SYBASE®

WHITE PAPER

10 Cool New Features in SQL Anywhere® 12

TABLE OF CONTENTS

FEATURE NUMBER 1: EVERY APP'S GOT A MAP

Spatial data support is the biggest single feature added to SQL Anywhere since Java in the Database in 1998. For those who don't remember that far back, Java in the Database was far more than just the ability to write stored procedures in Java, it included the use of Java classes as column data types, it even let you build indexes on those columns.

The similarities between Java and spatial are striking: spatial data support is built upon user-defined extended types with a distinct object-oriented flavor: the NEW constructor, instance methods, types and subtypes, even indexes on columns using the new types... different syntax but a lot of the same ideas.

At this point, folks who do remember Java in the Database are probably screaming with frustration: "Spatial data is nothing like Java in the Database!" and they're right: support for Java classes as column data types is gone, kaput, an impressive technical achievement with no market demand.

On the other hand, support for spatial types as column data types has a huge market, a bright future; think smartphones, location awareness, Google Maps... every app's got a map or is going to get one.

With any huge new feature there is an intellectual barrier to entry. In 1998 the barrier came in the form of a question "What's Java?" If you didn't have a clue about Java and you were a developer, you had a problem.

Now, the question is "What's spatial?" Same problem: If you're a developer asking that question you've got some reading to do before you can start building serious applications using the spatial data support in SQL Anywhere 12.

No, Let's Do This Instead

But, instead of study, let's take a modern, new-age approach: "Learning comes later, let's have some fun with it first!" Let's work through a demo that shows off just a bit of what spatial support in SQL Anywhere can do, and just how easy it is to write the code.

For those who insist "Tell me what spatial is!", start with "What is a Spatial Reference System?" at http://www.geod.nrcan.gc.ca/edu/geod/reference/reference01_e.php

...then, when you've had enough, come back here to Step 1.

Spatial Demo Step 1: Install Spatial Support

SQL Anywhere is supposed to have a small footprint, and many SQL Anywhere databases won't need spatial support, so the dbinit database initialization process doesn't automatically bloat up, er, fully populate the spatial catalog tables. It's up to you to do that, but they've made it easy; here's all you have to run to make this demo work:

```
CALL sa_install_feature ( 'st_geometry_predefined_srs' );
```

As it turns out, even that one statement isn't needed for this demo because a small amount of spatial support is included by default... not enough to make the tutorial in the SQL Anywhere Help work, but enough for this demo.

Spatial Demo Step 2: Get Yourself a Shapefile

What's a shapefile? When you start building real spatial applications you'll learn more than you ever wanted to know about shapefiles, but for the moment let's just say a shapefile is a map. You can create one, or grab one off the internet... guess which way is easier.

For this demo the florida.shapefiles.zip file was chosen, pretty much at random, from the CloudMade.com page at http://downloads.cloudmade.com/north_america/united_states/florida

Here's an excerpt from the readme that came with it:

```
This archive includes 5 filesets that contain data in ESRI shapefile format.
ESRI shapefile is a digital vector storage format for storing geometric
location and associated attribute information.
*.shp - shape format; the feature geometry itself
*.shx - shape index format; a positional index of the feature geometry
        to allow seeking forwards and backwards quickly
*.dbf - attribute format; columnar attributes for each shape, in dBase III
        format
See http://en.wikipedia.org/wiki/Shapefile for more information about ESRI
shapefiles.
```

Not mentioned in the readme, but included in each fileset, are the *.prj files which help identify what kind of spatial geometry is being used.

The readme mentions five filesets, this demo uses three of them: The "coastline" files contain an outline of Florida's seacoast plus Lake Okeechobee and some rivers, the "highway" files contain information about roads and streets and the "poi" files contain the locations some "points of interest". Here's the full list:

```
florida_coastline.dbf
florida_coastline.prj
florida_coastline.shp
florida_coastline.shx
florida_highway.dbf
florida_highway.prj
florida_highway.shp
florida_highway.shx
florida_poi.dbf
florida_poi.prj
florida_poi.shp
florida_poi.shx
```

Spatial Demo Step 3: Figure out the SRID

"Figure out the SRID" means "determine which one of a thousand different Spatial Reference Systems must be used when processing your shapefiles"... so, you guessed it, SRID means Spatial Reference Identifier. In the SQL Anywhere catalog tables it's given the column name srs_id, and it contains integer numbers like 2001 for the "Antigua 1943 / British West Indies Grid" and 21898 for the "Bogota 1975 / Colombia East Central zone".

So, how does one pick the SRID? Apparently (for those of us skipping the "study" part) one technique involves matching the contents of the prj project file with the string stored in the SQL Anywhere catalog column SYSSPATIALREFERENCESYSTEM.definition.

In this case, all the Florida project files are the same; here's what florida_coastline.prj contains:

```
GEOGCS["WGS 84",DATUM["WGS_1984",SPHEROID["WGS 84",6378137,298.257223563, AUTHORITY
["EPSG","7030"]],TOWGS84[0,0,0,0,0,0,0],AUTHORITY["EPSG","6326"]],PRIMEM["Greenwich"
,0,AUTHORITY["EPSG","8901"]],UNIT["degree",0.01745329251994328,AUTHORITY["EPSG","912
2"]],AUTHORITY["EPSG","4326"]]
```

The following query takes a few characters from the front of the project file and uses a "LIKE xxx%" predicate to compare it to the data in the database:

```
SELECT object_id,
   srs_name,
   srs_id AS SRID,
   definition
 FROM SYSSPATIALREFERENCESYSTEM
 WHERE definition
   LIKE 'GEOGCS\\["WGS 84",DATUM\\["WGS_1984",%
   ESCAPE '\\';
```

As always, I am cursed by the Demo Gods. Anyone else running a query like that would get one row, I get two, and I had to pick:

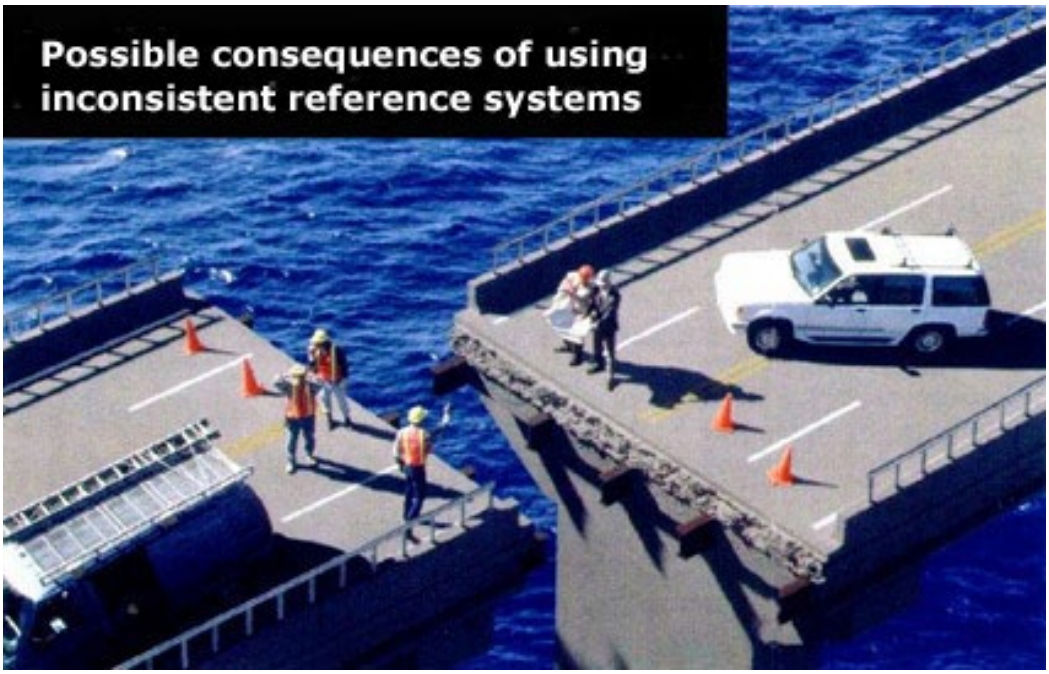
```
object_id srs_name SRID definition
--------- --------------- ---------- ----------------------------------------

   2951 WGS 84 4326 GEOGCS["WGS 84",DATUM["WGS_1984",SPHEROID[...
   2952 WGS 84 (planar) 1000004326 GEOGCS["WGS 84",DATUM["WGS_1984",SPHEROID[...
```

I picked the first row: SRID = 4326. Turns out it's probably the right pick. Or it might be the right pick. Or it might not make a difference.

Or... it might make a difference, judging from this warning in the "What is a Spatial Reference System?" article mentioned earlier:

Figure 1: Why it's important to pick the right SRID



But, we're not building a bridge here, we're doing a demo, so let's move on.

Spatial Demo Step 4: Create a Shapefile Table

Before we can load the shapefile data into SQL Anywhere, we need to create a table, and before we can do that we need to know what the data in the shapefile looks like

The built-in procedure sa_describe_shapefile will tell us that: it reads information from the .shp and .dbf files to create a result set containing the name and data type of each column in the .shp file.

You can call sa_describe_shapefile() to see what the data looks like and then code the CREATE TABLE by hand, or you can use the code in Figure 2 to have SQL Anywhere generate the entire CREATE TABLE for you, in a file ready to run.

Figure 2: Generating the CREATE statement for a shapefile table

```
 1  UNLOAD
 2  SELECT STRING (
 3   'CREATE TABLE Florida_Coastline ( \x0d\x0a',
 4   LIST ( STRING (
 5   ' "',
 6   describe.name,
 7   '" ',
 8   describe.domain_name_with_size,
 9   ',\x0d\x0a' ),
10   '' ORDER BY describe.column_number ),
11   ' ',
12   'PRIMARY KEY ( record_number ) );' )
13   FROM sa_describe_shapefile (
14   'florida_coastline.shp',
15   4326 ) AS describe
16   TO 'create_Florida_Coastline.sql' ESCAPES OFF QUOTES OFF;
```

Lines 1 and 16 in Figure 2 are an UNLOAD statement wrapped around a SELECT. The TO clause specifies the output filespec, and the ESCAPES OFF QUOTES OFF clauses make sure the data is written with no extra decorations.

Lines 2 through 15 return a single row consisting of a single string value. The STRING function concatenates the four arguments into one return string.

Lines 13 through 15 calls the new built-in sa_describe_shapefile procedure that is part of the spatial data support in SQL Anywhere 12. This procedure reads the shp shapefile named in the first argument, analyzes the data using the rules for SRID 4326 (the second argument), and returns one row describing each field in each record in the shapefile data: the field name and its domain_name_with_size.

It's not clear from the code, but sa_describe_shapefile also reads the associated dbf file, in this case florida_coastline.dbf.

The LIST aggregate function call on lines 4 through 10 builds an ordered list of column definitions from data in the result set returned sa_describe_shapefile. Figure 3 shows what the output looks like.

Figure 3: The generated file create_Florida_Coastline.sql

```
CREATE TABLE Florida_Coastline (
   "record_number" int,
   "geometry" ST_Geometry(SRID=4326),
   "NATURAL" varchar(9),
   "NAME" varchar(16),
   PRIMARY KEY ( record_number ) );
```

Spatial Demo Step 5: Load the Shapefile Table

After running the CREATE TABLE shown in Figure 3 above, the code in Figure 4 can be used to load the shapefile data into the Florida_Coastline table. The FORMAT SHAPEFILE clause is part of the new spatial data support, and once again, more than just the shp shapefile table is needed; behind the scenes, the LOAD TABLE in Figure 4 also reads the florida_coastline.shx and florida_coastline.dbf files.

Figure 4: Loading the Florida_Coastline table

```
LOAD TABLE Florida_Coastline
  USING FILE 'florida_coastline.shp' FORMAT SHAPEFILE;
7149 row(s) affected
Execution time: 7.547 seconds
```

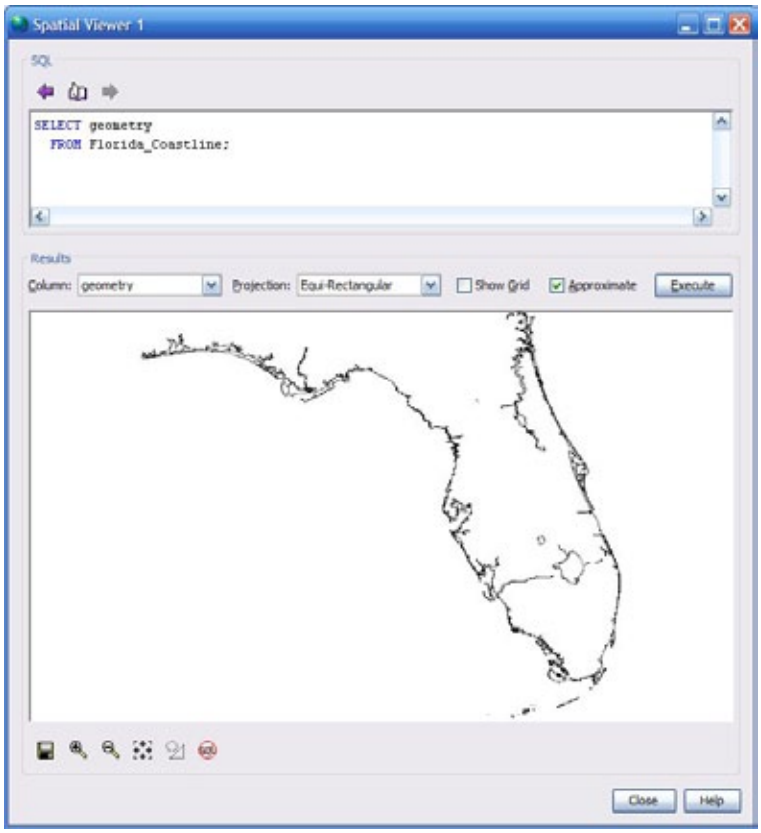
Spatial Demo Step 6: Display the Shapefile Table in ISQL

Now the magic begins: run this simple query in the new ISQL Tools - Spatial Viewer dialog

```
SELECT geometry
  FROM Florida_Coastline;
```

and you'll see the Florida coastline shown in Figure 5.

Figure 5: The Florida Coastline table in the Spatial Viewer



Spatial Demo Step 7: Create an SVG File

The Spatial Viewer is a wonderful tool for developing and debugging but I doubt many folks are going to embed the viewer itself into their spatial applications. Instead, they're going to export the data in formats like SVG, which stands for Scalable Vector Graphics. Figure 6 shows just how easy it is to create a 1 megabyte SVG file from all the data in the Florida coastline table.

Figure 6: Create the Florida_Coastline.svg file

```
UNLOAD
SELECT ST_Geometry::ST_AsSVGAggr ( geometry )
   FROM Florida_Coastline
TO 'c:\temp\Florida_Coastline.svg' ESCAPES OFF QUOTES OFF;
```

The aggregate function ST_Geometry::ST_AsSVGAggr is part of the new spatial data support; understanding the funky new "::" syntax is part of that "learning about spatial data" stuff mentioned earlier... but not discussed here.

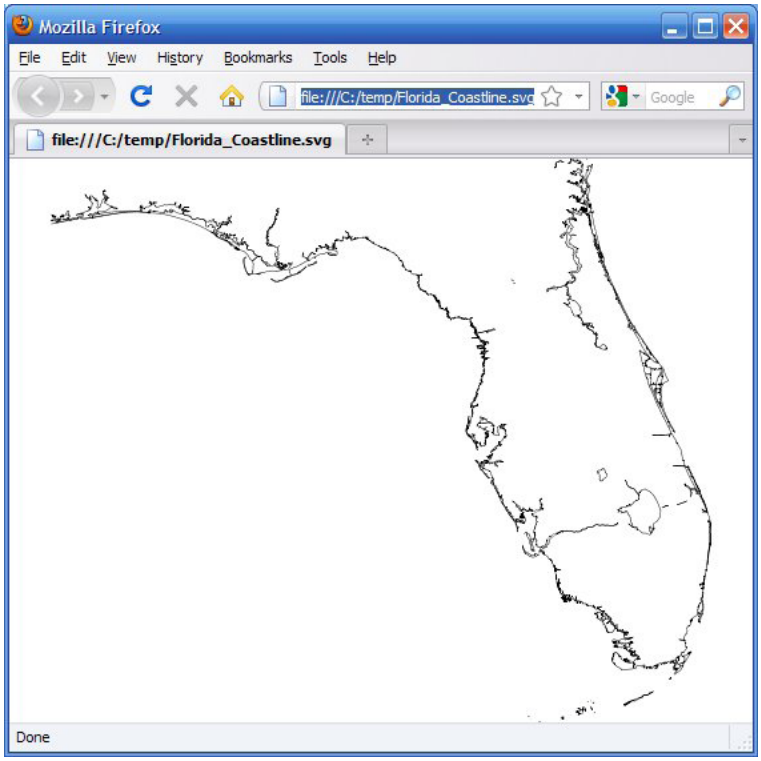
SVG files aren't binary images, they're text files containing XML like the snippet in Figure 7.

Figure 7: Scalable Vector Graphics files contain XML

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg xmlns="http://www.w3.org/2000/svg" version="1.1"
   viewBox="-87.63988 -31.01023 7.61663 6.48921">
<path fill="none" stroke="black" stroke-width="0.1%"
   d="M -87.63227,-30.29473 l 0,-.00052 "/>
<path fill="none" stroke="black" stroke-width="0.1%"
   d="M -87.5733,-30.29513 l -.00177,.00013 -.00474,.00118 -.00415,
   -.00062 -.00415,.00053 -.00502,-.00222 -.00558,-.00104 -.00691,
   -.0022 -.00418,-.00081 -.00472,.00035 -.00711,-.0001 -.00385,
   .00092 -.00443,.00311 .00294,-.00452 .00769,-.00132 -.00329,
   -.00577 .00417,.00371 .0086,.00124 .00265,-.00412 .00473,
   -.0008 .00338,-.00593 "/>
...
<path fill="none" stroke="black" stroke-width="0.1%
   d="M -80.0583,-26.50004 l .00063,-.00408 .00046,-.00424 .00046,
   -.00428 .00119,-.00451 .00123,-.00824 "/>
</svg>
```

In HTTP-speak, the “Content-Type” for an SVG file is “image/svg+xml” and browsers like Firefox can display SVG files directly; see Figure 8.

Figure 8: The Florida Coastline SVG file in Firefox



Spatial Demo Step 8. Load More Shapefiles

Spatial data gets really interesting when you mix and match data from multiple sources. Figure 9 shows how to load the Florida highway and points of interest data into their own tables.

Figure 9: Loading the Highway and POI tables

```
CREATE TABLE Florida_Highway (
   "record_number" int,
   "geometry" ST_Geometry(SRID=4326),
   "TYPE" varchar(14),
   "NAME" varchar(92),
   "ONEWAY" varchar(4),
   PRIMARY KEY ( record_number ) );
LOAD TABLE Florida_Highway
 USING FILE 'florida_highway.shp' FORMAT SHAPEFILE;
15% (17 of estimated 114 MB) complete after 00:00:46;
25% (28 of estimated 114 MB) complete after 00:01:16;
...
80% (92 of estimated 114 MB) complete after 00:04:06;
95% (109 of estimated 114 MB) complete after 00:04:49;
100% (114 of 114 MB) complete after 00:05:03
584926 row(s) affected
Execution time: 305.859 seconds
CREATE TABLE Florida_POI (
   "record_number" int,
   "geometry" ST_Point(SRID=4326),
    "CATEGORY" varchar(30),
   "NAME" varchar(115),
    PRIMARY KEY ( record_number ) );

LOAD TABLE Florida_POI
   USING FILE 'florida_poi.shp'
   FORMAT SHAPEFILE;

30903 row(s) affected
Execution time: 8 seconds
```

Not shown in Figure 9 is the code to generate the CREATE TABLE statements; see the earlier Figure 2 for a sample of that.

What Figure 9 does show is another Cool New Feature in SQL Anywhere 12: A long-running LOAD TABLE statement will now display "progress" statements in ISQL as it runs. This is stealth improvement, too cool to talk about in the What's New section of the Help.

Spatial Demo Step 9: Merge and Display the Shapefile Tables in ISQL

Figure 10 shows how easy it is to merge spatial data via simple UNION operators: points of interest categorized as "Health care" are merged with highways of type "motorway" and all of the coastline data.

Figure 10: Merge shapefile tables via UNION

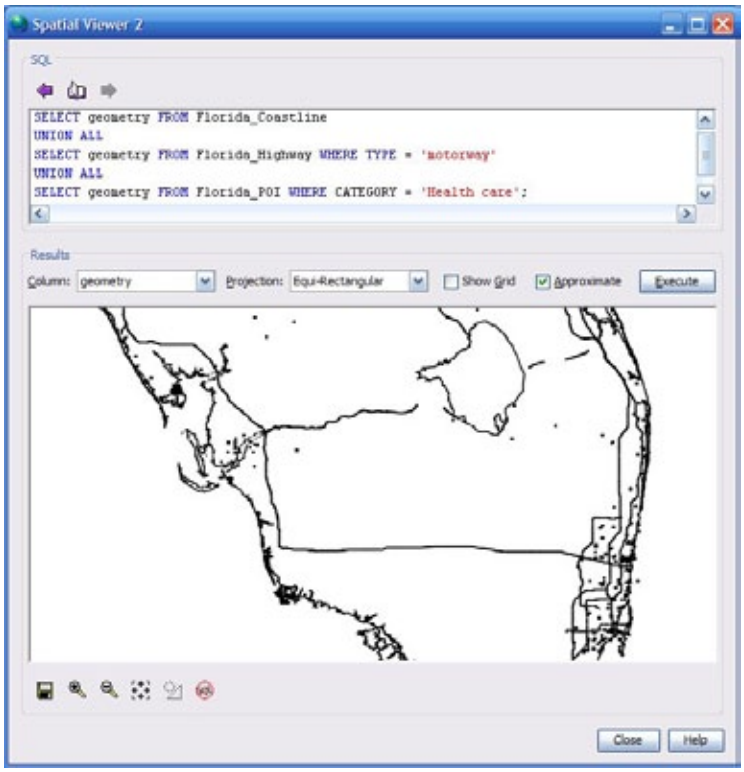
```
SELECT geometry FROM Florida_Coastline
UNION ALL
SELECT geometry FROM Florida_Highway WHERE TYPE = 'motorway'
UNION ALL
SELECT geometry FROM Florida_POI WHERE CATEGORY = 'Health care';
```

The UNION operators in Figure 10 are qualified with ALL for performance: The three result sets are large and they're all different so it's good to eliminate the sort-and-remove-duplicates processing implied by DISTINCT (the default).

Figure 10 answers the question "Where are all the health care facilities relative to the highways in Florida?" Little black dots show where the health care points of interest are located, and the lines show the highways and the coastline.

The Spatial Viewer lets you scroll and zoom. The image in Figure 11 has been zoomed to show the east-west portion of Interstate 75 known as Alligator Alley, and the answer to the question above is "You're out of luck for health care facilities on that stretch of road."

Figure 11: Merged shapefile tables in the Spatial Viewer



Spatial Demo Step 10: Display via Web Service in Firefox

Ever since 2003 the SQL Anywhere database server has contained a built-in HTTP server. That means the database server can listen to port 80, and stored procedures can respond to web service requests by returning HTML text to the browser. Or binary images. Or XML text. Or, now that spatial data support has been added, SVG images.

Figure 12 shows the code for a SQL Anywhere web service that returns the coastline, highway and health care data to the browser as an SVG image using Content-Type 'image/svg+xml'.

Figure 12: Web service returning SVG image

```
 1  CREATE SERVICE root
 2   TYPE 'RAW' AUTHORIZATION OFF USER DBA
 3   AS CALL p();
 4
 5  CREATE PROCEDURE p() RESULT ( html_string LONG VARCHAR )
 6  BEGIN
 7
 8  CALL dbo.sa_set_http_header( 'Content-Type', 'image/svg+xml' );
 9
10  SELECT REPLACE (
11   ST_Geometry::ST_AsSVGAggr ( geometry ),
12   '<rect width="0.1%" height="0.1%" stroke="black"',
13   '<rect width="0.3%" height="0.3%" stroke="red"' )
14   FROM ( SELECT geometry
15   FROM Florida_Coastline
16   UNION ALL
17   SELECT geometry
18   FROM Florida_Highway
19   WHERE Florida_Highway.TYPE = 'motorway'
20   UNION ALL
21   SELECT geometry
22   FROM Florida_POI
23   WHERE CATEGORY = 'Health care' ) AS map;
24
25  END;
```

Lines 1 through 3 in Figure 12 define a web service that will respond to browser requests and call the stored procedure p() to return a response. The special service name "root" specifies the default "no name required" setting; i.e., you don't have to specify a service name in the url, just type http://localhost. The 'RAW' type means anything goes; it's up to the stored procedure to do all the work building the response, SQL Anywhere isn't going to perform any special formatting.

The CALL at line 8 specifies the HTTP Content-Type.

The SELECT starting at line 10 does what a RAW web service requires of the procedure it calls: it returns a single row result set consisting of a single LONG VARCHAR column.

The inner SELECT on lines 14 through 23 is the same query as the one shown earlier in Figure 10, coded here as a derived table so some post-processing can be performed by the outer SELECT.

The REPLACE call on lines 10 through 13 performs two functions. First, it calls the aggregate function ST_Geometry::ST_AsSVGAggr first shown back in Figure 6. Then, it does a string replacement on the resulting XML text to fatten up the health care dots and color them red. This is done so they'll show up better in the output. Yes, it's a kludge. Yes, it works. This time, in this demo, it works.

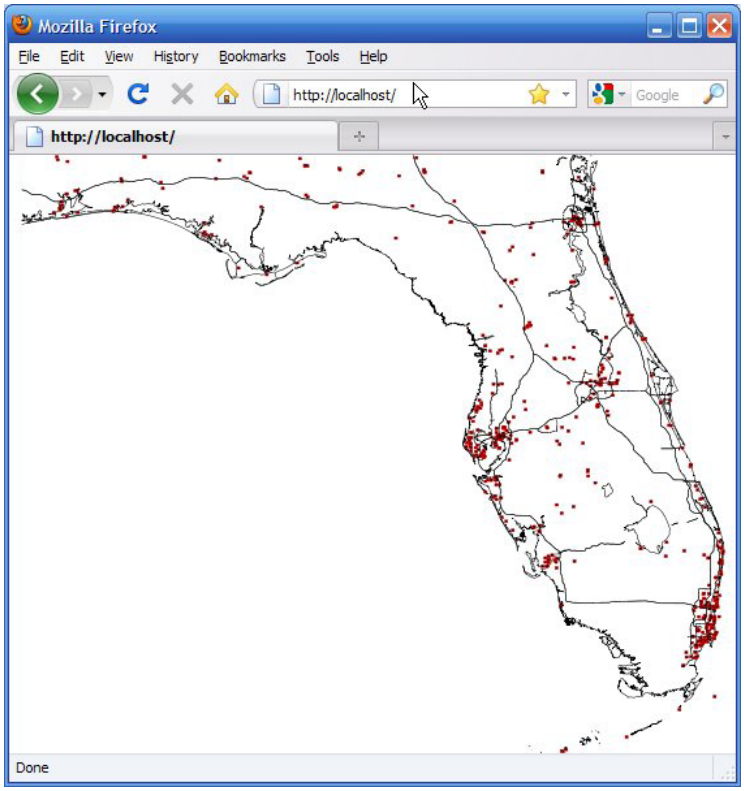
Figure 13 shows the command line for starting the SQL Anywhere database engine with the HTTP server enabled, followed by the url to use in the browser.

Figure 13: Launching and connecting to SQL Anywhere's HTTP server

```
"%SQLANY12%\bin32\dbeng12.exe" -xs http(port=80) ddd12.db
http://localhost
```

Figure 14 shows the results in Firefox: Fatter health care dots, colored red for extra visibility.

Figure 14: Merged shapefile tables in Firefox



FEATURE NUMBER 2: QUERY QUARANTINE

Here's what they call this feature in the docs: "read-only scale-out"... I'm not kidding, who comes up with these names?

And here's what read-only scale-out means: You can create one, two, three copies of your main database, as many copies as you want, and let those folks in Finance run their Queries From Hell against one of the copies, or two copies, or three copies, or as many copies as THEY want, and you won't ever have to hear your paying customers complain that they can't place an order on your website because some back-office numbers wonk is sucking up all the cycles.

Those copies are all kept up to date using the same real-time log record push process that was first implemented in "live backup" many years ago and later enhanced to give us high availability in Version 10.

The "read-only" part means the folks in Finance can't do updates, and "scale-out" means the folks in Finance can suck up the cycles on their OWN machines, separate from the server that's generating income rather than incurring expenses.

Pushing the folks in Finance onto their own servers is why I call it "Query Quarantine" and let's face it, who hasn't wanted to put them in quarantine at one time or another?

Query Quarantine Step 1: Create the Root Database

Here's the code: Figure 15 shows how to get started with a brand new database that's going to be used as the updatable "primary" or "root" database in the read-only scale-out setup.

Figure 15: Create the updatable root database and start the root server

```
1  "%SQLANY12%\bin32\dbinit.exe"^
2   c:\data\root.db
3
4  "%SQLANY12%\bin32\dbping.exe"^
5   -c "SERVER=root_server;DBF=c:\data\root.db;UID=dba;PWD=sql"^
6   -d^
7   -o c:\data\dbping_root_log.txt
8
9  "%SQLANY12%\bin32\dbspawn.exe"^
10  -f^
11  "%SQLANY12%\bin32\dbsrv12.exe"^
12  -n root_server^
13  -o c:\data\root_console_log.txt^
14  -su sql^
15  -x TCPIP(port=50000)^
16  c:\data\root.db^
17  -xp on
18
19 "%SQLANY12%\bin32\dbisql.com"^
20  -c "SERVER=root_server;DBN=root;UID=dba;PWD=sql"
```

Lines 1 and 2 in Figure 15 create a new SQL Anywhere database in the C:\data folder.

The dbping command on lines 4 through 7 meets the special requirement that the transaction log must exist before the database can be started as a read-only scale-out primary. The SERVER and DBF options, together with the -d, force dbping to actually start the database so it can be pinged... and starting a new database has the side effect of creating its transaction log.

Lines 9 through 17 start the database as a primary. The dbspawn utility is used for convenience so control will be returned to the batch file after the server starts rather than waiting until the server is stopped.

Line 12 specifies one of the server names that will be used for this database, and line 16 implicitly specifies the database name as "root". Here's a tip: Don't try to specify different database names when setting up root and copy databases, that is a one-way path to interesting error messages.

In this demo the root and all the copy databases will be run on the same computer, so different ports must be used. The code on line 15 specifies 50000 as the port for the primary.

The -xp on option on line 17 is what tells SQL Anywhere that read-only scale-out is being used, and it's the reason the transaction log must exist before the server is started.

The code on lines 19 and 20 starts the first of four ISQL sessions you'll see later on; this one is directly connected to the updatable primary database via the "root_server" name.

Query Quarantine Step 2: Create the Copies

Figure 16 shows how to define the SQL Anywhere "MIRROR SERVER" objects required to implement read-only scale-out, set some "MIRROR OPTION" values and then use BACKUP statements to create physical copies of the main database file.

Figure 16: Create the read-only copies of the root database

```
 1  CREATE MIRROR SERVER "primary_server"
 2   AS PRIMARY
 3   connection_string = 'SERVER=primary_server;HOST=localhost:50000';
 4
 5  CREATE MIRROR SERVER "root_server"
 6   AS PARTNER
 7   connection_string = 'SERVER=root_server;HOST=localhost:50000';
 8
 9  SET MIRROR OPTION auto_add_server = 'root_server';
10  SET MIRROR OPTION child_creation = 'automatic';
11  SET MIRROR OPTION authentication_string = 'abc';
12  SET MIRROR OPTION auto_add_fan_out = '10';
13
14  BACKUP DATABASE DIRECTORY 'c:/data/copy1';
15  BACKUP DATABASE DIRECTORY 'c:/data/copy2';
16  BACKUP DATABASE DIRECTORY 'c:/data/copy3';
```

I'll be honest, the code between lines 1 and 12 in Figure 16 is pretty much a mystery to me, especially the SET MIRROR OPTION statements... those I have no clue about, but they're necessary... I think. It's all new to SQL Anywhere 12, and I'm invoking the "learn later, play now" rule.

Here's what I do know: The two CREATE MIRROR SERVER statements on lines 1 and 5 refer to the same physical server by two different names: "primary_server" and "root_server" are both on port 50000, one is defined as a PRIMARY and the other as a PARTNER.

The BACKUP statements on lines 14 through 16 are easier to explain: they create three subfolders under the C:\ data folder, and in each of those subfolders they create the root.db and root.log files that are copies of the primary database. These are the database files that will be used as read-only copies.

Query Quarantine Step 3: Start the Copies

Figure 17 shows how to start all three read-only database copies on their own database servers.

Figure 17: Start the read-only databases on separate servers

```
1  "%SQLANY12%\bin32\dbspawn.exe"^
2   -f^
3   "%SQLANY12%\bin32\dbsrv12.exe"^
4   -n copy1_server^
5   -o c:\data\copy1\copy1_console_log.txt^
6   -su sql^
7   -x TCPIP(port=50001)^
8   c:\data\copy1\root.db^
9   -xp on
10
11 "%SQLANY12%\bin32\dbspawn.exe"^
12  -f^
13  "%SQLANY12%\bin32\dbsrv12.exe"^
14  -n copy2_server^
15  -o c:\data\copy2\copy2_console_log.txt^
16  -su sql^
17  -x TCPIP(port=50002)^
18  c:\data\copy2\root.db^
19  -xp on
20
21 "%SQLANY12%\bin32\dbspawn.exe"^
22  -f^
23  "%SQLANY12%\bin32\dbsrv12.exe"^
24  -n copy3_server^
25  -o c:\data\copy3\copy3_console_log.txt^
26  -su sql^
27  -x TCPIP(port=50003)^
28  c:\data\copy3\root.db^
29  -xp on
```

The three dbspawn-dbsrv12 commands Figure 17 are exactly the same as the one that starts the primary server in Figure 15, except for the following:

- different server names: copy1_server, copy2_server, copy3_server
- different ports: 50001, 50002, 50003
- different subfolders: c:\data\copy1, c:\data\copy2, c:\data\copy3

These differences are important to the process of actually starting the copy databases, but they're not needed for making connections. In fact, none of the different names and values specified in Figure 17 appear in any code anywhere else. It's magic.

Query Quarantine Step 4: Update the Root

Figure 18 shows a change that is made to the root database after the three copies have been created and started.

Figure 18: Update the root database

```
1  CREATE TABLE Hello ( MessageText VARCHAR ( 100 ) );
2  INSERT Hello VALUES ( 'Hello, world!' );
3  COMMIT;
4
5  SELECT MessageText,
6   DB_PROPERTY ( 'File' )
7   FROM Hello;
```

Lines 1 to 3 insert a new row into a new table, and the SELECT displays that row together with the physical file specification for the database.

Since all the databases are located on the same computer in this demo, the value returned by DB_PROPERTY ('File') will serve to answer the question "Which database am I looking at?"

Query Quarantine Step 5: Query the Copies

Figure 19 shows the command to start an ISQL session on one of the copy databases, plus a SELECT that serves two purposes: First, it proves that the Hello table and its data has been sent to the copy database, and second, the DB_PROPERTY ('File') tells us which copy is being used.

Figure 19: Connect to a read-only database and run a query

```
1  "%SQLANY12%\bin32\dbisql.com"^
2   -c "SERVER=primary_server;NODETYPE=copy;UID=DBA;PWD=sql"
3
4  SELECT MessageText,
5   DB_PROPERTY ( 'File' )
6   FROM Hello;
```

The NODETYPE=copy option on line 2 is what tells SQL Anywhere to not actually make a connection to SERVER=primary_server but to pass the connection over to one of the three copies started by the commands in Figure 17 earlier.

Which copy does it connect to? Sometimes copy 1, sometimes copy 2, sometimes copy 3, it depends.

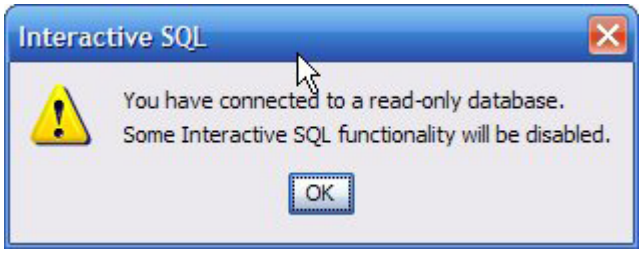
Depends on what? SQL Anywhere decides, usually based on load. You don't get to pick.

How does it know where the copy databases are? There is nothing in any of the code presented so far that explicitly associates any of the three servers in Figure 17 with the one called "primary_server".

The answer to that last question is... magic. It's magic. Let's move on.

Figure 20 shows the warning displayed by ISQL when the command in Figure 19 connects to a read-only database.

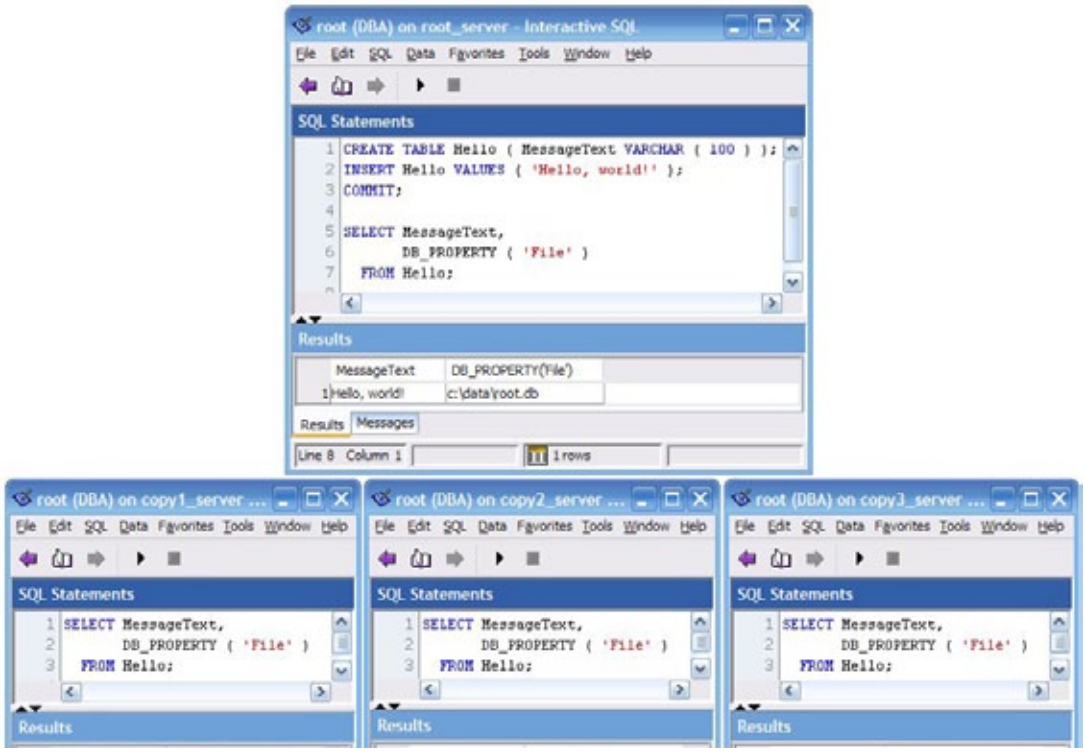
Figure 20: ISQL read-only warning



The top ISQL window in Figure 21 shows the commands from Figure 18 run against the primary database, and the bottom three windows are the result of running the code in Figure 19 three times.

In order to show all three copy databases in use, each copy was shut down after an ISQL session connected to it in order to force the next ISQL session to connect to a different copy.

Figure 21: Four ISQL sessions, four different databases



FEATURE NUMBER 3: SELECT FROM INSERT UPDATE

Have you ever wanted to INSERT a row and immediately retrieve that row without having to code a separate SELECT? How about UPDATE and retrieve?

Let's say the row has a DEFAULT AUTOINCREMENT column, do you ever need to know the value that was just inserted? What about an UPDATE SET column = expression, do you ever need to know the old and new values in your application?

Now you can do all of that by coding your INSERT and UPDATE statements as "DML derived tables" in the FROM clause of a SELECT. DML means Data Manipulation Language which is DBA-speak for insert, update and delete, as opposed to DDL or Data Definition Language which really means CREATE TABLE.

Figure 22 shows two DML derived tables in action, one INSERT and one UPDATE.

Figure 22: SELECT FROM INSERT and UPDATE

```
1  CREATE TABLE item (
2   item_number INTEGER NOT NULL DEFAULT AUTOINCREMENT PRIMARY KEY,
3   quantity INTEGER NOT NULL,
4   price DECIMAL ( 11, 2 ) NOT NULL,
5   inserted_on TIMESTAMP DEFAULT CURRENT TIMESTAMP,
6   updated_on TIMESTAMP DEFAULT TIMESTAMP );
7
8  BEGIN
9  DECLARE @quantity DECIMAL ( 11, 2 );
10 DECLARE @price DECIMAL ( 11, 2 );
11
12 SET @quantity = 10;
13 SET @price = 199.95;
14
15 SELECT item.item_number,
16  item.quantity,
17  item.price,
18  item.inserted_on
19  FROM ( INSERT item (
20  quantity,
21  price )
22  VALUES (
23  @quantity,
24  @price ) ) REFERENCING ( FINAL AS item );
25
26 WAITFOR DELAY '00:00:01';
27
28 SELECT item.item_number,
29  old_item.quantity AS old_quantity,
30  old_item.price AS old_price,
31  item.quantity,
32  item.price,
33  item.updated_on
34  FROM ( UPDATE item SET price = price * 1.1 )
35  REFERENCING ( OLD AS old_item FINAL AS item );
36
37 END;
38
39 item_
40 number quantity price inserted_on
41 ------ -------- ------ -----------------------
42  1 10 199.95 2010-04-03 07:56:41.712
43
44 item_ old_ old_
45 number quantity price quantity price updated_on
46 ------ -------- ------ -------- ------ -----------------------
47  1 10 199.95 10 219.95 2010-04-03 07:56:42.744
```

The FROM clause starting on line 19 in Figure 22 shows an INSERT statement that specifies values for two columns and lets the other three columns (item_number, inserted_on and updated_on) be assigned their DEFAULT values.

The INSERT statement is wrapped in (parentheses) to show it is a derived table. The INSERT is followed by a REFERENCING clause that works much like the REFERENCING clause in a trigger: it assigns the correlation name "item" to the result set representing the FINAL after-image of the inserted row.

Tip: In a demo it might be seem a bit confusing to reassign the table's own name "item" as the correlation name for a derived table, but in the real world it often helps: It really is the same data, and it's the only thing we're interested in, in the outer SELECT.

Part of the beauty of DML derived tables is that, as with triggers, you have access to the entire row rather than just the columns explicitly named in the INSERT. That's shown by the fact the SELECT list starting at line 15 names four columns whereas the INSERT starting at line 19 only specifies two.

The FROM clause on lines 34 and 35 shows a DML derived table based on a simple UPDATE. Here, the REFERENCING clause defines two separate correlation names: old_item contains the row before the UPDATE and item contains the after-image.

The SELECT list starting at line 28 names six columns, two of them from the row before the UPDATE and four from the final version of the row.

The output from the two SELECT statements is shown on lines 42 and 47 respectively.

FEATURE NUMBER 4: NO MORE PICKING -GN

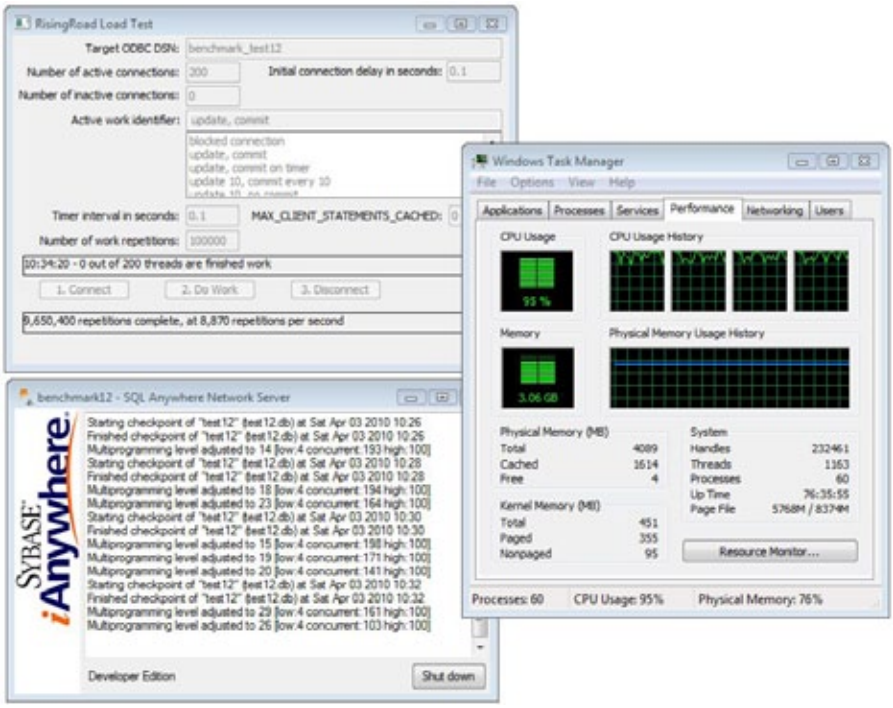
If you don't know what "picking -gn" means then move along, nothing to see here, these aren't the droids we're looking for.

Seriously, "picking -gn" means trying to figure out if the maximum number of SQL Anywhere tasks active at any one time should be changed to some number other than the default 20. The database server option -gn is called the multiprogramming level.

This is an enormously difficult task, pun intended... raising -gn when you have a lot of busy connections may be the right thing to do, or it may be exactly the wrong thing to do. It depends on factors like increased contention, increased server overhead, the possibility of thrashing, and the effect on memory allocation for parsing and optimizing and executing queries. Simple stuff like that... simple for Gandalf, that is, impossible for your average administrator.

Now, you don't have to pick -gn. By default the server will set the multiprogramming level to the -gn value when it starts up, then dynamically adjust the setting between the -gnl lower limit and the -gnh upper limit as the load changes. The defaults are -gn 20, -gnl x where x is the number of licensed CPUs, and -gnh y where y is 4 times the -gn number. Figure 23 shows how this works using a custom PowerBuilder load test program.

Figure 23: Dynamic adjustment of the multiprogramming level



Normally the program in Figure 23 is used to exercise the Foxhound monitor for SQL Anywhere by driving a flood of work through the target database. In this case, it's being used to demonstrate how the multiprogramming level is adjusted under heavy load: 200 separate threads each have a shared memory connection to a local database, and each thread is repeatedly executing random single-row UPDATE and COMMIT transactions as fast as SQL Anywhere can handle them (no "end-user think time" in this simulation).

The top left window shows that SQL Anywhere is processing over eight thousand transactions per second. That's an overall average since the test began, not a snapshot rate... and it's over 50% higher than any rate ever observed using any previous version of SQL Anywhere on the same computer, with the same configuration and exactly the same application and data.

The right hand window shows that the four cpus are being used to their limit; some of that is taken up by the 200 application tasks, most of it's SQL Anywhere, the point being this is a CPU-and-RAM intensive test, the disk drive is not heavily involved.

The bottom left window shows how SQL Anywhere is continuously adjusting the multiprogramming level. One thing that's apparent is the old default of 20 wasn't such a bad choice after all since in this test the actual rate is oscillating above and below the old number. What this oscillation means, I don't know, nor do I care... I'm just happy I no longer have to pick -gn.

Not evident from Figure 23 but clear from the testing is that if you're expecting heavy loads, set the minimum multiprogramming level to something reasonable like 10 or 15. Don't let it sink to 4 or some other ridiculously low number during idle periods because it will take a few minutes to rise again during heavy load. For the test shown here that meant the difference between 3,000 and 8,000 transactions per second.

FEATURE NUMBER 5: SEND EMAIL VIA GMAIL

OK, everyone sends email via Gmail, what's so hard about that?

Here's what "sending email via Gmail" means in this context: You want to write code that runs inside the database server to send an email whenever it detects some condition, like the amount of free disk space has sunk below a critical threshold or a backup has failed to run. You work in a Windows shop where MAPI is the email protocol of choice, there are no SMTP servers to be found, and the latest versions of Microsoft software will only let people send emails, not programs. So, no SMTP, and MAPI doesn't work.

What to do, oh what to do? Well, use SMTP anyway, that's for sure, MAPI sucks. And Google just happens to have an SMTP server you can use, smtp.gmail.com, as long as you have an account. All you need on top of that is a "security certificate" to get you past the firewalls at Google.

And that's where SQL Anywhere 12 comes in: earlier versions let you write code to send email from inside the server but you couldn't specify a security certificate. Now you can.

Where to get a security certificate? If you've got Internet Explorer 8 installed, you probably already have a security certificate that will work; here's how to copy that certificate into a file:

- Choose Tools – Internet Options – Content – Certificates – Trusted Root Certificates
- Scroll down and select Thawte Premium Server CA
- Click on Export
- Check Base-64 encoded X.509 (.CER)
- Save the file as ThawtePremiumServerCA.cer

Figure 24 shows how to call the SQL Anywhere built-in procedures xp_startsmtp and xp_sendmail to send an email via smtp.gmail.com.

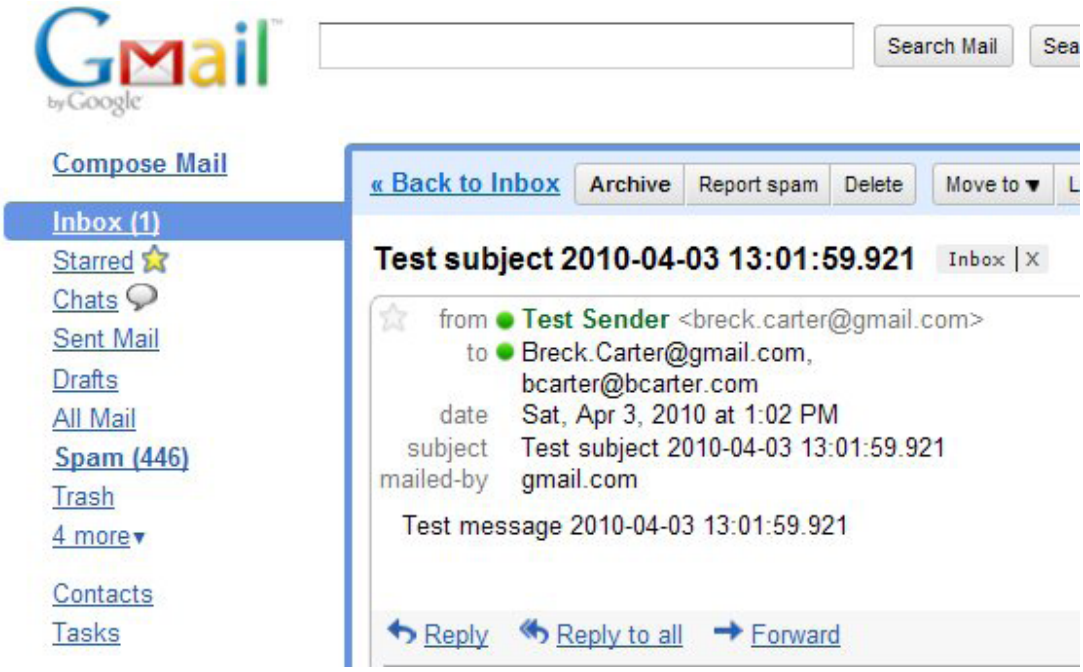
Figure 24: Sending email via smtp.gmail.com

```
1  BEGIN
2  DECLARE @return_code INTEGER;
3
4  DECLARE @smtp_sender LONG VARCHAR;
5  DECLARE @smtp_server LONG VARCHAR;
6  DECLARE @smtp_port INTEGER;
7  DECLARE @smtp_timeout INTEGER;
8  DECLARE @smtp_sender_name LONG VARCHAR;
9  DECLARE @smtp_auth_username LONG VARCHAR;
10 DECLARE @smtp_auth_password LONG VARCHAR;
11 DECLARE @trusted_certificates LONG VARCHAR;
12
13 DECLARE @smtp_recipient LONG VARCHAR;
14 DECLARE @smtp_subject LONG VARCHAR;
15 DECLARE @smtp_message LONG VARCHAR;
16
17 SET @smtp_sender = 'Breck.Carter@gmail.com';
18 SET @smtp_server = 'smtp.gmail.com';
19 SET @smtp_port = 587;
20 SET @smtp_timeout = 10;
21 SET @smtp_sender_name = 'Test Sender';
22 SET @smtp_auth_username = 'Breck.Carter@gmail.com';
23 SET @smtp_auth_password = '......';
24 SET @trusted_certificates = 'ThawtePremiumServerCA.cer';
25
26 @return_code = CALL xp_startsmtp (
27  smtp_sender = @smtp_sender,
28  smtp_server = @smtp_server,
29  smtp_port = @smtp_port,
30  timeout = @smtp_timeout,
31  smtp_sender_name = @smtp_sender_name,
32  smtp_auth_username = @smtp_auth_username,
33  smtp_auth_password = @smtp_auth_password,
34  trusted_certificates = @trusted_certificates );
35
36 MESSAGE STRING ( 'xp_startsmtp returned ', @return_code ) TO CLIENT;
37
38 SET @smtp_recipient = 'Breck.Carter@gmail.com;bcarter@bcarter.com';
39 SET @smtp_subject = STRING ( 'Test subject ', CURRENT TIMESTAMP );
40 SET @smtp_message = STRING ( 'Test message ', CURRENT TIMESTAMP );
41
42 @return_code = CALL xp_sendmail (
43  recipient = @smtp_recipient,
44  subject = @smtp_subject,
45  "message" = @smtp_message );
46
47 MESSAGE STRING ( 'xp_sendmail returned ', @return_code ) TO CLIENT;
48
49 CALL xp_stopsmtp();
50
51 END;
52
53 xp_startsmtp returned 0
54 xp_sendmail returned 0
55 Execution time: 2.407 seconds
```

Line 34 in Figure 24 shows what's new in SQL Anywhere 12: the trusted_certificates argument containing the filespec of your security certificate.

Figure 25 shows the result of running the code in Figure 24: a SQL Anywhere server has sent an email via smtp.gmail.com to a Gmail inbox... and to a regular POP3 inbox not shown here (the second email address in the recipient list).

Figure 25: An email sent via Gmail to Gmail



FEATURE NUMBER 6: PROXY TABLES ARE FAST!

This may be the biggest understatement in the history of SQL Anywhere, one single sentence in the What's New section of the Help:

"SQL Anywhere 12 includes many enhancements to improve remote data access performance."

Figure 26 shows what I'm talking about: code that copies 1.9 million rows, over 2G of data, from a SQL Server 2008 database to a SQL Anywhere in-memory database on another computer.

It took over an hour using SQL Anywhere 11.0.1 but only about 11 minutes on version 12... that's 500% faster!

Figure 26: Copying data from SQL Server to SQL Anywhere via proxy table

```
 1  CREATE SERVER mss
 2   CLASS 'MSSODBC'
 3   USING 'DSN=main_BRECK-PC';
 4
 5  CREATE EXTERNLOGIN DBA
 6   TO mss
 7   REMOTE LOGIN "sa"
 8   IDENTIFIED BY '......';
 9
10  CREATE EXISTING TABLE proxy_mss_source
11   AT 'mss.main.dbo.mss_source';
12
13  INSERT sa_target
14  SELECT *
15   FROM proxy_mss_source;
16
17  -- 11.0.1...
18  1925469 row(s) inserted
19  Execution time: 4229.875 seconds
20
21  -- Innsbruck...
22  1925469 row(s) inserted
23  Execution time: 686.859 seconds
```

FEATURE NUMBER 7: IMPROVED SUPPORT FOR DAFFYSQL SYNTAX

If I told you that RowGenerator.row_num contains the values 1 through 255, what would you say this query returned?

```
    SELECT  row_num
      FROM  RowGenerator
ORDER BY  row_num

LIMIT 100, 10;
```

Give up? OK, how about this one?

```
    SELECT  row_num
      FROM  RowGenerator
ORDER BY  row_num
LIMIT 10 OFFSET 100;
```

Still stumped? If I told you they both returned exactly the same result set as the following query, what would you say?

```
   SELECT  TOP 10 START AT 101
           row_num
     FROM  RowGenerator
ORDER BY row_num;

row_num
-------
 101
 102
 103
 104
 105
 106
 107
 108
 109
 110
```

Yes, the LIMIT clause is new to SQL Anywhere 12, exactly the same as TOP START AT except it uses zero as the starting point for numbering rows instead of 1.

An "offset", get it?

As in "Here's ten dollars, let me count it for you: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9."

Why implement LIMIT? And why include it in a list of cool features?

Because there are a lot of MySQL users out there who don't have TOP START AT, and they've written zillions of queries using LIMIT, and they'd like to migrate their apps to SQL Anywhere without rewriting everything. And PostgreSQL users too... welcome aboard!

Migrating to SQL Anywhere is definitely cool.

Oh, and here's a tip: You can't code LIMIT until you set this option, but it's a one-time thing:

SET OPTION reserved_keywords = 'LIMIT';

FEATURE NUMBER 8: DECLARE WITH DEFAULT

Sometimes it's the little things that count, like specifying DEFAULT values for local variables inside stored procedures, triggers and other blocks of code.

```
BEGIN
DECLARE @x INTEGER DEFAULT 1;
DECLARE @y TIMESTAMP DEFAULT CURRENT TIMESTAMP;
DECLARE @z VARCHAR ( 100 ) DEFAULT 'Hello';
SELECT @x, @y, @z;
END;
@x @y @z
-- ---------------------- -----
 1 2010-04-03 15:26:35.781 Hello
```

DEFAULT values reduce the likelihood of coding errors like forgetting to SET an initial value for a variable, or putting the SET in the wrong place, or accidentally moving the SET to the wrong place when rearranging code... NULL values can ruin your day.

What's next? NOT NULL constraints for local variables? Yes, I know, that's not a "little" thing, a constraint must be checked every time the value changes... but one can hope :)

FEATURE NUMBER 9: HOST AND PORT NO LONGER JUST HINTS

If you've ever connected to the wrong server and solved the problem by coding DOBROADCAST = NONE, if you've ever heard someone (like me) ranting "Always code DOBROAD = NONE!" then you'll appreciate this change: The HOST and PORT network protocol options in LINKS=TCPIP(HOST=xxx;PORT=yyy) are no longer mere "hints" to the client server interface software.

As of SQL Anywhere 12, if you code HOST then only that host will be used to make the connection.

And if you code PORT then only that port will be used to make the connection.

Kinda makes sense, right?

Remember this moment, you can tell your children about it, "Why, I remember when HOST and PORT didn't really mean anything, they were just hints! You kids have it so easy today! You never connect to the wrong database."

Tip: Don't confuse the HOST "network protocol option" discussed here with the brand-new HOST "connection parameter" which is a completely different beast.

FEATURE NUMBER 10: MORE THROUGHPUT, LESS DRAMA

Here's a scenario that happens quite often in the real world: Many end users, both customers and employees, are busy inserting rows by the thousands into detail tables. These detail tables are related to various master tables via foreign key relationships: the master tables are the parents, the detail tables are the children.

Along comes an administrator who updates a row in a master table... or worse, updates all the rows in a master table... with some change that shouldn't affect the work going on with the detail tables, such as a change to a column that is not part of the primary key.

The trouble is, it does affect the other work, at least it does in earlier versions of SQL Anywhere. Nobody else can insert any detail rows that will be children of the master rows the administrator has updated, not until the administer commits the update. In database-speak, the other connections are blocked by locks and their inserts sit waiting until the locks are released. Which, in a badly designed system with long running transactions, can be quite a while.

Figure 27 shows a simple master-detail parent-child relationship between two tables, plus an uncommitted non-key update to the parent table.

Figure 27: UPDATE non-key parent column in ISQL session 1

```
 1  CREATE TABLE parent (
 2   parent_pkey INTEGER NOT NULL,
 3   data INTEGER NOT NULL,
 4   PRIMARY KEY ( parent_pkey ) );
 5
 6  CREATE TABLE child (
 7   parent_pkey INTEGER NOT NULL,
 8   child_pkey INTEGER NOT NULL,
 9   data INTEGER NOT NULL,
10   PRIMARY KEY ( parent_pkey, child_pkey ),
11   FOREIGN KEY ( parent_pkey )
12   REFERENCES parent ( parent_pkey ) );
13
14  INSERT parent VALUES ( 1, 1 );
15  COMMIT;
16
17  UPDATE parent SET data = data + 1 WHERE parent_pkey = 1;
18
19  -- No COMMIT yet; display the locks...
20
21  SELECT STRING (
22   conn_name, ' - ',
23   table_name, ' - ',
24   IF index_id IS NULL
25   THEN 'null'
26   ELSE STRING ( index_id )
27   ENDIF,
28   ' - ',
29   lock_class, ' - ',
30   lock_duration, ' - ',
31   lock_type, ' - ',
32   IF row_identifier IS NULL
33   THEN 'null'
34   ELSE STRING ( row_identifier )
35   ENDIF )
36   AS "conn - table - index - class - duration - type - row"
37   FROM sa_locks()
38   ORDER BY 1;
```

Line 17 in Figure 27 shows an uncommitted update that doesn't affect the primary key, just a non-key column.

The SELECT at line 21 displays all the locks that exist right after the update; the output from this query will be shown later.

Figure 28 shows a child (detail) table insert to be run on a different connection; e.g., in a separate ISQL session.

Figure 28: Try to INSERT child row in ISQL session 2

```
 1  SET TEMPORARY OPTION BLOCKING = 'OFF';
 2
 3  BEGIN -- a block to catch EXCEPTION
 4
 5  DECLARE @sqlcode INTEGER;
 6  DECLARE @errormsg VARCHAR ( 32767 );
 7
 8  INSERT child VALUES ( 1, 1, 1 ); -- under parent row being updated
 9
10  MESSAGE 'INSERT child worked' TO CLIENT;
11
12  EXCEPTION
13   WHEN OTHERS THEN
14   SELECT SQLCODE, ERRORMSG()
15   INTO @sqlcode, @errormsg;
16   MESSAGE STRING (
17   'INSERT child failed: ', @sqlcode, ' - ', @errormsg )
18   TO CLIENT;
19  END;
```

Line 1 in Figure 28 turns off SQL Anywhere's normal behavior which is to wait instead of raising an exception when the connection is blocked by a lock held by some other connection. For this demonstration we want to see an error, not wait politely.

Line 8 is an attempt to insert a child row that is directly dependent on the parent row that was updated by the earlier code in Figure 27. If the insert works, the MESSAGE at line 10 will be displayed.

If the insert fails, the general-purpose EXCEPTION handler at line 12 will get control instead, and the MESSAGE at line 16 will appear... and that is indeed what happens when you run this test using SQL Anywhere 11.0.1:

```
INSERT child failed: -210 - User 'DBA' has the row in 'child' locked
```

Not only is session 2 prevented from performing work that has nothing whatsoever to do with the pending transaction in session 1, but the error message is misleading: There are no rows yet in the child table, how can one be locked?

No, don't try to answer that... instead, have a look at what happens with SQL Anywhere 12:

```
INSERT child worked
```

Much better... no confusing error message when BLOCKING is off, and no waiting when BLOCKING is on (the default). Users are allowed to proceed when their work doesn't interfere with one another.

Figure 27 shows the database locks that exist after the parent row was updated in Figure 27, before the code in Figure 28 is run.

Figure 29: Locks held after UPDATE in ISQL session 1

1	-- SQL Anywhere 11.0.1
2	
3	conn - table - index - class - duration - type - row
4	ddd11-1 - parent - null - Row - Transaction - Intent - 33619968
5	ddd11-1 - parent - null - Row - Transaction - Write - 33619968
6	ddd11-1 - parent - null - Schema - Transaction - Shared - null
7	ddd11-1 - parent - null - Table - Transaction - Intent - null
8	
9	-- SQL Anywhere 12
10	
11	conn - table - index - class - duration - type - row
12	ddd12-1 - parent - null - Row - Transaction - Intent - 36700160
13	ddd12-1 - parent - null - Row - Transaction - WriteNonPK - 36700160
14	ddd12-1 - parent - null - Schema - Transaction - Shared - null
15	ddd12-1 - parent - null - Table - Transaction - Intent - null

Lines 5 and 13 in Figure 29 show the difference between SQL Anywhere versions 11 and 12: the "Write" lock has become a "WriteNonPK" lock, a slightly-less-than-full row lock, one that isn't quite such a drama queen when it comes to blocking other connections.

CONCLUSION

Folks will probably remember SQL Anywhere 12 as being "spatial plus some other stuff". That's both fair and unfair: It's a fair comment because the new spatial support is really really really really really huge; this article only scratches the surface of the "cool demos" you can run let alone the real world business problems you can solve. At the same time, it's an unfair comment because there are a lot of other significant new features in SQL Anywhere 12. I've only mentioned a few of them here, along with a couple of not-so-significant-but-nonetheless-cool new features. This article compounds that unfairness by leaving out all mention of MobiLink and UltraLite: they really deserve their own Cool New Features articles.

ABOUT THE AUTHOR

Breck Carter is Principal Consultant at RisingRoad Professional Services, the creator of the Foxhound database monitor and schema troubleshooting utility for SQL Anywhere. His blog is at SQLAnywhere.blogspot.com and he manages the question and answer website at SQLA.stackexchange.com. Breck can be reached at breck.carter@gmail.com, and Foxhound can be found at www.risingroad.com/foxhound.

The content in this whitepaper is based on SQL Anywhere Innsbruck software only. Necessary screenshots and information will be updated upon the release of SQL Anywhere 12.

Sybase, Inc.
Worldwide Headquarters
One Sybase Drive
Dublin, CA 94568-7902
U.S.A
1 800 8 sybase

www.sybase.com

SYBASE®