Linker Systems' Guide to Writing Solid, Fast-Running SQL/C Code

By Sheldon Linker,
David B. Harrison, & Mark Jones

sol@linker.com
800-315-1174 / +1-949-552-1904
http://www.linkersystems.com

Introduction

This document is written with the hopes of improving the speed and efficiency of writing code, and the speed and efficiency of the code which is written. However, remember that in writing code, there are three factors which lead to success:

- The code must be correct.
- The code must run within the time and resource limits
- The code must be readable. Comment what's not obvious.

Contents

The Steps to Supercharging Your Program

8) Local area SQL pruning
    8.1) Combining queries
        8.1.1) Inner joins
        8.1.2) Outer joins
        8.1.3) Forked and recombined joins
        8.1.4) Insert Select
    8.2) Eliminating Fetch loops
    8.3) Breaking a conditional query up into 2 queries
    8.4) Instead of using Create Table/Insert Select/Declare, use Declare Select

9) Wide-area SQL pruning
    9.1) Dropping functions because you have the data
    9.2) Rewriting functions to take advantage of data
    9.3) Gathering required data into LOJ/Group By sets
    9.4) Using the "LOJ Select"/"LOJ selected" method
    9.5) Writing pure-C functions

10) Blocks
    10.1) Fetch
    10.2) Insert

11) After a successful sweep, do it again

12) Code reviews

13) Check the run log

A Brief Recap of Useful Portions of the Standard Library, When to Use Them, and When Not To

Feel free to skip this or any section if you already know the contents.

strcpy This is the string copy function.

Usage: **strcpy(***destination***,** *source***);**

Use it to copy a null-terminated string into a null-terminated string location. Remember that the length of the destination string location must be at least 1 byte larger than the length of the source string. So, if we use

```
strcpy(x, "Hello");
```

The length of the Hello string is 5 bytes, so the length of **x** must be at least 6 bytes.

Do not use this function to handle single characters or non-null-terminated strings.

strncpy This is the limited string copy function.

Usage: **strncpy(***destination***,** *source***,** *length***);**

Use it to copy a null-terminated string into a null-terminated string location. The length of the destination location must be at least 1 byte larger than the specified length. So, if we use

```
strncpy(x, y, 5);
```

Then x must be at least 6 bytes long. If y is longer than 5 characters, it will be truncated without any error indication.

This function can be used like **SUBSTRING**. The equivalent of

```
SET :result TO SUBSTRING(:myString, :index, :length)
```

is

```
strncpy(result, myString+index-1, length);
```

Do not use this function to handle single characters or non-null-terminated strings.

strcat This is the concatenation function, tacking one string onto the end of another.

Usage: **strcat(***destination***,** *source***);**

Remember that the destination must be at least the size of the first string, plus the size of the second string, plus a terminating byte.

strncat This is the same as **strcat**, except that a maximum number of characters (not including the null) is specified.

Usage: **strncat(***destination*, *source*, *length***);**

Remember that the size of the destination must be at least the size of the first string, plus the specified length, plus a terminating byte.

memcpy This is the non-terminated, or **CLOB/BLOB** form, much like **strcpy/strncpy**. It is used to copy a data block from one area to another, without regard to contents.

Usage: **memcpy(***destination*, *source*, *length***);**

There is a restriction that the two memory blocks (source and destination) may not overlap. memcpy, like strcpy and strncpy are very fast, and usually compile to a single instruction.

memmov This is the same as **memcpy**, except that the memory blocks may overlap. It does not usually compile to a single instruction, though.

Usage: **memmov(***destination*, *source*, *length***);**

strcmp This function compares two strings, each of which must be null-terminated. If the first string is greater than the second string, the result is positive, but not necessarily 1. If the two strings are equal, the result is 0. If the first string is less than the second string, then the result is negative, but not necessarily -1.

Usage: **result = strcmp(***first*, *second***);**

For example:

```
if (!strcmp(command,"Final"))
    Handle 'Final' command
```

strncmp This function compares two null-terminated strings, but the maximum length of the comparison, and thus the maximum effective lengths of the strings, is set by the third argument. Like **strcmp**, the result is positive, zero, or negative.

Usage: **result = strncmp(***first*, *second*, *maximum length***);**

memcmp This function compares two strings or other chunks of data. Like **strcmp** and **strncmp**, the result is positive, zero, or negative. The two items being compared will usually be character arrays, but on the AS/400 (not on PCs) can also be **_Packed** structures containing only **char**, **short**, **int**, and **long** data can be used for <, =, and > comparisons. **_Decimal**, pointer, **float**, and **double** data can be used for = and ≠ only.

Usage: *result* **= memcmp(***first*, *second*, *exact length***);**
or *result* **= memcmp(***first*, *second*, **sizeof** *first***);**

atol This takes a null-terminated character string expressing an integer value, and returns the value. It stops at the first character it sees which is not a dash or a digit.

Usage: **atol(***string***)**

For instance,

```
atol("-10")
```

has a value of -10. Note that

```
atol("-10m")
```

also has a value of -10, since processing of the string will stop between the 0 and the m.

```
atol(" 10")
```

will result in 0, because processing will stop before the blank.

isdigit
islower
isupper
isalpha

These functions may be implemented as functions, macros, or array look-ups. Thus, it is not safe to make assumptions about the number of times an argument is really used. For instance,

```
if (isdigit(*c++))
```

is not safe, because this could be a macro, in which case, it would evaluate to

```
if ((*c++)>='0' && (*c++)<='9')
```

which would give unfortunate side effects.

sprintf This is one of the most important formatting functions. It can save you the effort of horrendous sets of **strcat** or **EXEC SQL SET** operations. Use it as

sprintf(*targetString, formatString, itemsToBeFormatted***);**

The format codes of interest are:

%ld	Format a long
%9ld	Format a long to 9 digits, right justified
%-9ld	Format a long to 9 digits, left justified
%09ld	Format a long to 9 digits, zero padded
%s	Formats a null-terminated string (type **char *** or **char []**)
%9s	Formats a string to at least 9 spaces, left justified
%-9s	Formats a string to at least 9 spaces, right justified
%c	Formats a character (type **char**)
%%	Outputs a **%** symbol
otherwise	Outputs the characters as shown

For example,

```
sprintf(myField, "The answer is %ld%%.", myVal);
```

If **myVal** is 10, then **myField** is set to "The answer is 10%.", with a null-terminator.

sccanf This is also an important formatting function. It allows you to read one or more items from a string. The format is:

> result = **sscanf(***source*, *format*, *targets***);**

The format codes of interest are:

%ld	Reads a long
%s	Reads a string, up to white space or end of line
%c	Reads a character
%[*aeiou***]**	Reads a string consisting only of the indicated characters, in this case, vowels only
%[*a-z***]**	Reads a string consisting only of the indicated range, in this case, lower case letters only. Note that in EBCDIC, this is not a real range. To get lower case letters only, you would really need to code **%[a-ij-rs-z]**.
%[^.,]	Reads a string consisting of anything but the indicated characters
blank	Skips over any amount of white space.
otherwise	The indicated character is required.

The value returned is the number of items read from the string. For instance, let's say we wanted to read a contract number and an group number from a full group number. We might use:

```
if (sscanf(fullGroupNbr,"%ld-%ld",&contractNbr,
    &groupNbr)!=2)
        error code
else
     success code
```

In this case, this statement means: Read a long into **contractNbr**, then skip the dash, then read a long into **groupNbr**. If I didn't get 2 items read, do the error code; otherwise do the success code.

A Brief Recap of SQL/C Data Type Pairings

IBM/DB2 version:

CHAR	**char**	Use **=, ==**
CHAR(*n*)	**char [*n*]**	Use **memcpy, memcmp**
VARCHAR(*n*)	**char [*n+1*]**	Use **strcpy, strcmp**
INTEGER	**long**	Use **=, ==**
SMALLINT	**short**	Use **=, ==**
DECIMAL(*n*, *d*)	**decimal(*n*, *d*)**	Use **=, ==**

Unix/Oracle version:

CHAR	**char**	Use **=, ==**
CHAR(*n*)	**char [*n*]**	Use **memcpy, memcmp**
VARCHAR2(*n*)	**char [*n+1*]**	Use **strcpy, strcmp**
NUMBER(5 to **9)**	**long**	Use **=, ==**
NUMBER(3 or **4)**	**short**	Use **=, ==**
NUMBER(1 or **2)**	**char**	Use **=, ==**
other **NUMBER**	**double**	Use **=, ==**

A Brief Recap of the Order of Select Execution: (1) From (2) Where (3) Group By (4) Select (5) Distinct (6) Having (7) Order By

When planning for the efficiency of execution, remember that Select statements are executed in this order:

From — The files are opened, and the Joins are evaluated.

Where — The Where constraints are evaluated, in whatever order the optimizer likes.

Group By — The groups are formed.

Select — The items in the Select list are evaluated, and the field names, if any are assigned.

Distinct — Any sorting and culling is done after evaluation. Functions will get evaluated before this step, so make sure that functions somehow happen after this step if their execution is slow, or come up with a way of avoiding Distinct.

Having — Results are further culled here.

Order By — After everything else, the results are sorted.

This is how it's <u>supposed</u> to work. I'm not guaranteeing that it will.

Removing the Most Common Errors

1) How to check for uninitialized variables, and what to do about them

Sometimes a program will use the contents of a variable before the variable is set, or without the variable ever being set. An easy way to check for this is to go through each variable in the declaration list, and search for it. If it's initialized in the declaration, you're fine. If not, check the next usage. If it's a C or SQL statement that defines the variable, you're also fine. However, if the next use is a C or SQL statement that uses the variable, and there's no set, you've got a problem. Check regular variables, variables used as values by SQL, and variables used as null-flags by SQL.

2) How to check for "partially used" null flags, and what to do about them

Null flags are a bit trickier than regular variables or SQL value variables. Make sure that for each null variable in use, that each operation that sets the value variable also sets the null-flag variable, and that each operation which reads the value variable also reads the null-value variable.

Exception: You may reach a stage in the execution where the value can no longer possibly be null. In this case, you can stop using the null-value flag. For instance,

```
// Treat null as zero
if (nv_myValue<0) {
    v_myValue = 0;
    nv_myValue = 0;
}
```

If everywhere past this piece of code, `v_myValue` will always be non-null, then you could just as easily code

```
// Treat null as zero
if (nv_myValue<0)
    v_myValue = 0;
    // nv_myValue is no longer used
```

3) Checking SQLCA after a Fetch

A good rule of thumb is that you never know what's going to happen for sure. For instance, I've seen

```
for (;;) {
   EXEC SQL FETCH cursor INTO :variable;
   if (sqlca.sqlcode==100)
      break;
   if (sqlca.sqlcode<0)
      goto error;
```

and

```
for (;;) {
   EXEC SQL FETCH cursor INTO :variable;
   if (sqlca.sqlcode==100)
      break;
```

Both can get you into trouble. What if something unexpected happens? The following is a little more encompassing, and will get you out of possible problems. Replace the first block of code with:

```
for (;;) {
   EXEC SQL FETCH cursor INTO :variable;
   if (sqlca.sqlcode==100)
      break;
   if (sqlca.sqlcode)
      goto error;
```

and the second block with:

```
for (;;) {
   EXEC SQL FETCH cursor INTO :variable;
   if (sqlca.sqlcode)
      break;
```

The Importance of H Files

When you call a function in C, and do not define the function before you call it, C does no checking on the arguments, and assumes that the function returns int. This does not make for robust or error-free code.

So, it is better to define the routine. However, let's say I define a routine, myRoutine, to take two integer arguments, and return no value. You would normally be tempted to put:

```
        void myRoutine(long, long);
```

into your definition list in each program that uses **myRoutine**. Don't. Here's why: What if I change **myRoutine** to take a long and a short, or more or fewer arguments, or some other change? If I remember to find every single program that uses it, and make the same change everywhere, I'm fine. But if not, the routine and its callers may become out of sync. This, in turn, will lead to execution errors.

The way to handle this problem is to define the routine in an H file. This can be a common H file, such as **CPS.H**, or a routine-specific H file, such as **myRoutine.h**. Each caller to **myRoutine** should include the H file to get the definitions, using **#include**. The actual module containing **myRoutine** should also include the H file, thus insuring that everything remains in sync. Be sure to include the H file in the dependency list.

Similarly, if you have a data structure which is used in more than one module, it should be defined in an H file. If the structure need only be known to C, then use **#include** to include the H file. If the structure must also be known to SQL, then use **EXEC SQL INCLUDE** to include the H file.

How to Optimize Your Code, Using the Peep-hole Approach

The peep-hole approach means that you're just looking at a particular line of code, as well as the lines of code immediately before and after the line of code you're on. This type of work does not require any understanding of the code you're working on, and will in turn lead to code that you can more easily understand, allowing us to go on to more complicated optimizations.

1) Neaten the program up

Sounds like a lot of busy-work, right? Well yes, it does sound like busy-work, but it's not. It's actually one of the most important steps in optimization. The reason is that you and everyone else needs to be able to follow the flow of the work, and visual cues are required for a quick and complete understanding of the code. Comments alone will not do this.

There are several sub-rules to this rule:

1.1) Come up with a neat and consistent style, and stick with it.

In case you don't want to come up with your own neat and consistent style, here's mine:

Put { symbols on the same line as whatever caused them. For instance, "`if (x<2) {`". Reasoning: Don't waste a line. Wasted lines mean less code on a display page, and that means less logic on a display page, and that means more page flipping in order to see what's happening.

Use indents consistantly. I like 3 spaces, with items indented for multiline statements or clauses, and for new blocks. Reasoning: 1 or 2 spaces are two hard to distinguish. More than 3 spaces is more than needed. I wouldn't ding anyone for using 4, though. Example:

```
void main(void) {
   long aValue;
   long bValue;

   EXEC SQL DEFINE c CURSOR FOR
      SELECT ff.a
         sf.b
      FROM firstFile ff,
         secondFile sf
      WHERE ff.id=sf.id;
   EXEC SQL OPEN c;
   for (;;) {
      EXEC SQL FETCH c INTO :aValue, :bValue;
      if (SQLCA)
         break;
      myFunction("This long string will force another line",
         aValue, bValue;
   }
   EXEC SQL CLOSE c;
}
```

Don't go past column 72. Reasoning: Not everyone displays columns past 72.

Don't put more that one statement on a line. Reasoning: It makes things hard to debug. Bad example:

```
        if (x<3) break;
```

When you add comments to the right, place them in a consistent column. I like screen column 53. Also, note that starting comments with -- when in SQL means not having to spend space terminating the comment at the end of the line.

Don't use white space of more than one line.

2) Check the Create statement

There are several options that should and should not be in a create statement. For instance, determine whether your function is deterministic or not, whether it can run on multiple processors concurrently, whether is uses SQL, and whether it modifies any table, and the like. Failing to put a needed clause on the **CREATE FUNCTION** or **CREATE PROCEDURE** statement, or leaving one off can make your program much slower than it needs to be, or even cause it to fail.

3) Check the Includes

Again, avoid too much (not a big problem) or too little. Don't call in H files you don't need, and make sure to call in prototypes for anything you do use.

4) Check each variable declaration

Some of this has been covered above, but I'll cover it completely here, too.

4.1) Normal variables

For each normal variable, SQL value variable, and null-flag variable, check to see whether or not the variable is used, not used, properly used, or improperly used. Do a find for each variable declared. While you look through the places the variable is used, keep these rules in mind:

If the variable is defined but not read or written, get rid of the definition.

If the variable is defined, and written into, but not read, get rid of the definition, and the code that sets the variable.

If the variable is defined and read, or read before being written into, then you've got a bug to fix.

If the variable is defined, then set, then used, you're fine.

There's also a simple-minded approach to making sure that everything is initialized: Initialize it at declare-time.

4.2) Null variables

As above, make sure that null variables are used consistently with their value variables, set each time the value is set, and read each time the value is read. Also, when a value is always read from the same place (or only read from one place), check the field's definition. If the field is **NOT NULL**, then there's no point in using a null indicator. Certain SQL functions always return non-null, such as **COUNT**. No point in using a null for that, either. When you're stripping out unneeded null variables, you'll sometimes find yourself stripping out IFs based on those nulls, too.

4.3) Be suspicious of "char [2]"

There is only one legitimate use of `char [2]` variables, and that is to hold non-null-terminated strings defined as **CHAR(2)** in SQL. SQL **CHAR** variables should be stored in C `char` variables. This will cascade into other savings. When you change something from "`char v_field[2]`" to "`char v_field`", you can also change things like "`if (!strcmp(v_field,"Y"))`" to "`if (v_field=='Y')`".

4.4) The null block - Used or not?

In a procedure using nulls as a block, if none of your input variables can be null, and you have no output variables, then there's no point in defining or reading the null-variable block. You already know that everything in it is zero.

Note also that if you have a single argument, the null indicator for that can be treated as a `short`, rather than a structure containing a single `short`.

4.5) Replacing floating point with decimal (IBM/DB2 only)

Whenever possible, replace real and double with decimal of some sort. For instance, dollars and cents less than a million dollars should be `decimal(8,2)`, rather than `float`. If you must use floating-point, use `double`, rather than `float`, for reasons too complicated to get into.

4.6) Replacing decimal or double with integer (IBM/DB2 only)

Whenever using decimal numbers with nothing to the right of the decimal point, switch to using type long. For instance, instead of using `_Decimal(5,0)`, use `long`. The limit of long is 9 digits. So, replace `_Decimal(9,0)` and smaller with `long`, but don't replace `_Decimal(10,0)` or larger with `long`.

5) Can initial variable sets and loads be moved to the declare section?

If you find something of the form "`long x;`" in one place, and then the next use of x is something of the form "`x = 0;`" or "`x = *(long *)argv[2];`", check to see if there's any intervening decision code between the declaration and the set. Very often, you can combine the two lines. For this example, that would give you "`long x=0;`" or "`long x=*(long *)argv[2];`", respectively.

6) Protecting from null calls

Even though SQL supposedly protects us from null calls when we don't code permission to be called with null variables into our functions, the 400 lets these calls through. Thus, a function which returns a null when called with null arguments should check that first. Here's an example:

```
void myFunction(long *inParm, long *outParm, short *nInParm,
        short *nOutParm) {
    long inVal=*inParm;
    short nInVal=*nInParm;
    long outVal;
    short nOutVal;

    EXEC SQL SELECT secondary
        INTO :outVal :nOutVal
        FROM file
        WHERE primary=:inVal :nInVal;

    *outParm = outVal;
    *nOutParm = nOutParm;
}
```

should be rewritten as follows:

```
void myFunction(long *inParm, long *outParm, short *nInParm,
        short *nOutParm) {
    long inVal=*inParm;
    long outVal;
    short nOutVal;

    if (*nInParm<0) {
        // Set of outParm not needed on null
        *nOutParm = -1;
        return;
    }

    EXEC SQL SELECT secondary
        INTO :outVal :nOutVal
        FROM file
        WHERE primary=:inVal;

    *outParm = outVal;
    *nOutParm = nOutParm;
}
```

Note that there is an added check for null at the beginning, but that the Select statement doesn't need the null value as part of the search.

7) Ordering && and || to avoid executions

In C, `&` and `&&` are both "and", and `|` and `||` are both "or". However, it is important to note that both sides of `&` and `|` are always executed, and `&&` and `||` evaluate their left sides first, and then their right sides only if required. Here are some examples of why that is important to know. Consider:

```
if (x<3 & myFunction()==2)
```

Here, `myFunction` will be evaluated whether or not x<3. If `myFunction` carries out some important side-effect that must occur whether or not x<3, then this would be the right way to code it.

```
if (x<3 && myFunction()==2)
```

Here, if x≥3, the entire expression will be false, and thus **myFunction** will not be executed. If there is no requirement to call **myFunction**, then this latter form would be better. Note that coding

```
if (myFunction()==2 && x<3)
```

is equivalent to the first form, rather than the second, because the function will always be called. Thus,

&&	means	If the first is true and the second is true, return true. If the first is true and the second is false, return false. If the first is false, return false.
||	means	If the first is true, return true. If the first is false and the second is true, return true. If the first is false and the second is false, return false.

8) Use of "..." \n "..."

There is a tendency to use strings which fill fit only onto a single line. For instance,

```
strcpy(buffer, "This is the first part of a very long string.  ");
strcat(buffer, "This is the second part of a very long string.");
myFunction(buffer);
```

There is no need to do this sort of thing, as long strings can be split across multiple lines as shown below:

```
myFunction("This is the first part of a very long string.  "
    "This is the second part of a very long string.");
```

9) Replacing SQL with direct C

So far, what I've shown you will result in minor speed and size improvements. Here's where we get into major improvements. Replace items of the form

(IBM/DB2 examples)
EXEC SQL SELECT *expression* **INTO :***variable* **FROM dummyFile;**
EXEC SQL VALUES(*expression***) INTO :***variable***;**
EXEC SQL SET :*variable* = *expression*;

(Unix/Oracle example)
EXEC SQL SELECT *expression* **INTO :***variable* **FROM dual;**

with

variable = *expression*;

with the possible addition of

n*Variable* = *null expression*;

10) Replacing constant variables with constants

The C compiler will not keep multiple copies of a constant when it doesn't have to. For instance,

```
if (!strcmp(x,"Hello there") || !strcmp(y,"Hello there"))
```

There string "Hello there" is stored into your program only once, and it's address is used. Thus, there's no need to code something like

```
char helloThere[]="Hello there";

if (!strcmp(x,helloThere) || !strcmp(y,helloThere))
```

11) Replacing library calls with character compares

Very often, you'll get a parameter which came from a menu, and it has one of a set of values, and these values can be distinguished by the first character. Let's say that there's a menu which passes "Sometimes", "Always", or "Never" to our program. We could replace code as shown:

Typical method:

```
void myFunction(char *argument) {
    char value[10];

    strcpy(value, argument);
    if (!strcmp(value,"Sometimes")) {
        handle Sometimes case
    }
    else
        if (!strcmp(value,"Always")) {
            handle Always case
        }
        else {
            handle Never case
        }
}
```

Easier method, for this special case:

```
void myFunction(char *argument) {
    switch (*argument) {
        case 'S':
            handle Sometimes case
            break;
        case 'A':
            handle Always case
            break;
        default:
            handle Never case
    }
}
```

12) Replacing ganged IFs with SWITCHes or trinaries

The example above shows how to convert ganged IFs to a switch statement. However, when you're setting the same variable over and over again in a series of IFs, you can often combine these into trinaries. Take the following example:

```
if (x>=0)
    if (y>=0)
        quadrant = 1;
    else
        quadrant = 4;
else
    if (y>=0)
        quadrant = 2;
    else
        quadrant = 3;
```

Both "`if (y>=0)`" items, since they set the same variable, can be collapsed to trinaries:

```
if (x>=0)
    quadrant = y>-0 ? 1 : 4;
else
    quadrant = y>=0 ? 2 : 3;
```

We can convert this to trinaries, too:

```
quadrant = x>=0
    ? y>=0 ? 1 : 4
    : y>=0 ? 2 : 3;
```

This sort of thing can be done with strings, too. Here's the old version:

```
if (x>=0)
    if (y>=0)
        strcpy(quadrant, "first");
    else
        strcpy(quadrant, "fourth");
else
    if (y>=0)
        strcpy(quadrant, "second");
    else
        strcpy(quadrant, "third");
```

can be replaced with:

```
strcpy(quadrant, x>=0
    ? y>=0 ? "first" : "fourth"
    : y>=0 ? "second" : "third";
```

13) Dump unneeded {}s

Extra braces just take space. For instance, replace

```
if (sqlca.sqlcode) {
   break;
}
```

with

```
if (sqlca.sqlcode)
   break;
```

14) Dump moot ELSEs

Dump Else statements that don't need to be there. Here's the most common example:

```
if (sqlca.sqlcode==100)
   break;
else
   if (sqlca.sqlcode)
      goto error;
   else {
      success part
   }
```

This can more easily be written as

```
if (sqlca.sqlcode==100)
   break;
if (sqlca.sqlcode)
    goto error;
success part
```

15) Get rid of QXX routines (AS/400 only)

When common storage is used, resist the tendency to use the 400's routines designed for RPG compatibility, such as QXX retrieve and change data commands, affecting *LDA. Instead, use a common data structure (defined in an H file). Define the data structure normally in your main routine, and using "**extern**" in your subroutines. Then, link the routines together, and use C calls to get to the routines, rather than **CREATE PROCEDURE**s and **EXEC SQL CALL**s to get to the routines. Within the routines, you will be able to use the structures as directly shared storage, and not need any load or change routines. In doing this, you will see a huge speed difference, because the routines will not need to go through either SQL CALL overhead, or 400 QXX overhead.

Single Instance SQL Pruning

This section deals with taking a single SQL statement, usually a Select, and optimizing it.

1) Graphing

Select statements can often get quite complicated. There is the chance that there is something in the Select which does not need to be there. The easiest way to tell is to draw one circle for each file used, and label the center of the circle with the file code you use in the FROM clause, such as MG or COV. Next, for each relationship defined in a WHERE or ON clause, draw a line between the two circles. So, for instance, if we had a clause "**mg.groupId=cov.groupId**", we'd draw a line from the MG circle to the COV circle. Next, draw an arrow pointing to any controlled file. For instance, if I had a clause "**m.contractId=:contract**", I'd draw an arrow pointing to M. Last, for every item that appears in the Select clause, check off it's file. For instance, if I show "**F_Function(cov.coverageId), m.lastName**", I'd put a √-mark on each affected circle, COV and M, in this case. (Example attached at end.)

When you finish this graphing process, you should have a well-connected series of circles, with lines, checks, and arrows. Now, look first for any circles with no connections. Any circle with no connections is an uncontrolled join. Get rid of it. You're not using it, and it's costing you a great deal of time.

Next, look for circles with only one connection, and no checks or arrows. These are well-joined, but unused files. You can get rid of those, too. They're costing you time, but not as much.

Next, look for circles with two connections, but no checks or arrows. There is a chance for this type of file that it is unneeded. This is not to say that it is unneeded. Here are two snippets, one showing a needed connection:

```
FROM memberGroup mg,
    coverage cov,
    member m
WHERE mg.groupId=cov.groupId AND
    cov.memberId=m.memberId
```

Here, COV is needed to form the connection between MG and M. However, in

```
FROM coverage cov,
    memberGroup mg,
    fullGroupView fgv
WHERE cov.groupId=mg.groupId AND
    mg.groupId=fgv.groupId
```

the MG file is completely unneeded (unless it appears in some other clause). You can get rid of it and use

```
FROM coverage cov,
    fullGroupView fgv
WHERE cov.groupId=fgv.groupId
```

2) Distinct

Any time the word **DISTINCT** appears in a select, you should be highly suspicious. Let me give you a good distinct first. Let's say we want a list of every state the company operates in. The following would reasonably use Distinct for the job:

```
SELECT DISTINCT s.name
FROM plan p,
    state s
WHERE p.stateId=s.stateId
```

Almost every other usage of Distinct arises out of an uncontrolled join, or a poorly controlled join. It is much more efficient to find a way to control joins during the execution of the From or the Where clause, than it is to wait until the execution of the Distinct clause, especially if functions are involved.

In order to avoid the Distinct clause, it is sometimes helpful to "precook" a file. Here's an example: To find the latest item in the **GroupPlan** file, we want the one with the latest effective date on or before today. But, let's say I just want the plan state. It doesn't really matter, right? Well there are a few ways to do this:

```
SELECT DISTINCT fgv.fullGroupName,
    F_MessyFunction(fgv.groupId),
    s.name
FROM fullGroupView fgv,
    groupPlan gp,
    plan p,
    state s
WHERE fgv.groupId=gp.groupId AND
    gp.planId=p.planId AND
    p.stateId=s.stateId
```

Here, you'll get just one state per group, but it may take a very long time. Another method is:

```
SELECT fgv.fullGroupName,
    F_MessyFunction(fgv.groupId),
    s.name
FROM fullGroupView fgv,
    plan p,
    state s
WHERE p.planId=F_GetGroupPlanID(fgv.groupId) AND
    p.stateId=s.stateId
```

Here, the **F_GetGroupPlanID** function is likely to take a very long time, since it will start a new sub-search for each item. However, if we move the function's select into our select, we get something like this:

IBM/DB2 version:

```
SELECT fgv.fullGroupName,                        -- Retrieve my data
    F_MessyFunction(fgv.groupId),
    s.name
FROM fullGroupView fgv
JOIN (                                           -- Do current plan search
    SELECT gp.groupId
        MAX(gp.groupPlanId) groupPlanId
    FROM groupPlan gp
    WHERE gp.effectiveDate<=CURRENT DATE
    GROUP BY gp.groupId) gp2 ON
```

```
          fgv.groupId=gp2.groupId
       JOIN groupPlan gp3 ON
          gp2.groupPlanId=gp3.groupPlanId
       JOIN plan p ON
          gp3.planId=p.planId
       JOIN state s ON
          p.stateId=s.stateId
```

Unix/Oracle version:

```
       SELECT fgv.fullGroupName,                    -- Retrieve my data
          F_MessyFunction(fgv.groupId),
          s.name
       FROM fullGroupView fgv,
          (SELECT gp.groupId                        -- Do correct plan search
                MAX(gp.groupPlanId) groupPlanId
             FROM groupPlan gp,
             WHERE gp.effectiveDate<=SYSDATE
             GROUP BY gp.groupId) gp2,
          groupPlan gp3,
          plan p,
          state s
       WHERE fgv.groupId=gp2.groupId AND
          gp2.groupPlanId=gp3.groupPlanId AND
          gp3.planId=p.planId AND
          p.stateId=s.stateId
```

Believe it or not, this last method will be extremely fast, since the second select will execute first (since it doesn't have any dependencies on the outside Select), and then the effectively-keyed linkage will also happen very quickly. I've used Inner Joins here for simplicity of understanding, but simple joins through a Where clause would work, too.

3) Replacing temp files or multiple joins with WITH (The first part of this holds for all systems, but the "WITH" part holds for IBM/DB2 only)

Let's say that we had a temporary item, that was needed in two places. Here's an example:

```
       SELECT a.firstSum,
          a.secondSum,
          b.firstSum,
          b.secondSum
       FROM mainFile mf
             JOIN (
                SELECT SUM(aa.first) firstSum,
                   SUM(aa.second) secondSum,
                   aa.groupingField
                FROM summableFile aa
                GROUP BY aa.groupingField) a ON
                mf.key=a.groupingField,
          secondaryFile sf
             JOIN (
                SELECT SUM(bb.first) firstSum,
                   SUM(bb.second) secondSum,
                   bb.groupingField
                FROM summableFile bb
                GROUP BY bb.groupingField) b ON
                sf.key=b.groupingField
       WHERE mf.link=sf.link
```

You may quickly notice that we could save some time by using a temp file:

```
DROP TABLE qTemp/myTemp;
CREATE TABLE qTemp/myTemp(
    firstSum INTEGER,
    secondSum INTEGER);

INSERT INTO qTemp/myTemp
        SELECT SUM(first),
            SUM(second),
            groupingField
        FROM summableFile
        GROUP BY groupingField;

SELECT a.firstSum,
    a.secondSum,
    b.firstSum,
    b.secondSum
FROM mainFile mf
        INNER JOIN qTemp/myTemp a ON
            mf.key=a.groupingField,
    secondaryFile sf
        INNER JOIN qTemp/myTemp b ON
            sf.key=b.groupingField
WHERE mf.link=sf.link
```

This has the advantage of only computing the temporary table once. However, we can further simplify in this vein by using a WITH-temporary:

```
WITH myTemp AS (
        SELECT SUM(first),
            SUM(second),
            groupingField
        FROM summableFile
        GROUP BY groupingField)
SELECT a.firstSum,
    a.secondSum,
    b.firstSum,
    b.secondSum
FROM mainFile mf
        JOIN myTemp a ON
            mf.key=a.groupingField,
    secondaryFile sf
        JOIN myTemp b ON
            sf.key=b.groupingField
WHERE mf.link=sf.link
```

This form executes as a single SQL, and has the added advantage of a single optimizer pass. The temporary, **myTemp**, will likely be computed and stored in memory, and never written to the disk.

4) When to use CHAR instead of VARCHAR or VARCHAR2

When a field is always the same length, use **CHAR** instead of **VARCHAR** or **VARCHAR2**. For instance, let's say that we have a field that's always two initials, first and last. Since this field is always 2 characters, we should store it as **CHAR(2)**, rather than **VARCHAR(2)**. The reason for this is that **VARCHAR** also stores a length field in each record, whereas **CHAR** doesn't.

5) Don't use NUMBER unless you have to (IBM/DB2 only)

The **NUMBER** type is stored as characters, but **DECIMAL** is stored as zoned decimal, which is usually shorter. **INTEGER** is stored as 4 bytes, and **SHORTINT** is stored as 2 bytes. Size isn't a big deal on systems of this size, but smaller indices are faster than larger ones.

6) Don't check for what can't happen

This sounds pretty simple on the face of it, but there's a lot of it going on. Think about each IF. Here's an example of the type of thing we check for on occasion:

```
if (nv_var<0)
    v_left = 1;
else
    if (nv_var>=0)
        v_right = 1;
```

Here, if **nv_var** is not <0, then it must be ≥0, so don't check that condition.

Less obvious, here's another example:

```
SELECT COUNT(*)
INTO :count :nCount
FROM file
```

Since **COUNT** can't be null, there's no point in carrying a null variable.

7) Avoiding sub-selects and replacing them

There's a certain cost in time each time a Select is executed. Here are some tips on keeping down the number of selects:

7.1) Selects within a Select

When you do a Select within a Select, and the inner Select uses a variable from the outer Select, then that inner select must be executed for each outer row. Here's an example:

```
SELECT of.matchMark
FROM outerFile of
WHERE of.stamp=(
    SELECT MAX(if.stamp)
    FROM innerFile if
    WHERE of.key=if.key)
```

Since selection of the data from the **innerFile** depends on the current key of the **outerFile**, the inner Select must be used repeatedly. However, if we factor this out, we can make each Select executable only once:

(IBM/DB2 version)
```
SELECT of.matchMark
FROM outerFile of
JOIN (
    SELECT MAX(if.stamp),
        if.key
    FROM innerFile if
    GROUP BY if.key) ON
    of.key=if.key
```

(Unix/Oracle version)
```
SELECT of.matchMark
FROM outerFile of,
    (SELECT MAX(if.stamp),
          if.key
        FROM innerFile if
        GROUP BY if.key)
WHERE of.key=if.key
```

Here, nothing in the inner Select depends on the values of the outer Select.

7.2) With cursor loops

When doing **UPDATE**s or **DELETE**s, there is a tendency to use Selects in the Where clause. For instance,

```
DELETE FROM file
    WHERE key IN(
        SELECT x
        FROM y
        WHERE flag='Y')
```

Seems pretty simple, right? Well, yes, it is simple; but the optimizer just doesn't get it. Believe it or not, the following executes much faster:

```
EXEC SQL DEFINE CURSOR i AS
    SELECT x INTO :x
    FROM y
    WHERE flag='Y';
EXEC SQL OPEN i;
for (;;) {
    EXEC SQL FETCH i INTO :x;
    if (sqlca.sqlcode)
        break;
    EXEC SQL DELETE FROM file
        WHERE key=:x;
}
EXEC SQL CLOSE i;
```

A word of warning here: This rule is usually true. When making this change, check execution time to see if there was improvement. If not, go back to the shorter form.

8) Using Where Current Of

If you need to read a series of records from a file, and then update and/or delete some of them, it is much better to use **WHERE CURRENT OF** than to use a computed **WHERE**. Here is an example of the hard way:

```
EXEC SQL DEFINE CURSOR i AS
    SELECT key, whatever
    INTO :key, :whatever
    FROM file;
EXEC SQL OPEN i;
for (;;) {
    EXEC SQL FETCH i INTO :key;
    if (SQLCA)
        break;
    if (Function(whatever))
        EXEC SQL UPDATE file
            SET field=:value
            WHERE key=:key;
}
EXEC SQL CLOSE i;
```

Now, if we use **WHERE CURRENT OF**, this simplifies:

```
EXEC SQL DEFINE CURSOR i AS
    SELECT whatever
    INTO :whatever
    FROM file;
EXEC SQL OPEN i;
for (;;) {
    EXEC SQL FETCH i INTO :key;
    if (SQLCA)
        break;
    if (Function(whatever))
        EXEC SQL UPDATE file
            SET field=:value
            WHERE CURRENT OF i;
}
EXEC SQL CLOSE i;
```

Local Area SQL Pruning

Once your C and SQL statements are all taken care of individually, it's time to start on a more important process — that of collapsing the program. One of the most productive ways to do this is to try to merge nearby SQL statements into one another.

1) Combining queries

When you find two queries next to each other, try to combine them. There follows some examples.

1.1) Inner joins

Here's the simplest example commonly found:

```
EXEC SQL SELECT groupId INTO :groupId :nGroupId
    FROM memberGroup
    WHERE groupNbr=:groupNbr;

EXEC SQL SELECT corpGroupId INTO :corpGroupId :nCorpGroupId
    FROM corpGroup
    WHERE groupId=:groupId :nGroupId;

use :corpGroupId :nCorpGroupId
```

This is a form where we're going one step at a time to gather data we need. Sometimes, this is masked by C statements, making it look like there's an economy of searching going on:

```
EXEC SQL SELECT groupId INTO :groupId :nGroupId
    FROM memberGroup
    WHERE groupNbr=:groupNbr;

if (!SQLCA && :nGroupId>=0) {

    EXEC SQL SELECT corpGroupId INTO :corpGroupId :nCorpGroupId
        FROM corpGroup
        WHERE groupId=:groupId;

    if (!SQLCA && :nCorpGroupId>=0) {

      use :corpGroupId

    }
}
```

Either way, statements like this can be combined to run faster overall:

```
EXEC SQL SELECT cg.corpGroupId
    INTO :corpGroupId :nCorpGroupId
    FROM memberGroup mg,
        corpGroup cg
    WHERE groupNbr=:groupNbr AND
        mg.groupId=cg.groupId;
```

1.2) Outer joins

Sometimes, you may want values returned in the above instance, whether or not there is a matching secondary item. In the above example, this would mean that we'd want the group ID whether or not a corp group ID was available. To do this, we'd just join to the second file using an outer join:

IBM/DB2: This appears replacing the implicit join or **JOIN** with **LEFT OUTER JOIN**

```
EXEC SQL SELECT mg.groupId,
         cg.corpGroupId
     INTO :groupId :nGroupId,
         :corpGroupId :nCorpGroupId
     FROM memberGroup mg
     LEFT OUTER JOIN corpGroup cg ON
         mg.groupId = cg.groupId
     WHERE groupNbr = :groupNbr;
```

Unix/Oraclt: This appears adding **(+)**

```
EXEC SQL SELECT mg.groupId,
         cg.corpGroupId
     INTO :groupId :nGroupId,
         :corpGroupId :nCorpGroupId
     FROM memberGroup mg,
         corpGroup cg
         mg.groupId = cg.groupId
     WHERE groupNbr=:groupNbr AND
         mg.groupId=cg.groupId(+);
```

1.3) Forked and recombined joins

There are times when we have a search that looks something like this:

```
Carry out first search method.
If that was unsuccessful,
    Carry out second search method.
    If that was unsuccessful,
        Carry out third search method.
If any method was successful,
    Use the results of the successful search to get the next item.
```

A good example of such code is: Try to find a current or past coverage. If no current or past coverage, try to find a future coverage. If no past, present, or future coverage, try to find a deleted coverage. If something was found, get the related data. This might look something like this:

```
SELECT MAX(coveragePeriodId) INTO :id :nId
FROM coveragePeriod
WHERE coverageVersion = :version AND
   effectiveDate <= CURRENT DATE AND
   deleteRequest = 'N';

if (SQLCA==100) {

   SELECT MIN(coveragePeriodId) INTO :id :nId
      FROM coveragePeriod
      WHERE coverageVersion = :version AND
         -- effectiveDate is guaranteed to be in the future
         deleteRequest = 'N';
```

```
        if (SQLCA==100)

            EXEC SQL SELECT MAX(coveragePeriodId) INTO :id :nId
                FROM coveragePeriod
                WHERE coverageVersion = :version;
                    // deleteRequest guaranteed to be Y

    }

    if (!SQLCA)

        EXEC SQL SELECT lastUpdate INTO :lastUpdate
            FROM coveragePeriod
            WHERE coveragePeriodId = :id;
```

Whew! A big example, but one I need for the example below. Now, let's combine these provisional sub-searches into Left Outer Join searches. Then, I'll use Coalesce to join up at the end. There'll be more notes at the end:

(IBM/DB2)

```
    EXEC SQL SELECT cp7.lastUpdate INTO :lastUpdate :nLastUpdate
        FROM dummyFile
        LEFT OUTER JOIN (
            SELECT MAX(cp1.coveragePeriodId) coveragePeriodId
            FROM coveragePeriod cp1
            WHERE cp1.effectiveDate<=CURRENT DATE AND
                cp1.deleteRequest='N') cp2
        LEFT OUTER JOIN (
            SELECT MIN(cp3.coveragePeriodId) coveragePeriodId
            FROM coveragePeriod cp3
            WHERE cp3.deleteRequest='N') cp4
        LEFT OUTER JOIN (
            SELECT MAX(cp5.coveragePeriodId) coveragePeriodId
            FROM coveragePeriod cp5) cp6
        JOIN coveragePeriod cp7 ON
            COALESCE(cp2.coveragePeriodId,cp4.coveragePeriodId,
                cp6.coveragePeriodId)=cp7.coveragePeriodId;
```

(Unix/Oracle)

```
    EXEC SQL SELECT cp7.lastUpdate INTO :lastUpdate :nLastUpdate
        FROM dual,
            (SELECT MAX(cp1.coveragePeriodId) coveragePeriodId
                FROM coveragePeriod cp1
                WHERE cp1.effectiveDate<=CURRENT DATE AND
                    cp1.deleteRequest='N') cp2,
            (SELECT MIN(cp3.coveragePeriodId) coveragePeriodId
                FROM coveragePeriod cp3
                WHERE cp3.deleteRequest='N') cp4,
            (SELECT MAX(cp5.coveragePeriodId) coveragePeriodId
                FROM coveragePeriod cp5) cp6,
            coveragePeriod cp7
        WHERE doal.dual=TO_CHAR(cp2.coveragePeriodId)(+) AND
            doal.dual=TO_CHAR(cp4.coveragePeriodId)(+) AND
            doal.dual=TO_CHAR(cp6.coveragePeriodId)(+) AND
            NVL(cp2.coveragePeriodId,NVL(cp4.coveragePeriodId,
                cp6.coveragePeriodId))=cp7.coveragePeriodId
```

Ok, let me explain what you see here. I have the three may-hit searches here, each attached as Left Outer Joins. Since LOJs must attach to an existing record, I've used a

From table as **dummyFile** or **dual**, so that I will have a record to start working with. Once these three sub-searches are done, I need to take the best of their results, and use that to proceed to the real search. I find the best **coveragePeriodId** using **COALESCE** best **NVL**. Arrange the arguments in most-to-least important order. Then, use that to match as the search criterion last.

1.4) Insert Select

Whenever you find yourself using **SELECT** followed by **INSERT VALUES**, see if there's any way you can replace it with **INSERT SELECT**.

2) Eliminating Fetch loops

When you have something of the form,

```
INSERT INTO tempfile SELECT whatever
DECLARE CURSOR i FOR SELECT whatever FROM tempfile;
for (;;) {
    FETCH i INTO list;
   c and SQL statements, including
        UPDATE tempfile SET whatever = whatever WHERE whatever
}
```

Make every effort into moving the update data into the main **INSERT SELECT** statement.

3) Breaking a conditional query up into 2 queries (IBM/DB2 only — Oracle seems to figure this one out on its own)

There's a method which uses more code, but runs faster. You can break up statements of the form

```
EXEC SQL SELECT whatever
  FROM whatever
  WHERE conditions AND
    (:parameter :nParameter IS NULL OR
      :parameter = field);
```

(Note that I did not code **:nParameter** a second time because I already know it's not null in the latter case. I could have also replaced "**:parameter :nParameter IS NULL**" with "**:nParameter<0**".) The larger, faster version of this is:

```
if (nParameter<0)
    EXEC SQL SELECT whatever
        FROM whatever
        WHERE conditions;
else
    EXEC SQL SELECT whatever
        FROM whatever
        WHERE conditions AND
            :parameter = field;
```

4) Instead of using Create Table/Insert Select/Declare, use Declare Select

Don't create a temp table unless there's some program that needs the temp table. For purposes of reporting, you can return a cursor more easily than returning a temp table. For instance, rather than coding something of the form

```
DROP TABLE temp;
CREATE TABLE temp(
    field list);

INSERT INTO temp
    SELECT whatever;

DECLARE CURSOR result FOR SELECT * FROM temp;
```

You can just code

```
DECLARE CURSOR result FOR
    SELECT whatever;
```

However, note that you will have to use **AS** field names for any expression, so that each column gets a name.

Wide-area SQL Pruning

Once the SQL has been cleaned up locally, it's time to take a global look.

1) Dropping functions because you have the data

When your procedure calls a function, there are several checks you should perform. The first, and most rewarding check, is to see if the data the function is developing is already available to the program. Let's say you're processing plans, along with related records, and you call a function to determine the plan's state. Check to see if that information is already present in the caller routine. If so, just use the data directly, without calling the function.

2) Rewriting functions to take advantage of data

There are actually two methods here. First, when a procedure calls a function to retrieve data, see if the data can be presented to the function by the procedure in a more useful way. For instance, let's say that you're calling a function, and passing a coverage ID to the function. Then, the function first issues a query to convert the coverage ID into a member ID, then does something with the member ID. If the member ID was already present in the caller function, then you should rewrite the called function, or create a second called function, where the member ID, rather than the coverage ID is passed.

What if the member ID (or any other such secondary data item) is not present in the caller routine? It turns out that making a modification to an existing query to retrieve such data is far more efficient than having an extra query in a function to do so. In this case, change both the caller and the called function to front-load as much of the work as possible.

If you perform this step repeatedly, you may find that the entire functionality of the function can be moved into the calling procedure, making the called function unneeded. Remember, memory is cheaper than execution time in almost every circumstance.

3) Gathering required data into LOJ/Group By sets

One of the ways to front-load data into a function is to join to a Group By set. Note that there are reasons to do this, and reasons not to.

If you're going to retrieve data on a large number of items, or a significant percentage of the data records, then you want to have the data searched all at once, if possible. If, however, you're just going to search for data on a particular item, then after-the-fact subqueries are better. Here's an example:

If I'm writing a report, and I will list the number of dependents of each primary enrollee in the system, then I want to search for this first. If, however, I want to search for the number of dependents on just one member, then I would not want to precompute this. Here is a sample of code that works best for a large percentage of hits:

(IBM/DB2)

```
SELECT m.whatever                          -- Member items
   c.deps                                  -- Derived dependent count
FROM member m
LEFT OUTER JOIN (
    SELECT m2.peMemberId
        COUNT(*) deps
        FROM member m2
        WHERE m2.type='D'
        GROUP BY m2.peMemberId) c ON
        m.memberId=c.peMemberId;
```

(Unix/Oracle)

```
    SELECT m.whatever                        -- Member items
        c.deps                               -- Derived dependent count
    FROM member m
        (SELECT m2.peMemberId
                COUNT(*) deps
            FROM member m2
            WHERE m2.type='D'
            GROUP BY m2.peMemberId) c
    WHERE m.memberId=c.peMemberId(+);
```

Here, the member file controls the query; but, we're also doing a secondary query to count the dependents for each member. Note that this inner query will only create temporary rows for members having dependents. So, if we want to show Null for no dependents, and a number for dependents, we'd use Left Outer Join, as shown, because we don't want the query to drop out where there are no matches. If we wanted to show 0 for no members, then we'd use "`COALESCE(c.deps,0)`" or "`NVL(c.deps,0)`" instead of "`c.deps`", to change missing items to 0. If we <u>only</u> wanted to show members with dependents in our list, then we'd use `JOIN` or implicit join instead or `LEFT OUTER JOIN` or `(+)`, so that the nonexistence of a temporary row would prevent the main record from displaying.

4) Using the "LOJ Select"/"LOJ selected" method

You can use an extrapolation of this method to get to records meeting a particular criterion. For instance, let's say we have a coverage with a coverage version, and we need the corresponding coverage period record which is not deleted, and has the latest effective date on or before today. There are two steps to this: First, to find the effective date we're going to use, and then to retrieve the data for that date:

(IBM/DB2)

```
    SELECT whatever
    FROM coverage cov
    LEFT OUTER JOIN (
        SELECT MAX(cp.effectiveDate) effectiveDate,
            cp.coverageVersionId
        FROM coveragePeriod cp
        WHERE cp.deleted='N' AND
            cp.effectiveDate<=CURRENT DATE
        GROUP BY cp.coverageVersionId) cp2 ON
        cov.currentVersion=cp2.coverageVersionId
    LEFT OUTER JOIN coveragePeriod cp3 ON
        cp2.coverageVersionId=cp3.coverageVersionId AND
        cp2.effectiveDate=cp3.effectiveDate
    WHERE whatever;
```

(Unix/Oracle)

```
SELECT whatever
FROM coverage cov,
    (SELECT MAX(cp.effectiveDate) effectiveDate,
            cp.coverageVersionId
        FROM coveragePeriod cp
        WHERE cp.deleted='N' AND
            cp.effectiveDate<=CURRENT DATE
        GROUP BY cp.coverageVersionId) cp2,
    coveragePeriod cp3
WHERE whatever AND
    cov.currentVersion=cp2.coverageVersionId(+) AND
    cp2.coverageVersionId=cp3.coverageVersionId(+) AND
    cp2.effectiveDate=cp3.effectiveDate(+);
```

Note that we don't have to check for deleted items from the `cp3` instance of the file, since checking in the `cp`/`cp2` instance already dropped out the deleteds.

5) Writing pure-C functions

I've been discussing at length how to get rid of functions. Note, however, that if a function is written in pure C, there's no point in getting rid of it, since it will have little or no effect on total query time.

Blocks

Replacing single row fetch and insert statements with multiple-row Fetch and Insert statements can drastically improve performance of server-side code. Using a block fetch also provides a safe way of limiting the number of results returned from a query.

Using multiple-row fetches and inserts reduces the number of SQL calls and the data passing overhead. After the host structure array has been filled by the FETCH, the application can loop through the data in the array or storage area to process each of the individual rows. Multiple-row inserts have the same performance benefits.

Using a multiple row fetch also allows many rows to be processed in-memory, and block inserted (using multiple-rows) afterwards.

Note: A maximum of 32,767 rows can be fetched or inserted at one time.

1) Fetch

```
/* Multiple row fetch example, looping through milesCur cursor
   getRows is the fetch limit. If changed, change struct sizes.
   It can be used to dynamically limit number of received results,
   so long as getRows < number of elements in in the data
   structure (ex: fBlock[50] and getRows<=50). */
long haveRows=0;

_Packed struct {
   long facilityID;
   char status[11];
   char enrollmentStatus[11];
   long miles;
   char facilityNumber[7];
} fBlock[50];

struct
  short nv[5];                -- One for each field
} nv_fBlock[50];

for (;;) {
   EXEC SQL FETCH FROM milesCur FOR 50 ROWS
      INTO :fBlock :nv_fBlock;
   if (sqlca.sqlcode<0) {
      logmessage("Failed block fetch");
      goto error;
   }

   /* sqlca values are lost on next SQL call. So, it is important
      to retain the number of rows fetched in a local variable */
   haveRows = sqlca.sqlerrd[2];
   for (i=0; i<haveRows; i++)
      process row i here
}
```

Note: On some systems, you can used `_Packed`. On other systems, you may have to use some pragma or other, such as `#pragma align(1)`.

2) Insert

This example assumes that the **fBlock** data structure was filled in by the previous example.

```
if (haveRows) {
   EXEC SQL INSERT INTO ttFacSrh(facilityID,
          status,
          enrollmentStatus,
          miles,
          facilityNumber)
      :haveRows ROWS
      VALUES(:fBlock :nv_fBlock)
      WITH NC;

if (sqlca.sqlcode<0) {
   logmessage("Failed block insert");
   goto error;
}
```

After a Successful Sweep, Do It Again

Again? Why? Well, the reason is that making any one of these code improvements will likely open the door for another of the code improvements. For instance, combining two queries may make variables used for passing data from one query to the next unneeded, or may make that combined query subject to combination with yet a third. Programs can often collapse to a third of their original size through this process.

Code Reviews

Each piece of code, before anyone even bothers debugging it, should be reviewed by a manager and/or your peers. In the industry, this is called a code review. Advantages of code reviews are that everyone will understand the code better, and that you will be very careful in your coding, because you would much rather be in the position of being congratulated on your code, than to be in the position of defending it.

Here is where we get to our first emotional issue. Be proud of your good code, and have no emotional investment in your crappy code. I guarantee you, you will turn out some of each, and so will everyone else. You are part of a team, and the only thing that counts is what the team produces. Someone who attends code reviews and only points out bugs, but never writes a line of code is as valuable as the one who writes the code.

Check the Run Log

Run your code through the system as a part of a persistent job, either through the "green screen", or through the "navigator". After executing each procedure, check the run log, or job log, for execution errors and warnings. Do not allow any unexpected errors or warnings to persist into the released code.

An example of an error that can get you into trouble is a conversion error in an Insert Select. Your Insert will stop at the error, and the data inserted will be incomplete. If you don't check the run log, you will check the data you find, and never realize that some records were lost.

—

A final word: I was helped on this by a number of people, and each had a new slant for the document. Much was included, and much was left out, for simplicity. Both IBM and Oracle have entire manuals on optimization you might like for further reading.