

How To Partition a Table in Postgres Plus^(R)

A Postgres Evaluation Quick Tutorial From EnterpriseDB

November 19, 2009

EnterpriseDB Corporation, 235 Littleton Road, Westford, MA 01866, USA
T +1 978 589 5700 **F** +1 978 589 5701 **E** info@enterprisedb.com **www**.enterprisedb.com

Introduction

This tutorial shows you how to quickly create table partitions in Postgres Plus Standard or Advanced Servers and then query the data across the partitions.

This [EnterpriseDB](#) Quick Tutorial helps you get started with the [Postgres Plus Standard Server](#) or [Postgres Plus Advanced Server](#) database products in a Linux, Windows or Mac environment. It is assumed that you have already downloaded and installed Postgres Plus Standard Server or Postgres Plus Advanced Server on your desktop or laptop computer.

This Quick Tutorial is designed to help you expedite your Technical Evaluation of Postgres Plus Standard Server or Postgres Plus Advanced Server. For more informational assets on conducting your evaluation of Postgres Plus, visit the self-service web site, [Postgres Plus Open Source Adoption](#).

In this Quick Tutorial you will learn how to do the following:

- create a partitioned table using rules
- add test data to a rule partitioned table and view it
- create a partitioned table using triggers
- add test data to a trigger partitioned table and view it

Usage Note: While the examples in this tutorial are demonstrated in a Windows environment, the steps are the same for the Linux and Mac environments. You will notice slight variations between the operating systems; there are differences in the tools used (e.g. terminal windows and text editors), the use of forward slashes vs. back slashes in path specifications, and the installation directory locations.

Feature Description

Creating a Partitioned Table

Partitioning refers to splitting what is logically one large table into smaller physical pieces. Partitioning can provide several benefits:

1. Query performance can be improved dramatically in certain situations, particularly when most of the heavily accessed rows of the table are in a single partition or a small number of partitions. Partitioning allows you to omit the partition column from the front of an index, reducing index size and making it more likely that the heavily used parts of the index fits in memory.
2. When a query or update accesses a large percentage of a single partition, performance may improve because the server will perform a sequential scan of

the partition instead of using an index and random access reads scattered across the whole table.

3. A bulk load (or unload) can be implemented by adding or removing partitions, if plan that requirement into the partitioning design. `ALTER TABLE` is far faster than a bulk operation. It also entirely avoids the `VACUUM` overhead caused by a bulk `DELETE`.
4. Seldom-used data can be migrated to cheaper and slower storage media.

Table partitioning is worthwhile only when a table would otherwise be very large. The exact point at which a table will benefit from partitioning depends on the application; a good rule of thumb is that the size of the table should exceed the physical memory of the database server.

Postgres Plus uses inheritance, check-constraints and rules, or triggers to create partitioned tables.

More information about inheritance in Postgres Plus can be found at:

<http://www.enterprisedb.com/docs/en/8.4/pg/ddl-inherit.html>

More information about partitioning can be found at:

<http://www.enterprisedb.com/docs/en/8.4/pg/ddl-partitioning.html>

Using Rules vs. Triggers to Create Partitioned Tables

Executing a rule consumes significantly more system resources than a trigger, but the cost is paid once per query rather than once per row; using rules to establish partitioned tables might be advantageous for bulk-insert situations. In most cases, however, the trigger method offers better performance.

Be aware that the `COPY` command ignores rules. If you are using `COPY` to insert data, you must copy the data into the correct child table rather than into the parent. `COPY` does fire triggers, so you can use it normally if you create partitioned tables using the trigger approach.

Tutorial Steps

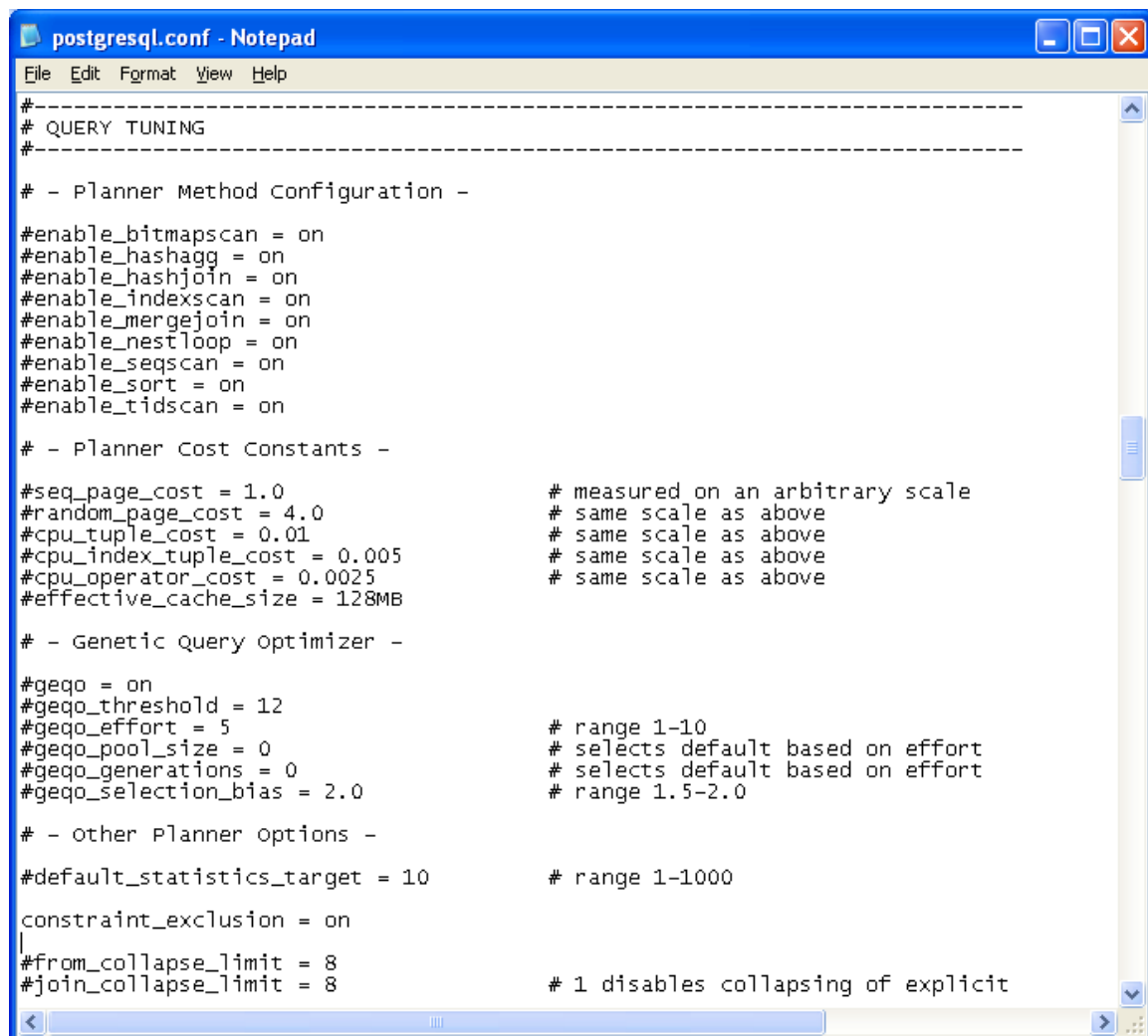
Creating Partitioned Tables with Rules

In this section, we create a master table named `sales`, and then partition the data stored in that table into two partitions based on the value of the `org` column. The following example demonstrates how to use *rules* to create partitioned tables.

In the example, the rule is a code snippet that intercepts the `INSERT` statement and re-directs the `INSERT` to add data to a partitioned table based on the value of the data.

Step 1 (IMPORTANT): Enable `constraint_exclusion`.

Since Postgres Plus uses the concept of constraint exclusion to enable partition boundary checking, it is critical that you set the following parameter in the `postgresql.conf` file. You can open the `postgresql.conf` file (shown below) from the Start menu by navigating to the Postgres Plus menu, and choosing Expert Configuration and then Edit `postgresql.conf`. When the `postgresql.conf` file opens, scroll down to the `QUERY TUNING` section and remove the pound sign (#) from in front of the `constraint_exclusion` entry. Set the `constraint_exclusion` parameter to `on`, and save the `postgresql.conf` file before exiting.



```
postgresql.conf - Notepad
File Edit Format View Help
#-----
# QUERY TUNING
#-----

# - Planner Method Configuration -
#enable_bitmapscan = on
#enable_hashagg = on
#enable_hashjoin = on
#enable_indexscan = on
#enable_mergejoin = on
#enable_nestloop = on
#enable_seqscan = on
#enable_sort = on
#enable_tidscan = on

# - Planner Cost Constants -
#seq_page_cost = 1.0           # measured on an arbitrary scale
#random_page_cost = 4.0       # same scale as above
#cpu_tuple_cost = 0.01        # same scale as above
#cpu_index_tuple_cost = 0.005  # same scale as above
#cpu_operator_cost = 0.0025    # same scale as above
#effective_cache_size = 128MB

# - Genetic Query Optimizer -
#geqo = on
#geqo_threshold = 12
#geqo_effort = 5               # range 1-10
#geqo_pool_size = 0           # selects default based on effort
#geqo_generations = 0         # selects default based on effort
#geqo_selection_bias = 2.0    # range 1.5-2.0

# - other Planner options -
#default_statistics_target = 10 # range 1-1000
constraint_exclusion = on
|
#from_collapse_limit = 8
#join_collapse_limit = 8      # 1 disables collapsing of explicit
```

After setting this parameter, you must signal the server to reload the configuration file using the `pg_ctl` utility. Open a terminal window, and navigate to:

```
C:\PostgresPlus\8.3R2AS\dbserver\bin
```

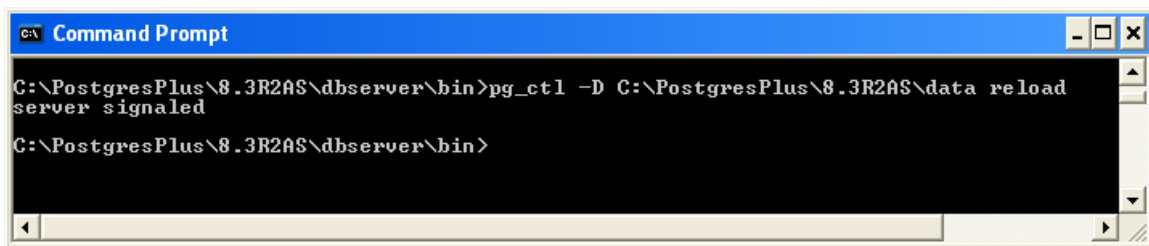
Enter the following command (as shown in the Terminal window):

```
pg_ctl -D <datadir> reload
```

<datadir> is the full path to your data directory.

By default, the data for a standard installation of Postgres Plus Advanced Server is stored in C:\PostgresPlus\8.3R2AS\data. The location of the data directory may vary depending on your version of Postgres Plus and the installation options chosen at install time.

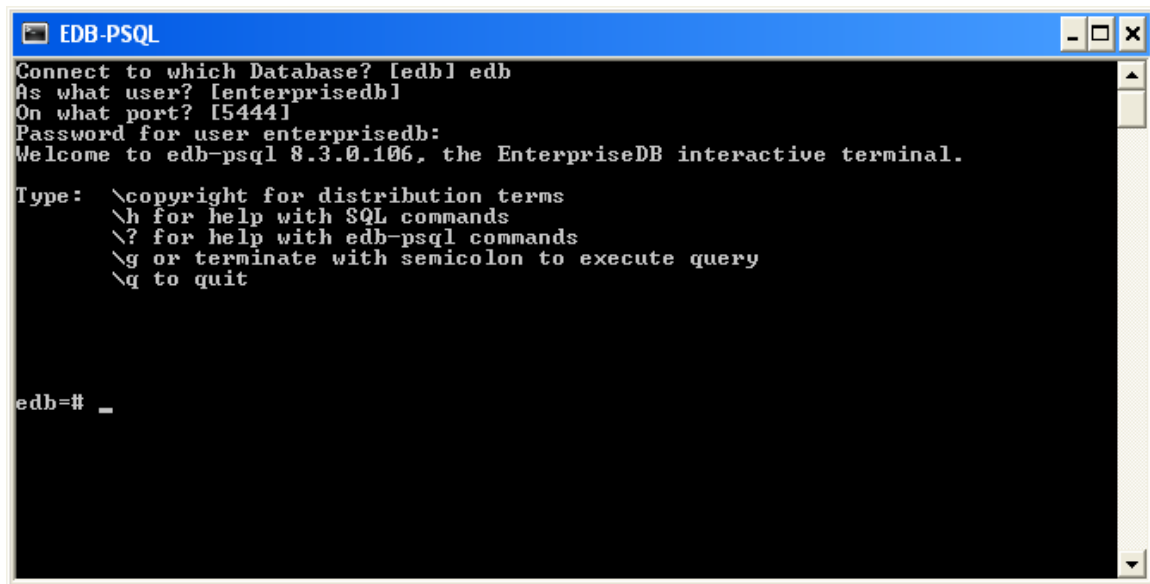
The following screenshot shows the process of reloading the configuration file in Windows:



Step 2: Create the parent table.

Note: You can use the PSQL command line to enter the rest of the commands used in this tutorial.

You can access the PSQL command line by opening the Start menu and navigating to the Postgres Plus menu. Choose Run SQL command line, and then EDB-PSQL to open the EDB-PSQL window:



When prompted, enter your connection information. When you've specified your connection information (Database name, user name, password and port number), Postgres Plus opens the PSQL command line. Enter the command:

```
CREATE TABLE sales(org int, name varchar(10));
```

Step 3: Create the child (partitioned) tables.

Use the PSQL command line to create the child tables:

```
CREATE TABLE sales_part1
(CHECK (org < 6))
INHERITS (sales);

CREATE TABLE sales_part2
(CHECK (org >=6 and org <=10))
INHERITS (sales);
```

Step 4: Create the rules.

This step creates the rules that Postgres Plus follows when adding a record to the parent table. The rules state that instead of inserting the data directly into the parent table, the data should be added to a given partitioned table based on the value of the `org` column. If the value of `org` is less than 6, the new record is added to `sales_part1`; if the value is greater than or equal to 6 and less than or equal to 10, the new record is added to `sales_part2`.

Note: Any data that does not fall within the range specified in the rules will be stored in the parent table. In our example, a row with a value of 11 in the `org` column would be stored in the `sales` table.

```
CREATE OR REPLACE RULE insert_sales_p1
AS ON INSERT TO sales
WHERE (org <6)
DO INSTEAD
INSERT INTO sales_part1 VALUES(NEW.org, NEW.name);

CREATE OR REPLACE RULE insert_sales_p2
AS ON INSERT TO sales
WHERE (org >=6 and org <=10 )
DO INSTEAD
INSERT INTO sales_part2 VALUES(New.org,New.name);
```

NOTE: You must create additional rules to handle UPDATE and DELETE operations.

Step 5: Add sample data to the new table.

```
INSERT INTO sales VALUES(1,'Craig');
INSERT INTO sales VALUES(2,'Mike');
INSERT INTO sales VALUES(3,'Michelle');
INSERT INTO sales VALUES(4,'Joe');
INSERT INTO sales VALUES(5,'Scott');
INSERT INTO sales VALUES(6,'Roger');
INSERT INTO sales VALUES(7,'Fred');
INSERT INTO sales VALUES(8,'Sam');
INSERT INTO sales VALUES(9,'Sonny');
INSERT INTO sales VALUES(10,'Chris');
```

Step 6: Confirm that the data was added to the parent table.

```
postgres=# SELECT * FROM sales;
org | name
-----+-----
1 | Craig
2 | Mike
3 | Michelle
4 | Joe
5 | Scott
6 | Roger
7 | Fred
8 | Sam
9 | Sonny
10 | Chris
```

Of course, that doesn't show you which partition the data actually resides in; it does demonstrate that you can still access the data in all partitions simply by querying the parent table. That means that if you partition a table that already exists and is in use by an application, you don't have to modify the application to be partition-aware.

You can see which rows are stored in each child partition with the following commands:

```

postgres=# SELECT * FROM sales_part1;
 org |   name
-----+-----
   1 | Craig
   2 | Mike
   3 | Michelle
   4 | Joe
   5 | Scott
(5 rows)

postgres=# SELECT * FROM sales_part2;
 org | name
-----+-----
   6 | Roger
   7 | Fred
   8 | Sam
   9 | Sonny
  10 | Chris
(5 rows)

```

As you can see, the data was sent to the appropriate partition.

The PostgreSQL EXPLAIN statement can provide some insight into how the Postgres Plus query optimizer will execute a given statement. You can use the EXPLAIN statement to analyze SELECT, INSERT, DELETE, UPDATE and DECLARE...CURSOR commands. The EXPLAIN statement does not actually execute the statement being analyzed, and any (time) cost analysis is only an estimate.

An EXPLAIN plan for an *unconstrained* query that selects all rows from the parent table demonstrates that the server must read each child table to satisfy the query:

```

postgres=# EXPLAIN SELECT * FROM sales;
               QUERY PLAN
-----
Result  (cost=0.00..54.30 rows=2430 width=70)
-> Append (...)
     -> Seq Scan on sales (...)
     -> Seq Scan on sales_part1 sales (...)
     -> Seq Scan on sales_part2 sales (...)

```

If you include the ANALYZE keyword in an EXPLAIN statement, Postgres Plus executes the command and provides performance costs for the execution.

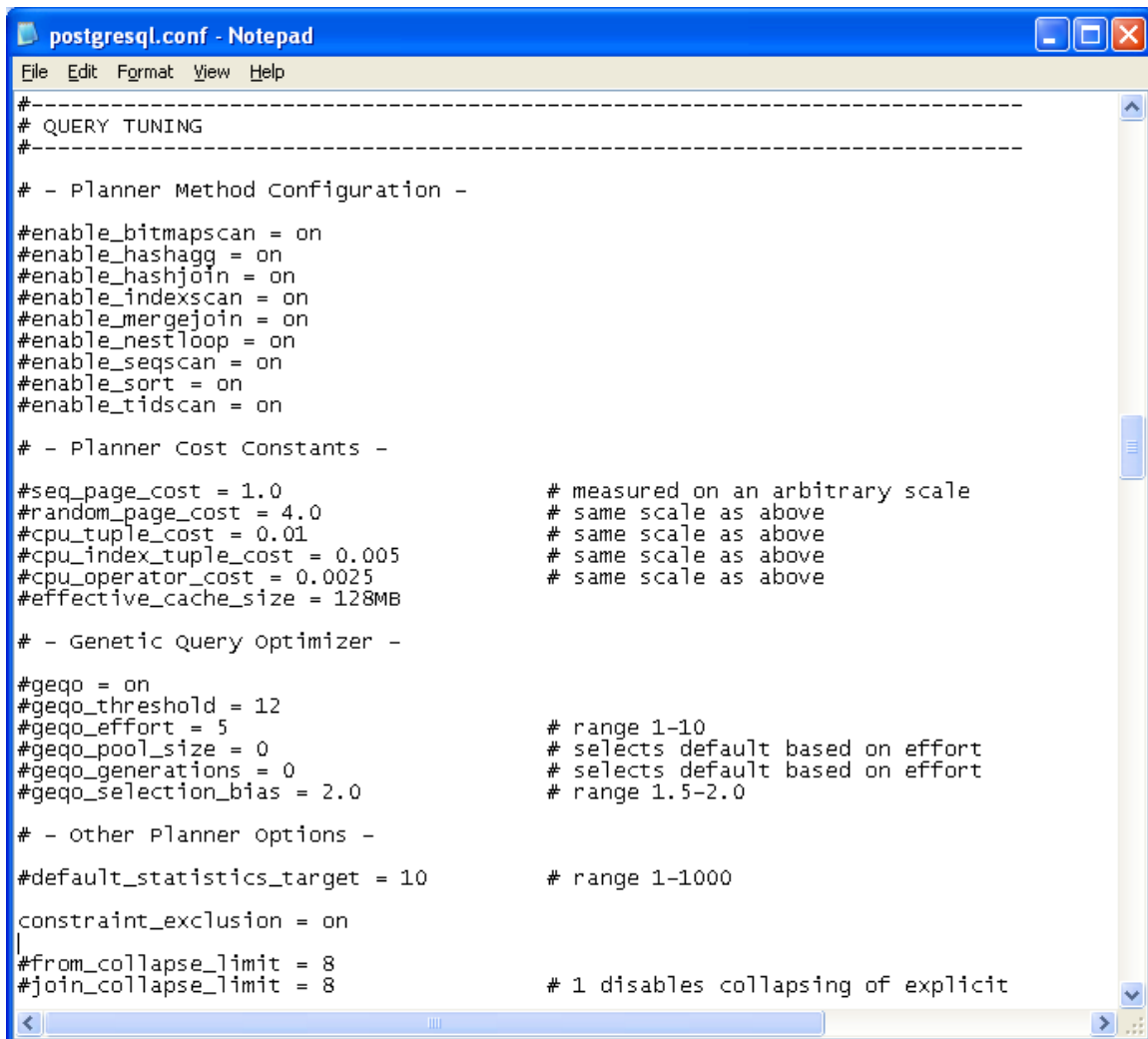
Creating Partitioned Tables with Triggers

In this example we create a master table named `sales`, and partition the data stored in that table into two partitions based on the value of the `org` column. The following example demonstrates how to use *triggers* instead of rules to implement partitioning.

In the example, a trigger intercepts an `INSERT` statement that adds data to a table, and redirects the `INSERT` statement to add data to the child/partitioned tables.

Step 1 (IMPORTANT): Enable `constraint_exclusion`.

Since Postgres Plus uses the concept of constraint exclusion to enable partition boundary checking, it is critical that you set the following parameter in the `postgresql.conf` file. You can edit the `postgresql.conf` file by opening the Start menu and navigating to the Postgres Plus menu. Choose Expert Configuration and then Edit `postgresql.conf` to open the `postgresql.conf` file (shown below). Scroll down to the `QUERY TUNING` section and remove the pound sign (#) from in front of the `constraint_exclusion` entry. Set the `constraint_exclusion` parameter to `on`, and save the `postgresql.conf` file before exiting.



```
postgresql.conf - Notepad
File Edit Format View Help
#-----
# QUERY TUNING
#-----

# - Planner Method Configuration -
#enable_bitmapscan = on
#enable_hashagg = on
#enable_hashjoin = on
#enable_indexscan = on
#enable_mergejoin = on
#enable_nestloop = on
#enable_seqscan = on
#enable_sort = on
#enable_tidscan = on

# - Planner Cost Constants -
#seq_page_cost = 1.0           # measured on an arbitrary scale
#random_page_cost = 4.0       # same scale as above
#cpu_tuple_cost = 0.01        # same scale as above
#cpu_index_tuple_cost = 0.005  # same scale as above
#cpu_operator_cost = 0.0025    # same scale as above
#effective_cache_size = 128MB

# - Genetic Query Optimizer -
#geqo = on
#geqo_threshold = 12
#geqo_effort = 5               # range 1-10
#geqo_pool_size = 0            # selects default based on effort
#geqo_generations = 0          # selects default based on effort
#geqo_selection_bias = 2.0     # range 1.5-2.0

# - other Planner options -
#default_statistics_target = 10 # range 1-1000
constraint_exclusion = on
|
#from_collapse_limit = 8
#join_collapse_limit = 8      # 1 disables collapsing of explicit
```

After setting this parameter, you must signal the server to reload the configuration file using the `pg_ctl` utility. Open a terminal window, and navigate to:

```
C:\PostgresPlus\8.3R2AS\dbserver\bin
```

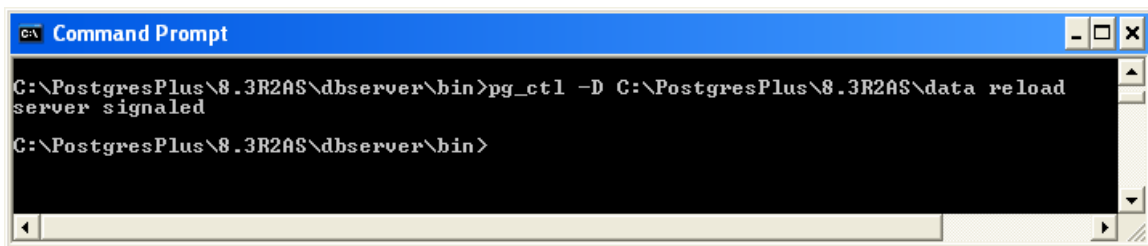
Enter the following command:

```
pg_ctl -D <datadir> reload
```

<datadir> is the full path to your data directory.

By default, the data for a standard installation of Postgres Plus Advanced Server is stored in C:\PostgresPlus\8.3R2AS\data. The location of the data directory may vary depending on your version of Postgres Plus and the installation options chosen at install time.

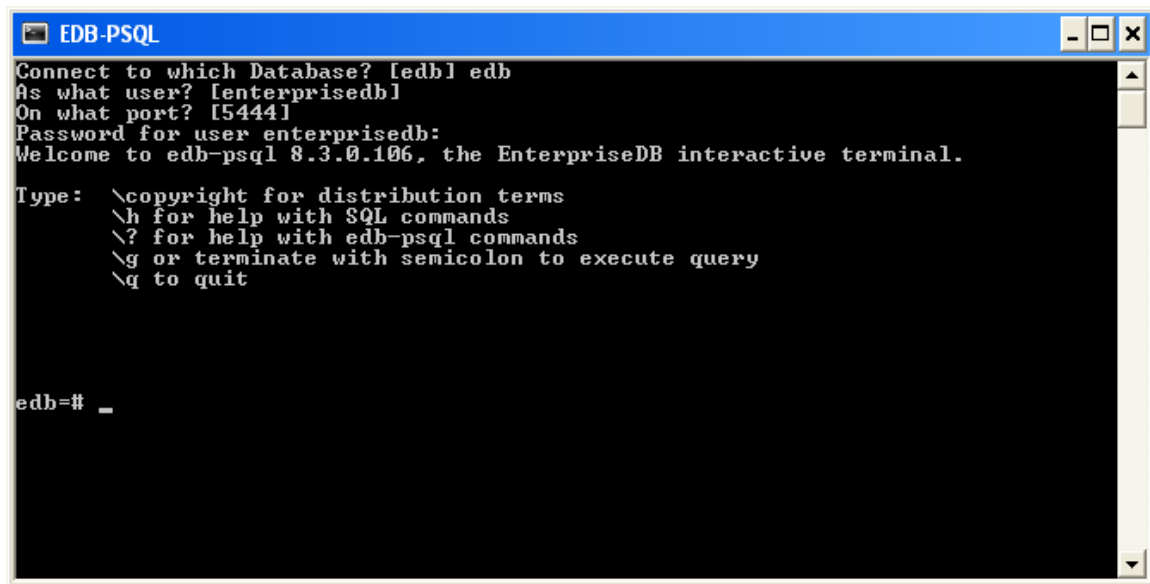
The following screenshot shows the process of reloading the configuration file in Windows:



Step 2: Create the parent table.

Note: You can use the PSQL command line to enter the rest of the commands used in this tutorial.

You can access the PSQL command line by opening the Start menu and navigating to the Postgres Plus menu. Choose Run SQL command line, and then EDB-PSQL to open the EDB-PSQL window:



When prompted, enter your connection information. When you've specified your connection information (Database name, user name, password and port number), Postgres Plus opens the PSQL command line. Enter the command:

```
CREATE TABLE sales(org int, name varchar(10));
```

Step 3: Create the child (partitioned) tables.

Use the PSQL command line to create the child tables:

```
CREATE TABLE sales_part1
(CHECK (org < 6))
INHERITS (sales);

CREATE TABLE sales_part2
(CHECK (org >=6 and org <=10))
INHERITS (sales);
```

Step 4: Create the triggers.

This example creates a trigger on the parent table that intercepts any operation against the parent table and instead, performs the action on the child/partition tables.

First, create the trigger function:

```
CREATE OR REPLACE FUNCTION sales_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
    IF ( NEW.ORG < 6) THEN
        INSERT INTO sales_part1 VALUES(NEW.*);
    ELSIF ( NEW.ORG >= 6 AND NEW.ORG <11) THEN
```

```

        INSERT INTO sales_part2 VALUES (NEW.*);
    ELSE
        RAISE EXCEPTION 'Organization out of range.  Fix
the sales_insert_trigger() function!';
    END IF;
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;

```

Then create the trigger:

```

CREATE TRIGGER insert_sales
    BEFORE INSERT ON sales
    FOR EACH ROW
    EXECUTE PROCEDURE sales_insert_trigger();

```

Step 5: Add data to the new table.

Use the PSQL prompt to add rows to the new table.

```

INSERT INTO sales VALUES (1, 'Craig');
INSERT INTO sales VALUES (2, 'Mike');
INSERT INTO sales VALUES (3, 'Michelle');
INSERT INTO sales VALUES (4, 'Joe');
INSERT INTO sales VALUES (5, 'Scott');
INSERT INTO sales VALUES (6, 'Roger');
INSERT INTO sales VALUES (7, 'Fred');
INSERT INTO sales VALUES (8, 'Sam');
INSERT INTO sales VALUES (9, 'Sonny');
INSERT INTO sales VALUES (10, 'Chris');

```

Step 6: Confirm that the data was added to the parent table.

```

postgres=# select * from sales;
 org |   name
-----+-----
   1 | Craig
   2 | Mike
   3 | Michelle
   4 | Joe
   5 | Scott
   6 | Roger
   7 | Fred
   8 | Sam
   9 | Sonny
  10 | Chris

```

As in our previous example (*Creating Partitioned Tables with Rules*), you can see that queries on the parent table return a result set that includes all of the rows in both of the

child partitions.

You can see which rows are stored in each child partition with the following commands:

```
postgres=# select * from sales_part1;
 org |   name
-----+-----
   1 | Craig
   2 | Mike
   3 | Michelle
   4 | Joe
   5 | Scott

postgres=# select * from sales_part2;
 org | name
-----+-----
   6 | Roger
   7 | Fred
   8 | Sam
   9 | Sonny
  10 | Chris
```

Again, the EXPLAIN plan demonstrates that the server reads each child table to satisfy an unconstrained query on the parent table:

```
postgres=# explain select * from sales;
              QUERY PLAN
-----
Result  (cost=0.00..63.90 rows=3390 width=42)
-> Append  (cost=0.00..63.90 rows=3390 width=42)
      -> Seq Scan on sales  (...)
      -> Seq Scan on sales_part1 sales  (...)
      -> Seq Scan on sales_part2 sales  (...)
```

Of course, you still need to add triggers for UPDATE and DELETE operations. Creating indexes on each of the partitions is also advisable to achieve 'local partitioned' indexes.

Conclusion

In this Quick Tutorial, we demonstrated how to use partitioned tables with Postgres Plus to provide more effective storage of large amounts of data and improve query performance.

The next topic you might be interested in is *Enabling Row Movement*. Enabling row movement allows the partition key to be updateable, and automatically moves the updated record to the appropriate partition. That topic is covered in another Quick Tutorial from EnterpriseDB.

If you haven't checked out EnterpriseDB's Evaluation Guide, please do so; it may help you move onto the next step in your Postgres Plus evaluation. The guide can be accessed at:

<http://www.enterprisedb.com/learning/documentation.do>

You should now be able to proceed with a Technical Evaluation of Postgres Plus.

The following resources should help you move on with this step:

- [Postgres Plus Technical Evaluation Guide](#)
- [Postgres Plus Getting Started resources](#)
- [Postgres Plus Quick Tutorials](#)
- [Postgres Plus User Forums](#)
- [Postgres Plus Documentation](#)
- [Postgres Plus Webinars](#)