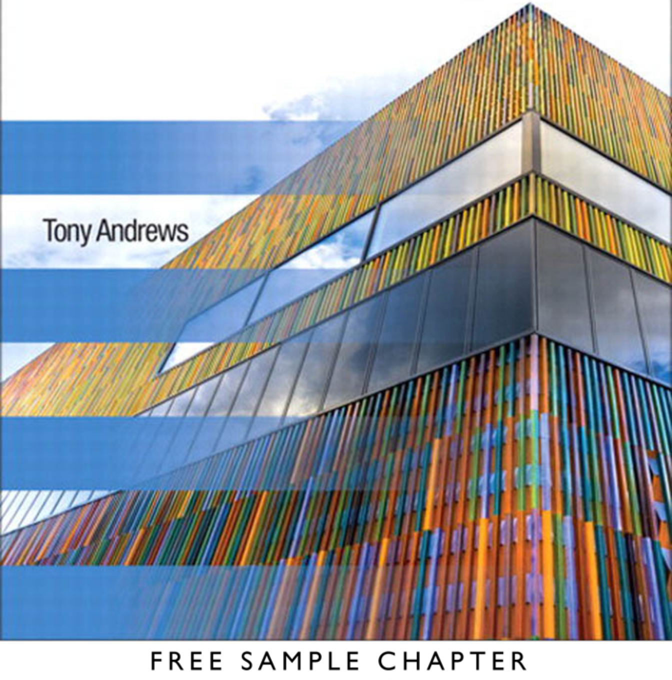
IBM PRESS

DB2 SQL Tuning Tips for z/OS Developers

Tony Andrews



This page intentionally left blank

DB2 SQL Tuning Tips for z/OS Developers

This page intentionally left blank

DB2 SQL Tuning Tips for z/OS Developers

Tony Andrews

IBM Press, Pearson plc

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

ibmpressbooks.com

IBM Press Program Managers: Steven M. Stansel, Ellice Uffer
Cover design: IBM Corporation

Published by Pearson plc
Publishing as IBM Press

The Library of Congress cataloging-in-publication data is on file.

ISBN-13: 978-0-13-303846-0
ISBN-10: 0-13-303846-7

Text printed in the United States on recycled paper at R.R. Donnelley in Crawfordsville, Indiana.

First printing October 2012

Associate Publisher
Dave Dusthimer

Marketing Manager
Stephane Nakib

Executive Editor
Mary Beth Ray

Senior Development Editor
Christopher Cleveland

Technical Reviewers
Chuck Kosin
David Simpson

Managing Editor
Kristy Hart

Designer
Alan Clements

Project Editor
Jovana Shirley

Copy Editor
Kitty Wilson

Indexer
Angela Martin

Compositor
Nonie Ratcliff

Proofreader
Sarah Kearns

Manufacturing Buyer
Dan Uhrig

Dedication

This book is dedicated to the many DB2 SQL professional developers who want to do the best they can but are not sure where to start improving performance. It is also for those who do not have the time or resources to improve their skills. I have worked with so many developers over the years and am continually impressed by their commitment and abilities to getting the jobs done and the commitment to their profession. But I have also noticed that when it comes to performance and tuning of SQL statements, programs, or applications, most are unsure of what exactly to do. Performance and tuning at times can be quite obvious and easy, and other times it can be quite involved.
My hope is that this book will help educate those in these areas and empower them for answers and direction.

Contents

Preface

Most relational tuning experts agree that the majority of performance problems among applications that access a relational database are caused by poorly coded programs or improperly coded SQL. Industry experts also note that poor performing SQL is responsible for as much as 80 percent of response-time issues. I personally agree with this. Of all the IT shops for which I have provided performance and tuning consulting work in, most of the performance issues are directly related to modifying application and SQL code, or adding and altering indexes. That is why I continually try to educate developers in the ways of SQL programming and associated performance issues. I also believe more developers should be educated in how to read and analyze DB2 Explain output. In addition I believe that every large development project involved with a RDMS should have an SQL technical expert as part of the project. There are many SQL developers in the IT industry, but based on my experience, I have found that less than 10% of them really know the performance issues involved with SQL programming or how to fix them. By having an SQL technical expert, many performance and logic issues can be caught before ever migrating to production.

The purpose of this book is to provide a reference for developers who need to tune an SQL statement or program. Most developers are too quick to blame the network, the database, the system, the high volume of transactions, etc., for the slowness of their program or application. Yet most of the time the slowness is directly related to their code. Hopefully, this book will give them something to fall back on before calling DBAs or others, and try first to improve the performance issue(s) at hand.

There are also times when SQL, along with the Explain, looks good, but is not performing efficiently. This book also provides some "Tuning Tips" to tweak the SQL into possibly optimizing differently than the optimizer had chosen. These are tips that many experts in the industry use to get poor performing SQL statements to optimize differently and possibly execute faster, especially when time is an issue in getting performance issues fixed.

I have included a list of SQL Standards and Guidelines that I have implemented in numerous shops. If there are no SQL standards set up in your IT shop, then these would be a great place to start. Many shops choose to add more items to the list specific to their applications.

It's one thing to have in place Standards and Guidelines, and then it's another to ensure they are followed. I have been in many shops where they show me their shop standards for SQL programming, COBOL programming, Java programming, etc., but have no quality assurance set up to ensure that the standards are being followed. All programs going into production should have some kind of code walkthrough or review to ensure standards are followed, and also to ensure that their SQL was efficiently written. The reviews are a way to ensure that program and SQL logic is correct and the design of the program fits its needs. There is a chapter of items that should be asked and checked when performing code walkthroughs involving SQL programming. The minimal amount of time for a code walkthrough far offsets any production performance or logic issues that may arise.

Many times developers have SQL code to tune and are not sure where to start. The first place I tell them is to look at all the predicates in each query and try to write them more efficient if possible. This book provides an appendix of poor performing SQL predicates and a more efficient way to rewrite them. It is important for developers to know whether a predicate is indexable or non indexable, and whether a predicate is Stage 1 or Stage 2. These are discussed in more detail later in this book.

Performance is not so much a DB2 issue as it is a relational issue. Developers have to be careful how they structure the queries, and how they design their application code around the queries. Database analysts and database modelers have to be careful how they design a database application. They need to take their time and do a good analysis. Performance depends on environment, applications and requirements. And performance, no matter how good it is, can always be better. Although the majority of tuning efforts are spent at the operating system and database level, the largest performance gains are obtained from tuning the application code. Applications that use SQL to retrieve data in a database are the best candidates for tuning. Since a relatively small number of SQL statements consume the majority of resources, SQL tuning often results in significant performance improvements. However, SQL tuning can at times be complex. This book provides a place to start and tries to keep simple the things that developers can do in order to get SQL driven programs and applications to perform more efficiently.

DISCLAIMER

The tuning tips and comments in this book are my own personal opinions based on many years of designing, programming, and tuning DB2 applications. Some of the tuning tips may not necessarily reflect the positions or opinions of IBM, any of their affiliates, or so called experts in the field. These tuning tips are based upon my personal experience and have all been used from time to time in applications I have been a part of to obtain better performance. I have personally used each of the 100+ tips when tuning DB2 SQL applications in order to get queries and programs to execute more efficiently.

Everyone knows the saying "It Depends." Please keep this in mind. Do not take every tip in this book and automatically expect instant performance gains. The tips are intended to give developers some direction and ideas to improve their queries or programs. Everyone should always conduct their own independent tests to verify the validity of any statements made in this book before automatically basing decisions upon those statements.

Acknowledgments

A big “thank you” goes out to the many developers and DBAs I have worked with over the years. I am a true believer in program design walkthroughs and program code walkthroughs. I have personally learned much over the years from sitting in with other co-workers and taking suggestions to improve programs, with the purpose of promoting the best and most efficient code into production. Many of those developers pushed me to put together this book as we developed some huge applications years ago. I was continually having meetings and sending emails on tips to our teams for coding techniques that would improve their query and program performance. Finally, one day they all said I need to put those tips in a book.

I would like to thank the two individuals who did the technical editing, Chuck Kosin and David Simpson. They are two of the best technicians I know in DB2, and I was honored that they became a part of this project. I really appreciated their many comments and suggestions as they made the material much better. I have known David for more than 3 years, as we’ve worked together at Themis Inc., and I can truly say he is one of the most knowledgeable and experienced DB2 people I have met in the industry. Chuck has a strong technical background and has been through technical editing before, when he worked with Craig Mullins on *DB2 Developer’s Guide*.

Additionally, many thanks to the understanding and patient folks at IBM Press who have worked with me on this edition of the book, specifically Mary Beth Ray, Steven Stansel, Christopher Cleveland, and all the editorial and production staff who were involved in producing the first edition of the book. It wasn’t easy for me to follow all the formatting and styles to help get the material together. It made me acknowledge to myself that I really do need to take some advanced Word classes.

I would like to thank my employer Themis Inc., in Westfield, New Jersey for supporting me on this project. I have never met or worked with a better group of technicians and communicators in all my years of training and consulting. They make it fun to stay on top of the newest and latest releases, and their real world experiences are invaluable when questions arise.

And, most importantly, thank you to my wife, Jan. She was the one to help build my confidence, release any fears, and push me to publish what I had learned and implemented on so many projects.

If you have any questions or comments about this text, you can contact me at tandrews@themisinc.com. You can also write to me in care of the publisher. I am open to any suggestions as my ego was parked many years ago during program walkthroughs. This book is all about educating the many developers in the ways of efficient DB2 SQL programming.

About the Author

Tony Andrews has more than 23 years' experience in the development of IBM DB2 relational database applications. Most of this time, he has provided development and consulting services to Fortune 500 companies and government agencies. Tony has written literally thousands of queries and programs during his development years, and he has also served as a DB2 database analyst. For the past 10 years, Tony has been splitting his time between consulting engagements and training. His main focus is to teach today's developers the ways of RDMS application design, development, and SQL programming—always with a special emphasis on improving performance. Tony's training, consulting, and speaking engagements are through his employer, Themis, Inc., an onsite and virtual instructor-led, hands-on IT training company recognized internationally. It offers more than 400 IT courses and helps to support International DB2 Users Group North America (IDUG NA) and Europe, Middle East, and Africa (IDUG EMEA), along with many DB2 user groups.

Tony is a current IBM champion and regular lecturer at industry conferences and local user groups. You may have seen him present at such events as IDUG NA and EMEA. He is well known for his "Top 25+ Tuning Tips for Developers" presentation.

Tony graduated from Ohio State University with a major in business and a minor in mathematical statistics. He currently resides in Dublin, Ohio.

Visit Tony's site at www.db2sqltuningtips.com, and follow him on Twitter, at www.twitter.com/tonyandrews12.

This page intentionally left blank

CHAPTER 3

SQL Standards and Guidelines

Every IT shop that has applications involving DB2 should have a set of SQL standards and guidelines for its developers to follow. This chapter is a start for developers and project managers to use as part of their development. Once you have a set of standards and guidelines, be sure to enforce them. Every program should have code walkthroughs to ensure that standards and guidelines are being followed.

The standards and guidelines that follow serve multiple purposes:

- Relate to performance
- Alleviate abends and/or production incident reporting
- Reduce I/O and CPU costs
- Increase productivity
- Improve client satisfaction
- Improve readability and understandability

The standards and guidelines that follow are grouped into two separate areas: one specific to COBOL SQL developers and the other specific to all SQL developers (no matter the language in which they are embedding their SQL code).

For COBOL Developers

1. The SQLCODE must be checked after every SQL statement. The Declare cursor statement is only a declarative, and it gets no return code from DB2. All other SQL calls get some return code. Return code data from the DB2 database system gets automatically loaded in the SQLCA communications area.
2. Every program must include the SQLCA and a DCLGEN for each table being coded against. The DCLGEN is predefined with host variables that match the column definitions. They are used to select data into, insert and update from, and serve as the host variables in any Where clause.

   If DCLGEN fields are not being used, then any program declaring variables in the code must make sure that the variable being declared *exactly* matches the definition in DB2. If it doesn't, then there is a possibility that DB2 may not choose an index to process. For example, if Column1 is defined as an Integer, then the host variable in COBOL should be defined as S9(9) comp.
3. Every program must have a consistent DB2 abend routine. For batch programs, it is easiest to have a called program that handles the display of the SQLCA fields and calls the DSNTIAR DB2 routine to display further DB2 messages. For online programs, sometimes it is good to write out the SQLCA and DSNTIAR

information to a file or table in order to fall back on errors that occur. The SQLCA contains a lot of information specific to a call that is critical to troubleshooting an error. It is important to write out all the information captured. Make sure that at least the SQLSTATE is displayed, along with the SQLCODE.

4. Never code Select * in a program. Only code for the columns needed. If a program needs all the columns, then code each one. This will prevent an abend if a new column is ever added to the table. The fewer columns being brought into the program, the more efficient the processing. (See tuning tip #3 and tuning tip #29 in Chapter 1, "SQL Optimization Top 100+.") More columns can have an effect on performance due to larger sort sizes, possible index-only processing, and join types. When DB2 looks at which join type is best, part of its analysis is the number of columns from each table being selected.

5. Make sure any columns defined as Nullable contain a null indicator host variable as part of the Select, Insert, or Update statements. This is most important in Select statements because DB2 will return an invalid -305 SQLCODE when it returns a column of null to the program and there is no null indicator specified. These null indicators must be defined in working storage as Pic S9(4) Comp.

   It is preferable to code the VALUE, COALESCE, or IFNULL SQL scalar function for any nullable columns because the program will not receive null indicators from DB2. This will alleviate -305 SQL errors where a program is not set up to handle the null indicator. It will also spare the program from having to define the null indicators in working storage.

   For example, Select COALESCE(PK_ID, 0) will return the PK_ID value if there is one, or it will return a zero if it is null. This could also be coded with the VALUE and COALESCE functions. All three would return the same result. The default specified must match the column definition. For example, since PK_ID is numeric, then the default must be a numeric—in this case, zero.

6. Any SQL statement that contains one of the following aggregate functions should have a Null-Indicator host variable as part of the select (MIN MAX, AVG, SUM). DB2 will return a null indicator to the program if it finds no data to process these functions, and the COBOL program will have to define a null indicator. If the program is not set up with a null indicator, an invalid -305 SQLCODE is returned. It is preferable to code the VALUE, COALESCE, or IFNULL function to alleviate any null indicator logic. For example:

   ```
   SELECT IFNULL(AVG(SALARY), 0)
   FROM EMP
   WHERE WORKDEPT = 'XYZ'
   ```

   This will either return the average if rows are found or a zero if no rows were met in order to calculate an average.

7. Minimize the number of times cursors are opened and closed during execution. If most of the time the open cursor and fetch retrieves only one row, then code a simple Select statement and execute the cursor processing only when a -811 (duplicate rows) SQLCODE is returned.

   Do not break up processing into multiple cursors unless performance seems to be an issue. If it takes a seven-table join, then code all seven tables in one cursor and let DB2 do the work. When you break it up, the process usually takes longer due to the extra times DB2 is sent SQL statements to process. So break up the join only when all other tuning efforts have been applied. Typically it would be more efficient to execute a seven-table join.

8. CASE expressions should always contain an ELSE clause. If none of the conditions in the CASE are met, then DB2 will return a null (via a null indicator) to the program. If the program is not set up to handle a null being returned from the CASE expression, then a -305 SQLCODE is returned, which usually causes the program to abend.
9. Always display counts for the number of Selects, Inserts, Updates, Deletes, and Open cursors that have been executed in the program. The overhead in COBOL to define the counters and increment them through the processing is minimal to the overall runtime of the program. Displaying these counts provides invaluable information when problems occur, helping a developer figure out which program to look into. Make sure the counts are displayed on every abend and at the end of processing.
10. Always display the values in host variables for a SQL statement that has an invalid SQL return code and the program goes into its abend error routine. Every developer knows how frustrating it is to have a program error out or even abend and not know what values were being processed.
11. Watch out for any SQL warnings that may occur in an SQL statement. Most programs seem to ignore warnings that many times help to detect potential problems. There are two indications of a warning message in the SQLCA: One is a positive SQLCODE other than +100; the other is a W in the SQLCA's SQLWARN0 field. When either of these exists, DB2 is issuing a warning that something worrisome happened on the prior call and that while you may have received data back, it may not be what you expected. When SQLWARN0 is a W, DB2 also provides helpful information about the problem in one or more of the other SQLWARNn fields. Also check warnings on every SQL statement return. For example:

```
Evaluate SQLCODE
  When 0
      If SQLWARN0 = 'W'
          Display '***  Warning error ***'
          Display 'Sqlstate = ' Sqlstate
      End-If
  When Other
      ...
End-Evaluate
```

12. Take advantage of the SQLERRD (3) out of the SQLCA. The third occurrence of the SQLERRD array is one of the most useful fields in the SQLCA. This field is populated after a successful insert, update, or delete with a count of the number of rows inserted, updated, or deleted. This is not populated when a mass delete with no Where logic is coded or populated due to deletes affected by delete cascade.
13. Take advantage of fetching rowsets in your cursor processing. (See tuning tip #46 in Chapter 1.) This should be strictly enforced for large cursors because of the runtime savings.

14. Apply all calculations within the COBOL code and then move the value to a host variable. Then reference the host variable in the SQL statement. Keep calculations out of SQL statements whenever possible.
15. Hard code any and all values known within an SQL statement. For example, if a program always processes the terminated rows on a table, then use the SQL statement Where Status_Code = 'T'. This is extremely helpful especially if frequency value statistics are present for the different values of Status_Code in the catalog tables. (See tuning tip #10 in Chapter 1.)

For All SQL Developers

1. All SQL join statements should have the columns from each table noted with a Correlation ID when referenced in Select, Where, Group By, or Order By clauses. A Correlation ID should be something other than a letter of the alphabet. Use something descriptive so others can understand from which table each column is coming. This makes the join logic more clear and readable.
2. Do not apply any SQL scalar functions against columns coded in the Where clause. This is especially important for columns that make up any index for a table. For example, coding Where Integer(CLM_ID) will automatically eliminate the use of the index for CLM_ID. As another example, the following:

   ```
   WHERE YEAR(HIREDATE) = 2003
   ```

   should be coded as:

   ```
   WHERE HIREDATE BETWEEN '2003-01-01' and '2003-12-31'
   ```

   to make it an indexable predicate.
3. Check your queries with the DB2 Explain tool. A Plan_Table under your ID will need to be created from the DBAs, or use the Plan_Table defined for theDB2 subsystem you are operating under. For example:

   ```
   Delete from Plan_Table
   ;

   Explain Plan Set Queryno = 11 for

    SELECT EMPNO, LASTNAME,
           FIRSTNME, WORKDEPT
    FROM EMP
    WHERE DEPTNO = ?



     ;

    Select * from Plan_Table
    Order by Queryno, Planno, Qblockno, Mixopseq
     ;
   ```

4. Watch out for Order By and Group By statements in queries. Each of these may cause a sort, which requires resource utilization. Code them only if needed. The fewer the columns and rows in a sort, the faster the sort will run, so make sure only the columns needed are coded.
5. When coding UNION statements in SQL, start with UNION ALL. By just coding UNION, a sort gets executed to eliminate duplicates, causing more resource utilization. Many times there are not duplicates, so UNION ALL should be the choice that prevents a sort from taking place. Avoid UNIONs if possible. Sometimes the logic can be rewritten using outer joins, case statements, etc.
6. Watch out for DISTINCT. This also causes a sort, which requires more runtime. Only code this when absolutely necessary. Many times a rewrite of the statement that can get the same results without the DISTINCT may run more efficiently. (See tuning tip #4 in Chapter 1.)
7. Be careful when using the CASE expression as part of the Select statement. This expression can have some considerable overhead during execution. If there are many rows being returned as part of the query, it may help to move that logic as part of your source code after each row is returned. This is especially true if your source is compiled code.
8. Do not use Select Count(*) for existence checking. Use this only when you need a total number of rows. It is best to code a Select using the FETCH FIRST 1 ROW ONLY and then check for SQLCODE = 0 or +100.
9. Always check the Performance Monitoring and Tuning guide for V9, and the Managing Performance guide for V10 for how to code (or how not to code) predicates to make them indexable and/or stage 1 versus stage 2. (See tuning tip #14 in Chapter 1.) The IBM Data Studio Visual Explain tool will also note any stage 2 predicates.
10. Watch out for <> (not equal) predicates. These predicates are non-indexable, but they are stage 1.
11. Make sure there is an understanding of inner vs. outer joins. Many times SQL is written with Table1 outer joined to Table2, and then inner joined to Table3. The inner join being coded last can offset the exceptions that took place in the outer join. Many times the three tables could all be coded with inner joins, which would run more efficiently. Outer joins are not inefficient, but if they bring in extra exception rows, and a subsequent inner join then gets rid of those extra rows, it was processing not needed.

    Also, make sure that if outer joins are coded, the program is set up to handle nulls being returned from the table where the join is not met. The VALUE, COALESCE, or IFNULL function should be used to keep DB2 from trying to send a null indicator back to the program.
12. Try to stay away from NOT logic in general. Try to keep predicates positive as much as possible. For example, the following predicate:

    ```
    WHERE NOT HIREDATE > :WS-DATE
    ```

    could be recoded as:

    ```
    Where HIREDATE <= :WS-DATE
    ```

13. When coding predicates, keep the logic away from the column to make it an indexable predicate. For example:

    ```
    WHERE SALARY * 1.10 > 100000.00
    ```

    is a non-indexable predicate and should be coded as:

    ```
    WHERE SALARY  > 100000.00 / 1.1
    ```

14. When using date-labeled durations (adding or subtracting years/months/days) to a date, it is logically important in which order they are coded and executed. For example, when adding, the order should be years first, then months, then days:

    ```
    SELECT CURRENT DATE + 2 YEARS + 3 MONTHS + 1 DAY
    ```

    When subtracting, the order should be just the opposite: days first, then months, then years:

    ```
    SELECT CURRENT DATE – 1 DAYS – 3 MONTHS – 2 YEARS
    ```

    This is important because if they are coded in a different order, the results could be incorrect! Results can be different due to date adjustments on the months. For example, subtracting 1 month from March 31 will result in February 28 or 29.

15. If you need to know the last day of a month, use the Last_Day SQL function to get it. For example:

    ```
    SELECT LAST_DAY(CURRENT DATE)
    INTO :HV1           --  Where HV1 is some Host Variable
    FROM SYSIBM.SYSDUMMY1
    ```

16. A more efficient way to get the same result as in #15 above is to use the Set statement. For example:

    ```
    SET :HV1 = LAST_DAY(CURRENT DATE)
    ```

NOTE Use the Set Host Variable assignment over the SYSIBM.SYSDUMMY1 whenever possible, especially when the statement may get executed hundreds or thousands of times within its runtime.

17. Take advantage of the many date functions in SQL instead of programming code to provide the information needed:

    Year/Month/Day returns only that portion of the date value.

    DAYOFWEEK/DAYOFWEEK_ISO returns a number (1–7), depending on whether the week begins on Sunday or Monday. DAYOFWEEK_ISO states Monday as the first day of the week.

    DAYOFMONTH/DAYOFYEAR returns the specific day number in a month (1–31) or year (1–366).

    LAST_DAY returns the last day of the month for a specific date. If the date was 10/15/2005, the date returned would be 10/31/2005.

    NEXT_DAY returns a timestamp representing the first weekday greater than the specified date. The function needs to have the weekday specified. For example:

    ```
    NEXTDAY('01/31/2005', 'MON')
    ```

returns the date of the next Monday after the date ‘01/31/2005’.

DAYS is used to get the days difference between two dates. For example:

```
SELECT DAYS(HIREDATE) - DAYS(BIRTHDATE)
```

returns the number of days difference.

WEEK returns a number (1–54) that represents the week of the year. Week 1 is the first week that contains the first day of the year.

WEEK_ISO returns a number (1–53) that represents the week of the year. Week 1 is the first week of the year that contains a Thursday, which is equivalent to the first week that contains January 4.

CHAR is used to get a date column back in a specific format (USA, ISO or JIS, EUR).

Subtracting two dates from each other returns a decimal number that has the number of years, months, and days difference between the dates:

```
SELECT DATE('2010-01-01') - DATE('2007-10-15')
FROM SYSIBM.SYSDUMMY1
```

returns 20217, which means 2 years, 2 months, 17 days. To get just the years difference, use:

```
SELECT YEAR(DATE('2010-01-01') - DATE('2007-10-15') )
```

18. Not Between is non-indexable. For example, the following predicate:

```
WHERE SALARY NOT BETWEEN 50000.00 and 100000.00
```

is a non-indexable predicate and should be coded as follows:

```
WHERE SALARY < 50000.00
OR SALARY > 100000.00
```

19. Watch out for the Like predicate. If the Like statement is a Begins With predicate, then that predicate is indexable. If the Like statement is a Contains or Ends With predicate, then it is non-indexable. For example:

```
WHERE LASTNAME LIKE 'A%'       - Begins with logic
WHERE LASTNAME LIKE '%A%'      - Contains logic
WHERE LASTNAME LIKE '%A'       - End with logic
```

20. Code only the columns needed in the Select. Extra columns can cause the optimizer to choose a different access path that may not be the best choice. Extra columns cause sorts to be more expensive and adds to transmission cost. Even one extra column (at times) can cause the optimizer to choose a different access path. Basically, the wider the result set, the more DB2 has to pull and ship.

21. Queries and/or cursors that bring back multiple rows in the result set should have For Fetch Only at the end of the query. This tells DB2 that there is no intention of updating any of the rows being fetched. Because of this, DB2 will try to avoid locking the pages and will possibly block the data rows being returned. For Read Only also does the same.

22. Code the most restrictive predicates first. This does not mean that this is the exact order in which DB2 will execute the queries. DB2 will always pick stage 1 indexable predicates first, no matter where they are coded. But within these, it is important to use the correct order.

23. Rewrite > Any and > All subqueries. For example, recode this:

```
SELECT EMPNO, LASTNAME
From Emp
Where Salary > Any
            (Select Salary
             from Emp
             Where Workdept = 'C11')
```

as follows:

```
Select Empno, lastname
From Emp
Where Salary >
             (Select Min(Salary)
              from Emp
              Where Workdept = 'C11')
```

Index