Sybase SQL “Show Plan” - Beginner's Introduction

By Dennis Adams

The Sybase “ShowPlan” utility is an invaluable tool for DBAs and developers to help identify the cause of long-running SQL statements, and aid better SQL design and/or Table & Index design.

This paper is intended to be a very brief introduction to “ShowPlan”, suitable for developers, or for DBA’s who may have had experience of other Relational Databases.

As with any introduction, this can only address the basic concepts. Once readers have understood these, they have a powerful tool for the world of Database Performance Tuning. Who knows, it could even become a full-time career !

The Sybase Query Optimizer

The Sybase Query Optimizer is a critical component of Adaptive Server.

SQL Queries are created by User Programs (including “isql”), and sent to the Adaptive Server via DBLIV or CTLIB calls. Once Adaptive Server receives the SQL statements, there are a number of operations which it needs to perform...

1. Parse the incoming SQL to check syntax
2. Verify that the tables in the query exist in the databases, and are accessible
3. Plan how to access the data
4. Compile the query
5. Access the data, and return the results.

Showplan is a powerful diagnostic tool for (3) - the access path for the actual data.

For example, if you wrote a simple query, such as...

```
1> select e.employee_name, d.department_name
2> from employee e, dept d where e.dept_no = d.dept_no
3> go
```

... there may be many different ways in which the data could be extracted. Just a few examples will suffice...

- Scan employee for all dept_nos and employee_name in a working table, and use the working table to read dept.
- Scan dept for all dept_nos and department_name in a working table, and use it to read employee
- If indexes are available, the working table could be joined to a clustered or non-clustered index.

It is the purpose of the Query Optimizer to determine the most cost-effective access path. It does this by estimating the amount of physical I/O required for each option, bearing in mind the fact that some pages will be cached in memory.

Showplan enables us to find out which plan the Optimizer has chosen, what indexes it intends to use, and which tables will be accessed most heavily.

Given this knowledge, we could alter our queries, create indexes, or alter our table design, caching etc. to improve application performance.[1]

Running ShowPlan

Showplan is very easy to run in ISQL. It needs to be run as a distinct transaction. At the same time, it may be a good idea to run “set noexec on”, so that the showplan is returned, but the server does not actually return the data. [2].

For the purposes of example, create a temporary heap table from a copy of the sysobjects, i.e..

```
1> select * into #dennis from sysobjects
2> go
(28 rows affected)
```

Then set showplan on, and enter a query on this table with a single *order by* clause.

```
1> set showplan on
2> set noexec on
3> go
1> select name from #dennis where type ='U'
2> order by name
3> go
```

At first glance, this looks a very simple query. A full table scan, perhaps ? In fact, showplan says that this query is broken down into two distinct steps. Both of these steps are output to the screen at the same time, but for simplicity, we can discuss each in turn. (The numbers on the right hand side are my annotations, for reference further on in this article)....

```
QUERY PLAN FOR STATEMENT 1 (at line 1).

    STEP 1
        The type of query is INSERT.                                   1
        The update mode is direct.                                     2
        Worktable1 created for ORDER BY.                               3

        FROM TABLE
            #dennis
        Nested iteration.                                              4
        Table Scan.                                                    5
        Ascending scan.
        Positioning at start of table.                                 6
        Using I/O Size 2 Kbytes.                                       7
        With LRU Buffer Replacement Strategy.                          8
        TO TABLE
            Worktable1.                                                9
```

The query optimizer has, as expected, decided to do a ***Table Scan (5)***. This is because the table is a simple heap, with no clustered or non-clustered indexes.

[1] Do not underestimate the effect of changing indexes, or SQL statements. The author has seen SQL runtimes changed out of all proportion by re-designing indexes or re-phrasing SQL statements. Response time improvements of 100% to 500% are not uncommon.
The difficulty comes with identifying the SQL statements which are having the most impact on your application, and addressing them systematically.
As with many things, the 80/20 rule applies - 80% of your performance problems can often be traced to 20% or less of the application. In some cases, by changing just 5 SQL statements, you can improve performance out of all recognition.

[2] CARE: “set noexec on” is a very dangerous command. Once you have run it, all other following commands will be “noexec’d”. For example, if you are doing testing, there is no point creating a table index if set noexec is on !. Even experienced DBAs can make this mistake.

This stop will be written ***TO TABLE Worktable1 (9)***, i.e. a working table will be created.

Why has the optimizer decided to do this ? Recall that the query has an "order by" clause. Since there is no index structure on #dennis, the only way Adaptive Server can return the results is to create a sorted work table. Hence the phrase ***Worktable1 created for ORDER BY (3)***.

We will discuss some of the other aspects of this later on. For now, the second step shows how this working table will be processed...

```
    STEP 2
        The type of query is SELECT.                                10
        This step involves sorting.

        FROM TABLE
            Worktable1.                                             11
        Using GETSORTED                                             12
        Table Scan.
        Ascending scan.
        Positioning at start of table.                              13
        Using I/O Size 2 Kbytes.                                    14
        With MRU Buffer Replacement Strategy.                       15
```

This is the final ***SELECT (10)*** operation which will return the data to our isql session by scanning ***Worktable1 (11).***

Recall that the worktable was created FOR ORDER BY. The second step involves sorting the data so that we can return it in the correct sequence (see ***3, 10, 12***)

This gives us the basic understanding of how Adaptive Server executes the query.

However, there is further additional information which can give us clues about caching and I/O etc.

Update Mode (2) of Direct indicates that ASE is performing it's normal update operation, i.e. Record each change in the Transaction Log, followed by the Data Pages. (The Transaction Log is flushed to disk when the transaction is committed.)

In some cases (e.g. when an insert-select is made simultaneously on the same table), ASE does a Deferred Mode update. This involves writing the entire list of Inset/Delete Entries to the Transaction Log first, then re-scanning the Transaction Log to do the actual processing. This is obviously slower, but in some cases it cannot be prevented.

I/O Size (6 and 14) is 2 Kbytes in this case. This is the default. ASE can be reconfigured so that the I/O size is increased. Using 16K I/O size, for example, may be a good idea for Data Warehouse or other complex query applications. You can change the I/O sizing by creating different caches, and binding tables or entire databases to them. [3]

This leads to look in more detail at the way data is accessed, particularly the Positioning at strategy ***(6 and 13).*** When doing a sort, the optimizer has simply said to start at the beginning of the table. However, if we were using an index, we might expect to see Positioning by key - i.e. The index was used to find the qualifying row before starting a scan or sort.

Alternatively, Positioning by Row Identifier (RID) could be used if ASE was intending to create a dynamic index of RIDs before processing them.

Finally, we can consider the effect of Data Caching....

The First Step was using ***LRU Buffer Replacement (7).*** To understand this, you need to visualise the Data Cache as a (almost) fixed-size sequential list in memory. The most recent pages are added to the "top" of the list, and the oldest at the "bottom".

---

[3] The command sp_cacheconfig can be used to create and change cache(s), and sp_bindcache to bind database objects to named caches.

LRU Buffer Replacement means that our pages are being read onto the “top” of the cache, and are “forcing out” the older, staler, least-recently-used cache pages on the bottom.

An alternative strategy is the MRU. This involves putting our new pages near the middle or bottom of the cache (where they would soon disappear again in the normal course of operations). There is an advantage in this - by not disturbing the pages at the top of the cache enables them to be preserved for other users.

Notice that Step 2 of this Showplan is using ***MRU Buffer Replacement (15).***

Other Showplan results

The above was a very simple example, but should help people interpret the showplan results. One further example illustrates a way in which indexes can affect the I/O behaviour. To do this, create a non-clustered index. (Don’t forget to “set noexec off” first !)..

```
1> create index #dennis_idx on #dennis (type)
2> go
```

This query involves the index column, a “group by” and “order by” clause...

```
1> select type, count(type) from #dennis group by type order by type
2> go
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).

    STEP 1
        The type of query is SELECT (into Worktable1).
        GROUP BY
        Evaluate Grouped COUNT AGGREGATE.
        FROM TABLE
            #dennis
        Nested iteration.
        Index : #dennis_idx
        Ascending scan.
        Positioning at index start.                                16
        Index contains all needed columns.                         17
                  Base table will not be read.
        Using I/O Size 2 Kbytes.
        With LRU Buffer Replacement Strategy.
        TO TABLE
            Worktable1.

    STEP 2
        The type of query is SELECT.
        FROM TABLE
            Worktable1.
        Nested iteration.
        Table Scan.
        Ascending scan.
        Positioning at start of table.
        Using I/O Size 2 Kbytes.
        With MRU Buffer Replacement Strategy.
```

By now, you should have some idea of what some of the lines mean.

As before, the optimizer has decided to create a temporary work table for sorting purposes. This is because we created a non-clustered index - i.e. the index is built distinct from the data.

As a further exercise, try creating a clustered index - the data will be automatically sorted in the correct sequence by the index clustering. Therefore, you will not require any sorting.

Notice also the Positioning clause (***16***) and comment about the Index (***17***). Since we had only specified "type" in the target list of columns, the optimizer had correctly deduced that it is unnecessary to go back to the original table at all. If this had been a huge table, the extra efficiency of *Index Covering* could be substantial.

Of course, these are very simply queries. With more complex queries, there are far more permutations.

Supposing an insert, update or delete causes a trigger to fire, which in turn accesses the inserted or deleted tables. The ASE has to identify the table contents by scanning the transaction log. This is shown as a ***Log Scan*** in showplan.

Performance Hints and Tips

The purpose of Showplan is to enable us to understand what ASE is doing internally to process our queries, and hence help us to re-design them to improve efficiency and performance. Here are a few comments on things to look out for...

The previous example showed where an index could be added to a table to enable us to extract information quickly, by relying on the index alone. This is called *Index Covering*. If you have a multi-column index (e.g. 6 columns), you do not need to specify all 6 columns, but you *do* need to specify at least the first (primary) column in order to get Index Covering.

We also discussed how using a clustered index could reduced the requirement for sorting. This could be a major benefit in certain reporting databases.

In most cases, it can be far more efficient for the Server to read an Index, and then extract the data by RowId, rather than scan an entire table. There are exceptions, of course. If a table is small (or if the query is expected to read a large percentage of it), it is easier to scan the table once, rather than taking the "double hit" of reading index pages first before reading data pages.

But how does the optimizer decide when to use (or not use) an index in the first place ? The answer is whether or not it finds a valid SARG (Search Argument) in the where clause. If a SARG is found, the optimizer will use clustered or non-clustered indexes if available.

The following rules are required to make a valid SARG in the where clause...

- Specify the column name without any functions, i.e.

```
type = "U"                     is valid, whereas
substring(type,1,1) = "U"      is not
```

- Use a scalar operator, such as "=", ">=", "<>", "is null", for example..

```
type !< "D"                    is valid
```

- Use a constant (or something that evaluates to a constant) in the where clause.

Using indexes can make a significant contribution to I/O and Cache Performance (not to mention associated locking contention as well). However, do not forget that indexes can also go "stale".

As more and more data is added to your tables, (and other data is archived off), your symmetrical binary tree index can become more and more "skewed". After long periods of this treatment, an index can become so inefficient that it is actually slower to use an index than to scan the entire table.

The solution to this is to put database housekeeping in place to regularly remodify tables which are subject to lots of inserts and deletes.

At the same time, you need to update the statistics on each table. This is one aspect of database housekeeping that can be easily forgotten. Briefly, it is the table statistics which are used by the query optimizer to determine the distribution of the data. Together with the SARG information, this enables the optimizer to make it's evaluation of the most efficient query plans.

Obviously, if the table has changed (new transactions added), the data distribution will change, and the query optimizer may make the wrong choices.

The statistics are gathered using a simple syntax. For example, to gather all the statistics on the "department" table...

```
1> update statistics dept
2> go
```

One bit of good news - when you update a clustered index, the statistics are automatically updated for you. (If the table has a non-clustered index, however, this is not the case).

Also, when you update the statistics on your tables, remember to run sp_recompile to recompile the procedures to take advantage of the new statistical knowledge which you have just gathered.

More Complex Query Plans

By definition, this article is only a brief introduction to Showplan.

There are many different types of Mode and Access types which we have not covered. For example, Deferred Updates can also be done as deferred_varcol or deferred_index. These are special forms of processing for updating varchar columns (which may involve page-splitting), and updating index columns.

Some issues, such as index matching and how sub-queries are processed, require much more discussion, and we have not discussed Parallel Queries at all.

But hopefully this article will help someone to begin to experiment with showplan, and have a better understanding of some of the articles and books on Sybase Tuning.

Showplan and other utilities

As I mentioned at the beginning of this article, showplan is a very powerful tool. But it is not just the only tool available to the DBA. There are others, some of which can be used in conjunction with showplan. Others would require an article in their own right....

Utility	**Shows ..**
set statistics io	number of physical and logical reads required by each query.
set statistics subquerycache on	number of cache hits and misses, and rows in cache for each query.
set statistics time on	time taken to parse and compile each command and time taken to execute each step of the query.
dbcc traceon	diagnostics explaining why alternative plans were chosen
sp_helpcache	summary information on data caches and objects bound to them
sp_sysmon	cache configuration statistics including cache utilisation and disk I/O patterns

In addition, there are a number of graphical tools available on the market which enable DBAs to extract showplan and other data easily.

But whether you are using a GUI tool or not, it is still valuable to know about how showplan works.

The key (pardon the pun !) is to be careful about interpreting the results. Try it on some of the queries which you are more familiar with first. Don't jump to conclusions. And don't change more than one thing at a time.

But if you think you have seen a performance issue, test it out with showplan. You may have found a "silver bullet" for your application team. Who knows, you may be pleasantly surprised by your own ability to make your server perform better.

Sources

There are literally hundreds of books on the Sybase Database,as well as the Sybase Manuals themselves. For the preparation of this article, I relied primarily on the Sybase source;

Sybase Adaptive Server Enterprise Performance and Tuning Guide. Document ID 32645-01-1150-02.

Dennis Adams is a Database Systems Consultant. A member of the British Computer Society, he can be contacted via email at Dennis.Adams@bcs.org.uk.