SQL Coding Quick Reference Card

DB2 Version 11

REV: April 2018

computer business international

Computer Business International, Inc.
www.cbi4you.com
866.CBI.4YOU
(Toll Free 866.224.4968)

General Rules about Predicate Evaluation

1. In terms of resource usage, the earlier a predicate is evaluated, the better.
2. Stage 1 predicates are better than stage 2 because they qualify rows earlier and reduce the amount of processing needed at stage 2.
3. When possible, try to write queries that evaluate the most restrictive predicates first. When predicates with a high filter factor are processed first, unnecessary rows are screened as early as possible. This can reduce processor cost at a later stage. However, a predicates restrictiveness is only effective among predicates of the same type and the same evaluation stage.

Order of Predicate Evaluation

The first set of rules:

1. Indexable predicates are applied first. All matching predicates on index key columns are applied first and evaluated when the index is accessed.

2. Other stage 1 predicates are applied next.

a. First, stage 1 predicates that have not been picked as matching predicates but still refer to index columns are applied to the index. This is called index screening. In general, DB2 chooses the most restrictive predicate as the matching predicate. All other predicates become index-screening predicates.

b. After data page access, stage 1 predicates are applied to the data.

3. Finally, the stage 2 predicates are applied on the returned data rows.

The second set of rules describes the order of predicate evaluation within each of the above stages:

1. All equal predicates (including column IN list, where list has only one element).

2. All range predicates and predicates of the form column IS NOT NULL.

3. All other predicate types are evaluated.

After both sets of rules are applied, predicates are evaluated in order in which they appear in the query. Because you specify that order, you have some control over the order of evaluation.

Predicate types

The type of a predicate depends on its operator or syntax, as listed below. The type determines what precessing and filtering occurs when the predicate is evaluated.

Type Definition

- **Subquery** Any predicate that includes another SELECT statement. Example: C1 IN (SELECT C10 FROM TABLE1)
- **Equal** Any predicate that is not a subquery predicate, has an equal operator and no NOT operator. Also included are predicates of the form C1 IS NULL. Example: C1=100
- **Range** Any predicate that is not a subquery predicate and has an operator in the following list: >, =>, <, <=, LIKE, or BETWEEN.
- **IN-list** A predicate of the form: column IN (list of values). Example: C1 IN (5,10,15)
- NOT Any predicate that is not a subquery predicate and contains a NOT operator. Example: COL1 <> 5 or COL1 NOT BETWEEN 10 AND 20.

In the Predicate type charts that follow, the following terms are used:

non subq means a noncorrelated subquery.
cor subq means a correlated subquery.
op is any of the operators >,>=, <,<=, ¬>, ¬<
value is a constant, host variable, or special register.
pattern is any character string that does not start with the special characters for percent (%) or underscore(_).
char is any character string that does not start with the special for percent(%) or underscore(_).
predicate is a predicate of any type.
expression is any expression that contains arithmetic operators, scalar functions, column functions, concatenation operators, columns, constants, host variables, special registers, date or time expressions.
noncol expr is a noncolumn expression, which is any expression that does not contain a column. That expression can contain arithmetic operators, scalar functions, concatenation operators, constants, host variables, special registers, date or time expressions.
an example of a noncolumn expression is:
CURRENT DATE - 50 DAYS

Indexable Predicates

Predicate Type	indexable	stage 1
COL = *value*	Y	Y
COL = *noncol expr*	Y	Y
COL IS NULL	Y	Y
COL *op value*	Y	Y
COL *op noncol expr*	Y	Y
value BETWEEN COL1 AND COL2	Y	Y
COL BETWEEN *value1 AND value2*	Y	Y
COL BETWEEN *noncol expr 1 AND noncol expr 2*	Y	Y
COL BETWEEN *expr-1 AND expr-2*	Y	Y
COL LIKE '*pattern*'	Y	Y
COL IN (*list*)	Y	Y
COL IS NOT NULL	Y	Y
COL LIKE *host variable*	Y	Y
COL LIKE UPPER ('*pattern*')	Y	Y
COL LIKE UPPER (*host-variable*)	Y	Y
COL LIKE UPPER (*SQL-variable*)	Y	Y
COL LIKE UPPER (*global-variable*)	Y	Y
COL LIKE UPPER (CAST('*pattern' AS data-type*))	Y	Y
COL LIKE UPPER (CAST(*host-variable AS data-type*))	Y	Y
COL LIKE UPPER (CAST(*SQL-variable AS data-type*))	Y	Y
COL LIKE UPPER (CAST(*global-variable AS data-type*))	Y	Y
T1.COL = T2.COL	Y	Y
T1.COL *op* T2.COL	Y	Y
T1.COL = T2 *col expr*	Y	Y
T1.COL *op* T2 col *expr*	Y	Y
COL = (*noncor subq*)	Y	Y
COL *op* (*noncor subq*)	Y	Y
COL = ANY (*noncor subq*)	Y	Y
(COL1,...COLn) IN (*noncor subq*)	Y	Y
COL = ANY (*cor subq*)	Y	Y
COL IS NOT DISTINCT FROM *value*	Y	Y
COL IS NOT DISTINCT FROM *noncor expr*	Y	Y
T1.COL1 IS NOT DISTINCT FROM T2.COL	Y	Y
T1.COL1 IS NOT DISTINCT FROM T1 *col expr*	Y	Y
COL IS NOT DISTINCT FROM (*noncor subq*)	Y	Y
SUBSTR(COL.,1,n) *op value* SUBSTR (COL,1,n) = *value*	Y	Y
DATE(COL) = *value*	Y	Y
DATE(COL) *op value*	Y	Y
YEAR(COL) = *value*	Y	Y
YEAR(COL) *op value*	Y	Y

Stage 1 Predicates

Predicate Type	indexable	stage 1
COL <> *value*	N	Y
COL <> *value*	N	Y
COL NOT BETWEEN *value 1 AND value 2*	N	Y
COL NOT IN (*list*)	N	Y
COL NOT LIKE '*char*'	N	Y
COL LIKE '*%char*'	N	Y
COL LIKE '*_char*'	N	Y
T1.COL <> T2 *col expr*	N	Y
COL op ANY (*noncor subq*)	N	Y
COL op ALL (*noncor subq*)	N	Y
COL IS DISTINCT FROM *value*	N	Y
COL IS DISTINCT FROM (*noncor subq*)	N	Y

Stage 2 Predicates

Predicate Type	indexable	stage 1
COL BETWEEN COL1 AND COL2	N	N
value NOT BETWEEN COL1 AND COL2	N	N
value BETWEEN *col expr and col expr*	N	N
T1.COL1 <> T1.COL2	N	N
T1.COL1 = T1.COL2	N	N
T1.COL1 *op* T1.COL2	N	N
T1.COL1 <> T1.COL2	N	N
COL = ALL *(noncor subq)*	N	N
COL <> *(noncor subq)*	N	N
COL <> ALL (*noncor subq*)	N	N
COL NOT IN (*noncor subq*)	N	N
COL = (*cor subq*)	N	N
COL = ALL *(cor subq)*	N	N
COL *op* (*cor subq*)	N	N
COL *op* ANY (*cor subq*)	N	N
COL op ALL (*cor subq*)	N	N
COL <> (*cor subq*)	N	N
COL <> ANY (*cor subq*)	N	N
(COL1,..COLn) IN (*cor subq*)	N	N
COL NOT IN (*cor subq*)	N	N
(COL1,..COLn) NOT IN (*cor subq*)	N	N
T1.COL1IS DISTINCT FROM T2.COL2	N	N
T1.COL1 IS DISTINCT FROM T2 *col expr*	N	N
COL IS NOT DISTINCT FROM (*cor subq*)	N	N
EXISTS (*subq*)	N	N
expression = value	N	N
expression <> value	N	N
expression op value	N	N
expression op (subq)	N	N
NOT XMLEXISTS	N	N
CASE expression WHEN expression ELSE expression END = value	N	N

SQL Coding Guidelines

1. Select only what you need. Make sure no unused columns are selected.
2. Avoid unneeded SELECT DISTINCT sorts. Ask yourself why you are getting duplicates in your results and whether the duplicates are appropriate. If your results contain duplicates that are not a true reflection of your data content, it could indicate an error in your SQL logic.
3. Avoid unnecessary constants in your SELECT list. Keep your result row length to a minimum.
4. Ensure your predicate is in the appropriate format for a search condition.
5. Make sure the length of any host variables or constants is no greater than the length attribute of the data column it is compared to.
6. Code constants in your predicate, not as a host variable. If DB2 knows the absolute value you are looking for it can perform a more accurate I/O estimate - possibly changing the access path.
7. If possible keep your comparisons positive instead of negative. Negative comparisons have a negative effect on performance.
8. Avoid unnecessary sorts; if possible match your ORDER BY to an Index. Avoid using derived result columns to order your data. Use the UNION ALL where possible, instead of UNION.
9. When possible, match the combination of WHERE clause predicate columns and ORDER BY columns to your indexes.
10. Avoid using the indexed columns of your predicate in an expression. Whenever possible apply the expression to the non-column value. Columns used in an expression (mathematical, function, labeled duration or CASE) cannot be used for index access.
11. Code your predicate as simply as possible.
12. Code as many Indexable predicates as possible.
13. Code your predicate in DB2's evaluation sequence, most restrictive predicate first, least restrictive last.
14. Code 2 or more solutions for each requirement.
15. EXPLAIN and test all SQL before embedding into your application program.
16. Code with the attitude "I can do this in one SQL statement" - do not use DB2 as just another access method; push as much of the logic into your SQL as you can. To maximize DB2 performance you need to let DB2 handle the majority of the work. The more work done in DB2, the less overhead in terms of data returned and cross-memory calls.
    - Put as much work in the SQL as feasible.
    - Perform Joins
    - Apply predicates
    - Perform calculations in the select list
    - Perform data transformation in the select list.
17. Avoid/Perform Sorts as needed.
18. Joined columns should be of the same data type and length.
19. Verify that all row filtering is applied as early as possible - in the ON clause if needed.
20. Code your Join Tables in the sequence you want DB2 to access them. The order of tables or views in the FROM CLAUSE can affect the access path. If your query performs poorly, it could be because the join sequence is inefficient. You can determine the join sequence within a query block from the PLANNO column in the PLAN_TABLE. If you think that the join sequence is inefficient, try rearranging the order of the tables and views in the FROM clause to match a join sequence that might perform better. Rearranging the columns may also cause DB2 to select the better join sequence.
21. Avoid unnecessary nested table expressions. Use them effectively to create a small subset or to influence the join sequence.
22. Match your Join and Row filtering predicates to your Indexes. You may want to request that an additional column or two be added to an existing index to achieve Index Only access.
23. Add Transitive Closure predicates where appropriate. DB2 performs predicate transitive closure only on equal and range predicates. Other types of predicates, such as IN or LIKE predicates, might be needed.
24. Verify the Join Method Chosen by DB2. If DB2 did not choose the method you intended, check your catalog statistics for Clustering Index, Cluster ratio and cardinality.
25. Review your Explain Results with your DBA. If you do not get the results you want, work with your DBA to achieve appropriate performance. With accurate statistics and appropriate SQL coding DB2 can make extremely accurate access path choices. However, if the SQL statement is poorly coded or the statistics in the DB2 catalog are inaccurate DB2 cannot optimally perform access path selection.
26. Use subqueries for what are suited - not as documentation.
27. If a subquery can be coded as a join - DO IT!
28. If there are multiple subqueries in any parent query, make sure that the subqueries are ordered in the most efficient manner. DB2 always performs all noncorrelated subquery predicates before correlated subquery predicates, regardless of coding order.
29. If your query involves column functions, make sure that they are coded as simply as possible. This increases the chances that they will be evaluated when the data is retrieved, rather that afterward.
30. When working with large objects, avoid materialization of unneeded portions of the entire object - use LOCATE and or POSSTR where appropriate.
31. Use functions in the WHERE clause with discretion - most are stage 2 predicates.
32. Use variance and standard deviation with care: The VARIANCE and STTDEV functions are always evaluated late (that is, COLUMN_FN_EVAL is blank). This causes other functions in the same query block to be evaluated late as well. For example, in the following query, the sum function is evaluated later than it would be if the variance function were not present: SELECT SUM(C1), VARIANCE(C1) FROM T1;
33. Having Clauses not used in selection paths.
34. Verify that appropriate statistics are available: RUNSTATS....KEYCARD FREQVAL
35. Consult your DBA before attempting to use optimizer HINTS.
36. Check your SQLCODEs after every SQL statement! Don't assume!
37. Sort Input files in table Cluster Sequence. If the next row you need is on the same DB2 page it may still be in the buffer and DB2 will not have to perform a physical I/O. Async vs Sync I/O.
38. Use DCLGENs. This guarantees you are using host variables that are defiined appropriately for the columns referenced.
39. When a value is fixed, code it as a literal - not as a valued host variable. This can improve the access path selection process. An absolute value filter factor is more accurate than an unknown.
40. Open Cursors as close to the First Fetch as possible. Resources are acquired at OPEN cursor time.
41. Include the QUERYNO clause in embedded SQL.
42. Avoid program, stage 3 predicates: Think DB2 - use appropriate joins. Do not query the first table, take the contents and query the next table, etc. The excessive API calls to DB2 are expensive and hinders DB2 ability to optimize your requirement.
43. Eliminate the unnecessary:
    - Avoid unnecessary counts. How many rows deleted? updated? Inserted? Use the SQLCA: SQLERRD(3) field instead.
    - Avoid unnecessary SELECTs before UPDATE. SQLCODE + 100 will tell you if it does not exist.
    - Avoid unnecessary SELECTs before DELETE. SQLCODE + 100 will tell you if it does not exist.
    - Avoid unnecessary SELECTs before INSERT. SQLCODE -803 will tell you if it already exists.
    - Avoid unnecessary random reads. If you are creating your own singleton select loop you probably should be using a cursor.
    - Avoid unnecessary existence checking. Ask yourself - Why are you checking?
44. Avoid ambiguous cursors. Declare all cursors with the FOR READ ONLY or UPDATE OF clause.
45. If possible, replace mass delete batch programs with the REORG DISCARD utility.
46. If possible, replace mass insert batch programs with the LOAD utility.
47. Access Data in a Consistent Order. This will minimize the chances of deadlocks occurring.
48. Commit work as soon as practical. This frees your held locks and minimizes the potential of other processes receiving a timeout due to your held locks.
49. Bind options should be part of your Application Design. Choose your options based on the environment you are working with and the application requirements.
50. Use Isolation Level UR where appropriate.
51. Update data at the end of the logical unit of work (as late as possible). This prevents acquiring your locks too early and holding them longer than necessary.
52. Close Cursors as soon as possible. This frees up held resources.
53. Roll back work as soon as an error is detected. This frees up held resources and guarantees data integrity.
54. Know your SQL coding rules.

SQL Coding Rules

RULE 1.	Know your data
RULE 2.	Know how your data is used
RULE 3.	Never say NEVER and Never say ALWAYS
RULE 4.	RTFM

computer business international

1.866.224.4968

DB2 is a registered trademark of IBM