Explaining DB2/2 and DB2/6000 Explain

August 1993

Berni Schiefer
schiefer@vnet.ibm.com

Share Session A465

IBM Canada Ltd. Laboratory
Database Technology
895 Don Mills Rd.
North York, Ontario

Contents

What is EXPLAIN?

EXPLAIN interprets access plans for static DML statements

- SELECT
  - prime candidate
- UPDATE
- INSERT
- DELETE

EXPLAIN is a DB2/2 and DB2/6000 tool used:

- by DBAs
- application programmers

Notes to the foil:

EXPLAIN is a tool shipped as a productivity aid with DB2/2 and DB2/6000. It is used primarily to analyze static Data Manipulation Language (DML) statements. Since all the non-select DML statements can essentially be viewed as first selecting a set of tuples (rows) and then changing them in some way they are in some sense variations on a SELECT. Hence, this presentation will focus on explaining SELECT statements.

It should be noted that the EXPLAIN tool will explain almost any SQL statement, including DDL, that may be included in your application.

EXPLAIN is a tool which is invaluable both to DBAs and application programmers. DBAs will use EXPLAIN both for assisting in the design and development of applications as well as trouble-shooting queries which have gone awry. Application programmers will use EXPLAIN to help them develop efficient applications. These ideas are explored in greater detail on the next page.

Why use EXPLAIN?

- To look for performance tuning opportunities
  - How useful are additional indexes?
  - Does rewriting the query help?
- To understand changes in query performance
  - Due to changes in the data model
  - Due to changes in the data
  - Due to migration to a new release
  - Due to application of service

Notes to the foil:

There are two basic reasons for using the EXPLAIN statement.

1. Although DB2/2 and DB2/6000 return the correct answer regardless of how the database or the query are structured, there are tremendous opportunities for performance tuning. This tuning might be a necessity or simply part of an ongoing effort to minimize the consumption of resources. In either case, the EXPLAIN tool can be a powerful aid in predicting how various tuning parameters affect the cost and choice of access plan for any given query. Using the EXPLAIN tool, we can determine whether additional access paths (e.g. additional indexes) might be helpful to a particular query and what benefits, if any, could be obtained by rewriting the query.

2. The EXPLAIN statement can also be used to monitor queries. Once a set of queries have been tuned and their explain output preserved, any changes to the database can be monitored for impact on the queries. Any change should be investigated to see if it results in improved or decreased performance. A different explain result can be caused by a wide variety of database changes. However, it is important to realize that only a very small fraction of the queries will ever use a different access path as a result of a single change.

Thus, EXPLAIN can be used to help tune a query and to ensure that other database changes do not adversely affect it.

How to invoke EXPLAIN?

- To invoke EXPLAIN use the following syntax:
  1. DB2/2

```
EXPLAIN <database> <package> <creator> <section>  <outfile>
e.g. EXPLAIN SAMPLE EXPDEMO SCHIEFER 0 RESULTS.EXP
```

  2. DB2/6000

```
db2expln -d <database> -p <package> -c <creator> -s <section>  -o <outfile>
e.g. db2expln -d sample -p expdemo -c schiefer -s 0 -o results.exp
```

- EXPLAIN will prompt for missing parameters
- Help for EXPLAIN command options is available

Notes to the foil: The syntax of DB2/2 EXPLAIN is described below:

EXPLAIN <database> <package> <creator> <section> <target>

where,

<database> is the name of the database in which packages are to be explained.
<package> is the name of the package to be explained
<creator> is the name of the creator of the package
<section> is the section number of the package to be explained. Optionally, the number 0 may be entered to explain all sections belonging to the package chosen.

Section numbers can be found by querying the system catalog SYSIBM.SYSSTMT.
<outfile> is the name of the file to which EXPLAIN will write the results.

A default output file of <package>.EXP will be used if no output file name is supplied. Note that EXPLAIN will write the output to the current directory.

If all of the parameters are not supplied, EXPLAIN will prompt the user for missing parameters.

Multiple packages can be explained with one invocation of explain. This is accomplished by accepting string constants for packages and creators with LIKE patterns. That is, the underscore (_) may be used to represent any given character, and the percent sign (%) may be used to represent the occurrence of zero or more characters.

For example, to explain all sections for all plans in a database named SAMPLE, with the results being written to the file MY.EXP, use the following command:

```
EXPLAIN SAMPLE % % 0 MY.EXP
```

Type **EXPLAIN ?** to obtain help.

The syntax of DB2/6000 EXPLAIN is described below in an extract of the help text provided.

```
db2expln [[-c [<creator>]]
          [-d [<dbname>]]
          [-o [<outfile>]]
          [-p [<pname>]]
          [-s [<sectnbr>]]
          [-h]]

Required Fields:
        -c <creator>     = package qualifier
        -d <dbname>      = database name containing packages
        -p <pname>       = package name
        -s <sectnbr>     = section number of package (for all sections, use zero)
        All options may be specified in any order.

        Package name and creator may be supplied in LIKE predicate form,
        which allows percent sign (%) and underscore (_) to be used as
        pattern matching characters to select multiple packages with one
        invocation.

        Prompting will occur for all required fields that are not supplied
        or are incompletely specified.

Optional Fields:
        -o <outfile>     = name of output file
        -h               = help

        If -o is specified without a file name, the user will be prompted for
        a file name (the default name is db2expln.out). If -o is not specified,

        then the output will be directed to stdout.
```

Type **db2expln -h** to obtain help. To invoke EXPLAIN on DB2/6000 to obtain the same result as for DB2/2 use the following invocation:

```
db2expln -d sample -p % -c % -s 0 -o my.exp
```

What does EXPLAIN do?

- EXPLAIN
  - Reads a stored package from the catalog
  - Interprets the contents of one or more sections
  - Writes ″text″ to an output file
- EXPLAIN answers the question:

  How will a DML statement be executed?

Notes to the foil: What actually happens when the EXPLAIN tool is invoked, is that a package containing the SQL associated with an application is loaded from the SYSIBM.SYSPLAN catalog.

Each package consists of one or more sections, each section corresponds to an SQL statement in the application. The embedded SQL statements will have been ″compiled″ from the original ASCII text into an internal form suitable for efficient execution. This compiled form of the SQL is called ″threaded code″.

The EXPLAIN tool then examines the threaded-code and interprets the encoded operations so it can produce ″English-like″ reports which describe how the SQL statement will be executed.

What does EXPLAIN output tell you?

- For each DML statement (section):
  - Whether an index or a table scan is used
    - which index
    - index-only access (DB2/6000)
    - selective or non-selective index scan
  - What kind of predicates were applied
    - selective index scans (#Key Columns > 0)
    - sargable index predicates
    - sargable predicates
    - residual predicates
    - # predicates (DB2/6000)
  - Order in which the tables are accessed
  - Join method used
  - Any sorts required
    - for a join (logical composite / new table)
    - for a distinct/group by/order by
- For delete/insert/update statements
  - additional work for referential integrity
- Lock intent information

Notes to the foil:

The output generated by the EXPLAIN tool consists of 3 parts.

1. The name of the package generated and the section number currently being explained. DB2/6000 will also provide the time that the package was bound to the database.

2. The text of the SQL statement being explained.

3. A detailed description of the access plan.

   The exact contents of the access plan are not described here. However, all the details regarding the possible text which may be provided by the EXPLAIN tool are documented in Appendix J of the DB2/2 Guide and Appendix G of the DB2/6000 Administration Guide.

   In general terms the detailed description gives the access path to each table (e.g. table scan or index scan etc.) as well as other operations (such as join and sort) that specify the entire access plan.

   Simple queries are easy to follow. However, more complex queries may take extra effort.

What′s new with EXPLAIN in DB2/2?

- Explain documentation incorporated into manuals
- Explain formats the SQL statement
  - We ″pretty print″ the SQL statement in a readable fashion
- Explain allows wildcards for package creator and package names
  - Can use % and _ to pattern match creators
  - similar to the LIKE predicate in SQL

Notes to the foil: Prior to DB2/2, documentation for EXPLAIN was not part of the documentation library. The text which accompanied the ″applet″ in ES 1.0 has been reworked and incorporated into the DB2/2 Guide.

Another readabilty enhancement is the formatting of the SQL statement. The previous version of the EXPLAIN tool extracted the text of the SQL statement from the SYSSTMT catalog and printed it unmodified. Now we break the query into logical sections using the major SQL control structures as reference points (e.g. SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY etc.)

Also, since you may want to explain more than a single package created by one or more individuals we now allow you provide wildcard characters that will be used to pattern match for both the package name and package creator. This will be valuable to users who wish to analyze all the applications acting on a given database.

What′s new with EXPLAIN in DB2/6000

- Explain tool renamed
  - DB2/2 name conflicts with UNIX command
  - "db2expln"
- Dynamic SQL explain
  - Sample Bourne shell script
  - "dynexpln"
- Explain ″UNIXized″.
  - Output to stdout (default) or a file
  - Uses UNIX argument style
  - Other usability enhancements
- Explain output enhanced
  - Predicate count given
  - Index ORing access method clarified
  - Index-only access indicated
  - {Aggregation,Sort,Distinct} pushdown indicated
    — See **Query Optimization for DB2/2 and DB2/6000**

Notes to the foil: With DB2/6000, the EXPLAIN tool has been further enhanced. Some of the changes relate to the UNIX environment in which DB2/6000 runs, while others were added to enhance the information presented by EXPLAIN.

Standard UNIX comes with a tool called "explain." Since it is bad practice to overwrite system commands, the EXPLAIN tool was renamed to "db2expln." Furthermore, standard UNIX commands pipe their output to stdout by default. They also provide command options in any order using a combination of a minus (i.e −) sign and a single character. DB2/6000 conforms to these standards.

A sample shell script named "dynexpln" is also provided. This simple Bourne Shell script takes an SQL statement as an argument, generates a trivial embedded SQL application program, binds it to the database, and then uses the regular EXPLAIN tool.

The EXPLAIN tool has also been enhanced to provide additional information about the access plan. Detailed information on what new information is available can be found in Appendix G of the DB2/6000 Administration Guide.

What influences the Optimizer?

```
                 Catalog
                    +              Database
  Query         Statistics       Configuration
    |               |             Parameters
    |               |                 |
    └───────────┐   |   ┌─────────────┘

            ┌────────────┐
            | Optimizer  |   <= assumptions
            └────────────┘
                  |

            "best plan"
```

- Catalog Statistics
  - SYSTABLES
  - SYSCOLUMNS
  - SYSINDEXES
- Database Configuration Parameters
  - buffpage

Notes to the foil: This slight digression from the main topic illustrates one of the most crucial aspects of understanding and using the EXPLAIN output. There are three major areas over which the user has control, even after the detailed database design (table and index definitions) has been completed. The user can:

1. Change the query

   The SQL language permits many equivalent representations of the same logical query. Also, in some cases, predicates can be added or removed without affecting the query result. The Optimizer optimizes each query somewhat differently depending on the query's structure. Thus, by changing the query it is possible to obtain different results in the EXPLAIN output.
2. Change the Statistics

   The Optimizer uses the catalog tables to obtain information about the underlying structure of the database. By making the statistics available and accurate the Optimizer can make a more informed decision. This is accomplished by invoking the RUNSTATS utility.
3. Change the database configuration parameters

   The Optimizer uses some of the database configuration parameters in making its decisions about how likely a page is to remain buffered and in determining the cost of sorting. By increasing the number of buffer pages available the estimated costs will decrease. This also implies that applications using static SQL should be bound with the same set of configuration parameters as those which will be in effect when the application is running.

The Optimizer also uses a set of constants for default filter factors, various CPU and I/O costs, the relative cost of I/O and CPU, the fraction of the buffer pages available, etc. The user has no way of changing these values. Upon receipt of a query, the Optimizer chooses an access plan that minimizes the total cost, using the information obtained from the factors described above.

Now that we understand what EXPLAIN is, how it is invoked, what kind of information it produces, and know in what ways we can influence the Optimizer's decision, we can start to look at how one might want to use EXPLAIN to answer questions that arise while tuning a query.

Questions for queries:

- Are we using
  - right indexes?
    - build new indexes after query analysis
  - a table scan when an index scan is better?
    - add an index
    - update statistics
  - an unclustered index?
    - reorganize data
    - update statistics
- Is our query
  - efficiently coded?
    - rewrite query
  - using realistic filter factors?
    - update statistics
    - use a dynamic query

Notes to the foil: Recall the discussion of the various factors that influence the Optimizer. Of the three main factors assume that the number of page buffers is set independently of any desire to tune a particular query. That leaves the statistics, the index definitions and the query itself as variables which one can modify to tune a query.

One way to determine whether performance tuning opportunities exist is to pose a question and then use EXPLAIN to answer the question. After making appropriate changes to the statistics or the structure of the query, the query should be executed to determine the actual change in elapsed time and resource consumption.

- It is often the case that an initial set of index definitions turns out to be insufficient. By careful examination of the query it is usually possible to discern additional helpful indexes. The select list, the join predicates and the local predicates are the the key parts of the query to examine.
- The best way to determine whether an index scan might be less expensive than a table scan is to look at the query, create one or more suitable indexes and then re-invoke the EXPLAIN tool. (Look for a change from ″Relation Scan″ to ″Index Scan″.)
- To discover whether an index scan is being used when a table scan might be better, update all the statistics so that the Optimizer has accurate statistics to determine the cost of using an index access path. If the data has been updated significantly, then an index that was previously clustered may no longer be clustered. After updating the statistics, the Optimizer may switch to using a table scan. (Look for a change from ″Index Scan″ to ″Relation Scan″.)
- If the statistics are accurate and the CLUSTERRATIO statistic in SYSINDEXES remains low, it may be useful to sort the data into key order so that a clustered index can be produced. (see the CLUSTERRATIO value in SYSINDEXES.) The benefit of a clustered index is that it reduces the number of I/O′s that have to be performed.
- Using our knowledge about the characteristics of DB2/2 and DB2/6000, as well as the data, we may be able to rewrite the query in an equivalent, but more efficient, form. (Invoke EXPLAIN for different query formulations and if the plan changes use elapsed time comparisons to determine which query is better.)
- Another area that can often be improved is the accuracy of filter factors. These can often be made more accurate by ensuring the availability of statistics on all tables. Since default filter factors must be used whenever host variables or parameter markers are used, a dynamic version

of the query allows the Optimizer to make better use of its statistics to compute accurate filter factors.

The selection of predicates in a query is important. Too few and the Optimizer doesn't have enough flexibility to filter out rows as early as possible and consider all the join combinations. Too many and the Optimizer may underestimate the join result size.

More questions for queries?

Have we

- made a table scan as inexpensive as possible?
  - reorganize the table to remove overflow rows
- made index scans as inexpensive as possible?
  - reorganize the table
- exploited the potential of index-only access?
  - add/modify an index
- provided join column indexes?
  - add indexes
- used equi-join predicates?
  - rewrite query
- provided too many / too few local predicates?
  - rewrite query

Notes to the foil: Here, we consider additional questions which can be answered using the catalog and EXPLAIN.

- To minimize the cost of a table scan:
  1. Reorganize your data periodically so that the number of rows that have overflowed onto new pages is small. (Look in the SYSTABLES catalog in column OVERFLOW.)

     In addition to the statistics, there are also other ways of reducing the cost of a table scan:
     a. Use suitable datatypes to make each row as short as possible. This will ensure that the table occupies as few pages as possible. For example, use SMALLINT rather than INTEGER where appropriate. Similarly, use the DATE and TIME datatypes rather than CHAR.
     b. Ensure that your database design reduces data redundancy. This will ensure that the table occupies as few pages as possible. Reducing data redundancy through normalization is a well-documented topic.
- To reduce the cost of an index scan:
  1. Reorganize the index. (Ensure that NLEVELS is as low as it can be.)
  2. Recluster the data (Ensure that CLUSTERRATIO is very high for at least one index.)
- Sometimes a query can obtain all the columns required from an index rather than the data page itself. This is called index-only access. Index-only access to the data can dramatically improve performance.
- Indexes on the join columns are recommended, since they reduce the costs of joins. (Match the join columns with those that are the initial columns of indexes.)
- Use equi-join predicates, if possible, since they allow the Optimizer to choose from additional join orders and join methods. (Examine your query for predicates that reference more than one table and ensure they are of the following form: T1.C1 = T2.C2)
- The Optimizer assumes that all predicates are independent. Thus, the filter factors of all predicates connected by AND predicates can be multiplied together to obtain a composite filter factor. If the independence assumption is wrong, then selectivity can be overestimated. If a column is functionally dependent on another, its predicate can be discarded. If it is not, and the estimated number of qualifying rows is greatly underestimated, an inappropriate join order may be selected.

Hints + Tips

1. Save EXPLAIN output for production applications
2. Save and print current catalog statistics.
   - Keep the DDL statements handy
   - CREATE TABLE
   - CREATE VIEW
   - CREATE INDEX
3. Use ″RUNSTATS ... AND INDEXES ALL″ if queries involve:
   a. unindexed columns with local predicates
   b. multiple join predicates between a pair of tables.
4. Rules of Thumb: Investigate if you have ...
   a. an OLTP application and no selective index scan
   b. an OLTP application and a merge scan join
   c. a browse application and a merge scan join
5. Merge scan join is an excellent choice
   - when no suitable indexes exist
   - for batch queries

Notes to the foil: In order to understand decisions the Optimizer has made it is essential to know what statistics are available to the Optimizer. It is therefore recommended that the relevant statistics are obtained and deciphered. Appendix A provides some example SQL statements that extract the important statistics.

When you are reading through the EXPLAIN output and want to understand why a particular access plan was chosen (or not chosen), it is important to have the details of the table and its associated views and indexes nearby. Remember that the SQL statement given in the EXPLAIN output may reference views (which may in turn reference other views). Hence, the final plan printed by EXPLAIN may bear little resemblance to the original query.

We have already seen several examples of the need for accurate statistics. Since the default method of collecting statistics is to only obtain table statistics, it is advisable to use “RUNSTATS ... AND INDEXES ALL” periodically to obtain all the statistics possible.

The rules of thumb listed are not complete but they provide some initial conditions that should always be investigated if they occur. Let′s examine then:

1. OLTP Applications

   OLTP transactions have the characteristic that they touch and return only a few rows. They typically have an equality predicate on some ″key″ column. Since this is the ideal environment for an index scan using range delimiting predicates, one ought to investigate if something else occurs.

   The same is true for joins. Nested loop joins are typically the most efficient join method when few rows need to be processed. Further investigation is warranted when a merge scan join is discovered in the EXPLAIN output.

2. Browse Applications

   Browse applications have the property that, although the search criteria may be quite vague which may cause a large number of rows to qualify, the user typically only scrolls through at most a few screens of information before deciding on more specific criteria and re-issuing the query. Since merge scan join with sorts will materialize the entire answer set before returning any rows to the user, this is not a suitable join method for browse applications. The fundamental problem here is that the user wants us to optimize for getting the first screenful of data while the Optimizer is attempting to minimize the consumption of resources for evaluating the entire query.

The solution is to provide suitable indexes and predicates so that there is no need for the Optimizer to choose a batch-oriented access path.

Merge scan join is, however, often the most efficient way of processing batch queries which typically process many rows. Also, if you suddenly want to join along attributes not previously foreseen, the merge scan join will greatly outperform a nested loop join.

Query

″Which non-manager employees earn more than 90% of the highest paid manager′s salary? Tell me their name, department name and earnings.″

```
SELECT S.ID,S.NAME,O.DEPTNAME,SALARY+COMM
FROM ORG O, STAFF S
WHERE
   O.DEPTNUMB = S.DEPT AND
   S.JOB < > ′Mgr′ AND
   S.SALARY+S.COMM > ALL( SELECT ST.SALARY*.9
                          FROM STAFF
                          WHERE ST.JOB=′Mgr′ )
ORDER BY S.NAME;
```

The examples which follow are extracts of DB2/6000 EXPLAIN output

A complete EXPLAIN output listing can be found in Appendix A, "DB2/6000 Explain Output" on page 23.

Notes to the foil: This is the sample query that will be used for the following examples. We note the following characteristics of the query which has two query blocks:

1. Query Block 1 consists of a 2-way join
   - The STAFF table has 2 local predicates (one of which involves a complex predicate involving an unquantified scalar subquery) and a join predicate.
   - The ORG table has no local predicates and 1 join predicate
2. Query Block 2 consists of an access to a single table
   - The STAFF table has 1 local predicate and produces at most 1 row due to the quantified predicate "ALL"

Our database uses the default number of buffers and sort heap size

1. DB2/2:

   buffpage = 25 4K pages
   sortheap = 2 64K segments

2. DB2/6000:

   buffpage = 1000 4K pages
   sortheap = 256 4K pages

Using the sample tables, we also know that this query returns exactly one row.

This example has been created with the sample tables in order to provide a familiar setting. Since the sample tables all contain very few rows, many of the decisions made by the Optimizer when choosing a plan become irrelevant since *any* access plan is inexpensive. If larger tables were to be chosen, then the differences would become much more pronounced.

Version 1

- No Indexes and No Statistics

```
Access Table Name = SCHIEFER.STAFF  ID = 17  #Columns = 2
  Scan Direction  = Forward
  Relation Scan
  Lock Intent Share
  Sargable Predicate(s)
    #Predicates = 1
Access Table Name = SCHIEFER.ORG  ID = 16  #Columns = 2
  Scan Direction  = Forward
  Relation Scan
  Lock Intent Share
  Sargable Predicate(s)
    #Predicates = 1
    Create/Insert Into Sorted Temp Table  ID = t1
      Sort  #Columns = 1
      Not Piped
Sorted Temp Table Completion  ID = t1
Access Table Name = SCHIEFER.STAFF  ID = 17  #Columns = 6
  Scan Direction  = Forward
  Relation Scan
  Lock Intent Share
  Sargable Predicate(s)
    #Predicates = 3
    ALL
    Create/Insert Into Sorted Temp Table  ID = t2
      Sort  #Columns = 1
      Piped
Sorted Temp Table Completion  ID = t2
Access Temp Table  ID = t2  #Columns = 5
  Scan Direction  = Forward
  Relation Scan
Merge Join
  Access Temp Table  ID = t1  #Columns = 2
    Scan Direction  = Forward
    Relation Scan
    Residual Predicate(s)
      #Predicates = 1
      Create/Insert Into Sorted Temp Table  ID = t3
        Sort  #Columns = 1
        Piped
Sorted Temp Table Completion  ID = t3
Access Temp Table  ID = t3  #Columns = 5
  Scan Direction  = Forward
  Relation Scan
```

Notes to the foil: Let us examine what the EXPLAIN results tell us about the Optimizer′s choice of access path and the assumptions made as a result of the unavailable statistics.

The first access plan fragment we see is an access of the STAFF table via a table scan. Furthermore, we can see that a sargable predicate is being applied which allows us to deduce that this is the subquery being evaluated. Un-correlated subqueries are evaluated just once and since this subquery returns just one row no temporary table is required.

Next, we see that the ORG table is also being accessed via a table scan. Since there are no single table predicates which reference the ORG table, none are indicated as being applied while scanning the table. The result of scanning the table is placed into a sorted temporary table for future use.

We now access the STAFF table again, this time for the outer query block. We can see this from the indication of an ALL sargable predicate being applied. This means that the results of evaluating the subquery are being used to reduce the number of qualifying rows from the STAFF table in the outer query block. Again, we sort the qualifying rows by placing them into a sorted temporary table.

Having prepared all the tables for later use, we now proceed to the main processing of the join. We access the temporary table that we created by sorting the STAFF table (ID = t2) and perform a merge scan join with the temporary table created by sorting the ORG table (ID = t1). Thus, the join order is STAFF joined with ORG. Now the reason for all the sorting is clear; since we have no indexes and a merge scan join requires sorted input, we had to sort the data. After the join, we again create a sorted temporary table to satisfy the ORDER BY requirements and finally read this temporary table to return the results to the user.

In summary, in the absence of any statistics, the Optimizer becomes assumption-based and has to choose both the estimated size of the tables, their column cardinalities, and the selectivity of any predicates. Not surprisingly, it thinks that a merge scan join involving sorts of the tables being joined is the best way to proceed. How do we improve the Optimizer′s ability to estimate? We issue a ″RUNSTATS″ on each of the tables.

Version 2

- No Indexes
- Full Statistics

```
Access Table Name = SCHIEFER.STAFF  ID = 17  #Columns = 2
  Scan Direction  = Forward
  Relation Scan
  Lock Intent Share
  Sargable Predicate(s)
    #Predicates = 1
Access Table Name = SCHIEFER.STAFF  ID = 17  #Columns = 6
  Scan Direction  = Forward
  Relation Scan
  Lock Intent Share
  Sargable Predicate(s)
    #Predicates = 3
    ALL
    Create/Insert Into Sorted Temp Table  ID = t1
      Sort  #Columns = 1
      Piped
Sorted Temp Table Completion  ID = t1
Access Temp Table  ID = t1  #Columns = 5
  Scan Direction  = Forward
  Relation Scan
Nested Loop Join
  Access Table Name = SCHIEFER.ORG  ID = 16  #Columns = 2
    Scan Direction  = Forward
    Relation Scan
    Lock Intent Share
    Sargable Predicate(s)
      #Predicates = 1
```

Notes to the foil: In this second version, we still don′t provide any indexes but we now have accurate statistics. Let′s look at how these statistics have affected the Optimizer′s choice of access path.

The processing of the subquery has not changed. We continue to scan the table looking for the largest salary of any manager.

Next, we can see that the access plan has changed dramatically. The join order, however, remains the same. Rather than sorting the two tables and then joining them, we now sort only the STAFF table and then use the nested loop join method to join the tables together. Now that we know there are only 8 rows in ORG and 35 in STAFF, we don′t have to worry about being unable to cache them in memory.

Since a nested loop join doesn′t require the input to be sorted, we might ask why we are sorting at all. The answer, of course, is the ORDER BY clause. Now the reason we don′t sort after the join (which is what many other products would do) is that the join does not reduce the number of rows but does add additional columns that would have to be carried during a sort. Hence, it makes sense to sort the STAFF rows before joining them to the ORG table.

Thus, just using statistics has allowed the Optimizer to obtain a much better estimate of the size and cost of the answer. Rather than being assumption-based, the Optimizer is now statistic-based. However, the Optimizer has only one choice of access path. How we do we create additional access paths for the Optimizer to consider? We have to add indexes. Where? On the primary key columns of each table which are used as the join columns (i.e. DEPTNUMB in ORG, and ID in STAFF).

Version 3

- Primary Key Indexes
- No additional Statistics

```
Access Table Name = SCHIEFER.STAFF  ID = 17  #Columns = 2
  Scan Direction  = Forward
  Relation Scan
  Lock Intent Share
  Sargable Predicate(s)
    #Predicates = 1
Access Table Name = SCHIEFER.STAFF  ID = 17  #Columns = 6
  Scan Direction  = Forward
  Relation Scan
  Lock Intent Share
  Sargable Predicate(s)
    #Predicates = 3
    ALL
    Create/Insert Into Sorted Temp Table  ID = t1
      Sort  #Columns = 1
      Piped
Sorted Temp Table Completion  ID = t1
Access Temp Table  ID = t1  #Columns = 5
  Scan Direction  = Forward
  Relation Scan
Nested Loop Join
  Access Table Name = SCHIEFER.ORG  ID = 16  #Columns = 2
    Scan Direction  = Forward
    Relation Scan
    Lock Intent Share
    Sargable Predicate(s)
      #Predicates = 1
```

Notes to the foil: In this version of our query we have added indexes but not performed a ″RUNSTATS″ after creating the indexes. We only have statistics for the tables but none for the indexes.

If we compare this plan to the one shown in the previous version, we can observe that there are no differences. Just adding indexes without the corresponding statistics forced the Optimizer to make a conservative estimate of the cost of using the index and, hence, continue using the previous plan. So, we see that the addition of indexes has not changed the access path chosen by the Optimizer in this example. However, the Optimizer is using inaccurate information about the cost of using an index. How can we remedy this? As before, we perform a ″RUNSTATS ... AND INDEXES ALL″.

Version 4

- Primary Key Indexes
- Full Statistics

```
Access Table Name = SCHIEFER.STAFF   ID = 17   #Columns = 2
  Scan Direction   = Forward
  Relation Scan
  Lock Intent Share
  Sargable Predicate(s)
    #Predicates = 1
Access Table Name = SCHIEFER.STAFF   ID = 17   #Columns = 6
  Scan Direction   = Forward
  Relation Scan
  Lock Intent Share
  Sargable Predicate(s)
    #Predicates = 3
    ALL
    Create/Insert Into Sorted Temp Table   ID = t1
      Sort   #Columns = 1
      Piped
Sorted Temp Table Completion   ID = t1
Access Temp Table   ID = t1   #Columns = 5
  Scan Direction   = Forward
  Relation Scan
Nested Loop Join
  Access Table Name = SCHIEFER.ORG   ID = 16   #Columns = 2
    Scan Direction   = Forward
    Index Scan:   Name = SCHIEFER.PKORG   ID = 1   #Key Columns = 1
    Lock Intent Share
```

Notes to the foil: This version of the query produces a different plan to perform the join than previous versions.

Most of the plan fragments remain unchanged. We still scan the STAFF table to evaluate the subquery, scan the STAFF table again (applying the results of the subquery) and sort the remaining rows.

We still choose STAFF as the outer table in the join and ORG as the inner. Now, however, we use the primary key index on ORG to reduce the cost of joining the tables. Recall that the nested loop join evaluates the inner table for each row from the outer table. In order to avoid scanning the entire ORG table for each outer table row, we use the index to quickly pinpoint the matching row.

As before, the early sort of the STAFF table allows us to forgo any additional sorting for the ORDER BY and return the results directly. Thus, as a result of using the most accurate information possible, we have now started using the indexes we created. However, we used only the very simplest of heuristics to guide the creation of indexes. What else can we do? We can look at the query and decide what additional indexes might be helpful.

Version 5

- Additional Indexes
- STAFFIDX: JOB,SALARY
- ORGIDX: DEPTNUMB,DEPTNAME
- No additional Statistics

```
Access Table Name = SCHIEFER.STAFF  ID = 17  #Columns = 2
  Scan Direction  = Forward
  Index Scan:  Name = SCHIEFER.STAFFIDX  ID = 2  #Key Columns = 1
    Index-only Access
  Lock Intent Share
Access Table Name = SCHIEFER.STAFF  ID = 17  #Columns = 6
  Scan Direction  = Forward
  Relation Scan
  Lock Intent Share
  Sargable Predicate(s)
    #Predicates = 3
    ALL
    Create/Insert Into Sorted Temp Table  ID = t1
      Sort  #Columns = 1
      Piped
Sorted Temp Table Completion  ID = t1
Access Temp Table  ID = t1  #Columns = 5
  Scan Direction  = Forward
  Relation Scan
Nested Loop Join
  Access Table Name = SCHIEFER.ORG  ID = 16  #Columns = 2
    Scan Direction  = Forward
    Index Scan:  Name = SCHIEFER.ORGIDX  ID = 2  #Key Columns = 1
      Index-only Access
    Lock Intent Share
```

Notes to the foil: In this version of the query, we have added additional indexes to both the STAFF and ORG tables. Examination of the query indicated that we do not have a suitable index for evaluating the subquery. There is a sargable local predicate on the JOB column but no corresponding index. Also, the subquery references only two distinct columns so by extending the index to include the SALARY column we can achieve index-only access (only DB2/6000 shows this). As a result we add an index on JOB,SALARY to the STAFF table.

There is an analogous situation for the inner table of the nested loop join. There are no local predicates so we are really just linking information to the STAFF table. Since we are adding just two columns, again we can extend the index to include both required columns. So we add an index on DEPTNUMB,DEPTNAME to the ORG table.

Having added the indexes, we see that even without collecting additional statistics on these new indexes, they match the requirements of the query so well that they have been chosen anyway.

For this version of the query, we have given the Optimizer the opportunity to choose different indexes. For two of the three tables, the access path has changed. This indicates that our tuning efforts have likely been successful. There are always more things that can be done but we have already succeeded in radically altering the access plan to one that utilizes indexes and performs only a single sort.

Summary

EXPLAIN is a powerful tool

- Interprets packages containing static SQL
- Makes the Optimizer′s final decision visible
- Provides insight into possible tuning opportunities
- Provides invaluable help in trouble-shooting performance problems

Appendix A. DB2/6000 Explain Output

```
DB2/6000 Version 1.1.0, 5622-044 (c) Copyright IBM Corp. 1991, 1993
Licensed Material - Program Property of IBM
IBM DATABASE 2 AIX SQL Explain Function

******************* PACKAGE ************************************

Package Name = SCHIEFER.DYNEXPLN
        Prep Date = 1993/08/12
        Prep Time = 15:45:19:092

------------------- SECTION --------------------------------------
Section = 1


SQL Statement:
  SELECT S.ID, S.NAME, O.DEPTNAME, SALARY+COMM
  FROM ORG O, STAFF S
  WHERE O.DEPTNUMB = S.DEPT AND S.JOB <> 'Mgr' AND S.SALARY+S.COMM >
          ALL(
     SELECT ST.SALARY*.9
     FROM STAFF ST
     WHERE ST.JOB='Mgr' )
  ORDER BY S.NAME


Access Table Name = SCHIEFER.STAFF  ID = 17  #Columns = 2
  Scan Direction  = Forward
  Relation Scan
  Lock Intent Share
  Sargable Predicate(s)
    #Predicates = 1
Access Table Name = SCHIEFER.ORG  ID = 16  #Columns = 2
  Scan Direction  = Forward
  Relation Scan
  Lock Intent Share
  Sargable Predicate(s)
    #Predicates = 1
    Create/Insert Into Sorted Temp Table  ID = t1
      Sort  #Columns = 1
      Not Piped
Sorted Temp Table Completion  ID = t1
Access Table Name = SCHIEFER.STAFF  ID = 17  #Columns = 6
  Scan Direction  = Forward
  Relation Scan
  Lock Intent Share
  Sargable Predicate(s)
    #Predicates = 3
    ALL
    Create/Insert Into Sorted Temp Table  ID = t2
      Sort  #Columns = 1
      Piped
Sorted Temp Table Completion  ID = t2
Access Temp Table  ID = t2  #Columns = 5
  Scan Direction  = Forward
  Relation Scan
Merge Join
  Access Temp Table  ID = t1  #Columns = 2
    Scan Direction  = Forward
    Relation Scan
    Residual Predicate(s)
      #Predicates = 1
      Create/Insert Into Sorted Temp Table  ID = t3
        Sort  #Columns = 1
        Piped
Sorted Temp Table Completion  ID = t3
Access Temp Table  ID = t3  #Columns = 5
  Scan Direction  = Forward
  Relation Scan
```

Figure 1. Sample DB2/6000 Explain Output

Appendix B. Obtaining Catalog Statistics

```
SELECT T.NAME,T.CARd,T.NPAGES,T.FPAGES,T.OVERFLOW
FROM SYSIBM.SYSTABLES
WHERE T.NAME IN (list of tablenames)
ORDER BY 1;

SELECT DISTINCT C.TBNAME, C.NAME, C.COLNO, C.COLCARD,
                 C.HIGH2KEY, C.LOW2KEY, C.AVGCOLLEN
FROM SYSIBM.SYSCOLUMNS C
WHERE TB.NAME IN (list of tablenames)
ORDER BY 1,3;

SELECT I.TBNAME,I.NAME,I.UNIQUERULE,
       I.NLEAF, I.NLEVELS,
       I.FIRSTKEYCARD, I.FULLKEYCARD,
       I.CLUSTERRATIO, I.COLNAMES
FROM SYSTEM.SYSINDEXES I
WHERE TB.NAME IN (list of tablenames)
ORDER BY 1,2
```

Appendix C. References for DB2/2

- DB2/2 Guide (S62G-3663)

  EXPLAIN (Appendix J)
  Configuration Parameters (Chapter 8)
  Performance Tuning (Chapter 12)

- DB2/2 Command Reference (S62G-3670)

  syntax of RUNSTATS

- Comprehensive Database Perf. for OS/2 2.0 ES

  by Bruce Tate, Tim Malkemus, and Terry Gray
  Available from IBM as G362-0012
  ISBN 0-442-01325-6

- OS/2 Notebook

  edited by Dick Conklin
  Available from IBM as G362-0003
  ISBN 1-55615-316-3

Appendix D. References for DB2/6000

- DB2/6000 Admin. Guide (S609-1571)

  EXPLAIN (Appendix G)
  Performance Tuning (Chapter 9)
  Configuration Parameters (Chapter 10)

- DB2/6000 Command Reference (S609-1575)

  syntax of RUNSTATS