SYBASE
An SAP Company

Migrating a PostgreSQL Database to SQL Anywhere 12

A WHITEPAPER FROM SYBASE, AN SAP COMPANY

Contents:

Introduction

Migrating data from PostgreSQL to SQL Anywhere can be a straightforward process if there are not a lot of PostgreSQL extensions in use within your database and application. SQL Anywhere simplifies migration by including built-in tools that facilitate a smooth transition from PostgreSQL (and other RDBMS's) to SQL Anywhere.

The first part of this document discusses in detail differences between SQL Anywhere and PostgreSQL, including data type differences, feature differences, and syntax differences. Some of the features that are unique to PostgreSQL can hinder migration. Approaches to how you might choose to deal with these issues are provided. The second part of this document includes a systematic explanation of how to migrate data from a PostgreSQL database into a SQL Anywhere database using the Sybase Central Data Migration wizard. Finally, the third part of this document supplies an example of how you might migrate an existing application running against PostgreSQL to one that runs against SQL Anywhere.

This document was written for SQL Anywhere version 12 and later, and PostgreSQL version 9.1 and later.

Differences between PostgreSQL 9.1 and SQL Anywhere 12

The following sections describe some of the differences between PostgreSQL and SQL Anywhere that you may encounter during migration, along with some suggested solutions that can be used as starting points to resolve any issues that arise during migration. There are many ways to optimize your code with SQL Anywhere features that are missing from PostgreSQL.

It is highly recommended that you review the SQL Anywhere documentation as well as the developer resources, including samples and technical documents, available on the SQL Anywhere Tech Corner website at http://www.sybase.com/developer/library/sql-anywhere-techcorner when moving to SQL Anywhere.

Data types

In most cases, the PostgreSQL data types can map directly to SQL Anywhere data types. The following table lists some examples:

PostgreSQL data type	Equivalent SQL Anywhere data type	Notes
BIGINT	BIGINT	
BIGSERIAL	BIGINT	With a default system-defined autoincrement value
BIT	BIT	
BIT VARYING(n)	VARBIT(n)	In SQL Anywhere, length is 1 by default
BIT VARYING	LONG VARBIT	
BOOLEAN	TINYINT OR BIT	
BOX	ST_POLYGON(ST_POINT, ST_POINT)	The two ST_POINTS represent the lower-left and upper right corners
BYTEA	LONG BINARY	
CHARACTER VARYING(n)	VARCHAR(n CHAR)	Length in characters must be defined
CHARACTER(n)	CHAR(n CHAR)	Length in characters must be defined
CIDR	N/A	No equivalence
CIRCLE	ST_CIRCULARSTRING(ST_POINT, ...)	
DATE	DATE	
DOUBLE PRECISION	DOUBLE	
INET	N/A	No equivalence

INTEGER	INTEGER	
INTERVAL	INTEGER	Can be converted to date format
LINE	N/A	No equivalence
LSEG	ST_LINESTRING(ST_POINT, ...)	
MACADDR	N/A	No equivalence
MONEY	MONEY	
NUMERIC	NUMERIC	
PATH(open)	ST_LINESTRING(ST_POINT, ...)	
PATH(closed)	ST_POLYGON(ST_POINT, ...)	
POINT	ST_POINT	
POLYGON	ST_POLYGON(ST_POINT, ...)	
REAL	FLOAT	
SMALLINT	SMALLINT	
SERIAL	INTEGER	With a default system-defined autoincrement value
TEXT	TEXT	
TIME	TIME	
TIME WITH TIME ZONE	N/A	Timestamp with timezone
TIMESTAMP	TIMESTAMP	
TIMESTAMP WITH TIME ZONE	TIMESTAMP WITH TIME ZONE	
TSQUERY	N/A	No equivalence
TSVECTOR	N/A	No equivalence
TXID_SNAPSHOT	N/A	No equivalence
UUID	UNIQUEIDENTIFIER	
XML	XML	

Note: In addition to the differences in data types themselves, there is also a difference in the declaration of data types. PostgreSQL provides an optional parameter for its numeric types that allow you to specify the maximum display width for integer types. For example, an INT(4) column would return the value '1' as '<s><s><s>1', where <s> is a space. The optional ZEROFILL modifier on the type definition would replace the

spaces in the previous example with zeros and add the UNSIGNED attribute to the column. For example, '1' is returned as '0001'. The merge of display format and data values in the type definition is not supported by SQL Anywhere. The CAST and CONVERT functions, along with the various string manipulation functions are available to format data values when they are retrieved from the database.

The following data types differ from SQL Anywhere more substantially than by syntax:

MEDIUMINT: These are 3-byte integer values. They can easily be simulated using an INTEGER (4 bytes) or SMALLINT (2 bytes) in SQL Anywhere, depending on the expected range of values for the column.

YEAR: Year is a 2 or 4 digit value. The SQL Anywhere DATE data type can be used to hold year values, but uses slightly more storage space. Date arithmetic and conversion can be performed using the SQL Anywhere built-in functions listed under "Date and Time Functions" in the "SQL Functions" chapter of the "SQL Anywhere Server - SQL Reference" manual.

The following data types do not match exactly, and will require some work to migrate to SQL Anywhere:

NCHAR/NVARCHAR: As of PostgreSQL 5, an NCHAR value is stored in PostgreSQL using the UTF8 character set. SQL Anywhere supports a variety of character sets, including UTF8. With a database created using the proper collation, the use of a special data type to store international values is not required, though SQL Anywhere does support the NCHAR data type. To learn more about the latest international character set support in SQL Anywhere, see the chapter "International Language and Character Sets" in the "SQL Anywhere Server - Database Administration" manual.

ENUM: An ENUM value is a string object whose value must be chosen from a list of supplied values enumerated in the column definition when a table is created. The enumerated values can also be inserted or retrieved by their index position in the ENUM definition. The index value 0 is reserved for the empty string. The ENUM data type is represented in SQL Anywhere by a TINYINT column. There are a few options to accomplish the same behavior as the PostgreSQL ENUM, but changes to the client application will almost certainly be required. Some options are:

- Altering the client side application to remove the need for ENUM values
- Translating the ENUM values on the client side
- Adding some logic to the server side to attempt to mimic the PostgreSQL behavior of ENUM values by using stored procedures, triggers, computed columns, view, and/or a mapping table for the ENUM types

For example, a view could be created on the table containing the ENUM fields to allow for the return of the values as a string, while a regular SELECT could be used to return them as a number. Here is an example of a view that could be used:

```
CREATE TABLE enumtbl (pkey INTEGER NOT NULL PRIMARY KEY, enumval TINYINT);

CREATE VIEW v_enumtable AS
  SELECT pkey
    CASE WHEN 0 then ''
        WHEN 1 then 'val1'
        WHEN 2 then 'val2'
        WHEN 3 then 'val3'
    ELSE NULL
  END
FROM enumtbl;
```

Then, a query may look something like this:

```
SELECT pkey, enumval FROM v_enumtable;
```

Alternatively, a mapping table could be created for the ENUM vales and whenever you retrieve data from `enumtbl`, a join can be made to the mapping table containing the ENUM strings.

```
CREATE TABLE enummap( enumval TINYINT NOT NULL PRIMARY KEY, enumstr CHAR(16));
```

Then a query may look something like this:

```
SELECT pkey, enumval FROM enumtbl, enummap
WHERE enumtbl.enumval = enummap.enumval;
```

An insert on the table can be done directly if you are using the index values of the ENUM; otherwise, a stored procedure could be used to insert a row into any table containing an ENUM. The stored procedure would contain the logic to decode the ENUM values. Following is a sample stored procedure implementation to deal with an ENUM column equivalent in SQL Anywhere (using the same table definition as above):

```
CREATE PROCEDURE sp_insert_enumval (IN pkeyval int, IN enum CHAR(16))
BEGIN
 DECLARE enum_map TINYINT;
 IF enum IS NOT NULL THEN
   CASE enum
     WHEN '' THEN SET enum_map = 0
     WHEN 'val1' THEN SET enum_map = 1
     WHEN 'val2' THEN SET enum_map = 2
     WHEN 'val3' THEN SET enum_map = 3
   ELSE SET enum_map = 0
   END CASE
 END IF;


 INSERT INTO enumtbl VALUES( pkeyval, enum_map);
END
```

SET: A SET value is a string object whose value must be chosen from a list of values supplied when the column is defined. It is different from the ENUM type in that 0 or more values from the list can be combined to create a valid value for the column. Each value in the set is assigned a binary value and data can be assigned or retrieved by using a number representing the combination of values to be set. For example, specifying a value of 9 would insert the first and fourth value from the set into the column. Depending on how many values are in the set (64 is maximum), anything from a TINYINT to a BIGINT is required to map a SET value from PostgreSQL to SQL Anywhere. To achieve the same behavior as PostgreSQL, methods similar to those demonstrated above with the ENUM data type can be used.

PostgreSQL function mappings to SQL Anywhere

Many of the functions in both PostgreSQL and SQL Anywhere have the same name. Most PostgreSQL functions that have different names have an equivalent SQL Anywhere version. PostgreSQL contains a few built-in functions that do not exist in SQL Anywhere. Most of these functions can be created in SQL Anywhere as user-defined functions that perform the same activity. If you give these functions the same name in the SQL Anywhere database, you will not need to modify the existing client application's SQL statements. Here are some examples of how SQL Anywhere user-defined functions can supply the same functionality as their PostgreSQL built-in counterparts:

```
CREATE FUNCTION FROM_UNIXTIME (IN fromdt bigint default 0, IN fmt varchar(32) default 'Mmm dd,
yyyy hh:mm:ss' ) RETURNS datetime
BEGIN
 RETURN(dateformat(dateadd(second, fromdt, '1970/01/01 00:00:00', fmt))
END;

CREATE FUNCTION SEC_TO_TIME (IN sec bigint default 0) RETURNS time
BEGIN
 RETURN (dateadd(second, sec, '1970/01/01 00:00:00') )
END;
```

The following sections detail many of the PostgreSQL functions along with their SQL Anywhere equivalents. The list is extensive, but not exhaustive, as the list of function in both SQL Anywhere and PostgreSQL changes with each release.

Aggregate Functions

Almost all PostgreSQL aggregate functions are identical to SQL Anywhere aggregate functions, with the exception of the following which have no equivalence:

- ARRAY_AGG
- BOOL_AND
- BOOL_OR
- EVERY
- STRING_AGG

String Functions

PostgreSQL function	**Equivalent SQL Anywhere function**	**Notes**
ASCII(string)	ASCII(string)	
BTRIM(string text [, characters text])	N/A	No equivalence
CHR(int)	CHAR(integer)	
CONCAT(str “any” [, str “any [, ...]]))	STRING (a, b, ...)	
CONCAT_WS(sep text, str “any” [, str “any” [,...]])	STRING(str1, sep, str2, sep ...)	
CONVERT(String bytea, src_encoding name, dest_encoding name)	CSCONVERT(string,dest,src)	
CONVERT_FROM(String bytea, src_encoding name)	CSCONVERT(String, ‘db_charset’,src)	
CONVERT_TO(String text, dest_encoding name)	CSCONVERT(String, dest)	
DECODE(String text, ‘base64’)	BASE64_DECODE	
ENCODE(Data bytea, ‘base64’)	BASE64_ENCODE	
FORMAT(Formatstr text [, str “any” [, ...]])	N/A	No equivalence
INITCAP(String)	N/A	No equivalence
LEFT(Str text, n int)	LEFT(String, Integer)	For negative integers, use RIGHT

LENGTH(String)	LENGTH(String)	
LPAD(String text, Length int[, Fill text])	N/A	
LTRIM(String text [, Characters text])		
MD5(String)	HASH	
PG_CLIENT_ENCODING()	N/A	No equivalence
REGEXP_MATCHES(String text, Pattern text [, Flags text])	REGEXP_SUBSTR	
REGEXP_REPLACE(String text, Pattern text, Replacement text [, Flags text])	REGEXP_SUBSTR REPLACE	Use the two in conjunction to represent REGEXP_REPLACE
REGEXP_MSPLIT_TO_ARRAY(String text, Pattern text [, Flags text])	N/A	No equivalence
REGEXP_SPLIT_TO_TABLE(String text, Pattern text [, Flags text])	N/A	No equivalence
REPEAT(String text, Numer int)	REPEAT(String, Integer)	
REPLACE(String text, Number int)	REPLACE(String, Integer)	
REVERSE(String)	REVERSE(String)	
RIGHT(String text, n int)	RIGHT(String, Integer)	For negative integers, use LEFT
RPAD(String text, Length int[, Fill text])	N/A	No equivalence
RTRIM(String text [,Characters text])	RTRIM(String)	No support for character trim
SPLIT_PART(String text, Delimiter text, Field int)	N/A	SELECT row_value FROM sa_split_list(string, delimiter) WHERE line_num = field
STRPOS(String, Substring)	LOCATE(String, Substring)	
TO_ASCII(String text [, Encoding text])	TO_CHAR(String [, Charset])	
TO_HEX(Number Int|Bigint)	INTTOHEX(Integer)	
TRANSLATE	N/A	No equivalence

NOTE: SQL Anywhere does not support the Quote functions in PostgreSQL

Numeric Functions

The following numeric functions are identical in PostgreSQL and SQL Anywhere:

- ABS
- CEILING
- DEGREES
- EXP
- FLOOR
- MOD
- PI
- POWER
- RADIANS
- SIGN
- SQRT

The following numeric functions have direct equivalences between PostgreSQL and SQL Anywhere:

PostgreSQL function	**Equivalent SQL Anywhere function**	**Notes**
CBRT(dp)	POWER(Numeric, 1/3)	
DIV(Y numeric, X numeric)	ROUND(Numeric, Integer)	
LN(dp or numeric)	LOG(Numeric)	
LOG(numeric)	LOG10(Numeric)	
LOG(X numeric, B numeric)	LOG(X Numeric)/LOG(B numeric)	
RANDOM()	RAND	
ROUND(dp or numeric)	ROUND(Numeric, 0)	
ROUND(V numeric, S int)	ROUND(Numeric, Integer)	
SETSEED(dp)	RAND(Integer)	
TRUNC(dp or numeric)	TRUNCNUM(Numeric, 0)	
TRUNC(V numeric, S int)	TRUNCNUM(Numeric, Integer)	

NOTE: WIDTH_BUCKET in PostgreSQL does not have a SQL anywhere equivalent

Date and Time Functions

The date and time functions vary the most between the two database servers, with no identical functions. The following lists date and time functions in PostgreSQL with equivalences in SQL Anywhere:

PostgreSQL function	**Equivalent SQL Anywhere function**	**Notes**

AGE(Timestamp, Timestamp)	DATEDIFF(Datepart, Getdate(), Timestamp)	
AGE(Timestamp)	DATEDIFF(Datepart, Getdate(), Timestamp)	
CLOCK_TIMESTAMP()	GETDATE()	
CURRENT_DATE	CURRENT DATE	
CURRENT_TIME	CURRENT TIME	
CURRENT_TIMESTAMP	CURRENT TIMESTAMP	
DATE_PART('Datepart', TIMESTAMP Timestamp)	DATEPART(Datepart, Timestamp)	
EXTRACT(Field fromtimestamp)	DATEPART(Datepart, Timestamp)	
LOCALTIME	CURRENT TIME	
LOCALTIMESTAMP	CURRENT TIMESTAMP	
NOW()	CAST(GETDATE() AS TIMESTAMP WITH TIMEZONE)	
TIMEOFDAY()	NOW()	

The following PostgreSQL date time functions have no equivalence in SQL Anywhere:

- DATE_PART(Text, Interval)
- DATE_TRINC(Text, Timestamp)
- EXTRACT(Field frominterval)
- ISFINITE(Date)
- ISFINITE(Timestamp)
- ISFINITE(Interval)
- JUSTIFY_DAYS(Interval)
- JUSTIFY_HOURS(Interval)
- JUSTIFY_INTERVAL(Interval)
- STATEMENT_TIMESTAMP()
- TRANSACTION_TIMESTAMP()

Syntax Mappings

Most of the syntax features of PostgreSQL are available in SQL Anywhere, but occasionally the syntax for accessing those features is different. The following charts detail many of these statements along with their SQL Anywhere equivalents. For specific examples of SQL Anywhere syntax listed below, see the "SQL Statements" chapter of the "SQL Anywhere Server - SQL Reference" manual.

Operators

PostgreSQL operator	**Equivalent SQL Anywhere**	**Notes**

	operator	
!=	<>	
<=>	(expr1 = expr2 OR ((expr1 IS NULL) AND (expr2 IS NULL)))	The <=> operator represents equality including NULL values (NULL=NULL is true).
ISNULL(expr)	IS NULL expr	
INTERVAL(N, N1, N2,...)	None built in	A user defined function could easily be used to achieve the same functionality. For example: if (N < N1) then 0 elseif (N < N2) then 1 elseif...
!	NOT	
&&	AND	
||	OR	
A XOR B	((a AND (NOT b)) OR ((NOT a) AND b))	The SQL Anywhere expression is complex for large numbers of XOR arguments, so an alternative migration technique is recommended dependent on the application scenario.

Data Manipulation Language

PostgreSQL statement	Equivalent SQL Anywhere statement	Notes
INSERT ...	INSERT ...	
ON DUPLICATE KEY UPDATE	ON EXISTING UPDATE	SQL Anywhere also offers the options ERROR and SKIP for existing rows.
SELECT .... INTO OUTFILE	UNLOAD SELECT DBISQL OUTPUT TO	
SELECT/UPDATE/DELETE ... LIMIT	FIRST or TOP n	
DEFAULT '0' NOT NULL auto_increment	NOT NULL DEFAULT AUTOINCREMENT	
LIMIT offset, numRows	TOP numRows START AT offset	

Insert IGNORE	INSERT ... ON EXISTING SKIP	
Replace ...	INSERT ... ON EXISTING UPDATE	
GROUP_CONCAT	LIST	
INSERT INTO ... DEFAULT VALUES	INSERT INTO ... VALUES (DEFAULT)	
LOAD DATA INFILE	LOAD TABLE	

Miscellaneous Syntax

The following is a miscellaneous list of compatibility items that do not fit into the aforementioned categories. It also includes mappings between functions that are not exactly the same, but are designed to provide the same functionality.

PostgreSQL syntax	**Equivalent SQL Anywhere syntax**	**Notes**
VERSION()	@@version global variable	
PostgreSQL_insert_id()	@@identity global variable	
LAST_INSERT_ID variable	@@identity global variable	
PostgreSQL_affected_rows()	@@rowcount global variable	
ANALYZE TABLE	sa_table_page_usage, sa_table_fragmentation	SQL Anywhere also offers access to other properties via the property() function.
OPTIMIZE TABLE	CREATE STATISTICS	SQL Anywhere has a self-tuning optimizer that automatically maintains statistics, so statistics do not need to be updated manually.
CHECK TABLE	sa_validate () procedure	
USE database-name		There is no equivalent in SQL Anywhere. Each database running on a server requires its own connection.
LOCK TABLES (name) WRITE	LOCK TABLES table-name IN EXCLUSIVE MODE	SQL Anywhere supports row-level locking, so table locks are generally not required.

UNLOCK TABLES	COMMIT	A COMMIT releases all locks, unless a cursor is opened using the WITH HOLD clause.
DO	CALL	
FLUSH/RESET	sa_flush_cache, sa_flush_statistics	Most of the other flushable elements in PostgreSQL are automatically managed by SQL Anywhere and do not need to be flushed.
REGEXP/RLIKE	SIMILAR	SIMILAR works differently from the PostgreSQL REGEX syntax, but performs the same function. It may suit the needs where the PostgreSQL REGEXP expression is being used.
BINARY str	CAST str AS BINARY	
CURDATE() | CURRENT_DATE()	CURRENT DATE	
CURTIME() | CURRENT_TIME()	CURRENT TIME	
SYSDATE() | LOCALTIME() | CURRENT_TIMESTAMP() | NOW()	NOW(), CURRENT TIMESTAMP	
UTC_DATE()	CURRENT UTC TIMESTAMP	
DATABASE()	CURRENT DATABASE	
LOAD_FILE(file)	xp_read_file(file)	In SQL Anywhere, the contents of a file are returned as a long binary field, while in PostgreSQL they are returned as a string.
CONNECTION_ID()	CONNECTION_PROPERTY ('Number')	

Other migration issues

The following is a list of miscellaneous notes to keep in mind when migrating from PostgreSQL to SQL anywhere:

- The identifiers in PostgreSQL are optionally enclosed with the back quote (‘), while SQL Anywhere uses the double quote (“) or , alternatively, square brackets ([]).
- Some words that are keywords in SQL Anywhere are not in PostgreSQL, such as ‘comment’ and ‘session’. These keywords must be enclosed in double quotes in order to be used with SQL Anywhere. Alternatively, you can use the SQL Anywhere NON_KEYWORDS option to change the list of recognized keywords. For information about the NON_KEYWORDS option, see

"NON_KEYWORDS option [compatibility]" in the "Database Options" chapter of the "SQL Anywhere Server - Database Administration" manual.

- The minimum timestamp value in SQL Anywhere is '0001-01-01 00:00:00', while it is '0000-0000-00 00:00:00' in PostgreSQL.
- Timestamps in PostgreSQL have the format of YYYY-MM-DD hh:mm:ss. SQL Anywhere includes fractions of a second as part of the timestamp value. The TIME_FORMAT option allows you to specify the exact format used to return datetime value. For information about the TIME_FORMAT option, see "TIME_FORMAT option [compatibility]" in the "Database Options" chapter of the "SQL Anywhere Server - Database Administration" manual.
- While PostgreSQL allows the use of single or double quotes around string literals, by default single quotes must be used to enclose string values in SQL Anywhere and double quotes signify the use of a database object identifier. This behavior can be changed by setting the QUOTED_IDENTIFIER option in the database. For information about the QUOTED_INDENTIFIER option, see "QUOTED_IDENTIFIER option [compatibility]" in the "Database Options" chapter of the "SQL Anywhere Server - Database Administration" manual.

Migrating a PostgreSQL database to a SQL Anywhere database

Migrating data from PostgreSQL to SQL Anywhere is a straightforward process, with minor issues occurring only if you are using the PostgreSQL-specific data types mentioned previously. Data migration can be accomplished using the Data Migration wizard that is part of Sybase Central. Alternatively, a more customized migration can be done using the sa_migrate set of stored procedures in SQL Anywhere. The PostgreSQLdump utility, coupled with the SQL Anywhere LOAD TABLE statement, could also be used to migrate the data. Note that if the PostgreSQL SET or ENUM data types are used in the PostgreSQL database, you may have some additional considerations when migrating your PostgreSQL database to SQL Anywhere. For information about these data types and the differences from SQL Anywhere, see "Data types" on page 3.

Requirements

- This document assumes you have a PostgreSQL database running on any of its supported platforms and SQL Anywhere 12 installed on any of the supported Windows platforms.
- If you have not created a PostgreSQL database, you can create a few tables in the PostgreSQL test database to walk through the migration steps.
- The PostgreSQL ODBC 5.1 (or later) driver must also be installed on the computer running the SQL Anywhere database.

Creating a SQL Anywhere database

You must first create a SQL Anywhere database to migrate the PostgreSQL database to. The following steps explain how to create a new database using Sybase Central.

1. Start Sybase Central. From the Start Menu, choose Program → SQL Anywhere 12 → Administration Tools → Sybase Central.
2. Create a new SQL Anywhere 12 database. Choose Tools → SQL Anywhere 12 → Create Database. Follow the instruction in the wizard to create a new database.

Creating a data source for the PostgreSQL database

The migration process requires an ODBC connection to the source database. Therefore, you need to create an ODBC data source (DSN) for the PostgreSQL database.

1. Download and install the PostgreSQL ODBC 5.1 driver if you have not already done so. The most recent driver is located at http://dev.PostgreSQL.com/downloads/connector/odbc/.
2. From Sybase Central, choose Tools → SQL Anywhere 12 → Open ODBC Administrator. The ODBC Data Source Administration dialog appears.
3. Click Add to add a new DSN. The Create New Data Source wizard appears.
4. Select the PostgreSQL ODBC 5.1 Driver from the list of available drivers and then click Finish. The PostgreSQL ODBC 5.1 Driver - DSN Configuration dialog appears.
5. Type a name for the data source in the Data Source Name field. For example, name the data source 'PostgreSQL migrate'.
6. Supply the server name, user ID, password, and database name on the logic page of the PostgreSQL ODBC Connector dialog.
7. Click the Test Data Source button to ensure you have configured the data source correctly.

8. Click OK.

Migrating the PostgreSQL database to SQL Anywhere

The steps to migration are outlined below.

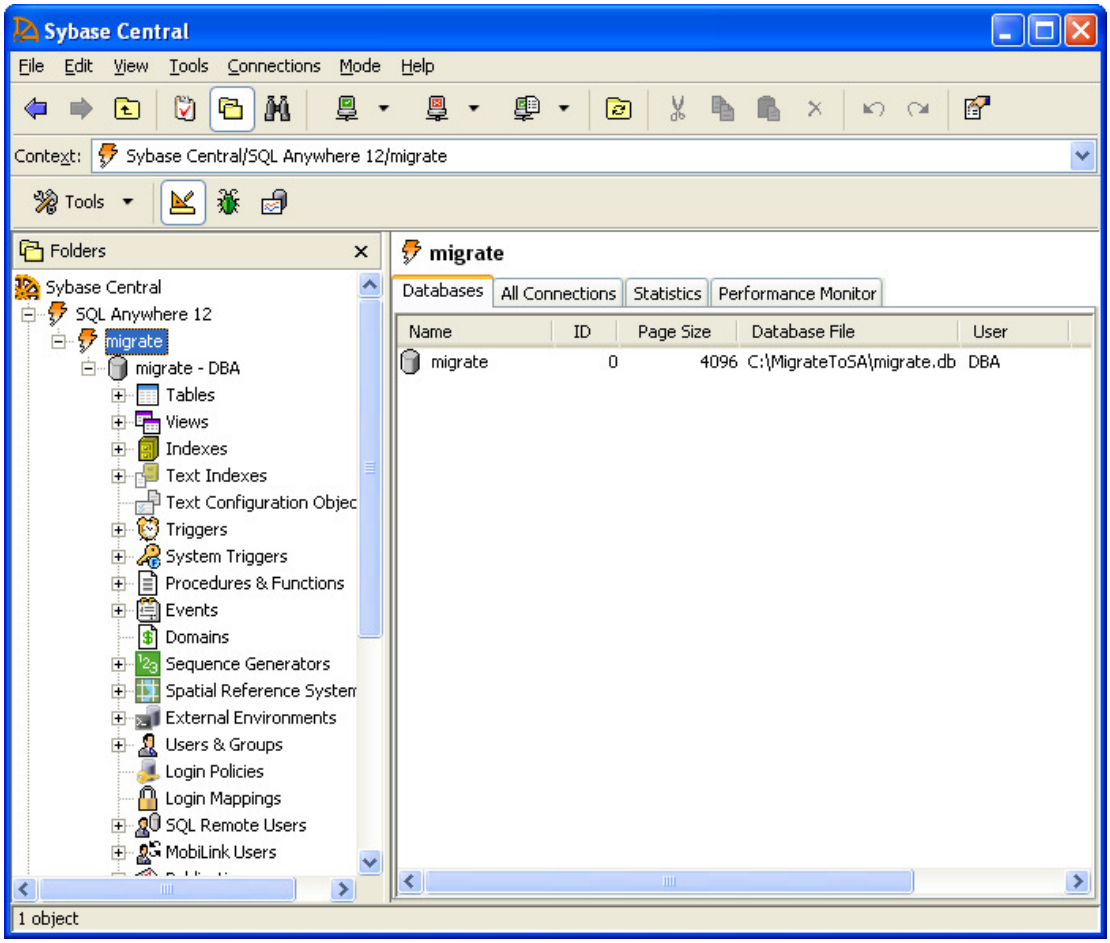
Connecting to the SQL Anywhere Database

In order to migrate the new SQL Anywhere database, you must first connect to the SQL Anywhere database.

1. If you are not already connected, from Sybase Central, choose Connections → Connect with SQL Anywhere 12. The Connect dialog appears.
2. Type a valid User ID and password for your database. By default, there is a user 'DBA' with password 'SQL'.
3. If the database is not yet running, from the Actions dropdown, select 'Start and connect to a database on this computer'. If it is already running, simply select 'Connect to a database on this computer'. Click the Browse button and then select the SQL Anywhere database file you created.
4. Click Connect. The SQL Anywhere database server starts automatically.

Creating a Remote Server and External Login

The next step is to tell Sybase Central where to find the PostgreSQL database. This is done by creating a remote server.



1. In the left pane of Sybase Central, expand your database server and database icons. In the example below, the database named migrate is running on a database server that is also named migrate.
2. In Sybase Central, from the Tools menu, select SQL Anywhere 12 → Migrate Database. The Migrate Database Wizard will appear.
3. Select the SQL Anywhere 12 database you just created and click Next.
4. Click 'Create Remote Server Now...' and follow the instructions in the wizard to create a remote server that connects to your PostgreSQL database.
    a. On the first page of the wizard, type a name for the remote server, for example 'PostgreSQL migrate' and then click Next.
    b. Choose PostgreSQL as the type of remote server. Click Next.
    c. Select the Open Database Connectivity (ODBC) Option and type the name of the ODBC data source for your PostgreSQL database in the connection information field. For example, if you named your ODBC data source 'PostgreSQL migrate' when you created it, type 'PostgreSQL migrate' in the connection information field.
5. Click Next.
6. Do not choose to make the server read only. Click Next.
7. If the remote server does not define a user that is the same as the user ID you are connected to the SQL Anywhere database with, you must create an external login for your current user. For

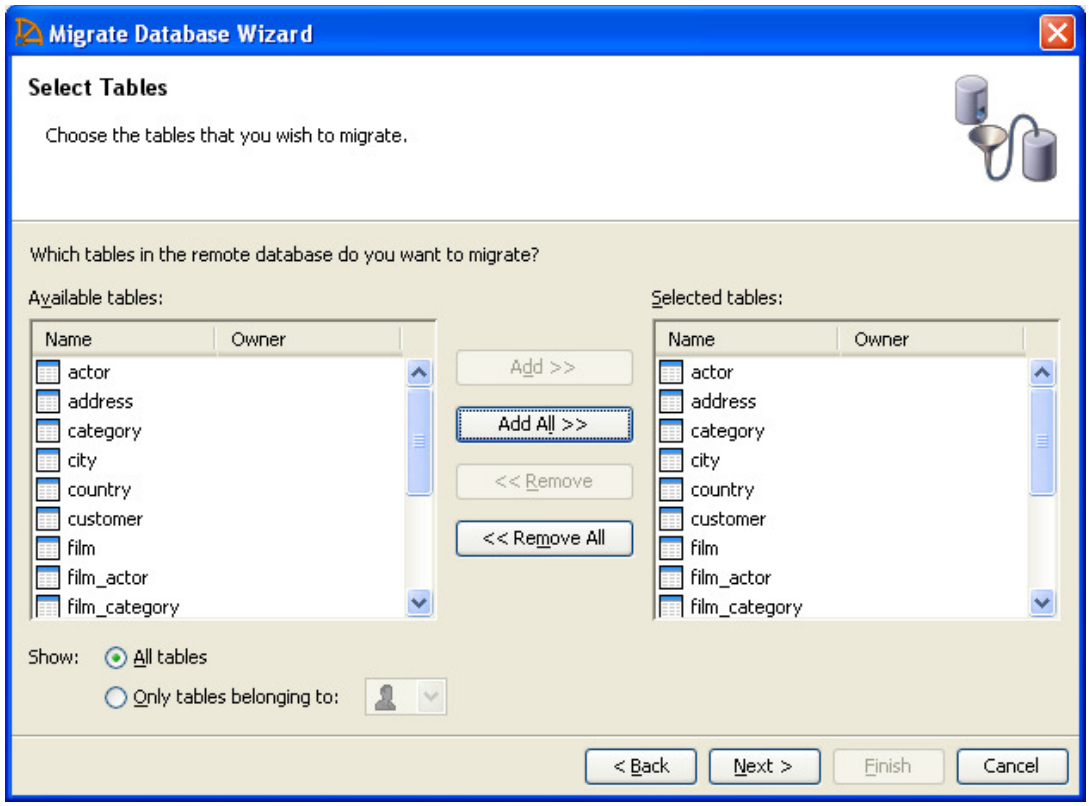
example, if you connected to the SQL Anywhere database with the user ID DBA, and your PostgreSQL database does not contain a user ID DBA, then you must create an external login. Type a user name from the PostgreSQL database in the Login Name field. Type the password for this user in the Password and Confirm Password fields.

8. Use the "Test Connection" button to ensure you can connect. Then, click Finish.

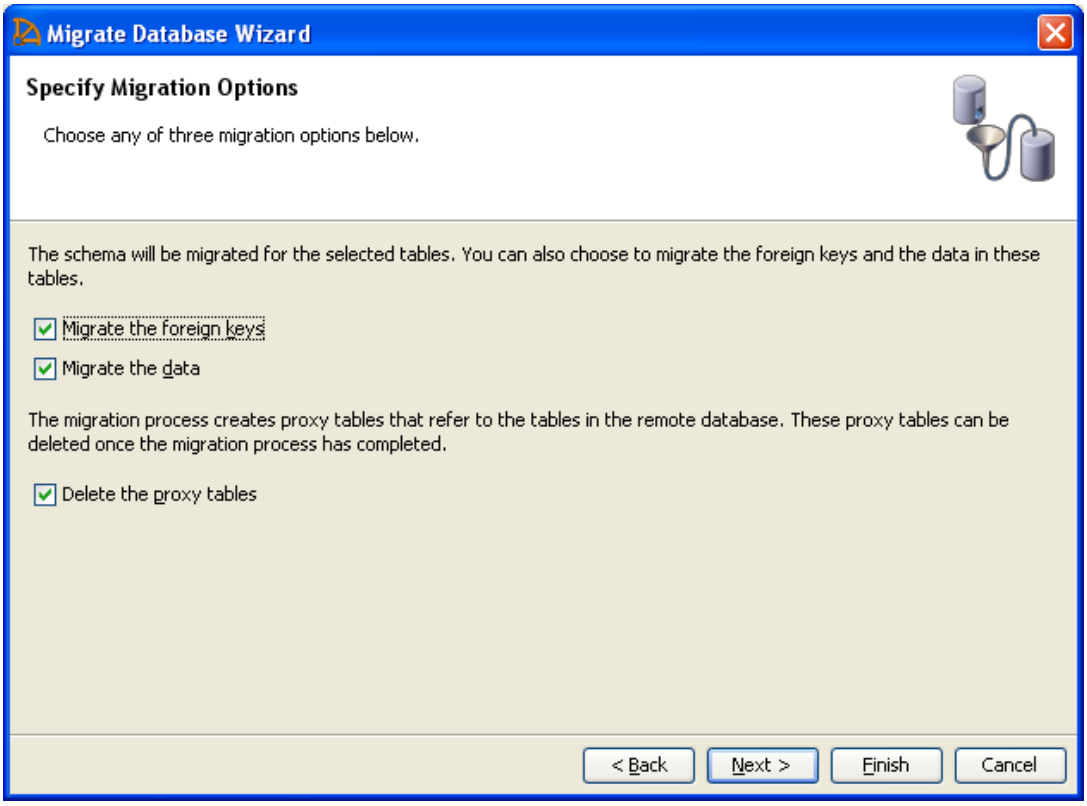
Migrating the PostgreSQL database

Now you are ready to migrate your PostgreSQL database: SQL Anywhere is running, connected and able to communicate to the PostgreSQL database via ODBC. Back in the Migration wizard, you now have a remote server from which to perform the migration.

1. From Sybase Central, choose Tools → SQL Anywhere 12 → Migrate Database. The Database Migration Wizard appears.
2. Select the current database and then click Next.
3. Select the PostgreSQL remote server you created, for example, PostgreSQL migrate, and then click Next.
4. The tables in the PostgreSQL database appear in the left list box. Select the ones you would like to add by clicking on the table name on the left and clicking Add or simply clicking Add All to migrate all tables.



5. Select the SQL Anywhere database user you wish to own the tables or create a new user. Click Next.
6. Select the options you wish to use.



7. Click Finish to start the migration. The Migration Database window appears. You can close this window when the status changes to 'Completed'.

Tweaking the new SQL Anywhere database

Now that you have migrated the PostgreSQL schema and data to the SQL Anywhere database, you can start enjoying the benefit SQL Anywhere brings. One immediate benefit is transactional support.

Since not all PostgreSQL tables support referential integrity, your PostgreSQL schema may not have foreign keys. If in step 6 of 'Migrating the PostgreSQL Database' you chose to ignore foreign key relationships, you can add referential integrity support later. To add referential integrity support:

1. List the foreign keys in my PostgreSQL database by issuing the following SQL statement against the PostgreSQL database:
   ```
   SHOW TABLE STATUS FROM database_name
   ```
   Alternatively, `SHOW CREATE TABLE table_name` will also reveal any foreign key relationships. The referential constraints are listed under the comment column for each table in the form.
   ```
   (column_name) REFER ref_db_name/ref_table_name(ref_column_name)
   ```
2. Specify referential integrity constraints:
   a. You can use Sybase Central to add foreign keys to your database.
   b. Alternatively, for each of the foreign keys, issue the following SQL statement against the SQL Anywhere database (using the Interactive SQL utility: dbisql):
   ```
   ALTER TABLE "table-name"
       ADD FOREIGN KEY "foreign_key_name" ("column_name")
       REFERENCES "ref_table_name" ("ref_column_name");
   ```
   With the new foreign key constraints in place, the SQL Anywhere database checks for referential integrity automatically and greatly improves data integrity.

Properly placed indexes improve database performance significantly, while poorly placed ones hinder performance with equal significance. SQL Anywhere 12 offers the Index Consultant that inspects database usage and workload and recommends changes to the indexing structure as needed. PostgreSQL dictates that foreign key columns must have indexes explicitly defined, but this is not the case in SQL Anywhere. Also, for each Primary key, PostgreSQL creates a primary index that is redundant in SQL Anywhere. The Index Consultant will likely recommend removing the redundant indexes that are copied from the PostgreSQL database during the migration process. The Index Consultant can prove to be a useful tool to boost the performance of the migrated SQL Anywhere database. For information about optimizing your schema, refer to your SQL Anywhere documentation and the SQL Anywhere developer resources available online at http://www.sybase.com/developer/library.

Migrating applications from PostgreSQL to SQL Anywhere

Application migration from PostgreSQL to SQL Anywhere depends on the interface used to access your PostgreSQL application. The following are some of the more popular interfaces that should required only minimal work to migrate:

- **ODBC**: Both SQL Anywhere and PostgreSQL support the ODBC 5.1 API specification. Generally, migration of these applications involves changes the ODBC data source to point to SQL Anywhere instead of PostgreSQL. There may be some specific differences in terms of the implementations of creating API functions, but given the maturity of the ODBC specification, these should be minor.
- **JDBC:** PostgreSQL has a type 4 JDBC Driver (100% Java implementation). To migrate to the SQL Anywhere equivalent, the Sybase jConnect driver should be used. However, to achieve the maximum performance benefits of SQL Anywhere, it is recommended that you use the SQL Anywhere JDBC driver. The SQL Anywhere JDBC driver is a type 2 JDBC driver. The SQL Anywhere JDBC drivers support all of the core elements of the JDBC 3.0 and JDBC 4.0 specifications.
- **Python:** For information about migrating Python applications, see "Migrating a Python application from PostgreSQL to SQL Anywhere" on page 12.
- **Perl:** For information about migrating Perl applications, see "Migrating a Perl application from PostgreSQL to SQL Anywhere" on page 12.
- **PHP:** For information about migrating PHP applications, see "Migrating a PHP application from PostgreSQL to SQL Anywhere" on page 13.

Applications written using the other interfaces supported by PostgreSQL will require more work to migrate as there is no support for these drivers in SQL Anywhere. This includes the PostgreSQL C/C++ API and the Tcl, and Eiffel access drivers. In some cases, a third-party driver may be found that allows you to bridge to ODBC and natively access SQL Anywhere.

Migrating a PYTHON application from PostgreSQL to SQL Anywhere

Migrating Python applications from PostgreSQL to SQL Anywhere is very simple. The SQL Anywhere Python database interface, sqlanydb, is included with SQL Anywhere. This API specification defines a set of methods that provides a consistent database interface independent of the actual database being used.

The sqlanydb module implements the Python Database API specification v2.0. The module is thread-safe when using Python with threads.

To use the sqlanydb module, you will need to have the Python ctypes module included with your Python installation.

Migrating a Perl application from PostgreSQL to SQL Anywhere

Migrating Perl applications from PostgreSQL to SQL Anywhere is very simple. You have the option of using ODBC to connect using the DBD::ODBC driver or using the native SQL Anywhere driver (called DBD::SQLAnywhere) that is included with SQL Anywhere.

If you are already using the DBD::ODBC driver, application migration is simply a matter of changing your connection string to refer to SQL Anywhere. Once that is complete, there may be some minor tweaks required to deal with the differences between SQL Anywhere and PostgreSQL as discussed in pervious sections of this paper, but minimal work is required to complete the migration.

Some PostgreSQL-specific methods can be migrated to SQL Anywhere equivalents by using queries or standard DBD functionality. For example:

PostgreSQL	Equivalent SQL Anywhere	Note
PostgreSQL_insertid	SELECT @@identity	
is_blob, is_num, is_not_null, length, name, table, type	NAME, TYPE, SCALE, PRECISION, NULLABLE	All of these property items are DBD standard elements.
is_key, is_pri_key	SELECT .. FROM syscolumn WHERE	Detection of indexes/keys can be done by looking at the table and column definitions in the system tables.

Migrating a PHP application from PostgreSQL to SQL Anywhere

Migrating a PHP application from PostgreSQL to SQL anywhere is simple. You have the option of using ODBC to connect to SQL Anywhere or using the SQL Anywhere PHP module.

Windows users may prefer to migrate to the ODBC API. Setting up a DSN in Windows for use with ODBC is simple. In addition, the Windows binary for PHP already has built-in ODBC support.

Linux users, on the other hand, may find the PHP module more convenient to set up. SQL Anywhere support can be compiled into PHP using the –with sqlanywhere=[path_to_sa] flag when calling the configure script. Details about the module can be found in the "SQL Anywhere PHP API" chapter of the "SQL Anywhere Server – Programming" manual.

If the PHP application is already using ODBC to connect to the PostgreSQL database, then there is no need to change the function calls. You can skip the section below and go directly to "PHP migration notes" on page 14.

Function mapping

The PostgreSQL, ODBC and SQL Anywhere APIs are very similar. It is often possible to map one function directly to another. Sometimes, when a function has no equivalent counterpart, you must be creative and write alternative code that achieves the same result. In certain cases, you may be better off rewriting small portions of the code to take advantage of advanced features provided by SQL Anywhere. For example, with transaction support, the application can efficiently maintain atomicity and easily ensure data integrity.

The following table lists some commonly used PostgreSQL functions and their ODBC and SQL Anywhere equivalents:

PostgreSQL	Equivalent SQL Anywhere (ODBC)	Equivalent SQL Anywhere (PHP Module)
PostgreSQL_close	odbc_close	sasql_disconnet
PostgreSQL_connect	odbc_connect	sasql_connect
PostgreSQL_errno	odbc_error	sasql_errorcode
PostgreSQL_error	odbc_errormsg	Sasql_error
PostgreSQL_escape_string	See "PostgreSQL_escape_string" note.	sasql_escape_string()

PostgreSQL_fetch_row	odbc_fetch_row	sqlanywhere_fetch_row
PostgreSQL_insert_id	See "PostgreSQL_insert_id" note.	sasql_insert_id
PostgreSQL_num_fields	odbc_num_fields	sqlanywhere_num_fields
PostgreSQL_num_rows	odbc_num_rows	sqlanywhere_num_rows
PostgreSQL_query	odbc_exec	sqlanywhere_query
PostgreSQL_select_db	none	none

Notes:

PostgreSQL_escape_string: The ODBC php driver does not provide a way to escape a SQL string. However, this can be easily done by replacing each single quote with two single quotes.

PostgreSQL_insert_id: This function returns the last inserted ID of an autoincrement column. The same result can be obtained by issuing the following SQL statement:

SELECT @@identity

PHP migration notes

These are subtle differences in the way SQL strings are treated by the various database vendors. For example, timestamps in PostgreSQL have the format YYYY-MM-DD hh:mm:ss, while SQL Anywhere supports timestamps with fractions of a second. Extra work must be done to remove the fractional second portion of the SQL Anywhere timestamp for sue with strtotime() for example.

SQL Anywhere via ODBC also provides support for transactions and prepared statements. The odbc_commit and odbc_rollback functions terminate a transaction as you would expect. One point to notice is that PHP defaults to autocommit, meaning every statement is committed as soon as it is successfully executed. The odbc_autocommit function is used to set the autocommit behavior to enable the use of large transactions. Prepared statements are useful if the same queries, possibly with different parameters, are to be executed many times. This can help create efficiency as each dynamic SQL statement is built within the engine once only. The odbc_prepare and odbc_execute functions are used to execute prepared statements.

Summary

Migrating from PostgreSQL to SQL Anywhere involves migrating the database, changing PostgreSQL function calls to SQL Anywhere calls, and tweaking the schema and SQL statements to resolve any differences between the databases. Typically, some performance gains can be achieved by utilizing advanced features available in SQL Anywhere.

SYBASE®