Steve Babcock

4/16/2017

Two New Database Ideas

The maintainers of the SQLite source allow users to submit proposals for new ideas. However their website warns, "…most ideas are bad ideas." Nevertheless, I have implemented two extensions into the database which provide more concise syntax, slight improvements in the efficiency of some queries, and security against injection attacks. The first is a new keyword called FORECAST. It is used with queries that contains both a function, and a condition on its output. At the written and computational levels, it often avoids the need for subqueries. The second idea provides an alternative set of randomized keywords which the user is forced to use. Sounds counterintuitive at first, but there is also a conversion program to assist in changing standard, human-readable queries for use on web-servers. The majority of SQL injection attacks make use of keywords in some way. So with a database system using these randomized keywords, how would any attacker know how to write a malicious input?

Consider the following table and query…

Purchase

Customer	DateOfSale	Item	Price
Evan	20170401	Coca-cola	2
Sally	20170211	Pepsi	1
Dave	20160311	Dr Pepper	3
Vivian	20151031	Mountain Dew	10
Tyler	20141212	Sprite	5
Becky	20130511	Fanta	4

```
SELECT Total FROM (SELECT SUM(Price) AS Total FROM Purchase) WHERE Total > 10;
```

A traditional way to write the sum of the Price column would likely use a subquery. Anyone who has written much SQL knows syntax can quickly become confusing as more and more layers of nested subqueries begin to unfold. It may be tempting to write the above as…

```
SELECT SUM(Price) AS Total FROM Purchase WHERE Total > 10;
```

This seems to make sense, and has greater readability. However upon running this query, we find this outcome…

```
Error: misuse of aggregate: SUM()
```

This occurs because the WHERE clause can only refer to the input of a query. By input, I mean before functions are processed, and the nature of table columns is inherently changed in some way. This is where FORECAST comes into play; it provides a similar set of condition upon the output of a query. Thus it can reference column values mutated by functions and given new names. So the above can be rewritten as…

```
SELECT SUM(Price) AS Total FROM Purchase FORECAST Total > 10;
```

As you can see, this new keyword is very similar to WHERE, but most resembles HAVING in its implementation. As queries become more complex, avoiding subqueries as much as possible really helps keep syntax concise and readable.

```
SELECT SUM(Price) AS Tot FROM Purchase WHERE Price > 3 FORECAST Tot > 10;
```

In SQLite, each SELECT statement is parsed and processed into the following…

```
struct Select {
	ExprList *pEList;      /* The fields of the result */
	u8 op;                 /* One of: TK_UNION TK_ALL TK_INTERSECT TK_EXCEPT */
	LogEst nSelectRow;     /* Estimated number of result rows */
	u32 selFlags;          /* Various SF_* values */
	int iLimit, iOffset;   /* Memory registers holding LIMIT & OFFSET counters */
	#if SELECTTRACE_ENABLED
	char zSelName[12];     /* Symbolic name of this SELECT use for debugging */
	#endif
	int addrOpenEphm[2];   /* OP_OpenEphem opcodes related to this select */
	SrcList *pSrc;         /* The FROM clause */
	Expr *pWhere;          /* The WHERE clause */
	ExprList *pGroupBy;    /* The GROUP BY clause */
	Expr *pHaving;         /* The HAVING clause */
	Expr *pForecast;       /* The FORECAST clause */
	ExprList *pOrderBy;    /* The ORDER BY clause */
	Select *pPrior;        /* Prior select in a compound select statement */
	Select *pNext;         /* Next select to the left in a compound */
	Expr *pLimit;          /* LIMIT expression. NULL means not used. */
	Expr *pOffset;         /* OFFSET expression. NULL means not used. */
	With *pWith;           /* WITH clause attached to this select. Or NULL. */
};
```

Notice the property Expr *pForecast has been added. This is the single most place is the source code where FORECAST is defined. The function which allocates each Select struct is sqlite3SelectNew(). By examining how this function is called with the queries we discussed above shows a small improvement in efficiency. After building the debug version of the new sqlite3.c, open it with gdb or lldb, and then place a breakpoint on sqlite3SelectNew(). During the execution of…

```
SELECT Tot FROM (SELECT SUM(Price) AS Tot FROM Purchase) WHERE Tot > 10;
```

You'll notice the breakpoint causes execution to halt twice - once for the subquery and then once again for the outer query. Interestingly though, this breakpoint is only reached once when the FORECAST version of this query is ran.

SELECT statements are processed in the function sqlite3Select() which is rather lengthy. Here is a very high-level overview of its main components and flow…

```
sqlite3WhereBegin(pParse, pTabList, pWhere…);
sqlite3WhereIsDistinct();
sqlite3WhereOrderedInnerLoop();
selectInnerLoop();
```

First the database scan begins with several arguments including the WHERE clause. Then processes to perform the DISTINCT and ORDER BY actions happen. selectInnerLoop() outputs the final result. For a statement with a subquery, this will all happen twice. FORECAST modifies this flow in a very simple way.

```
sqlite3WhereBegin(pParse, pTabList, pWhere…);
sqlite3WhereIsDistinct();
sqlite3WhereOrderedInnerLoop();
sqlite3ExprIfFalse(pParse, pForecast…);
selectInnerLoop();
```

By adding this function just before outputting the query results, we throw out any rows that do not satisfy the conditions provided in FORECAST. This allows us to avoid repeating the sqlite3Select() and creating two instances of the Select struct. Below are some results with a table containing much more data, and we can see small improvements in the running time of some queries using the new keyword.

```
//Created simple Junk table with one column 'Stuff' and imported junkInput.txt
//Following are the initial results using FORECAST compared to subquery method

with sq as (select stuff||stuff||stuff||stuff||stuff||stuff as b from junk) select b from sq where
substr(b,1,1) = 'x';

//2010 Windows laptop   Run Time: real 1.631 user 0.234375 sys 0.218750
//2010 Windows laptop   Run Time: real 1.906 user 0.265625 sys 0.234375
//2010 Windows laptop   Run Time: real 1.629 user 0.187500 sys 0.218750
//2010 Windows laptop   Run Time: real 1.533 user 0.156250 sys 0.187500
//2010 Windows laptop   Run Time: real 1.504 user 0.187500 sys 0.187500
//2010 Windows laptop   Run Time: real 1.808 user 0.218750 sys 0.203125
```

```
select stuffllstuffllstuffllstuffllstuffllstuff as b from junk forecast substr(b,1,1) = 'x';
```

```
//2010 Windows laptop   Run Time: real 1.531 user 0.203125 sys 0.171875
//2010 Windows laptop   Run Time: real 1.530 user 0.281250 sys 0.234375
//2010 Windows laptop   Run Time: real 1.662 user 0.187500 sys 0.265625
//2010 Windows laptop   Run Time: real 1.540 user 0.250000 sys 0.140625
//2010 Windows laptop   Run Time: real 1.591 user 0.156250 sys 0.234375
//2010 Windows laptop   Run Time: real 1.519 user 0.171875 sys 0.265625
```

Avg Real Time With Subquery: 1.6685
Avg Real Time With FORECAST: 1.5622
Improvement 6.37%

This first trial showed very promising results, but when ran on a newer machine, the data greatly converged between the two queries.

```
//traditional subquery
//2015 Mac-Mini  Run Time: real 0.074 user 0.054080 sys 0.009100

//with forecast
//2015 Mac-Mini    Run Time: real 0.074 user 0.053880 sys 0.009019
```

The results tend to be very consistent on my Mac-Mini where the real time of the queries is equal. Interestingly, the user and sys times are up and down from one to the other. In general, I would conclude FORECAST shows potential to speed up some queries, and that difference has an inverse relationship to the performance of the hosting machine. Oftentimes, it will be insignificant if existing at all. Nevertheless, the improvement of SQL syntax is still a benefit.

Injection ranks number one on OWASP’s 2013 list of website vulnerabilities. The most commonly used defense is parameterization where the template of a prepared statement is sent to the DBMS before any user input. This allows the user input to be separately recognized, and any SQL commands an attacker may have entered can be ignored. The cat and mouse game of cyber attack and defense is ever changing. I believe the best defense is actually a good offense, and that security in computing is hopeless through a philosophy of responding to attacks once they become known. So I propose a new defense based on the idea that an attacker cannot write injection attacks if he or she doesn’t know the language the database is expecting. This is achieved by randomizing SQL keywords, and allowing the database manager to convert queries into this non-human readable language through a special program.

Consider the following table and query…

User Name	Password
Steve	password
Dude	Sweet

```
SELECT COUNT(1) FROM User WHERE Name = 'Steve' AND Password = ‘password’;
```

The above will be formed with expected inputs of Steve and password. However, if an attacker types the following, Steve   stuff’ OR ‘1’=‘1   he or she could access Steve’s account without knowing the password if no traditional security measures are implemented.

```
SELECT COUNT(1) FROM User WHERE Name = 'Steve' AND Password = 'stuff' OR
‘1'='1';
```

The unexpected, always true condition allows the intent of the query to be bypassed. Nearly all injection attacks make use of keywords in some way or another. So if we encode our database’s language in the following way, these attacks become much more difficult,

Keyword	Randomized Version
SELECT	XZQJOUBRSALC
FROM	XZIBCLQUNDSU
WHERE	XZHVMTORYDGW
AND	XZGSEMFANFEO
OR	XZNJUMKUANJO

Now the syntax accepted by the database for the same query as above will be…

```
XZQJOUBRSALC COUNT(1) XZIBCLQUNDSU User XZHVMTORYDGW Name =
'Steve' XZGSEMFANFEO Password = ‘password';
```

Notice there still actually exists an injection attack if the user were to input the following…

```
Steve      asdf' XZNJUMKUANJO '1'='1
```

But without any knowledge of the specific setup of this database, it would be mathematically very difficult to guess the encoding of OR. Each encoded keyword begins with XZ and is followed by ten random letters. So the number of possibilities for each is 26 ^ 10 or over a hundred trillion. Even if the encoding were to be discovered by an attacker, other measures such as parameterization could stop them. It could then be fairly simple to re-randomize the database’s encoding, and update web server files that interact with it.

I believe a similar idea could greatly improve operating system security because a common attack exploits buffer overflows by injecting malicious byte code as input. The success of the attack again hinges on a critical assumption, that the hacker knows the machine’s language and thus can write attack code. For example, here is the byte code to create a new shell in 64 bit Linux…

```
\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05
```

What this programs does is use a sys call to execve with the argument “\bin\sh”. The Linux manual for system calls states execve is mapped to \x3b which can be seen near the end of our payload just before a interrupt is sent to the CPU. There’s nothing special about the order system calls are mapped, and theoretically could be scrambled. Say execve became \x2e instead. Then it becomes extremely difficult for an attacker to write a buffer overflow without knowing this new encoding. The tricky part will be compiling and running any other applications on this machine. If a compiler is also rewritten with the new opcode values, I believe everything would work smoothly. Though the machine would be limited to installing applications where the source code is available. Perhaps it would be possible to convert binary objects from the standard opcodes to the new randomized values. This would be a significantly harder task as byte values in the data section of applications should remain unchanged.

In conclusion, FORECAST and encoded keywords are two new ideas for use in databases. They can provide better syntax, faster query processing in some cases, and levels of security unexpected by today's hackers. Ever since my first database class, I've thought needing a subquery when functions force a column(s) to be renamed was kind of silly. This new keyword eliminates these unneeded subqueries. While more complex defenses exist, very complex SQL injection attacks exist as well. I think these defenses overlook the fundamental reason databases can be hacked. That is the hacker correctly assumes his or her injection code will be accepted by the DBMS.

References

Database Security: Threats and Security Techniques. Deepika, Nitasha. International Journal of Advanced Research in Computer Science and Software Engineering. https://www.ijarcsse.com/docs/papers/Volume_5/5_May2015/V5I4-0780.pdf. 5 May 2015. Accessed 5 April 2017.

OWASP Top 10 2013. https://www.owasp.org/index.php/Top_10_2013-Top_10. Last modified 21 August 2015. Accessed 5 April 2017.

Searchable Linux Syscall Table For x86 and x86_64. https://filippo.io/linux-syscall-table/. Last modified 18 February 2017. Accessed 5 April 2017.

SQLite Documentation. https://sqlite.org/docs.html. Last modified 28 March 2017. Accessed 5 April 2017.