Top 10 Cool New Features In SQL Anywhere® 11

BY BRECK CARTER

SYBASE® iAnywhere®

OVERVIEW

SQL Anywhere 11 has a surprisingly large number of new features, especially when you consider the huge number of features introduced only two years ago in SQL Anywhere 10. This article presents a Top 10 list of new features in SQL Anywhere 11 that developers are likely to find most interesting. Some of the features that didn't make the Top 10 list are described as well, including features that might be important to management but aren't necessarily cool as far as developers are concerned. Also included is a short list of features that are more weird than wonderful.

Once again it's time for the SQL Anywhere Oscars, when folks tune in to see which of their favorite features got picked, and which ones got left out. And to see some weird stuff... it's all here, the Top 10 list of cool new features in SQL Anywhere 11.

THE TROUBLE WITH 11 IS 10

When SQL Anywhere 10 was released two years ago, I wrote that the number "10" was a big problem in itself: it was difficult to choose which ten out of the many dozens of cool new features to talk about.

With SQL Anywhere 11, the number "10" is still giving me grief, as in "I'm still trying to learn about version 10, and here comes 11!" I agree with this request posted on the version 11 beta forum: "Add more hours to my day" to spend figuring out all the new stuff.

Here's an example of how fast SQL Anywhere is changing: it's only been a short while since I first used OPENXML to parse and load giant XML files directly into SQL Anywhere tables, and that feature's been around since 9.0.1. As soon as I posted a "how to use OPENXML" article in April, it became the Number One most popular page on my SQL Anywhere blog, and it's stayed that way ever since.

Here's another example: using HTTP sessions to maintain state with SQL Anywhere's built-in web server was introduced in version 10, and it was just the other week that I got around to working with it. No, it wasn't on my Top 10 list for version 10, big mistake... it's really cool!

Having said all that, version 11 is not quite as ambitious as 10, so picking the Top 10 was not so hard. But first, it's time for some people to take offense when they read about some of their favorite features that got left out.

TIME TO TAKE OFFENSE

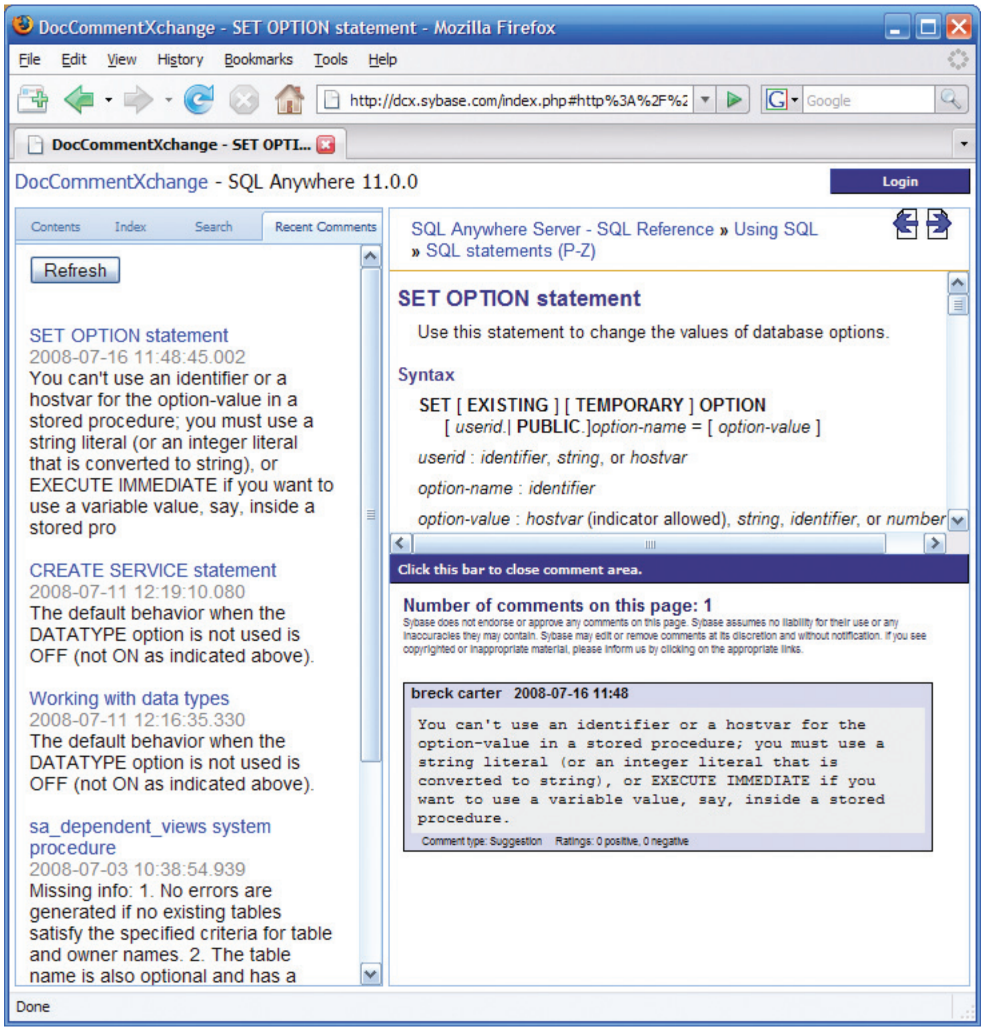
Here are a few of the new features in SQL Anywhere 11 that didn't make the Top 10 list.

Support for the .NET Entity Framework and LINQ is a big, important new feature, yes, but not cool, at least not to me. If you are going to use a relational database, you should learn and use the language of relational databases... which is SQL, not some obfuscating wrapper like LINQ. Don't try to tell me that this is a reasonable way to express a simple WHERE clause: Customer customer = this._db.Customers.Where("it.ID = @ID", new ObjectParameter("ID", id)).First();

Full text searching and regular expressions are two new features that you can use instead of LIKE predicates. The first one's very cool (full text search) and it's in the Top 10 list, but not regular expressions. It's going to be a very long time before I learn what "positive lookbehind zero-width assertion" means, or code something like '(?<=new\\s)york' in an otherwise readable SELECT. If you already know about regular expressions, you might be happy, but maybe not if you grew up worrying about maintainability.

Here's one I hope I'll regret leaving out: DCX, which stands for "DocCommentXchange", which is the windy name for hypertext links from the Help to a website that lets you read other people's comments on individual topics... and enter your own. The DCX interface looks very much like the Help, so when you click on a link you'll think you're looking at the same thing (see Figure 1): same topics, laid out the same way, same Contents, Index and Search tabs. What's new is the comments area in each topic, which shows you what other people have said and which lets you login and add your own. Also new is the Recent Comments tab on the left. Today, the dcx.sybase.com website runs on a SQL Anywhere 10 database. Soon, it will be running on SQL Anywhere 11, using the built-in HTTP server instead of Apache, and using PHP stored procedures instead of a PHP client server application... maybe it will even use full text search instead of LIKE. I hope they'll get all that done, and that's when I'll start wishing I'd included DCX in the Top 10.

FIGURE 1:
DocCommentXchange
at dcx.sybase.com


DocCommentXchange - SET OPTION statement - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://dcx.sybase.com/index.php#http%3A%2F%2
Google
DocCommentXchange - SET OPTI...
DocCommentXchange - SQL Anywhere 11.0.0
Login
Contents Index Search Recent Comments
Refresh
SQL Anywhere Server - SQL Reference » Using SQL » SQL statements (P-Z)
Click this bar to close comment area.
Done

SET OPTION statement
2008-07-16 11:48:45.002
You can't use an identifier or a hostvar for the option-value in a stored procedure; you must use a string literal (or an integer literal that is converted to string), or EXECUTE IMMEDIATE if you want to use a variable value, say, inside a stored pro

CREATE SERVICE statement
2008-07-11 12:19:10.080
The default behavior when the DATATYPE option is not used is OFF (not ON as indicated above).

Working with data types
2008-07-11 12:16:35.330
The default behavior when the DATATYPE option is not used is OFF (not ON as indicated above).

sa_dependent_views system procedure
2008-07-03 10:38:54.939
Missing info: 1. No errors are generated if no existing tables satisfy the specified criteria for table and owner names. 2. The table name is also optional and has a

SET OPTION statement

Use this statement to change the values of database options.

Syntax

SET [EXISTING] [TEMPORARY] OPTION
[*userid*.| PUBLIC.]*option-name* = [*option-value*]

userid : *identifier*, *string*, or *hostvar*

option-name : *identifier*

option-value : *hostvar* (indicator allowed), *string*, *identifier*, or *number*

Number of comments on this page: 1

Sybase does not endorse or approve any comments on this page. Sybase assumes no liability for their use or any inaccuracies they may contain. Sybase may edit or remove comments at its discretion and without notification. If you see copyrighted or inappropriate material, please inform us by clicking on the appropriate links.

breck carter 2008-07-16 11:48

```
You can't use an identifier or a hostvar for the
option-value in a stored procedure; you must use a
string literal (or an integer literal that is
converted to string), or EXECUTE IMMEDIATE if you
want to use a variable value, say, inside a stored
procedure.
```

Comment type: Suggestion Ratings: 0 positive, 0 negative

Here's another tough choice: Background synchronization support in UltraLite. That means when you're deploying a sackload of code to a handheld device, you don't have to jump through hoops to design an application that keeps your users from messing things up by continuing to work while a MobiLink synchronization is in progress. You can now start synchronizations on a second thread and UltraLite will only upload changes that have already been committed; any changes made after that will wait until the next sync. The reason this is cool? It's the Number One reason for moving to SQL Anywhere 11, for my very first client who's making the move. Not enough to make the Top 10 list, but enough to get mentioned here.

Here are a couple of very important under-the-hood features: Index-only retrieval, where SQL Anywhere won't even look at the table if the data needed to satisfy a query is found in an index, and multiple index scan where up to 4 different indexes on the same table may be used to satisfy a query instead of the current limit of one index per table. Some queries will run much faster because of these features, that's why they're mentioned here. They're under the hood, that's why they're not in the Top 10 Cool list... it's like when folks are shopping for a minivan, they want to see the cupholders, not hear about the camshafts.

MySQL can be used for the consolidated database in a MobiLink synchronization setup. There, I said it, MobiLink now works with MySQL. I really don't understand why anyone uses MySQL, it's an administrative nightmare of settings and options and pitfalls and confusion, and it's NOT free, not when you count support. But, it's popular. And now you can use SQL Anywhere in a MySQL shop to implement self-administering remote databases that synchronize via MobiLink with your legacy MySQL database. Speaking of legacy, MobiLink now also works with DB2 on the mainframe. Not cool, unless it's a way for you to get SQL Anywhere in the door, then it's very cool.

THE WEIRD AND WHIMSICAL IN SQL ANYWHERE 11

First up in the whimsy category: The SQL Anywhere executables folder on Windows is now called "Bin32", not "win32". That means if you have batch files containing paths like "%SQLANY10%\win32\..." you're going to have to change them in two places: "%SQLANY11%\Bin32\...". Someone told me Microsoft's to blame, but I find it hard to reconcile whimsy with The Borg.

The "Printable Books" version of the Help is now installed by default. That means if you don't want those giant PDFs installed you'll have to choose the "Custom" option when installing the documentation. The regular compiled HTML version of the Help is still there, still as excellent as always: all the manuals in one place, with full search capability... who needs paper? Definitely weird.

The whole business of writing stored procedures in languages other than SQL has been vastly expanded and improved, that's one of the Top 10 Cool New Features. What's weird is that if your choice of language happens to be C or C++, your heart will sink when you read this line: "It should also be noted that the compiled native function must use the native function call API." That API has been around for a long time, and it's one of the funkiest interfaces in the known coding universe... it's really too bad we still have to use it. But only for C and C++, not Perl or PHP or Visual Basic or any of the other languages. And at least the API hasn't gotten any worse.

INSTEAD OF triggers is a recent cool new feature; what's weird is that the engineers haven't take the step to also give us triggers on system catalog tables. More and more sophisticated applications are being written that dynamically manipulate schema objects, e.g., CREATE TABLE, and the ability to dynamically capture and track those changes would be welcome. Even if the code inside system catalog triggers could only manipulate user tables, not issue further schema changes, they would satisfy 80% of the need or more. I'm betting this will happen before the funky native function call API is improved.

TOP 10 COOL NEW FEATURES

The order's not important... it's hard enough picking ten, there's no point in trying to rank them.

1. UNLOAD to a Variable

The LOAD and UNLOAD statements now work with variables as well as files. I don't mean just using a variable to hold a file specification, I mean loading and unloading data stored in variables as well as files. You can UNLOAD all the rows and columns of a table into a single string variable, and the reverse: LOAD a table from a string:

```
BEGIN
   DECLARE s LONG VARCHAR;
   UNLOAD TABLE t1 TO s;
   LOAD TABLE t2 FROM s;
END;
```

Here are some advantages of using variables instead of files:

- Improved concurrency because variables are local to connections whereas files are global.
- Improved flexibility because SQL Anywhere has many powerful functions for manipulating strings but not many functions for fiddling with files.
- Improved speed because there's no file I/O involved... but don't get carried away and try to use UNLOAD LOAD as a substitute for INSERT SELECT, it's not **that** fast.

And yes, it works with UNLOAD SELECT as well as UNLOAD TABLE, so you can write a query to get complete control over what is stored in the variable.

This feature is personally embarrassing to me. For years I've been asking for it, and when I first got my hands on it I couldn't figure out what to do with it! The reason was that over the intervening years other features had became available to solve my problems, the most important being the enhanced LIST aggregate function. Using the LIST and STRING functions together with LIST's delimiter and ORDER BY clauses makes it possible to perform set-oriented string manipulation. For example, you can join several tables and turn the data a formatted HTML web page with a single SELECT.

Look, I'm not the sharpest knife in the drawer. Just because someone gives me a new feature doesn't mean that right away I'm going to know what it's good for. That was true with LIST, and it's true with UNLOAD to a variable. But once someone shows me, woohoo! I'm off to the races! That's what happened when Ivan Bowman showed me how to combine this new feature with the next one in the list.

2. FROM OPENSTRING

The new OPENSTRING clause lets you name a file or variable in the FROM clause of a SELECT and have SQL Anywhere treat the data inside that file or variable as a set of rows and columns. That's a little bit like using a proxy table to treat a file like a table via ODBC remote access middleware, but it's way easier and more flexible, not to mention that OPENSTRING lets you use variables as well as files.

This feature is also embarrassing to me. At first I thought it was one of those funky XML things, and the word "open" is really off-putting these days with the way it's used and over-used for everything. So I skipped right over OPENSTRING thinking "yeah, right, any day now we're going to start saying OpenCorpse instead of roadkill."

Then Ivan Bowman of Wikipedia "IvanAnywhere" fame changed my mind with this example using OPENSTRING together with UNLOAD SELECT into a variable:

"One neat use case I've wondered about is the following (I'm using SYSDOMAIN here just for the sake of an example):

```
CREATE VARIABLE @var LONG VARCHAR;

UNLOAD
   SELECT *
     FROM SYSDOMAIN
INTO VARIABLE @var;

CREATE VIEW V AS
   SELECT *
     FROM OPENSTRING ( VALUE @var ) WITH
              ( domain_id   SMALLINT,
                domain_num  VARCHAR ( 128 ),
                type_id     SMALLINT,
                "precision" SMALLINT )
           AS T;
```

What the above code does is copy a result set into a connection-level variable @var, then uses a view V to refer to that result set. You could create the value for @var in a connect procedure.

This could be a good way to manage local option settings or connection-local caches of hot data (avoiding contention on hot rows). I'd want to be careful about how that type of thing is used, but it is a really neat tool to have in your back pocket. You could even create INSTEAD OF triggers on the view to really make it look more like a table.

The OPENSTRING feature also seems like a perfect fit for some cases of dynamic IN lists, where the IN list is specified as a string. It even allows using a multi-column IN list, something that is not directly supported in SQL Anywhere.

OPENSTRING may also have some interesting uses when combined with LIST. It could appear as a lateral join, decoding a single column value into a result set. I've used sa_splitlist() for a number of interesting solutions in the past, and OPENSTRING covers all of those and more because it decodes rows, not just columns, it is much faster, and it offers more parsing options for handling more complex data.

I'm curious to see what uses these new features will find in the real world."

— Ivan T. Bowman, Distinguished Engineer, Sybase iAnywhere

Amen, Ivan, on that last point... it's hard to predict what folks will do with new features. Sometimes, they take off, like the LIST function, sometimes not so much, like the syntactic nightmare that is the ANSI standard recursive union. I'm thinking OPENSTRING could be one of the popular ones, and UNLOAD into a variable as well.

Ivan mentioned another recent feature, one that didn't make it on to any Top 10 list because it was introduced without any fanfare in version 10.0.1: INSTEAD OF triggers let you write triggers on tables and views that execute instead of the triggering action. In other words, if an INSTEAD OF INSERT ON T trigger exists, when an INSERT T statement executes it fires the trigger but it does not insert the row. Any action you want as a result of the INSERT T has to be coded inside the trigger.

The cool part is it lets you write triggers on views, even views that are not themselves updatable. Whether or not the view is updatable doesn't matter because the triggering INSERT, UPDATE or DELETE isn't going to do an insert, update or delete itself if a corresponding INSTEAD OF trigger exists, and you are free to code whatever you want inside the trigger (except for a COMMIT, of course, that restriction stands).

3. Immediate Materialized Views

Here's an example of a materialized view used to create a list of employees by department:

```
CREATE MATERIALIZED VIEW DeptList AS
SELECT Departments.DepartmentID,
       Employees.EmployeeID,
       Departments.DepartmentName,
       Employees.GivenName,
       Employees.Surname,
       Employees.Phone
  FROM Departments
       INNER JOIN Employees
          ON Employees.DepartmentID = Departments.DepartmentID;
```

All materialized views have to be initialized before they can be used:

```
REFRESH MATERIALIZED VIEW DeptList;
```

With an ordinary (manual) materialized view, if you change the underlying table but don't do another REFRESH you'll see old data:

```
UPDATE Employees SET DepartmentID = 400 WHERE EmployeeID = 1336;

SELECT *
  FROM DeptList
 ORDER BY DepartmentName,
       Surname,
       GivenName;

DepartmentID  EmployeeID  DepartmentName  GivenName  Surname  Phone
============  ==========  ==============  =========  =======  ==========
300            1336         Finance          Janet      Bigelow  6175551493
```

Starting with SQL Anywhere 11, you can change the view so that any updates to the underlying tables are immediately applied to the data stored in the materialized view; here are the steps to do that:

```
TRUNCATE TABLE DeptList;

CREATE UNIQUE INDEX xid ON DeptList (
   DepartmentID,
   EmployeeID );

ALTER MATERIALIZED VIEW DeptList IMMEDIATE REFRESH;

REFRESH MATERIALIZED VIEW DeptList;
```

The TRUNCATE TABLE is necessary if the manual view has already been populated with data: the view has to be empty before changing it to immediate. The CREATE UNIQUE INDEX is another requirement for immediate views: SQL Anywhere has to be able to find the rows when it needs to update them. The ALTER changes the view's refresh mode to IMMEDIATE, and a one-time REFRESH is needed to initially populate the view... yes, that's a long list of steps, but you only have to do them once.

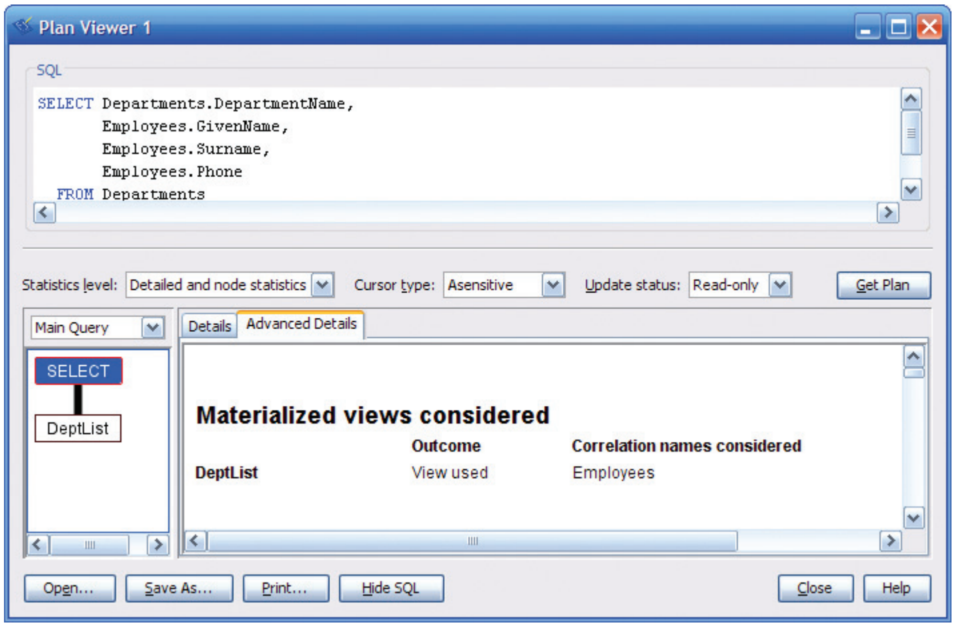
Now, if you update a base table and query the view, the new data shows up right away without the need for another explicit REFRESH:

```
UPDATE Employees SET DepartmentID = 400 WHERE EmployeeID = 1336;

SELECT *
  FROM DeptList
 ORDER BY DepartmentName,
        Surname,
        GivenName;

DepartmentID  EmployeeID  DepartmentName  GivenName  Surname  Phone
============  ==========  ==============  =========  =======  ==========
400             1336         Marketing        Janet      Bigelow  6175551493
```

Here's the real reason materialized views are cool: you don't have to know the view exists to get the benefits. Even though the following query uses the base tables, Figure 2 shows that SQL Anywhere used the DeptList view instead:

```
SELECT Departments.DepartmentName,
        Employees.GivenName,
        Employees.Surname,
        Employees.Phone
  FROM Departments
        INNER JOIN Employees
           ON Employees.DepartmentID = Departments.DepartmentID;
```

FIGURE 2: Materialized View Automatically Used



```
SELECT Departments.DepartmentName,
       Employees.GivenName,
       Employees.Surname,
       Employees.Phone
  FROM Departments
```

Statistics level: Detailed and node statistics Cursor type: Asensitive Update status: Read-only

Materialized views considered

	Outcome	Correlation names considered
DeptList	View used	Employees

4. Backward Compatibility is Back

If you have a database created with version 10 software, you can run it with SQL Anywhere 11 without having to rebuild. That's important with very large databases: you can start testing with version 11 without committing to a time-consuming no-going-backward rebuild.

When version 10 was introduced, it came as a surprise that rebuilding was a prerequisite for any database created with earlier versions. With version 9 software, folks were used to the fact that it could be used with older database files, even ones created with version 5.5. Version 10 broke that compatibility path, and it was done for a very good reason: to jettison a lot of legacy code from the database engine, eliminating the maintenance effort that code required, and freeing that time to be spent in areas that matter a lot more... like cool new features.

With version 11 a new compatibility path has started: no rebuild is required when going from 10 to 11. Earlier than that, say, if you're moving from 9 to 11, you still have to rebuild.

And here's an important point: If performance is critical to you, it's still a good idea to rebuild as soon as you commit to moving from 10 to 11. The only way to take advantage of improvements in the new database file format introduced with version 11 is to do a full unload and reload.

5. Stored Procedures in Perl, PHP, C#, VB

With Version 11 of SQL Anywhere you can now write stored procedures in nine different languages: Perl, PHP, C# and Visual Basic have been added, and support continues for Java, C, C++ and two flavors of SQL, ANSI/Watcom and Transact. Actually, the number might be higher than 9; for example, a .Net 2.0 CLR version of Fortran should work, as long as you understand that "should work" is how speakers of Tech Support Language say "you might be the first one to actually try it."

The details of stored procedure support vary for different languages; for example, VB can return result sets but Perl can only return LONG VARCHAR strings.

It is also important to note that procedures written in any of the non-SQL languages now run in executable environments that are completely separate from the database server itself. That means if your procedure crashes, it won't take the database down with it. And once again, details vary; for example, you get one external Perl environment set up for each connection, but with VB there is only one external environment started for each database. Figure 3 gives an overview of the differences in support among the various languages.

FIGURE 3: Differences In Language Support

	1 external environment per...	Procedure can return a...
Perl	connection	LONG VARCHAR
PHP	connection	LONG VARCHAR
C#, VB	database	result set
Java	database	result set
C, C++	connection	result set
SQL	n/a	result set

The ability to safely call procedures written in different languages is a big deal, almost worth its own entry in the Top 10 list. Previously, C and C++ procedures ran inside the database server, with catastrophic results if your code went crazy: not just one failed user connection, but a crashed server affecting everyone.

Another big deal is the ability to use the current database connection inside a procedure written in one of the non-SQL languages. In other words, you don't have to start a new connection to pass a SQL statement back to the database that called the procedure. In previous versions, you couldn't issue any SQL statements at all from inside a C or C++ stored procedure, even if you were willing to start a new connection, and now you have same-connection database access with all the languages.

This is cool stuff, I've got a sackload of code I'd like to embed in databases without having to jump through hoops to pass data back and forth. Here's one example: embedding the API for a file comparison utility to work on documents stored as blobs in the database, so SQL Anywhere's built-in HTTP server can be used to display the comparison results via web service calls from the browser.

6. In-Memory Server Mode

The in-memory server option is an entirely new mode of operation for improving the speed of inserts and updates at the cost of durability. You can create and load a database file in "SQL Anywhere Classic" mode and then run it in one of the two in-memory modes: Never Write or Checkpoint Only. You can switch between in-memory modes and back to Classic mode by stopping and restarting the engine... it's the same software, just using different command-line parameters.

And before anyone gets all upset, SQL Anywhere 11 isn't introducing some kind of cheesy non-transactional auto-commit storage engine like MyISAM in MySQL. SQL Anywhere is still a fully-relational database management system, fully transactional, almost fully ACId even when running in-memory mode.

ACId means Atomic, Consistent, Isolated and durable, the Four Horsemen Of The Transaction. Durability means a COMMIT is permanent. I'm using a lowercase "d" in ACId because the Checkpoint Only flavor of in-memory mode preserves all committed transactions up to the last checkpoint, but doesn't allow forward recovery beyond that point. And nothing's preserved, no recovery at all for the Never Write flavor... the name kinda gives that away.

In-memory mode is most effective when your database RAM cache is large enough to hold the entire database, when you are doing lots of inserts and updates and commits, and when you're willing to sacrifice some safety to get a lot of speed. I'm thinking of continuous monitoring as a candidate application, where huge amounts of data are received in short periods of time, and where it's more important to keep up with the input than to guarantee that 100% of the data is recovered after a crash.

Both in-memory modes allow updates to the data in the cache; in other words, "Never Write" doesn't mean "read only". With Checkpoint Only mode, updated data is written to the database file on a CHECKPOINT. With Never Write, it's up to you to save your data if permanence is important; three speedy ways to do that are database backup, the database unload utility and individual UNLOAD TABLE statements. Figure 4 shows the details of how Never Write and Checkpoint Only stack up against one another, and against Classic mode.

FIGURE 4: Comparing Modes of Server Operation

	-im nw	-im c	
	Never	Checkpoint	
	Write	Only	Classic
Transactional	yes	yes	yes
.db file updated	no	yes	yes
Transaction log	no	no	yes
Temporary file	no	no	yes
Checkpoint log	no	yes	yes
Checkpoints	no	yes	yes
Dirty page flushes	no	yes(a)	yes
Unlimited growth	no	yes(b)	yes
Recoverable	no	yes(c)	yes

```
  (a) Dirty pages are flushed only on checkpoint.

  (b) Without a temporary file, checkpoints are the only way to keep the
cache from being exhausted.

  (c) After a crash, a restart will perform the usual automatic recovery to
the last good checkpoint, but without a transaction log, it is not possible
to run a forward recovery from the last good checkpoint to the last
committed transaction.
```

7. MERGE Statement

The new MERGE statement brings the power of set-oriented processing to bear on INSERT, UPDATE and DELETE logic when you need to copy data from one table to another. Here is a simple example that starts with an input table containing a set of transactions:

```
CREATE TABLE transaction (
   part_number  INTEGER NOT NULL PRIMARY KEY,
   action        VARCHAR ( 6 ) NOT NULL,
   description  LONG VARCHAR NULL,
   price          DECIMAL ( 11, 2 ) NULL );

INSERT transaction VALUES ( 1, 'Add', 'widget',   10.00 );
INSERT transaction VALUES ( 2, 'Add', 'gizmo',    20.00 );
INSERT transaction VALUES ( 3, 'Add', 'thingie', 30.00 );
INSERT transaction VALUES ( 4, 'Add', 'bouncer', 40.00 );
```

Here's the output table, plus the MERGE statement which turns the four input 'Add' transactions into INSERT operations:

```
CREATE TABLE part (
   part_number  INTEGER NOT NULL PRIMARY KEY,
   description  LONG VARCHAR NOT NULL,
   price         DECIMAL ( 11, 2 ) NOT NULL,
   exceptions   LONG VARCHAR NOT NULL DEFAULT '' );

MERGE INTO part USING transaction ON PRIMARY KEY
 WHEN NOT MATCHED AND transaction.action = 'Add'
    THEN INSERT ( part_number,
                  description,
                  price )
         VALUES ( transaction.part_number,
                  transaction.description,
                  transaction.price )
 WHEN MATCHED AND transaction.action = 'Change'
    THEN UPDATE SET part.description
                       = COALESCE ( transaction.description,
                                    part.description ),
                    part.price
                       = COALESCE ( transaction.price,
                                    part.price )
 WHEN MATCHED AND transaction.action = 'delete'
    THEN DELETE
 WHEN MATCHED AND transaction.action = 'Add'
    THEN UPDATE SET part.exceptions
                   = STRING ( part.exceptions,
                          'duplicate Add ignored. ' )
 WHEN NOT MATCHED AND transaction.action = 'Change'
    THEN INSERT VALUES ( transaction.part_number,
                         transaction.description,
                         transaction.price,
                         'Change accepted, new part inserted. ' )
 WHEN NOT MATCHED AND transaction.action = 'delete'
    THEN SKIP;
```

The MERGE statement works as follows:

- The INTO clause specifies the output or target table.
- The USING clause names the input or source table.
- The ON clause specifies how row matching is done, in this case using the primary keys for both tables.
- The first WHEN clause says what to do when an 'Add' transaction doesn't match any row in the part table: INSERT a new row.
- The second WHEN says what to do with a matching 'Change': UPDATE the description and/or price columns.
- The third WHEN turns a matching 'delete' transaction into a DELETE operation.
- The fourth WHEN records an error: 'duplicate Add ignored.'
- The fifth WHEN turns a mismatching 'Change' into an INSERT.
- The last WHEN simply ignores an attempt to 'delete' a part that isn't there.

In the first run of the MERGE, all it does is execute the first WHEN clause four times to fill the part table:

```
part_number  description  price  exceptions
===========  ===========  =====  ==========
1             widget       10.00
2             gizmo        20.00
3             thingie      30.00
4             bouncer      40.00
```

Here's where it gets interesting; six new transactions that exercise all the remaining WHEN clauses:

```
TRUNCATE TABLE transaction;
INSERT transaction VALUES ( 1, 'Change', NULL,       15.00 );
INSERT transaction VALUES ( 2, 'Change', 'gizmoid', NULL );
INSERT transaction VALUES ( 3, 'delete', NULL,       NULL );
INSERT transaction VALUES ( 4, 'Add',     'bouncer', 45.00 );
INSERT transaction VALUES ( 5, 'Change', 'gong',     55.00 );
INSERT transaction VALUES ( 6, 'delete', NULL,       NULL );
```

Here's what the part table looks like after running the MERGE again:

```
part_number  description  price  exceptions
===========  ===========  =====  ==========
1             widget       15.00
2             gizmoid      20.00
4             bouncer      40.00  Duplicate Add ignored.
5             gong         55.00  Change accepted, new part inserted.
```

There is a dark side to MERGE, however: it won't let you have multiple input rows updating one output row. That's not SQL Anywhere's fault, it's a Pointy-Haired Committee decision, and I predict it's going to be the Number One complaint about MERGE: "The ANSI SQL/2003 standard does not allow rows in target-object to be updated by more than one row in source-object during a merge operation." The Number Two request? Folks will ask for the ability to call a procedure from a WHEN clause.

8. CREATE EVENT ... TYPE DEADLOCK

A new EVENT type has been introduced for catching and reporting on cyclical deadlocks. Deadlocks sometimes occur on busy systems when two different connections each lock a row and then immediately ask for a lock on the row the other guy's got. Both connections get blocked, neither one can get what it wants, and neither one will ever proceed until some third party takes action to break the deadlock.

SQL Anywhere itself is that third party; it has always been able to detect deadlocks, and it instantly "resolves" them by picking one of the connections as the victim, issuing a ROLLBACK and returning an error message to that application.

So far, so good; one user wins, the other can try again, and the maintenance programmer can start looking for the error... and it almost always is a programming error, some flaw in the order in which SQL statements are executed. The problem is that the standard deadlock error message gives absolutely no clue about what tables or rows were involved in the deadlock, or what either connection was doing at the time. The new CREATE EVENT ... TYPE DEADLOCK statement fixes that by giving you the ability to record and report on all the pertinent details: the connection numbers and user ids for both connections, and the actual SQL statements that were executing at the time the deadlock occurred.

Here is an example of a deadlock event handler that captures the details and stores them in a table:

```
CREATE EVENT deadlock TYPE DEADLOCK HANDLER
BEGIN
   -- Copy fresh entries from sa_report_deadlocks()
   INSERT deadlock (
      snapshotId,
      snapshotAt,
      waiter,
      who,
      what,
      object_id,
      record_id,
      owner,
      is_victim,
      rollback_operation_count )
   SELECT *
     FROM sa_report_deadlocks()
    WHERE NOT EXISTS (
       SELECT * FROM deadlock
        WHERE deadlock.snapshotId
            = sa_report_deadlocks.snapshotId
          AND deadlock.snapshotAt
            = sa_report_deadlocks.snapshotAt );
   COMMIT;
   MESSAGE STRING ( 'dIAG ', CURRENT TIMESTAMP,
      ' deadlock' ) TO CONSOLE;
END;
```

The built-in sa_report_deadlocks() procedure isn't new, but it has been greatly improved for SQL Anywhere 11. It returns a result set containing at least two rows for each deadlock that has occurred since the server started: one for the victim, one for the other connection. The INSERT copies that data into a permanent user-defined table called "deadlock". The WHERE NOT EXISTS is needed to prevent old rows from being copied again when another deadlock happens.

Here's what the table looks like; it has one extra column "row_number" to act as an artificial primary key, plus columns for everything returned by sa_report_deadlocks():

```
CREATE TABLE deadlock (
   row_number      BIGINT NOT NULL
                       DEFAULT AUTOINCREMENT,
   snapshotId      BIGINT NOT NULL,
   snapshotAt      TIMESTAMP NOT NULL,
   waiter          INTEGER NOT NULL,
   who             VARCHAR ( 128 ) NOT NULL,
   what            LONG VARCHAR NOT NULL,
   object_id       UNSIGNED BIGINT NOT NULL,
   record_id       BIGINT NOT NULL,
   owner           INTEGER NOT NULL,
   is_victim       BIT NOT NULL,
   rollback_operation_count UNSIGNED INTEGER NOT NULL,
   PRIMARY KEY ( row_number ) );
```

In order to make this work, you have to tell the server to gather the report data:

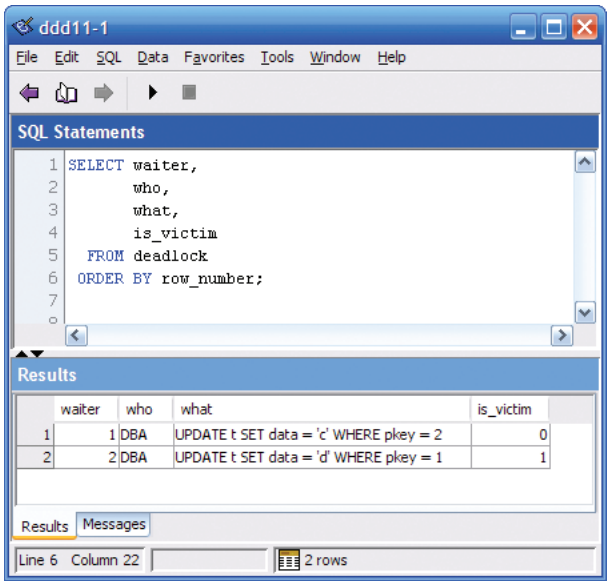
```
SET OPTION PUBLIC.LOG_DEADLOCKS = 'ON';
```

It also helps a lot if you enable the 'lastStatement' connection property so the report data will show the actual SQL statements that were blocked. The easiest way to do that is to specify the dbsrv11 -zl option when starting the server.

Figure 5 shows what the following query displayed after a deadlock; "waiter" is the connection number, "who" is the user name, "what" is the CONNECTION_PROPERTY ('lastStatement') and "is_victim" tells you which connection got the ROLLBACK:

```
SELECT waiter,
       who,
       what,
       is_victim
  FROM deadlock
 ORDER BY row_number;
```

FIGURE 5: Displaying Deadlock Details



```
SELECT waiter,
       who,
       what,
       is_victim
  FROM deadlock
 ORDER BY row_number;
```

	waiter	who	what	is_victim
1	1	DBA	UPDATE t SET data = 'c' WHERE pkey = 2	0
2	2	DBA	UPDATE t SET data = 'd' WHERE pkey = 1	1

Line 6 Column 22 2 rows

9. It’s The Little Things That Count

In every release of SQL Anywhere, many features are introduced with little or no fanfare because nobody thinks they are significant. Sometimes, one of these features becomes hugely popular and important, like the magic DEFAULT TIMESTAMP which works with UPDATE as well as INSERT. I call these features The Little Things That Count, and somewhere in following list is lurking at least one that deserves to be in the Top 10 Cool list, only we don’t know it yet:

- The function call PROPERTY (‘tCPIPAddresses') answers the question “have I connected to the right server?” by telling you what address the server’s listening on; e.g.: 192.168.1.51:49270.
- You can press Ctrl- to comment and uncomment blocks of code in dbisql.
- You can use the DEFAULT keyword in an UPDATE as well as INSERT; e.g., UPDATE t SET c = DEFAULT.
- ENDIF and END IF can be used interchangeably; no longer do you need to remember which one goes with the IF statement and which one with the IF expression.
- Line numbers are displayed next to your code in the dbisql “SQL Statements” frame.
- You can use the ampersand “&” as a line continuation character in @configuration files, which really helps when you’re setting up long parameter strings for High Availability and the built-in HTTP server.
- JSON or JavaScript Object Notation is now available for web service result sets, in addition to HTML, SOAP, XML and raw formats... maybe JSON will save us all from XML.
- The new PRIORITY option can be used by an application to increase or decrease the priority level at which its SQL requests are executed. For example, a high-speed OLTP connection can set its priority to 'Critical' while a long-running OLAP query can use 'Background'... or any one of the five other levels between those two extremes.
- Even if you don’t care about UNLOAD into a variable, one of the other enhancements to LOAD and UNLOAD is sure to make you happy: encryption, compression, using LOAD and UNLOAD in dbisql... how about optional transaction logging so you can use LOAD and UNLOAD in a mirrored database environment?
- Speaking of mirrored databases: now you can run read-only queries against the secondary server in a High Availability setup; now you can move your heavy OLAP sessions off your primary OLTP server and use the secondary server for something other than just failover.
- The NewPassword connection parameter lets the user specify a new password even if the current one’s expired.

Oh, and that business about expired passwords? There’s a whole mess of new features under the topic of “Login Policies”. It’s not in the Top 10 list because, let’s face it, security is not cool... but logging in without having to call the DBA, that’s priceless.

10. Full Text Search

If this was a "Top 1 List" then full text search would still be on it: it's the number one coolest feature in SQL Anywhere 11. With full text search you don't have to use a separate piece of software to implement fast searches across multiple LONG VARCHAR columns, and you don't have to store the data outside the database to do it. If you've ever run SELECT LIKE '%word%' queries against a large table, you know what "slow" means, and you're going to love full text search.

Here is an example of how it works, using a table that contains all 6.5 million entries downloaded from the English version of Wikipedia:

```
CREATE TABLE enwiki_entry ( -- 6,552,490 rows, 17.2G total
   page_number      BIGINT NOT NULL,
   from_line_number BIGINT NOT NULL,
   to_line_number   BIGINT NOT NULL,
   page_title       VARCHAR ( 1000 ) NOT NULL,
   page_id          VARCHAR ( 100 ) NOT NULL,
   page_text        LONG VARCHAR NOT NULL,
   PRIMARY KEY CLUSTERED ( page_number ) );
```

The first step is to define a text index on the columns to be searched:

```
CREATE TEXT INDEX tx_page_text
   ON enwiki_entry ( page_text )
   MANUAL REFRESH;
```

The second step is to build the text index, a process that can take quite a long time if the table is very large:

```
REFRESH TEXT INDEX tx_page_text
   ON enwiki_entry
   WITH EXCLUSIVE MODE
   FORCE BUILD;
```

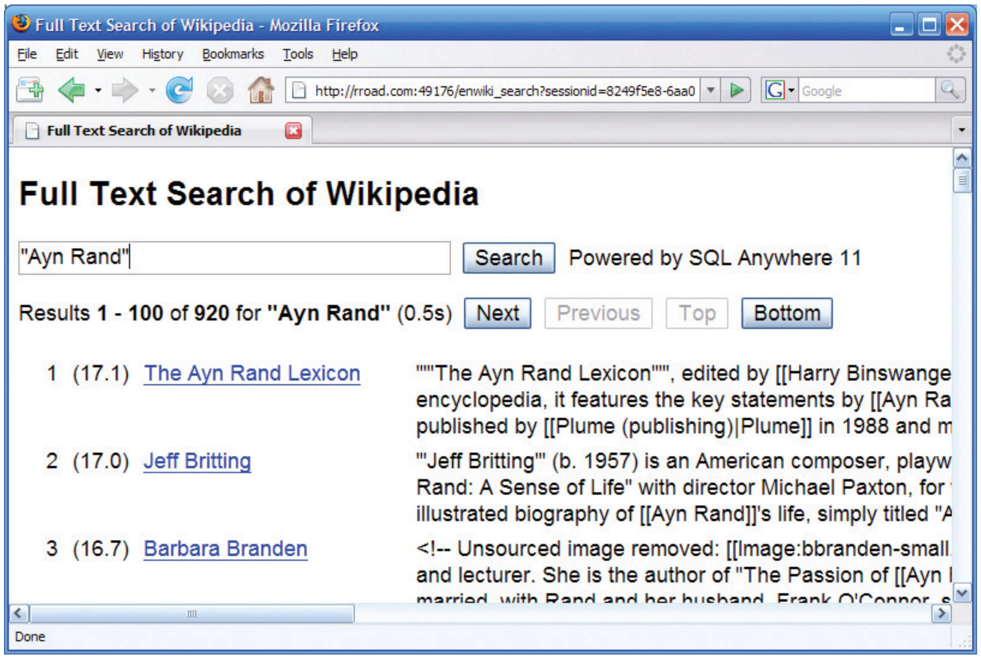
Once the index is built, however, queries that use the index are very fast. The following SELECT uses the new CONTAINS clause to find all the Wikipedia entries containing the exact phrase "Ayn Rand":

```
SELECT score,
       enwiki_entry.page_title,
       LEFT ( enwiki_entry.page_text, 500 ) AS excerpt
  FROM enwiki_entry
       CONTAINS ( enwiki_entry.page_text,
                  '"Ayn Rand"' )
 ORDER BY score DESC;
```

The CONTAINS clause applies the query string '"Ayn Rand"' to the enwiki_entry.page_text column using the full text index previously defined on that column, thus limiting the result set to rows that match. The CONTAINS clause also returns an implicit "score" column which measures how closely each row matches the query string. The ORDER BY clause uses that column to sort the best matches to the top, and Figure 6 shows the results in a browser display produced by a SQL Anywhere 11 web service.

FIGURE 6:

Full Text Search of Wikipedia



Full Text Search of Wikipedia - Mozilla Firefox

http://rroad.com:49176/enwiki_search?sessionid=8249f5e8-6aa0

Full Text Search of Wikipedia

"Ayn Rand" Search Powered by SQL Anywhere 11

Results **1 - 100** of **920** for **"Ayn Rand"** (0.5s) Next Previous Top Bottom

1 (17.1) The Ayn Rand Lexicon """The Ayn Rand Lexicon""", edited by [[Harry Binswange
encyclopedia, it features the key statements by [[Ayn Ra
published by [[Plume (publishing)|Plume]] in 1988 and m

2 (17.0) Jeff Britting "'Jeff Britting'" (b. 1957) is an American composer, playw
Rand: A Sense of Life" with director Michael Paxton, for
illustrated biography of [[Ayn Rand]]'s life, simply titled "A

3 (16.7) Barbara Branden <!-- Unsourced image removed: [[Image:bbranden-small.
and lecturer. She is the author of "The Passion of [[Ayn I
married with Rand and her husband Frank O'Connor s

Done

Here's a tip: Don't do like I did and leave out an important column from the CREATE TEXT INDEX statement. In Figure 6 the main Wikipedia entry entitled "Ayn Rand" doesn't appear on the first page, but it should, and it would have if I hadn't forgotten to include the enwiki_entry.page_title column in the index. When a full text index specifies multiple columns the CONTAINS clause calculates a score that counts both columns, and in this case the row with "Ayn Rand" in the title would have received a very high score for the query string '"Ayn Rand"'.

There are many, many options available with full text searching, I've only touched on the basics. Here's one of the extras: If you build an index on two columns, you can refer to one column in the CONTAINS clause and the search will only look at that column... another reason not to make the mistake I did, better to index on more columns, not fewer, you'll have more freedom when designing your queries.

Oh, and by the way: The default boolean operator is AND, just like in Google. In other words, the query strings 'Ayn Rand' and 'Ayn AND Rand' are the same, and they're different from 'Ayn OR Rand'.

Breck Carter is principal consultant at RisingRoad Professional Services, providing consulting and support for SQL Anywhere databases and MobiLink synchronization with Oracle, SQL Server and SQL Anywhere. He is also author of the SQL Anywhere blog.

Breck can be reached at **breck.carter@gmail.com**.

WORLDWIDE HEADQUARTERS
ONE SYBASE DRIVE
DUBLIN, CA 94568-7902
U.S.A.
CONTACT_US@iANYWHERE.COM
NORTH AMERICA
T 1-800-801-2069
1-519-883-6898
EUROPE, MIDDLE EAST, AFRICA
+44 1628 597 100
ASIA PACIFIC
+852 2506 8700
JAPAN
+81 3 5210 6380

www.sybase.com/iAnywhere