

Paper 064-29

Creating Efficient SQL - Union Join without the Union Clause

Paul D Sherman, San Jose, CA

ABSTRACT

Following good object oriented practice of building query models we address the issue of poor query performance during use of the UNION clause. Although column-wise joins are rather well understood, the notion of the UNION or row-wise join is often perceived incorrectly as a gluing together of independent fullselect result sets rather than via model relations through higher levels of abstraction. This perception is excusable because there often does not exist suitable abstraction tables or normalization key values, and we must think creatively how a necessary abstraction can be derived. In the example to follow we help farmer Jack analyze taste and laboratory test data to determine the quality of his bean stalks.

Skill Level: Novice to Advanced, familiarity with SAS/ACCESS, SAS/CONNECT, Data step merge and Proc SQL pass-through operation.

INTRODUCTION

Of the four taboo Schema Query Language keywords probably the one most dangerous to query performance is the UNION clause(1). Unfortunately, the UNION clause is also the most widely used. A typical database programmer individually develops pieces of their query in stand-alone fashion, perhaps over the course of many weeks. Independently each query is perfectly sound. The trouble arises when one desires aggregated output, a result set comprising all queries: One simply glues the queries together, arranging each to return the same quantity and type of output columns or variables. The result is a single set of rows derived from each constituent query.

STYLES OF JOINS

In simple terms let us compare the two styles of table joins using both SQL and SAS® data step syntax. In the column-wise join two columns 'acol' and 'bcol' are returned with a result set only as large as the *intersection* of tables a and b tied together using values of their common key variable 'key'.

```
SELECT a.col as acol,          data _null_;
       b.col as bcol          merge a b;
FROM a                        by key;
       inner join b ON a.key = b.key  run;
```

<u>KEY</u>	<u>ACOL</u>	<u>BCOL</u>
p	one	six
q	two	seven
r	three	eight

The row-wise join returns a single column 'col' instead with a result set containing all the rows of both constituent tables a and b, regardless of the values of each table's 'key' column.

```
SELECT col FROM a            data _null_;
UNION                        set a b;
SELECT col FROM b            run;
```

<u>COL</u>
one
two
three
six
seven
eight

The ambiguity whether to list table a or b first, before the UNION statement is a big hint that this query will perform sub-optimally at the database. When constituent table sizes are small one may not notice the reduction in query performance. However, when UNION members are complex multi-table fullselects or of enormous content row size performing a UNION query might not even work, instead returning the dreaded 'Query Timeout' run time error.

WHAT MIGHT GO WRONG

First, at compile time one may overlook each query's output or SELECT list. It's easy to do especially when queries are particularly lengthy or complex. Further, one may not be aware which columns are numeric and which are alphabetic, since SQL is by nature not a strongly typed language. One doesn't know until run-time, too late that is, which type of data resides in each column. There are ways around this however, through use of the type cast() function, but such ways are not commonly used. Luckily, column count and type mis-match are easily debugged - improperly UNIONed queries won't even run at all.

```
select aNumCol from a
UNION
select aCharCol from b
```

Alot more subtle and, worse yet, platform specific is the issue of database performance. Since each member of a UNION clause is itself a fully formed, stand alone fullselect statement, database implementations execute each member simultaneously in separate threads of execution or processes on the database. While it may seem useful to do more than one activity at the same time, multi-tasking in other words, the analogy stops here. It is never useful to do the same sort of activity more than once, it's called redundancy. Each process allocates its own memory space, requests its own access rights and performs its own execution of code. When two processes call upon the same content, each must be scheduled in turn, thus they are not truly multi-task independent anymore. One trades off lack of fore sight with ease of query assembly. To make matters worse, such redundant query perhaps having many UNIONed members is syntactically correct and will run - albeit sometimes and erratically very slowly depending upon database load, an external characteristic.

WHAT TO DO

Rather than wait until it's too late to assemble or glue together output content, think about each UNION member as a data source object in its own right, just as if it was another base table. The correct place for defining all a query's necessary tables is the FROM clause, which is the first clause to be interpreted by the query processor. Thinking about when each table should be included and how it should relate to the other tables is precisely the up-front design practice which leads to optimal query performance. Rather than tying together two sets at phase V each derived from the same content, it's far better to do a single fetch once and tie the sets together sooner in phase II.

```

/-----\
|   SELECT ==union== SELECT   |   V
| /-----\ /-----\       |
| | FROM ... | | FROM ... |   |   I
| |   ON    | |   ON    |   |   II
| | WHERE   | | WHERE   |   |   III
| | GROUP   | | GROUP   |   |   IV
| \-----\ \-----\       |
|-----|

```

Step-V assembly : the UNION clause joins content much too late

```

/-----\
|   SELECT       SELECT       |   V
| /-----\ /-----\       |
| | FROM ... | | FROM ... |   |   I
| \-----\ \-----\       |
| |           | |           |   |   II
| |   ON ===== ON   |   |
| \-----\ \-----\       |
| | WHERE     | | WHERE     |   |   III
| | GROUP     | | GROUP     |   |   IV
| \-----\ \-----\       |
|-----|

```

Step-II assembly : give each table object a proper place in the query model

The expensive variable filtering and selection steps, in the WHERE and SELECT clauses respectively, are performed only once regardless how many data source table object members are included and joined.

A dilemma: Probably the single most reason one chooses (unknowingly) a late step-V assembly is the lack of suitable keys upon which to relate. When the keys are available in each subset but simply unSELECTed, they can simply be brought out for relationship purposes. Tables which don't have a key column are a bit more difficult to deal with, but there are ways around this problem as will be seen later.

EXAMPLE - ASSURING QUALITY

Every good query begins with a clearly stated problem and plan. In an effort to insure continuing high quality of his beans, farmer Jack hires a taste tester and analytical laboratory to sample and report various aspects of his bean crop. The testers and the laboratory each return their information at different times which Jack loads into separate database tables. Different beans go to each inspection, since the tests are destructive. After tasting you cannot measure your bean's size and weight, and who would want to eat something after being through a laboratory? However, Jack only needs to know the summary results per each bean stalk from which the samples come.

Jack's two quality tables are BEAN.TASTE and BEAN.LAB, each having a beanid column as unique key. Each row is exclusive to both tables, since the same bean cannot be measured twice or by more than one test. Therefore, one would not find any common matching rows by BEANID alone. To combine both results one might UNION the tables like so:

```
select beanid from bean.taste
UNION
select beanid from bean.lab
```

It is not clear nor specified which table to list first, since both results are of equal character and importance. Realizing this subtle degeneracy is precisely the secret to good query performance: There must never be any ambiguity as to when a table object is fetched and related. There's a time and a place for everything.

```
SELECT ...          SELECT ...
FROM bean.taste    ?? FROM bean.lab
UNION              ? or ? UNION
SELECT ...         ?? SELECT
FROM bean.lab      FROM bean.taste
```

Since each type of analysis, test and lab, provide different attributes, the common set of distinct attributes is the higher level of abstraction to which each table can be related, in turn. These attributes are explicitly named in the PARM column of Jack's bean quality tables. Although one might see further ambiguity in the sequence of table relation, there is none. From a performance perspective it makes good sense to fetch tables from smallest to largest size (most abstract to most concrete). Technically, the inner most loop should spin around as little as possible. In our present example there are fewer lab measurements than tastes, so we choose to join bean.lab first.

```

/-----\          +-----t--+
| PARS |          | BEAN |
+-----+          | TASTE |
| PARM |-----o PARM |
\-----/          +-----+
| | BEAN |          | VALUE |
| | LAB  |          | BEANID |
| |     |          +-----+
| |     |          +---o PARM | |
| |     |          | VALUE |
| |     |          | BEANID |
+-----+          +-----+

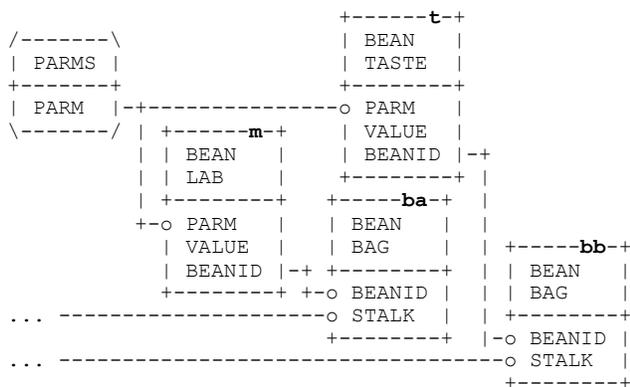
```

Temporary table parameters, coded as a VALUES list, contain one row each for all possible attribute names among both data tables. Of course, there never is the same parm name in both measurement tables; labs don't taste their beans and testers don't care about size or weight.

```
FROM ...
  inner join (
    values
      'Taste', 'Aroma', 'Flavor', 'Size', 'Weight'
  ) as parms (parm) ON 1=1
```

Notice the odd looking relationship expression; it effectively performs a cartesian product multiplication with the result set presently collected. In other words, each value of parm is added to every existing row. This is exactly what normalization means - the size of a column set giving unique values of key variables is its degree. We are essentially adding one more degree to the query model. The "1=1" relationship looks weird but is perfectly correct for our model, and never hinders query performance. When creating an abstract pseudo table there will always appear a trivial relationship expression, because we are creating an additional level of normalization.

Related in this manner, the outer-join insures a result set accumulates any available optional data and never is filtered by it. Not every bean is measured, and we wish to process all beans regardless whether they are actual crop or quality determining surrogates. Further, TASTE and LAB never join rows to the same parm, insuring the result set is never cartesian. Next, we must de-reference each result's BEANID through an instance of BEAN.BAG to obtain its stalk. We must use separate BAG instances since beans in one quality data table are not the same as those in the other table.



Remaining is only a simple connection of each BAG instance to a common and more abstract BEAN.STALK table. Since we have insured uniqueness of parm names across TASTE and LAB, we may coalesce() their values together into a single output result column. When BEAN.TASTE.VALUE has an entry, BEAN.LAB.VALUE will always be null, and vice-versa.

```

SELECT ...
      coalesce(t.value, m.value) as col
    
```

In general, there will always as many levels of coalesce() as there are outer-joined table objects, less one. Essentially here is where each member of a UNION or row-join is implemented. Since through our carefully devised abstraction each outer-join constituent 'raises its hand' only one at a time, there will only be one argument of the nested coalesce() being non-null. The coalesce() function simply picks out this value for any and all result set rows. It is a critical error, of course, when all coalesce() arguments don't have the same variable type - we cannot select a char format column from one table and fold it onto an integer format column from another table. However, SQL is rather sloppy in that it is possible for char and varchar to coalesce() together.

parm	aval	bval	cval	dval	out_val
a	'A'	.	.	.	'A'
b	.	'B'	.	.	'B'
c	.	.	'C'	.	'C'
d	.	.	.	'D'	'D'

Pictorially, think about four separate mutually exclusive tables multiplexing their column values together, one by one, over as many rows as there are tables and normalized parameter values.

```

          /----a/   coalesce(aval,
parm      /aval /
(a,b,c,d) ----+----o/----/b/   coalesce(bval,
          |      /bval /
          |----o/----/c/ =====> coalesce(cval,
          |      /cval*/
          |----o/*****/d/          dval
          |      /dval /          )
          |----o/----/          ) ) as out_val
    
```

Does this multiple nesting of coalesce() functions inhibit query performance? Not at all, because this scalar function has all the information it needs during inspection of any given result set row to make its determination which argument to return, and pass through to the SELECT clause. If more than one argument is non-null, coalesce() simply returns the first non-null argument as listed from left to right. The simple non-null selection is very simple, probably internally implemented in very low level code and operates very quickly with almost no database resource overhead. A SELECT clause may appear messy, but looks are only skin deep.

Aggregating over the entire model using STALK and PARM as unique keys, Jack easily tabulates how many beans are of each type of taste and lab measurement. Stalks which have the greatest number of "taste=superb" beans of course are his prize winning plants. The final SQL statement is shown below. Notice that we have incorporated dictionary table BEAN.TASTEDEF providing measurement values in plain english, for those measurements where a description is available. Because the relationship is in first normal form to the existing query's result set there will be minimal adverse performance degradation. In practice and for best results one wraps the aggregated fullselect inside a single member FROM clause and performs other ancillary dictionary table joins *after* the aggregation, thereby keeping the column size of the result set at each step of the database's access plan always as small as possible.

```

SELECT stalk.stalk,
       parms.parm,
       coalesce(m.value, t.value)) as value,
       count(coalesce(m.value, t.value)) as numbeans,
       tastes.descrip
FROM   bean.stalk as stalk
       inner join (values 'Taste','Aroma','Flavor','Size','Mass') as parms (parm) ON 1=1
       left outer join bean.lab as m ON parms.parm = m.parm
       left outer join bean.bag as ba ON m.beanid = ba.beanid
                               and stalk.stalk = ba.stalk
       left outer join bean.taste as t ON parms.parm = t.parm
       left outer join bean.bag as bb ON t.beanid = bb.beanid
                               and stalk.stalk = bb.stalk
       left outer join bean.tastedef as tastes ON parms.parm = tastes.name
                                               and bean.taste.value = tastes.value

GROUP BY stalk.stalk, parms.parm, tastes.descrip
HAVING  parms.parm = 'Taste'
       and coalesce(t.value, m.value)) = 'Yuck'
       and 100.0 * count(coalesce(t.value, m.value)) / count(stalk.stalk) > 40.0

```

Optional use of a post-aggregate filter allows automatic selection of any particular subset of results desired, such as finding those stalks which yield more than forty percent of their beans tasting terrible.

Typical output from this SQL statement is

STALK	PARM	VALUE	NUMBEANS	DESCRIP
Stalk14	Taste	3.1	154	Superb
Stalk14	Taste	3.2	5	Yuck
Stalk14	Size	12.2	12	.
Stalk14	Size	15.1	102	.
Stalk14	Size	19.0	2	.
Stalk14	Weight	3.8	157	.
Stalk14	Weight	5.0	2	.
Stalk14	Weight	6.4	1	.
Stalk15	Taste	3.1	25	Superb
Stalk15	Taste	3.2	200	Yuck

Obviously, farmer Jack will need to re-plant Stalk15 very soon.

WHEN A KEY DOES NOT EXIST

Critical to the unique conditional or outer-joining process is the existence of unique keys on which to relate table objects. When a table is extremely simplistic or ill-defined, it may lack this necessary attribute. As long as there is at least one column which can relate in strict equality to any other table's column there is hope. One might need to go through many levels of indirection, higher normal forms for instance, but the end result will still be of sound and good performance.

Here we show how column VAL can be tied into the main query model through column U by a third normal form de-reference of two other available tables and the relationships (D.u -> D.t=E.s), (E.s -> E.r=MYTAB.q) and (MYTAB.q -> MYTAB.val).

```

+----+
| D |
+----+
...| u | + E | |MYTAB|
| t |--| +----+ +-----+
+----+ |--| s | | val |---+o... to outer-join
| r |--| q | |---o... multiplex
+----+ +-----+ |---o...

```

Sometimes finding an appropriate pseudo key requires creative thought. In Jack's example each attribute table contains an ID column whose values are unique in themselves. De-referencing through BEAN.BAG gives a column common to both tables. Further, since each table always has different ID values we are assured never to have a cartesian product from both tables trying to join at the same row-time.

```

FROM ...
inner join bean.bag as ba ON ... = ba.stalk
left outer join bean.taste as t ON ba.id = t.id
inner join bean.bag as bb ON ... = bb.stalk
left outer join bean.lab as m ON bb.id = m.id

```

We were lucky in this example. Often an external or foreign table doesn't have any column with common or useable attribute values. In this case we must create our own de-reference table. Depending upon the available column values one may need to code a few redundant instances to cover all value combinations. Again, this seemingly redundant situation is actually a good thing; it means that a table contains more than one type of information content - an undesirable object property - which should be separated(2).

```
FROM ...
  inner join (values ('A',1)) as foo_ (i,o) ON ... = foo_.i
  left outer join foo ON foo_.o = foo.col
```

It is *not* appropriate to simplify this redundancy using inequality, range and OR-like relationships. Such will be ignored during the database's access plan determination phase, leaving fragmented FROM clause objects and terribly slow query performance. Much too many rows will be fetched, only to be filtered away later when the inequalities and other expressions are applied. When there are different pseudo key values they must always be addressed by separate table instances or de-referencing table rows.

```
      xxx wrong xxx
FROM ...
  inner join (values ('A',1,5)) as foo_ (i,of,ot) ON ...
  left outer join foo ON foo.col >= foo_.of
                    and foo.col <= foo_.ot
  inner join (values ('B',3,4)) as bar_ (i,of,ot) ON ...
  left outer join bar ON bar.col >= bar_.of
                    and bar.col <= bar_.ot
```

```
      --- correct ---
FROM ...
  inner join (values ('A',1)) as a_ (i,o) ON ...
  left outer join a ON a.col = a_.o
  inner join (values ('A',2)) as b_ (i,o) ON ...
  left outer join b ON b.col = b_.o
  inner join (values ('A',3)) as c_ (i,o) ON ...
  left outer join c ON c.col = c_.o
  inner join (values ('A',4)) as d_ (i,o) ON ...
  left outer join d ON d.col = d_.o
  inner join (values ('A',5)) as e_ (i,o) ON ...
  left outer join e ON e.col = e_.o
  inner join (values ('B',3)) as p_ (i,o) ON ...
  left outer join p ON p.col = p_.o
  inner join (values ('B',4)) as q_ (i,o) ON ...
  left outer join q ON q.col = q_.o
```

This may pose trouble when there are only decimal or floating point column variables available, but it's important to remember that relationships should only be of discrete quantities. It is possible, therefore, to discretize a continuous variable with clever use of int(), trunc(), floor(), ceil() or other scalar function.

CONCLUSION

There is rarely a need for the UNION clause. Such apparent need indicates lack of proper query thought. We have demonstrated how farmer Jack can join together two different attribute tables of bean quality, derived from independent external sources, and determine his aggregate stalk quality. We also show a few tips for getting around the sticky situation of missing or strange pseudo keys. It is the aim of this article to help every database programmer achieve sub-minute execution time for any of their queries.

REFERENCES

Delwiche, Lora D. and Slaughter, Susan J., *The Little SAS® Book: A Primer*, Cary, NC: SAS Institute, Inc., 1995. 228pp.
 Edwards, Betty, *Drawing on the Right Side of the Brain*, Los Angeles: J. P. Tarcher, 1979.
 Sherman, Paul, *Creating Efficient SQL - Four Steps to a Quick Query*, in *Proceedings of the Twenty-Seventh Annual SUGI Conference*. Cary, NC: SAS Institute, Inc., 2002. p073-27.
 Sherman, Paul, *Demeter in the Database*, in *Proceedings of the Twenty-Eighth Annual SUGI Conference*. Cary, NC: SAS Institute, Inc., 2003. p172-28.

1. The four taboo Schema Query Language keywords are UNION, DISTINCT, WHERE and ORDER BY. CASE...END is sometimes added to the list as a fifth taboo keyword, but its impact on query performance is not nearly as profound as with the other four. Three of these have already been discussed in detail in an earlier article by the author.
2. One must remember that rows of a table are different objects. Only when the key values of rows are identical can rows be combined together. A subject of much controversy and debate, the database table - and all of its rows - is in fact not analogous to an object in an object-oriented sense.

ACKNOWLEDGEMENTS

Good query models depend on meaningful, realistic ways of describing observations, and I thank Katesuda Chinmeteepitak for providing such description of plants, materials and the analytic observation process in general. Howard Sherman deserves honor for so clearly introducing me to the unity of opposites - such as abstraction and concreteness - and the method of dialectics. Farmer Jack gives many thanks to Jelly Belly Corporation for providing samples showing us bean counting can taste good, too. Lastly, I am grateful to IBM Corporation and Hitachi Global Storage Technology for providing the tools and opportunity which made this work possible.

TRADEMARK CITATION

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute, Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are registered trademarks or trademarks of their respective companies.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Paul D Sherman
 310 Elan Village Lane, Apt. 117
 San Jose CA 94134
 Phone: 408-383-0471
 Email: sherman@idiom.com

APPENDIX - FARMER JACK'S DATA**BEAN.TASTE**

<u>PARM</u>	<u>VALUE</u>	<u>BEANID</u>
Taste	3.1	843001
Flavor	2.2	843001
Aroma	1.1	843001

BEAN.LAB

<u>PARM</u>	<u>VALUE</u>	<u>BEANID</u>
Size	14.2	734001
Weight	5.0	734001

BEAN.TASTEDEF

<u>NAME</u>	<u>DESCRIP</u>	<u>VALUE</u>
Aroma	Fragrant	1.1
Aroma	Offensive	1.2
Aroma	Rotten	1.3
Aroma	Stink	1.4
Flavor	Sweet	2.1
Flavor	Sour	2.2
Flavor	Bitter	2.3
Flavor	Bland	2.4
Taste	Superb	3.1
Taste	Yuck	3.2

DATABASE DEFINITION STATEMENTS

```
create table bean.taste (
  parm char(8) not null,
  value varchar(32),
  beanid integer not null,
  primary key (beanid, parm)
);
```

```
create table bean.lab (
  parm char(8) not null,
  value double,
  beanid integer not null,
  primary key (beanid, parm)
);
```

```
create table bean.tastedef (  
    name char(8) not null,  
    value double,  
    descrip char(32) not null,  
    primary key (name, descrip)  
);  
  
insert into bean.taste (beanid, parm, value) values  
    (843001, 'Taste', 'Superb'),  
    ...  
;  
  
insert into bean.lab (beanid, parm, value) values  
    (734001, 'Size', 14.2),  
    (734001, 'Weight', 5.0),  
    ...  
;  
  
insert into bean.tastedef (name, value, descrip) values  
    ('Aroma', 1.1, 'Fragrant'),  
    ('Aroma', 1.2, 'Offensive'),  
    ('Aroma', 1.3, 'Rotten'),  
    ('Aroma', 1.4, 'Stink'),  
    ('Flavor', 2.1, 'Sweet'),  
    ('Flavor', 2.2, 'Sour'),  
    ('Flavor', 2.3, 'Bitter'),  
    ('Flavor', 2.4, 'Bland'),  
    ('Taste', 3.1, 'Superb'),  
    ('Taste', 3.2, 'Yuck'),  
;
```