Random pleasures

In the search for generically modified solutions, Mark Whitehorn takes the reader's advice.

In the January column I discussed methods of returning a random selection of rows from a table. I provided two solutions, but one made use of Access-specific features, while the other didn't return an exactly pre-definable number of rows. I didn't think there was a totally generic solution using only set-based operations, but threw it open to the readers.

Linda : Table

ID	Foo	Baa
1	afds	j
2	afg	hjd
3	fg	gj
4	fg	gfhj
5	afg	gfh
6	fda	swtr
7	h	hse
8	sth	hjs
9	th	trhte
10	th	j
11	t	rsthj
12	afg	tdgj
13	adf	rsth
14	gad	jy
15	f	srth
16	dah	s
17	dsah	ytjs
18	ah	rth
19	h	dsyj
20	ht	dsy

Record: 1 of 200

Screenshot 1: The base table

Thanks to all who responded. Several people came up with code solutions which were interesting but, by definition, non-generic.

Ken Sheridan agrees that a universal, set-based solution is unlikely, because the SQL standard doesn't include a pseudo-random number generator. However, he did come up with an entirely set-based solution that answers a slightly different question. This question is, how do you find a subset that consists of rows which are evenly distributed throughout the table with respect to the values in the primary key?

In other words, given a table as used in the January issue (see screenshot 1), how would you extract, say, the rows in which the ID value was 10, 20, 30 and so on?

My simple answer is something like:

```
SELECT ID, foo
FROM Linda
WHERE 0 = id mod 10;
```

This uses the MOD function (which is part of the SQL standard). MOD returns the remainder of an integer division. So, for example, 7 MOD 3 returns the figure 1, while 40 MOD 10 returns zero.

This works but – and it's an important but – it only works when the primary key values are evenly distributed. Even using an autonumber field, as soon as you start deleting records, the efficiency of this solution begins to degrade (OK, it stops working).

Ken's solution is more complex, but it works with a non-sequential set of primary-key values (screenshot 2); mine doesn't (screenshot 3). It is also worth studying as an example of the flexibility that can be achieved with an intimate knowledge of SQL.

Ken's solution against a non-sequential set of PK values ...

ID	foo
50757	y
110278	gj
177723	fd
217241	gh
261253	fg
301967	hrw
354543	yteu
406657	fdsg
434194	gwt
489146	ewrt
538492	fdsg
598206	fs
642953	gfd
725599	tw
799968	fdsg
844537	th
892714	rew
926326	fds
968358	utik
995034	fs

Record: 1 of 20

Screenshot 2: Ken's solution works with non-sequential primary key values

```
SELECT L1.ID, L1.foo
FROM Linda AS L1 INNER JOIN ↙
Linda AS L2 ON L1.ID >= L2.ID
GROUP BY L1.ID, L1.foo
HAVING COUNT(*) MOD (SELECT ↙
INT(COUNT(*) / 20) FROM ↙
Linda) =0;
```

(*Key: ↙ code string continues*)

(INT is also part of the SQL standard.)

All you want to know is, how does it work? Try dissecting it yourself, because there is fun to be had in working it out. If you get stuck (or bored) the solution is below.

We can dissect out the first part of the statement and add another field to the list to make it easier to see what is happening.

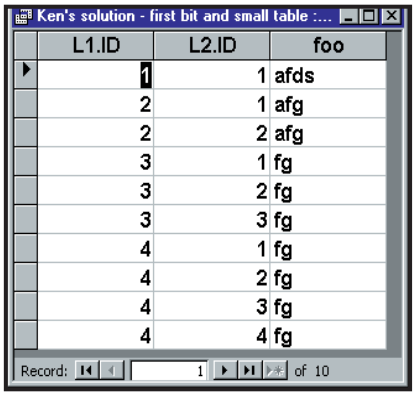
```
SELECT L1.ID, ↙
L1.foo, L2.ID
FROM Linda3 AS L1 ↙
INNER JOIN Linda3 ↙
AS L2 ON L1.ID >= ↙
L2.ID;
```

Mark's simple solution against a non-sequential set of PK values...

ID	Foo
259010	th
317310	t
798180	dah
257390	fdsg
159210	e
894750	e
122200	wy
459570	req
535410	rwh
816350	rwy
695560	ag
201140	qer
990330	h
916110	yuyt
972800	yy
960690	ttw
398020	dfhg
845230	hwr

Record: 11 of 18

Screenshot 3: Mark's solution working less well



This says, show me every record, joined to itself on the ID field. Oh, and also show me every record joined to every record that has a higher value in the ID field. I've modified this query to run against a version of the base table that has only four rows (and then sorted the answer table) to make this more apparent (screenshot 4).

Given 200 rows in the base table, this is going to generate a large answer table (20,100 rows in fact). However, the interesting point is that the different values of L1.ID appear a different number of times. In the L1.ID field, the value 1 appears only once (matched to itself). The value 2 appears twice, the value 3 appears three times and so on. This is why we get 20,100 rows in the answer table because it is equal to 1+2+3+4...+199+200.

So, the first part of the SQL statement generates a table in which each 'row' from the original table is represented a different number of times. This process works quite happily if the numbers in the ID field are not a continuous set, because it relies upon their relative – rather than their absolute – position.

Now think about another part of the SQL statement:

```
SELECT Int(Count(*)/20) ↙
FROM Linda;
```

This counts the number of rows in the original table (200) and divides by 20 – returning the value 10. Remember that our intention is to extract 20 records from the original table. The value that has just been calculated (10) essentially tells us that if we take every tenth record from the original table, we will get 20 rows in all.

So, the only remaining question is, how can we extract every tenth record from the answer table? Well, rather neatly, we can group the rows in the answer table and count the number of

Screenshot 4: Part of the SQL statement

times each is represented. As already discussed, the first is represented once, the second twice, and so on. So if we count the number in each group, we get a count in each group of 1, 2, 3 and so on.

All we have to do is to divide this number by 10 (as provided by SELECT Int(Count(*)/20) FROM Linda) and use the MOD function to give us only the rows in which the answer is a whole number. This magic is performed by the bit that reads:

```
HAVING ↙
COUNT(*) MOD (SELECT ↙
INT(COUNT(*) / 20) FROM ↙
Linda) =0;
```

Before anyone else does it, I'll point out (as Ken did) that this is a poor solution for large data sets because of the non-linear relationship inherent in the line:

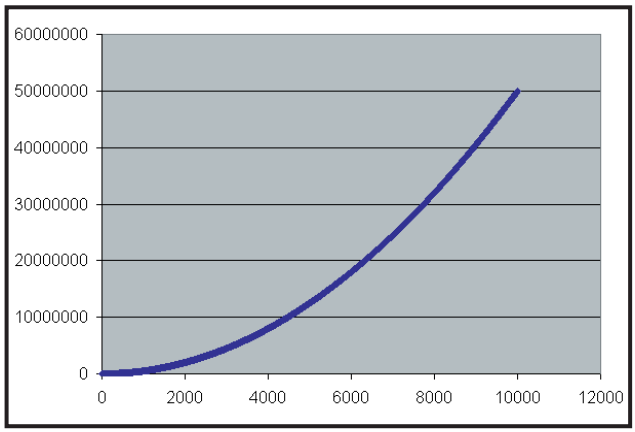
```
INNER JOIN Linda AS L2 ON ↙
L1.ID >= L2.ID
```

As the number of rows in the original table increases, the solution becomes increasingly slower because the number of records in the answer table grows so quickly (remember that it is based on 1+2+3..+n where n is the number of rows in the base table) as in screenshot 5.

Linda Ratcliffe, who originally posed the question, was delighted with the answer. However, she noted that in a single session, running the query will give a different set of 20 records each time, but the next time you open the database and run the query it repeats the same sets of records as in the previous session.

This is due to the way that computers generate random numbers. There is no easy way of getting a computer to 'choose' a number at random, they are too deterministic, so they typically use what are called pseudo-random numbers – essentially an almost-random set of numbers generated from an initial seed number; one side effect is as described. I use the term side-effect and not problem because, depending upon the app, this can be exactly what the user wants.

When repeated random numbers aren't desirable, you could try searching the help system for the word Randomize, as this will point to an often-applied solution.



Screenshot 5: Yes, it does really mean that 1,000 rows in the original table generate 50,005,000 rows in the answer table.....

Net addresses

I asked about how to get the address of a network card for someone's database. Ian Barker sent in some code that he uses. It originally came from the VB Knowledge Base on the January 1998 MSDN CD, Article ID: Q175472. He's tidied it up, added some comments and optional formatting for the ID. This worked on two of the three machines upon which I tried it (it simply returned zeros on the third). So give it a try, but you might have to do some tweaking.

The example MDB file (and the one used above) are available on the cover CD-ROM.

Teacher's pet

Any teachers reading this might like to have a look at the PDA column in this issue for an interesting program called Nstore. This mixes the reporting systems of databases and school teachers in an interesting way.

CONTACTS

Mark Whitehorn welcomes your feedback on the Databases column. Contact him via the *PCW* editorial office, or email: **database@pcw.co.uk**