Controlling Plans on DB2

DB2 offers relatively sparse vendor-specific tools to control execution plans, so the methods used to tune on DB2 are comparatively indirect. There are three main steps involved in tuning on DB2:

1. Provide the optimizer with good statistics about the tables and indexes, so it can calculate the costs of alternatives accurately.
2. Choose the optimization level that DB2 applies to your query.
3. Modify the query to prevent execution plans that you do not want, mainly using the methods described earlier in Section 4.1.

4.3.1 DB2 Optimization Prerequisites

Proving that a little knowledge is a dangerous thing, cost-based optimizers often do a terrible job if they do not have statistics on all the tables and indexes involved in a query. It is therefore imperative to maintain statistics on tables and indexes reliably; this includes regenerating statistics anytime table volumes change much or anytime tables or indexes are rebuilt. It is safest to regenerate statistics periodically, during times when load is relatively quiet, nightly or at least weekly. Edit a file *runstats_schema.sql* from the Unix prompt and type the following commands, replacing <*Schema_Name*> with the name of the schema that contains the objects you wish to run statistics on:

```
-- File called runstats_schema.sql
SELECT 'RUNSTATS ON TABLE <Schema_Name>.' || TABNAME || ' AND INDEXES ALL;'
FROM SYSCAT.TABLES
WHERE TABSCHEMA = '<Schema_Name>';
```

To use this script, log into *db2*, escape to the shell prompt with `quit;`, and run the following two commands from the Unix shell:

```
db2 +p -t < runstats_schema.sql > tmp_runstats.sql
grep RUNSTATS tmp_runstats.sql | db2 +p -t > tmp_anal.out
```

These commands can be scheduled to run automatically. Check *tmp_anal.out* in case any of the analyses fail.

Often, queries include conditions on highly skewed distributions, such as conditions on special types, codes, or flags, when these columns have only a few values. Normally, the CBO evaluates selectivity of a condition based on the assumption that all nonnull values of a column are equally selective. This assumption generally works well for foreign and primary keys that join business entities, but it breaks down when the columns have permanent special meanings and certain meanings apply much more rarely than others.

For example, in an `Orders` table, you might have a `Status_Code` column with three possible values: '`CL`' for closed (i.e., fulfilled) orders, '`CA`' for cancelled orders, and '`OP`' for

open orders. Most orders, by far, would be fulfilled; so, once the application has been running for a few months, you'd expect 'CL' to point to a large and steadily increasing number of orders. A steady, significant fraction of orders would end up cancelled, so 'CA' would also eventually point to a large list of orders. However, as long as the business keeps up with incoming orders, the number of open orders would remain moderate and steady, even as data accumulates for years. Quite early, a condition that specified `Status_Code='OP'` would be selective enough to justify indexed access, if you had an index with that leading column, and it is important to enable the optimizer to realize this fact, preferably without a lot of manual tuning. This requires two things:

- The SQL must mention the specific selective value, rather than use a bind variable. Use of bind variables is commonly attractive, since it makes SQL more general and easier to share between processes. However, this need to hardcode especially selective values is the exception to that rule. If you use `Status_Code= ?` instead of `Status_Code='OP'`, you will deny the CBO potential knowledge of the selectivity of the condition at parse time, when it does not yet know whether the bind variable `?` will be assigned to a common or a rare `Status_Code`. Fortunately, in these cases, the usual reason to prefer using bind variables does not generally apply; since these special codes have special business meanings, it is unlikely that the SQL will ever require substituting a different value than the single selective value.
- You need to provide the CBO with special statistics that quantify how rare the uncommon code, type, or status values are, so it can know which values are highly selective.

DB2 stores special statistics on distribution, when you request them. To create distribution statistics for the example case, given an index named `Order_Stts_Code` and the schema owned by `ApplProd`, use the following command:

```
RUNSTATS ON TABLE ApplProd.Orders
  WITH DISTRIBUTION FOR INDEX ApplProd.Order_Stts_Code;
```

Anytime you have a column with a skewed distribution and an index that you wish to use when your condition on the column has a high degree of selectivity, be sure to create distribution statistics in the manner shown here.

4.3.2 Choosing the Optimization Level

DB2 offers multiple optimization levels. An optimization level is basically a ceiling on how clever the optimizer attempts to be when it considers the range of possible execution plans. At optimization level `0`, DB2 chooses the lowest cost plan within a subset of the plans it considers at level `1`; at level `1`, it considers just a subset of the plans it considers at level `2`; and so on. Nominally, the highest optimization level should always yield the best plan, because it chooses the lowest cost plan from the widest possible range of alternatives.

However, the plans enabled by the higher optimization levels tend to be less robust and often prove disappointing. In spite of the optimizer's calculations to the contrary, these less robust plans often run longer than the best robust plan that the lower-level optimization sees. Higher levels of optimization can also take longer to parse, since the optimizer has additional degrees of freedom to explore. Ideally, you parse every statement at the lowest level that is capable of finding the best execution plan for a given query.

DB2 offers seven levels of optimization: `0`, `1`, `2`, `3`, `5`, `7`, and `9`.[1] Level `5` is normally the default, although database administration can override this default choice. I have never needed levels of optimization higher than `5`; levels `7` and `9` appear mainly to enable relatively exotic query transformations that are rarely useful. However, I have frequently found excellent results with the lowest level of optimization, level `0`, when level `5` produced a poor plan. Before executing a query (or checking an execution plan), set level `0` with the following SQL statement:

[1] Levels `4`, `6`, and `8` are not available, presumably for historical reasons, although I have never found these reasons documented.

```
SET CURRENT QUERY OPTIMIZATION 0;
```

When you wish to return to level `5` for other queries that require it, use the same syntax, replacing `0` with `5`. If you find a poor plan at level `5`, I recommend trying level `0` after first verifying correct statistics on the tables and indexes involved. Level `0` frequently yields just the sort of robust plans that usually work best for real-world applications.

4.3.3 Modifying the Query

Most manual tuning on DB2 uses the SQL changes described earlier in Section 4.1. However, one particular manual technique deserves special mention, because it proves useful more often on DB2 than on Oracle and SQL Server. DB2 stores index records even for null values of indexed columns, and it appears to treat null like just another indexed value.

When DB2 lacks special statistics on distribution (see Section 4.3.1), DB2 estimates the selectivity of `Indexed_Column IS NULL` to be just as high as `Indexed_Column = 198487573` or any other nonnull value. Therefore, older DB2 versions often choose to drive to selective-looking `IS NULL` conditions on indexed columns. Occasionally, this works out fine. However, in my experience, `IS NULL` conditions are rarely anywhere near as selective as the average individual nonnull value, and indexed access driven by `IS NULL` conditions is almost always a mistake.

Therefore, when you find an `IS NULL` condition on an indexed column in a DB2 query, you often should prevent use of the index. The simplest equivalent condition that prevents index use is `COALESCE(Indexed_Column, Indexed_Column) IS NULL`. This version is perfectly equivalent to the original condition `Indexed_Column IS NULL`, but the

COALESCE( ) function prevents index use.

In addition to tuning techniques that can apply to any database, there are three useful techniques specific to DB2 that I describe in the following sections.

4.3.3.1 Place inner joins first in your FROM clause

One sometimes useful technique is simply to list inner joins first in your FROM clause. This appears never to hurt, and on older versions of DB2 I have seen this simple technique produce greatly improved execution plans.

4.3.3.2 Prevent too many outer joins from parsing at once

Older versions of DB2 can take minutes to parse queries with more than about 12 outer joins, and even then they might fail with errors. Fortunately, there is a workaround for this problem, using the following template for the SQL. The workaround uses DB2's nested-tables syntax, in which an outer query contains another query inside a FROM clause that is treated like a single table for purposes of the outer query:

```
SELECT ...
FROM (SELECT ...
      FROM (SELECT ... FROM <all inner joins and
                             ten outer joins>
                       WHERE <Conditions pertinent
                              to this innermost nested table>) T1
            LEFT OUTER JOIN <Joins for the 11th
                             through 20th outer join>
      WHERE <Conditions, if any, pertinent
             to this outermost nested table>) T2
     LEFT OUTER JOIN <The rest of the outer joins (at most 10)>
WHERE <Conditions, if any, pertinent to the outer query>
```

This template applies to a query with 21-30 outer-joined tables. With 11-20 outer-joined tables, you need only a single nested table. With more than 30 outer-joined tables, you need even deeper levels of nesting. In this syntax, DB2 effectively creates nested views on the fly, as defined by the queries inside parentheses in the FROM clauses. For purposes of handling outer joins, DB2 handles each of these smaller queries independently, sidestepping the problem of too many outer joins in a single query.

At my former employer, TenFold, we found this technique so useful that we enhanced the EnterpriseTenFold product to automatically generate this extraordinarily complex SQL when required. Admittedly, it is not an easy solution for manually written SQL, but it still might be the only technique that works if you run into slow or failed parses of many-way outer joins on DB2.

4.3.3.3 Let DB2 know when to optimize the cost of reading just the first few rows

Normally, DB2 calculates the cost of executing the entire query and chooses the plan it

expects will run the fastest end to end. However, especially for online queries, you often care only about the first few rows and prefer to optimize to get the first rows soonest.

The technique to read the first rows fast, usually following nested loops, is to add the clause `OPTIMIZE FOR` <*n*> `ROWS` (or `OPTIMIZE FOR 1 ROW`), where <*n*> is the number of rows you actually need to see fast out of the larger rowset that the query might theoretically return. This clause goes at the very end of the query and instructs DB2 to optimize the cost of returning just those first <*n*> rows, without regard to the cost of the rest of the query execution. If you actually know how many rows you want and trust the optimizer to calculate the best plan, you can choose <*n*> on that basis. If you want to force a robust, nested-loops plan as strongly as possible, just use `OPTIMIZE FOR 1 ROW`.

In practice, this technique tends to dictate nested-loops joins, because they avoid reading whole rowsets before even beginning a join. However, it is possible for an explicit `ORDER BY` clause to defeat any attempt to reach the first rows fast. The `ORDER BY` clause usually requires a sort following the complete query, usually postponing return of the first row regardless of the execution plan. You can leave out a sort condition if you want to force nested-loops joins by this technique, performing the sort in your application if necessary. The `OPTIMIZE FOR 1 ROW` hint is the equivalent of the `FIRST_ROWS` hint on Oracle and the `OPTION(FAST 1)` hint on SQL Server.

Techniques to force precisely chosen execution plans on DB2 are sparse, in contrast to the extraordinary detail that DB2 reveals about the execution plan you already have and why DB2 chose it. However, in fairness, I should mention that the available techniques, in combination with DB2's fairly good optimizer, have proven sufficient in my own experience.