
Beyond Showplan

By
Brian Davignon

Corporate Headquarters	EMEA Headquarters
Embarcadero	Embarcadero Technologies,
Technologies	Ltd.
100 California Street	Thames House
12th Floor	17 Marlow Road
San Francisco, CA 94111	Maidenhead
www.embarcadero.com	Berkshire
	SL6 7AA
	United Kingdom
	www.embarcadero.co.uk

Table of Contents

What's Going on Here?3
Clauses.....4
About Embarcadero Technologies 11

What's Going on Here?

OK, so you've used showplan, statistics io, and statistics time to identify how your query has been optimized, and what it's costing you. Great, that's a good starting point. Sometimes all the answers are right there in the query plan or resource counts. They will point out a tablescan, the fact that your query is using a different index than you expected, or that most of your I/O relates to one particular table or SQL statement. You know you have an index on the search columns, so why did it choose a tablescan? You've done your homework and identified that one index is more selective than another, and yet the optimizer decided to access the table with the less selective index. You suspect that one join order is better than another, based on your knowledge of the data and relationships, but it didn't choose that join order. Unfortunately, these tools won't help you in those situations because they fail to answer the question 'Why?', they only answer the questions 'How?' and 'How much?'. You've reached the point in tuning your query where you need to dig into your other toolbox and break out the dbcc traceflags. This article will focus on traceflag 302, although it does include output from the 310 flag as well. The 302 traceflag provides information on index selection, while the 310 traceflag relates to join order selection. The definitions and commands used for this article are shown below.

```
create table abc      create table xyz
(a int not null,     (a int not null,
b int not null,      b int not null,
c int not null,      c int not null,
d int not null)      d int not null)
create index ix1      create index ix3
on abc(b,c)           on xyz(a,b)
create index ix2
on abc(d)
(200,000 rows)      (1300 rows)
dbcc traceon(3604,302,310)
set showplan on
declare @d int
select @d = 2000
select abc.a, abc.b, abc.d, xyz.c, xyz.d
from abc, xyz
where abc.b=xyz.a
and abc.c=xyz.b
and abc.d > @d
```

The output below shows that the optimizer is entering a scoring routine to determine how to access table 'abc'. Varno=0 simply means that it is the first table listed in the 'from' clause. Sybase uses a cost-based optimizer so it won't necessarily end up being first in the chosen join order. That cost is measured in terms of physical and logical I/O, which is really what this dbcc report is all about. How much did the optimizer estimate accessing a table in a certain manner would cost, and was it an accurate estimate? The report starts by displaying the table name along with page and row counts. These counts will be very close to actual values, possibly exact if dbcc checktable() has been run recently. They will be used by the optimizer as a basis for estimating total I/O.

DBCC execution completed. If DBCC printed error messages, contact a user with System Administrator (SA) role.

Clauses

Entering `q_score_index()` for table 'abc' (objectid 1954626552, varno = 0). The table has 198060 rows and 1579 pages.

The optimizer will then score each sarg, or search argument, provided in the query. Here it begins by scoring the 'Greater Than' clause, which references column 'd' of table 'abc'.

Scoring the SEARCH CLAUSE:
d >

The optimizer will always estimate the cost of doing a tablescan, regardless of what indexes are available. It may turn out that a tablescan will actually be faster than using existing indexes. For example, the table may only occupy 2 data pages, but every index access requires at least 3-4 pages to be read (root, intermediate, leaf, data). Your query may be returning a high percentage of the table's rows, which would certainly be faster using a tablescan. The optimizer's job is to find the least expensive access method given the tables, indexes, and query provided.

In the next few lines of the report you can see that an estimate is being performed against `indid 0`, which is always the base table, so this is an estimate of pages that need to be read for a tablescan. It estimates that 1579 pages will need to be read, but it also shows that these reads will be done with a 4K i/o size, which is 2 pages at a time, so it knows the actual cost will be 790 reads, not 1579 reads. Keep that in mind as you read through the output. The optimizer is trying to reduce the amount of reads the query must perform, and large I/O usage can have a large impact on the final estimate. If a large index is bound to a cache that contains a 16K buffer pool, and one that is half the size is not, a query like 'select count(*) from abc' would actually perform fewer reads by scanning the leaf level of the larger index. The reason is that you can read 8 times as many pages into cache with a single read. Most of your query's time is spent waiting in the i/o queue, not actually performing the read, so if you can reduce the amount of reads, you will reduce the amount of wait time. That's not to say that you should run out and start adding large I/O buffer pools all over the place, it's merely that the optimizer will consider things like that when generating its estimates. You can ignore the Relop bits, which is just a bitmap value representing the logical operator '>'.

Base cost: indid: 0 rows: 198060 pages: 1579 prefetch: S
I/O size: 4 cacheid: 0 replace: LRU
Relop bits are: 11

This next section states that my sarg is using a subquery, expression, or local variable. In my case it's a local variable, but in either case the problem is that the optimizer does not know the value of the sarg until runtime. It cannot be determined at optimization time, even though I clearly initialize it to 2000 before it gets to that statement. TIP: That problem can be easily rectified by moving the 'select' into its own stored procedure, then passing that value into the stored procedure after it has been initialized. That solution allows the optimizer to 'see' those values before the stored procedure is optimized. Unfortunately that is not how the code was written, so the

optimizer can only deal with what's in front of it. There are two options for estimating the I/O cost in this situation, using either the magicSC or the densitySC method.

The magicSC scoring method uses default percentages based on the logical operator. Using an equality operator (=), the optimizer estimates that 10% of the rows will be returned. The estimate for a closed end query (>= and <=, >= and <, > and <=, > and <) will be 25%, and for an open end query (>, >=, <, <=)" an estimate of 33% will be used. These estimates may very well be considerably far from the truth, but the optimizer has no way of knowing this because I used a local variable. It can only guess-timate.

The densitySC method can only be used if a valid distribution page and density table exist. The density table is located on the distribution page, and contains information on the percentage of duplicates across various combinations of key columns. If an index contains columns a and b, then an entry in the density table will exist for the percentages of duplicates for the combination of column a and b. This can be used to further determine an index's usefulness when a query contains a sarg for both columns.

SARG is a subbed VAR or expr result or local variable (constat = 60)--use magicSC or densitySC

The only index that contained a leading column used in the search clause is indid 3, so it is the only other access method evaluated besides the tablescan. The magicSC scoring was used for this query since not knowing the value of the local variable rendered the density table useless. The .33 selectivity of the index is a result of the open end query causing an estimate of 33% of the rows. Using no prefetch simply means that reads will be done at the default rate of 2K. The index height is 3, meaning that it takes 3 reads to traverse the index tree, plus a data page read. The optimizer has estimated that it will have to read a total of 65653 index and data pages and that it will yield 65360 rows. That's probably pretty far off, but it wasn't the optimizer's fault, it was the coder's fault for not providing a sarg that the optimizer could use to generate a valid estimate.

Estimate: indid 3, selectivity 0.330000, rows 65360 pages 65653 index height 3

This process of evaluating indexes continues for all indexes that could possibly satisfy the sarg, or that could be used to cover the query in the case of a non-matching index scan. The output would be similar to the above, but if a valid sarg is provided it would contain even more information about how its estimate was done. This will be looked at later when we see the dbcc report of the corrected query. Once it has evaluated each index, it will report on which index was deemed to be cheapest, along with other details of the access method. As you can see below, it chose the only index that was evaluated, indid 3, and indicates that it will cost 65653 pages, generate 65630 rows, and will use 2K I/O in the data cache with id=0, which is the default data cache.

Cheapest index is index 3, costing 65653 pages and generating 65360 rows per scan, using no data prefetch (size 2) on dcacheid 0 with LRU replacement Search argument selectivity is 0.330000.

After evaluating all of the sargs, the cost of accessing each table on the join columns must now be estimated. Once those numbers are in hand, the optimizer will have enough information to estimate the cost of performing the join in various orders. Note that it is estimating the cost of accessing table 'abc' and has indicated which join clause it is performing the estimate on. Once again it will evaluate the cost of doing a tablescan, which is considered the 'base cost'.

Entering q_score_index() for table 'abc' (objectid 1954626552, varno = 0).

The table has 198060 rows and 1579 pages.

Scoring the JOIN CLAUSE: b EQ a c EQ b

Base cost: indid: 0 rows: 198060 pages: 1579 prefetch: S I/O size: 4 cacheid: 0 replace: LRU
Relop bits are: 5

The index selectivity is derived by dividing the estimate of rows by the total rows in the table. The estimate of rows is discussed later when we cover the distribution page and density table. The optimizer has estimated that with an index height of 3, it will require 245 index and data page reads to perform a single iteration of the join. This means that if 10 rows qualify in the outer table, and table 'abc' is chosen as the inner table, it will require 2450 reads against table 'abc' and indid 2 to complete the join. The join selectivity is determined by multiplying the index selectivity by the total rows, and then dividing total rows by the result (198060 / (198060 * 0.001221)).

Estimate: indid 2, selectivity 0.001221, rows 242 pages 245 index height 3 Cheapest index is index 2, costing 245 pages and generating 242 rows per scan, using no data prefetch (size 2) on dcacheid 0 with LRU replacement Join selectivity is 819.187500.

Because the join can be performed with either table as the inner table, the optimizer will estimate pages, rows, and join selectivity for table 'xyz' as well. Based on a row total of 1309 and an index selectivity of 0.000763, the optimizer has estimated 1 row would be returned for each iteration of the join if this were the inner table chosen in the join. The cost would be 2 index pages and 1 data page for each row returned. Note the join selectivity is a higher number for this table.

Entering q_score_index() for table 'xyz' (objectid 1922626438, varno = 1). The table has 1309 rows and 24 pages. Scoring the JOIN CLAUSE: a EQ b b EQ c Base cost: indid: 0 rows: 1309 pages: 24 prefetch: S I/O size: 4 cacheid: 0 replace: LRU Relop bits are: 4 Estimate: indid 2, selectivity 0.000763, rows 1 pages 3 index height 2 Cheapest index is index 2, costing 3 pages and generating 1 rows per scan, using no data prefetch (size 2) on dcacheid 0 with LRU replacement Join selectivity is 1310.700000.

The fact that the row and page costs are significantly lower for table 'xyz' than they were for table 'abc' does not guarantee it will be chosen as the inner table. Those numbers only refer to a single iteration of the join. The sargs will determine how many iterations will be performed. If only 5 rows qualify for table 'xyz' based on sarg estimates, and it costs 245 page reads for each iteration of the join against table 'abc', the total cost of joining to table 'abc' is only 1225 pages. If 5000 rows qualify for table 'abc' based on sarg estimates, and it costs 1 page read for each iteration of the join against table 'xyz', the total cost of joining to table 'xyz' is 5000 pages. The optimizer must consider the cost of accessing each table based on sarg estimates, determine how many rows will be returned, and the cost of joining the qualifying rows to the inner table. The combination of these 3 things will determine the final cost of processing the query. You can imagine how much the optimizer must consider when there are multiple sargs and several tables involved in joins. The output generated by these traceflags can be enormous, which is why I used a simple example so we could focus on the individual components.

The 310 traceflag output below will have to be covered in detail another day, but for now I wanted to point out what type of information is contained in it. 'Query Is [Not] Connected' indicates whether or not the proper number of join clauses have been supplied to avoid a Cartesian product. The output shows that two join orders were evaluated, '0-1' and '1-0',

indicating the varno (table) order. The lp and pp values are logical and physical I/O estimates based on the estimated rows returned for the sarg, and the join selectivity for the join columns on the inner table. The report shows a second NEW PLAN cost, based on doing a tablescan on table 'abc' instead of using index ix2, but both estimates are based on the same join order, with table 'abc' as the outer table. The total cost is in milliseconds, and is based on reads necessary to access all qualifying rows in the outer table, reads necessary to perform the join, and includes both data and index page reads. 'Total Permutations' indicates how many join orders were considered, and 'Total Plans' includes all table access methods and join orders considered.

Query is Connected

0 - 1 -

NEW PLAN (total cost = 552319):

varno=0 (abc) indexid=3 (ix2)

path=0xe44f0128 pathtype=sclause method=NESTED ITERATION

outerrows=1 rows=65360 joinsel=1.000000 cpages=65653 prefetch=N iosize=2 replace=LRU

lp=65653 pp=1579 corder=4

varno=1 (xyz) indexid=2 (ix3)

path=0xe44f05d0 pathtype=join method=NESTED ITERATION

outerrows=65360 rows=65275 joinsel=1310.700000 cpages=3 prefetch=N iosize=2 replace=LRU

lp=196079 pp=24 corder=1

jnvar=0 refcost=0 reppages=0 reftotpages=0 ordercol[0]=1 ordercol[1]=2

NEW PLAN (total cost = 413513):

varno=0 (abc) indexid=0 ()

path=0xe44f0128 pathtype=sclause method=NESTED ITERATION

outerrows=1 rows=65360 joinsel=1.000000 cpages=1579 prefetch=S iosize=4 replace=LRU

lp=1579 pp=1579 corder=0

varno=1 (xyz) indexid=2 (ix3)

path=0xe44f05d0 pathtype=join method=NESTED ITERATION

outerrows=65360 rows=65275 joinsel=1310.700000 cpages=3 prefetch=N iosize=2 replace=LRU

lp=196079 pp=24 corder=1

jnvar=0 refcost=0 reppages=0 reftotpages=0 ordercol[0]=1 orderc

1 - 0 -

TOTAL # PERMUTATIONS: 2

TOTAL # PLANS CONSIDERED: 7

CACHE USED BY THIS PLAN:

CacheID = 0: (2K) 24 (4K) 1579 (8K) 0 (16K) 0

The 'Final Plan' is the plan that will be executed and it details the join order, page and row estimates, and the tablescan or index used for each table's access. It is followed by the showplan output, which indicates a tablescan was chosen for accessing table 'abc' first, followed by a join to table 'xyz' using index 3.

FINAL PLAN (total cost = 424170): varno=0 (abc) indexid=0 ()

path=0xe44f0128 pathtype=sclause method=NESTED ITERATION

outerrows=1 rows=65360 joinsel=1.000000 cpages=1579 prefetch=S iosize=4 replace=LRU

lp=1579 pp=1579 corder=0

varno=1 (xyz) indexid=2 (ix3)

path=0xe44f05d0 pathtype=join method=NESTED ITERATION

outerrows=65360 rows=65275 joinsel=1310.700000 cpages=3 prefetch=N iosize=2 replace=LRU

lp=196079 pp=24 corder=1

jnvar=0 refcost=0 reppages=0 reftotpages=0 ordercol[0]=1 ordercol[1]=2

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is DECLARE.

QUERY PLAN FOR STATEMENT 2 (at line 2).

STEP 1

The type of query is SELECT.

QUERY PLAN FOR STATEMENT 3 (at line 3).

STEP 1

The type of query is SELECT.

FROM TABLE

abc

Nested iteration.

Table Scan.

Ascending scan.

Positioning at start of table.

Using I/O Size 4 Kbytes.

With LRU Buffer Replacement Strategy.

FROM TABLE

xyz

Nested iteration.

Index : ix3

Ascending scan.

Positioning by key.

Keys are:

a

b

Using I/O Size 2 Kbytes.

With LRU Buffer Replacement Strategy.

The corrected query is shown below, followed by some new information that is now found in the dbcc output. This information shows that index ix2, with indid=3, is now being evaluated based on statistics that exist on the distribution page for that index, instead of using default percentages based on the logical operator. In the real world I would have broken the query into a separate proc, but the use of a constant in place of a local variable will suffice for demonstration purposes.

```
select abc.a, abc.b, abc.d, abc.e, xyz.c, xyz.g
from abc, xyz
where abc.b=xyz.a
and abc.c=xyz.b
and abc.d > 2000
```

The excerpt below looks similar to what we've seen previously, where the optimizer estimates the cost of a tablescan. However, since a valid sarg was provided, it is now able to more accurately estimate the number of pages and rows that match the sarg. This is something we haven't seen yet and is covered next.

Entering q_score_index() for table 'abc' (objectid 1954626552, varno = 0).

The table has 198060 rows and 1579 pages.

Scoring the SEARCH CLAUSE:

d GT

Base cost: indid: 0 rows: 198060 pages: 1579 prefetch: S

Relop bits are: 11

A distribution page is the place where Sybase stores index key value samples, and the optimizer will try to use these sample values to estimate how many rows will be returned. The output shows that a qualifying distribution page was found. That page number, 619713, is found in the column 'distribution' of the sysindexes table. The word qualifying should be taken lightly, since the information contained on the distribution page will only be accurate if the index was rebuilt or 'update statistics' was run recently.

A step is merely a sample interval. The number of steps that can fit on a 2K distribution page is determined by the size of the index key. This index was able to store 334 steps on its distribution page as indicated in the output.

Qualifying stat page; pgno: 619713 steps: 334

Search value: 2000

Let's talk about steps a little bit before we move on. Smaller index keys result in more steps, and therefore the estimate will generally be more accurate. If a table has 100,000 rows and the distribution page can hold 100 steps, then every 1000th key value will be stored, including the first and last key values. Depending on the logical operator used, the optimizer should therefore be no more than 2000 rows off on its estimate. If a smaller key is defined so that the same table's distribution page can hold 1000 steps, then every 100th key value will be stored and the optimizer's estimate should be no more than 200 rows off. For instance, if a query searches for a value of 5000, and the optimizer finds sequential steps contain values of 4500 and 5200, it knows that all rows with a key value of 5000 are contained within this one step range. If it also knows that it has stored every 1000th key value as a step, then it can estimate that it will return no more than 999 rows. The reason it is 999 and not 1000 is that if row 1000 contains key value 4500 and row 2000 contains key value 5200, there could be 999 values in between. Again, this is only an estimate, and the estimate is only as good as the number of rows divided by the number of steps (roughly), and also on how recent these statistics were updated. In this example, there are 198060 rows and 334 steps, so roughly every 592nd key value will be stored as a step.

Below we can see that an exact match was found on the distribution page. In fact, the search value was found on two steps. In this case, the optimizer uses the midseveralSC scoring method. In simple terms, it means there may be more rows with that value prior to the first step it found the value on, and there may also be more rows with that value after the last step it found the value on, so it considers that when generating its estimate. Based on that information, it has estimated 1480 rows will be returned at a cost of 1490 index and data pages. It has estimated that this is the cheapest. However, since the estimate is more accurate this time, it is not only the cheapest index, but is also less expensive than the tablescan. Therefore, it will be chosen as the access method for table 'abc' instead of a tablescan.

Match found on statistics page

equal to 2 rows on the statistics page in middle of page--use midseveralSC

Estimate: indid 3, selectivity 0.007473, rows 1480 pages 1490 index height 3

Cheapest index is index 3, costing 1490 pages and

generating 1480 rows per scan, using no data prefetch (size 2)

on dcacheid 0 with LRU replacement

Search argument selectivity is 0.007473.

I should also mention that, prior to ASE 11.9, the steps only contain values for the leading columns of the index. If you have a composite key made up of 2 columns and your sarg references both columns, the optimizer can use the density table mentioned earlier to more accurately estimate resulting rows. Remember that the density table contains a percent of duplicates found in combinations of composite key columns. If the optimizer determines that 500 rows fall between two steps for your search argument, and it also knows that the composite key contains 10% duplicates, it can estimate 50 rows instead of 500. Of course this estimate could be off as well because it is possible that all of your duplicates for this index fall outside the values contained in these steps.

I've omitted most of the remaining output since it was fairly redundant, but you can see below the new Final Plan, along with the more English-like showplan. This plan was chosen because the total cost is 30936, compared with the previous plan's total cost of 424170, so we can expect a significant savings on response time as well. This cost estimate is milliseconds, not reads, but is based on 18ms physical and 2ms logical I/O estimates. Whether those times are valid is trivial, since those same weights are used when evaluating each access method.

```
FINAL PLAN (total cost = 30936):
varno=0 (abc) indexid=3 (ix2)
path=0xe38cb928 pathtype=sclause method=NESTED ITERATION
outerrows=1 rows=1480 joinrel=1.000000 cpages=1490 prefetch=N iosize=2
replace=LRU lp=1490 pp=1046 corder=4
varno=1 (xyz) indexid=2 (ix3)
path=0xe38cbdd0 pathtype=join method=NESTED ITERATION
outerrows=1480 rows=1473 joinrel=1310.700000 cpages=3 prefetch=N iosize=2
replace=LRU lp=4438 pp=14 corder=1
jnvar=0 refcost=0 reppages=0 reftotpages=0 ordercol[0]=1 ordercol[1]=2
```

QUERY PLAN FOR STATEMENT 1 (at line 1).

STEP 1

The type of query is SELECT.

FROM TABLE

abc

Nested iteration.

Index : ix2

Ascending scan.

Positioning by key.

Keys are:

d

Using I/O Size 2 Kbytes.

With LRU Buffer Replacement Strategy.

FROM TABLE

xyz

Nested iteration.

Index : ix3

Ascending scan.

Positioning by key.

Keys are:

a

b

Using I/O Size 2 Kbytes.

With LRU Buffer Replacement Strategy.

There is a wealth of information that can be derived from the traceflags, too much to cover in this space, but hopefully this introduction has at least motivated you to the point where they become part of your tuning toolbox. Following are a few of the things that the traceflags can help point out, along with some possible reasons.

Optimizer did not find a qualifying stat (distribution) page

Is there not a valid index for the sarg?

Are your index statistics missing?

Large temp table not indexed?

An erroneous estimate was made for an index

Large amount of rows causing huge range of keys between steps?

Large key causing too few steps to be stored?

Too many columns or too large of a composite key size? This can produce a large density table, which reduces amount of step information that can be stored on distribution page.

Erratic key distribution in your table (5 rows for 1 key value, 100,000 for another key value)

A subquery, expression, or local variable was used as a sarg

As in our example, optimizer must rely on default percentages

As with any tool, the traceflags are only as good as the user. Using state-of-the-art automotive diagnostic tools won't turn you into a mechanic, but if you're already an accomplished mechanic, having better tools can certainly make you a more successful one and save you time and energy. Effective use of these traceflags requires that you have a firm grasp of index and table design, a basic understanding of distribution pages and density tables, as well as knowledge and experience in various tuning methods. It won't always be as easy as adding an index, updating statistics, or removing a local variable. But I think with these tools you will find it easier to get to the root of the problem, and that's the first step in solving it.

Brian Davignon is a Sr. DBA Consultant with Soaring Eagle Consulting, Ltd. in Tampa, FL. He has over 12 years of experience in development and database administration, including mainframe and client/server. Brian is a Certified DBA and Performance and Tuning Specialist (CSPDBA, CSPPTS) who has been working with Sybase and Microsoft server products since 1994. As a Certified Sybase Instructor, he taught hundreds of students on Intro and Advanced Sybase topics for over 3 years while also actively consulting. Brian can be reached at briand@soaringeagleltd.com.

About Embarcadero Technologies

Embarcadero Technologies, Inc. is a leading provider of strategic data management solutions that help companies to improve the availability, integrity, accessibility, and security of corporate data. Nearly 12,000 customers, including 97 of the Fortune 100, rely on Embarcadero Technologies solutions to maximize their return on corporate data assets and to meet the challenges of explosive data growth, escalating data security requirements, and complex, multi-platform data environments.

© 2006, Embarcadero Technologies, Inc. Embarcadero, the Embarcadero Technologies logos and all other Embarcadero Technologies product or service names are trademarks of Embarcadero Technologies, Inc. All other trademarks are property of their respective owners.