Table Of Contents

A Product of Paragon Corporation

Happy New Year

Welcome to the January 2008 Issue of Postgres OnLine Journal Magazine. In this issue we will have a special feature *PostgreSQL 8.3 Cheatsheet* to commemorate the upcoming PostgreSQL 8.3 release and the new year. This cheat sheet will look similar in format to the Postgis Cheatsheet and will cover standard PostgreSQL features as well as new features added to the 8.3 release.

In future issues we hope to provide similar cheatsheets that highlight certain PostgreSQL advanced and specialty features. Any thoughts on what topics people would like to see in a cheatsheet are welcome.

Other interesting topics that will be covered in this issue to name a few

- Part 2 of our PostgreSQL Anatomy Series. We shall delve into the details of the database structure.
- CrossTab queries using TableFunc contrib
- Using Open Office Base with PostgreSQL
- Setting up PgAgent and using it for scheduled backups.

On another note - check out Andrew Dunstan's, minimum update Trigger. It will be nice to see this make it into the PostgreSQL 8.4 release. Granted we haven't had much of a need of this feature, but when you need it, it comes in very handy as demonstrated in Hubert Lubaczewski's related article Avoiding Empty Updates. We remember the first time we started working on MySQL a long long time ago - MySQL had this built in, but you couldn't turn it off. In certain situations such as when you have triggers this feature is often a misfeature. Granted I guess there are only a few cases where having this automatically on could be annoying especially when all the other Databases you work with don't do this and there is probably some overhead involved with checking which may not always outweigh the update/logging cost. Any rate as far as check-off lists goes for people who consider this a feature, it will be nice to cross this off the list as one reason why one would choose MySQL over PostgreSQL and better yet in PostgreSQL it is optional.

Back to Table Of Contents

Sun Purchasing MySQL and PostgreSQL advances

MySQL and Sun?

We just read that Sun is purchasing MySQL for a little under a billion. We are a little shocked and not quite sure what to make of it or how this affects Sun's investment in PostgreSQL. Further comments on the deal on Jignesh Shah's blog and Josh Berkus blogs. Jignesh and Josh both work at Sun and do PostgreSQL work as well.

Couple of random thoughts

- First, better Sun than Oracle. The thought of Oracle eating up MySQL has always been rather disturbing to us especially since we do a fair amount of MySQL consulting and don't care much for Oracle as a company. I suppose it could still happen.
- Given the fact that Sun is a large contributor to the PostgreSQL project, does this mean PostgreSQL fans can't make fun of MySQL anymore? Are we like friends now? This could take away some fun and add a little fun at the same time.
- Will this mean MySQL will have no qualms of using PostgreSQL underlying storage engine and what would it be called? - **MyPost**

Overall we think the move should prove positive for both camps.

PostgreSQL 8.3 really around the corner

8.3 is now on release candidate 8.3 RC1 and as Bruce Momjian noted, it looks like there might be an RC2.

We've been playing around with the 8.3 betas and RCs and really like the integrated Full Text Indexing and XML features. The new features make it possible to do a quickie REST service-based application. In the next issue of this journal, we hope to demonstrate creating REST services using 8.3 with server side - (PHP and/or ASP.NET) and front-end Adobe FLEX. We would have liked to demonstrate SilverLight/MoonLight as well, but we want to wait till Silverlight 2.0 hits release. We'll try to use the Pagila demo database for the upcoming demo app as Robert Treat has suggested.

Back to Table Of Contents

Stored Procedures in PostgreSQL *Intermediate*

Question: Does PostgreSQL support stored procedures?

Short Answer: Sort Of as Stored functions.

Longer Answer:
By strict definition it does not. PostgreSQL as of even 8.3 will not support the **Create Procedure** syntax nor the Call Level calling mechanism that defines a bonafide stored procedure supporting database (this is not entirely true), since EnterpriseDB does suport CREATE PROCEDURE to be compatible with Oracle. In PostgreSQL 8.4, this may change. Check out Pavel Stehule: Stacked Recordset and Pavel Stehule: First Real Procedures on PostgreSQL for details.

For all intents and purposes, PostgreSQL has less of a need for CREATE PROCEDURE than other databases aside from looking more like other databases. For example in SQL Server -> 2005 - although you can write functions that return tables and so forth, you have to resort to writing CLR functions marked as unsafe to actually update data in a stored function. This gets pretty messy and has its own limitations so you have no choice but to use a stored procedures, which can not be called from within an SQL query. In MySQL 5.1 the abilities of functions are even more limiting - they can't even return a dataset. In PostgreSQL, you can write a function marked as VOLATILE that updates data and that can do all sorts of wacky things that are useful but considered by some to be perverse such as the following:

```
SELECT rule_id, rule_name, fnprocess_rule(rule_id) As process_result
FROM brules
WHERE brules.category = 'Pay Employees'
ORDER BY brules.rule_order
```

Another thing stored procedures can usually do that functions can not is to return multiple result sets. PostgreSQL can simulate such behavior by creating a function that returns a set of *refcursors*. See this .NET example Getting full results in a DataSet object: Using refcursors way down the page, that demonstrates creating a postgresql function that returns a set of refcursors to return multiple result sets using the Npgsql driver.

Prior to PostgreSQL 8.1, people could yell and scream, but PostgreSQL doesn't support Output Parameters. As weird as it is for a function to support such a thing, PostgreSQL 8.1+ do support output parameters and ODBC drivers and such can even use the standard **CALL** interface to grab those values.

At a glance it appears that PostgreSQL functions do all that stored procedures do plus more. So the question is, is there any reason for PostgreSQL to support bonafide stored procedures aside from the obvious **To be more compatible with other databases** and not have to answer the philosophical question, **But you really don't support stored procedures?**.

There must be some efficiency benefits to declaring something as a store procedure and having it called in that way. Not quite sure if anyone has done benchmarks on that. So for the time being PostgreSQL functions have the uncanny role of having a beak like a duck and the flexibility of a beaver, but having the makeup of a Platypus.

Back to Table Of Contents

SQL Math Idiosyncracies *Beginner*

Question: What is the answer to SELECT 3/2?

Answer: In integer math, it is 1. A lot of people especially those coming from MySQL or MS Access backgrounds are surprised to find out that in PostgreSQL
`3/2 = 1`
. They view this as some sort of bug.

In actuality, the fact that 3/2 = 1 and 1/3 = 0 is part of the ANSI/ISO-SQL standard that states mathematical operations between two values must be of the same data type of one of the values (not necessarily the same scale and precision though). This is not some idiosyncracy specific to PostgreSQL. If you try the same operation in SQL Server, SQLite,FireBird, and some other ANSI/ISO SQL compliant databases, you will get the same results. So it seems MySQL and MS Access are the odd-balls in this arena, but arguably more practical.

Why is this an SQL Standard? We haven't found any definitive answer to that, but we have our guesses. Our guess is because it is less ambiguous (more precise) and speedier processor wise to only offer the level of accuracy specifically requested for. In terms of standards and a lot of domains (e.g. Engineering), precision is more important than accuracy. In the case of 3/2 it is not quite obvious the benefit, but say you have 1/3. MySQL displays that as .3333 (although internally its probably storing 0.333333...), MS Access displays it as 0.333333333333333. Is MS Access more right? Both are not completely accurate and its ambiguous how inaccurate they are. In the case of PostgreSQL and other ANSI/ISO databases its quite clear how accurate. They very precisely discard the remainder.

There is one particular behavior in PostgreSQL that seems somewhat contradictory to the above, and that is the way it treats Averages. It returns averages in much the same way as MySQL where as something like SQL Server or SQLite returns a truncated integer average when averaging integers. For example, lets say you have a table of all integers. If you do an Average e.g.

```
--Here we are using a more portable example
--instead of our preferred generate_series approach
--so it can be tested on multiple database platforms
CREATE TABLE dumnum(num integer);
INSERT INTO dumnum(num)
        VALUES(1);
INSERT INTO dumnum(num)
        VALUES(2);

SELECT AVG(num) as theavg, AVG(CAST(num As numeric(10,3))) as theavgm,
SUM(num)/COUNT(num) As intavg,
        4/7 As intmath, 4./7 As floatmath,
                CAST(4./7 As numeric(10,6)) as precmath,
        4.000/7 As floatmath2,
                CAST(4./7 As integer) As precintmath
FROM dumnum;

--For mysql the implementation of
--CAST is a little peculiar.
--Although MySQL happily accepts numeric and integer, int(11) as data types in table creation and
converts to decimal
--It doesn't appear to do the same in CAST (e.g. you can't use numeric or integer in CAST)
--so the above example doesn't work
--Use instead

SELECT AVG(num) as theavg, AVG(CAST(num As decimal(10,3))) as theavgm,
SUM(num)/COUNT(num) As intavg,
        4/7 As intmath, 4./7 As floatmath,
                CAST(4./7 As decimal(10,6)) as precmath,
        4.000/7 As floatmath2,
                CAST(4./7 As SIGNED) As precintmath
FROM dumnum;
```

Speaking of other databases - has anyone seen the FireFox extension for browsing and creating SQLite databases? It is extremely cute. The following tests on SQLite we ran using this FireFox SQLite management tool.

Running the above on PostgreSQL, SQL Server 2005, SQLite, FireBird, and MySQL yields the following

- PostgreSQL 8.2/8.3 RC1: 1.5000000000000000; 1.5000000000000000; 1; 0; 0.57142857142857142857; 0.571429; 0.57142857142857142857; 1 - Note when casting back to Int Postgres rounds instead of truncating.
- SQL Server 2005: 1; 1.500000; 1; 0; 0.571428; 0.571429; 0.571428; 0 (Casting back to integer SQL Server truncates)
- SQLite: 1.5; 1.5; 1; 0; 0.5714285714285714; 0.5714285714285714; 0.5714285714285714; 0 (The CAST to numeric is bizarre, but given SQLite's lax thoughts on data types - it simply ignores any CASTING it doesn't understand. For example you can say CAST(1 as boo) and it will happily do nothing. SQLite truncates similar to Microsoft SQL Server when casting back to integer.)
- Firebird: 1; 1.500; 1; 0; 0; 0.000000; 0.571; 0
  Evidentally Firebird pays attention to the number of decimals you place after your multiplier where as the others do not. Similarly when casting back to integer, Firebird follows the same behavior of truncating that SQL Server 2005, SQLite follow.
- MySQL 5: 1.5000; 1.5000000; 1.5000;0.5714 ;0.5714 ;0.571429;0.5714286 ;1 (MySQL does averaging the same way as Postgres with fewer significant digits and Casting also rounds just as Postgres. It violates the 3/2 rule as previously stated, but its behavior of CAST to decimal is in line with the other databases (except for SQLite).

In terms of the number of significant digits displayed, those are more presentational issues than actual storage so all the more reason to stay away from floating point values.

One can argue that PostgreSQL,SQLite, and MySQL are really not in violation of standards here when it comes to averaging, because after all the ANSI/ISO standard talks about operations between numbers to our knowledge, not functions. So presumably Averaging as a function is left up to the implementation discretion of the database vendor. Nevertheless it is still a bit disconcerting to witness these conflicting behaviors.

Given these disparities between databases, the best thing to do when dealing with operations between numbers is to be very precise and there are a couple of ways of doing this.

Here are some guidelines.

- When you care about precision don't cast to or use floats and doubles. Those introduce rounding errors not to mention the precision and representation in each Db is probably all over the place. Use numeric or decimal data type. Decimal and numeric are more or less the same in most databases and in SQL Server and Postgres decimal is just a synonym for numeric. Numeric doesn't exist in MySQL. According to Celko, the distinction in SQL-92 standard between the two is that "DECIMAL(s,p) must be exactly as precise as declared, while NUMERIC(s,p) must be at least as precise as declared". So I guess decimal would be preferable if supported and there was actually a difference. Its not perfect, but its less up to the whims of the database vendor except in the bizarre case of SQLite
- To not loose data, when dealing with integers, do a CAST or multiply by 1. or for optimum portability measure 1.0000 (how precise you want) first
- Do a final cast or round of your value after the initial cast to make sure you have the precision you want. It seems that PostgreSQL for example throws out this precision/scale info even when CASTING and then applying an operation, a second cast is needed to get the right precision. Keep in mind when CASTING PostgreSQL appears to round instead of truncate like the other databases (except MySQL). Example below to demonstrate.

```
SELECT CAST(x*1.0000/y As numeric(10,4)) As thepreciseavg,
x*1.00000/y As lessprecisebutmoreaccurate
FROM generate_series(1,4) As x, generate_series(3,10) As y
```

Needless to say the various different behaviors in databases trying to conform to some not so well-defined standard, leaves one feeling a little woozy.

Back to Table Of Contents

Deleting Duplicate Records in a Table *Intermediate*

Question:

How do you delete duplicate rows in a table and still maintain one copy of the duplicate?

Answer:

There are a couple of ways of doing this and approaches vary based on how big your table is, whether you have constraints in place, how programming intensive you want to go, whether you have a surrogate key and whether or not you have the luxury of taking a table down. Approaches vary from using subselects, dropping a table and rebuilding using a distinct query from temp table, and using non-set based approaches such as cursors.

The approach we often use is this one:

```
DELETE
FROM    sometable
WHERE   someuniquekey NOT IN
        (SELECT         MAX(dup.someuniquekey)
        FROM            sometable As dup
        GROUP BY        dup.dupcolumn1, dup.dupcolumn2, dup.dupcolum3)
```

We prefer this approach for the following reasons

1. Its the simplest to implement
2. It works equally well across many relational databases
3. It does not require you to take a table offline, but of course if you have a foreign key constraint in place, you will need to move the related child records before you can delete the parent.
4. You don't have to break relationships to do this as you would with drop table approaches

The above presumes you have some sort of unique/primary key such as a serial number (e.g. autonumber, identity) or some character field with a primary or unique key constraint that prevents duplicates. Primary candidates are serial key or OID if you still build your tables WITH OIDs.

If you don't have any of these unique keys, can you still use this technique? In PostgreSQL you can, but in other databases such as SQL Server - you would have to add a dummy key first and then drop it afterward. The reason you can always use this technique in Postgres is because PostgreSQL has another hidden key for every record, and that is the **ctid**. The ctid field is a field that exists in every PostgreSQL table and is unique for each record in a table and denotes the location of the tuple. Below is a demonstration of using this ctid to delete records. Keep in mind only use the ctid if you have absolutely no other unique identifier to use. A regularly indexed unique identifier will be more efficient.

```
--Create dummy table with dummy data that has duplicates
CREATE TABLE duptest
(
  first_name character varying(50),
  last_name character varying(50),
  mi character(1),
  name_key serial NOT NULL,
  CONSTRAINT name_key PRIMARY KEY (name_key)
)
WITH (OIDS=FALSE);

INSERT INTO duptest(first_name, last_name, mi)
SELECT chr(65 + mod(f,26)), chr(65 + mod(l,26)),
CASE WHEN f = (l + 2) THEN chr(65 + mod((l + 2), 26)) ELSE NULL END
FROM
        generate_series(1,1000) f
        CROSS JOIN generate_series(1,456) l;

--Verify how many unique records we have -
--We have 676 unique sets out of 456,000 records
SELECT first_name, last_name, COUNT(first_name) As totdupes
FROM duptest
GROUP BY first_name, last_name;

--Query returned successfully: 455324 rows affected, 37766 ms execution time.
DELETE FROM duptest
        WHERE   ctid NOT IN
        (SELECT         MAX(dt.ctid)
        FROM            duptest As dt
        GROUP BY        dt.first_name, dt.last_name);

--Same query but using name_key
--Query returned successfully: 455324 rows affected, 3297 ms execution time.
DELETE FROM duptest
        WHERE   name_key NOT IN
        (SELECT         MAX(dt.name_key)
        FROM            duptest As dt
        GROUP BY        dt.first_name, dt.last_name);

--Verify we have 676 records in our table
SELECT COUNT(*) FROM duptest;
```

A slight variation on the above approach is to use a DISTINCT ON query. This one will only work in PostgreSQL since it uses the DISTINCT ON feature of PostgreSQL, but it does have the advantage of allowing you to selectively pick which record to keep based on which has the most information. e.g. in this example we prefer records that have a middle initial vs. ones that do not. The downside of using the DISTINCT ON, is that you really need a real key. You can't use the secret **ctid** field, but you can use an oid field. Below is the same query but using DISTINCT ON

```
--Repeat same steps above except using a DISTINCT ON query instead of MAX query
--Query returned successfully: 455324 rows affected, 5422 ms execution time.
DELETE FROM duptest
        WHERE duptest.name_key
        NOT IN(SELECT DISTINCT ON (dt.first_name, dt.last_name)
        dt.name_key
       FROM duptest dt
       ORDER BY dt.first_name, dt.last_name, COALESCE(dt.mi, '') DESC) ;
```

Note: for the above if you want to selectively pick records say on which ones have the most information, you can change the order by to something like this

```
ORDER BY dt.first_name, dt.last_name, (CASE WHEN dt.mi > '' THEN 1 ELSE 0 END + CASE WHEN dt.
address > '' THEN 1 ELSE 0 END ..etc) DESC
```

Back to Table Of Contents

Setting up PgAgent and Doing Scheduled Backups *Beginner*

What is PgAgent?

PgAgent is a basic scheduling agent that comes packaged with PgAdmin III (since pre-8.0 or so) and that can be managed by PgAdmin III. PgAdmin III is the database administration tool that comes packaged with PostgreSQL. For those familiar with unix/linux cronjobs and crontab structure, PgAgent's scheduling structure should look very familiar. For those familiar with using Microsoft SQL Server Scheduling Agent or Windows Scheduling Tasks, but not used to crontab structure, the PgAdmin III Job Agent interface to PgAgent should look very welcoming, but the schedule tab may look a little unfamiliar.

PgAgent can run both PostgreSQL stored functions and sql statements as well as OS shell commands and batch tasks.

Why use PgAgent over other agents such as cronjob, Microsoft Windows Scheduled Tasks, or Microsoft SQL Server Agent?

For one thing, since PgAgent runs off of standard Postgres tables, you can probably more easily programmatically change jobs from it from within PostgreSQL sql calls that insert right into the respective PgAgent pga_job, pga_jobstep, pga_jobagent, pga_schedule tables to roll your own App integrated scheduler.

Compared to CronTab, PgAgent has the following advantages:

- You can have multiple steps for a job without having to resort to a batch script.
- You can have multiple schedules for a job without having to repeat the line.
- Is cross platform
- For running PostgreSQL specific jobs such as stored function calls or adhoc sql update statements etc. it is a bit easier granted the PostgreSQL account used is a super user or has sufficient rights to the dbs.

Compared to Windows Scheduled Tasks - PgAgent has the following advantages:

- You can go down to the minute level
- Have several steps per job
- Have multiple schedules per job
- Is cross platform
- For running PostgreSQL specific jobs such as stored function calls it is easier than using windows scheduled tasks.

Compared to SQL Server Agent - PgAgent has the following advantages:

- SQL Server Agent comes only with Microsoft SQL Server Workgroup and above so not an option say for people running SQL Server Express editions or no SQL Server install.
- Is cross platform

Some missing features in PgAgent which would be nice to see in later versions would be some sort of notification system similar to what SQL Server Agent has that can notify you by email when things fail and a maintenance wizard type complement tool similar to what SQL Server 2005 Maintenace Wizard provides that allows users to walk thru a set of steps to build automated backup/DB Maintenance tasks. This is a bit tricky since it would need to be cross-platform. Granted the job history display in PgAdmin that provides success and time taken to perform task is a nice touch and makes up for some of this lack and you can always roll your own by running some monitor to check the job event logs.

How to install PgAgent

Note the docs describe how to install PgAgent: http://www.pgadmin.org/docs/1.8/pgagent-install.html, but the example to install it in a db called PgAdmin seems to send people off in the wrong direction. We shall highlight the areas where people most commonly screw up in installation, but for master reference, refer to the official PgAgent install docs listed above.

While you can install PgAgent in any database, to our knowledge, you can only administer it via PgAdmin III if it is installed in the maintenance database which is usually the database called **postgres**. For ISPs, having the ability to install it in any db and rolling your own agent interface may be a useful feature.

Other note that is not explicitly stated, but is useful to know: PgAgent need not be installed on the same Server/Computer as your PostgreSQL server. It just needs to have the pgAgent files, which you can get by installing PgAdmin III or copying over the necessary files. PgAgent service/daemon also needs necessary access to the PostgreSQL database housing the job tables. If you are using it to backup databases to a remote server, the account it runs under will also need network file access or ftp access to the remote server. You can also have multiple PgAgent's running on different servers that use the same schedule tables.

To install PgAgent, there are basically three steps

1. Make sure you have plpgsql language installed in the **postgres** database. Which you do with the sql command runin postgres database.

```
CREATE TRUSTED PROCEDURAL LANGUAGE 'plpgsql'
       HANDLER plpgsql_call_handler
       VALIDATOR plpgsql_validator;
```

2. Run the PgAgent.sql using PgAdmin III or psql and run it in the db **postgres** - found in /path/to/PgAdmin III/1.8/scripts (on windows this is usually in "C:/Program Files/PgAdmin III/1.8/scripts"). This creates a schema catalog in the postgres database called **pgAgent** with the helper pgagent tables and functions.
3. Install the PgAgent server service/Daemon process: On windows - you run a command something like below - **the -u user is not the PostgreSQL user but the computer user that the PgAgent will be running under.**

```
"C:\Program Files\PostgreSQL\8.2\bin\pgAgent" INSTALL pgAgent -u postgres -p somepassword
hostaddr=127.0.0.1 dbname=postgres user=postgres
```

   After you install on Windows - you should go into Control Panel -> Administrative Tools -> Services - "PostgreSQL Scheduling Agent - pgAgent" -> and start the service. If the service doesn't start - most likely you typed the postgres computer account password in wrong. Simply switch to the Log On tab and retype the password or change to use a different account.

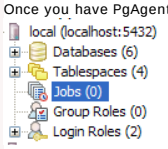
   Keep in mind - if you wish PgAgent to run scripts that require File Network access (e.g. copying files to network servers, you need to have the service run under a network account that has network access to those servers.

   On Unix/Linux systems - it varies how its installed. It is usually run under the root account and the line is added to startupscripts usually /etc/init.d or I think on MacOSX its */etc/xinetd.d*

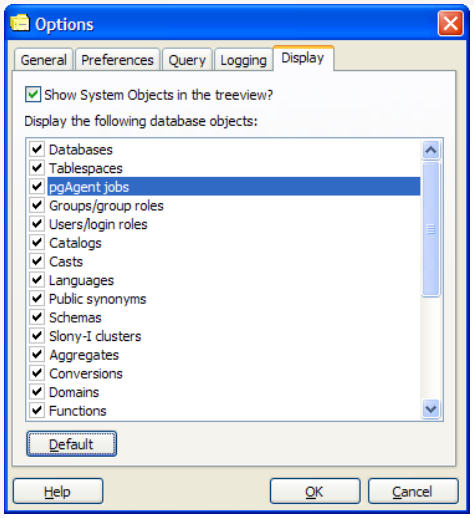
```
/path/to/pgagent hostaddr=127.0.0.1 dbname=postgres user=postgres
```

   Note: as the docs say - its probably best not to specify the password. Instead - you can set the postgres account to be trusted from server you have PgAgent installed on or use the ~pgpass approach.

Once you have PgAgent installed, and open/refresh PgAdmin III, you should see another section called **Jobs** that looks like below:



If per chance, you do not see the new Jobs icon, make sure that you have PgAgent jobs checked by going to File->Options->Display



Creating Backup Jobs

Creating backup jobs is done with a shell script of some sort. In Windows this can be done with a .bat file and specifying the file in the PgAgent job or by writing the command directly in the PgAgent job. In Linux/Unix - this is done with a .sh file and specifying that in the PgAgent job or writing the command directly in the PgAgent job.

Generally we go with a .bat or .sh file, because using a shell script allows you more granular control - such as backing up multiple databases or having a separately date named file for each daily backup.

Below is a sample batch script for Windows that backs up selected databases and then does a full Pg_dumpall as well

```
@echo off
REM - backup directory can be a file server share that the PgAgent windows service account has
access to
set BACKUPDIR="/path/to/backup/"
set PGHOST="localhost"
set PGUSER="postgres"
set PGBIN="C:/Program Files/PostgreSQL/8.2/bin/"
for /f "tokens=1-4 delims=/ " %%i in ("%date%") do (
 set dow=%%i
 set month=%%j
 set day=%%k
 set year=%%l
)

for /f "tokens=1-3 delims=: " %%i in ("%time%") do (
 set hh=%%i
 set nn=%%j
)

REM - It would be nice to use gzip in the pg_dumpall call (or if pg_dumpall supported compression
as does the pg_dump)
REM here as we do on the linux/unix script
REM - but gzip is not prepackaged with windows so requires a separate install/download.
REM Our favorite all purpose compression/uncompression util for Windows is 7Zip which does have a
command-line
%PGBIN%pg_dumpall -h %PGHOST% -U %PGUSER% -f %BACKUPDIR%fullpgbackup-%year%%month%.sql
%PGBIN%pg_dump -i -h %PGHOST% -U %PGUSER% -F c -b -v -f "%BACKUPDIR%db1-%year%%month%%day%%hh%.
compressed" db1
%PGBIN%pg_dump -i -h %PGHOST% -U %PGUSER% -F c -b -v -f "%BACKUPDIR%db2-%year%%month%%day%%hh%.
compressed" db2
```

Below is an equivalent Linux/Unix backup shell script

```
#!/bin/bash
#backup directory can be a file server share that the PgAgent daemon account has access to
BACKUPDIR="/path/to/backup"
PGHOST="localhost"
PGUSER="postgres"
PGBIN="/usr/bin"
thedate=`date --date="today" +%Y%m%d%H`
themonth=`date --date="today" +%Y%m`

#create a full backup of the server databases
$PGBIN/pg_dumpall -h $PGHOST -U $PGUSER | gzip > $BACKUP_DIR/fullbackup-$themonth.sql.gz

#put the names of the databases you want to create an individual backup below
dbs=(db1 db2 db3)
#iterate thru dbs in dbs array and backup each one
for db in ${dbs[@]}
do
        $PGBIN/pg_dump -i -h $PG_HOST -U $PGUSER -F c -b -v -f $BACKUPDIR/$db-$thedate.compressed
$db
done

#this section deletes the previous month of same day backup except for the full server backup
rm -f $BACKUPDIR/*`date --date="last month" +%Y%m%d`*.compressed
```

Save the respective above scripts in a (dailybackup.bat for windows pgagent) or (dailybackup.sh for Linux/Unix pgagent) file.

For bash unix scripts make sure it has unix line breaks (not windows) - you may use dos2unix available on most linux/unix boxes to convert windows line breaks to unix linebreaks. When saving as .sh make sure to give the .sh file execute rights using chmod on linux/unix. Also change the db1, db2 and add additional lines for other databases you wish to backup to the respective names of your databases and add additional as needed.

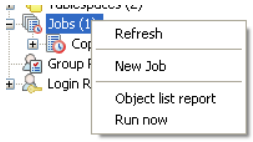
```
cd /path/toscriptfolder
dos2unix dailybackup.sh
chmod 771 dailybackup.sh
/path/toscriptfolder/dailybackup.sh  #this is to test execution of it
```

771 permissions gives execute rights to public and all rights (read,write,execute) to owner and group. Alternatively you could do 640 instead which would remove all rights from public, but then you will need to do a Change owner **chown** to change ownership to account you are running PgAgent under. Note the above script and commands we tested on a CentOS box so commands and script may vary if you are running on MacOSX or another Linux variant.

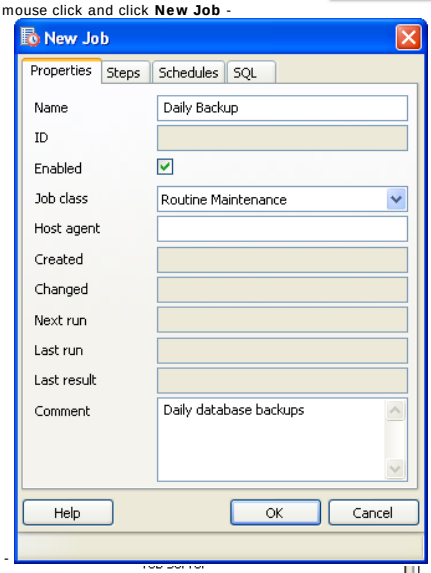
A couple of notes about the above which are more preferences than anything.

- We like to create a dump all backup which would contain all the databases and just overwrite it daily but keep one for each month. This is more for major disaster recovery than anything else.
- We prefer the Postgres Native Compressed format for our date stamped backups. The reason for that is with the pg_dump compressed format, it takes up less space, deals with binary objects well, and has the benefit that you can restore individual database objects for it. This is very useful in cases where someone screws up and they come back to you days or months later.
- You will note that the date stamp format we have included includes the Hour and would create a file something of the form - dbname-2008010102.compressed - the reason for that is that it sorts nicely by name and date of backup and if disk space was an issue, you could easily include a line that deletes say backups older than a month. Going down to the hour level allows us to quickly create emergency backups by clicking the **Run Now** on PgAdmin Jobs interface that wouldn't overwrite the current days backup.
- In practice we also like to have at least one of the backups ftped to a remote location and include that as part of the script and/or backed up to a remote server that has good connectivity with the pgagent server. This helps in cases of complete server failure. This step is not included here since its too OS and install specific to get into.

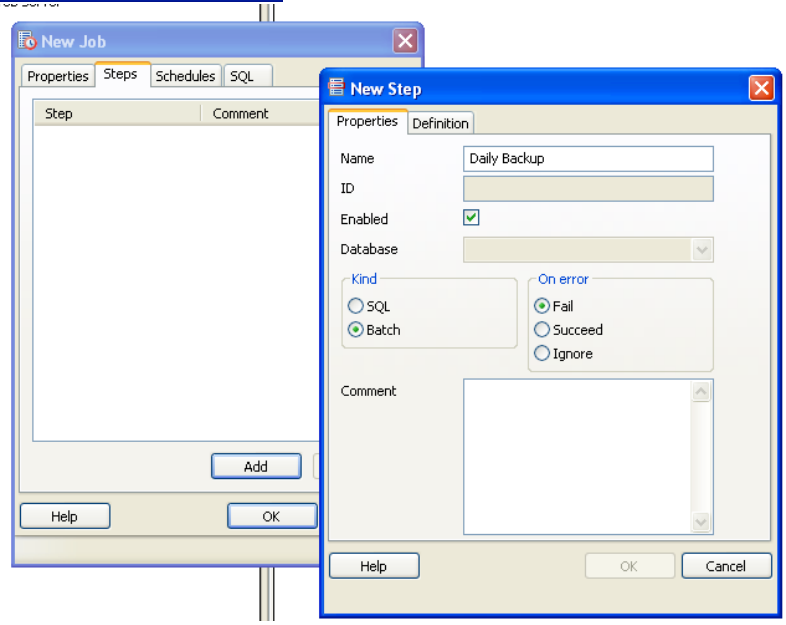
Next to create the PgAgent backup job follow the following steps.



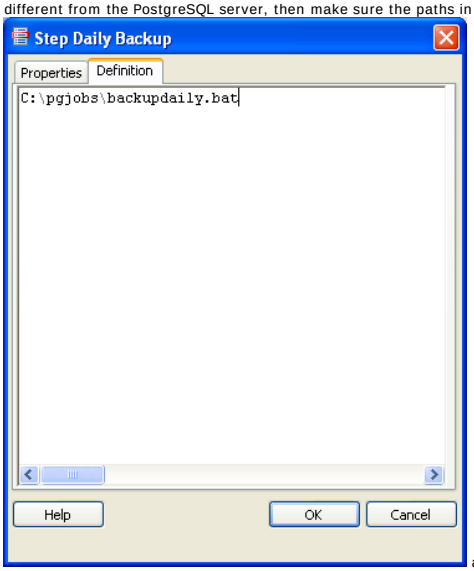
1. Open up PgAdmin - navigate to jobs section, right mouse click and click **New Job** -



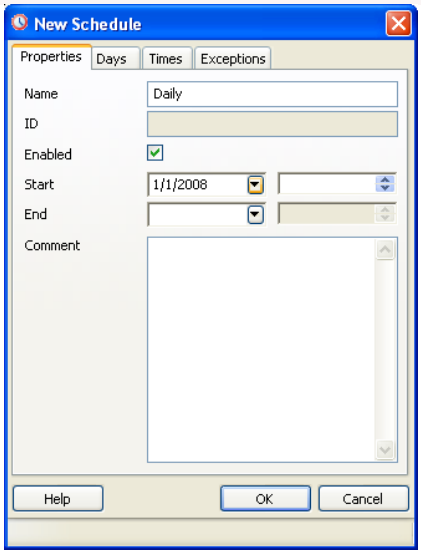
2. Fill in the properties tab as shown in this snapshot -



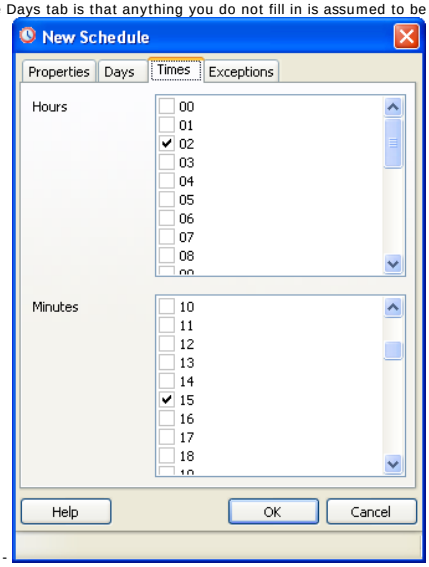
3. Switch to the Steps tab and select **Batch** and fill in details as shown -
4. Switch to the Definition tab and type in the path to the batch or sh file. Keep in mind the path is in context of the PgAgent service. So if you have PgAgent installed on a server that is different from the PostgreSQL server, then make sure the paths in your script and path to the file is set as if you were the PgAgent account on PgAgent server. As show here



and then click the **OK** button.



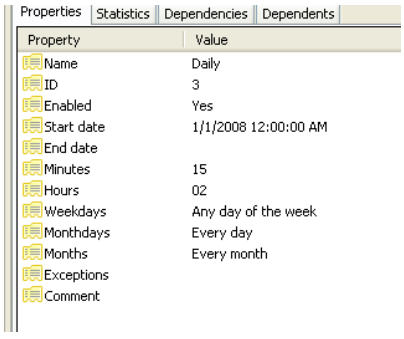
5. Next switch to the Schedules tab and click to add a Schedule.
6. Next Switch to Times tab. The reason we are skipping the Days tab is that anything you do not fill in is assumed to be **All** since we want all days, we leave that tab blank. This diagram



shows setting the backup time to be 02:15 AM every day -

Once the job is saved, the hierarchy in PgAgmin looks like the below snapshots

Clicking on the Daily Schedule Icon



Clicking on the respective objects in the Job Hierarchy such as a Step or schedule gives you detailed information about each of those. The statistics tab gives you details such as how long a step took, whether or not it succeeded or failed and when it was run.

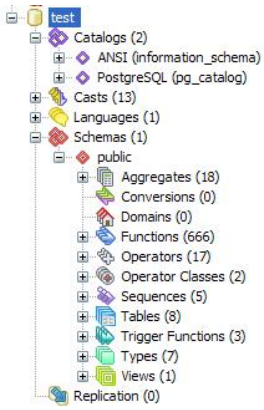
Keep in mind that while PgAgent is closely related to PostgreSQL and uses PostgreSQL for scheduling and logging, there isn't any reason you can not use it as an all-purpose scheduling agent. In fact we use it to backup MySQL as well as PostgreSQL databases, do automated web crawls, download remote backups etc. Using the SQL Job Type option, you can use it to run postgresql functions that rebuild materialized views, do other standard postgresql specific sql maintenance tasks, etc. On top of that PgAdmin provides a nice interface to it that you can use on any computer (not just the one running PgAgent).

Back to Table Of Contents

The Anatomy of PostgreSQL - Part 2 - Database Objects *Beginner*

In the first part of this series, The Anatomy of PostgreSQL - Part 1, we covered PostgreSQL Server object features. In this part, we shall explore the database and dissect the parts.

Here we see a snapshot of what a standard PostgreSQL database looks like from a PgAdmin interface.



1. Catalogs - these hold meta data information and built-in Postgres objects
2. Casts - control how Postgres casts from one datatype to another.
3. Languages - these are the languages you can define stored functions, aggregates and triggers in.
4. Schemas - logical containers for database objects.
5. Aggregates - holder for aggregate functions and custom built aggregate functions.
6. Conversions
7. Domains
8. Functions
9. Operators
10. Operator Classes
11. Operator Families - this is not shown in the diagram and is new in PostgreSQL 8.3
12. Sequences - objects for implementing autonumbers
13. Tables - self-explanatory but we'll cover the various object properties of a table such as indexes, rules, triggers, and constraints.
14. Trigger Functions - these are functions you create that get called from a PostgreSQL table trigger body.
15. Types - this is one of the key elements that qualifies PostgreSQL as an object relational database, the fact that one can define new data types.
16. Views - virtual tables

Catalogs and Schemas

Schemas are a logical way of separating a database. They are designed simply for logical separation not physical separation. In PostgreSQL each database has a schema called **public**. For sql server people, this is equivalent to SQL Server's **dbo** schema. The default schema search path in postgresql.conf file is *$user*, *public*. Below are some fast facts and comparisons

- **Note: $user is a place holder for the name of the logged in user - which means if there is a schema with the same name as the user, then that is the schema that is first searched when non-schema qualified queries are run and if such a schema exists, non-qualified schema create table etc. are put in the user's schema.**
- If a schema with the user's name does not exist, then non-schema qualified statements go against the **public** schema
- This is very similar in practice to Oracle and SQL Server 2000 in that the user's schema is the first searched. SQL Server 2000 didn't really have schemas, but had owners which behaved sort of like schemas.
- SQL Server 2005 is a little different in that default schemas can be designated for each user or user group.
- Unlike Oracle and SQL Server 2000, SQL Server 2005 and PostgreSQL do not make the restriction that a schema map directly to the name of a user. You can name schemas however you like without regard to if there is a user with that name.
- PostgreSQL does not allow designating a default schema for each user. The schema search path setting is globally set at the server service/daemon level in the postgresql.conf file and not part of the user's profile.

Catalogs is actually a prefabrication of PgAdmin to make this distinction of calling Schemas that hold meta-like information "Catalogs". First Catalogs is a misnomer and in fact in some DBMS circles, Catalogs are another name for databases so its a bit confusing, but then some people (such as Old world Oracle - thought of the Database as the server and each schema as a separate database. So its all very confusing anyway.). We like to think of schemas as sub-databases. One may ask what is the difference between a "PgAdmin catalog" and a schema. The short-answer, as far as PostgreSQL is concerned, there isn't a difference. A PgAdmin catalog is a schema. In fact as far as we can tell, the schemas *information_schema*, *pg_catalog*, and *pgagent* are hard-wired in the PgAdmin logic to be grouped in something called *Catalogs*.

The information_schema is a very important schema and is part of the ANSI standard, but is not quite so standard. It would be nice if all relational databases supported it, but they don't all do - MySQL 5, SQL Server (2000+), and PostgreSQL (7.4+) support them. Oracle and DB2 evidentally still don't, but there is hope. For the DBMS that support the information_schema, there are varying levels, but in all you can be pretty much assured to find *tables*, *views*, *columns* with same named fields that contain the full listings of all the tables in a database, listings of views and view definition DDL and all the columns, sizes of columns and datatypes.

The pg_catalog schema is the standard PostgreSQL meta data and core schema. You will find pre-defined global postgres functions in here as well as useful meta data about your database that is very specific to postgres. This is the schema used by postgres to manage things internally. A lot of this information overlaps with information found in the information_schema, but for data present in the information_schema, the information_schema is much easier to query and requires fewer or no joins to arrive at basic information.

The pg_catalog contains raw pg maintenance tables in addition to views while the information_schema only contains read-only views against the core tables. So this means with sufficient super rights and a bit of thirst for adventure in your blood, you can really fuck up your database or make fast changes such as moving objects to different schemas, by directly updating these tables, that you can't normally do the supported way.

The other odd thing about the pg_catalog schema is that to reference objects in it, you do not have to schema qualify it as you would have to with the information_schema. For example you can say

```
SELECT * FROM pg_tables
```

instead of

```
SELECT * FROM pg_catalog.pg_tables
```

You will notice that also all the global functions are in there and do not need to be schema qualified. Interestingly enough pg_catalog appears nowhere in the search path, so it appears this is just hard-wired into the heart of PostgreSQL to be first in the search path.

To demonstrate - try creating a dummy table in the public schema with name **pg_tables**. Now if you do

```
SELECT * from pg_tables
```

- guess which table the results are for?

Casts, Operators, Types

Ability to define Casts, Operators and Types is a fairly unique feature of PostgreSQL that is rare to find in other databases. Postgres allows one to define automatic casting behavior and how explicit casts are performed. It also allows one to define how operations between different or same datatypes are performed. For creating new types, these features are extremely important since the database server would not have a clue how to treat these in common SQL use. For a great example of using these features, check out Andreas Scherbaum's - BOOLEAN datatype with PHP-compatible output

For each table that is created, an implicit type is created as well that mirrors the structure of the table.

Conversions

Conversions define how characters are converted from one encoding to another - say from ascii_to_utf8. There isn't much reason to touch these or add to them that we can think of. If one looks under pg_catalog - you will find a hundred someodd conversion objects.

Domains

Domains are sort of like types and are actually used like types. They are a convenient way of packaging common constraints into a data type. For example if you have an email address, a postal code, or a phone number or something of that sort that you require to be input in a certain way, a domain type would validate such a thing. So its like saying "I am a human, but I am a kid and need constraints placed on me to prevent me from choking on steak."

Example is provided below

```
CREATE DOMAIN us_fedid As varchar(11)
CHECK ( VALUE ~ E'^\\d{3}-\\d{2}-\\d{4}$' OR  VALUE ~ E'^\\d{2}-\\d{7}$');

CREATE TABLE us_members (
        member_id SERIAL NOT NULL PRIMARY KEY,
        federal_num us_fedid
);
```

Functions

This is the container for stored functions. As mentioned in prior articles, PostgreSQL does not have stored procedures, but its stored function capability is in general much more powerful than you will find in other database management systems (DBMS) so for all intents and purposes, stored functions fill the stored procedure role. What makes PostgreSQL stored function architecture admirable is that you have a choice of languages to define stored functions in. **SQL** and **PLPGSQL** are the languages pre-packaged with PostgreSQL. In addition to those you have PLPerl, PLPerlU, PLPython, PLRuby, PLTCL, PLSH (shell), PLR and Java. In terms of ease of setup across all OSes, we have found **PLR** to be most friendly of setups. **PLR** on top of that serves a special niche in terms of analysis and graphing

capability not found in the other languages. It opens up the whole R statistical platform to you. For those who have used SAS,S, and Matlab, R is of a similar nature so its a popular platform for scientists, engineers and GIS analysts.

Operator Classes, Operator Families

Operator Classes are used to define how indexes are used for operator operations. PostgreSQL has several index options to choose from with the most common being **btree** and **gist**. It is possible to define your own internal index structure. If you do such a thing, then you will need to define Operator Classes to go with these. Also if you are defining a new type with a specialty structure that uses a preferred type of index, you will want to create an Operator Class for this.

Sequences

Sequence objects are the equivalent of identity in Microsoft SQL Server and Auto Increment in MySQL, but they are much more powerful. What makes a sequence object more powerful than the former is that while they can be tied to a table and auto-incremented as each new record is added, they can also be incremented independent of a table. The same sequence object can also be used to increment multiple tables. It must be noted that Oracle also has sequence objects, but Oracle's sequence objects are much messier to use than PostgreSQL and Oracle doesn't have a slick concept of SERIAL datatype that makes common use of sequences easy to create and use.

Sequence objects are automatically created when you define a table field as type *serial*. They can also be created independently of a table by executing a DDL command of the form

```
CREATE SEQUENCE test_id_seq
  INCREMENT 1
  MINVALUE 1
  START 200;
```

If you wanted to manually increment a sequence - say in use in a manual insert statement where you need to know the id being assigned, you can do something of the following.

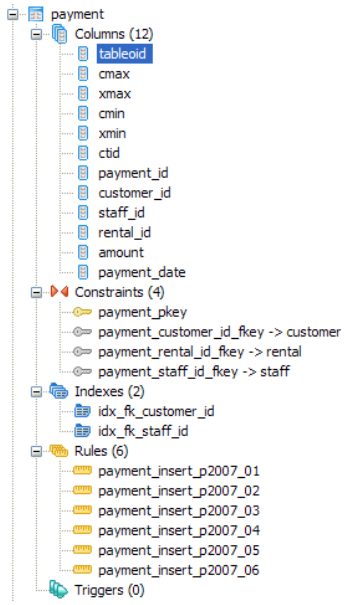
```
newid := nextval('test_id_seq');
INSERT INTO mytesttable(theid, thevalue)
        VALUES(newid, 'test me');
INSERT INTO mytest_children(parent_id, thevalue)
        VALUES(newid, 'stuff, more stuff');
```

Here are some sequence fast facts

- When you create a new table with a serial data type, and integer field is created, a sequence object is automatically created with the name *tablename_fieldname*_seq where tablename and fieldname are the tablename and fieldname of the table containing the serial field and the default value of the new field is set to the next value of the sequence object. The sequence is created in the same schema as the table.
- PostgreSQL makes no restriction on how many serial/sequence fields you can have in a table.
- Sequences can be incremented independently of a table.
- An auto-created sequence object (as a result of serial data type definition) is automatically dropped when the table is dropped (this is not the case for Pre-7.4 PostgreSQL, but is for PostgreSQL 8 and above).

Tables

We've already covered sequences which can exist independent or dependent of tables. We already know tables hold data. Now we shall look at the objects that hang off of a table. Below is a snapshot of the **payment** table in Pagila demo database



Columns - We all know what columns are. What is a little interesting about PostgreSQL - is that it has 6 system columns that every table has. These are *tableoid, cmax, xmax, cmin, xmin, ctid* and sometimes *oid if you CREATE TABLE WITH OIDS*. If you do a **SELECT *** on a table, you will never see these fields. You have to explicitly select them. The tableoid is the same for all records in a given table.

If you did a
```
SELECT COUNT(DISTINCT tableoid) FROM payment
```
in the *pagila* database, you will notice it returns **5**. How can that be when we said all records in a table have the same tableoid? This happens because the payment table is a parent to 5 tables and we don't even have any data in the payment table. So what the 5 is telling us here is that the **payment** table is comprised of data from 5 tables that inherit from it. When you do a select from a parent table, it in turn queries its children that are not **constraint excluded** by the query.

Rules - tables can have rules bound to them. In this case, the payment table has 6 rules bound to it, which redirect inserts to the child table containing the data that fits the date criterion. Using rules for table partitioning is a common use case in PostgreSQL. In other databases such as SQL Server Enterprise 2005 - this would be called **Functional Partitioning** and the equivalent to the PostgreSQL rules (in combination with contraints) would be equivalent to **Partitioning Functions**. Partitioning is only really useful for fairly large tables, otherwise the added overhead would probably not result in any speed gain and could actually reduce speed performance. PostgreSQL partitioning strategy is fairly simple and easy to understand when compared to some high-end commercial databases. In PostgreSQL 8.4 this strategy will probably become more sophisticated.

Triggers - PostgreSQL allows one to define Triggers on events BEFORE INSERT/UPDATE, AFTER INSERT/UPDATE and for EACH ROW or EACH STATEMENT. The minor restriction in PostgreSQL is that the trigger body can not be written directly in the trigger envelop. The trigger envelop must call a triggering function and the triggering function is a special kind of function that returns a **trigger**.

Indexes, Keys and Foreign Key Constraints - These objects are equivalent and behave the same as in other databases. PostgreSQL support referential integrity constraints and CASCADE UPDATE/ DELETE on these.

Views

Last but not least, our favorite - Views. Views are the best thing since sliced-bread. They are not tables but rather saved queries that are presented as tables (Virtual Tables). They allow you to do a couple of interesting things

- Abstract a complicated relational structure into a commonly used easy to digest flat-file view well suited for reporting.
- Just like stored functions/stored procs, one can use a view to limit user's ability to query certain columns and rows, but unlike cumbersome stored procedures/stored functions (that require you to pass in arguments in a certain order and unable to inspect the structure of the return value until its returned), these are presented as a harmless familiar looking table structure. For a more detailed description of the pros and cons of using views, stored procs, stored functions, triggers etc. check out our Choice Between Stored Procedures, Functions, Views, Triggers, Inline SQL

article. For a detailed example of setting up a view check out Database Abstraction with Updateable Views

- Here is an interesting example posed by Magnus Hagander Database or schema that demonstrates using View in combination with schemas to control input and visibility of rows.

Back to Table Of Contents

Trojan SQL Function Hack - A PL Lemma in Disguise *Advanced*

Have you ever noticed that in PostgreSQL you can put set returning functions in the SELECT part of an sql statement if the function is written in language SQL or C. Try the same trick for PL written functions such as plpgsql, plperl, plr etc, and you get a slap on the wrist of the form **ERROR: set-valued function called in context that cannot accept a set**. For Plpgsql and other PL languages you must put the set returning function in the FROM clause.

Below is a simple example:

```
--Build test data
CREATE TABLE test
(
  test_id serial NOT NULL,
  test_date date,
  CONSTRAINT pk_test PRIMARY KEY (test_id)
)
WITH (OIDS=FALSE);

INSERT INTO test(test_date)
        SELECT current_date + n
                FROM generate_series(1,1000) n;

--test function with sql
CREATE OR REPLACE FUNCTION fnsqltestprevn(id integer, lastn integer)
  RETURNS SETOF test AS
$$
        SELECT *
        FROM test
        WHERE test_id < $1 ORDER BY test_id
        LIMIT $2
$$
  LANGUAGE 'sql' VOLATILE;

--Test example 1 works fine
SELECT (fnsqltestprevn(6,5)).*;

--Test example 2 works fine
SELECT *
        FROM fnsqltestprevn(6,5);

--Same test function written as plpgsql
CREATE OR REPLACE FUNCTION fnplpgsqltestprevn(id integer, prevn integer)
  RETURNS SETOF test AS
$$
DECLARE
    rectest test;
BEGIN
        FOR rectest
                IN(SELECT *
                        FROM test
                        WHERE test_id < id
                        ORDER BY test_id LIMIT prevn)
                                                                LOOP
                RETURN NEXT rectest;
        END LOOP;
END;
$$
LANGUAGE 'plpgsql' VOLATILE;

--Test example 1 - gives error
-- ERROR:  set-valued function called in context that cannot accept a set
SELECT (fnplpgsqltestprevn(6,5)).*;

--Test example 2 works fine
SELECT *
        FROM fnplpgsqltestprevn(6,5);
```

So it appears that PostgreSQL is not quite as democratic as we would like.

```
--But what if we did this?
CREATE OR REPLACE FUNCTION fnsqltrojtestprevn(id integer, prevn integer)
        RETURNS SETOF test AS
$$
        SELECT * FROM fnplpgsqltestprevn($1, $2);
$$
LANGUAGE 'sql' VOLATILE;

--Test example 1 - works fine
SELECT (fnsqltrojtestprevn(6,5)).*;

--Test example 2 works fine
SELECT *
        FROM fnsqltrojtestprevn(6,5);
```

All interesting, but so what? you may ask. It is bad practice to put set returning functions in a SELECT clause. Such things are commonly mistakes and should be avoided.

Functional Row Expansion

It turns out that there are a whole class of problems in SQL where the simplest way to achieve the desired result is via a technique we shall call **Functional Row Expansion**. By that, we mean that for each record in a given set, we want to return another set of records that can not be expressed as a constant join expression. Basically the join expression is different for each record or the function we want to apply is too complicated to be expressed as a static join statement or join at all.

Taking the above example. Lets say for each record in test, you want to return the 4 records preceding including the current one. So basically you want to explode each row into 5 or fewer rows. Your general gut reaction would be do something as follows:

these give error: ERROR: function expression in FROM cannot refer to other relations of same query level

```
                SELECT test.test_id As ref_id, test.test_date as ref_date, targ.*
                        FROM test ,
                                (SELECT tinner.*
                                        FROM test as tinner
                                                WHERE tinner.test_id <= test.test_id
                                        ORDER BY tinner.test_id LIMIT 5) As targ;
```

```
                SELECT test.test_id As ref_id, test.test_date as ref_date, targ.*
                        FROM test,fnsqltrojtestprevn(test.test_id, 5) As targ;
```

--But this does what you want

```
SELECT test.test_id As ref_id, test.test_date as ref_date,
        (fnsqltrojtestprevn(test.test_id, 5)).*
  FROM test
```

Keep in mind what makes the above tricky is that you want to return at most 4 of the preceding plus current. If you want to return all the preceding plus current, then you can do a trivial self join as follows:

```
SELECT test.test_id As ref_id, test.test_date as ref_date, targ.*
  FROM test INNER JOIN
         test As targ ON targ.test_id <= test.test_id
  ORDER BY test.test_id, targ.test_id
```

So as you can see - its sometimes tricky to tell when you need to use this technique and when you don't.

For this trivial example, writing the function as an SQL only function works fine and is the best to use. SQL functions unfortunately lack the ability to define dynamic sql statements, among other deficiencies so resorting to using a pl language is often easier which means you lose this useful feature of sql functions. Stuffing a pl function in an SQL function just might do the trick. We haven't tried this on other pl languages except plpgsql, but we suspect it should work the same.

Back to Table Of Contents

CrossTab Queries in PostgreSQL using tablefunc contrib *Intermediate*

The generic way of doing cross tabs (sometimes called PIVOT queries) in an ANSI-SQL database such as PostgreSQL is to use CASE statements which we have documented in the article What is a crosstab query and how do you create one using a relational database?.

In this particular issue, we will introduce creating crosstab queries using PostgreSQL **tablefunc** contrib.

Installing Tablefunc

Tablefunc is a contrib that comes packaged with all PostgreSQL installations - we believe from versions 7.4.1 up (possibly earlier). We will be assuming the one that comes with 8.2 for this exercise. Note in prior versions, tablefunc was not documented in the standard postgresql docs, but the new 8.3 seems to have it documented at http://www.postgresql.org/docs/8.3/static/tablefunc.html.

Often when you create crosstab queries, you do it in conjunction with GROUP BY and so forth. While the astute reader may conclude this from the docs, none of the examples in the docs specifically demonstrate that and the more useful example of **crosstab(source_sql,category_sql)** is left till the end of the documentation.

To install tablefunc simply open up the **share\contrib\tablefunc.sql** in pgadmin and run the sql file. Keep in mind that the functions are installed by default in the *public* schema. If you want to install in a different schema - change the first line that reads

```
SET search_path = public;
```

Alternatively you can use **psql** to install tablefunc using something like the following command:

```
path\to\postgresql\bin\psql -h localhost -U someuser -d somedb -f "path\to\postgresql\share
\contrib\tablefunc.sql"
```

We will be covering the following functions

1. crosstab(source_sql, category_sql)
2. crosstab(source_sql)
3. Tricking crosstab to give you more than one row header column
4. Building your own custom crosstab function similar to the crosstab3, crosstab4 etc. examples
5. Adding a total column to crosstab query

There are a couple of key points to keep in mind which apply to both crosstab functions.

1. Source SQL must always return 3 columns, first being what to use for row header, second the bucket slot, and third is the value to put in the bucket.
2. crosstab except for the example crosstab3 ..crosstabN versions return unknown record types. This means that in order to use them in a FROM clause, you need to either alias them by specifying the result type or create a custom crosstab that outputs a known type as demonstrated by the crosstabN flavors. Otherwise you get the common *a column definition list is required for functions returning "record"* error.
3. A corrollary to the previous statement, it is best to cast those 3 columns to specific data types so you can be guaranteed the datatype that is returned so it doesn't fail your row type casting.
4. Each row should be unique for row header, bucket otherwise you get unpredictable results

Setting up our test data

For our test data, we will be using our familiar inventory, inventory flow example. Code to generate structure and test data is shown below.

```
CREATE TABLE inventory
(
  item_id serial NOT NULL,
  item_name varchar(100) NOT NULL,
  CONSTRAINT pk_inventory PRIMARY KEY (item_id),
  CONSTRAINT inventory_item_name_idx UNIQUE (item_name)
)
WITH (OIDS=FALSE);

CREATE TABLE inventory_flow
(
  inventory_flow_id serial NOT NULL,
  item_id integer NOT NULL,
  project varchar(100),
  num_used integer,
  num_ordered integer,
  action_date timestamp without time zone
        NOT NULL DEFAULT CURRENT_TIMESTAMP,
  CONSTRAINT pk_inventory_flow PRIMARY KEY (inventory_flow_id),
  CONSTRAINT fk_item_id FOREIGN KEY (item_id)
      REFERENCES inventory (item_id)
      ON UPDATE CASCADE ON DELETE RESTRICT
)
WITH (OIDS=FALSE);

CREATE INDEX inventory_flow_action_date_idx
  ON inventory_flow
  USING btree
  (action_date)
  WITH (FILLFACTOR=95);

INSERT INTO inventory(item_name) VALUES('CSCL (g)');
INSERT INTO inventory(item_name) VALUES('DNA Ligase (ul)');
INSERT INTO inventory(item_name) VALUES('Phenol (ul)');
INSERT INTO inventory(item_name) VALUES('Pippette Tip 10ul');


INSERT INTO inventory_flow(item_id, project, num_ordered, action_date)
        SELECT i.item_id, 'Initial Order', 10000, '2007-01-01'
                FROM inventory i;

--Similulate usage
INSERT INTO inventory_flow(item_id, project, num_used, action_date)
        SELECT i.item_id, 'MS', n*2,
                '2007-03-01'::timestamp + (n || ' day')::interval + ((n + 1) || ' hour')::interval
                FROM inventory As i CROSS JOIN generate_series(1, 250) As n
                WHERE mod(n + 42, i.item_id) = 0;

INSERT INTO inventory_flow(item_id, project, num_used, action_date)
        SELECT i.item_id, 'Alzheimer''s', n*1,
                '2007-02-26'::timestamp + (n || ' day')::interval + ((n + 1) || ' hour')::interval
                FROM inventory as i CROSS JOIN generate_series(50, 100) As n
                WHERE mod(n + 50, i.item_id) = 0;

INSERT INTO inventory_flow(item_id, project, num_used, action_date)
        SELECT i.item_id, 'Mad Cow', n*i.item_id,
                '2007-02-26'::timestamp + (n || ' day')::interval + ((n + 1) || ' hour')::interval
                FROM inventory as i CROSS JOIN generate_series(50, 200) As n
                WHERE mod(n + 7, i.item_id) = 0 AND i.item_name IN('Pippette Tip 10ul', 'CSCL
(g)');

vacuum analyze;
```

Using crosstab(source_sql, category_sql)

For this example we want to show the monthly usage of each inventory item for the year 2007 regardless of project. The crosstab we wish to achieve would have columns as follows: **item_name, jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec**

```
--Standard group by aggregate query before we pivot to cross tab
--This we use for our source sql
 SELECT i.item_name::text As row_name, to_char(if.action_date, 'mon')::text As bucket,
                SUM(if.num_used)::integer As bucketvalue
        FROM inventory As i INNER JOIN inventory_flow As if
                ON i.item_id = if.item_id
        WHERE (if.num_used <> 0 AND if.num_used IS NOT NULL)
          AND action_date BETWEEN date '2007-01-01' and date '2007-12-31 23:59'
        GROUP BY i.item_name, to_char(if.action_date, 'mon'), date_part('month', if.action_date)
        ORDER BY i.item_name, date_part('month', if.action_date);

--Helper query to generate lowercase month names - this we will use for our category sql
SELECT to_char(date '2007-01-01' + (n || ' month')::interval, 'mon') As short_mname
                FROM generate_series(0,11) n;
```

--Resulting crosstab query --**Note: For this we don't need the order by month since the order of the columns is determined by the category_sql row order**

```
SELECT mthreport.*
        FROM
        crosstab('SELECT i.item_name::text As row_name, to_char(if.action_date, ''mon'')::text As
bucket,
                SUM(if.num_used)::integer As bucketvalue
        FROM inventory As i INNER JOIN inventory_flow As if
                ON i.item_id = if.item_id
          AND action_date BETWEEN date ''2007-01-01'' and date ''2007-12-31 23:59''
        GROUP BY i.item_name, to_char(if.action_date, ''mon''), date_part(''month'', if.
action_date)
        ORDER BY i.item_name',
        'SELECT to_char(date ''2007-01-01'' + (n || '' month'')::interval, ''mon'') As
short_mname
                FROM generate_series(0,11) n')
                As mthreport(item_name text, jan integer, feb integer, mar integer,
                        apr integer, may integer, jun integer, jul integer,
                        aug integer, sep integer, oct integer, nov integer,
                        dec integer)
```

The output of the above crosstab looks as follows:

item_name text	jan integer	feb integer	mar integer	apr integer	may integer	jun integer	jul integer	aug integer	sep integer	oct integer	nov integer	dec integer
CSCL (g)			870	3884	9000	9792	11691	14625	15068	13230	7290	
DNA Ligase (ul)			420	1650	3360	3364	3960	4860	5348	6600	3888	
Phenol (ul)			270	1096	2245	2361	2376	3210	3810	4410	2430	
Pippette Tip 10ul			196	1550	3800	4560	6432	6888	6440	3520	1952	

Using crosstab(source_sql)

crosstab(source_sql) is much trickier to understand and use than the crosstab(source_sql, category_sql) variant, but in certain situations and certain cases is faster and just as effective. The reason why is that crosstab(source_sql) is not guaranteed to put same named buckets in the same columns especially for sparsely populated data. For example - lets say you have data for CSCL for Jan Mar Apr and data for Phenol for Apr. Then Phenols Apr bucket will be in the same column as CSCL Jan's bucket. This in most cases is not terribly useful and is confusing.

To skirt around this inconvenience one can write an SQL statement that guarantees you have a row for each permutation of Item, Month by doing a cross join. Below is the above written so item month usage fall in the appropriate buckets.

```
        --Code to generate the row tally - before crosstab
        SELECT i.item_name::text As row_name, i.start_date::date As bucket,
                        SUM(if.num_used)::integer As bucketvalue
                FROM (SELECT inventory.*,
                          date '2007-01-01' + (n || ' month')::interval As start_date,
                          date '2007-01-01' + ((n + 1) || ' month')::interval +  - '1 minute'::
interval As end_date
                        FROM inventory CROSS JOIN generate_series(0,11) n) As i
                                LEFT JOIN inventory_flow As if
                ON (i.item_id = if.item_id AND if.action_date BETWEEN i.start_date AND i.end_date)
        GROUP BY i.item_name, i.start_date
        ORDER BY i.item_name, i.start_date;


        --Now we feed the above into our crosstab query to achieve the same result as
        --our crosstab(source, category) example
        SELECT mthreport.*
        FROM crosstab('SELECT i.item_name::text As row_name, i.start_date::date As bucket,
                        SUM(if.num_used)::integer As bucketvalue
                FROM (SELECT inventory.*,
                          date ''2007-01-01'' + (n || '' month'')::interval As start_date,
                          date ''2007-01-01'' + ((n + 1) || '' month'')::interval +  - ''1
minute''::interval As end_date
                        FROM inventory CROSS JOIN generate_series(0,11) n) As i
                                LEFT JOIN inventory_flow As if
                ON (i.item_id = if.item_id AND if.action_date BETWEEN i.start_date AND i.end_date)
        GROUP BY i.item_name, i.start_date
        ORDER BY i.item_name, i.start_date;')
                As mthreport(item_name text, jan integer, feb integer,
                        mar integer, apr integer,
                        may integer, jun integer, jul integer, aug integer,
                        sep integer, oct integer, nov integer, dec integer)
```

In actuality the above query if you have an index on action_date is probably more efficient for larger datasets than the crosstab(source, category) example since it utilizes a date range condition for each month match.

There are a couple of situations that come to mind where the standard behavior of crosstab of not putting like items in same column is useful. One example is when its not necessary to distiguish bucket names, but order of cell buckets is important such as when doing column rank reports. For example if you wanted to know for each item, which projects has it been used most in and you want the column order of projects to be based on highest usage. You would have simple labels like **item_name, project_rank_1, project_rank_2, project_rank_3** and the actual project names would be displayed in project_rank_1, project_rank_2, project_rank_3 columns.

```
SELECT projreport.*
        FROM crosstab('SELECT i.item_name::text As row_name,
                if.project::text As bucket,
                if.project::text As bucketvalue
        FROM inventory  i
                        LEFT JOIN inventory_flow As if
        ON (i.item_id = if.item_id)
        WHERE if.num_used > 0
GROUP BY i.item_name, if.project
ORDER BY i.item_name, SUM(if.num_used) DESC, if.project')
```

```
        As projreport(item_name text, project_rank_1 text, project_rank_2 text,
                        project_rank_3 text)
```

Output of the above looks like:

item_name text	project_rank_1 text	project_rank_2 text	project_rank_3 text
CSCL (g)	MS	Mad Cow	Alzheimer's
DNA Ligase (ul)	MS	Alzheimer's	
Phenol (ul)	MS	Alzheimer's	
Pippette Tip 10ul	Mad Cow	MS	Alzheimer's

Tricking crosstab to give you more than one row header column

Recall we said that crosstab requires exactly 3 columns output in the sql source statement. No more and No less. So what do you do when you want your month crosstab by Item, Project, and months columns. One approach is to stuff more than one Item in the item slot by either using a delimeter or using an Array. We shall show the array approach below.

```
SELECT mthreport.row_name[1] As project, mthreport.row_name[2] As item_name,
        jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
        FROM
        crosstab('SELECT ARRAY[if.project::text, i.item_name::text] As row_name,
                to_char(if.action_date, ''mon'')::text As bucket, SUM(if.num_used)::integer As
bucketvalue
        FROM inventory As i INNER JOIN inventory_flow As if
                ON i.item_id = if.item_id
          AND action_date BETWEEN date ''2007-01-01'' and date ''2007-12-31 23:59''
          WHERE if.num_used <> 0
        GROUP BY if.project, i.item_name, to_char(if.action_date, ''mon''),
                date_part(''month'', if.action_date)
        ORDER BY if.project, i.item_name',
        'SELECT to_char(date ''2007-01-01'' + (n || '' month'')::interval, ''mon'') As
short_mname
                FROM generate_series(0,11) n')
                As mthreport(row_name text[], jan integer, feb integer, mar integer,
                        apr integer, may integer, jun integer, jul integer,
                        aug integer, sep integer, oct integer, nov integer,
                        dec integer)
```

Result of the above looks as follows:

project text	item_name text	jan integer	feb integer	mar integer	apr integer	may integer	jun integer	jul integer	aug integer	sep integer	oct integer	nov integer	dec integer
Alzheimer's	CSCL (g)				666	2295	864						
Alzheimer's	DNA Ligase (ul)				330	1140	480						
Alzheimer's	Phenol (ul)				226	775	291						
Alzheimer's	Pippette Tip 10ul				162	608	192						
Mad Cow	CSCL (g)				666	2295	2954	4035	4935	3990			
Mad Cow	Pippette Tip 10ul				684	2156	2940	4320	4620	3780			
MS	CSCL (g)			870	2552	4410	5974	7656	9690	11078	13230	7290	
MS	DNA Ligase (ul)			420	1320	2220	2884	3960	4860	5348	6600	3888	
MS	Phenol (ul)			270	870	1470	2070	2376	3210	3810	4410	2430	
MS	Pippette Tip 10ul			196	704	1036	1428	2112	2268	2660	3520	1952	

Building your own custom crosstab function

If month tabulations are something you do often, you will quickly become tired of writing out all the months. One way to get around this inconvenience - is to define a type and crosstab alias that returns the well-defined type something like below:

```
CREATE TYPE tablefunc_crosstab_monthint AS
   (row_name text[],jan integer, feb integer, mar integer,
        apr integer, may integer, jun integer, jul integer,
        aug integer, sep integer, oct integer, nov integer,
        dec integer);

CREATE OR REPLACE FUNCTION crosstabmonthint(text, text)
  RETURNS SETOF tablefunc_crosstab_monthint AS
'$libdir/tablefunc', 'crosstab_hash'
  LANGUAGE 'c' STABLE STRICT;
```

Then you can write the above query as

```
SELECT mthreport.row_name[1] As project, mthreport.row_name[2] As item_name,
        jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
        FROM
        crosstabmonthint('SELECT ARRAY[if.project::text, i.item_name::text] As row_name, to_char
(if.action_date, ''mon'')::text As bucket,
SUM(if.num_used)::integer As bucketvalue
        FROM inventory As i INNER JOIN inventory_flow As if
                ON i.item_id = if.item_id
          AND action_date BETWEEN date ''2007-01-01'' and date ''2007-12-31 23:59''
          WHERE if.num_used <> 0
        GROUP BY if.project, i.item_name, to_char(if.action_date, ''mon''), date_part(''month'',
if.action_date)
        ORDER BY if.project, i.item_name',
        'SELECT to_char(date ''2007-01-01'' + (n || '' month'')::interval, ''mon'') As
short_mname
                FROM generate_series(0,11) n')
                As mthreport;
```

Adding a Total column to the crosstab query

Adding a total column to a crosstab query using crosstab function is a bit tricky. Recall we said the source sql should have exactly 3 columns (row header, bucket, bucketvalue). Well that wasn't entirely accurate. The **crosstab(source_sql, category_sql)** variant of the function allows for a source that has columns **row_header, extraneous columns, bucket, bucketvalue**. Don't get extraneous columns confused with row headers. They are not the same and if you try to use it as we did for creating multi row columns, you will be leaving out data. For simplicity here is a fast rule to remember. **Extraneous column values must be exactly the same for source rows that have the same row header and they get inserted right before the bucket columns.**

We shall use this fact to produce a total column.

```
--This we use for our source sql
 SELECT i.item_name::text As row_name,
        (SELECT SUM(sif.num_used)
                FROM inventory_flow sif
                        WHERE action_date BETWEEN date '2007-01-01' and date '2007-12-31 23:59'
                                AND sif.item_id = i.item_id)::integer As total,
                                to_char(if.action_date, 'mon')::text As bucket,
```

```
                SUM(if.num_used)::integer As bucketvalue
        FROM inventory As i INNER JOIN inventory_flow As if
                ON i.item_id = if.item_id
        WHERE (if.num_used <> 0 AND if.num_used IS NOT NULL)
          AND action_date BETWEEN date '2007-01-01' and date '2007-12-31 23:59'
        GROUP BY i.item_name, total, to_char(if.action_date, 'mon'), date_part('month', if.
action_date)
        ORDER BY i.item_name, date_part('month', if.action_date);

--This we use for our category sql
SELECT to_char(date '2007-01-01' + (n || ' month')::interval, 'mon') As short_mname
                FROM generate_series(0,11) n;

--Now our cross tabulation query
SELECT mthreport.*
        FROM crosstab('SELECT i.item_name::text As row_name,
        (SELECT SUM(sif.num_used)
                FROM inventory_flow sif
                        WHERE action_date BETWEEN date ''2007-01-01'' and date ''2007-12-31 23:59''
                                AND sif.item_id = i.item_id)::integer As total,
                to_char(if.action_date, ''mon'')::text As bucket,
                SUM(if.num_used)::integer As bucketvalue
        FROM inventory As i INNER JOIN inventory_flow As if
                ON i.item_id = if.item_id
        WHERE (if.num_used <> 0 AND if.num_used IS NOT NULL)
          AND action_date BETWEEN date ''2007-01-01'' and date ''2007-12-31 23:59''
        GROUP BY i.item_name, total, to_char(if.action_date, ''mon''), date_part(''month'', if.
action_date)
        ORDER BY i.item_name, date_part(''month'', if.action_date)',
        'SELECT to_char(date ''2007-01-01'' + (n || '' month'')::interval, ''mon'') As short_mname
                FROM generate_series(0,11) n'
        )
                As mthreport(item_name text, total integer, jan integer, feb integer,
                        mar integer, apr integer,
                        may integer, jun integer, jul integer, aug integer,
                        sep integer, oct integer, nov integer, dec integer)
```

Resulting output of our cross tabulation with total column looks like this:

item_name text	total integ	jan integ	feb integ	mar integ	apr integ	may integ	jun integ	jul integ	aug integ	sep integ	oct integ	nov integ	dec integ
CSCL (q)	85450			870	3884	9000	9792	11691	14625	15068	13230	7290	
DNA Ligase (ul)	33450			420	1650	3360	3364	3960	4860	5348	6600	3888	
Phenol (ul)	22208			270	1096	2245	2361	2376	3210	3810	4410	2430	
Pippette Tip 10ul	35338			196	1550	3800	4560	6432	6888	6440	3520	1952	

If per chance you wanted to have a total row as well you could do it with a union query in your source sql. Unfotunately PostgreSQL does not support windowing functions that would make the row total not require a union. We'll leave that one as an exercise to figure out.

Another not so obvious observation. You can define a type that say returns 20 bucket columns, but your actual crosstab need not return up to 20 buckets. It can return less and whatever buckets that are not specified will be left blank. With that in mind, you can create a generic type that returns generic names and then in your application code - set the heading based on the category source. Also if you have fewer buckets in your type definition than what is returned, the right most buckets are just left off. This allows you to do things like list the top 5 colors of a garment etc.

Back to Table Of Contents

Using MS Access with PostgreSQL *Intermediate*

Many in the PostgreSQL community use Microsoft Access as a front-end to their PostgreSQL databases. Although MS Access is strictly a windows application and PostgreSQL has its roots in Unix, the two go well together. A large part of that reason is because the PostgreSQL ODBC driver is well maintained and has frequent updates. You can expect one new ODBC driver release every 4-6 months. There exist only 32-bit production quality drivers. The 64-bit driver is of alpha quality. In addition to other front-ends to PostgreSQL that utilize the ODBC driver used by Windows developers, there is VB 6 (VB. NET/C# use the ADO.NET driver also very well maintained), Visual FoxPro, Delphi, to name a few).

People who have never used Microsoft Access or anything like it and consider themselves hard-core programmers or database purists, dismiss Microsoft Access as a dangerous child's toy, causing nothing but grief when real programmers and database administrators have to debug the disorganized mess of amateurs. They dream of the day when this nuisance is rid of and their company can be finally under the strict bureaucratic control of well-designed apps that no one cares to use.

Beneath the croft of this dinkiness/dangerous toy is a RAD and Reporting tool that can connect to any database with an ODBC or ADO driver. It serves the unique niche of

1. Empowering a knowledge worker/beginner programmer/DB user who is slowly discovering the wonders of relational databases and what time savings such a tool can provide.
2. On the other side - it is inviting to the pragmatic (lazy) database programmer who has spent precious time to investigate its gems. The pragmatist sees it as a tool which provides a speedy development environment and intuitive reporting environment. It allows one to give more freedom to less experienced users, thus relieving one of tedious requests for information. By using it as a front-end to a strong server-side database such as PostgreSQL, it allows one to enforce a sufficient level of data integrity and control. The pragmatist realizes that often the best way to maintain order is to not fight disorder because the more you try to restrict people's freedoms, the craftier they get in devising ways of circumventing your traps. The pragmatic programmer also takes the view of *Give a man a fish and he will pester you for more fish. Teach a man to fish and he will help you catch bigger fish.*

In this article - we'll walk thru:

1. How to install the PostgreSQL ODBC driver and gotchas to watch out for
2. How to link to PostgreSQL tables and views via Linked tables
3. Pass-thru queries - what they are and how to create them?
4. How to export access tables and even other linked datasources to PostgreSQL - e.g. using MS Access as a simple exporting/importing tool
5. Quick setup of a form that uses the new TSearch functionality in PostgreSQL 8.3

For this example we will be using Microsoft Access 2003, PostgreSQL 8.3 RC2. For the database, we will be using the pagila 0.10 database (8.3 version).

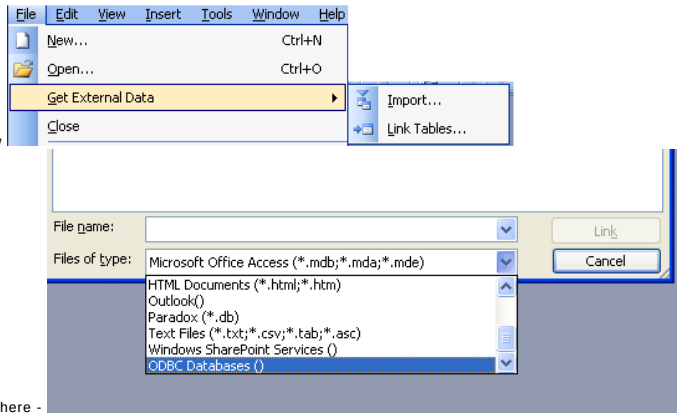
Installing PostgreSQL ODBC Driver

The latest PostgreSQL 32-bit ODBC Driver can be download by choosing a mirror from http://wwwmaster.postgresql.org/download/mirrors-ftp and then navigating to the pub/postgresql/odbc/versions/msi/ folder. The current version is psqlodbc_08_03_0100.zip which was released Jan-22-2008. For those who desperately need 64-bit ODBC, you can compile your own or try the AMD 64-bit test version.

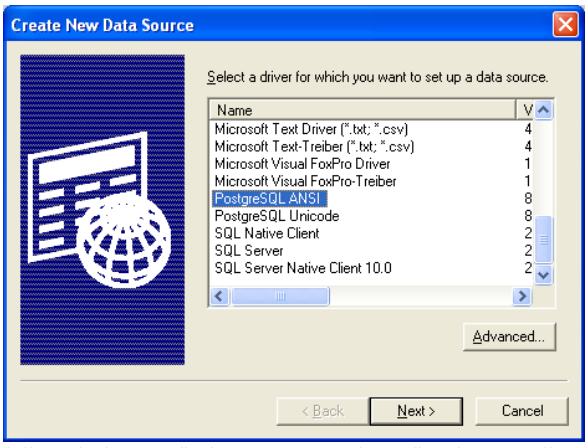
1. Unzip psqlodbc_08_03_0100.zip
2. Run the psqlodbc.msi file (If you have an older version of the PostgreSQL driver, uninstall it first before installing the new one)

How to link to PostgreSQL tables and views via Linked tables

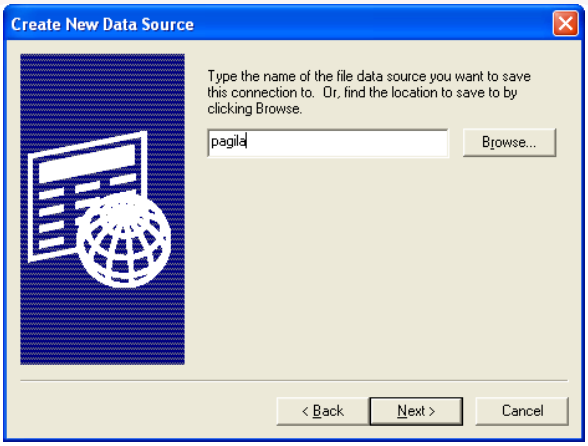
1. Create a blank Access Database
2. Go to Files->Get External Data->Linked Tables As shown below



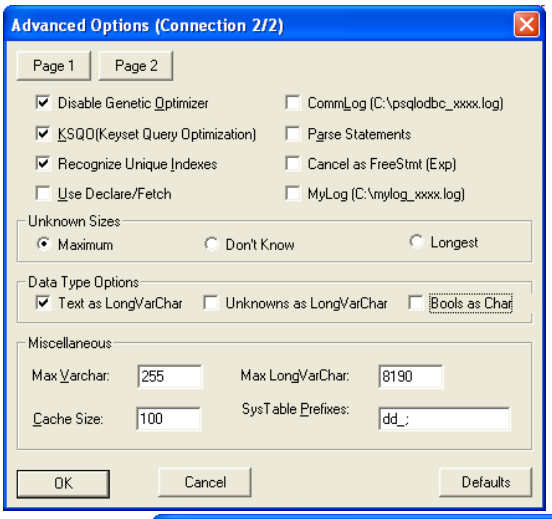
3. Click on drop down and switch to ODBC Datasources as shown here -
4. Switch to File Datasource. Note we are using File DSN instead of Machine Datasource because File DSN string gets embedded in the MS Access Database, therefore you do not have to setup the DSN on each computer that will use the MS Access Database. Machine DSNs have to be setup on each individual pc. File DSNS are also normally kept in files that sit in C:\Program Files\Common Files\ODBC\Data Sources and this default path can be changed from ODBC manager to a Network location if you want users to be able to share File DSNs.
5. Click New Button
6. Select driver as shown here.



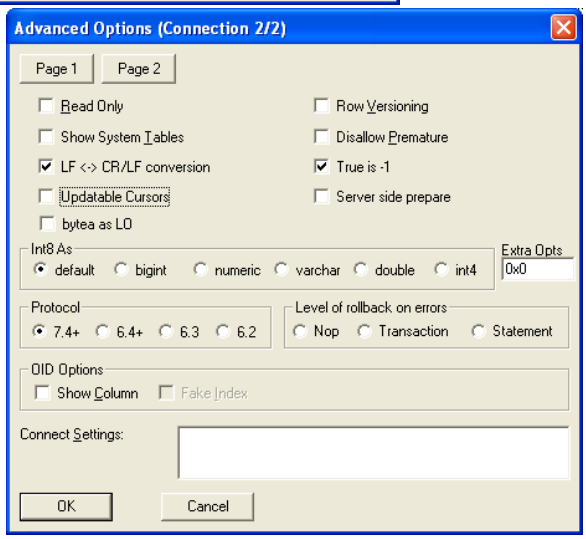
Note: in this picture we have selected the ANSI driver since our database is WIN-1252 encoded. Choose Unicode if your database encoding is UTF-8 or a non Latin Encoding, choose ANSI if your database encoding is SQL_ASCII, EUC_JP, BIG5,Shift-JIS, or a LATIN character set



7. Click Next and type in a name
8. Click Next and fill in relevant server, db.



9. Click the **Connection** button and uncheck **Bools as char** as shown



10. Click the **Page 2** button and check True is -1, and uncheck updateable cursors as shown and then click OK
11. Now select the tables you want and click **Save Password**.

    If you are missing primary keys on tables, Access will prompt you for what fields or set of fields you would like to use as the primary key. This doesn't make any structural changes to the actual table, but in the linked structure, Access will pretend this is the primary key and use that accordingly for table updates and such. This is particularly useful for views where the concept of primary keys does not exist and you want your updateable views to be updateable from Access. If you click OK or Cancel to the question without picking a set of fields, that table will be marked as readonly, which is the desired behavior for a lot of reporting views.

Pass-thru queries - what they are and how to create them

Access has a query feature called Pass-thru Queries available in the Query Designer. What this lets you do is pass a native PostgreSQL query directly to PostgreSQL so that it is not translated by the JET driver. Note pass-thru queries have visibility into the PostgreSQL db, and not your access database so don't expect to be using Access tables in them.

Pros

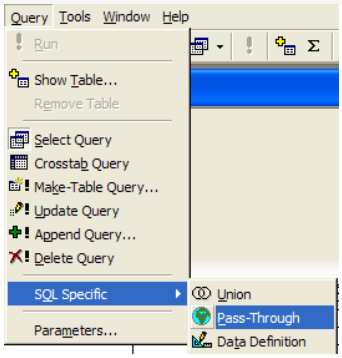
- You can use native PostgreSQL functions and every other sweet function in PostgreSQL that Access has no clue what to do with such as full text search queries and Postgis spatial queries.
- Skips the JET translation layer so is faster especially if you are joining with other tables in PostgreSQL
- You can reference PostgreSQL tables and views you don't have linked in.

Cons

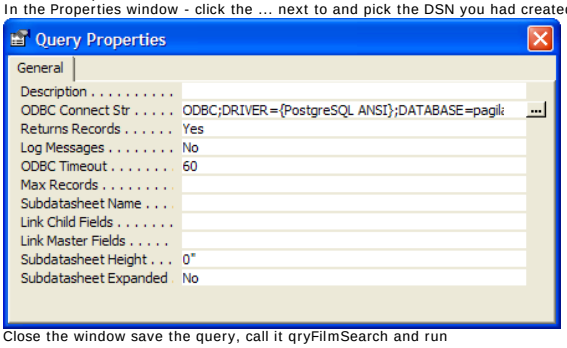
- Unlike using linked tables in queries, you can't access any tables, jet functions, or custom access functions you have in your access database
- Pass-thru queries are never updateable.

One example use is to for example use the sophisticated full text functionality in of PostgreSQL directly in MS Access. Below is an example using the Pagila database.

- Create a new MS Access Query and select **Design View** and don't bother picking any tables



- Under the Query menu choose -> SQL Specific -> Pass Through as shown
- Type
  ```
  SELECT * FROM film WHERE fulltext @@ to_tsquery('fate&india');
  ```
  in the query window
- Click the Properties icon 
- In the Properties window - click the ... next to and pick the DSN you had created earlier and choose to save password as show



- Close the window save the query, call it qryFilmSearch and run

Using Microsoft Access as an Exporting/Importing tool

In addition to linking tables, Microsoft Access can be used as a simple conduit for importing and exporting data in and out of PostgreSQL.

To export data to PostgreSQL from any linked table or physical table in Microsoft Access - do the following:

- Rename the table to the name you would like it named to.
- Make sure the default schema of the user you are using in Postgres, is the schema you want to export the data to.
- Go to File->Export-> Select ODBC Datasources which is way at the bottom and select the DSN you had created. One gotcha here is that PostgreSQL will maintain the casing of the fields in the table and the table name, so its best to rename all your fields to lowercase first so you don't have to be quoting them everytime you use them.

To import data from PostgreSQL into a Microsoft Access database for distribution etc. Do the following

- Choose File->Get External Data->Import
- Again select ODBC Datasource and use the DSN we created
- Select the set of views, tables etc you want to import and then click OK.

Building a form with a Pass-thru Query that uses TSearch

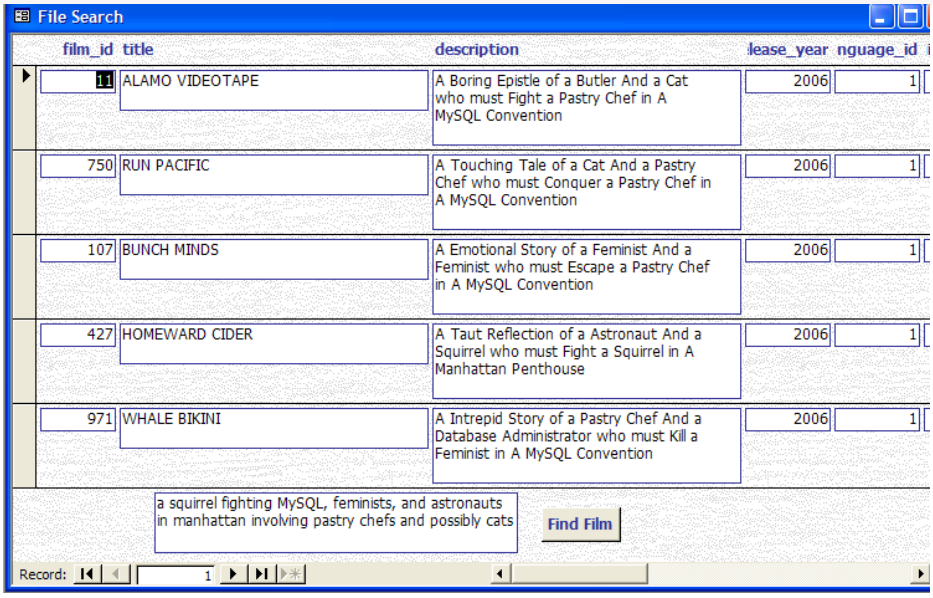
In this little example, we'll demonstrate how to create simple form bound to a pass-thru query and programmatically change the pass-thru query via user input.

- First bind the form to the passthru query you created above and just build the form with a wizard
- Next place a text box on form and name it **txtSearch**
- Next add a button on the form and name it **cmdFindFilm** and label it **Find Film**
- Now put in code for the onclick event of the button that looks like this

To programmatically change a pass thru query in response to a user's input so you can use it as a record source of a form, you can write something like this:

```
Private Sub cmdFindFilm_Click()
    Dim qdf As Object
    Dim tSearchText As String
    If Me.txtSearch.Value > "" Then
        tSearchText = Replace(Replace(Me.txtSearch.Value, " ", "|"), "'", "''")
        Set qdf = CurrentDb.QueryDefs("qryFilmSearch")
        qdf.SQL = "SELECT * FROM film WHERE fulltext @@ to_tsquery('" & tSearchText & "') ORDER BY ts_rank
(fulltext, to_tsquery('" & tSearchText & "')) DESC, title"
        qdf.Close
        Me.Requery
    Else
        MsgBox "Please type in a search criteria"
    End If
End Sub
```

Below is a snapshot of our finished form with a sample query we ran. Aint it cute.



Gotchas

File DSN does not let you change the port number

I suspect this is a bug. When setting up file dsns via ODBC manager, for some reason the port is greyed out so if you are not running on the standard 5432 port, you have to edit the generated .dsn file manually. On top of that the file doesn't get generated with all the necessary info if a successful connection is not made. To get around this annoyance, you can go into .dsn file (in this case C:\Program Files\Common Files\ODBC\Data Sources\pagila.dsn) and change the port number before linking. **Remember, once a table is linked with a file DSN, the actual DSN config gets encoded directly in the linked table meta data so you do not need to make the File DSN file accessible to users who use the access database. This is not true for Machine DSNs, only File DSNs.**

Below is something like what the DSN file should look like.

```
[ODBC]
DRIVER=PostgreSQL ANSI
UID=pagila
XaOpt=1
LowerCaseIdentifier=0
UseServerSidePrepare=0
ByteaAsLongVarBinary=0
BI=0
TrueIsMinus1=1
DisallowPremature=0
UpdatableCursors=0
LFConversion=1
ExtraSysTablePrefixes=dd_
CancelAsFreeStmt=0
Parse=0
BoolsAsChar=0
UnknownsAsLongVarchar=0
TextAsLongVarchar=1
UseDeclareFetch=0
Ksqo=1
Optimizer=1
CommLog=0
Debug=0
MaxLongVarcharSize=8190
MaxVarcharSize=255
UnknownSizes=0
Socket=4096
Fetch=100
ConnSettings=
ShowSystemTables=0
RowVersioning=0
ShowOidColumn=0
FakeOidIndex=0
Protocol=7.4-1
ReadOnly=0
SSLmode=disable
PORT=5432
SERVER=localhost
DATABASE=pagila
```

Tables Pre-fixed with schemas

One of our pet peeves is that when you link all the tables you want it prefixes the tables with the schema and its not schema.tablename its schema_tablename e.g. public_actors.

This is especially annoying if you use MS Access as a quick sql generator that you then use to paste back into your postgresql database as a view. This is an issue when you try to link any schema supporting database in MS Access. E.g. public_actors just is no good. Just actors works fine if you have default schemas in place or do not have a schema segmented database (e.g. everything is in public). Below is a VB subroutine we use to strip off the schema prefix.

```
Sub StripSchemaName(schemaname As String)
   'schemaname that prefixes the table e.g. public
   '--EXAMPLE use from immediate window -
   '--  StripSchemaName "public"
   Dim tdf As Object
   Dim i As Integer
   For Each tdf In CurrentDb.TableDefs
      If Left(tdf.Name, Len(schemaname)) = schemaname Then
         'plus 2 to strip the _ as well
         tdf.Name = Mid(tdf.Name, Len(schemaname) + 2)
      End If
   Next
   MsgBox "Done"
End Sub
```

Dealing with Booleans

One of the problems with using PostgreSQL as a back-end to MS Access is that Postgres has a true boolean data type where as MS Access has a Yes/No field which internally maps to -1 and 0. In

earlier versions of PostgreSQL, there was an auto-cast in place to cast boolean to integer and vice-versa. This was later taken out. So now you get errors like **operator does not exist boolean = integer** when trying to do queries against these fields.

Note the below example is useful for transparently casting Access's (True/False (-1/0) to PostgreSQL True/False)

The below was adapted from Bahut ODBC PostgreSQL boolean mess. In Bahut's rendition he uses plpgsql functions. We revised to just use plain sql functions. The reason being is that in general when a function can be written in SQL, it performs much better than a plpgsql or other PL language written function, because the sql functions are more transparent to the Postgres query planner for appying indexes and so forth. In this case, the SQL variants are more succinct as well.

```
CREATE OR REPLACE FUNCTION inttobool(integer, boolean) RETURNS boolean
AS $$
   SELECT CASE WHEN $1=0 and NOT $2 OR ($1<>0 and $2) THEN true ELSE false END
$$
LANGUAGE sql;

CREATE OR REPLACE FUNCTION inttobool(boolean, integer) RETURNS boolean
AS $$
   SELECT inttobool($2, $1);
$$
LANGUAGE sql;

CREATE OR REPLACE FUNCTION notinttobool(boolean, integer) RETURNS boolean
AS
$$
  SELECT NOT inttobool($2,$1);
$$
LANGUAGE sql;

CREATE OR REPLACE FUNCTION notinttobool(integer, boolean) RETURNS boolean
AS $$
       SELECT NOT inttobool($1,$2);
$$
LANGUAGE sql;

CREATE OPERATOR = (
PROCEDURE = inttobool,
LEFTARG = boolean,
RIGHTARG = integer,
COMMUTATOR = =,
NEGATOR = <>
);

CREATE OPERATOR <> (
PROCEDURE = notinttobool,
LEFTARG = integer,
RIGHTARG = boolean,
COMMUTATOR = <>,
NEGATOR = =
);

CREATE OPERATOR = (
PROCEDURE = inttobool,
LEFTARG = integer,
RIGHTARG = boolean,
COMMUTATOR = =,
NEGATOR = <>
);

CREATE OPERATOR <> (
PROCEDURE = notinttobool,
LEFTARG = boolean,
RIGHTARG = integer,
COMMUTATOR = <>,
NEGATOR = =
);
```

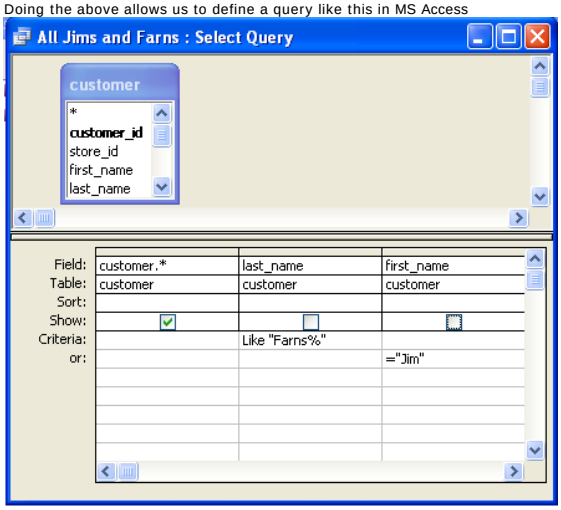
PostgreSQL is case-sensitive

One of the most annoying things for people coming from a Windows environment is that PostgreSQL is case-sensitive whereas MS Access in-general is not (except when querying case sensitive databases). Explaining this to users and training them on case sensitivity is just a lot of hassle, not to mention the time-loss of having to upper case things. Hopefully this will change in the future so that PostgreSQL supports different collation depending field by field similar to the way SQL Server 2005 does. Needless to say, when running a query in MS Access, one has three options:

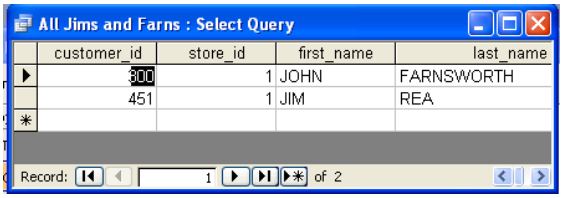
1. Write your query along the lines of
   `upper(somefield) LIKE UCase('abc%')`
   and make sure you have a functional index on upper(somefield)
2. Use the custom data type such as **citext** which you need to compile yourself.
3. or Put functional upper(somefield) indexes on your common fields and use the freedom that PostgreSQL gives you to redefine varchar operators in your database by doing the below. NOTE that this gives you the benefit of not having to redefine varchar fields as citext or anything like that thus making it more portable to transfer back and forth between non-case sensitive dbs or use the same schema as non-case sensitive dbs. Note we couldn't do the below with **text** because that is defined high up and can not be overwritten. We can overwrite the behavior of varchars however because varchars get implicitly cast to text and use the text operators. By using PostgreSQL's operator overload feature, we can define special behavior for varchar when used in comparators. When Postgres sees there is such an operator, it will use that instead of cast varchar to text and using the default text operators. The downside is that this will not work with PostgreSQL text (NOTE: varchar in PostgreSQL/ANSI SQL maps to text in MS Access and text in PostgreSQL/ANSI maps to memo in MS Access - all very confusing) . In most cases this is a non-issue since most searches are done on short Access text fields rather than memo fields. **NOTE: Use with caution. We haven't thoroughly tested this technique to catch all the possible situations where it can go wrong. It seems to behave correctly from our naive tests.**

```
CREATE OR REPLACE FUNCTION ci_caseinsmatch(varchar, varchar) RETURNS boolean
AS $$
   SELECT UPPER($1)::text = UPPER($2)::text;
$$
LANGUAGE sql
IMMUTABLE STRICT;

CREATE OPERATOR = (
PROCEDURE = ci_caseinsmatch,
LEFTARG = varchar,
RIGHTARG = varchar,
COMMUTATOR = =,
NEGATOR = <>
);


CREATE FUNCTION ci_like(varchar, varchar) RETURNS boolean
AS $$
   SELECT UPPER($1)::text LIKE UPPER($2)::text;
$$
LANGUAGE sql;

CREATE OPERATOR ~~(
  PROCEDURE = ci_like,
  LEFTARG = varchar,
  RIGHTARG = varchar,
  RESTRICT = likesel,
  JOIN = likejoinsel);
```

Doing the above allows us to define a query like this in MS Access



Which yields:



And can now be written in SQL even in PgAdmin without all that messy upper lower stuff and still uses indexes if you have them defined on say upper(first_name) or doing range case-insensitive searches e.g. (customer.last_name between 'f' and 'h') :

```
SELECT customer.*
FROM customer
WHERE customer.last_name
Like 'Farns%' OR customer.first_name = 'Jim';
```

Which will give you all customers with first name Jim or last name like Farns. Best of all, if you put in a functional index on last name and first name like below, it will use those indexes when doing equality or between ranges etc..

Back to Table Of Contents

Using OpenOffice Base 2.3.1 with PostgreSQL *Beginner*

For those who are not familiar with OpenOffice Base. OpenOffice Base is the equivalent of Microsoft Access in the OpenOffice Open source suite. While it is not as feature rich as Microsoft Access, it has been getting increasingly better and has some unique features that even Microsoft Access lacks. Unfortuantely you can't just convert an access mdb to its format like you can with other Open office suite products - Word to Writer Writer to Word etc. However you can open MS Access databases in OOBase, but you can't take advantage of the forms and reports in an MS Access Database.

One thing I always liked about Microsoft Access was the ease with which you could link to various different kinds of datasources and generate rapid queries and so forth. Microsoft Access has a particular feature called Access Projects which ties it very closely with Microsoft SQL Server. What an MS Access Project does is connect you with a specific SQL Server database and allow you to browse all the objects, create forms and reports etc against the objects etc. Unfortunately MS Access Project only works with SQL Server. For other datasources you need to use linked tables and can't make design changes and browse a database as you can with Access Projects.

We had looked at Openoffice Base a while ago and thought they are making progress, but still not quite good enough to put to daily use. When we revisited Open Office Base recently, we were surprised to find a couple of neat nuggets.

1. They now had a native SDBC driver for postgresql instead of having to rely on the jdbc or odbc driver. You can still use the jdbc and odbc bridges, and unfortunately for Mac OSX users, you are stuck using the jdbc driver.
2. They have this Access Project like feature except it was better than Access in that it worked with other server side dbs. Any that had a driver - e.g. PostgreSQL, MySQL etc.
3. It had a relational designer viewer similar to what Access had and when we opened up a PostgreSQL db it laid out all the relationships we had carefully defined before with foreign key constraints etc.

In the next couple of sections we'll lay out how to quickly setup OpenOffice, install the native PostgreSQL driver and JDBC PostgreSQL driver and connect to a PostgreSQL database in OpenOffice Base. Please forgive us for using Windows paths in this. We figured it would be easier for people to follow since most users coming to this site are windows users and a lot of Linux users already use OO and paths are too different from Linux/Mac OSX installs.

Installing Open Office

1. Download open office from here and install. It takes about 5 minutes to install after download.

Installing the PostgreSQL Native SDBC driver

Please keep in mind that the PostgreSQL Native SDBC driver only works for Linux and Windows (not Mac), and is of beta quality. Meaning probably best not to fiddle around with a production database or at least have your db backed up.

1. Download postgresql-sdbc-0.7.5.zip from http://dba.openoffice.org/drivers/postgresql/index.html
2. Click on "C:\Program Files\OpenOffice.org 2.3\program\soffice.exe". Alternatively just open up any Open Office Writer.
3. Tools -> Extensions Manager -> Expand Office Org Extensions -> Click Add and point at the postgresql-sdbc-0.7.5.zip file (in earlier versions of Open Office e.g 2.1 and lower - this was under Tools->Package Manager)
4. Exit soffice and close any quick start soffice task items

Connecting to PostgreSQL from OOBase using SDBC driver

1. Start -> All Programs -> OpenOffice.org 2.3 -> OpenOffice.org Base
2. Connect to an existing database
3. Select postgresql which is probably way at the bottom
4. Click next
5. For connection settings - put in a connection to a postgresql db which should look something like:
   ```
   host=localhost dbname=somedb
   ```
6. Next - fill in username and password when prompted
7. Take default for remaining screens

Installing the PostgreSQL JDBC Driver

Note in general the PostgreSQL JDBC driver is said to be slower than the sdbc one since it goes thru a JDBC layer. We have not tested this theory. The JDBC driver however is more production quality and has the additional benefit of working in Mac OSX as well which is not currently supported by the SDBC driver.

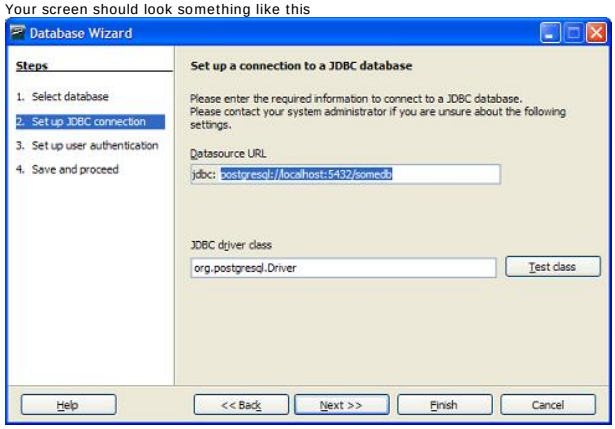
To install do the following

1. Download the JDBC 4 PostgreSQL 8.2 driver from http://jdbc.postgresql.org/download.html
2. Create a folder in C:\Program Files\OpenOffice.org 2.3\program\ called jdbcdrivers. It can be called anything really.
3. Copy the downloaded jar into that folder.
4. Click on "C:\Program Files\OpenOffice.org 2.3\program\soffice.exe" - Again you can just open "OpenOffice.org Writer"
5. Tools -> Options -> Java -> Class Path -> Add Archive -> point at the jdbcdrivers/postgresql-8.2-507.jdbc4.jar file you just created. **Note:** we tried using the Add Path and pointing at the folder, but that did not work.
6. Exit soffice and close any quick start soffice task items

Connecting to PostgreSQL from OOBase using JDBC driver

1. Start -> All Programs -> OpenOffice.org 2.3 -> OpenOffice.org Base
2. Connect to an existing database
3. Select JDBC which is the default.
4. Click next
5. In JDBC driver class - type *org.postgresql.Driver* - **Case is important**. and then click the **Test Class**. You should get a message that says loaded successfully.
6. For connection settings - put in a connection to a postgresql db which should look something like:
   ```
   postgresql://localhost:5432/somedb
   ```

   Your screen should look something like this



7. Next - fill in username and password when prompted
8. Take default for remaining screens

Differences between using the SDBC driver and JDBC driver

From our observation we noticed the following differences between the drivers

1. With the SDBC driver, you see the information_schema and pg_catalog schema. This does not seem to show using the JDBC driver.
2. You can create tables with both drivers, however, the SDBC driver seems incapable of creating serial columns in its current state while the JDBC one can.
3. Once a table is created, you can not edit it with the JDBC driver, but you can with the SDBC. Although the SDBC coughs when it sees a serial and insists on redefining it.Although it shows an

AutoValue Yes/No option. This did not seem to work.

So general conclusion. Stick with PgAdmin when creating tables and adding columns. Both drivers seem deficient in that area. Other caveat, OOBase seems to follow the proper casing paradigm of MS Access. This is annoying for PostgreSQL use, since it will by default create proper cased tables and field names which then will always need to be quoted. We didn't see a mechanism to switch this off.

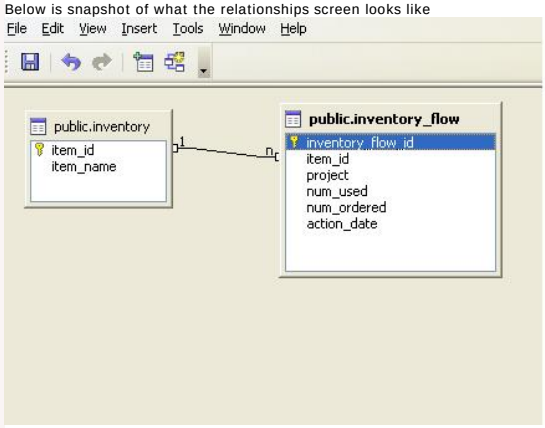
Viewing Relationships and Creating new ones

One thing that is nice about OOBase is that you can see your table relationships all laid out and add new ones. This seems to work equally well with both drivers. To do so do the following

1. In OOBase go to Tools -> Relationships
2. For the tables in PostgreSQL where you have already created foreign key constraints, you should see these nicely laid out
3. You can add new tables to the layout and draw lines between tables, right click properties to set/view cascade actions - similar to the way MS Access works.

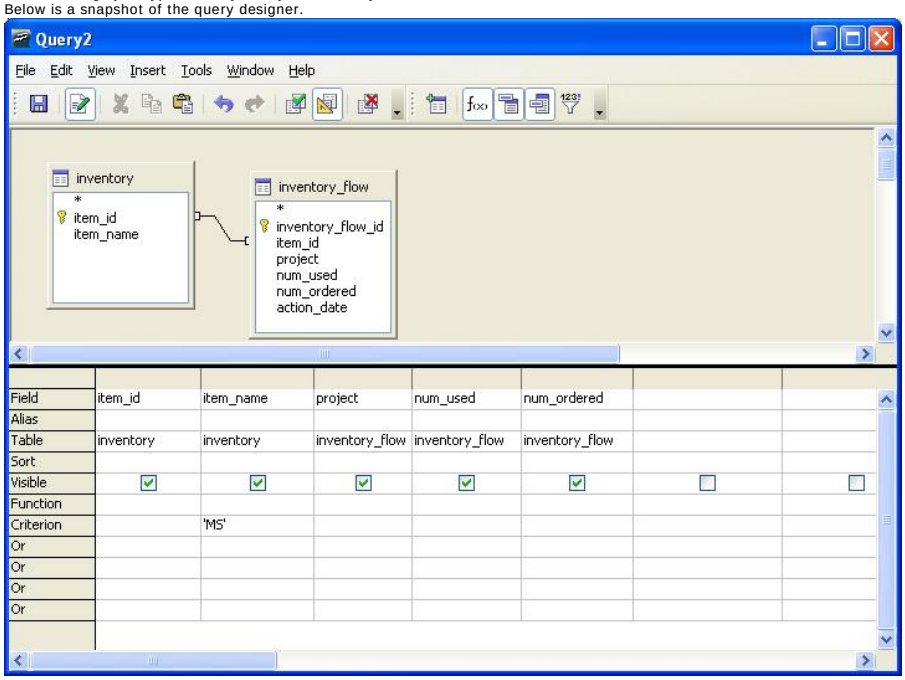
The layout is stored in the .odb file, but the actual foreign key constraints defined gets stored in the PostgreSQL database. Unfortunately we couldn't find a *Print Relationships* feature like what Microsoft Access has.

Below is snapshot of what the relationships screen looks like



Query Designer

The Query designer is a nice feature, but has some rough spots. If you are used to MS Access query designer, it has a similar feel. Links are automatically drawn when you drag in related tables, you can drag and drop links between two tables, right click to change join type. All very comfy and Accessy.
Below is a snapshot of the query designer.

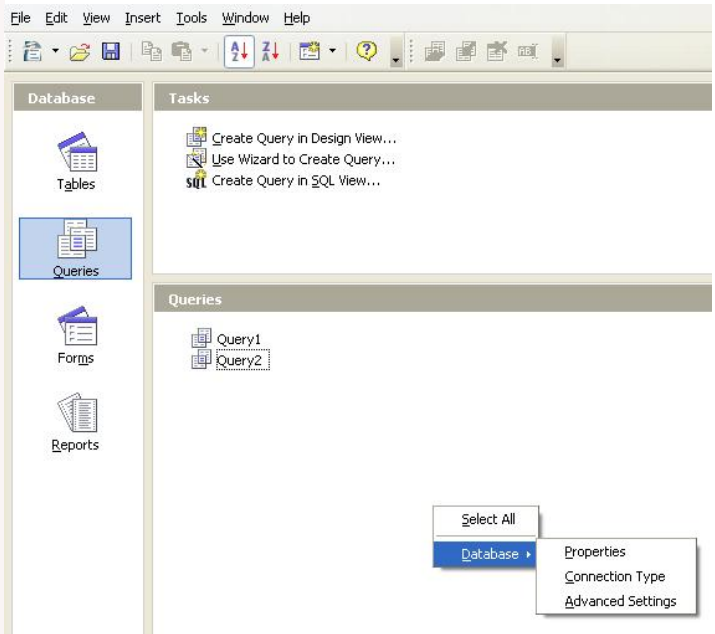


It seems to be able to create queries fine. We didn't really stress test though. Queries are saved in the .odb file not the PostgreSQL database.

NOTE: If you are using the Query Designer with PostgreSQL SDBC/JDBC, make sure to uncheck *Use Outer Join Syntax (OJ)* otherwise your LEFT and RIGHT JOIN queries will fail with a nasty error.

To get to where the OJ setting is,

1. switch to **Queries** tab
2. right mouse-click
3. Select Database->Advanced Settings->Special Settings

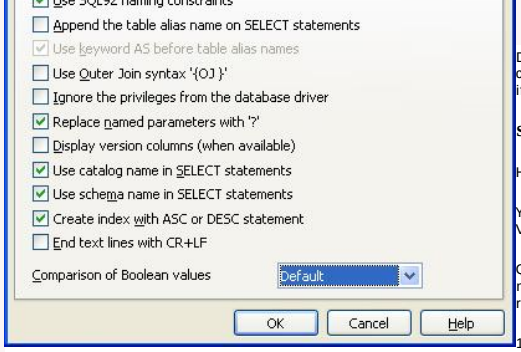


Advanced Settings Tab looks like this

Views

OOBase lets you graphically create views similar to query designer, and saves them in the database, but there doesn't seem to be a mechanism to see the SQL of them or change them once created. From then on they are treated as tables. Sometimes creating a view just doesn't work when you go to save.

Data Editing



Data can be edited from forms, queries, and tables, but not Views (at least not using the PostgreSQL SDBC/JDBC drivers). Data can be filtered and so forth. Again very similar to what you do with MS Access except in Access, you can edit data in linked Views if you denote a primary key. There doesn't seem to be a mechanism to do that in OOBase.

Some other useful features

Hiding Tables you don't care to see can be done easily with *Tools->Table Filter*.

You can run ad-hoc sql commands against the database with *Tools->SQL*. This is more designed for running action queries like Vacuum Analyze.

Query builder has an option for you to run direct SQL command directly. This mode is equivalent to MS Access - Pass-thru Query mode. This will allow you to use advanced features of PostgreSQL SQL dialect. Unfortunately if you choose this option, you can't really use the query designer very easily - although you can start with Query designer and then do the following:

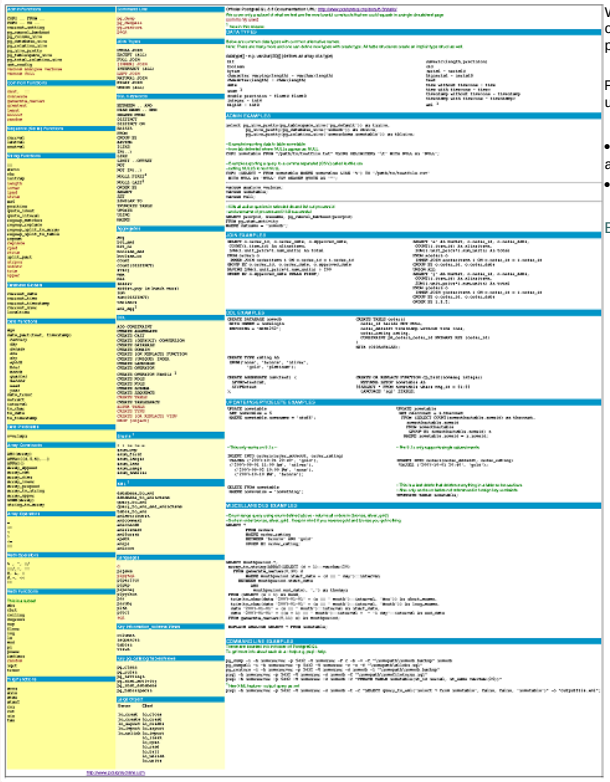
1. View->Switch Design View On/Off
2. Edit->Run SQL command directly

In upcoming version 2.5 of open office - I think its on the road map to allow Design view changes even in Run SQL command directly mode

Back to Table Of Contents

PostgreSQL 8.3 Cheat Sheet Overview

Below is a Thumbnail view of a PostgreSQL 8.3 Cheat Sheet that covers prior PostgreSQL constructs plus new 8.3 features. PDF version of this cheat sheet is available at PostgreSQL 8.3 Cheat sheet in PDF 8/12 by 11", PostgreSQL 8.3 Cheat sheet in PDF A4 and the PostgreSQL 8.3 Cheat sheet in HTML.



We apologize for the small size of the fonts. We tried to keep it to a minimum but had trouble deciding what to leave out to keep it to a one page sheet. So perhaps it would have been better as a 2 part cheatsheet. Anyrate we hope people find it useful.

For people who love cheat sheets, here are two other PostgreSQL cheat sheets on the web which we have found useful and cover slightly different set of things than we do in ours.

- Lorenzo Alberton's PostgreSQL cheat sheet - his is similar in structure to ours but for older versions of PostgreSQL and his focus is a little different.
- Pete Freitag's PostgreSQL cheat sheet - his is much simpler, but has several useful SQL examples.

Back to Table Of Contents

Reader Comments

Happy New Year

Dave
Even though I would never prefer MySQL over Postgres, I don't think features like these are withholding a lot of users from switching. Being able to scale postgres over more then 1 server is critical. Right now as a user we need to fall back to solutions as drbd or ill-featured, based on old versions, replication solutions.

For all it's weaknesses, MySQL does have working replication (at least I guess it's master/slave replication is usable).

Richard Broersma
I haven't used MySQL at all and I haven't yet used Slony as PostgreSQL's replication solution.
I am curious about the ways that Slony falls short as a Primary-Secondary replication solution when compared with MySQL's replication solution?

Sun Purchasing MySQL and PostgreSQL advances

Otis Gospodnetic
The real question to ask is whether Sun will keep investing in PG now that it owns MySQL, of course.

I look forward to that RESTy demo. How about a Java on the server side? Seems appropriate and would be useful to many!

Simon Kinsella
There's absolutely nothing to stop MySQL using the PostgreSQL storage engine, with or without the Sun deal. Postgres code is open source, freely available, and permissively licensed under the BSD license, enabling anyone to incorporate it into their own projects. Oracle, MS or anyone else could do the same, if they wanted to. Maybe some have?!

I would hope that the deal doesn't greatly affect Sun's interest in PostgreSQL development, as Postgres serves a different market to MySQL (albeit with some overlap of course). Not all Sun shops are web hosts!

sim

Ian McGowan
I too look forward to that REST application :-)

Otis Gospodnetic
What different market does PG serve? I've seen people say that a few times now. OLAP? What stops one from using MySQL for OLAP? Anything else?

This is an honest question! Thanks!

Jose Bustamante
Oracle is growing as a company only because of its acquisition.

I agree with your opinion. The things would be "a little bit" different if Oracle would have bought MySQL.

Regina
I wasn't thinking there was an issue in terms of licensing but in terms of just PR. Before there was a definite mild friction between MySQL and PostgreSQL so for MySQL to use any bit of PostgreSQL work would be seen as raising a white flag.

Given that Sun is presumably a close ally of PostgreSQL and they now own MySQL - I would presume where it makes sense they would want to integrate efforts. MySQL can take the corporate stance "My corporate masters made me do it." Not because PostgreSQL storage iis just better than MySQL? Also I'm not sure of that whole Falcon storage thing they have going. Has anyone seen it in action or is it some sort of vaporware?

Regina
There are a couple of big names. There is Netezza which originally at least was a spin-off of the PostgreSQL code and probably some PostgreSQL code still lives on in their product. They seem to be popular in large OLAP areas. Then there is GreenPlum Bizgres.

Why would anyone not use MySQL? I think people do but just because thats all they know, but frankly for any serious analytical stuff - the deficiencies in MySQL offerings in terms of SQL and functions, indexes etc. makes it a poor choice for any serious analytical analysis.

Jonathan Rogers
With all these big fish eating smaller fish, the whole IT industry is beginning to look like one big chess game.

Given Oracle's boldness, I wouldn't be surprised if Larry Ellison tries to take over Sun. Of course I think Sun has some friends in hidden pockets and some pockets they don't know they have that will come to their defense if such a silly thing were to take place.

Stored Procedures in PostgreSQL

Jason Lustig
The real question is: Are "Stored functions" (or whatever you will call them) pre-compiled by Postgres so that when you call them the database does not have to re-compile them? From what I understand, this is one of the major advantages of stored procedures in other databases, that by "stored" they mean "pre-compiled".

Richard Broersma Jr.
To me sematics are involved.

SELECT statements should do as they imply. The should build or select a result set based on a query construct.

Using SELECT to preform a batch of work that doesn't return a result set seems counter intuitive.

On the otherhand, CALL fits this role nicely.

Regina
pre-compiled I've always found to be a confusing term as far as databases are concerned and especially when talking about Postgres because there are so many choices of languages to build functions.

When people usually ask that question, it seems what they really mean to ask is can functions use cached plans. The answer is yes. In fact Postgres can use cached answers as well using the STABLE, IMMUTABLE and other function qualifications and of course the C functions are obviously compiled.

The idea that only stored procedures can use cached plans is old and for all intensive purposes is a myth. In most sophisticated databases it hasn't been true for over 10 years and therefore is a poor reason to choose a stored procedure.

For example if you write a dynamic query like in php or asp.net or whatever, SQL Server, Postgres, Oracle etc. can tell whether to cache a plan or not based on the pattern of the statement (even sometimes when its not a parameterized statement) so even those use cached (pre-compiled plans).

Regina
I have mixed feelings about this. On the one hand I think you are right its kind of unclear and probably should be avoided for clarity and maintainability, but on the other hand its useful for simulating for loops in sql that do something.

Take this example that kills all queries currently running by a particular user.

```
SELECT procpid, pg_cancel_backend(procpid)
FROM pg_stat_activity
WHERE usename = 'joeymemoryhogger';
```

Sure it may be considered a perverse thing to do, but sometimes perverse problems call for perverse solutions. :)

Anonymous Grammar Nazi
"For all intensive purposes" -> "For all intents and purposes"

Regina
Thanks for the catch. We've corrected it.

Pavel Stehule
You can use prepared statement outside stored procedures. But who do it? With stored procedures you have prepared statements gratis, without any more work. Second, with stored procedures you can better decompose application. But main impact on speed is minimalisation of network and conversion traffic. plpgsql variables are in native PostgreSQL format, when you would do same work outside, you have to do lot of conversation:
server,tcp <->tcp, libpq, drivers, com, ..

Leo and Regina
I think its common at least in .NET to write prepared statements w/o stored proces and a lot of that is done for you in .NET with data adapters etc. and a lot of ORM wrapper type classes I think do it too.

For simple inserts and updates we usually don't bother with stored procs, but like you said for real stuff where you've got lines and lines of code especially when its used in multiple sections of an app, we usually use stored procs and functions because it compartmentalizes the logic nicely and keeps network traffic (of transfering the statements across) to a minimum.

Matt Peters
Using stored procs/fns provide increased database security and architectural decoupling for data-driven applications that cannot be matched through ORM or prepared statements. For example - you must take complicated measures to prevent SQL injection whenever ad-hoc or prepared statements are allowed. As for decoupling: the wide and deep table that worked well in proof-of concept stages of development may require refactoring, normalization, partitioning, etc when preparing for production. With the defined interface of a stored proc/fn, the application tier does not need to be made aware of the details of the database design changes. It is akin to providing a service interface without exposing the inner structures of the tier.

Pete
You sure your comment about prepared statements is right?
Writing prepared statements more or less forces you to define the type of parameters at least in jdbc and ado.net.

Although I guess it would be possible to create a badly implemented driver that doesn't protect you, but I haven't seen that.

For example in java jdbc one would write a prepared statement of the form
"Update atable set avalue = ?, anothervalue = ? WHERE anid = ?";

And then you would do
st.setString(1, "new value for avalue");
st.setInt(2, 5);

Note the setString, setInt etc. forces you to declare what is past in and throws an error if it fails. In all drivers I've worked with
setString properly escapes quotes etc.

Now on the other hand - I have seen people write easy to hack stored procs to get around the limitations of stored procs.
E.g. if you do a lot of analytical apps - writing every permutation of criteria as stored procs gets tiring. So
some misguided folks, thinking stored procs are the holy grail of sql injection protection
write a stored proc that does something like this the below. This is pseudo stored proc code.

CREATE PROCEDURE adhocsql(@somewhere varchar(8000)) AS
EXECUTE("SELECT * FROM atable WHERE " + @somewhere);

In this case - no matter how you call the above - even in a prepared call, you better really sanitize that @somewhere. You would have been better doing an adhoc sql query.

SQL Math Idiosyncracies

Peter Eisentraut
This is actually not entirely correct. The SQL standard says that precision and scale of the division result are implementation-defined. So all of these guys including MySQL and Access are correct as far as the SQL standard is concerned. Oracle will also give you a fractional result for 3/1 and 3/2.

Regina
So are you saying that because Oracle's internal implementation of integer is data type number with 0 scale then its perfectly valid for it to return fractional results -- since it is still returning a number and scale and precision are implementation defined?

I guess I hadn't quite thought about it that way. I always thought of integer and numeric being different data types.

Deleting Duplicate Records in a Table

Symbiatch
Have you checked if this is faster or slower than the form I've seen used many times and have gotten used to:

delete from tab a where exists (select 1 from tab b where a.uniq1=b.uniq1 and a.uniq2=b.uniq2 and a.prkey>b.prkey)

Richard Broersma Jr.
This method depends upon a unique id. If an auto-number wasn't designed onto a table, the table.CTID could be used in-place of this.

Since the CTID is a postgresql-ism, some don't like to use it. But it is an option that is available to use in a case like this.

Leo and Regina
Glad you asked. For this particular example we chose not to show that approach since it was considerably slower than the above (so slower we don't bother waiting for it to finish). I suspect it depends on if you have indexes on the dupe fields and the ratio of duplicates to non-dupes. This example is odd in that there are more duplicates than actual rows we are keeping. So we may try it when its the reverse case and it may win out.

Just FYI. Writing this example with the exists would be something like

DELETE FROM duptest a WHERE EXISTS (SELECT 1 FROM duptest b WHERE a.first_name=b.first_name and a.last_name=b.last_name and a.name_key < b.name_key)

Setting up PgAgent and Doing Scheduled Backups

Paolo saudin
Very nice and useful post :-) !!
Thanks

Albert Cervera i Areny
This seems a really cool addition to PgAdmin III. Congratulations and thank you!

Jan
Very good post.

I just missed some information about setting up the daily backup job with the local authorization set to password on Windows. Passing the password to pgAgent with password=**** is not sufficient (and not save), as pg_dump requires a password too!
This is not easy to detect, as pgAgent just waits for infinity.

So setting the password in %APPDATA%\postgresql\pgpass.conf finally solved the issue.Now my db will get its daily backup :)

Thank you.

The Anatomy of PostgreSQL - Part 2 - Database Objects

Thom Brown
This is a great overview on database objects! Thanks for posting it.

gigiduru
Awesome introspection of the postgresql database. Keep 'em coming!!!

CrossTab Queries in PostgreSQL using tablefunc contrib

SunWuKung
This is nice, but as I see it always presumes that you know your data before you do the crosstab. Even if you specify your columns with an sql statement you still need to enumerate all resulting columns individually in the As mthreport(...) part.
I have searched extensively but could not find a plpgsql based solution for the situation where you don't know what the categories will be. If you have any solution for that please let me know.
Thx.
SWK

Regina
Good question. Sadly I don't think there is an easy answer. To get around the issue say in PHP and so forth, we use dummy names in our sql statement or a dummy type with lots of output columns and then in PHP to display the header, loop thru our category sql recordset.

The problem is not so much with crosstab as with PostgreSQL inability to deal with dynamic record types or ability to do record introspection. This has been discussed as a future enhancement for example here

http://archives.postgresql.org/pgsql-patches/2005-07/msg00458.php
But unfortunately haven't heard any recent talk of it.

Denis Bitouzé
I'm pretty new in RDBMS and in PostGreSQL, and I recently discovered crosstab utility so maybe I'm wrong but, as you say "I have searched extensively but could not find a plpgsql based solution for the situation where you don't know what the categories will be", did you have a look at

http://www.ledscripts.com/tech/article/view/5.html ?

I am also looking for a solution for dynamical categories...

Cheers,
--
Denis

Using MS Access with PostgreSQL

Richard Broersma Jr.
Here is one other point about Access and ODBC linked booleans:

It seems that access sees ODBC nulls as false. When Access tries to update a field with a boolean null, the update will fail since ACCESS uses all of the table's old column values in the where clause of an update statement:

WHERE ...
AND boolean_field = 'false'::boolean
AND ...

However, since boolean_field actually is null the update fails.

David Fetter
It's a shame Access hasn't been updated since the 7.4 series, which has long, white whiskers on it.

Regina
I always found it strange that the PostgreSQL ODBC driver says 7.4 on it even though well it obviously works with 8.0 versions and has been continuely updated. I think it is mostly a labeling issue on the ODBC driver but would be good PR to say 7.4-8.3 or something like that.

As far as functionality, I don't think it would make too much of a difference where ODBC is concerned.

Using OpenOffice Base 2.3.1 with PostgreSQL

pabloj
What about the ODBC driver, which is probably the most familiar to windows users?

Leo Hsu and Regina Obe
We didn't include ODBC because from what we have read compared to the jdbc and sdbc drivers it is not as capable as far as Open Office is concerned. On top of that it only works on windows.

In another article we will demonstrate using MS Access (doing linked tables and pass thru queries with PostgreSQL). In that article we will cover using the ODBC driver.

Cosmin
I struggled for a few hours to connect OpenOffice to a PostgreSQL database as the documentation is really scarce. I found your step by step tutorial excellent. Thank you!

PostgreSQL 8.3 Cheat Sheet Overview

Anders
Your "DML examples" are "DDL examples".

(and your comment functionality gives a quite verbose error if cookies are disabled)

pyre
You should think about making that cheatsheet image a PNG rather than a JPG. The fonts are small enough that JPG can't really handle it without the compression mangling the characters. It takes an effort to read some of that JPG.

Leo and Regina
Thanks we've corrected.

Richard Heycock
Any change of producing an A4 pdf?

roScripts - Webmaster resources and websites
PostgreSQL 8.3 Cheat Sheet Overview - Postgres OnLine Journal

Leo and Regina
We've added an A4 version. Unfortunately we don't have any A4 paper to verify if it prints out right. Let us know how that works out.

Richard Heycock
It's pretty good. If you could centre it that would be even better but I really am nit picking :-)

Anyway thanks for doing that I now have a copy pinned on the wall next to my desk.

Official PostgreSQL 8.3 Documentation URL: http://www.postgresql.org/docs/8.3/static/

We cover only a subset of what we feel are the most useful constructs that we could squash in a single cheatsheet page

commonly used

[1] New in this release.

DATA TYPES

Below are common data types with common alternative names.
Note: There are many more and one can define new types with create type. All table structures create an implicit type struct as well.

datatype[] - e.g. varchar(50)[] (defines an array of a type)

```
bit
boolean
bytea
character varying(length) - varchar(length)
character(length) - char(length)
date
enum [1]
double precision - float4 float8
integer - int4
bigint - int8
numeric(length,precision)
oid
serial - serial4
bigserial - serial8
text
time without timezone - time
time with timezone - timez
timestamp without timezone - timestamp
timestamp with timezone - timestampz
xml [1]
```

ADMIN EXAMPLES

```
select pg_size_pretty(pg_tablespace_size('pg_default')) as tssize,
       pg_size_pretty(pg_database_size('somedb')) as dbsize,
       pg_size_pretty(pg_relation_size('someschema.sometable')) as tblsize;
```

```
--Example importing data to table sometable
--from tab delimted where NULLs appear as NULL
COPY sometable FROM "/path/to/textfile.txt" USING DELIMITERS '\t' WITH NULL As 'NULL';
```

```
--Example exporting a query to a comma separated (CSV) called textfile.csv
--setting NULLS to text NULL
COPY (SELECT * FROM sometable WHERE somevalue LIKE '%') TO '/path/to/textfile.csv'
 WITH NULL As 'NULL' CSV HEADER QUOTE AS '"';
```

```
vacuum analyze verbose;
vacuum sometable;
vacuum full;
```

```
--Kills all active queries in selected db and list out process id
--and username of process and if kill successful
SELECT procpid, usename, pg_cancel_backend(procpid)
FROM pg_stat_activity
WHERE datname = 'somedb';
```

JOIN EXAMPLES

```
SELECT o.order_id, o.order_date, o.approved_date,
 COUNT(i.item_id) As nlineitems,
 SUM(i.unit_price*i.num_units) As total
FROM orders o
 INNER JOIN orderitems i ON o.order_id = i.order_id
GROUP BY o.order_id, o.order_date, o.approved_date
HAVING SUM(i.unit_price*i.num_units) > 200
ORDER BY o.approved_date NULLS FIRST;
```

```
SELECT 'x' As bucket, o.order_id, o.order_date,
 COUNT(i.item_id) As nlineitems,
 SUM(i.unit_price*i.num_units) As total
FROM xorders o
 INNER JOIN xorderitems i ON o.order_id = i.order_id
GROUP BY o.order_id, o.order_date
UNION ALL
SELECT 'y' as bucket, o.order_id, o.order_date,
 COUNT(i.item_id) As nlineitems,
 SUM(i.unit_price*i.num_units) As total
FROM yorders o
 INNER JOIN yorderitems i ON o.order_id = i.order_id
GROUP BY o.order_id, o.order_date
ORDER BY 1,3,2;
```

DDL EXAMPLES

```
CREATE DATABASE somedb
 WITH OWNER = somelogin
 ENCODING = 'WIN1252';
```

```
CREATE TABLE orders(
  order_id serial NOT NULL,
  order_addeddt timestamp without time zone,
  order_rating rating,
  CONSTRAINT pk_orders_order_id PRIMARY KEY (order_id)
)
WITH (OIDS=FALSE);
```

```
CREATE TYPE rating AS
 ENUM('none', 'bronze', 'silver',
       'gold', 'platinum');
```

```
CREATE AGGREGATE sum(text) (
  SFUNC=textcat,
  STYPE=text
);
```

```
CREATE OR REPLACE FUNCTION cp_test(somearg integer)
  RETURNS SETOF sometable AS
$$SELECT * FROM sometable where msg_id = $1;$$
  LANGUAGE 'sql' STABLE;
```

UPDATE/INSERT/DELETE EXAMPLES

```
UPDATE sometable
 SET somevalue = 5
 WHERE sometable.somename = 'stuff';
```

```
UPDATE sometable
 SET calccount = s.thecount
  FROM (SELECT COUNT(someothertable.someid) as thecount,
    someothertable.someid
    FROM someothertable
     GROUP BY someothertable.someid) s
  WHERE sometable.someid = s.someid;
```

```
--This only works on 8.1+ --
INSERT INTO orders(order_addeddt, order_rating)
 VALUES ('2007-10-01 20:40', 'gold'),
 ('2007-09-01 11:00 AM', 'silver'),
   ('2007-09-02 10:00 PM', 'none'),
   ('2007-10-10 PM', 'bronze');
```

```
--Pre 8.1+ only supports single values inserts
INSERT INTO orders(order_addeddt, order_rating)
 VALUES ('2007-10-01 20:40', 'gold');
```

```
DELETE FROM sometable
 WHERE somevalue = 'something';
```

```
--This is a fast delete that deletes everything in a table so be cautious.
--Also only works on tables not referenced in foreign key contraints
TRUNCATE TABLE sometable;
```

MISCELLANEOUS EXAMPLES

```
--Enum range query using enum defined above - returns all orders in (bronze, silver, gold)
--Sorts in order bronze, silver, gold.  Keep in mind if you reverse gold and bronze you get nothing
SELECT *
       FROM orders
       WHERE order_rating
       BETWEEN 'bronze' AND 'gold'
       ORDER BY order_rating;
```

```
SELECT monthperiod.*,
 array_to_string(ARRAY(SELECT (d + 1)::varchar(20)
   FROM generate_series(0,30) d
     WHERE monthperiod.start_date + (d || ' day')::interval
    BETWEEN monthperiod.start_date
        AND
         monthperiod.end_date), ',') as thedays
 FROM (SELECT (n + 1) As mnum,
  trim(to_char(date '2007-01-01' + (n || ' month')::interval, 'Mon')) As short_mname,
  trim(to_char(date '2007-01-01' + (n || ' month')::interval, 'Month')) As long_mname,
  date '2007-01-01' + (n || ' month')::interval As start_date,
  date '2007-01-01' + ((n + 1) || ' month')::interval +  - '1 day'::interval As end_date
 FROM generate_series(0,11) n) As monthperiod;

 EXPLAIN ANALYZE SELECT * FROM sometable;
```

COMMAND LINE EXAMPLES

These are located in bin folder of PostgreSQL
To get more info about each do a --help e.g. psql --help

```
pg_dump -i -h someserver -p 5432 -U someuser -F c -b -v -f "\somepath\somedb.backup" somedb
pg_dumpall -i -h someserver -p 5432 -U someuser -c -o -f "\somepath\alldbs.sql"
pg_restore -i -h someserver -p 5432 -U someuser -d somedb -l "\somepath\somedb.backup"
psql -h someserver -p 5432 -U someuser -d somedb -f "\somepath\somefiletorun.sql"
psql -h someserver -p 5432 -U someuser -d somedb -c "CREATE TABLE sometable(st_id serial, st_name varchar(25))"
```

[1] New XML feature - output query as xml
-P "t" only output rows

```
psql -h someserver -p 5432 -U someuser -d somedb -P "t" -c "SELECT query_to_xml('select * from sometable', false, false, 'sometable')" -o "outputfile.xml";
```

Admin Functions

```
COPY .. FROM ..
COPY .. TO ...
current_setting
pg_cancel_backend
pg_column_size
pg_database_size
pg_relation_size
pg_size_pretty
pg_tablespace_size
pg_total_relation_size
set_config
vacuum analyze verbose
vacuum full
```

Common Functions

```
cast,  ::
coalesce
generate_series
greatest
least
nullif
random
```

Sequence (Serial) Functions

```
currval
lastval
nextval
```

String Functions

```
||
ascii
chr
initcap
length
lower
lpad
ltrim
md5
position
quote_ident
quote_literal
regexp_matches
regexp_replace
regexp_split_to_array
regexp_split_to_table
repeat
replace
rpad
rtrim
split_part
strpos
substr
trim
upper
```

Database Globals

```
current_date
current_time
current_timestamp
current_user
localtime
```

Date Functions

```
age
date_part(text, timestamp)
 century
 day
 decade
 dow
 doy
 epoch
 hour
 month
 quarter
 second
 week
 year
date_trunc
extract
interval
to_char
to_date
to_timestamp
```

Date Predicates

```
overlaps
```

Array Constructs

```
ANY(array)
ARRAY[4,5,6],..]
ARRAY()
array_append
array_cat
array_dims
array_lower
array_prepend
array_to_string
array_upper
SOME(array)
string_to_array
```

Array Operators

```
=
<>
<
>
<=
||
```

Math Operators

```
% , ^, |/
||/,!, !!
@, &, |
#,~, <<
>>
```

Math Functions

This is a subset

```
abs
cbrt
ceiling
degrees
exp
floor
log
ln
mod
pi
power
radians
random
sqrt
trunc
```

Trig Functions

```
acos
asin
atan
atan2
cos
cot
sin
tan
```

Command Line

```
pg_dump
pg_dumpall
pg_restore
psql
```

JOIN Types

```
CROSS JOIN
EXCEPT (ALL)
FULL JOIN
[INNER] JOIN
INTERSECT (ALL)
LEFT JOIN
NATURAL JOIN
RIGHT JOIN
UNION (ALL)
```

SQL Keywords

```
BETWEEN .. AND
CASE WHEN .. END
DELETE FROM
DISTINCT
DISTINCT ON
EXISTS
FROM
GROUP BY
HAVING
ILIKE
IN(..)
LIKE
LIMIT ..OFFSET
NOT
NOT IN(..)
NULLS FIRST[1]
NULLS LAST[1]
ORDER BY
SELECT
SET
SIMILAR TO
TRUNCATE TABLE
UPDATE
USING
WHERE
```

Aggregates

```
avg
bit_and
bit_or
boolean_and
boolean_or
count
count(DISTINCT)
every
max
min
stddev
stddev_pop (a bunch more)
sum
sum(DISTINCT)
variance
xml_agg[1]
```

DDL

```
ADD CONSTRAINT
CREATE AGGREGATE
CREATE CAST
CREATE (DEFAULT) CONVERSION
CREATE DATABASE
CREATE DOMAIN
CREATE [OR REPLACE] FUNCTION
CREATE (UNIQUE) INDEX
CREATE LANGUAGE
CREATE OPERATOR
CREATE OPERATOR FAMILY [1]
CREATE ROLE
CREATE RULE
CREATE SCHEMA
CREATE SEQUENCE
CREATE TABLE
CREATE TABLESPACE
ALTER TABLE
CREATE TYPE
CREATE [OR REPLACE] VIEW
DROP [object]
```

Enums [1]

```
> < <= >= =
enum_cmp
enum_first
enum_larger
enum_last
enum_range
enum_smaller
```

XML [1]

```
database_to_xml
database_to_xmlschema
query_to_xml
query_to_xml_and_xmlschema
table_to_xml
xmlattributes
xmlcomment
xmlconcat
xmlelement
xmlforest
xpath
xmlpi
xmlroot
```

Languages

```
c
pljava
plpgsql
plperl(u)
plphp
plproxy
plpython
plr
plruby
plsh
pltcl
sql
```

Key information_schema Views

```
columns
sequences
tables
views
```

Key pg_catalog Tables/Views

```
pg_class
pg_rules
pg_settings
pg_stat_activity
pg_stat_database
pg_tablespaces
```

Large Object

Server	Client
lo_creat	lo_close
lo_create	lo_creat
lo_export	lo_create
lo_import	lo_export
lo_unlink	lo_import
	lo_lseek
	lo_open
	lo_read
	lo_tell
	lo_unlink
	lo_write