DB2 9 for z/OS - SQL Performance News

by Wolfgang Hini, HiPerformance Software GmbH, 26.07.2010

Content

DISTINCT Sort avoidance with non-unique Index

Prior to DB2 9, DISTINCT can only use a unique index to avoid a sort, where GROUP BY can also use a duplicate index for sort avoidance.

Example:
SELECT DISTINCT LASTNAME FROM EMP

With V9 no sort is performed to eliminate the duplicates.

GROUP BY collapsing

In V8 grouping is done after sort input processing. In V9, DB2 applies the group collapsing optimization for GROUP BY queries without a column function and for DISTICT. The improvement is fewer workfile getpages and less cpu time.

Note: THE GROUP BY and DISTINCT SORT avoidance is available in compatibility mode and requires a REBIND.

Global query optimization

The purpose of the enhancement to global query optimization is to improve query performance by allowing the DB2 V9 optimizer to generate more efficient access paths for queries that involve multiple parts. The changes are within the DB2 optimizer and the DB2 runtime components. There is no external function. However, DB2 provides details for the way in which a query that involves multiple parts is performed. Also, since the way in which a query with multiple parts is performed is no longer fixed to the way in which the query was coded, the EXPLAIN output is modified to make it easier to tell what the execution sequence is for these types of queries (PARENT_QBLOCKNO). Global query optimization addresses query performance problems that are caused when DB2 breaks a query into multiple parts and optimizes each of those parts independently. While each of the individual parts may be optimized to run efficiently, when these parts are combined, the overall result may be inefficient.

Example:

```
select * from dept t1 where exists
 (select * from emp t2 where t1.deptno = t2.workdept and t2.lastname like 'M%')
```

Prior to V9, DB2 breaks this query into two parts: the correlated subquery and the outer query. Each of these parts is optimized independently. The access path for the subquery does not take into account the different ways in which the table in the outer query may be accessed and vice versa.

Global query optimization allows DB2 to optimize a query as a whole rather than as independent parts. This is accomplished by allowing DB2 to:

- Consider the effect of one query block on another
- Consider reordering query blocks

Subquery processing is changed due to the new consideration for cross-query block optimization. All subqueries are now processed by the DB2 optimizer differently than before, and the new processing is summarized as follows:

- The subquery itself is represented as a "virtual table" in the FROM clause that contains the predicate with the subquery.
- This "virtual table" may be moved around within the referencing query in order to obtain the most efficient sequence of operations.
- Predicates may be derived from the correlation references in the subquery and from the subquery SELECT list.
- These predicates can be applied to either the subquery tables or the tables that contain the correlated columns depending on the position of the "virtual table".
- When determining the access path for a subquery, the context in which the subquery occurs is taken into consideration.
- When determining the access path for a query that references a subquery, the effect that the access path has on the subquery is taken into consideration.

Note: The global query optimization functionality is available in compatibility mode and requires a REBIND.

Index on Expression

Prior to DB2 9, the create index statement only allowed you to use the column name from the table the index is being built on. The same value for each column stored on the table was copied into the index and used to quickly identify the row when searching by that column name. Now with DB2 9, the create index statement supports a feature known as key expressions. Here's why key expressions should interest you.
Many times when you're writing complex queries, you'd like to the optimizer to use an index, but instead it performs a table space scan. These situations usually occur when you're using scalar functions or arithmetic calculations. Let's consider a couple of examples:
Say you have a column on the table called lastname and the data in this column is stored in lowercase. But because end users can enter last names in upper and lower case, you must convert everything to upper case to perform the search. The following sample contains DDL to create an index on the table EMPL and then SQL to search for employees with a last name of 'SMITH'.

```
CREATE INDEX XEMP_LNNP  ON EMPL
  (lastname);

SELECT EMPNO, FIRSTNME, LASTNAME, SALARY, BONUS, COMM
FROM EMPL
WHERE UPPER(lastname) = 'SMITH'
```

No match: In this sample, DB2 cannot use index XEMPL_LNNP because of the scalar function UPPER.

To resolve this in DB2 9, you can use an expression to store the data in the index as upper case.

```
CREATE INDEX XEMPL_LNUP ON EMPL
         (UPPER(LASTNAME,'UNI')      );
```

Now DB2 can use index XEMPL_LNUP and match on the data.

Next is an example of the types of queries you'd see in a data decision support or data warehousing application. Say you need to present a list of employees that have a total compensation package greater than $12,000 a month. The compensation package is made up of salary, bonus and commission. In DB2 8 your index on the columns may be defined this way:

```
CREATE INDEX XEMPL_COMP ON EMPL
 (salary, bonus, comm);

SELECT EMPNO, FIRSTNME, LASTNAME, SALARY, BONUS, COMM
FROM EMPL
WHERE salary + bonus + comm. > 12000.00
```

However, due to the arithmetic calculations in the where clause, the optimizer cannot use the EMPL_X3 index to retrieve all employees making more than $12,000. With DB2 9 though you can create an index with the expression to calculate the total and store this in the index. The optimizer can now match on the column because the data was precalculated and stored in the index. Index EMPL_X4 shows how you'd use a key expression to calculate and store the results in an index.

```
CREATE INDEX XEMPL_COMSUM ON EMPL
 (salary + bonus + comm);
```

A common problem is the search for names (lastname, street, city) where many similar spelling exist. For example the german name 'MEYER' exists as 'MAIER', 'MAIR', 'MEYER', 'MEIER'.
To search all 'MEYER' you can use the following SQL

```
SELECT EMPNO, FIRSTNME, LASTNAME, SALARY, BONUS, COMM
FROM EMPL
WHERE lastname IN ('MEYER', 'MAYER', 'MAIER', 'MAYR', 'MEIER')
```

but the person you are searching for is spelled as 'MEIR' and therefor not found with this SQL.

Alternatively you can use the soundex function.

```
SELECT EMPNO, FIRSTNME, LASTNAME, SALARY, BONUS, COMM
FROM EMPL
WHERE soundex(lastname) = soundex('Meyer')
```

The Select with the soundex function is not indexable in DB2 version 8. Now in DB2 9 you can define an index like this:

```
CREATE INDEX XEMPL_SOU ON EMP
  (sundex(lastname));
```

The search with soundex is now indexable.

Notes:
If you're going to use expressions, consult the DB2 9 SQL Reference guide. A list of rules and restrictions can be found under the CREATE INDEX statement.
The basic rule is that you must reference at least one column from the table. The column named must not be of type LOB, XML or DECFLOAT. The referenced columns cannot include any FIELDPROCs or a SECURITY LABEL.
Also, key expressions must not include:

- A subquery -- an aggregate function
- A not deterministic function
- A function that has an external action
- A user-defined function
- A sequence reference
- A host variable
- A parameter marker
- A special register
- A CASE expression
- An OLAP specification

Conclusion
Index on expression shows significant improvements in query performance. Laboratory test results show dramatic improvement when index on expression is used. There is a significant reduction in overall CPU and DB2 getpages.
Although there is CPU cost on the evaluation of the expression, the CPU overhead is considered reasonable. These operations are LOAD, CREATE INDEX, REBUILD INDEX, REORG TABLESPACE, and CHECK INDEX.

MERGE and SELECT FROM MERGE

The MERGE statement combines the conditional UPDATE and INSERT operation in a single statement. This provides the programmer with ease of use in coding SQL. It also reduces the amount of communication between DB2 and the application as well as network traffic. The statements can be either dynamic or static.

Example:

```
MERGE INTO EMP
 USING (VALUES('123456','Hans','F','Beck','F01'))
         AS NEW (EMPNO, FIRSTNME, MIDINIT, LASTNAME, WORKDEPT)
ON EMP.EMPNO = NEW.EMPNO
WHEN MATCHED THEN UPDATE SET WORKDEPT = 'F01'
WHEN NOT MATCHED THEN
 INSERT (EMPNO, FIRSTNME, MIDINIT, LASTNAME, WORKDEPT)
  VALUES(NEW.EMPNO, NEW.FIRSTNME, NEW.MIDINIT, NEW.LASTNAME, WORKDEPT)
NOT ATOMIC CONTINUE ON SQLEXCEPTION;
```

Example:

```
SELECT * FROM FINAL TABLE (
       MERGE INTO EMP
        USING (VALUES('123456','Hans','F','Beck','F01'))
                AS NEW (EMPNO, FIRSTNME, MIDINIT, LASTNAME, WORKDEPT)
       ON EMP.EMPNO = NEW.EMPNO
       WHEN MATCHED THEN UPDATE SET WORKDEPT = 'F01'
       WHEN NOT MATCHED THEN
        INSERT (EMPNO, FIRSTNME, MIDINIT, LASTNAME, WORKDEPT)
         VALUES(NEW.EMPNO, NEW.FIRSTNME, NEW.MIDINIT, NEW.LASTNAME, WORKDEPT)
       NOT ATOMIC CONTINUE ON SQLEXCEPTION
)
```

Result:

```
EMPNO   FIRSTNME      MIDINIT  LASTNAME         WORKDEPT  PHONENO
---------+---------+---------+---------+---------+---------+------
123456  Hans          F        Beck             F01       -------
```

Refer to the DB2 Version 9.1 for z/OS SQL Reference, SC18-9854, for a detailed explanation about the MERGE SQL syntax.

Performance:
The overall results of using the new functionality compared to the base case indicate that:

- The static MERGE statement performs 13% better.
- The static SELECT FROM MERGE statement performs 19% better.
- The dynamic MERGE statement performs 20% better.
- The dynamic SELECT FROM MERGE statement performs 9% better.

Conclusion:
The usage of MERGE and SELECT FROM MERGE provides a powerful and more efficient way of coding SQL than the traditional way of using multiple SQL statements to provide the same functionality. The usage of this new SQL syntax provides better performance, less interaction across the network, less DB2 and application interaction, and less SQL to code.

Notes:

- The MERGE operation and the access of the intermediate result table from the MERGE operation does not use parallelism.

- The MERGE and SELECT FROM MERGE operations are available in new-function mode.

Refer to the Redbook DB2 9 for z/OS Performance Topics SG24-7473.

SELECT FROM UPDATE or DELETE

The SELECT FROM UPDATE or DELETE closes the gap for DB2 family compatibility:

- DB2 for Linux, UNIX®, and Microsoft® Windows®: SELECT FROM INSERT/UPDATE/DELETE
- DB2 for z/OS V8: SELECT FROM INSERT
- DB2 for z/OS V9: SELECT FROM UPDATE/DELETE
-

The new syntax for V9 allows SELECT from a searched UPDATE or searched DELETE. A single statement can now be used to determine the values **before or after** records are updated or deleted. Work files are used to materialize the intermediate result tables that contain the modified rows.

Example:

```
select firstnme, lastname, salary, empno from old table (
update emp set salary = 3500 where empno = '123456');
```

Result:

```
---------+---------+---------+---------+---------+
FIRSTNME      LASTNAME              SALARY  EMPNO
---------+---------+---------+---------+---------+
Hans          Beck             -----------  123456
```

Important:

- Use SELECT FROM DELETE with segmented table space with caution because mass delete is disabled in SELECT FROM DELETE.
- Additional CPU usage in some cases (Refer to the Redbook DB2 9 for z/OS Performance Topics SG24-7473)

FETCH FIRST and ORDER BY in subselect and fullselect

In DB2 V8, the ORDER BY and FETCH FIRST n ROWS ONLY clauses are only allowed at the statement level as part of a SELECT statement or a SELECT INTO statement. You can specify the clauses as part of a SELECT statement and write:

Example 1: Restriction in V8 – only in Select

```
select workdept from emp t2
   order by salary desc
   fetch first 1 rows only
```

Now DB2 V9 allows specification of these new clauses as part of subselect or fullselect.

Example 2.1: in subselect or fullselect

```
-- v8 solution
select * from dept t1 where t1.deptno in
  (select workdept from emp t2,
             table( select max(salary) as maxsalary  from emp) as t3
           where t2.salary = t3.maxsalary
  );

-- v9 compatibility mode
select * from dept t1 where t1.deptno in
  (select workdept from emp t2
   order by salary desc
   fetch first 1 rows only) ;
```

Example 2.2:

```
select * from (
select salary, workdept, 'Best ' from emp
order by salary desc fetch first 1 rows only) as best
union all
select * from (
select salary, workdept, 'Worst' from emp
order by salary asc  fetch first 1 rows only) as worst
```

TRUNCATE SQL statement

To empty a table, you have to either do a mass delete, using DELETE FROM table name without a WHERE clause, or use the LOAD utility, with REPLACE REUSE and LOG NO NOCOPYPEND.
If there is a delete trigger on the table, using the DELETE statement
requires you to drop and subsequently recreate the delete trigger to empty the table without firing the trigger. LOAD REPLACE works on a table-space level instead of on a table level. You cannot empty a specific table if the table space that belongs contains multiple tables.
The TRUNCATE statement addresses these problems. The TRUNCATE statement deletes all rows for either base tables or declared global temporary tables. The base table can be in a simple table space, a segmented table space, a partitioned table space, or a universal table space. If the table contains LOB or XML columns, the corresponding table spaces and indexes are also truncated.

EXAMPLE.

```
  truncate emp;
---------+---------+---------+---------+--
DSNE615I NUMBER OF ROWS AFFECTED IS 45147
```

Performance
The TRUNCATE operation can run in a normal or fast way. The way that is chosen depends on the table type and its attributes. Users cannot control which way the TRUNCATE statement is processed.
Eligible table types are simple, partitioned, or segmented. Table attributes that determine the way in which the table is truncated are change data capture (CDC), multiple-level security (MLS), or VALIDPROC.
The normal way of processing implies that the TRUNCATE operation must process each data page to physically delete the data records from that page. This is true in the case where the table is either simple or partitioned regardless of table attributes or a table has CDC, MLS, or a VALIDPROC.
The fast way of processing implies that the TRUNCATE operation deletes the data records without physically processing each data page. This is true in the case where a table is either **segmented or in a universal table space** with no table attributes.

Note:
The TRUNCATE SQL statement is available in new-function mode.

REOPT AUTO

Despite all the enhancements in DB2 optimization, the host variable impact on dynamic SQL optimization and execution is still visible and can result in less than efficient access paths. Customers require DB2 to come up with the optimal access path in the minimum number of prepares. For dynamic SQL, DB2 V8 has REOPT(NONE), REOPT(ONCE) and REOPT(ALWAYS). **Static only supports REOPT(NONE) and REOPT(ALWAYS)**. DB2 V9 introduces the new reoptimization construct REOPT(AUTO).
REOPT(AUTO) can specify whether dynamic SQL queries (only Dynamic SQLs using Dynamic Statements Cache) with predicates that contain
parameter markers are to be automatically reoptimized when DB2 detects that one or more changes in the parameter marker values renders dramatic selectivity changes. The newly generated access path replaces the current one and can be cached in the statement cache. Consider using this functionality especially when queries are such that a comparison can alternate between a wide range or a narrow range of unknown host variable values.
REOPT AUTO can reduce the number of prepares for dynamic SQL (both short and full) when used for queries that have host variable range changes. It may also improve execution costs in a query such as:

Example:

```
SELECT * FROM EMP WHERE SALARY > :hostvar
```

Tablespace Scan for hostvar = 0 and possibly index scan with hostvar = 100000.

Notes:
For statements for which REOPT(AUTO) may result in frequent re-optimization, note that the current implementation has a 20% limitation of re-optimization of the total executions on an individual statement. This means that, if every statement execution would result in re-optimization because of the different disparate host variable values, re-optimization will not necessarily occur due to this limit. For this kind of processing, use REOPT(ALWAYS) to avoid this limitation.

The reason for the new REOPT function is reported through a field in IFCID 0022 records. The number of the predicate that triggers REOPT is recorded. IFCID 0003 records the number of REOPT times due to the parameter marker value changes at the thread level. The number of REOPTs caused by REOPT(AUTO) is recorded in a new field of IFCID 0316 records.

Usage in individual Dynamic SQL Statements with java :

NULLIDR1: Packages are Bound in this collection with REOPT(ONCE)
NULLIDRA: Packages are Bound in this collection with REOPT(ALWAYS)

jdbcCollection=NULLIDR1 > You will be using REOPT(ONCE)
jdbcCollection=NULLIDRA > You will be using REOPT(ALWAYS)

BIGINT, VARBINARY, BINARY, and DECFLOAT

For optimal performance, we recommend that you map the Java data types that are used in applications to the DB2 column data types. The main reason is to provide for efficient predicate processing. It also minimizes data type conversion cost.
In DB2 V8, usage of the decimal data type was required for large integers. Some of the issues that need to be addressed are that, in Java, longs are primitive types, which are stack allocated. BigDecimals are heap allocated, which is more expensive; they take up a lot more storage and need to be garbage collected.
Native support of the decimal floating point (DECFLOAT) data type in DB2 V9 enables DECFLOAT data to be stored or loaded into DB2 tables; it also allows for manipulation of DECFLOAT data. These data types provide portability and compatibility to the DB2 family and platform.

Appendix A – Sample Table EMP

```
  CREATE TABLE EMP
     (EMPNO                CHAR(6) FOR SBCS DATA NOT NULL,
      FIRSTNME             VARCHAR(12) FOR SBCS DATA NOT NULL,
      MIDINIT              CHAR(1) FOR SBCS DATA NOT NULL,
      LASTNAME             VARCHAR(15) FOR SBCS DATA NOT NULL,
      WORKDEPT             CHAR(3) FOR SBCS DATA WITH DEFAULT NULL,
      PHONENO              CHAR(4) FOR SBCS DATA WITH DEFAULT NULL,
      HIREDATE             DATE WITH DEFAULT NULL,
      JOB                  CHAR(8) FOR SBCS DATA WITH DEFAULT NULL,
      EDLEVEL              SMALLINT WITH DEFAULT NULL,
      SEX                  CHAR(1) FOR SBCS DATA WITH DEFAULT NULL,
      BIRTHDATE            DATE WITH DEFAULT NULL,
      SALARY               DECIMAL(9, 2) WITH DEFAULT NULL,
      BONUS                DECIMAL(9, 2) WITH DEFAULT NULL,
      COMM                 DECIMAL(9, 2) WITH DEFAULT NULL,
      CONSTRAINT EMPNO
      PRIMARY KEY (EMPNO),
      CONSTRAINT NUMBER
        CHECK (PHONENO >= '0000' AND PHONENO <= '9999'))
    IN DBSAMPLE.DSN8SYYE
    EDITPROC  DSN8EAE1
    AUDIT NONE
    DATA CAPTURE NONE
    CCSID      UNICODE
    NOT VOLATILE;
--
  GRANT DELETE,INSERT,SELECT,UPDATE ON TABLE EMP
    TO PUBLIC AT ALL LOCATIONS;
--
  COMMIT;
--
------------------------------------------
-- Database=DBSAMPLE
--    Index=XEMP1 On EMP
------------------------------------------
--
  CREATE UNIQUE INDEX XEMP1
    ON EMP
     (EMPNO                 ASC)
    USING STOGROUP SYSDEFLT
    PRIQTY 12
    FREEPAGE 0 PCTFREE 10
    GBPCACHE CHANGED
    CLUSTER
     (PART 1 VALUES('099999'),
      PART 2 VALUES('199999'),
      PART 3 VALUES('299999'),
      PART 4 VALUES('999999'))
    BUFFERPOOL BP0
    CLOSE NO
    COPY NO;
```

```
--
  COMMIT;
--
-------------------------------------------
-- Database=DBSAMPLE
--    Index=XEMP2 On EMP
-------------------------------------------
--
  CREATE INDEX XEMP2
    ON EMP
     (WORKDEPT              ASC)
    USING STOGROUP SYSDEFLT
    PRIQTY 12
    ERASE  NO
    FREEPAGE 0 PCTFREE 10
    GBPCACHE CHANGED
    NOT CLUSTER
    BUFFERPOOL BP0
    CLOSE NO
    COPY NO
    PIECESIZE 2 G;
--
  COMMIT;
--
-----------------------------------------------
-- Database=DBSAMPLE
--    Index=XEMP_LNNP On EMP
-----------------------------------------------
--
  CREATE INDEX XEMP_LNNP
    ON EMP
     (LASTNAME              DESC)
    NOT PADDED
    USING STOGROUP SYSDEFLT
    PRIQTY 12
    ERASE  NO
    FREEPAGE 0 PCTFREE 10
    GBPCACHE CHANGED
    NOT CLUSTER
    BUFFERPOOL BP0
    CLOSE NO
    COPY NO
    PIECESIZE 2 G;
--
  COMMIT;
--
CREATE INDEX EX74519.XEMP_LNUP
  ON EX74519.EMP
   (UPPER(LASTNAME,'UNI')        ASC)
  USING STOGROUP SYSDEFLT
  PRIQTY 12
  ERASE  NO
  FREEPAGE 0 PCTFREE 10
  GBPCACHE CHANGED
```

```
  NOT CLUSTER
  BUFFERPOOL BP0
  CLOSE NO
  COPY NO
  PIECESIZE 2 G;

  CREATE INDEX EX74519.XEMP_SE
    ON EX74519.EMP
     (soundex(LASTNAME)              ASC)
    USING STOGROUP SYSDEFLT
    PRIQTY 12
    ERASE  NO
    FREEPAGE 0 PCTFREE 10
    GBPCACHE CHANGED
    NOT CLUSTER
    BUFFERPOOL BP0
    CLOSE NO
    COPY NO
    PIECESIZE 2 G;

commit;
```