Gaining Speed With Programming

R:BASE Technologies, Inc.

Gaining Speed With Programming

by R:BASE Technologies, Inc.

Welcom to Gaining Speed With Programming!

This document illustrates multiple ways of executing and retrieving the same set of records. By taking advantage of the latest improvements in R:BASE operations, the examples demonstrate how you can replace many older methods of retrieving data with quicker, more efficient sections of code.

To provide you with enhanced flexibility in retrieving data, R:BASE supports full extensions for ANSI Level 2 Structured Query Language (SQL) and partial 1992 ANSI Level II SQL. Accordingly, the following topics demonstrate R:BASE's capabilities to perform multi-table SELECTs and correlated sub-SELECTs. In addition, comparative examples of code illustrate which programming techniques produce the best results.

As you review the examples, notice that the line-by-line changes in the code are highlighted in a bold typeface font.

Gaining Speed With Programming

Printed: December 2009 in Murrysville, PA

First Edition

Table of Contents

1 Finding Minimum and Maximum Values

This section demonstrates techniques for finding the minimum and maximum values from the same table.

Note the use of two separate COMPUTE commands. Example 2 runs faster because R:BASE allows both COMPUTEs to be executed in one command. Example 3 is the fastest because it uses the SELECT INTO *varlist* command.

```
SET VAR t1 TIME, t2 TIME, vDiff INTEGER

--Example 1 - Slow
SET VAR t1 = (.#TIME)
SET VAR vMin INTEGER
SET VAR vMax INTEGER
COMPUTE vMin AS MIN Altitude FROM Airports
COMPUTE vMax AS MAX Altitude FROM Airports
SET VAR t2 = (.#TIME)
SET VAR vDiff = (.t2 - .t1)
PAUSE 2 USING .vDiff

--Example 2 - Faster
SET VAR t1 = (.#TIME), vMin INTEGER, vMax INTEGER
COMPUTE vMin AS MIN Altitude, vMax AS MAX Altitude +
   FROM Airports
SET VAR t2 = (.#TIME), vDiff = (.t2 - .t1), +
  vTime = (RTIME(0,0,.vDiff))
PAUSE 2 USING .vTime

--Example 3 - Fastest
SET VAR t1 = (.#TIME), vMin INTEGER, vMax INTEGER
SELECT MIN (Altitude), MAX (Altitude) INTO vMin, vMax +
   FROM Airports
SET VAR t2 = (.#TIME), vDiff = (.t2 - .t1), +
  vTime = (RTIME(0,0,.vDiff))
PAUSE 2 USING .vTime
```

2 Surrounding the WHERE Clause with Parentheses

You can easily gain processing speed by NOT surrounding your WHERE clauses with parentheses where non-text values are being used. When R:BASE sees parentheses, the values are evaluated as text, even though the values are not. By removing the parentheses, your code speeds will increase.

```
SET VAR t1 TIME, t2 TIME, vDiff INTEGER, +
v1 TEXT = 'Seattle', v2 INTEGER = 93, v3 INTEGER = 1000

--Example 4 - Slow
SET VAR t1 = (.#TIME), CheckNum INTEGER = 0
WHILE CheckNum < 40 THEN
  SELECT AptCode INTO vAptCode INDICATOR ivAptCode +
  FROM Airports WHERE (StCode = .v2 AND CityName +
```

```
  CONTAINS .v1 AND Runway > .v3)
  SET VAR CheckNum = (.CheckNum + 1)
ENDWHILE
SET VAR t2 = (.#TIME), vDiff = (.t2 - .t1), +
  vTime = (RTIME(0,0,.vDiff))
PAUSE 2 USING .vTime

--Example 5 - Faster
SET VAR t1 = (.#TIME), CheckNum INTEGER = 0
WHILE CheckNum < 40 THEN
  SELECT AptCode INTO vAptCode INDICATOR ivAptCode +
  FROM Airports WHERE StCode = .v2 AND CityName +
  CONTAINS .v1 AND Runway > .v3
  SET VAR CheckNum = (.CheckNum + 1)
ENDWHILE
SET VAR t2 = (.#TIME), vDiff = (.t2 - .t1), +
  vTime = (RTIME(0,0,.vDiff))
PAUSE 2 USING .vTime
```

3 Accumulating Data with SELECT

With R:BASE's procedural language, you can use various code structures to accumulate variables from the data values in two different tables. Example 6 uses two DECLARE CURSOR commands to repeatedly locate the values and do the accumulations until all data has been used. Example 7 is faster. It uses one DECLARE CURSOR and one COMPUTE command to accumulate the sum. Example 8, the fastest, uses an INSERT command with a SELECT statement to accumulate the sum.

```
SET VAR t1 TIME, t2 TIME, vDiff INTEGER

--Example 6 - Slow
SET VAR t1 = (.#TIME)
SET VAR vLength = 0

DROP TABLE Totals
CREATE TABLE Totals (State TEXT 15, Length INTEGER)
DECLARE c1 CURSOR FOR SELECT State, StCode +
  FROM States +
  WHERE State IN ('Washington', 'Oregon', 'California')
OPEN c1
WHILE SQLCODE = 0 THEN
  FETCH c1 INTO vState INDICATOR ivState, +
    vStCode INDICATOR ivStCode
  IF SQLCODE <> 0 THEN
    BREAK
  ENDIF
  DECLARE c2 CURSOR FOR SELECT Runway +
    FROM Airports WHERE StCode = .vStCode
  OPEN c2
  WHILE SQLCODE = 0 THEN
    FETCH c2 INTO vRunway INDICATOR ivRunway
    IF SQLCODE = 0 THEN
      SET VAR vLength = (.vLength + .vRunway)
    ENDIF
  ENDWHILE
```

```
  INSERT INTO Totals VALUES (.vState, .vLength)
  DROP CURSOR c2
ENDWHILE
DROP CURSOR c1
SET VAR t2 = (.#TIME), vDiff = (.t2 - .t1), +
  vTime = (RTIME(0,0,.vDiff))
PAUSE 2 USING .vTime

--Example 7 - Faster
SET VAR t1 = (.#TIME)
DROP TABLE Totals
CREATE TABLE Totals (State TEXT 15, Length INTEGER)
DECLARE c1 CURSOR FOR SELECT State, StCode FROM States +
  WHERE State IN ('Washington', 'Oregon', 'California')
OPEN c1
WHILE SQLCODE = 0 THEN
  FETCH c1 INTO vState INDICATOR ivState, +
    vStCode INDICATOR ivStCode
  IF SQLCODE <> 0 THEN
    BREAK
  ENDIF
  COMPUTE vLength AS SUM Runway FROM Airports +
    WHERE StCode = .vStCode
  INSERT INTO Totals VALUES (.vState, .vLength)
ENDWHILE
DROP CURSOR c1
SET VAR t2 = (.#TIME), vDiff = (.t2 - .t1), +
  vTime = (RTIME(0,0,.vDiff))
PAUSE 2 USING .vTime

--Example 8 - Fastest
SET VAR t1 = (.#TIME)
DROP TABLE Totals
CREATE TABLE Totals (State TEXT 15, Length INTEGER)
INSERT INTO Totals SELECT State, SUM(Runway) +
  FROM States,Airports +
  WHERE States.StCode = Airports.StCode +
  AND State IN ('Washington', 'Oregon', 'California') +
  GROUP BY State
SET VAR t2 = (.#TIME), vDiff = (.t2 - .t1), +
   vTime = (RTIME(0,0,.vDiff))
PAUSE 2 USING .vTime
```

4 Using Nested Cursors

Nested cursors must be used correctly for efficient execution of code. Notice how the following examples use the DECLARE CURSOR command differently. Both examples work correctly, but only the second attains normal processing speed.

In Example 9, the DECLARE CURSOR is placed inside the WHILE loop of the first cursor. This requires the second DECLARE CURSOR command to execute every time the first command finds a row. In Example 10, both DECLARE CURSOR commands are executed at the start of the program, and the second

DECLARE CURSOR command is opened inside the WHILE loop of the first command. By restricting the second DECLARE CURSOR from executing as often, this example runs much faster.

```
SET VAR t1 TIME, t2 TIME, vDiff INTEGER

--Example 9 - Slow
SET VAR t1 = (.#TIME)
DECLARE c1 CURSOR FOR SELECT State, StCode +
  FROM States +
  WHERE State IN ('Washington', 'Oregon', 'California')
OPEN c1
WHILE SQLCODE = 0 THEN
  FETCH c1 INTO vState INDICATOR ivState, +
    vStCode INDICATOR ivStCode
  IF SQLCODE <> 0 THEN
    BREAK
  ENDIF
  DECLARE c2 CURSOR FOR SELECT Runway +
    FROM Airports WHERE StCode = .vStCode
  OPEN c2
  SET VAR vLength = 0
  WHILE SQLCODE = 0 THEN
    FETCH c2 INTO vRunway INDICATOR ivRunway
    IF SQLCODE = 0 THEN
      IF vRunway > 5000 THEN
        SET VAR vRunway = (.vRunway + 100)
      ELSE
        SET VAR vRunway = (.vRunway + 50)
      ENDIF
      UPDATE Airports SET Runway = (.vRunway) +
        WHERE CURRENT OF c2
    ENDIF
  ENDWHILE
  DROP CURSOR c2
ENDWHILE
DROP CURSOR c1
SET VAR t2 = (.#TIME), vDiff = (.t2 - .t1), +
  vTime = (RTIME(0,0,.vDiff))
PAUSE 2 USING .vTime


Example 10 - Faster
SET VAR t1 = (.#TIME)
SET VAR vRunway INTEGER = NULL, vStCode INTEGER = NULL
DECLARE cursl CURSOR FOR SELECT State, StCode +
  FROM States +
  WHERE State IN ('Washington', 'Oregon', 'California')
DECLARE c2 CURSOR FOR SELECT Runway FROM Airports +
  WHERE StCode = .vStCode
OPEN c1
WHILE SQLCODE = 0 THEN
  FETCH c1 INTO vState INDICATOR ivState, +
    vStCode INDICATOR ivStCode
```

```
  IF SQLCODE <> 0 THEN
    BREAK
  ENDIF
  OPEN c2
  SET VAR vLength = 0
  WHILE SQLCODE = 0 THEN
    FETCH c2 INTO vRunway
    IF SQLCODE = 0 THEN
      IF vRunway > 5000 THEN
        SET VAR vRunway = (.vRunway + 100)
      ELSE
        SET VAR vRunway = (.vRunway + 50)
      ENDIF
      UPDATE Airports SET Runway = (.vRunway) +
        WHERE CURRENT OF c2
    ENDIF
  ENDWHILE
  CLOSE c2
ENDWHILE
DROP CURSOR c1
SET VAR t2 = (.#TIME), vDiff = (.t2 - .t1), +
  vTime = (RTIME(0,0,.vDiff)
PAUSE 2 USING .vTime
```

5 Writing Commands with SELECT

Often, flexible problem solving requires the use of the SELECT command. The first two of the following examples use the DECLARE CURSOR command to calculate an average and update a table. The third example, however, uses a SELECT command to create a file with an UPDATE command, which executes much faster.

```
SET VAR t1 TIME, t2 TIME, vDiff INTEGER

--Example 11 - Slow
SET VAR t1 = (.#TIME)
DECLARE cursl CURSOR FOR SELECT StCode FROM States +
  WHERE StCode > 90
OPEN c1
WHILE SQLCODE = 0 THEN
  FETCH c1 INTO vStCode INDICATOR ivStCode
  IF SQLCODE <> 0 THEN
    BREAK
  ENDIF
  SELECT AVG(Altitude) INTO vAve FROM Airports +
    WHERE StCode = .vStCode
  DECLARE c2 CURSOR FOR SELECT Lon_Deg +
    FROM Airports WHERE StCode = .vStCode
  OPEN c2
  SET VAR vLength = 0
  WHILE SQLCODE = 0 THEN
    FETCH c2 INTO vLon_Deg INDICATOR ivLon_Deg
```

```
    IF SQLCODE = 0 THEN
      UPDATE Airports SET Lon_Deg = (Lon_Deg / .vAve) +
        WHERE CURRENT OF c2
    ENDIF
  ENDWHILE
  DROP CURSOR c2
ENDWHILE
DROP CURSOR c1
SET VAR t2 = (.#TIME), vDiff = (.t2 - .t1), +
  vTime = (RTIME(0,0,.vDiff))
PAUSE 2 USING .vTime

--Example 12 - Faster
SET VAR t1 = (.#TIME)
DECLARE c1 CURSOR FOR SELECT StCode FROM States +
  WHERE StCode > 90
OPEN c1
WHILE SQLCODE = 0 THEN
  FETCH c1 INTO vStCode INDICATOR ivStCode
  IF SQLCODE <> 0 THEN
    BREAK
  ENDIF
  SELECT AVG(Altitude) INTO vAve FROM Airports +
    WHERE StCode = .vStCode
  UPDATE Airports SET Lon_Deg = (Lon_Deg / .vAve) +
    WHERE StCode = .vStCode
ENDWHILE
DROP CURSOR c1
SET VAR t2 = (.#TIME), vDiff = (.t2 - .t1), +
  vTime = (RTIME(0,0,.vDiff))
PAUSE 2 USING .vTime

--Example 13 - Fastest
SET VAR t1 = (.#TIME)
SET HEADINGS OFF
OUTPUT temp.$$$
SELECT ('UPDATE Airports SET Lon_Deg = (Lon_Deg / ') = 41 +
  AVG(Altitude) = 5 (') WHERE StCode = ')=17 StCode = 2 +
  FROM Airports GROUP BY StCode +
  WHERE StCode > 90
OUTPUT SCREEN
RUN temp.$$$
SET VAR t2 = (.#TIME), vDiff = (.t2 - .t1), +
  vTime = (RTIME(0,0,.vDiff))
PAUSE 2 USING .vTime
```

6 Speeding Up SELECT Processing

By making SELECTs faster, you can speed up the commands, joins, and rules that use them in their WHERE clauses.

Use Multi-Table SELECTs

Because R:BASE uses indexes when joining tables in multi-table SELECTs, SELECTs can run faster than sub-SELECTs. If you're using a sub-SELECT with the IN operator in the WHERE clause of a SELECT command, replace the sub-SELECT with a multi-table SELECT. Incorporate the sub-SELECT's table and conditions into the main SELECT. Consider the speed of the following examples:

```
SELECT CustId FROM Customer WHERE CustId IN +
  (SELECT Custid FROM Invoice WHERE +
  InvDate BETWEEN 06/15/1998 AND .#DATE)
```

This command may take 15 seconds using a sub-SELECT with an **IN** operator. The sub-SELECT must go through all rows in the invoice table for each row in the *customer* table. By changing the command to use a multi-table SELECT, it looks like this:

```
SELECT DISTINCT CustId FROM Customer, Invoice WHERE +
  Customer.CustId = Invoice.CustId AND InvDate BETWEEN +
  06/15/1998 AND .#DATE
```

This command should take only 8 seconds and looks only at the rows in the invoice table where *custid* exists and *invdate* matches a value. You need the DISTINCT on the multi-table SELECT because there are many rows in the invoice table for the linking column, and using the join will make many rows. Another option is to use the INNER JOIN syntax. The INNER JOIN also requires the DISTINCT to achieve the same results. Consider the following example:

```
SELECT DISTINCT CustId FROM Customer INNER JOIN Invoice ON +
  Customer.CustId = Invoice.CustId WHERE InvDate BETWEEN +
  06/15/1998 AND .#DATE
```

This command should take even fewer seconds as the INNER JOIN is a more current SQL standard and will only return rows when there is at least one row in both tables that match the join condition. When testing your own database your results may differ based upon column indexes and the choice of the fields in queries. For more information on joins, refer to "Using Joins" later in this section.

Use Faster Sub-SELECTs

If you can't use a multi-table SELECT, then take a look at your sub-SELECTs. A sub-SELECT is a SELECT command enclosed in parentheses and used in a WHERE clause. It's called a sub-SELECT because it's subordinate to the main command, which can be SELECT, EDIT USING, PRINT, EDIT, BROWSE or any other command that can include a WHERE clause.

A sub-SELECT returns a list of values that are compared to another value, using this structure:

```
...WHERE colname OP (sub-SELECT)
```

OP is the operator, with IN or NOT IN as the most frequent choices.

The operator tells R:BASE how to compare the list of items returned by the sub-SELECT to the list of rows requested by the main command. R:BASE doesn't use indexes to speed up this comparison, so it's important to use other techniques. Here are three ways to speed up a sub-SELECT:

1. Correlate the Sub-SELECT

Correlate the sub-SELECT to the main part of the query using an indexed column. That is, add a WHERE clause to the sub-SELECT (or add onto an existing sub-SELECT WHERE clause). Make sure the WHERE clause uses indexes to create the comparison list by choosing rows that match on a linking column with the main part of the command.

When a sub-SELECT can execute on its own, it isn't correlated. A correlated sub-SELECT uses a *table. column* item in the sub-SELECT's WHERE clause that is not in the sub-SELECT's column list. In other words, to correlate a sub-SELECT to the main query, add a WHERE clause to the sub-SELECT that uses an indexed item listed only in the main query column list.

In some cases, you can do this by using a copy of the table *t2* in the sub-SELECT and the table itself in the main command. The following is an example using RULES as the main command. The sub-SELECT is correlated to the main command, making the uniqueness rule faster.

```
RULES 'Value must be unique.' +
  FOR TblName SUCCEEDS +
  WHERE ColName IS NOT NULL +
  AND ColName NOT IN +
  (SELECT ColName FROM TblName t2 +
  WHERE t2.ColName = TblName.ColName)
```

The sub-SELECT is correlated because it can't stand on its own. It uses *tblname.colname* in its WHERE clause, which only the main command can access. The sub-SELECT's table is *t2*, not *tblname*.

The correlated sub-SELECT is faster because it forces R:BASE to do an internal join, which always uses indexes. For another example of using a correlated sub-SELECT to speed up a command, refer to "Four Examples of Outer Joins" later in this section.

2. Use EXISTS or NOT EXISTS

Use EXISTS or NOT EXISTS as the operator instead of IN or NOT IN, especially when it only matters that a sub-SELECT finds a value. For example, when you use a rule to ensure that a new value is unique, you don't care what the value is, just whether it already exists. Therefore, you can speed up the same rule by using the NOT EXISTS operator instead of NOT IN. **Note in the following example that neither EXISTS nor NOT EXISTS uses a column name.**

```
RULES 'Value must be unique.' +
  FOR TblName SUCCEEDS +
  WHERE ColName IS NOT NULL +
  AND NOT EXISTS +
  (SELECT ColName FROM TblName t2 +
  WHERE t2.ColName = TblName.ColName)
```

3. Limit the Sub-SELECT List

If you can't use a multi-table SELECT, and you can't correlate the sub-SELECT, try adding an indexed WHERE clause to the sub-SELECT. This will limit the number of items in the list that the sub-SELECT returns. R:BASE uses the index in the WHERE clause of the sub-SELECT. The fewer items that need to be compared, the faster the sub-SELECT.

Using Joins

As mentioned above, multi-table SELECTs join tables. When you perform a join, you specify one column from each table to join on. These two columns contain data that is shared across both tables. You can use multiple joins in the same SQL statement to query data from as many tables as you like.

A join can be an inner join, an outer join, or a self join. An inner join, like the INTERSECT command, includes only those rows that match on the linking columns. An outer join, like UNION, includes all rows that match as well as all rows that don't match on linking columns.

Most of the time, you'll do an inner join, though you will sometimes find it useful to do an outer join. For example, you need an outer join (like the UNION command) to get all rows in these cases:

- When joining a *customer* table with an *orders* table to list the customers who ordered something in the current month (inner join) as well as those who didn't order anything (outer join).
- When joining a *budget* table with an *expense* table to list each budget item, whether or not there was an expense for that item in the current month.
- When comparing a header (master table) on the "one" side of a one-to-many relationship against a detail (transaction table) on the "many" side to see all the rows of data, whether or not they have associated details.

In R:BASE, there are two different syntactical ways to express joins. The first, called *explicit join notation*, uses the keyword JOIN, whereas the second is the *implicit join notation*. The implicit join notation uses commas to separate the tables to be joined in the FROM clause of a SELECT statement. Thus, it always computes a cross join and the WHERE clause may apply additional filtered criteria. That filter criteria is comparable to join predicates in the explicit notation.

Example of an explicit "inner" join:

```
SELECT ALL FROM Product +
  INNER JOIN TransDetail ON +
  TransDetail.Model = Product.Model
```

Example of an implicit "inner" join:

```
SELECT ALL FROM Product t3, TransDetail t2 +
  WHERE t3.Model = t2.Model
```

Both of the above examples will result in the same output, only the example of the explicit "inner" join will be faster.

JOIN Types

Depending on your requirements, you can do an "INNER" join or an "OUTER" join. The differences are:

- **INNER JOIN:** This will only return rows when there is at least one row in both tables that match the join condition.
- **LEFT OUTER JOIN:** This will return rows that have data in the left table (left of the JOIN keyword), even if there's no matching rows in the right table.
- **RIGHT OUTER JOIN:** This will return rows that have data in the right table (right of the JOIN keyword), even if there's no matching rows in the left table.
- **FULL OUTER JOIN:** This will return all rows, as long as there's matching data in one of the tables.

More About OUTER JOIN

When you use an outer join, rows are not required to have matching values. The table order in the FROM clause specifies the left and right table. You can include a WHERE clause and other SELECT clause options such as GROUP BY. The result set is built from the following criteria:

- In all types of outer joins, if the same values for the linking columns are found in each table, R:BASE joins the two rows.
- For a left outer join, R:BASE uses each value unique to the left (first) table and completes it with nulls for the columns of the right (second) table when the linking columns do not match.
- A right outer join uses unique values found in the right (second) table and completes the rows with nulls for columns of the left (first) table when the linking columns do not match.
- A full outer join first joins the linking values, followed by a left and right outer join.

Four Examples of Outer Joins

Below, from slowest to fastest, are four examples of how to list all the invoice numbers in an *invoice* table, whether or not they have related rows in a *transx* table.

In each example, *invoice* has 1,000 rows and *transx* has 8,000 rows. There are 20 matches and 980 non-matches. In other words, in the first three examples the first SELECT (inner join) finds 20 rows and the second SELECT (outer join) finds 980 rows. And, in the last example, the LEFT OUTER JOIN performs the query with just one SELECT.

Uncorrelated Sub-SELECT

This first example shows how to list the invoice numbers with a simple sub-SELECT that doesn't have a WHERE clause correlating it to the main SELECT. It's slow, taking a long time to complete.

```
SELECT t2.InvId, SUM(t3.TPrice) +
  FROM Invoice t2, Transx t3 +
```

```
  WHERE t3.InvId = t2.InvId +
  GROUP BY t2.InvId +
  UNION +
    SELECT Invoice.InvId, $0.00 +
      FROM Invoice +
      WHERE InvId NOT IN +
        (SELECT InvId FROM Transx)
```

Correlated Sub-SELECT

Adding a correlated WHERE clause to the sub-SELECT makes it many times faster.

```
SELECT t2.InvId, SUM(t3.TPrice) +
  FROM Invoice t2, Transx t3 +
  WHERE t3.InvId = t2.InvId +
  GROUP BY t2.InvId +
  UNION +
    SELECT t1.InvId, $0.00 +
      FROM Invoice t1 +
      WHERE InvId NOT IN +
        (SELECT InvId FROM Transx +
        WHERE Transx.InvId = t1.InvId)
```

Correlated and NOT EXISTS

By changing NOT IN to NOT EXISTS for use with the correlated sub-SELECT, you can add a little more speed.

```
SELECT t2.InvId, SUM(t3.TPrice) +
  FROM Invoice t2, Transx t3 +
  WHERE t3.InvId = t2.InvId +
  GROUP BY t2.InvId +
  UNION +
    SELECT tl.InvId, $0.00 +
      FROM Invoice t1 +
      WHERE NOT EXISTS +
        (SELECT InvId FROM Transx +
        WHERE Transx.InvId = t1.InvId)
```

LEFT OUTER JOIN

By using a LEFT OUTER JOIN, and bypassing the second SELECT, you can add even more speed.

```
SELECT t2.InvId, SUM(t3.TPrice) +
  FROM Invoice t2 LEFT OUTER JOIN Transx t3 ON +
  t3.InvId = t2.InvId +
  GROUP BY t2.InvId
```

7 Speeding Up Applications

These tips show you how to use SQL to replace programs, complex reports, and relational commands as a technique for speeding up your applications.

In a network environment, where creating new tables and changing database structure can be difficult, SQL is the optimal choice for efficient processing. As a relational language, SQL works well with the relational database design. Try using SQL as an alternate method to temporary tables and try replacing

relational commands such as INTERSECT, SUBTRACT, and UNION.

Compare Headers and Details

Consider the task of listing all rows where a total in a header row does not match the total of all the detail values as stored in the detail row of that transaction.

The solution uses as an example the *ConComp* sample database that you received with your R:BASE installation media. In *ConComp*, the *transmaster* table is on the one side of a one-to-many relationship, and *transdetail* is on the many side. They're related by the *transid* column.

Transmaster holds the total dollar figure for a given transaction in its *netamount* column, and *transdetail* holds a total for each detail row in its *extprice* column.

The following SELECT command lists those *transid* values where the sum of the details is not equal to the total shown in the master row, which find no records:

```
SELECT t2.TransId, +
  MAX(t1.NetAmount), SUM(t2.ExtPrice) +
  FROM TransMaster t1 LEFT OUTER JOIN +
  TransDetail t2 ON +
  (t1.TransId = t2.TransId) +
  GROUP BY t2.TransId HAVING +
  SUM(t2.ExtPrice) <> MAX(t1.NetAmount)
```

You have to use the maximum of the *netamount* column instead of just listing *netamount* in the SELECT list because you're not grouping by *netamount*. If R:BASE saw a bare column name in the SELECT list, it would expect to see that column in the GROUP BY clause. Therefore, you have to use a SELECT function. Since there's only one row, you can use any of these functions and still get the same answer: AVG, MIN, or MAX.

Tally Multiple Columns and Save the Result

To count the number of rows for multiple columns, use the SELECT function COUNT. It counts the rows in each group of columns specified in the GROUP BY clause. Here's an example using *ConComp*:

```
SELECT EmpCity, EmpState, COUNT(*) FROM Employee +
  GROUP BY EmpCity, EmpState
```

The result is a table like this:

EmpCity	EmpState	COUNT(*)
Duvall	WA	1
Redmond	WA	2
Seattle	WA	4
Woodinville	WA	1

Multi-Column, Multi-Table Tally

You can have both multiple columns and multiple tables in your counts. The WHERE clause links the tables by specifying the linking columns, and COUNT counts the rows in each group specified in the GROUP BY clause.

```
SELECT t2.EmplName, t1.EmpId, +
  t2.EmpState, SUM(t1.Bonus), COUNT(*) +
  FROM SalesBonus T1 LEFT OUTER JOIN +
  Employee T2 ON +
  (t1.EmpId = t2.EmpId) +
  GROUP BY +
  t2.EmpState, t1.EmpId, t2.EmplName
```

The following table was produced by using the *ConComp* database that comes with R:BASE. It shows that you can set up the count and include extra information like a sum for each group. The following SELECT command counts the employee name (E*mplName*), state (E*mpState*), and identification number (E*mpId*) and includes the total and number of their bonuses.

t2.EmplName	t1.EmpId	t2.EmpState	SUM(t1.Bonus)	COUNT(*)
Wilson	102	WA	$743.50	4
Hernandez	129	WA	$302.73	3
Smith	131	WA	$940.50	3
Coffin	133	WA	$27.00	1
Simpson	160	WA	$631.88	2

Display Count Only If More Than One

Sometimes you may want to tally rows for a group of columns and show only those groups where the row count is more than one. You can do this by adding a HAVING clause to limit the groups displayed.

A HAVING clause limits the groups selected by a GROUP BY clause in much the same way a WHERE clause limits the rows selected by a command.

The following *concomp* example shows how to use the HAVING clause to limit groups:

```
SELECT EmpCity, EmpState, COUNT(*) +
  FROM Employee +
  GROUP BY EmpCity, EmpState +
  HAVING COUNT(*) > 1
```

The following table shows only those groups with a row count of more than one.

EmpCity	EmpState	COUNT(*)
Redmond	WA	2
Seattle	WA	4

Save Tallies in a View or Table

To save tallies in a view, use SELECT with CREATE VIEW, as in this example:

```
CREATE VIEW TallySav (Model, Number_Orders, Total_Units) +
  AS SELECT Model,COUNT(Model), SUM(Units) +
  FROM TransDetail +
  GROUP BY Model ORDER BY 3 DESC
```

To save tallies in a table, create a table to hold the tallies:

```
CREATE TABLE TallyOut (Model TEXT 6, Number_Orders INTEGER, +
  Total_Units  INTEGER)
```

Then use SELECT with INSERT, as in this example:

```
INSERT INTO TallyOut SELECT Model, COUNT(Model), SUM(Units) +
  FROM TransDetail +
  GROUP BY Model ORDER BY 3 DESC
```

Delete Duplicates Based on a Column List

Using SQL, you can find duplicates based on a list of columns. All you need is a SELECT command that does the equivalent of a TALLY and uses a HAVING clause to select the groups that have counts greater than one. To delete the duplicates while keeping the first row in each group, use the SELECT in a DECLARE CURSOR structure.

For example, using *ConComp*, add a few duplicate rows to T*ransMaster*. Then run *sqlxdupe* (below) to delete duplicates based on the *TransId, CustId,* and *EmpId* columns:

```
-- SQLXDUPE.CMD - Delete duplicates
-- based on a set of columns.
DROP CURSOR c1
DECLARE c1 CURSOR FOR +
  SELECT TransId, CustId, EmpId +
  FROM TransMaster WHERE TransId IN +
    (SELECT TransId FROM TransMaster +
    GROUP BY TransId, CustId, EmpId +
    HAVING COUNT(*) > 1 ) +
  GROUP BY TransId, CustId, EmpId
OPEN c1
FETCH c1 INTO vTransId INDICATOR viTrans, +
  vCustId INDICATOR viCust, vEmpId INDICATOR viEmp
WHILE SQLCODE <> 100 THEN
  DELETE ROWS FROM TransMaster +
  WHERE TransId = .vTransId AND +
  CustId = .vCustId AND EmpId = .vEmpId +
  AND COUNT > 1
FETCH c1 INTO vTransId INDICATOR viTrans, +
  vCustId INDICATOR viCust, vEmpId INDICATOR viEmp
ENDWHILE
CLEAR VAR vTransId, viTrans, vCustId, +
   viCust, vEmpId, viEmp
```

Modify *sqlxdupe* to use your own columns and tables. This example also shows a good DECLARE CURSOR structure, with INDICATOR variables for each variable fetched by the FETCH command.

Replace Relational Commands with SQL

Try using SELECT with CREATE VIEW or INSERT in place of relational commands like INTERSECT, SUBTRACT, or UNION.

Use SELECT Instead of INTERSECT

This INNER JOIN replaces an INTERSECT:

```
SELECT t1.ColList, t2.ColList FROM Tbl1 t1, +
  INNER JOIN Tbl2 t2 ON (t1.LinkCol = t2.LinkCol)
```

Use SELECT Instead of SUBTRACT

This sub-SELECT replaces a SUBTRACT:

```
SELECT ColList FROM Tbl2 WHERE +
  NOT EXISTS (SELECT LinkCol FROM +
  Tbl1 WHERE Tbl1.LinkCol = Tbl2.LinkCol)
```

This LEFT OUTER JOIN replaces a SUBTRACT:

```
SELECT ColList FROM Tbl2 LEFT OUTER JOIN
  Tbl1 ON Tbl1.LinkCol = Tbl2.LinkCol
```

Use SELECT Instead of UNION

You can replace UNION with outer joins (implicit):

```
SELECT t1.ColList, t2.ColList FROM Tbl1 tl, +
Tbl2 t2 WHERE (t1.LinkCol = t2.LinkCol) +
UNION ALL SELECT t1.ColList, Constants +
  FROM Tbl1 t1 WHERE NOT EXISTS +
  (SELECT LinkCol FROM Tbl2 WHERE +
  Tbl2.LinkCol = t1.LinkCol) +
UNION ALL SELECT Constants, t2.ColList, +
  FROM Tbl2 t2 WHERE NOT EXISTS +
  (SELECT LinkCol FROM Tbl1 WHERE +
  Tbl1.LinkCol = t2.LinkCol)
```

You can replace UNION with a FULL OUTER JOIN (explicit):

```
SELECT t1.ColList, t2.ColList +
  FROM Tbl1 tl, +
  FULL OUTER JOIN Tbl2 t2 +
  ON (t1.LinkCol = t2.LinkCol)
```

All the above examples assume no null values exist in the linking columns.

8 Using Optimizing Techniques

Use the following techniques in your command files and EEPs to speed up processing:

- **Group similar commands**

  R:BASE loads into memory the information needed to process each command. If the command is already in memory, R:BASE does not need to read the disk again. For example, try grouping separate SET VARIABLE commands into one SET VARIABLE whenever possible.

- **Minimize disk access**

  Minimize disk access by using Custom EEPs, which run from the computer's memory, and by combining small command files into command blocks within a procedure file.

- **Use WHILE loops**

  Use WHILE loops instead of IF structures or GOTO processing whenever possible. All the commands contained within a WHILE loop are completely read and are retained in memory as long as the WHILE loop is processing. Use BREAK or change the condition for a natural exit rather than using GOTO to exit from a WHILE loop.

- **Use forward GOTO searches**

  Use forward GOTO searches whenever possible. When R:BASE encounters a GOTO, it performs a forward search for the matching label more efficiently than by seeking it in memory, even though the labels are retained internally.

- **Predefine variables**

  Make sure that the data type of each variable is unambiguous by explicitly assigning the data type when the variable is defined.

- **Reduce redundancy**

  Eliminate duplication of command sections where possible. If you have a set of commands duplicated in several places within a form, use a "Custom Form Action" which can be called upon at any place in the form. If you have a set of commands duplicated in several places within the entire application, use

"Stored Procedures", which can be called upon from almost any place within the application.

For multiple commands duplicated in a file, put those commands in a separate command block in the procedure file and use RUN to execute the commands where needed.

- **Eliminate rules checking**

  Set RULES OFF when not needed for data verification. This saves time by preventing R:BASE from checking the data for rule violations.

- **Experiment with Manual Table-Order Optimization**

  By default, R:BASE uses its own internal algorithm optimizer that determines the best order for joining tables. You can turn off the automatic optimizer using the MANOPT setting. MANOPT (default:OFF) disables the automatic table-order optimization that R:BASE performs when running queries. This gives maximum control over the order in which columns and tables are assembled in response to a query. With MANOPT set to ON, R:BASE uses the order of the tables in the FROM clause and the order of the columns in the column list of the SELECT clause to construct the query.

9 Summary

As you reviewed the programming examples, you may have discovered ideas for improving your applications. The power you gain from these examples comes mainly from the SELECT command. Start experimenting with it to see how much more easily you can search through your database.

The flexibility provided by R:BASE support for ANSI level 2 SQL means you can use multi-table SELECTs, sub-SELECTs, correlated sub-SELECTs, and JOINs to speed up data retrieval. The examples that illustrate these capabilities were chosen for their value in helping you obtain a thorough understanding of the advantages they offer.

In addition, continue to review the lists of optimizing techniques. When implemented along with the other tips, they can super charge your application!