Conversion of Microsoft SQL/ASP applications to PostgreSQL

Written by Ethan Townsend, Produced by Project A

6/23/05

Introduction:

This manual was compiled by Project A Web Development (http://www.projecta.com) as a project supported by Jim Teece. It was written by Ethan Townsend as a documented means of migrating our ADO ASP application, SIB (Site-in-a-box), from Microsoft SQL Server 7.0 to an open source database. SIB is a large database-driven application that includes many views, stored procedures, and complex SQL executed from the ASP pages. The primary goals were to:

1) Modify the application so that it could seamlessly switch between the existing SQL Server database and the new open source database.

2) Have the ability to offload existing databases from SQL Server to the new database server.

3) Make no modifications to the existing, working SQL Server.

4) Maximize ease-of-use, stability, performance, and security.

The following open source database servers were installed and analyzed to determine their feasibility in replacing SQL Server:

- MySQL 4.1: The stable version of MySQL at the time of testing.
- MySQL 5: The development version of MySQL.
- PostgreSQL 7.4:

- PostgreSQL 8.0.3:
- Firebird 1.5.3:
- MaxDB 7.5:
- MaxDB 7.6:

After reviewing the alternatives, we settled on PostgreSQL 8.0 for the following reasons:

- Good documentation,
- Best performance on Windows platform (using the included ODBC driver).
- Custom operators and functions allowed a compatibility layer to minimize code changes (only about 300 changes were necessary in ~700,000 lines of code).
- Easy to install and use (a GUI was included and the installer was straightforward).

Part 1: Installing PostgreSQL on Windows XP

PostgreSQL's download site (http://www.postgresql.org/download/) has 2 items needed to install and use a new PostgreSQL server, the first is the database program.

1) Get the windows installer from the web site (from http://www.postgresql.org/ftp/binary/v8.0.3/win32/postgresql-8.0.3.zip as of this writing)

2) Checksum the file (always a good idea) and run it on the windows machine. Be sure to have the installer: "enable remote connections" and "Run as a service"

3) The database can now be accessed through the pgAdmin III program under the start menu.

4) In order to allow remote machines to connect to the database, edit the pg_hba.conf file (linked through the start menu)

The file includes examples, but basically the format goes like this:

```
host   <database>   <user>        <ip-address>        md5
```

This line allows the database username <user> to connect to database <database> from <ip-address>. md5 encryption is used for authentication. For example: add the following line to allow any computer with a 10.10.0.* IP address to access the database:

host all all 10.10.0.1/24 md5

Or add a line for each computer/user that will be using the database for increased security.

5) By default, PostgreSQL only supports 100 concurrent connections, if you need more than this, edit **max_connections** in postgresql.conf (also linked through the start menu).

6) You can fine-tune fine-tune Postgres' memory parameters to suit the amount of RAM in the server by editing postgresql.conf,

7) Be sure to restart the service (use the shutdown and start scripts in the start menu or directly from the services control panel) in order for changes to take effect.

The second item is the PostgreSQL ODBC driver.

1) Download the windows installer from the ODBC project site (http://gborg.postgresql.org/project/psqlodbc/) and checksum.

2) Install the ODBC driver on the Webserver to allow the web application to connect to the database.

3) Install on the MS SQL Database Server to allow exporting data from the old MS SQL Server to the new PostgreSQL server.

Part 2: Recreating old databases on the new database Server. (On MS SQL Version 7.0)

Next you need to create a new database on the PostgreSQL server that is compatible with the old database on SQL server (that is, has the same tables, views, accepts the same information, has the same stored procedures and triggers, etc.). This consists of 3 steps:

1) Exporting a T-SQL script from SQL Server that contains the script necessary to recreate an empty database.

2) Converting the script to PostgreSQL-compatible SQL script.

3) Importing the SQL script into PostgreSQL.

Note: The MS Data transfer wizard (Part 3) will attempt to create tables automatically, but it ignores key information, defaults, and doesn't transfer views, and more often than not, it fails. It is simpler to create the entire database structure by hand and then transfer data into it. This also allows you more

control over which PostgreSQL data types you want to map existing SQL Server data types.

1) Log into the MS SQL database server and open Enterprise Manager.

2) Select the database to copy, and click "Generate SQL Scripts"

3) Be sure to go to the options tab and check "Script Triggers" and "Script PRIMARY Keys, FOREIGN Keys, Defaults and Check Constraints"

4) Click OK and save the SQL script to disk.

This Script contains the T-SQL script necessary to recreate the database, this T-SQL will need to be edited and converted to standard SQL in order to be run on PostgreSQL.

The following changes were necessary for our application. Many of these differences may apply to your database and possibly other syntax changes as well, depending upon the complexity of the database.

Many of these are made much easier with a awk/sed script, look at the attached bash script (mssqltopgsql.sh) before implementing these changes by hand.

There are 4 major sections to the SQL script:

1) CREATE TABLE section

2) ALTER TABLE/CREATE INDEX section

3) CREATE VIEW section

4) STORED PROCEDURE/TRIGGER section.

Syntax changes necessary in all section:

The first things to fix are non SQL-compliant T-SQL commands and syntax that are unique to SQL Server and unavailable on PostgreSQL, this is applied to the entire script.

1) Elimination of brackets (**[,]**), object owner prefixes (**dbo.**) and file group keywords (**ON PRIMARY, TEXTIMAGE_ON, CLUSTERED**) NOTE: This means that you will lose any advantages from custom file groups, if file group control is needed, it can be done with a PostgreSQL **tablespace**.

2) Elimination of unsupported commands (like **SET ANSI_NULLS** and **SET QUOTED_IDENTIFIERS**).

3) Elimination of **WITH NOCHECK** keywords. PostgreSQL does not support checking constraints against existing data, so this is redundant. If you need to check constraints, be sure to add the constraint before inserting data.

4) Conversion of **GO** keywords into semicolons (**;**)

5) In PostgreSQL, all tables, views, column names etc. are returned in lower case, it's probably simplest to convert the entire script to lower case before entering it into the database (the database is case insensitive, but not case preserving, which means that running **CREATE TABLE TheTable;** will create a table called **thetable**, but any commands made referring to **TheTable** will perform as expected).

6) When referencing system objects, the format is different. In the first part of the SQL script, if you chose to script drop statements, you will see something like this:

```
if exists (select * from sysobjects where
        id = object_id(N'[dbo].[<tablename>]')
        and OBJECTPROPERTY(id, N'IsTable') = 1)
drop table [dbo].[<tablename>]
GO
```

In Postgresql, these objects are typically stored in pg_<object>, for example, tables are stored in **pg_tables** and views are stored in **pg_views**. You can see if a table exists with this command:

```
select count(*) from pg_tables where tablename = '<tablename>';
```

However, since you will be creating this database from a blank template, there is no need to use drop statements. The entire database can be dropped if an error occurs.

The major changes to the **CREATE TABLE** section will be:

1) Converting MS SQL data types to supported data types in PostgreSQL:

- **varchars** or **nvarchars** can be mapped to **varchars**. Unicode and ASCII can both be stored in PostgreSQL varchars, this is determined by the database encoding, be sure to set the default when you create the database.

- similarly, **ntext** and **text** can be mapped to **text** data type, however, PostgreSQL **text** is no different than PostgreSQL **varchar**. The **varchar** type, however, can hold up to 1GB of information (1 GB is the max for each row), much more than the SQL Server record limit of about 8K. Because of this, most **text** fields can actually be stored as **varchar**s, which may be preferable to text for convenience (no more TEXTPTRs).

- **datetime** and **smalldatetime** can usually be mapped to **timestamptz**. This adds precision to the stored data in the case of **smalldatetime**.

  If a **datetime** column is only used to store the time, then **time** should be used instead. If only the date is stored, then **date** may be used instead.

  In addition, PostgreSQL has an **interval** column type that stores a length of time. This may be preferable if you are currently using a **datetime** column for this.

- **bit** represents a boolean value, however, instead of being mapped to **boolean**, these could be converted to **int2** or **smallint** (the record will take twice as much storage space) because it behaves more like SQL Server's **bit**. Specifically, it can accept integer values (be sure to check the TrueIsMinus1=1 flag in the connection string or DSN, since this is what ADODB uses) **boolean** values need to be assigned and compared with '0' and '1', with quotes around them (SQL Server does not require quotes).

- **image** types or other binary data blobs can be mapped to **bytea** data type (for small binary data), or **lo** data type, which is more compatible with SQL Server's datatypes when using ADO.

- **bytea** type is not good for large (> 100 MB) binary data, and they can't be used for sizes larger than 1GB. Instead, use the PostgreSQL extension that allows **lo** data type.

  Note that **lo** are stored in a special table and have associated security risks and complexities (You need to delete the OID reference to the **lo** as well as the **lo** itself.) These are more similar to SQL Server's binary data types.

- **int** and **numeric** datatypes should be fine.

- **decimal** types are supported in PostgreSQL, but this is obsolete, they should be mapped to **numeric** to ensure future compatibility.

- **tinyint** types can be mapped to **smallint**, which uses 2 bytes instead of 1.

- **uniqueidentifier** column types do not exist in PostgreSQL, bigint is usually an acceptable substitute (see below).

2) Instead of **int IDENTITY(1,1)** for an auto-incrementing column, Postgres uses sequences, the easiest way to transform this is to use the **serial** (or **bigserial**

for 8 byte numbers) data type. If you need to start the sequence at a specified seed or have it decrement or increment by more than 1, you must create the sequence manually. To do this, use PostgreSQL's **CREATE SEQUENCE** command, like this:

```
CREATE SEQUENCE <sequence name> START <start value> INCREMENT
<increment value>;
```

And then assign the default value for the column to use this sequence, using the **nextval** function:

```
CREATE TABLE <table name> ( <colname> int default nextval('<sequence
name>' ) not null
```

You may need to assign **MINVALUE** if you want negative numbers

```
CREATE SEQUENCE <sequence name> MINVALUE <minvalue>;
```

3) PostgreSQL does not have a **uniqueindentifier** type, often, however, using a bigserial type should work instead, but this will create a column that is unique only to that table. If you need an identifier that is unique across tables, create all columns of type int or int8, and specify default values to use the next value of a single sequence (as in the part on **IDENTITY** columns above). The same sequence must be used for all columns.

To do this, create a sequence and have all columns that must be globally unique use this same sequence. You probably want to use bigint type to allow large numbers (sequences only use 8 bytes, so larger values are not possible).

If you need the id to be globally unique (across multiple computers) then you may need to use char(40) and write your own function for creating the GUID.

4) PostgreSQL doesn't support a column name of "default" (SQL Server does) , change any column names called "default" and note which ones were changed so that they can be changed in the views and stored procedures and ASP code later.

In the **ALTER TABLE** section:

1) MSSQL applies default values as constraints, PostgreSQL does not. Instead of adding a constraint for a default value, use the following syntax:

```
ALTER TABLE <tablename> ALTER <columnname> SET DEFAULT
(<value>);
```

Alternatively, default values can be added in the column declaration line.

2) Be sure to remove any default values that call unsupported functions (e.g. newid())

3) Other **ALTER TABLE** statements (foreign keys, primary keys, check constraints) should work.

In the **CREATE VIEW** section apply any relevant syntax changes from above (syntax, datatypes, etc.) and look for the additional differences:

1) When selecting a limited number of rows, use **LIMIT** instead of **TOP**. For example: instead of **SELECT TOP 10** <statement> use **SELECT** <statement> **LIMIT 10**. etc.

2) Views in PostgreSQL can not be modified! That means that you cannot insert, update, or delete directly from a view. However, if you need to insert or update into a view, PostgreSQL rules can be used to emulate updateable views.

You can use PostgreSQL rules (**CREATE OR REPLACE RULE**) to rewrite queries so that they affect the underlying table instead of the view.

As a simple example, suppose you have a table "testtable" that has a column "testvalue" and a view "testview" that presents "testvalue" as "presentvalue":

```
CREATE TABLE testtable (testvalue int);

CREATE VIEW testview as select (testvalue) as
presentvalue from testtable;
```

Then, if you want to be able to insert into the view, use **NEW** and **OLD** to specify the view's column labels. **NEW** represents the newly changed/inserted values of the row (in the view) being updated or inserted, **OLD** represents the old value of the row in the view that is being changed.

```
CREATE OR REPLACE RULE rule_testview_insert AS ON INSERT TO
testview DO INSTEAD INSERT INTO testtable (testvalue) VALUES (NEW.
presentvalue);
```

If you want to be able to update the view:

```
CREATE OR REPLACE RULE rule_testview_update AS ON UPDATE TO
testview DO INSTEAD UPDATE testtable SET testvalue = NEW.presentvalue
where testvalue = OLD.presentvalue;
```

If you want to be able to delete from the view:

CREATE OR REPLACE RULE rule_testview_delete AS ON DELETE TO testview DO INSTEAD DELETE FROM testtable where testvalue = OLD.presentvalue;

This should simulate the functionality of an updateable view on "testview"

For more complicated views, the queries can be customized in order to do whatever is necessary.

The **CREATE INDEX** commands should be fine. Although PostgreSQL offers partial indexing, which may provide a performance/storage improvement over SQL Server depending upon your structure.

In **CREATE PROCEDURE** and **CREATE TRIGGER** sections:

1) Creating new procedures in PostgreSQL is probably the most difficult and time-consuming part of migration. The SQL Server stored procedures must be re-written in another language, PostgreSQL supports SQL, PL/pgsql (similar to Oracle's PL/SQL), PL/Tcl, PL/Perl and C subroutines by default, and you can add other languages as well.

For most stored procedures, PL/pgsql provides similar syntax and functionality, so this chapter will focus on converting stored procedures to PL/pgsql.

2) **CREATE PROCEDURE** becomes **CREATE FUNCTION**, PostgreSQL functions need parameters, if you simply want to execute some commands, use **VOID** as the parameter type:

CREATE FUNCTION foo (int param1, varchar param2) returns varchar AS $$

```
DECLARE
      foo int;
BEGIN
RETURN 'this is the result';
END
$$ language 'plpgsql';
```

The **$$**'s serve as quotes, surrounding the program script, with the bonus that you can use single quotes within the program text block.

The PostgreSQL documentation has a great entry on PL/pgsql (http://www.postgresql.org/docs/8.0/static/plpgsql.html). Read that for guidance.

A few tips for porting stored procedures:

- PostgreSQL functions do not support output parameters, avoid them.
- PostgreSQL functions do not support default values, all parameters must be passed to the function (although function overtyping is supported, allowing wrapper functions that allow defaults).
- use **FOUND** instead of **@@FETCH_STATUS**. Found is true if the last **SELECT**, **SELECT INTO**, or **FETCH** statement returned a value.
- instead of **@@IDENTITY** use **currval(***sequence name***)** by default the sequence name is <tablename>_<columnname>**_seq**.
- add semicolons as appropriate.
- remove **DEALLOCATE** cursor statements, the function is accomplished with **CLOSE** and **DEALLOCATE** has a different use.
- Use **PERFORM** instead of **EXEC** to call stored procedures/functions.
- For variable declarations, **SELECT foo = foocolumn** won't work. use either **SELECT INTO** or the PL/pgsql definition **:=** (as in **foo := foocolumn**.)
- Error handling is different, when a PL/pgsql program encounters an error the execution is immediately stopped (can't check **@@ERROR**) or execution is sent to the **EXCEPTION** block (if it exists) after the current **BEGIN** block. Thus, if you want to return specific error values, you will need an **EXCEPTION** block. Here's an example:

```
CREATE FUNCTION foo () returns int AS $$

DECLARE

      retval int;

BEGIN
```

```
        retval := -1;

        <SQL STATEMENTS 1>

        retval := -2;

        <SQL STATEMENTS 2>

        RETURN 0;

EXCEPTION

        when other then return retval;

END;

$$ language 'plpgsql';
```

This function will return -1 if an error occurred during SQL STATEMENTS 1 and -2 if an error occurred during SQL STATEMENTS 2. Note that the transaction is rolled back after the error so if the function returns -2, then the effects of SQL STATEMENTS 1 were NOT committed.

3) Also note that you need to add languages to a database before they can be used. The easiest way is to add a language first to the "template1" database, all new databases will then copy that database as a default. To add PL/pgsql to a database, use the shell command:

```
createlang plpgsql -U <user> <database>
```

In order to replace the functionality of **xp_sendmail**, to be able to send e-mail from a PostgreSQL database. You will need to:

1) Add TCL/u (unrestricted TCL, has more access to the computer, use **createlang pltclu** ...)

2) insert the pgMail script (available from http://sourceforge.net/projects/pgmail) after configuring and customizing it.

3) It might be nice to add a wrapper function called xp_sendmail in case .asp code ever calls it directly.

pgMail sends directly using SMTP.

Once the script is created, it can be imported into PostgreSQL with the GUI or command line. On a linux machine. use psql with redirection:

psql -h <server> **-U** <user> **<** <scriptfile>

or, using pgAdmin III:

- Select the server.
- Create a new database from the "Edit->New Object->New Database" menu.
- Click on the "Execute Arbitrary SQL Queries" button.
- Open the script file and click "Execute Query".
- Be sure to check for any errors. You can probably ignore any "NOTICE:" alerts, they represent implicitly defined object names (sequences, primary keys, etc.)

Part 3: Copying existing data into the new database:

Existing data can be copied from a MS SQL Server database to a PostgreSQL database.

1. Log into the MS SQL Server and Select "Import and Export Data" from the start menu. Alternatively, select the database in Enterprise Manager, and select "Export Data"
2. Under the "Choose Data Source" window that pops up, select a database connection to the local database, the default settings should be fine, at the bottom choose the database to connect to.
3. Under "Choose a Destination" select PostgreSQL under Destination, and choose (or create a new) appropriate DSN. The DSN contains connection settings for the PostgreSQL server such as host, user, password, and database.

   Click the Connection button and click the "True as -1" checkbox if you are importing bit values.
4. Select "Copy table(s) from the source database"
5. Click "Select All"
6. If there were any changed table names, input that here.

7. If there were any changed column names, click the transform button next to the changed table and choose the appropriate new column name in the destination menu.
8. Unfortunately, if you are using smallint to represent booleans, then true values in bits will be transferred as -1 instead of 1. A short VB Script transformation on each bit column would fix this, but would take a lot of time. Another, more practical solution is to create the database transfer the data (with -1 values) and then run update statements to correct each bit column.

    ```
    ex: UPDATE <tablename> SET <columnname> = -<columname>;
    ```

    This would need to be done on each bit column
9. Large text fields (larger than SQL Server's varchar limit of about 8000) may throw errors during transfer.
10. Note that unique identifier columns will not export to integer columns. If a sequence is in place on those columns (as outlined in part 2) then simply export all the other columns of the table, if there is a column that refers to other unique identifiers (a foreign key constraint), then this data cannot be transferred automatically. One possible solution is to create a temporary table (on SQL Server) with an identity column and a uniqueidentifier column, import all of your uniqueidentifier values into the table, and then use the identity column as the mapping to the new database. Use the "Transform" button in the window where you choose which tables to transfer and write a VB script to apply this mapping.

Hopefully the data migration was a success. One common error:

Violating constraints: (especially foreign key constraints). Because the data transfer does not order the transfer in a way that must satisfy any constraints, it's very possible that the wizard will import a table with a foreign key on another table that doesn't yet have any data.

You should build the tables, then import the data, and THEN apply constraints. Note that the database will not check constraints on the data being entered, so that if there is a problem with the database (or a problem during transfer) it will not be detected.

Part 4: ASP Code changes:

The goal was to have code that can be run identically on SQL Server or PostgreSQL. Changing only the DB connection string should be sufficient to change the database backend. There are a few differences between PostgreSQL and SQL Server that need to be accommodated in order to allow this to happen. Most of the important changes can be done in the database itself, but a few changes to the .asp code are necessary as well.

Note that the following suggestions tend to implement functions in PostgreSQL that simulate existing functions in SQL Server. This could severely hurt performance.

The emphasis is on creating cross-platform code so that there is no down time while databases are moved over to PostgreSQL.

After the transfer is complete, then the code can be optimized for the PostgreSQL database.

1. Opening RecordSets must specify "select * from <tablename>" instead of simply providing "<tablename>".

2. Executing stored procedures cannot use the "exec" command ("exec foo"). They must instead use a generic ADODB stored procedure syntax. There are several ways of doing this. The simplest is to use the ODBC escape sequence with call: ("{call foo}"), but, if the stored procedure/function takes parameters, the best way is to set the ADODB command's type to adCmdStoredProc:

```
Set cmd = Server.CreateObject("ADODB.Command")

cmd.ActiveConnection = db

cmd.CommandType = adCmdStoredProc

cmd.CommandText = "foo"

cmd.Execute
```

   Since most stored procedures will already be executed through ADODB Commands, the only change needed will be to change the CommandType parameter.

   Note that parameters to the procedure must be passed to the Command object if this method is used, not with the CommandText parameter. All of these methods are outlined in this Microsoft support article: http://support.microsoft.com/kb/q164485/

   Also, PostgreSQL functions do not have output parameters, that means that only one value can be returned from the function (using adParamReturnValue)

   This will need to be changed any time a stored procedure is called from a web page using SQL commands ("exec foo" or "foo").

3. MSSQLusesTOP while PostgreSQL uses **LIMIT** for restricting the size of returned RecordSets. A cross-database approach to fixing this discrepancy is to modify the RecordSet's MaxRecords attribute before calling open, unfortunately, the current PostgreSQL ODBC driver doesn't support this functionality. As an example, if this is implemented in the future, the code changes would be like this:

```
Set rs = Server.CreateObject("ADODB.Recordset")
rs.open "SELECT TOP 10 * from testtable"
```

Will become:

```
Set rs = Server.CreateObject("ADODB.Recordset")
rs.MaxRecords=10
rs.open "SELECT * from testtable"
```

In the meantime, creating a view to limit the results for each query, or using server side cursors and keeping track of the maximum number of rows in the ASP page seems to be the only solutions.

4. PostgreSQL does not support a column named "default" (mentioned earlier). If the database being migrated has a table with a column named default, this will need to be changed in the .asp code and in the SQL Server database to a different name.

5. PostgreSQL is not case preserving when it comes to table or column names. This is only an issue if a line of code compares (case sensitive) the name of a RecordSet field to a string that is not also returned from the RecordSet field name. Making the comparison case insensitive (by forcing both sides to lower case, for example) would solve the problem. Changing the MS SQL Server table and column names to all lower case (and any necessary .asp code changes) would also solve the problem.

6. Remember that if **boolean** types were used, then any reference to their values must be changed to include quotes (**'0'** instead of **0**) Hopefully this was not necessary as this is a common and difficult to find change.

7. All functions used must exist on both servers. For most existing T-SQL functions a compatibility layer can be created on the PostgreSQL server (See Part 5). However, functions that take arguments other than cstrings or integers (i.e. **CONVERT**, **DATEDIFF**, **DATEPART**, **DATEADD**) cannot be emulated on PostgreSQL.

   This means that any use of **CONVERT** in the ASP code must be changed to **CAST** (in the case of simple type conversion) XXEx: **CONVERT(varchar, intcolumn)** becomes **CAST(intcolumn as varchar)**

   Or, in the case of dates, casting to a character string and then using **substring** or some other hack to operate on it. Unless you need specific formats, casting a **datetime** as a **varchar** is probably good enough.

   In the case of **DATEDIFF**, you can utilize the fact that it is stored internally as two **int4**'s, so subtracting two dates and casting the resulting **datetime** as an **int** will return the date difference in days.

Ex: **DATEDIFF(dd, foo1, foo2) = 0** becomes **CAST(foo2 - foo1 as int) = 0.**

Casting as a float will return the difference more precisely, but still in units of days.

Ex: **DATEDIFF(hh,foo1,foo2) > 5** becomes (**CAST(foo2 - foo1 as float) * 24) > 5**.

For both of these, you will have to use **CREATE CAST** on the PostgreSQL server to allow casting from **timestamptz/timestamp/interval** to **int4**. See the attached **mssqlcomp.sql**

Although this may be more cumbersome, it will work on both MS SQL Server 7.0 and PostgreSQL 8.0 (with appropriate casts).

DATEADD can be done similarly.

Ex. **DATEADD(hh, 5, GETDATE())** becomes **(GETDATE() + 5)**

DATEPART is more difficult. Extracting the day, year, or month is as simple as calling **DAY**, **YEAR**, **MONTH** and writing an appropriate function in PL/pgsql, but extracting the day of the week cannot be done using any other SQL Server function.

One possible solution is to use modulo division base 7 on the date casted into int. As SQL Server uses the number of days since 1/1/1900 this results in Monday = 0 ... Sunday = 6.

All other changes can be implemented in the database, as outlined in part 5.

Connecting:

If you are using ADODB, it is a simple matter to open the PostgreSQL database instead of MS SQL, just change the database connection string

The database connection string is of this form: **"Driver={PostgreSQL}; Server=<Server>; Database=<db>; UID=<user name>; PWD=<password>; USEFETCHDECLARE=1; TRUEISMINUS1=1;BI=1;"**

where Server is the ip address or hostname of the PostgreSQL server, db is the PostgreSQL database name, user name and password are the PostgreSQL database user information.

USEFETCHDECLARE uses server-side cursors and results in a significant speedup on recordset insertions on large tables.

TRUEISMINUS1 allows "= True" statements to be mapped to a 1 in an integer column, this is necessary to use MS SQL's "bit" column types as booleans.

BI=1 maps bigints to ints that ADO can deal with. You need this if you ever access bigints (using the **COUNT** function returns a bigint!).

other useful options include "FETCH=100;" which determines the number of records fetched by the cursor a time. 100 is the default, it can be tweaked for performance.

A DSN can also be created on the webserver with the necessary options, then all that need to be specified are the DSN, UID, and PWD.

Part 5: Creating a compatibility layer in the PostgreSQL database in order to support common MS SQL functions

As mentioned above, the compatibility layer we set up is intended to quickly migrate to PostgreSQL. The performance of the new database will most likely suffer until the code can be optimized for PostgreSQL.

PostgreSQL and MS SQL both extend the SQL standard with useful functions. Although most common functions exist in both databases, they are called by different names.

Here are a few mappings from common MS SQL functions to appropriate PostgreSQL functions.

MS SQL	Postgresql	description
+	||	String concatenation operator
replicate(char, int)	repeat(char, int)	repeat given character given number of times. (not used in SIB).
space(int)	repeat(' ',int)	Specified number of spaces.
isnull()	coalesce()	makes a substitution for null values.
getdate()	now()	Returns the current time and date.

datediff() - Difference between two dates, just use minus symbol in PostgreSQL, use extract() function to get days, months, etc. (**extract (day from** <time1> - <time2>**)**);

convert() to_char() Converts a date (or other non-text data type) to a specified format as a string.

charindex() position() index of substring

In order to get compatibility with MS SQL functions, it will be necessary to write wrapper functions in PostgreSQL, probably using PL/pgsql.

PostgreSQL is extremely flexible (or this type of compatibility layer would not be possible!)

You can create your own functions, data type, casts, and operators.

CREATE FUNCTION will create these wrapper functions. Note that the type **anyelement** can be used to accept any form of input.

The "+" operator can be wrapped by using PostgreSQL's user-defined operators. (**CREATE OPERATOR** command)

These, along with necessary casts, are mostly implemented (with the exception of unsupportable functions mentioned above) in the attached SQL script (mssqlcomp.sql)

Part 6: Maintenance and Jobs

One of the biggest downsides to using PostgreSQL is it's lack of scheduled jobs. There is a project that adds this functionality for the linux version (pgjobs), but not for windows. Any scheduled operations on Windows must be created through the windows scheduler, and then use the command-line client to run the desired operation (**psql testdb < script.txt**).

1) Maintenance operations:

PostgreSQL needs several routine maintenance tasks in order to maintain its performance. "**vacuum**", "**analyze**" and "**reindex**".

vacuum cleans any rows that have been deleted or updated but not yet removed from the database. This is similar to MS SQL's "Shrink Database" feature. This should be done daily.

vacuum must be run at least once every one billion transactions or PostgreSQL could suffer Transaction ID wraparound. This is a very important point for high traffic sites.

analyze updates statistics on rows in order to optimize queries. This can be done at the same time as **vacuum** by running "**vacuum analyze;**"

reindex rebuilds indexes, this should be done weekly.

All of these maintenance tasks can be done from pgAdmin III, the GUI that comes with PostgreSQL. However, unlike MS SQL Server, there is no connection to the Windows scheduler, so there is no way to automate this process from the GUI. Windows scheduler could be set up to run **vacuumdb -az** (a vacuums all databases, z analyzes the databases) daily or more often depending upon use.

2) Backups

PostgreSQL supports live backups and Point in Time Recovery (PITR) by storing the WAL files.

However, automated backups have the same issue. They must be automated by running the backup program **pg_dump -f** <filename> <database> (or **pg_dumpall** for all databases) from a scheduled script.

Note that for any database with blob data (e.g. images) you must use **pg_dump -F t -f** < filename> in order to keep the binary data. This creates a tar archive.

pg_restore <filename> will restore the database to a backup.

Additionally, by archiving WAL (Write Ahead Log) files, the data between backups can be saved. This allows another database server to be set up using the last backup and the archived WAL files that represents the old database up until the point where the last WAL file was saved (each is typically 16MB).

To do this, edit the **archive_command** setting in postgresql.conf.

This command will be run on each filled WAL file, with the %p being substituted by the path to the file, and %f being substituted by the filename.

An example command that would copy the WAL file to c:\backups\ would be **archive_command = 'cp %p c:\backups\%f'**

Additionally, if the backups and WAL files are saved to another backup PostgreSQL server, the backup server can read the WAL files in to maintain a mirror of the original server.

If the live server goes down, switching the DNS entry to point to the backup would restore all but the most recent transactions.

Appendix: Files and scripts

mssqltopgsql.sh

This is a linux/unix script that will help significantly when converting a SQL Server script to PostgreSQL.

```
#!/bin/bash
#
# sed script to convert MS SQL Server's output into a form readable
# by postgresql.
#
# requires mssqltopgsqlawk.txt and mssqltopgsqlsed.txt
#
#
# note that this is just a tool to help the administrator, there are
many things
# that still need to be checked by hand or find/replace.
# However, use of this script should save a lot of time.
# Run this at your own risk.
#
# One thing to be aware of, if you set default multiple word string
values, this script will
# not properly deal with them, and it converts everything to lower case,
including
# the default value, be sure to check any default values for varchar
types.
# This script will mark these spots with "<-- FIX ME!"
#
# Useage:  ./mssqltopgsql.sh <T-SQL Script name> > <PG-SQL Script name>
#
# Ethan Townsend
# Project A
# 6/20/05

sed -f mssqltopgsqlsed.txt $1 | awk -f mssqltopgsqlawk.txt
```

mssqltopgsqlawk.txt

```
# awk script to convert MS SQL defaults to PostgreSQL defaults.
#
# takes:  "constraint <constraintname> default (<defaultvalue>) for
<columnname>,
# and converts it to:
# "alter <columnname> set default <defaultvalue>
#
```

```
# if the default value contains spaces, this won't work and instead it
places a "<-- FIX ME!" label
#
# also removes "set quoted identifer" and "exec " statements.
#
# Ethan Townsend
# Project A
# 6/24/05

/constraint [a-z_0-9]* default / {if ($7 != "") print($0,"<-- FIX ME!");
if ($7 != "") next}
/constraint [a-z_0-9]* default / {if ($4 == "(newid())") next}
/constraint [a-z_0-9]* default / {endstr = "";if (index($6,",") > 0)
endstr = ",";gsub("[^a-z0-9]","",$6);print("\talter",$6,"set
default",$4,endstr);next}
/constraint [a-z_0-9]* / {print("\tadd",$0);next}
/alter table [a-z0-9]*/ {print($1,$2,$3);next}
/^exec / {next}
/^set quoted_identifier / {next}
/./ {print($0)}
```

mssqltopgsqlsed.txt

```
# this is the sed script to help convert tables and views
# From Microsoft SQL Server to PostgreSQL
#
# Ethan Townsend 6/21/05
# Project A

# remove brackets
s/\[//g
s/\]//g

# remove filegroup commands
s/ ON PRIMARY/ /g
s/ TEXTIMAGE_ON PRIMARY/ /g
s/ CLUSTERED/ /g
s/ NONCLUSTERED/ /g

# remove unsupported commands.
s/ WITH NOCHECK/ /g
s/ WITH  FILLFACTOR = 90/ /g

# use ; instead of GO syntax.
s/GO/;/g

# make lower case (i know there must be a better way)
s/A/a/g
s/B/b/g
s/C/c/g
```

```
s/D/d/g
s/E/e/g
s/F/f/g
s/G/g/g
s/H/h/g
s/I/i/g
s/J/j/g
s/K/k/g
s/L/l/g
s/M/m/g
s/N/n/g
s/O/o/g
s/P/p/g
s/Q/q/g
s/R/r/g
s/S/s/g
s/T/t/g
s/U/u/g
s/V/v/g
s/W/w/g
s/X/x/g
s/Y/y/g
s/Z/z/g

# use serial data type for index column.
s/ uniqueidentifier / bigserial /g
s/ rowguidcol / /g
s/ int identity ([0-9, -]*) not null / serial /g
s/ numeric([0-9, ]*) identity ([0-9, ]*) not null / serial /g

# get rid of mistakes.

s/ serial null / int null /g
s/ bigserial null / bigint null /g

# we don't use dbo. prefix
s/dbo.//g

# this is redundant
s/ top 100 percent / /g

# change datatypes
s/ bit / smallint /g
s/ datetime / timestamptz /g
s/ smalldatetime / timestamptz /g
s/ nvarchar / varchar /g
s/ ntext / text /g
s/ image / bytea /g
s/ tinyint / smallint /g

# convert functions
s/ isnull/ coalesce/g
```

```
s/getdate()/now()/g

# in case of boolean default values
#s/ (0) / ('0') /g
#s/ (1) / ('1') /g
```

mssqlcomp.sql

This is a SQL script that emulates the most commonly used SQL Server functions.

```
-- This file emulates some common MS SQL Server functions in PostgreSQL

-- maps getdate() to now()
create or replace function getdate() returns timestamptz as $$
begin
return now();
end;
$$ language 'plpgsql';

-- maps isnull() to coalesce()
create or replace function isnull(anyelement, anyelement) returns
anyelement as $$
begin
return coalesce($1,$2);
end;
$$ language 'plpgsql' immutable;

-- This allows the use of "+" when joining strings.
create or replace function strcat(text, text) returns text as $$
begin
return $1 || $2;
end;
$$ language 'plpgsql' immutable;

create operator + (procedure = strcat, leftarg = text, rightarg = text);

-- these simulate day, month, and year functions in T-SQL
-- day function for each date/time type.

create or replace function day(timestamptz) returns int as $$
begin
return extract(day from $1);
end;
$$ language 'plpgsql' immutable;

create or replace function day(timestamp) returns int as $$
begin
return extract(day from $1);
end;
$$ language 'plpgsql' immutable;
```

```
-- month function for each date/time type.
create or replace function month(timestamptz) returns int as $$
begin
return extract(month from $1);
end;
$$ language 'plpgsql' immutable;

create or replace function month(timestamp) returns int as $$
begin
return extract(month from $1);
end;
$$ language 'plpgsql' immutable;

-- year function for each date/time type.
create or replace function year(timestamptz) returns int as $$
begin
return extract(year from $1);
end;
$$ language 'plpgsql' immutable;

create or replace function year(timestamp) returns int as $$
begin
return extract(year from $1);
end;
$$ language 'plpgsql' immutable;

-- emulate ms sql string functions.

create or replace function space(integer) returns text as $$
begin
return repeat(' ',$1);
end;
$$ language 'plpgsql' immutable;

create or replace function charindex(text, text) returns int as $$
begin
return position($1 in $2);
end;
$$ language 'plpgsql';

create or replace function len(text) returns int as $$
begin
return char_length($1);
end;
$$ language 'plpgsql';

create or replace function left(text, int) returns text as $$
begin
return substr($1, 0, $2);
end;
$$ language 'plpgsql';
```

```
-- a function to allow mailing.  It is currently a wrapper that allows
-- this functionality to be added later.
create or replace function xp_sendmail (tofield text, message text,
subject text) returns int as $$
declare
begin
return 0;
end;
$$ language 'plpgsql';

-- these functions provide casts from timestamps to ints (# of days).
Must be created as super-user.
create or replace function mscomp_int4(interval) returns int4 as $$
begin
return extract(day from $1);
end;
$$ language 'plpgsql' immutable;

create cast (interval as int4) with function mscomp_int4(interval);

create or replace function mscomp_int4(timestamptz) returns int4 as $$
begin
return mscomp_int4($1 - '1/1/1900');
end;
$$ language 'plpgsql' immutable;

create cast (timestamptz as int4) with function mscomp_int4
(timestamptz);

create or replace function mscomp_int4(timestamp) returns int4 as $$
begin
return mscomp_int4($1 - '1/1/1900');
end;
$$ language 'plpgsql' immutable;

create cast (timestamp as int4) with function mscomp_int4(timestamp);

create or replace function mscomp_float(interval) returns float as $$
begin
return (extract(epoch from $1) / 86400);
end;
$$ language 'plpgsql' immutable;

create cast (interval as float) with function mscomp_float(interval);

create or replace function mscomp_float(timestamptz) returns float as $$
begin
return mscomp_float($1 - '1/1/1900');
end;
$$ language 'plpgsql' immutable;
```

```
create cast (timestamptz as float) with function mscomp_float
(timestamptz);

create or replace function mscomp_float(timestamp) returns float as $$
begin
return mscomp_float($1 - '1/1/1900');
end;
$$ language 'plpgsql' immutable;

create cast (timestamp as float) with function mscomp_float(timestamp);
```