

## Automating Cross-Tab Views with PostgreSQL

Published [Jul 17, 2007 @ 15:06:55](#)

*Please note that this article needs some editing, but I thought I would put up the information while it was fresh in my head :)*

### Introduction

Cross tabulation (or "cross tabs", as they are often referred to) displays the joint distribution of two or more variables. It is often the case in my database projects that I need to allow the end-user to create and define "variables" or "configuration parameters" at run-time without having to change anything in the code.

### The Problem

One example of this is the ability to create user defined variables for an application. So let's say you're building a piece of contact management software. The customer insists on being able to define their own fields for the contacts at run-time.

The amateur programmer might not be able to handle such a requirement. They might approach it with something ugly like this: Create a base table for "user defined values", whereby the column names are the "variable" names that the customer can put in, and as they customer actually creates these variables, the application simply does an "alter table" on it and adds columns to the table. Not only does this create a very "wide" table, but you're also limited on the names of the variables you can use and your indexing options.

### Where to go from here...

The more experienced programmer will likely create a few tables to accomplish the task:

- A table for the basic contact (to store "obvious" values, such as "name")
- A table to define the user fields ("UserDefinedField")
- A table to store the values of each field-per-user

( See attached erd01.jpg and create\_postgresql.sql )

Assuming that the user is then able to get all of this working within the application as they want (which is another article all together), the big concern then becomes being able to display this data all at once. Often the client asks to be able to export these types of values to an excel spreadsheet for offline perusing or importing into other applications.

Enter the cross-tab query.

---

### Cross-Tabs in PostgreSQL

What we basically want to do here is "rotate" the values in the UserDefinedField table and turn them into the "columns" of a cross-tabulation of the values from ContactUserDefinedField table, with one row per contact. The way you do this in PostgreSQL is with a series of case statements and some grouping trickery. This is probably best explained with a sample query:

```
SELECT c.ContactID,
       MAX(
         CASE
           WHEN cudf.UserDefinedFieldID = 1 THEN
             cudf.Value
           ELSE ''
         END
       ) as "Variable 1",
       MAX(
         CASE
```

```

        WHEN cudf.UserDefinedFieldID = 2 THEN
            cudf.Value
        ELSE ''
    END
) as "Variable 2"
FROM Contact c join ContactUserDefinedField cudf using (ContactID)
    join UserDefinedField udf using (UserDefinedFieldID)
GROUP BY c.ContactID;

```

Notice that the key here is that you're doing a literal comparison of `cudf.UserDefinedFieldID` and the actual ID (1) value of "variable 1", and the same with variable 2. This means that in your `UserDefinedField` table, you have two rows:

```

ID | Value
-----
1  | Variable 1
2  | Variable 2

```

You can quickly see that manually writing this query isn't a possibility, as the whole point here is that the client can define the variables at run-time. The answer to this is actually quite simple and elegant:

1. Create a view that is, in effect, the above query.
2. Have the above view dynamically generated from within the database so that when a user defined field is added, the view is automatically re-created!

### **usp\_CreateView\_v\_ContactCrossTab**

Below is a sample function that you can modify to dynamically create a view that is a "rotated" version of your table.

```

CREATE OR REPLACE FUNCTION usp_CreateView_v_ContactCrossTab( )
RETURNS VOID
AS $func$
DECLARE
    drop_view varchar;
    dynsql varchar;

    has_customdata int4;

    r record;
BEGIN
    has_customdata = 0;
    drop_view = 'drop view v_ContactCrossTab';
    dynsql = 'create view v_ContactCrossTab as SELECT c.ContactID, '

    for r in
        select * from UserDefinedField udf order by FieldName
    loop
        has_customdata = 1;
        -- raise notice '%', r.userfieldname;
        dynsql = dynsql || 'MAX(CASE WHEN cudf.UserDefinedFieldID='''
            || replace(r.UserDefinedFieldID, ''', ''''') || ''' THEN cudf.Value ELSE '''' END) AS "'
            || replace(r.FieldName, ''', ''''') || "', '";
    end loop;

    -- trim off ending ','
    dynsql = substring(dynsql, 0, length(dynsql)-1);

    dynsql = dynsql || ' FROM Contact c join ContactUserDefinedField cudf using (ContactID)
        join UserDefinedField udf using (UserDefinedFieldID)
        GROUP BY c.ContactID;';

```

## Automating Cross-Tab Views with PostgreSQL

```
-- do we have the view (so do we need to drop it?)
for r in
    select * from pg_views where lower(viewname) = lower('v_ContactCrossTab')
loop
    execute drop_view;
end loop;

if has_customdata = 1 then
    -- raise notice 'SQL To Execute: %', dynsql;
    execute dynsql;
end if;

END
$func$
LANGUAGE 'plpgsql';
```

Once you have created this function, you can have it automatically (re)build your view simply by calling it:

```
select usp_CreateView_v_ContactCrossTab();
```

If you create a trigger and have it call this function automatically on insert/update/delete of the UserDefinedFields table, this view will truly become dynamic.

You can now query just like any other table:

```
select *
from Contact join v_ContactCrossTab using(ContactID);
```