Extending PostgreSQL: Types, Functions, Operators, and Aggregates

Due Date- 11-Oct-2007
Group :- 2 (Two Students)

In this project you will create a new user defined type in PostgreSQL. You will have to d efine the new type currency, add new functions that manipulate currency, define operat ors for currency that will allow you to build an index on currency attributes, and you will create aggregate functions for currency. Finally, you will write some queries that use the functions and aggregates you defined.
The functions you implement will use C and PL/pgSQL. You will need to read parts of th e "PostgreSQL 7.2.1 Documentation" to be able to complete the project .

Make sure your code runs smoothly on your account in intel Lab.
Steps in this Project:
1. Define a New User Defined Type
2. Implement User Defined Functions
3. Implement User Defined Operators
4. Implement User Defined Aggregates
5. Define an Index on the New Type
6. Write Queries for the New Type

Define a New User Defined Type

In this section you will define a new user defined type currency. The new type represent s a value in a specific currency for example '150 INR', or '12.5 EUR'. Create a PostgreS QL database. Create the following table in that database:

```
CREATE TABLE currencies(
currency_type varchar(3),
conversion_rate float4 NOT NULL, units/INR
last_updated timestamp NOT NULL
DEFAULT CURRENT_TIMESTAMP,
PRIMARY KEY(currency_type)
);
```

Make a comma
delimited file which have currency data. Name it Currency.data. Load it in to currencies t able.

Before continuing you should read sections II:1113 of the"PostgreSQL 7.2.1 Programm er's Guide"
You can find the complete source files for this project which can help you in src/tutorial. When implementing your function use version0 calling conventions for C language func tion (There is version1 convention also which you will not use).
Following the example from the PostgresSQL documentation:

- Define the C structure for currency.

- Define the C input and output functions for currency functions. The string representation of currency is “<value> <currency_type>” (see the examples above). Make sure to raise an error (using the elog function) and to return null whenever the format of the argument for the input function is wrong, or the specified currency type is not one of the types listed in currencies.data (you can assume this list is static).

- In SQL, define the input and output functions using the CREATE FUNCTION command.

- Declare the new data type, using the CREATE TYPE command.

You should write all your C code in one file currency.c, and one file currency.sql for all your SQL code. Make sure to include "postgres.h", and to use Postgres’ palloc() and pfree() functions for memory allocation.
After completing step 1, you should be able to create new currency attributes. Try to create a test table with a currency attribute, and insert and select from the table, to test your implementation.

Implement User Defined Functions

Now you will implement some useful functions that operate on currency:

- currency_gettype(currency) RETURNS varchar –
  returns the type of a currency ('INR', ‘EUR’, etc.)

- currency_getvalue(currency) RETURNS real –
  returns the value of a currency (150, 12.5, etc.)

- currency_convert(currency, varchar) RETURNS currency –
  returns the original currency converted to a new currency type (based on the currencies table). Since this function needs to access a table in your database, you should consider implementing it in PL/pgSQL.

Try to convert currencies from the test table you created in step 1.

Implement User Defined Operators

In this step you will implement operators that operate on currency.
The operators you need to implement are:

- Arithmetic operators: +, , *, /

- Boolean operators: =, >, <, >=, <=, !=

- All the operators should use the market value (i.e. the actual value based on the current exchange rate) of each currency.

- The arithmetic operators should return a currency of the same type as the left argument. For * and / the right argument should be of type real. You might find it useful (not required) to add another * operator where the left argument is of type real and the right argument is of type currency.

Try to write some queries on your test table, using the new operators. See what happens when you combine the operators with the conversion function from step 2 (e.g., make sure that two currencies that were equal before a conversion are still equal after a conversion).

Implement User Defined Aggregates

Try to write a query that counts all your items in your test table. Now try to write a query that sums the market values of all your items in your test table. As you can see, COUNT can operate on all data types, but the default SUM aggregate cannot operate on new data types. In this section you will overload SQL aggregates, to support currency.
You need to implement the following aggregates: SUM, AVG, MIN, MAX. The interpretation of these aggregates should use the market value of the currencies.

Define an Index on the New Type

Try to create an index on a currency attribute in your test table. As you see, you need to add one function and update some system tables to allow a btree index to be created on a new data type. Implement the comparison function, and update the system tables to support building btree indexes on currency attributes. The index should be sorted by market value. you can find the complete source code in complex.source in src/tutorial.
Try now to create an index on your test table. Since currency exchange rates are dynamic your index might become invalid, once the rates are updated in the currencies table. To deal with that you need to rebuild the index after each update of the exchange rates. You can do that, by either dropping and recreating the index, or by invoking the REINDEX command.

Write Queries for the New Type

This section tests your implementation of all the previous sections. Add the following tables to your database:

```
CREATE TABLE products(
product_id int,
product_name varchar(100) NOT NULL,
marginal_cost currency NOT NULL,
PRIMARY KEY(product_id)
);
```

```
CREATE TABLE product_tree(
parent_product_id int,
child_product_id int,
quantity int NOT NULL,
assembly_cost currency NOT NULL,
PRIMARY KEY(parent_product_id, child_product_id),
FOREIGN KEY(parent_product_id) REFERENCES products(product_id),
FOREIGN KEY(child_product_id) REFERENCES products(product_id)
);
```

A sample instance of the products table:
(product_id, product_name, marginal_cost){ (1, Prod_1 ,'150 INR'), (2, Prod_2, '100 IN R'),
(3, Prod_3, '50 INR')}
A sample instance of the product_tree table:
(parent_product_id, child_product_id, quantity, assembly_cost)={(1, 2, 12, '20 INR'),(1, 3, 40, '10 INR')}
Implement the following functions in SQL:

- product_cost(int, varchar) RETURNS currency –

returns the total cost of a product with the product_id specified in the first argument, in the curr ency type indicated by the second argument. The total cost of a product includes its marginal c ost plus the total cost of all its children times the number of children prducts plus assembly cos ts. Note that child products might have children themselves (assume no cycles, but do not ass ume any limit on the number of levels of a product tree). In addition, each product in the tree c ould be imported from a different country, therefore having a cost in a different currency type. For example using the above instances of the two tables, the following values are the correct r esult of the function:

- sort_products() RETURNS SETOF products –

returns the list of all products sorted by their total cost in descending order.

- max_product(currency) RETURNS products –

returns the most expensive product that costs less then the provided argument.

You have to submit a tgz file comprising currency.c, currency.sql and README. Make s ure to include all sql and pgsql code in currency.sql .