SQL Programmer's Checklist
DB2 V5

18.3.1999

The Finnish DB2 Users Group,
Performance special interest group

Inquiries: anne.lesell@fi.ibm.com

MIS = Matching Index Scan

MIA = Multiple Index Scan

Sentence "... Matching Index Scan (MIS) access path is not possible" means that in these cases access path will be either Table space Scan or Non Matching Index Scan.

A. Unexpected access paths

1. Check that host variables and literals have the same data type and length as the columns

The same rule applies to join predicates as well.

Examples of different types:

Different data types, such as
INTEGER and DECIMAL or
INTEGER and SMALLINT
Different column lengths, such as
DECIMAL(7,2) and DECIMAL(6,2) or
CHAR(3) and CHAR(4)
Different precisions, such as
DECIMAL(7,3) and DECIMAL(7,4)

Example of join

```
SELECT      EMPNO, MGRNO
FROM        DSN8510.EMP, DSN8510.DEPT
WHERE       WORKDEPT = DEPTNO
```

where WORKDEPT is defined as CHAR(4) and
DEPTNO is defined as CHAR(3)

In these cases, optimizer is usually not able to use Matching Index Scan access path.

If column definition is the one that is longer or more precise, MIS is possible but data conversion is needed.

Wrong variable definitions can be avoided by always using **DCLGEN** structures as host variables.

2. Avoid scalar functions in WHERE clauses for those index columns you want to use. E.g. substring can often be replaced with LIKE or BETWEEN.

Typical scalar functions are SUBSTR, concatenation and different functions that modify data type.

In the following example MIS is not possible:

```
 MOVE  'A'     TO HV.

 SELECT        DEPTNAME
 FROM          DSN8510.DEPT
 WHERE         SUBSTR(DEPTNO,1,1) = :HV
```

Optimizer can be made to use MIS -access path if scalar functions actions are moved outside the SQL -clause or are substituted by a more efficient SQL -clause.

In this example scalar function SUBSTR can be replaced by BETWEEN or LIKE , which both are MIS.

```
 MOVE 'A'      TO START.
 MOVE 'A99'    TO STOP.

 SELECT        DEPTNAME
 FROM          DSN8510.DEPT
 WHERE         DEPTNO BETWEEN :START AND :STOP

or
 SELECT        DEPTNAME
 FROM          DSN8510.DEPT
 WHERE         DEPTNO LIKE 'A%'
```

With LIKE predicate, check that host variable or constant doesn't begin with ‘%’ or ‘_’, and that there isn’t a field procedure on the column (for example Scandinavian sort).

3. Avoid arithmetic expressions in a predicate. Perform calculations before the SQL statement and use the result in the query.

```
variable + 3    <- O.K.       (non-column expression is indexable)
column + 3      <- not  O.K. (column expression is not indexable)
```

When column expression is used in WHERE clause, optimizer is not able to use MIS for those columns.

It’s recommended to count values into host variables before SQL:

```
ADD 100        TO HV1.
```

```
SELECT        ACSTAFF
FROM          DSN8510.PROJACT
WHERE         PROJNO = :HV1
AND   ACTNO = :HV2
AND   ACSTDATE = :HV3
```

Acceptable way is to count the value with variable in SQL. In this case with index (PROJNO,ACTNO, ACSTDATE), the access path is MIS (MC=3)

```
SELECT        ACSTAFF
FROM          DSN8510.PROJACT
WHERE         PROJNO = :HV1 + 100
AND   ACTNO = :HV2
AND   ACSTDATE = :HV3
```

Foul and prohibited way is to count the value with column in SQL. In this case with index (PROJNO,ACTNO, ACSTDATE), the access path is non-matching index scan.

```
SELECT        ACSTAFF
FROM          DSN8510.PROJACT
WHERE         PROJNO – 100 = :HV1
AND   ACTNO = :HV2
AND   ACSTDATE = :HV3
```

4. Use => and =< comparison operators when comparing a host variable to two columns. When used as a range predicate for one column, BETWEEN and => =< comparison operators are usually equally efficient.

E.g. WHERE col BETWEEN :hv1 AND :hv2.

BETWEEN is usually an efficient range predicate. However, when a

variable is compared with two columns, it is better to use comparison operators >= and <=. In the following example MIS is possible:

```
SELECT        PROJNO, PROJNAME
FROM DSN8510.PROJ
```

```
WHERE         :CHKDATE >= PRSTDATE
AND   :CHKDATE <= PRENDATE
```

(Assuming index (PRSTDATE,PRENDATE))

The corresponding BETWEEN structure is a Stage 2 predicate and rules out MIS:

```
….
WHERE         :CHKDATE BETWEEN  PRSTDATE
AND PRENDATE
```

When comparing a column with two variables, BETWEEN is usually as efficient as comparison operators >= and <=. The only difference comes with interpolators defaults. In some cases DB2 chooses MIS easier for operators >=, <= than for the corresponding BETWEEN statement.

5. Avoid negations (e.g. NOT BETWEEN, NOT IN) in a predicate for index columns.

If negations (e.g. NOT BETWEEN, NOT IN) are used in WHERE clause for the index columns, MIS is not possible:

Foul and prohibited way :

```
SELECT        LASTNAME, EMPNO
FROM          DSN8510.EMP
WHERE         EMPNO NOT BETWEEN '000100' AND '000350'
```

Access path will be Table space Scan or Non Matching Index Scan.

NOT BETWEEN can be replaced with two sentences or with UNION when result table is small:

```
SELECT        LASTNAME, EMPNO
FROM          DSN8510.EMP
WHERE         EMPNO < '000100'
UNION ALL
SELECT        LASTNAME, EMPNO
FROM          DSN8510.EMP
WHERE         EMPNO > '000350'
```

Both select clauses use MIS as an access path, if the result table is small compared to the rows in the table.

NOT IN -predicate can be replaced only with IN-predicate.

6. Keep in mind that the type of a subquery (correlated/non-correlated) has a great influence on the performance of the query.

A correlated subquery refers to at least one column of the outer query. The outer query is executed first. The subquery is evaluated for each qualified row of the outer one.

When a subquery includes another subquery, then each subquery is executed for every qualifying row of outer queries.

For example:

```
SELECT EMPNO,                       <==outer query
LASTNAME,
WORKDEPT,
EDLEVEL
FROM     DSN8510.EMP CURRENT_ROW
WHERE  EDLEVEL >
             (SELECT AVG(EDLEVEL)      <==subquery
              FROM    DSN8510.EMP          (executes n times)
              WHERE WORKDEPT =                                  (
```

If the subquery has already been evaluated for a given correlation value, then the subquery might not have to be re-evaluated

UNCORRELATED SUBQUERY is not dependent on outer query and it executes from bottom to top. First execute the innermost subquery and then the result table does compare to every qualifying row of the next outer query.

For example:

```
SELECT   EMPNO,                     <== outer query
LASTNAME,
WORKDEPT,
EDLEVEL
FROM       DSN8510.EMP
WHERE    EDLEVEL >
              (SELECT AVG(EDLEVEL)    <== inner query
              FROM      DSN8510.EMP)      (executes once)
```

If the subquery result table has a lot of rows, the executing SQL-clause may be heavy. The rows of subquery are stored in the assorted result table, which is accessed via sparse index.

7. Use OPTIMIZE FOR n ROWS when a program needs only the first 'n' rows of a result table. Don't take a shortcut by using a 'n' value other than the real number of rows. Investigate the access paths with and without OPTIMIZE FOR options. Use OPTIMIZE FOR n ROWS only if it has a positive influence on the access path.

If the result table is big and only a fraction of rows are needed , adding OPTIMIZE FOR n ROWS may lead to a more efficient access path.

For example:

```
SELECT        LASTNAME, EMPNO, WORKDEPT
FROM           DSN8510.EMP
WHERE         WORKDEPT LIKE 'D%'
ORDER BY WORKDEPT
```

assumption: WORKDEPT is not a clustering index

The access path will likely be LIST PREFETCH. In OPEN CURSOR all qualifying rows are collected and sorted to a work table according to ORDER BY. Cost of sorting has become cheaper, therefore sorting is more likely to happen during access.

OPTIMIZE FOR n ROWS is made without sorting, as in the following example. The access path will be MIS with no sorting required.

```
SELECT        LASTNAME, EMPNO, WORKDEPT
FROM           DSN8510.EMP
WHERE         WORKDEPT LIKE 'D%'
ORDER BY WORKDEPT
```

OPTIMIZE FOR 2 ROWS

The OPTIMIZE clause is ignored if the query causes the whole answer table to be materialized in the OPEN phase (E.g. sorting needed for UNION, DISTINCT or ORDER BY)

8. When an OR structure cannot be converted into an IN list (for instance WHERE col < :hv1 OR col > :hv2) the access path cannot be MIS.

The access path cannot be MIS, but is at it's best Multiple Index Access, or in many cases NON-MIS or Table Space Scan. Keep in mind, that index-only access is not possible with MIA.

For example:

```
SELECT      LASTNAME, EMPNO
FROM        DSN8510.EMP
WHERE       EMPNO < '000100' OR EMPNO > '000350'
```

Or can be replaced with UNION as follows:

```
SELECT      LASTNAME, EMPNO
FROM        DSN8510.EMP
WHERE       EMPNO < '000100'
UNION ALL
SELECT      LASTNAME, EMPNO
FROM        DSN8510.EMP
WHERE       EMPNO > '000350'
```

Access path will be Matching Index Scan, if the number of rows in the result table is small, compared to the number of rows in table.

9. When fetching one row and DB2 chooses to retrieve data from a table instead of an index-only access, add a column function in the SELECT clause (if possible) to make DB2 to use the index-only access.

10. Check that the access path of column functions MIN and MAX is one-fetch index scan (I1).

One fetch index scan is the most efficient access path. It is feasible with MIN and MAX functions when:

- There is only one table in the query
- There is only one column function (either MIN or MAX)
- Either no predicate or all predicates are matching predicates for the index
- There is no GROUP BY
- There is an ascending index column for MIN and a descending index column for MAX

Column functions are on :

- The first index column if there are no predicates
- The last matching column of the index if the last matching predicate is a range type
- The next index column (after the last matching column) if all matching predicates are an equal type

11. When comparing NOT NULL column to a subquery which may get a null value as result, use COALESCE (VALUE) function in subquery.

The function ensures that the subquery will not return a null value as a result when it is compared to the not null -column.

For example:

```
CREATE TABLE T1
(C1 INTEGER NOT NULL,
 C2 INTEGER);
CREATE INDEX X1 ON T1
C1 ASC);
CREATE INDEX X2 ON T1
C2 ASC);
```

For column C1 DB2 will not use index X1 because the result of the subquery may be null value

```
SELECT .....
FROM        T1
WHERE        C1 =
(SELECT     MIN(C1)
FROM        T1
WHERE ….              );
```

For column C2 DB2 will use index X2 because the column C2 may hold null value.

```
SELECT .....
FROM        T1
WHERE        C2 =
(SELECT     MIN(C2)
FROM        T1
WHERE ….            );
```

For column C1 DB2 will use index X1 because the result of the subquery will never be null value

```
SELECT .....
FROM        T1
WHERE       C1 =
(SELECT     VALUE(MIN(C2),999999)
FROM        T1
WHERE ….            );
```

12. When using OUTER JOIN, be aware of WHERE clauses. Remember that in many cases the result table is formed first and WHERE clause is only checked afterwards. Use ON-clause instead.

In the example the FROM clause will be executed first and WHERE clause after that.

In the FROM clause there is an outer join between tables E1 and E2 (Note E1 and E2 are table expressions which define row sets with no common rows - this is for demonstrating how FULL JOIN works). The Final WHERE does not work as one would expect, because after join E1.LASTNAME columns have null values.

```
SELECT      VALUE (E1.LASTNAME,'XXX') AS E1NAME,
E1.EMPNO,
VALUE (E2.LASTNAME,'XXX') AS E2NAME,
E2.EMPNO
FROM
        ( SELECT    LASTNAME, EMPNO
        FROM              DSN8510.EMP
        WHERE             LASTNAME <= 'K99') AS E1
FULL JOIN
```

```
              ( SELECT LASTNAME, EMPNO
                FROM     DSN8510.EMP
                WHERE    LASTNAME > 'K99') AS E2
ON          E1.EMPNO = E2.EMPNO
WHERE       E1.LASTNAME LIKE '%S%'
ORDER BY    1;
```

Notice the strong influence of the WHERE-clause. Clause E2 mentioned before, produces no rows in result table. The reason is, that the FROM clause is executed first, and because it includes this JOIN predicate, which does not match, E1.LASTNAME gets a null value, and the row is excluded in the WHERE -clause.

Correction 1: Move the WHERE clause to the first table expression.

```
SELECT      VALUE (E1.LASTNAME,'XXX') AS E1NAME,
            VALUE (E2.LASTNAME,'XXX') AS E2NAME,
FROM        ( SELECT    LASTNAME, EMPNO
              FROM      DSN8510.EMP
              WHERE     LASTNAME <= 'K99'
              AND LASTNAME LIKE '%S%') AS E1
FULL JOIN   ( SELECT    LASTNAME, EMPNO
            FROM        DSN8510.EMP
            WHERE       LASTNAME > 'K99') AS E2
            ON          E1.EMPNO = E2.EMPNO
ORDER BY    1;
```

2: Check null value separately

```
SELECT      VALUE (E1.LASTNAME,'XXX') AS E1NAME,
            E1.EMPNO,
            VALUE (E2.LASTNAME,'XXX') AS E2NAME,
            E2.EMPNO
FROM        ( SELECT    LASTNAME, EMPNO
            FROM        DSN8510.EMP
            WHERE       LASTNAME <= 'K99') AS E1
FULL JOIN   ( SELECT    LASTNAME, EMPNO
            FROM        DSN8510.EMP
```

```
                WHERE       LASTNAME > 'K99') AS E2
                ON          E1.EMPNO = E2.EMPNO
                WHERE       E1.LASTNAME LIKE '%S%'
                OR          E1.LASTNAME IS NULL
    ORDER BY    1;
```

13. When joining more than 2 tables with outer join, DB2 always builds up a work table.

14. Don't check the contents of a host variable in WHERE clause.

Equally, avoid useless WHERE -clauses, you don't have to put everything in SQL-clause.

For example:

```
….
WHERE         :HV1 = :HV2
AND   NOT  :HV1 = SPACE
```

Check values of the host variables outside the SQL clause

another one:

```
…..
WHERE         COL = :HV1
AND           COL > 0
```

Check the variable HV1 in program, not in SQL. Execute SQL clause only when HV1 is greater than 0.

B. Unnecessary sorts

1. Use the order of columns in an index for the columns in the ORDER BY clause whenever possible.

If DB2 decides to use the index according the ORDER BY list, the rows are in right sequence without sorting.

The result table of the following query will be processed in order of (col1, col2). DB2 can utilize any of such indexes which begins with order by -columns . Good indexes are for instance (col1, col2) or (col1, col2, col3):

```
SELECT      col1, col2, col3
FROM        table1
WHERE       col1 BETWEEN :hv1 AND :hv2
ORDER BY    col1, col2
```

Notice that with LIST PREFETCH, ORDER BY always causes sorting. MIA and sometimes fetching rows in other sequence than clustering index make LIST PREFETCH as an access path.

2. Keep in mind that when a cursor requires sorting, all the result table rows are retrieved into a work file when the cursor is opened. A sort is required if an index is not used for ORDER BY, GROUP BY, DISTINCT, UNION or join.

DB2 tends to materialize the result table one row at a time.

If the correct order (ORDER BY, GROUP BY, DISTINCT, UNION, JOIN) is not achievable by index, or optimizer finds access path with sorting to be the most efficient one, all qualifying rows will be fetched to a temporary table during the OPEN CURSOR phase. Also the result table of non-correlated subquery is being sorted during the OPEN CURSOR phase. When the result table is big, it will take too long for the application to open the cursor.

To avoid sorting, you might consider adding a new index, changing the sort order, or limiting the size of the result table with additional restrictive clauses. OPTIMIZE FOR n ROWS might make the difference too.

In Batch programs sorting is acceptable, if the program fetches all qualifying rows. Cursor is defined by WITH HOLD -option to keep the cursor position in COMMIT. Make sure that time between commits will not be too long because of sorting.

When there is no sufficient index (col1, col2...) browsing the table goes as follows:

```
DECLARE     cursor1 CURSOR FOR
SELECT      col1, col2
FROM                table1
WHERE       col1 > :hv1
ORDER BY    col1, col2
```

```
OPEN  cursor1
```

All qualifying rows are fetched to the work table and sorted in open cursor phase, that is the result table is materialized on OPEN phase.

```
FETCH       cursor1
INTO   :col1, :col2
```

FETCH will get one row at a time from the work table materialized in open phase

3. Do not use DISTINCT if there is no need to exclude duplicates.

If table has UNIQUE-index (col1), the next query will not cause sorting. But when UNIQUE-index is like (col1,col2,col3), the result table is sorted.

```
SELECT DISTINCT col1, col2
FROM     table1
```

When used with column function , DISTINCT will not lead to sorting if the column is the first column of an index. The following query may use index (col1) or index (col1,col2,col3):

```
SELECT COUNT (DISTINCT col1)
FROM         table1
```

4. Specify a UNION with the ALL option unless eliminating of duplicate rows is necessary.

UNION (without ALL) leads always to a sort for excluding duplicates, even when result tables are exclusive and therefore duplicates don't exist.

5. Adding an extra column, which is not in the index, to ORDER BY clause, causes a sort even if there is an equal predicate on that column

Check the SQL-clauses generated by your code generator!

Example:

Index A, B, C, D
and clause:

```
…..
WHERE       A = :hv1
AND   B = :hv2
AND   C = :hv3
AND   D > :hv4
AND   E = :hv5
ORDER BY    A, B, C, D, E
```

6. If columns in ORDER BY clause are from more than one table or only from the inner table, DB2 may end up with an unnecessary sort.

Indexes:

```
T1:   col1, col2
T2:   col3

SELECT      A.col1, A.col2, B.col3, B.col4
FROM        T1 A, T2 B
WHERE       ….
ORDER BY    A.col1, A.col2, B.col3
```

C. CPU utilization

1. In mass inserts LOAD utility is more efficient than an INSERT statement. Remember, database is not available during the LOAD

Load adds the rows to the end of the table in the order of file loaded. This is why it is recommended to sort the file in the order of clustering index before loading. Table is loaded with LOG NO (changes are not logged) and the imagecopy is taken. Loading saves CPU and programming is not needed.

When using INSERT, DB2 tends to add new rows to pages they actually belong to. This insert mechanism takes more CPU than LOAD. Inserted rows are logged. PCTFREE and FREEPAGE -parameters are ignored by INSERT -clause.

2. Use FOR FETCH ONLY in a distributed environment when a SELECT statement is used only for retrieving data.

In a distributed environment BLOCK FETCH reduces the number of messages sent across the network and returns fetched rows efficiently. DB2 triggers BLOCK FETCH when it can detect that the retrieved rows can not be updated or deleted. QMF appends FOR FETCH ONLY to SELECT statements automatically, but for dynamic SQL in an application program or SPUFI the decision to use block fetch is based on the cursor declaration. If a cursor is not used for update or delete, use FOR FETCH ONLY to ensure block fetch. This causes DB2 to send a buffer full of rows to the local DB2:

```
DECLARE cur CURSOR FOR
SELECT col1, col2
FROM table1
FOR FETCH ONLY
```

3. Wild card ('%', '_') as the first character in a LIKE predicate prevents DB2 from using MIS access path, but it is still more efficient than fetching all the rows and filtering the unnecessary ones in the application program.

For example:

```
SELECT    EMPNO, LASTNAME, WORKDEPT
FROM              DSN8510.EMP
WHERE     WORKDEPT LIKE '%1'       <<--- Table space scan
```

But if the program needs all rows qualifying LIKE '%1'-predicate, using LIKE is more efficient than fetching all rows and checking weather the condition applies in program.

```
SELECT    EMPNO, LASTNAME, WORKDEPT
FROM      DSN8510.EMP
WHERE     WORKDEPT LIKE  'A%'   <<---Matching Index Scan
```

If the string includes ‘%’ or ‘_’ -signs, they can be fetched in predicate with ESCAPE-definition.

For example

```
SELECT      col1
FROM                table1
WHERE       col2 LIKE '10+% RAISE%'
            ESCAPE '+'
```

Clause fetches all rows where col2 begins with ‘10% RAISE’

4. Declare a cursor WITH HOLD option in batch programs.

When a cursor is defined with WITH HOLD -option, COMMIT doesn’t close the cursor. WITH HOLD cannot be used in CICS pseudo conversational transactions.

5. Check occurrence with cursor whenever the result table may have more than one row

Recommended method
- cursor
- EXISTS

Not recommended method
- simple SELECT
- SELECT COUNT(*)

6. List only the columns you need in SELECT clause. Colums, on which you have an equal predicate, shouldn’t be listed in SELECT clause.

7. When updating rows, avoid listing columns whose contents don’t change in SET clause. Watch particularly key, foreign key and other index columns.

8. Avoid executing extra SQL clauses.

For example:

```
SELECT      CURRENT DATE
FROM SYSIBM.SYSDUMMY1
```

+ updating table1

Correct way:

```
UPDATE      table1
SET         col1 = CURRENT DATE
….
```

D. Security

1. Update and delete rows with cursor except when unique key is known.

By using a cursor you assure, that updates and deletes affect only those rows desired. Lock management speaks in favor of using cursor.

```
DECLARE CURSOR cursor1 CURSOR FOR
SELECT      col1, col2, col3
FROM        table1
WHERE       col1 = :hv1
FOR UPDATE OF     col3

.....

UPDATE      table1
SET         col3 = :hv3
WHERE CURRENT OF cursor1
```

The exception is when you update one row and the unique key of the row is known. In this case you can update directly.

```
UPDATE      table1
SET         col3 = :hv3
WHERE        pkey = :hvpkey
```

Using the cursor for updating is slightly heavier than direct UPDATE-clause.

Notice that you cannot update with read-only cursor. Cursors including ORDER BY, UNION, UNION ALL, DISTINCT, column functions and JOIN, are read-only.

2. When using WHERE clause in creating a view, use WITH CHECK OPTION when the view is used to update a table.

WITH CHECK OPTION it is guaranteed that you cannot insert or update rows which cannot be seen through the view.

For example:

```
CREATE VIEW DSN8510.VPROJ01
          (PROJNO,PROJNAME,PROJDEP,RESPEMP)
 AS SELECT
          PROJNO,PROJNAME,PROJDEP,RESPEMP
          WHERE        PROJDEP LIKE 'AA%'
          WITH CHECK OPTION
```

In this case WITH CHECK OPTION it ensures that rows inserted or updated via view, are always applied by condition PROJDEP LIKE ‘AA%’.

WITH CHECK OPTION it cannot be defined on read-only view or if subquery is included in the view..

3. Remember to use the symbol ‘:’ preceding a host variable. It will be mandatory in DB2 V6

E. Clarity and maintenance

1. Never use SELECT * clause.

When columns needed are listed in SELECT clause, the clause is not so much dependent on table structure. Maintenance work is much easier.

Additionally SELECT -clause becomes self-documentary. The connections between columns and variables are visible in SELECT -clause.

Listing unused columns use more CPU and in some cases it causes DB2 to retrieve data from the table instead of index only access.

2. List only the columns you need in SELECT clause.
See E1.

3. In an INSERT statement, name the columns for which you provide insert values.

Benefits:

- INSERT clause is self-documentary
- Correspondence between column and data can be checked directly from INSERT clause
- INSERT clause is independent of table construction

4. In FOR UPDATE OF clause of the SELECT statement name only the columns to be updated. If you intend to delete rows, name the columns of the primary key.

When defining updating cursor, list only updated columns in FOR UPDATE OF -list. In SELECT -list updated columns need not to be mentioned. When the primary key of the table is col1 and col3 it is updated:

```
DECLARE CURSOR cursor1 CURSOR FOR
SELECT        col1, col2
FROM          table1
WHERE         col1 = :w1
FOR UPDATE OF     col3
```

If you intend to delete rows, list in FOR UPDATE OF the columns of the primary key.

```
DECLARE CURSOR cursor2 CURSOR FOR
SELECT        col1, col2
FROM          table1
WHERE         col11 = :w1
FOR UPDATE OF col1
```

5. Write an IN list instead of many OR predicates.

DB2 converts several OR -conditions automatically to an IN-list. IN -list is still more clear.

```
SELECT        col1, col2
 FROM         table1
 WHERE        col1 IN ('A1', 'B2', 'C3')
```

6. Use DCLGEN structures as host variables.

DCLGEN generates the correct type and length to the variables derived from the columns of the table. DCLGEN also provides a program with the LABEL -definitions of the columns, which enhances the clarity and maintainability of a program.

7. Name the columns in an ORDER BY clause. Exceptions: A column derived from a function or from an expression and every column in the result table of a UNION statement must be ordered by the column number unless renamed with AS clause.

Naming columns adds to the clarity and safety of a program, especially when it is necessary to make changes to it.

There are situations where a column has no name:

- SQL-clause is UNION or UNION ALL. In this case all columns of the resulting table have no name.
- ordering column is a constant or a clause.
- ordering column is a column function.
- Constants.

These ‘nameless columns’ may be referenced by AS-clause. If not the clause has to be referenced by its ordering number. AS -clause is recommended. Using the ordering number is bad programming.

8. Use descriptive column names. Comment and label the columns. Give the same name to the same piece of information at all times. (DB2 Catalog will act as a ‘poor man's data dictionary’.)

F. Locks

1. Rows fetched with read-only cursor must not be updated. Cursor must be defined with FOR UPDATE OF-option. Otherwise there will be a kangaroo-cursor i.e. jumping over rows, rereading rows and eventually looping. (see also G2).

When updating or deleting rows, use SELECT FOR UPDATE - cursor to avoid locking.

Use updating cursor even when reading only one row, which may be updated or deleted.

When PLAN or PACKAGE is bound by ISOLATION CS, the updating cursor locks the browsed page by U-lock, which prevents other simultaneous updates. For update and delete the lock is changed to X-lock.

When using SELECT FOR UPDATE -cursor deletions and updates are made a row at a time, and locks may be released in suitable intervals by COMMIT. Doing the same without cursor may update/delete a great number of rows and hereby keep many pages locked for too long.

```
DECLARE CURSOR cursor1 CURSOR FOR
SELECT        col1, col2
FROM          table1
WHERE         col1 = :hv1
FOR UPDATE OF     col3
...

UPDATE        table1
SET           col3 = :hv3
WHERE CURRENT OF cursor1
```

Observe! FOR UPDATE OF cannot be defined for read-only cursors. Cursors including ORDER BY, UNION, UNION ALL, DISTINCT, column functions and JOIN, are read-only.

2. When batch programs are executed concurrently with online programs consider retrying an operation after a deadlock or timeout.

In those batch programs that run concurrently with online transactions, return code for deadlock/timeout (-911) may be implemented to execute the interrupted LUW again a couple of times.

In Deadlock- and timeout-situations, batch programs receive a SQL -code of -911, and task is automatically rolled back to the preceding COMMIT-point. (In realtime programs SQL -code may be -913, depending of the parameters of the running system, in which case the task is not rolled back to the commit point.) If execution is continued all statements executed after the preceding COMMIT -point must be executed again

3. Define a cursor with FOR FETCH ONLY when the cursor is read-only.

Use FOR FETCH ONLY -option to define a cursor if your intention is only to read the rows. DB2 does not implement locking when consistency is otherwise guaranteed (lock avoidance). Although lock avoidance is used, warm pages are locked. That is why COMMIT is also useful in reading batch-programs.

Lock avoidance needs:

- Isolation Level (CS)
- Currentdata(no)
- Non-updating cursor
- There are no current updates to the row or to the page (cold row)

In an ambiguous case FOR FETCH ONLY -option tells DB2 that cursor is read-only.

4. Consider using UR-option in read-only cursors. This improves concurrency , but may provide the reading program with a logically inconsistent data.

G. Unpredictable results

1. Only Write ORDER BY clause when the result table must be in a certain sequence.

ORDER BY clause is the only way to ensure the order of the result table.

ORDER BY columns have to be listed on the SELECT -statement.

Do not use ORDER BY clause unnecessarily, if the order has no value, because sorting may prove to be a heavy operation.

2. Issuing INSERT, UPDATE and DELETE statements from the same application process while a cursor is open can cause unpredictable results. Updates change the positions of the row, which may result in 'kangaroo cursor' .

See F1.

3. Be aware of wrong conclusions with MIN and MAX results.

You have to be careful in interpretation of the column functions (AVG, COUNT, MAX, MIN, SUM) results.

For example

```
SELECT      MIN(DEPTNO)
FROM        TABLE1
WHERE       DEPTNO > :hv
```

If condition is not true, SQL code is 0 and the answer is one row with a NULL -value.

Here is an example of a table that has one key column and two data columns. One data column allows a NULL -value. The table was provided by some rows including null values and all possible queries were made. The result, is in the table listed below. WHERE -clause provides NULL-answers, but real NULL-values are just left out of the calculations

Function	Not found no nulls	Not found nulls	Found no nulls	Found nulls
AVG	NULL	NULL	value	value*
MAX	NULL	NULL	value	value**
MIN	NULL	NULL	value	value**
SUM	NULL	NULL	value	value***

* = sum of real values/ quantity of real values
** = real max/min value
*** = sum of real values

4. Null value fulfills only the IS NULL predicate. All other predicates exclude null values.

5. Keep in mind the execution sequence in an OUTER JOIN.

For example:

Make a report by the department about the average salary of employees hired after the year 1998. If no employees are hired for the department after 1998 the average salary is 0.

Example 1 (wrong results)

```
SELECT      DEPTNAME,
VALUE(AVG(SALARY) , 0) AS AVGSALARY
FROM        DEPT D LEFT JOIN WORKER W
ON D.DEPTNO = W.DEPTNO
WHERE       YEAR(HIREDATE) >= 1999
GROUP BY    DEPTNAME
ORDER BY    DEPTNAME
```

Results are wrong:

If department has no employees hired after 1998 the department is left out of the report (LEFT JOIN does not help because it works at the FROM level, first FROM and after that WHERE)

Example 2 (right results)

```
SELECT      DEPTNAME,
VALUE(AVG(SALARY) , 0) AS AVGSALARY
FROM        DEPT D LEFT JOIN
( SELECT    DEPTNO, SALARY
FROM        WORKER
WHERE       YEAR(HIREDATE) >= 1998 ) AS W
ON     D.DEPTNO = W.DEPTNO
GROUP BY DEPTNAME
ORDER BY DEPTNAME;
```