone	timetz	12 bytes	00:00:00+1359 - 24:00:00-1359	time of day only, with time zone
timestamp [(p)] [without time zone]		8 bytes	4713 BC - 5874897 AD	both date and time
timestamp [(p)] with time zone	timestamptz	8 bytes	4713 BC - 5874897 AD	both date and time, with time zone

Greenplum

PostgreSQL Functions & Operators

- PostgreSQL has a wide range of built-in functions and operators for all data types.

- Users can define custom operators and functions.

  Use caution, new operators or functions may adversely effect Greenplum MPP functionality.

Built-In Functions (select function)

CURRENT_DATE **returns the current system date**

Example: 2006-11-06

CURRENT_TIME **returns the current system time**

Example: 16:50:54

CURRENT_TIMESTAMP **returns the current system date and time**

Example: 2008-01-06 16:51:44.430000+00:00

LOCALTIME **returns the current system time with time zone adjustment**

Example: 19:50:54

LOCALTIMESTAMP **returns the current system date and time with time zone adjustment**

Example: 2008-01-06 19:51:44.430000+00:00

CURRENT_ROLE **or** USER **returns the current database user**

Example: jdoe

Greenplum

PostgreSQL Comparison Operators

The usual comparison operators are available:

Operator	Description
=	Equal to
!= OR <>	NOT Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
x BETWEEN y AND z	Short hand for *x >= y and x <= z*
x IS NULL	True if x has NO VALUE
'abc' LIKE '%abcde%'	Pattern Matching
POSIX Comparisons	POSIX Pattern Matching

Greenplum

PostgreSQL Mathematical Functions

Commonly used Mathematical functions:

Function	Returns	Description	Example	Results
+ - * /	same	Add, Subtract, Multiply & Divide	1 + 1	2
%	Integer	Modulo	10%2	0
^	Same	Exponentiation	2^2	4
|/	Numeric	Square Root	|/9	3
||/	Numeric	Cube Root	||/8	2
!	Numeric	Factorial	!3	6
& | # ~	Numeric	Bitwise And, Or, XOR, Not	91 & 15	11
<< >>	Numeric	Bitwise Shift left, right	1 << 4 8 >> 2	16 2

Greenplum

PostgreSQL Mathematical Functions

Handy Mathematical functions:

Function	Returns	Description	Example	Results
Abs	same	Absolute Value	Abs(-998.2)	998.2
Ceiling (numeric)	Numeric	Returns smallest integer not less than argument	Ceiling(48.2)	49
Floor (numeric)	Numeric	Returns largest integer not greater than argument	Floor(48.2)	48
Pi()	Numeric	The π constant	Pi()	3.1419…
Random()	Numeric	Random value between 0.0 and 1.0	Random()	.87663
Round()	Numeric	Round to nearest integer	Round(22.7)	23

Greenplum

PostgreSQL String Functions

Commonly used string functions:

Function	Returns	Description	Example	Results
String || String	Text	String concatenation	'my' || 'my'	'mymy'
Char_length (string)	Integer	number of chars in string	Char_length('mymy')	4
Position (string in string)	Integer	Location of specified sub-string	Position('my' in 'ohmy')	3
Lower(string)	Text	Converts to lower case	Lower('MYMY')	'mymy'
Upper(string)	Text	Converts to upper case	Upper('mymy')	'MYMY'
Substring (string from n for n)	Text	Displays portion of string	Substring('myohmy' from 3 for 2)	'oh'
Trim(both,leading,trailing from string)	Text	Remove leading and/or trailing characters	Trim(' mymy ')	'mymy'

PostgreSQL String Functions

Handy string functions:

Function	Returns	Description	Example	Results
`Initcap (string)`	`Text`	`Changes case`	`Initcap ('my my')`	`'My My'`
`Length (string)`	`Integer`	`Returns string length`	`Length ('mymy')`	`4`
`Split_part (string, delimiter, occurrence)`	`Text`	`Separates delimited list`	`Split_part('one|two|three','|',2)`	`'two'`

PostgreSQL Date Functions

Commonly used date functions:

Function	Returns	Description	Example	Results
Age (timestam,timestamp)	Timestamp	Difference in years, months and days	Age('2008-08-12' timestamp, current_timestamp)	0 years 1 month 11 days
Extract (field from timestamp)	Integer	Returns year, month, day, hour, minute or second	Extract(day from current_date)	11
Now()	Timestamp	Returns current date & time	Now()	2008-09-22 11:00:01
Overlaps	Boolean	Simplifies comparing date ranges	WHERE ('2008-01-01','2008-02-11') overlaps ('2008-02-01','2008-09-11')	TRUE

Greenplum

Module 6

Greenplum SQL

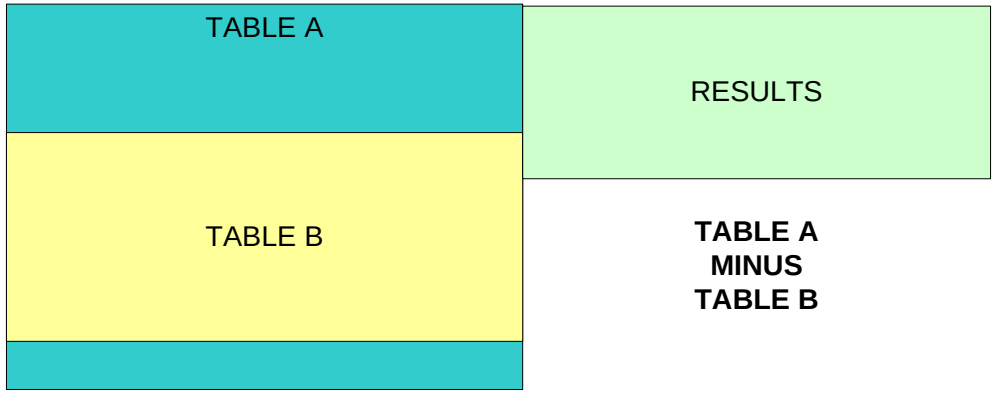
Set Operations – EXCEPT – Example

Returns all rows from the first SELECT statement except for those also returned by the second SELECT.

```
SELECT    t.transid
          c.custname
FROM      facts.transaction t
     JOIN dimensions.customer c
       ON c.customerid = t.customerid
```

EXCEPT

```
SELECT t.transid
       c.custname
FROM   facts.transaction t
  JOIN dimensions.customer c
    ON c.customerid = t.customerid
WHERE  t.transdate BETWEEN
       '2008-01-01' AND '2008-01-21'
```



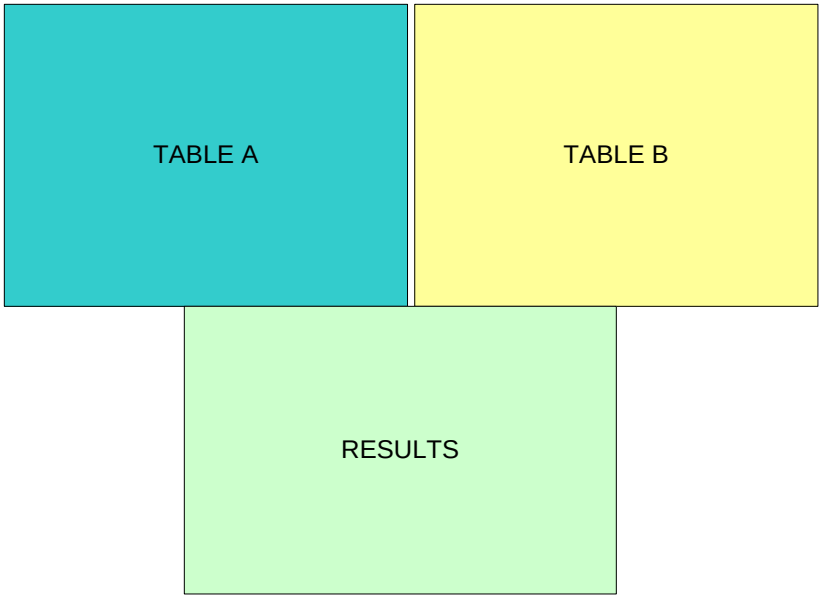
Greenplum

Set Operations – UNION – Example

```
SELECT t.transid
       c.custname
FROM   facts.transaction t
  JOIN dimensions.customer c
    ON c.customerid = t.customerid
WHERE  t.transdate BETWEEN
       '2008-01-01' AND '2008-05-17'
```

UNION

```
SELECT t.transid
       c.custname
FROM   facts.transaction t
  JOIN dimensions.customer c
    ON c.customerid = t.customerid
WHERE  t.transdate BETWEEN
       '2008-01-01' AND '2008-01-21'
```



The UNION operation DOES NOT ALLOW DUPLICATE ROWS

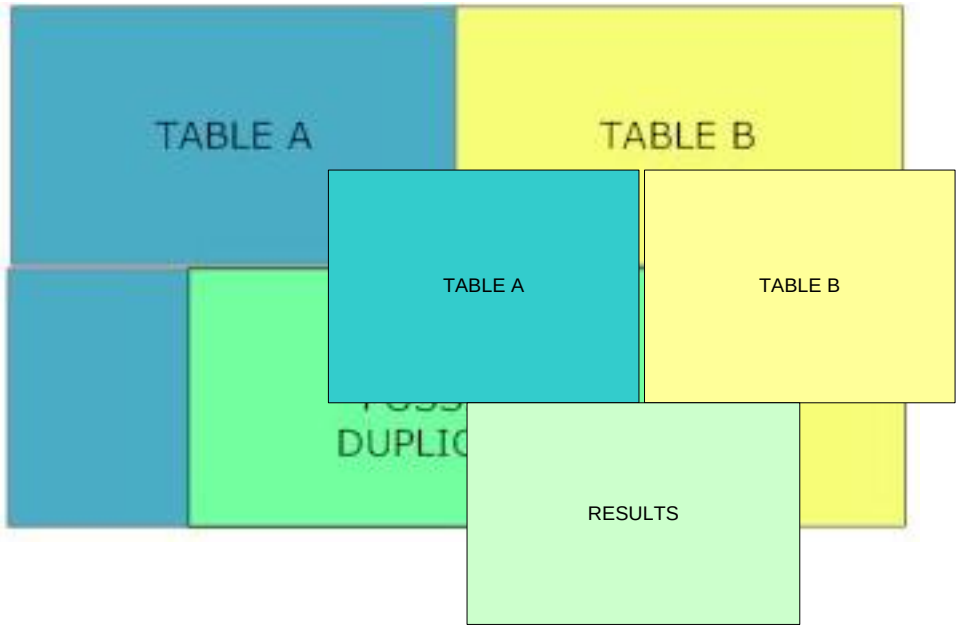
Greenplum

Set Operations – UNION ALL - Example

```
SELECT t.transid
       c.custname
FROM   facts.transaction t
  JOIN dimensions.customer c
    ON c.customerid = t.customerid
WHERE  t.transdate BETWEEN
       '2008-01-01' AND '2008-05-17'
```

UNION ALL

```
SELECT t.transid
       c.custname
FROM   facts.transaction t
  JOIN dimensions.customer c
    ON c.customerid = t.customerid
WHERE  t.transdate BETWEEN
       '2008-01-01' AND '2008-01-21'
```



The UNION ALL operation MAY CREATE DUPLICATE ROWS

Greenplum allows Sub-Queries

IN clause is fully supported …

ORIGINAL QUERY:

```
SELECT    *
FROM      facts.transaction t
WHERE     t.customerid IN (SELECT customerid
                           FROM dimensions.customer c);
```

HOWEVER, THIS WILL PERFORM BETTER IN MOST CASES:

```
SELECT  t.*
FROM    facts.transaction t
INNER JOIN     dimensions.customer c
               ON c.customerid = t.customerid;
```

Greenplum

Module 10

PostGREs Functions

Greenplum

Types of Functions

Greenplum supports several function types.

- **query language functions (functions written in SQL)**
- **procedural language functions (functions written in for example, PL/pgSQL or PL/Tcl)**
- **internal functions**
- **C-language functions**

This class will only present the first two types of functions: Query Language and Procedural Language

Query Language Function – More Rules

- **May contain SELECT statements**
- **May contain DML statements**
  - **Insert, Update or Delete statements**
- **May not contain:**
  - **Rollback, Savepoint, Begin or Commit commands**
- **Last statement must be SELECT unless the return type is *void*.**

Query Language Function – Example

This function has no parameters and no return set:

```
CREATE FUNCTION public.clean_customer()
  RETURNS void AS 'DELETE FROM dimensions.customer
    WHERE state = ''NA''; '
 LANGUAGE SQL;
```

It is executed as a SELECT statement:

```
SELECT public.clean_customer();

clean_customer
-----------
(1 row)
```

Greenplum

Query Language Function – With Parameter

```
CREATE FUNCTION public.clean_specific_customer(which
   char(2))
   RETURNS void AS 'DELETE FROM dimensions.customer
     WHERE state = $1; '
 LANGUAGE SQL;
```

It is executed as a SELECT statement:

```
SELECT public.clean_specific_customer('NA');

clean_specific_customer
-----------------------

(1 row)
```

Greenplum

SQL Functions Returning Sets

When an SQL function is declared as returning SETOF *sometype*, the function's final SELECT query is executed to completion, and each row it outputs is returned as an element of the result set.

```
CREATE FUNCTION dimensions.getstatecust(char)
        RETURNS SETOF customer AS $$
    SELECT *
      FROM dimensions.customer WHERE state = $1;
$$ LANGUAGE SQL;

SELECT *,UPPER(city)
FROM dimensions.getstatecust('WA');
```

This function returns all qualifying rows.

Greenplum

Procedural Language Functions

- **PL/pgSQL Procedural Language for Greenplum**
  - **Can be used to create functions and procedures**
  - **Adds control structures to the SQL language**
  - **Can perform complex computations**
  - **Inherits all user-defined types, functions, and operators**
  - **Can be defined to be trusted by the server**
  - **Is (relatively) easy to use.**

Structure of PL/pgSQL

- **PL/pgSQL is a block-structured language.**

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
DECLARE quantity integer := 30;
BEGIN
RAISE NOTICE 'Quantity here is %', quantity;
-- Prints 30 quantity := 50;
-- Create a subblock
   DECLARE quantity integer := 80;
   BEGIN
       RAISE NOTICE 'Quantity here is %', quantity;
       -- Prints 80
       RAISE NOTICE 'Outer quantity here is %', outerblock.quantity;
      -- Prints 50
   END;
RAISE NOTICE 'Quantity here is %', quantity;
-- Prints 50
RETURN quantity;
END;
$$ LANGUAGE plpgsql;
```

Greenplum

PL/pgSQL Declarations

- **All variables used in a block must be declared in the declarations section of the block.**
- **SYNTAX:**

name [CONSTANT] type [NOT NULL] [{ DEFAULT | := } expression];

- **EXAMPLES:**

user_id integer;

quantity numeric(5);

url varchar;

myrow tablename%ROWTYPE;

myfield tablename.columnname%TYPE;

somerow RECORD;

PL/pgSQL %Types

- **%TYPE provides the data type of a variable or table column.**
- **You can use this to declare variables that will hold database values.**

SYNTAX: `variable%TYPE`

EXAMPLES:

```
custid customer.customerid%TYPE
tid    transaction.transid%TYPE
```

Greenplum

PL/pgSQL Basic Statements

- **Assignment**

```
variable := expression;
```

- **Executing DML (no RETURN)**

```
PERFORM myfunction(myparm1,myparm2);
```

- **Single row results**

  Use `SELECT … INTO mytarget%TYPE;`

PL/pgSQL – Executing Dynamic SQL

- **Oftentimes you will want to generate dynamic commands inside your PL/pgSQL functions**
- **This is for commands that will involve different tables or different data types each time they are executed.**
- **To handle this sort of problem, the EXECUTE statement is provided:**

```
EXECUTE command-string [ INTO [STRICT] target ];
```

Greenplum

PL/pgSQL – Dynamic SQL Example

Dynamic values that are to be inserted into the constructed query require careful handling since they might themselves contain quote characters.

```
EXECUTE 'UPDATE tbl SET '
     || quote_ident(colname)
     || ' = '
     || quote_literal(newvalue)
     || ' WHERE key = '
     || quote_literal(keyvalue);
```

Greenplum

PL/pgSQL – FOUND

- **FOUND is set by:**
  - **A SELECT INTO statement sets FOUND true if a row is assigned, false if no row is returned.**
  - **A PERFORM statement sets FOUND true if it produces (and discards) one or more rows, false if no row is produced.**
  - **UPDATE, INSERT, and DELETE statements set FOUND true if at least one row is affected, false if no row is affected.**
  - **A FETCH statement sets FOUND true if it returns a row, false if no row is returned.**
  - **A MOVE statement sets FOUND true if it successfully repositions the cursor, false otherwise.**
  - **A FOR statement sets FOUND true if it iterates one or more times, else false.**

Greenplum

PL/pgSQL – Returning from a Function

- **`RETURN NEXT` and `RETURN QUERY` do not actually return from the function — they simply append zero or more rows to the function's result set.**

EXAMPLE:

```
CREATE OR REPLACE FUNCTION getAllStores()
RETURNS SETOF store AS
$BODY$
DECLARE r store%rowtype;
BEGIN
FOR r IN SELECT * FROM store WHERE storeid > 0 LOOP
   -- can do some processing here
   RETURN NEXT r;
  -- return current row of SELECT
END LOOP;
RETURN;
END
$BODY$
LANGUAGE plpgsql;
```

Greenplum

PL/pgSQL – Conditionals

IF – THEN – ELSE conditionals lets you execute commands based on certain conditions.

```
IF boolean-expression THEN
       statements
[ ELSIF boolean-expression THEN
       statements
[ ELSIF boolean-expression THEN
   statements
[ ELSE statements ]
END IF;
```

TIP: Put the most commonly occurring condition first!

Greenplum

PL/pgSQL – Simple Loops

- **With the `LOOP`, `EXIT`, `CONTINUE`, `WHILE`, and `FOR` statements, you can arrange for your PL/pgSQL function to repeat a series of commands.**

LOOP SYNTAX:

```
[ <<label>> ]
LOOP
    statements
END LOOP [ label ];
```

PL/pgSQL – Simple Loops

EXIT EXAMPLES:

```
LOOP
  -- some computations
  IF count > 0
    THEN EXIT;
    -- exit loop
  END IF;
END LOOP;
LOOP
   -- some computations
   EXIT WHEN count > 0;
   -- same result as previous example
END LOOP;
BEGIN
 -- some computations
   IF stocks > 100000
      THEN EXIT;
 -- causes exit from the BEGIN block
END IF;
END;
```

Greenplum

PL/pgSQL – Simple Loops

CONTINUE can be used within all loops.

```
CONTINUE EXAMPLE:
LOOP
    -- some computations
       EXIT WHEN count > 100;
       CONTINUE WHEN count < 50;
       -- some computations for count IN [50 .. 100]
END LOOP;
```

Greenplum

PL/pgSQL – Simple Loops

The WHILE statement repeats a sequence of statements so long as the boolean-expression evaluates to true. The expression is checked just before each entry to the loop body.

```
WHILE EXAMPLE:
WHILE customerid < 50 LOOP
    -- some computations
END LOOP;
```

Greenplum

PL/pgSQL – Simple For (integer) Loops

This form of `FOR` creates a loop that iterates over a range of integer values.

EXAMPLE:

```
FOR i IN 1..3 LOOP
  -- i will take on the values 1,2,3 within the loop
END LOOP;

FOR i IN REVERSE 3..1 LOOP
  -- i will take on the values 3,2,1 within the loop
END LOOP;

FOR i IN REVERSE 8..1 BY 2 LOOP
  -- i will take on the values 8,6,4,2 within the loop
END LOOP;
```

Greenplum

PL/pgSQL – Simple For (Query) Loops

Using a different type of FOR loop, you can iterate through the results of a query and manipulate that data accordingly.

EXAMPLE:

```
CREATE FUNCTION nukedimensions() RETURNS integer AS $$
DECLARE mdims RECORD;
BEGIN
 FOR mdims IN SELECT * FROM pg_class WHERE schema='dimensions' LOOP
  -- Now "mdims" has one record from pg_class
  EXECUTE 'TRUNCATE TABLE ' || quote_ident(mdimss.relname);
 END LOOP;
RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

Greenplum

PL/pgSQL – Trapping Errors

- **By default, any error occurring in a PL/pgSQL function aborts execution of the function, and indeed of the surrounding transaction as well.**
- **You can trap errors and recover from them by using a BEGIN block with an EXCEPTION clause.**

```
BEGIN
    statements
    EXCEPTION WHEN condition [OR condition ... ] THEN
      handler_statements
           [ WHEN condition [ OR condition ... ] THEN
      handler_statements ... ]
END;
```

Greenplum

PL/pgSQL – Common Exception Conditions

NO_DATA	No data matches predicates
CONNECTION_FAILURE	SQL cannot connect
DIVISION_BY_ZERO	Invalid division operation 0 in divisor
INTERVAL_FIELD_OVERFLOW	Insufficient precision for timestamp
NULL_VALUE_VIOLATION	Tried to insert NULL into NOT NULL column
RAISE_EXCEPTION	Error raising an exception
PLPGSQL_ERROR	PL/pgSQL error encountered at run time
NO_DATA_FOUND	Query returned zero rows
TOO_MANY_ROWS	Anticipated single row return, received more than 1 row.

Greenplum