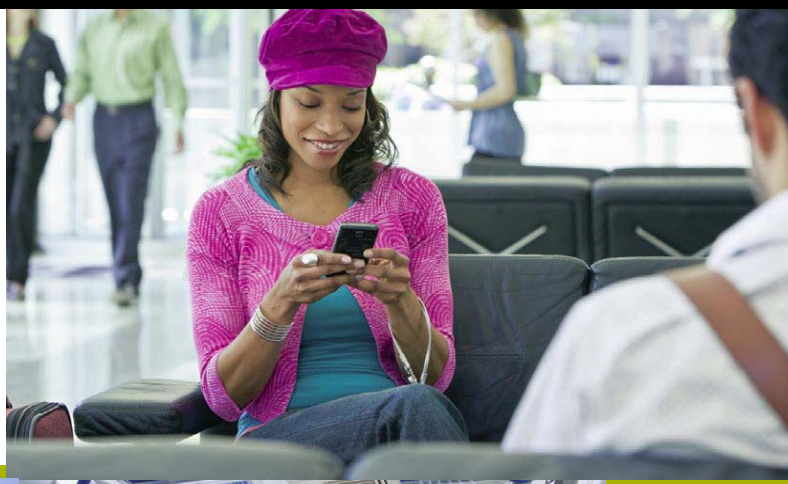
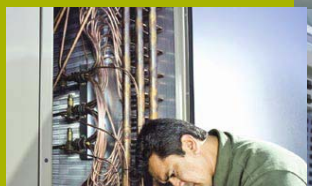
SYBASE®

Sybase ASE 15 Best Practices: Query Processing & Optimization

TECHNICAL WHITE PAPER



www.sybase.com



Contents

Principal author

Rob Verschoor

Contributing authors

Sudipto Chowdhuri
Bill Cox
Mark Kusma
Nitin Sadalgekar
Peter Thawley
Raymundo Torres
Ningzhen Zhu

Revision History

Version 1.0	February 2008	First version
Version 1.1	May 2009	Cover Compatibility Mode in 15.0.3 ESD#1; also, minor editing.

Executive Summary

Historical Overview

When Sybase first began to design its relational database management system in 1984, the world of Information Technology (IT) was radically different – applications were monolithic with data access code and business logic tightly bound; transaction processing (OLTP) was predominately a batch process; and decision support (DSS) reporting never, ever, ran at the same time as transaction processing. Sybase's pioneering concept of client/server computing was based on two simple ideas. First, many businesses could offer better service and value to their customers if transaction processing were done "on-line". Second, businesses could service their customers better, and therefore grow faster, by decentralizing application development and not using the mainframe. To enable this radical approach, Sybase had to design and build a substantially different type of database kernel to support this new "on-line enterprise".

Sybase Adaptive Server Enterprise (i.e., the product formerly known as the Sybase SQL Server) became the leading RDBMS product for high performance transaction processing supporting large numbers of client application connections on relatively small, inexpensive hardware. To be successful in this demanding area, Sybase focused heavily to optimize and streamline all the layers of our database engine for transaction processing workloads. While this focus turned out to be the asset that catapulted Sybase to a dominant position in high-end OLTP applications during the 1990's, it increasingly became a liability in some environments as application workloads have changed and hardware performance has increased.

With more and more enterprises requiring a mixture of both transaction processing and operational decision support reporting on the same data at the same time, it became clear that Sybase needed to re-architect its query processing layer for the future. Sybase began the process in the 11.9.2 release when we re-architected the "bottom half" of the query optimizer that dealt with the statistics and algorithms that drive query costing. We then embarked upon a significant effort to replace the top half of the optimizer and the query execution layer with one that could meet the next generation of application requirements. Equally important, we needed it to be more extensible so that in the future, we could add support for new index types or new join strategies to more quickly respond to the needs of our customers. With ASE 15, Sybase has delivered this.

With this historical context in mind, it is important for customers and partners who are looking to leverage the many benefits of ASE 15.0.x understand the significance of these changes. A good analogy for how to approach this effort is just like breaking in the engine on your new car. Should you wind up the engine's RPMs to that red line and cruise your favorite winding back road at 100mph? As much as some of us would like to say yes, I suspect most of us know the right answer. Well, using the new ASE 15 query optimizer and query execution is similar – go slower than you (or your boss) wants, learn the important characteristics of the system, and know what to do in case the red light comes on. This document is meant to help you with these last two!

What's New with the ASE 15 Optimizer and Query Processing Engine

The first and most important thing to know about ASE 15's query optimizer and query execution engine is that it's basically totally new. The shift in requirements over the years towards mixed workloads (i.e., concurrent OLTP and DSS application profiles) stretched our traditional relational processing methods of "nested loop" joins to their breaking point. If you consider the fact that it is now common to see queries joining ten's of tables with many complex search arguments and/or aggregation, it was clear that both the query optimizer and the query execution engine needed a major overhaul. More importantly, we needed to design a solution to ensure both the optimizer and execution engine would be extensible. This allows us to add new and innovative algorithms (for joining, grouping, union, etc) and storage structures (e.g. index types) more easily without kludging existing code and making it unmanageable or error-prone. To facilitate the modularity and extensibility of the query execution engine, we have rebuilt the execution engine using a more modern, consumer-driven, iterator-based execution model. This model implements relational operators as primitive building blocks and provides both the ability to easily solve nagging limitations as well as add important new join strategies and optimizations. Before we get into the details of how to use and troubleshoot the new optimizer and query processing engine, let's first gain an understanding of the major changes which affect both applications and operations.

Long-time ASE customers will appreciate the elimination of limitations which all of us have suffered through. Over the years, Sybase has steadily been reducing the occurrences when data type mismatches would prevent a predicate from being considered during optimization. Unfortunately, our efforts were "hardwired" so we had to add special case code for each and every case. Although not "rocket science", it was a resource-intensive approach that did not scale as we added new data types. More interestingly, how many times have you wished the optimizer and QP engine could use multiple indexes on a table through index "union" or "intersection" operations? Well, wish no more! Now, queries with AND and/or OR predicates can potentially use multiple different indexes based on data distribution statistics' estimated selectivity for each predicate.

How about the dilemma we faced in the past with optimization of queries that joined large numbers of tables resulted in a "less than stellar" query plan due to join costing that did not use real data distribution statistics (histograms) but rather "magic numbers" from our engineer's heads? Now, statistics for joins, both to derive join order as well as to feed more accurate row estimates into costing of inner tables of the join order, should be more accurate because we dynamically create join histograms at optimization time.

Why is all this important? Well, frankly speaking, it's to ensure that you don't treat ASE 15 as "just another release". As much as we would like to say that you could simply upgrade and point your applications at the upgraded servers, the depth and breadth of change in one of the most fundamental areas of a database, query execution, necessitates a more focused testing regimen. This paper is meant to provide you with the clear facts and best practices to reduce this effort as much as practically possible.

Understanding ASE 15's Query Processing

What is an optimization goal?

A central concept of ASE 15's new query processing engine is the 'optimization goal'. In essence, this can be seen as a hint to the ASE query optimizer, providing an indication of the nature of the query being optimized. To illustrate the underlying idea, a typical OLTP query and a typical DSS query will normally end up with very different query plans due to the different data access patterns used by these queries. OLTP queries tend to hit one or just very few rows, and join only a few well-indexed tables; DSS typically hits many rows but returns just a few, and can join a large number of tables.

Because of their different access patterns, OLTP queries will often perform best when they use a classic nested-loop join whereas a DSS query is more likely to run faster with a hash join. By indicating that a query is for OLTP or DSS purposes, the optimizer may save itself a lot of work (time, memory, CPU) when generating a query plan.

ASE 15 provides three optimization goals (ordered from 'narrow' to 'wide', corresponding to the number of options and strategies they allow the optimizer to consider):

- **allrows_oltp** (best for OLTP queries)
- **allrows_mix** (default after upgrading to ASE 15)
- **allrows_dss** (best for DSS queries)

The different optimization goals have the following effect on the ASE query optimizer:

- **allrows_oltp** offers the narrowest selection of join methods: the query optimizer is allowed to consider nested-loop joins only
- **allrows_mix** allows also merge joins to be considered by the optimizer, as well as parallel plans (if the ASE server is configured for parallelism).
- **allrows_dss** offers the widest selection of join methods: also hash joins may be considered by the optimizer

For **allrows_mix** and **allrows_dss**, additional low-level processing algorithms are enabled which are disabled for **allrows_oltp**.

It should be noted that when the optimization goal is widened, the query optimizer might use significantly more resources (time and procedure cache) to generate a query plan. If the optimizer generates the same query plan, with only nested-loop joins, under **allrows_dss** and **allrows_oltp**, you should expect the optimization under **allrows_dss** to take more time and procedure cache than under **allrows_oltp**.

The optimization goal can be defined on the level of the ASE server, session or individual query:

```
-- server-wide default:
sp_configure 'optimization goal', 0, 'allrows_dss'

-- session-level setting (overrides server-wide setting):
set plan optgoal allrows_dss

-- query-level setting (overrides server-wide and
-- session-level settings):
select * from T1, T2 where T1.a = T2.b
```

```
plan '(use optgoal allrows_dss)'
```

The session-level optimization goal can also be set in a login trigger; see page 8 for more information.

Optimization criteria

An optimization goal is actually shorthand for a collection of on/off settings for a series of properties known as 'optimization criteria'. Individually, an optimization criterion allows (when it is enabled) or disallows (when disabled) the optimizer to consider a particular algorithm (as for access methods, for joins, for grouping, sorting, etc.). Some examples of optimization criteria are:

```
set hash_join on      -- enables hash joins
set store_index off   -- disables reformatting
```

Note that Sybase does not recommend using explicit settings for optimization criteria unless advised by Sybase Technical Support, or as part of a troubleshooting process. In production code, Sybase recommends using optimization goals instead unless advised differently by Technical Support.

Choosing the best optimization goal for your application

The choice of optimization goal can have a big impact on the performance of your queries. Therefore, two important questions must be answered when upgrading to ASE 15:

1. Which optimization goal should be set server-wide?
2. Is it necessary to specify session- or query-level optimization goals that are different from the server-wide setting?

Since it should be considered unfeasible to analyze all queries in your system, a pragmatic approach should be taken towards these issues. Please note that blindly setting the server-wide optimization goal to `allrows_dss` (since it is the widest optimization goal) is not recommend since this may consume more resources, including compilation time, than needed. It is worth trying to determine if a narrower optimization goal will do.

As for the choice of the server-wide optimization goal, two approaches can be distinguished:

A. Pick the optimization goal that provides the best overall performance for your system.
   This should be set so that it matches the type of workload of the majority of your queries. If you have a pure OLTP or pure DSS system, then it is obvious which optimization goal to pick.
   Ideally, you should experiment with the different optimization goals to see which one provides best results for your mix of applications and queries. See the section "Experimenting with optimization goals" for some suggestions.

B. Just set the server-wide optimization goal to `allrows_oltp` -- for the simple reason that this optimization goal resembles the behaviour of ASE 12.x most closely. This will reduce the possibility of running into unexpected query plan issues after upgrading to ASE 15, but may also not let you benefit from the full potential of ASE 15's capabilities.

The second issue (the need for session- or query-specific optimization goal settings) may require more work. When you know that certain applications have different workload characteristics than the rest of your system, for example, some OLTP workload in a mostly-DSS system, then it may be a good idea to set the appropriate session-level optimization goal for that application. In this example, that would mean the server-wide optimization goal would be set to `allrows_dss` while the session-level optimization goal would be set to `allrows_oltp`.

Note that this can quickly become an open-ended undertaking: there is almost no limit on the amount of effort you can spend on analyzing your queries and deriving the best optimization goal setting for a session or an individual query. As always, you have to strike a balance between performance gains and effort spent when engaging in performance tuning.

Experimenting with optimization goals

As explained above, you need to pick an optimization goal to set as your server-wide default. The easiest approach is just to pick one, see how things behave overall, and try another one to see if that's any better. When using stored procedures, make sure to run `sp_recompile` on all tables, or reboot ASE, after changing the server-wide optimization goal in order to make it take effect by recompiling existing plans. Also, make sure the statement cache is disabled to get a good comparison.

While this approach may work, it is perhaps a good idea to make a more informed decision about the optimization goal to use. Fortunately, this is not too difficult. First, you need to collect or capture a representative set of real-life SQL queries, for example, through the MDA tables, or with auditing or Monitor Server (for some suggestions, see page 32). Put all these queries in a file as separate batches, with 'go' to terminate every batch, just as you'd do when using `isql`. Then, put the following at the start of the file:

```
set plan optgoal allrows_oltp
go
set showplan on
set statistics time, io on
go
…rest of file…
```

Now run this file through **isql** and capture the output in another file:

```
isql -U username -P passwd -S YOUR_15_SERVER -e -i your_file > output_file
```

Time how long it takes before the run has completed.

Repeat this with the optimization goal above changed to `allrows_mix` and `allrows_dss`, so that you have three output files of query plan output.

From the completion times of the different runs, it may be immediately apparent which optimization goal provides the best result overall, and that might be a candidate for the server-wide optimization goal setting.

However, it will be interesting to count how many times each join type has been chosen. This can be done quickly with `grep` for each of the output files:

```
grep -c "NESTED LOOP JOIN" <filename>
grep -c "MERGE JOIN" <filename>
grep -c "HASH JOIN" <filename>
```

Suppose there appear to be just a few hash joins used when the optimization goal was set to `allrows_dss`. For the queries using a hash join, it would be worth checking if they ran much slower with the other optimization goals. With the approach above, this will be relatively easy to compare - since `set statistics time` was enabled, the elapsed time for each query will be in the output file. If it turns out that, for those specific queries,

hash joins make a big difference indeed, you could consider setting the session-specific, or even query-specific, optimization goal for those queries to `allrows_dss`. If, on the other hand, there appear to be many queries using a hash join, and allrows_dss turns out to provide very good performance overall, perhaps it is better to set the server-wide goal to `allrows_dss` instead.

With this approach, it is relatively easy to get an idea of the impact of using different optimization goals. Of course, the quality of the results depends on how representative the sampled queries are.

When running tests like these, make sure the statement cache is disabled (either by setting the configuration parameter `"statement cache size"` to 0, or by running `set statement_cache off` at the start of your session). Also, when using stored procedures before re-running with a different optimization goal, run `sp_recompile` on all tables, or reboot ASE, after changing the server-wide optimization goal so as to make sure the new goal takes effect.

When comparing performance of queries under different optimization goals, especially when a reboot is involved, make sure to test under comparable situations with respect to data cache usage. It does not make sense to compare a query where most data needs to be read from disk, with a query where most data is already in the data cache. In general, it is best to compare queries with a 'warm cache', whereby the query is run once or twice without measuring its performance, followed by the final measurement run.

How to set session-level optimization goals

Let's assume you have determined that a particular application would benefit from using `allrows_mix` while the server-wide goal is set to `allrows_oltp`. How do you implement this? In its simplest form, you modify the application to make it execute the statement `set plan optgoal allrows_mix` before submitting any other statements. However, it may not be possible, or impractical, to modify the SQL code submitted by an existing application.

Fortunately, there is an easy solution: you can set the session-level optimization goal in a login trigger with `set plan optgoal`. This way, the optimization goal for a session can be set as desired, without changing any existing application code. Please refer to the ASE System Administration Guide for more information about login triggers: chapter "Managing User Permissions", section "Row-level access control", subsection "Using login triggers" (though please note that login triggers can be -and usually are- used independently of row-level access control, as is indeed the case for setting session-level optimization goal as described above).

Parallel query processing in ASE 15

Since version 11.5, ASE has supported intra-query parallelism, whereby a single query is processed by multiple worker processes. Many Sybase customers have used parallelism, usually to improve response times for DSS-type queries. For this type of query, where a large number of rows are accessed but only a small result set is returned, parallelism tends to be most efficient in pre-15.0. Parallel query processing tends to require significantly more resources (CPU, memory, disk I/O) than serial processing, but when such resources are available parallelism may deliver better performance for specific queries.

ASE 15 also supports parallelism, but in a different way than pre-15 versions. Without going into too many details that would exceed the scope of this document, in summary we can say that parallelism in ASE 15 can occur at more levels inside a query than in pre-15. In addition, parallelism in ASE 15 is designed for use in conjunction with semantically partitioned tables, which pre-15 did not support.

Since the new query processing features of ASE 15 offer potential performance benefits for DSS-type queries in particular, Sybase recommends to initially not use parallel processing when upgrading to ASE 15, even when parallelism was used in ASE 12.x. Since serial processing is more resource-efficient than parallel processing, avoiding parallelism may allow you to deliver better overall performance with the same hardware. It may well be, and has indeed been observed by customers, that ASE 15 in serial mode runs queries faster than ASE 12.x with

parallelism. For these reasons, best explore ASE 15's capabilities in serial mode first before looking into parallelism. For DSS-type queries, where parallelism was often deployed in ASE 12.x, it is specifically suggested to look into the effects of the optimization goal `allrows_dss`.

Parallelism may deliver better response times than serial processing for queries using semantic table partitioning in ASE 15, or for DDL commands (like `create index`). For such cases, it may be worthwhile to explore the impact of parallelism in ASE 15.

Performance monitoring tips

This section contains some suggestions for monitoring performance aspects of your queries. These require the MDA tables to be installed and enabled.

- To determine if any sessions are currently executing queries that consume a lot of disk I/O, use `sp_monitor`:

  ```
  sp_monitor "connection", "diskio"
  ```

  To find sessions with long-running queries, replace `"diskio"` by `"elapsed time"`. To find CPU-intensive queries, replace `"diskio"` by `"cpu"`. Note that `sp_monitor` has many other handy options; use `sp_monitor "help"`, and see the ASE documentation for details.

- To identify the top-20 of recently executed statements with respect to Logical I/O, use this query:

  ```
  select top 20
  SPID, CpuTime, LogicalReads, PhysicalReads,
  ProcName=object_name(ProcedureID,DBID),
  MemUsageKB, StartTime, EndTime,
  DurationSecs = datediff(ss, StartTime, EndTime),
  KPID, BatchID
  from master..monSysStatement
  order by LogicalReads desc
  ```

  To find the top-20 queries for CPU consumption, change the order by clause to sort on `CpuTime` instead. To find the top-20 long-running queries, sort on the duration (`"order by 9 desc"`).

  To find the SQL text corresponding to these statements, use the `KPID` and `BatchID` columns to query the `monSysSQLText` table. Note that `monSysSQLText`, like `monSysStatement`, is a so-called "historical" MDA table, meaning that it contains only a limited amount of history. To capture information from these tables over longer periods of time, you should frequently select from these MDA tables and store the result set in a permanent table.

Troubleshooting tips

ASE 15 was designed to offer a greatly improved query processing environment. In the rare event that query plans or query performance are not what you expect, here are some things to try to isolate the problem.

- When experimenting with different optimization goals, make sure no cached plans are used: changing the session-level or server-wide optimization goal will not cause cached plans to be recompiled.
  For stored procedures, either execute them `with recompile`, or run `sp_recompile` on one of the

tables being accessed. For batches, make sure the statement cache is disabled by running `set statement_cache off` first.

- When using different session-level optimization goals in a single batch or stored procedure (i.e. `set plan optgoal` statements), ensure to be running 15.0.2 ESD#2 or later. This is recommended because some aspects of session-level optimization goals were changed in that release, making the feature more intuitive and easier to use.
  Full details on the scope and semantics of session-level optimization settings, will be described in another whitepaper, titles *"Changes to scope and semantics of session-level optimization settings in ASE 15.0.2"* (see URL on page 41).

- When you want to guarantee that a stored procedure is always optimized with a particular optimization goal, irrespective of any server-wide or session-level settings, put `set plan optgoal allrows_xxx` as the first statement in the body of the stored procedure. This requires ASE version 15.0.2 ESD#2 or later.

- If it is suspected that merge joins are causing sub-optimal performance in your system, and a server-wide optimization goal is `allrows_dss`, you could change this setting to `allrows_oltp`. However, this also implies that hash joins can no longer be used.
  In 15.0.2 or later, if you don't want to lose the possibility to use hash joins, you can also keep `allrows_dss` as the server-wide goal, and set the configuration parameter `'enable merge join'` to 0; this will disable merge joins server-wide for all optimization goals.

- If your SQL code from ASE 12.x contains explicitly forced join orders (with `set forceplan`), these forcings should ideally be re-evaluated when upgrading to ASE 15 (as Sybase has always recommended when talking about such forcings). Of course, if your system runs to full satisfaction in ASE 15, there may be no need to change existing forcings. However, they may also stop you from taking full benefit of the capabilities of ASE 15. It is therefore recommended to evaluate the impact of removing these forcings. If this is impractical, in 15.0.1 ESD#2 or later, traceflag 15307 can be enabled to nullify the effect of any `set forceplan` statements when query plans are compiled.
  Likewise, any explicit forcings for indexes or for prefetch/parallelism/buffer replacement strategy can be nullified by enabling traceflag 15308.
  Traceflags 15307 and 15308 can be set at boot time. They can also be enabled with `dbcc traceon`, but note that their effect is server-wide, so all sessions are affected.
  Note that both traceflags do not affect any query plan properties defined by abstract query plans.

- If your system seems to be consuming an unreasonable amount of space in `tempdb`, you can use the MDA tables to see if one particular session consumes a lot of space in a worktable.
  Assuming the MDA tables are enabled, run the following query:

```
select SPID, DBName, ObjectName, PartitionSize
from master..monProcessObject
where DBID = tempdb_id(SPID)
order by SPID
```

  Look for sessions that have a large value for `PartitionSize`. Worktables have an `ObjectName` of `'temp worktable'`. By querying `master..monProcessSQLText` and/or `master..monProcessStatement` you can find the corresponding SQL statement for the corresponding sessions. To stop sessions from filling up `tempdb`, and thus affecting other sessions also requiring `tempdb` space, you can create a resource limit of type `'tempdb_space'`. Another possible

remedy is creating multiple temporary databases and assigning these to specific users. See the ASE documentation for further details on these features.

- If you encounter a slow query and you suspect that reformatting is causing the sub-optimal performance, first follow the recommendations with respect to statistics (see page 21). If the reformatting persists and performance remains unsatisfactory, you can verify whether disabling reformatting improves things as follows:

```
set store_index off   -- disables reformatting
go
select …              -- your query
go
```

Or:

```
select …              -- your query
plan '(use store_index off)'
```

Before using individual optimization criteria settings in production code, Sybase recommends verifying this solution with Technical Support.

- Under `allrows_dss`, if you encounter an unreasonably slow query, or if the query runs into a 701 error during optimization, first follow the recommendations with respect to statistics (see page 21). If the issues persist, try if the following solves the problem:

```
set bushy_space_search off   -- relevant only under allrows_dss!
go
select …              -- your query
go
```

Or:

```
select …              -- your query
plan '(use bushy_space_search off)'
```

Before using individual optimization criteria settings in production code, Sybase recommends verifying this solution with Technical Support.

- When running large numbers of identical or very similar client-generated SQL queries (i.e. not stored procedures, and also not in execute-immediate) where the difference lies in the search argument (i.e. a different primary key value), and you're running 15.0.1 or later, consider enabling the statement cache as well as literal auto-parameterization. These may significantly reduce the time and resources spent on query optimization and therefore improve overall performance.
  Literal auto-parameterization is an enhancement to the statement cache in ASE 15.0.1 or later. When the statement cache is enabled, a query's plan will be cached so that an exactly identical query does not need to be compiled again, but the cached plan can be used instead, saving time and resources. With literal auto-parameterization enabled, this caching is extended to almost-identical queries that differ only in a constant value. For example, take these two queries:

```
select CustName from Customers where CustID = 123
select CustName from Customers where CustID = 456
```

These queries are not identical, but they are likely to end up with the same query plan. Enabling literal auto-parameterization has the effect that the statement cache factors out the constant value in the **where**-clause and caches a plan for all queries looking like this:

```
select CustName from Customers where CustID= <integer-constant>
```

This means both queries above would use the same cached query plan. Literal auto-parameterization can greatly improve the effectiveness of the statement cache, and therefore it is recommended to experiment with this feature if your applications use many almost-identical SQL queries that are not part of stored procedures and are not run in execute-immediate.

The statement cache is enabled sever-wide with the configuration parameter `'statement cache size'`. On session level, the statement cache can be disabled with `set statement_cache off`.

Literal auto-parameterization is enabled server-wide with the configuration parameter `'enable literal autoparam'`, and on session level with `set literal_autoparam on` (or `off`). It applies only when the statement cache is enabled too.

Using 'Compatibility Mode' in ASE 15.0.3 ESD#1

For customers who prefer to limit the required testing effort for upgrading to ASE 15, Sybase has introduced a new feature named 'Compatibility Mode' in ASE 15.0.3 ESD#1.
Compatibility mode is a query processing enhancement allowing qualifying T-SQL queries to be processed with a method of query optimization and query execution very similar to that used in ASE 12.5, instead of using the ASE 15 query processing algorithms.
Compatibility mode is not an optimization goal, but a query processing option on a higher level which determines whether 12.5-like optimizer and query execution logic will be used (which exclude all ASE 15 features), or whether the ASE 15 features are used; in the latter case, the considerations in the rest of this document apply.

Compatibility mode has been made available to customers to provide more control over the precise moment when ASE 15-specific query processing features become active in their queries and applications, thus allowing gradual introduction of ASE 15 query processing aspects after migrating to ASE 15. Consequently, the required testing effort can be better focused and planned than when migrating to ASE 15 without using compatibility mode.

It should however be noted that, as a consequence of using compatibility mode, applications may not be able to fully benefit from the query processing enhancements of ASE 15. This is a conscious choice that customers will need to make when using compatibility mode.
It is expected that customers, after having migrated to ASE 15, will eventually disable compatibility mode so as to take full advantage of the performance improvements in ASE 15.

For full details on compatibility mode, please refer to the whitepaper " *Using 'Compatibility Mode' In ASE 15.0.3 ESD#1 For ASE 15 Migration*" (see URL on page 41).

If all else fails…

In case none of the suggestions in the current document help you to resolve query performance issues in ASE 15, Sybase recommends that you upgrade your ASE installation to the latest ESD available for your platform. As always, Sybase is making continuous product improvements and your issue may already have been fixed in a later ESD than you are currently running. If your problem still persists, you should open a case with Sybase Technical

Support. Please refer to page 35 for guidance about the information to collect before contacting Technical Support.

Obsolete optimization commands in ASE 15

Various optimization-related settings from 12.x are no longer when using the ASE 15 query processing features. Although the following commands still exist in ASE 15, they do not perform any function anymore, and simply return a success status:

- `set sort_merge`: in ASE 15, replaced by `set merge_join`, optimization goals and the configuration parameter `'enable merge join'` (in 15.0.2)
- `set jtc`: Join Transitive Closure is always enabled in ASE 15
- `set table count`: no longer applies in ASE 15

In addition, the following features no longer apply in ASE 15:

- configuration parameter `'enable sort-merge join and JTC'`: replaced by optimization goals and by the configuration parameter `'enable merge join'` (in 15.0.2).
- boot-time trace flags 334 and 384, which enabled merge joins and JTC individually, no longer have any function in ASE 15.

However, note that when using compatibility mode (see page 12), then these commands and features will still function as in pre-15.0 for those statements that are processed in full compatibility mode or restricted compatibility mode. For statements that are processed using ASE 15 mode, these commands are no-ops as described above.

Resource Recommendations for ASE 15

Procedure cache

One of the consequences of the redesigned query processing engine in ASE 15 is that ASE requires more procedure cache. This increased memory requirement applies to optimization and execution of queries.

One reason for this increased requirement is that ASE has many more join methods and data access algorithms to consider than prior to version 15. This is especially true for `allrows_dss`, which will generally tend towards consuming more procedure cache for query optimization than `allrows_mix`, which in turn usually needs more procedure cache than `allrows_oltp`.

It is difficult to predict exactly how much more memory ASE 15 will need compared with 12.5. As a rule of thumb, Sybase recommends to plan for sizing of the procedure cache in ASE 15 between 2 to 6 times larger than in ASE 12.5. When `allrows_dss` is used, it is likely that more procedure cache may be needed than when `allrows_oltp` is used. Use the 'procedure cache' section of `sp_sysmon` to monitor procedure cache usage.

Procedure cache usage limitation in 15.0.2 ESD#2

In ASE 15.0.2 ESD#2, a limitation was implemented on the amount of procedure cache that can be used by the ASE query optimizer. This is based on the setting of configuration parameter `'max resource granularity'` (settable on session level with `set resource_granularity`). This setting is an integer between 1-100, reflecting a percentage. The default setting is 10.

This feature works in two ways. First, when the optimizer has consumed more procedure cache than:

```
<max_resource_granularity> ÷ 100 × <configured procedure cache size>
```

… and when at least one full plan has already been generated, the optimizer will time out and use the current best plan as the final plan.
Second, when the optimizer has not been able to generate a full plan yet, but has already consumed an amount of procedure cache of:

```
max(50, <max_resource_granularity>) ÷
                                100 × <configured procedure cache size>
```

… then the optimizer aborts the query plan generation to avoid consuming all of the procedure cache. Since the query has still no plan, its execution will then fail.

This new behaviour was introduced in 15.0.2 ESD#2. It can be disabled with traceflag 15380.

Other resource usage aspects of ASE 15

One of the new aspects of ASE 15 is the feature of table partitions. Although semantic data partitioning is an optional license feature, it matters even when you do not use semantic data partitioning: in ASE 15, each table and index always consists of at least one partition. Consequently, the configuration parameter `'number of open partitions'` must be set high enough to avoid unnecessary performance penalties.

Sybase recommends using `sp_countmetadata` to estimate the required setting for this parameter, and use `sp_monitorconfig` to monitor its usage (see page 26).

Statistics In ASE 15

Why statistics matter - especially in ASE 15

ASE uses a cost-based query optimizer to choose the best plan for a particular query. To achieve that goal, the optimizer relies on statistics about the tables, indexes, partitions, and columns referenced in a query for estimating the cost -in terms of I/O and CPU time- of different possible query plans. The optimizer then chooses the query plan method that has the lowest cost. However, this cost estimate cannot be accurate if the statistics themselves are not accurate. As a result, inaccurate statistics could lead to a suboptimal choice of plans and result in slower performance than necessary.

Some statistics, such as the number of pages or rows in a table, are updated automatically during query processing (these are stored in `systabstats`). Other statistics, notably the histograms on columns, as well as density information, (both stored in `sysstatistics`), are updated only when `update statistics` runs, or when indexes are created. In practice, it is critical that the histograms be up to date. It is a DBA responsibility to schedule and run `update statistics`.

In ASE 15, having up-to-date statistics has become even more important than it already was.

ASE 15 is more susceptible to statistics issues than previous ASE releases due to now having multiple algorithms for sorting, grouping, unions, joining and other operations, thus offering the optimizer many more possible choices. In addition, ASE 15 uses statistics in more ways than in ASE 12.x, for example for determining the join order in multi-table queries. Good statistics are therefore important to help avoid generating a suboptimal query plan.

Recommendation: run update index statistics

It is recommended to maintain up-to-date histograms for all columns referenced in WHERE-clauses, either as join predicates or as search arguments. Since these columns are typically part of an index, in practice this means that running `update index statistics` for all user tables is likely to do the job. If such columns are not part of an index, you should at least generate histograms for these non-indexed columns, or consider indexing them.

Compared with `update statistics`, running `update index statistics` also generates histograms for the non-leading index columns, which provides the optimizer with more information to generate optimal query plans. However, `update index statistics` has some potential disadvantages compared with `update statistics`. Since it needs to perform a sort operation for every non-leading index column (the leading index column is already sorted in the index tree by definition), `update index statistics` will take more time and resources to run than `update statistics`.

Using 'sampling' with update index statistics

One possible consequence is that `update index statistics` may run into a 701 error when the table is too large to perform the sort for a column with the available amount of procedure cache.

When this happens, the remedy -apart from possibly increasing the size of the procedure cache- is to run `update index statistics` with the "sampling" option. Instead of reading the column values for the entire table, only a percentage of the table will be read. This makes sense since the final histogram for a large table is based on just a fraction of the column values read. A positive side effect is also that `update index statistics` with sampling runs faster than without sampling, so it may also be used for large tables when `update index statistics` takes too long to run. Note that when using sampling, there is always a

possibility that the histogram contains less accurate information than when the full table was read. However, this potential downside is difficult to quantify.

In the following example, only about 5% of the table is read for creating the histograms for non-leading index columns:

```
update index statistics big_table with sampling=5
```

The lowest possible value for sampling is 1%. Setting the percentage to 0 is equivalent to 100%.

In the `optdiag` output, the sampling percentage used is displayed as "Sampling Percent:" (0 means: no sampling used).
Instead of specifying the sampling percentage explicitly, a default sampling percentage can also be configured server-wide, by means of the configuration parameter `"sampling percent"`. However, this setting will apply to all user tables and that may not always be desirable.

Speeding up update statistics with parallelism

It may be possible to speed up `update statistics` for non-leading index columns or non-indexed columns by using parallelism. For such columns, histogram creation requires that all column values be sorted first. By performing this sort operation with parallel worker threads, the run time of `update statistics` can often be improved (assuming your system has enough resources to use parallelism).

To use parallel sorting, specify the `with consumers` clause:

```
update index statistics big_table
with consumers=20              -- use 20 worker processes for the sort
```

Exactly how much improvement can be achieved with parallelism, and what level of parallelism would be optimal, highly depends on the specific system hardware, ASE configuration and workload. You should experiment to determine what works best for your situation.

Note that parallelism for `update statistics` does not apply to the table scan or index scan with which the column values are retrieved.

When to run update statistics?

Statistics should be an accurate reflection of the data in a table. This implies that statistics may need to be updated when data is changed more frequently. One example is a `datetime` column, holding the date that a row was inserted. Another is an identity column, which will continuously insert higher values. For these types of columns, histograms will quickly become out-of-date, since all new rows being inserted will have values higher than the previous maximum value in the histogram. This may contribute to the optimizer generating sub-optimal query plans.
ASE 15 offers a new feature to assist in deciding whether `update [index] statistics` may be necessary for a table. The new built-in function `datachange(`*`table_name, partition_name, column_name`*`)` function measures the amount of change in the table (or partition) since the last time `update [index] statistics` was run. Depending on this percentage, you may decide to `run update [index] statistics` more, or less, frequently.
Specifically, `datachange()` measures the number of `inserts`, `updates`, and `deletes` that have occurred on the given table, partition, or column. This information helps you determine if running `update [index] statistics` may be needed.

An example of the `datachange()` function is:

```
1> select datachange('my_table', null, null)
2> go
 --------------------------
                243.340418
```

In this output, the percentage of changes to the table `my_table` is 243%, meaning that the number of changes is higher then the number of rows in the table. This indicates that it is a good candidate to run `update [index] statistics` now.
The `datachange()` built-in function can also be run on individual columns:

```
1> select datachange('my_table', null, 'id_number')
2> go

 --------------------------
                  0.017761
```

The result of 0.017% indicates that the `id_number` column has had a relatively low amount of change, and therefore the statistics for that column are still likely to be accurate. It is good practice to run `datachange` on a regular basis against commonly used tables and columns. This gives a basis for determining when it is useful to run `update [index] statistics`. Lastly, by specifying the second parameter of `datachange()`, which is a partition name, only the changes for that partition are reported. You may need to use `sp_help` to determine a table's partition names.

ASE has the functionality to automatically check objects using `datachange()`, and schedule `update [index] statistics` accordingly using ASE's built-in job scheduler feature. For more details on setting up and using this functionality, please refer to the 15.x manual titled "*Performance and Tuning Series: Query Processing and Abstract Plans*". In chapter 6, "*Using Statistics to Improve Performance",* refer to the section titled "Automatically updating statistics".

NB. For partitioned tables, it is possible to run `update [index] statistics` on an individual partition. While that may allow you divide a long-running `update statistics` run for a large, partitioned table into multiple shorter runs, keep in mind that updating statistics on a partition only updates histograms on local indexes for that partition: histograms for global indexes on the table will not be updated.

How many histogram steps?

Another issue is how many steps a histogram should have in order to provide the optimizer with an accurate reflection of the data distribution in a column. The default of 20 histogram steps may be adequate for columns with a small number of distinct values ("low cardinality"), or for tables with a small number of rows. However, for tables with a large number of rows, having columns with many distinct values 20 steps may be insufficient. This is especially true when a column has a high degree of data skew (that is, the values for the column are not evenly distributed over the rows).

Obviously, if your system runs to full satisfaction with 20 histogram steps, do not change anything. However, if you suspect sub-optimal query plans are being generated, you may want to consider increasing the granularity of histograms by increasing their number of steps and determine whether that has a positive effect. As always, there can be too much of a good thing. Increasing the number of histogram steps can lead to higher resource consumption, especially procedure cache usage, and longer optimization times before a query plan is generated.

Due to the optimization timeout limit feature in ASE 15, this can in turn also result in sub-optimal plans being generated.

Exactly how many steps a histogram should have is hard to predict. In practice, you need to see if a higher number of histogram steps results in better query plans and better overall performance.

Nevertheless, as a practical guideline it may be useful to start by increase the step count to 200. If that does not have the desired positive effect, try a higher number, like 500. An alternative guideline is to use one histogram step for every 10,000 data pages. However, more than 1000-2000 histogram steps is generally considered too high a number: if no positive effect has been achieved, the histogram steps may not contain the solution to your problem.

The number of histogram steps is determined when `update [index] statistics` is executed. This is done as follows:

- The configuration parameter `'number of histogram steps'` defines the default for all `create index` and `update [index] statistics` commands for new histograms. The out-of-the-box default is 20.

- For `update [index] statistics`, the number of steps can also be set explicitly with the clause `using` *nnn* `values`:

```
update index statistics big_table using 300 values
```

- For existing histograms, `update [index] statistics` uses the current number of steps in the existing histogram when generating a new histogram. The configuration parameter `'number of histogram steps'` does not apply to existing histograms. Only when `update [index] statistics` explicitly specifies `using` *nnn* `values`, will the step count in the existing histogram be overridden, and *nnn* be used as the number of steps for the new histogram.

- The number of histogram steps established at this point is the target number of steps for the histogram. In the **optdiag** output, this is displayed as "Requested step count:".

- Next, the `'histogram tuning factor'` comes into play. This configuration parameter multiplies the number of steps determined above for the purpose of generating an internal, intermediate histogram first. If 100 histogram steps are chosen, and "histogram tuning factor" is set to 20, then the intermediate histogram may have as many as 2000 steps. This is done to increase the chance of identifying so-called "frequency cells", which reflect duplicate data values. If no frequency cells are found, the histogram is compressed back to the original number of steps; if frequency cells are found, these are kept. The actual number of resulting histogram steps is displayed in the `optdiag` output as "Actual step count:".

- Prior to 15.0.1 ESD#1, the `"histogram tuning factor"` was set to 1 by default. As of 15.0.1 ESD#1, the out-of-the-box default has been increased to 20 due to positive experiences with this feature. The histogram tuning factor can range from 1 to 100; however, it is recommended to set it (or leave it set at) 20 unless advised differently by Sybase Technical Support. To avoid unexpected excessively large step counts, and corresponding procedure cache usage, when customers upgrade from 12.5 to 15, as of 15.0.2 ESD#2, the following formula is used when histogram tuning factor is still set to the default of 20:

```
min( max (400, requested_steps),
                    histogram_tuning_factor * requested_steps)
```

This formula has the effect of assuming that `'histogram tuning factor'` is set to 1 (as was likely the case for those customers before they upgraded to 15).

Identifying missing statistics

ASE 15 has a new feature to flag cases where no statistics exist for columns where the optimizer would have expected them. When running the command `set option show_missing_stats on`, the optimizer will print a message when it encounters such a case.

By default, the output is written to the ASE console (that's not the same as the ASE errorlog!); enable traceflag 3604 to get the output sent to the client.

Two examples of these messages are:

```
NO STATS on column t.col1
NO STATS on density set for t={col2, col3, col4}
```

When seeing such messages, it is recommended that the DBA run update statistics for the columns involved. The corresponding commands would be:

```
update statistics t (col1)
update statistics t (col2, col3, col4)
```

Depending on which indexes exist on this table, the same effect might be achieved by running `update index statistics t`. Note that creating these statistics for the reported columns does not guarantee that a different query plan will be generated by the optimizer. However, since the optimizer signals it would have liked to consider additional options, it is best to ensure the required statistics are available.

It is recommended to enable `set option show_missing_stats on` at least for some time in the ASE 15 test environment. This option can be set in a login trigger, so application code changes can be avoided. Please refer to the ASE documentation for more information about login triggers.

Old statistics and upgrading to ASE 15

Before or after upgrading to ASE 15, it is recommended to determine whether very old statistics exist. Such old statistics may exist for various reasons:

- Since ASE 11.9, when an index is dropped, histograms for the index columns are no longer dropped automatically at the same time. Therefore, if these histograms are never explicitly updated – and that may never happen when those columns are not part of another index – the old histograms remain.

- If a histogram was ever created for a non-indexed column, it will still be around if it was never updated or deleted.

Irrespective of the reason why they came to exist, old statistics may not provide an accurate reflection of the current data distribution in the column. In rare cases, this may result in needlessly inefficient query plans after upgrading.
To determine if old statistics exist, query the column `sysstatistics.moddate` (this is the date/time when the existing histograms and densities were last updated). If this indicates that very old statistics indeed exist, use the `optdiag` utility to determine which statistics are involved. If, at a later point, you are seeing sub-optimal

performance for queries where tables are involved that you know have very old statistics, consider updating (the preferred method) or deleting (the faster method) those old statistics.

However, before deleting any statistics, please make a backup copy with the `optdiag` utility. This allows you to restore the original statistics (again, with `optdiag`) in the case that the freshly updated statistics should result in unexpected performance effects.

In fact, it may be a good idea to make a backup copy of the statistics for all your tables after upgrading to ASE 15, but before running `update [index] statistics` to refresh the statistics.

To delete statistics, use one of these commands:

```
delete statistics t   -- delete all histograms and densities on table t
delete statistics t(a)   -- delete the histogram on column t.a
delete statistics t(a,b) -- delete the densities for columns (t.a, t.b)
```

Summary of recommendations for statistics

Please refer to the preceding pages for details on each of these recommendations.

- Keep statistics on user tables up-to-date by running `update [index] statistics` regularly
- Use the `datachange()` function to determine whether `update [index] statistics` should be run more often
- Preferably run `update index statistics` instead of `update statistics`. For large tables, use sampling if necessary.
- If you suspect sub-optimal query plans are generated, consider increasing the number of histogram steps for the tables involved.
- Use the command `set option show_missing_stats on` to determine cases where statistics may be missing; and then create those statistics.
- Consider deleting or updating old statistics after upgrading to ASE 15 when these are associated with sub-optimal query plans; however, always make a backup copy first with `optdiag`.

Recommended Testing Before Upgrading To ASE 15

Why should you test?

Before upgrading your production systems to ASE 15, it is important to gather details about the performance characteristics of your applications on the current, pre-15 version of ASE, preferably in a production environment. Gathering such data is important, since any discussions about performance differences after upgrading to ASE 15 will not be very useful if these discussions are not based on hard numbers. For the 12.x-vs-15 performance comparison to be valuable, it is strongly recommended that these tests be performed as realistically as possible. This means:

- Run tests for as many application functions as possible, or at least the most critical functions. For each function, measure the response time or throughput; preferably also perform these measurements for each query executed by the application (see page 32 for suggestions)
- Run the performance measurements in your current ASE 12.x production system
- run the same tests before upgrading your production system to ASE 15, in a fully configured ASE 15 test system with a copy of the full ASE 12.x production database, as well as realistic workload (i.e. the same queries as in 12.x, and with the same level of concurrent user activity)

Also, preferably run these tests before upgrading your production system to ASE 15, so that you can identify any issues before going live on ASE 15. Capturing the "performance footprint" of your current ASE 12.x production environment will provide a good baseline for any comparisons with ASE 15. The measurements should provide specific numbers such as number of logical I/Os, elapsed time, compilation time, CPU utilization etc. for apples-to-apples comparison. Differences in the measured values will help identify individual queries performing poorly as well as any server-level configuration issues.
There are some different angles to looking at performance. To enable a sensible comparison of performance in ASE 12.x and ASE 15, performance data needs to be gathered on two levels:

- for individual queries, run in isolation
- for individual queries, run with full workload by other users
- for ASE as a whole, from a server-wide resource usage perspective

Please ensure that the configurations of the ASE 12.x and ASE 15 server are as identical as possible. Having said that, the ASE 15 server may have more resources allocated to it than 12.x, and in some cases (see page 14) that is in fact required. Also, make sure the tests are executed in similar circumstances.
These are some aspects that are critical to doing a fair comparison between 12.x and 15:

- Ensure the cache is 'warmed up' the same way in 12.x and in 15 during the tests
- Use identical cache/buffer pool configurations
- Make sure the amount of procedure cache in ASE 15 is a multiple of what is used in 12.x (see page 14)
- Use similar data device layout/placement, especially for log devices and for `tempdb`?
- If you run tests that modify data, consider setting up test systems whereby the original database can be restored after each test run.

These considerations are critical as they will affect the performance being measured. Differences in these areas can therefore lead to false alarms and wasted time following up on non-existent issues.

Please note…

It is important to observe that the approach towards testing as advocated above is not specific for ASE 15, but is a best practice that applies to any major upgrade of any software product. The best chance of avoiding potential risks around upgrades is to run as realistic an environment as possible before putting your actual production systems through the upgrade. Should you be unable to follow this approach, the natural implication is that you simply will not discover any issues until your production system does.

From a risk management perspective, you should ask yourself how important it is to identify any performance issues prior to the upgrade, and how much opportunity you will have to identify and fix such issues after your production system has been upgraded. Although, again, this is not specific for ASE, Sybase recommends performing thorough testing as outlined in this document to minimize the risk of disruptions due to an upgrade.

Using Compatibility Mode

As described on page 12, ASE 15.0.3 ESD#1 provides a feature named 'Compatibility Mode' to offer customers a migration path to ASE 15 that requires less testing effort at the moment of migration.
Nevertheless, Sybase still recommends to perform sufficient testing on applications that are migrated to ASE 15, even when using compatibility mode.

Identifying queries that run slower

Assuming you are able to run realistic, representative tests on both ASE 12.x and ASE 15, your next steps depend on what you find in ASE 15. Along the lines of the suggestions on page 7, it is recommended to run your tests in ASE 15 at least three times, each time with a different server-wide optimization goal.

In the ideal situation, you can establish a server-wide optimization goal under which everything runs the same or faster in ASE 15, in which case there is probably not much else left to do. In the event that some applications would appear to run slower in ASE 15 than in ASE 12.x, you will first need to identify the queries involved, then determine why they run slower, and finally figure out how to resolve this. The main challenge lies in identifying those queries that run slower.

If may be immediately apparent which queries run slower in ASE 15, for example when a clearly defined application function appears to perform less than in 12.x. In such a case, the corresponding SQL queries can hopefully be identified relatively easily by using the MDA tables or by application tracing (see page 32). If the slower queries cannot be identified easily, the best approach is to capture all SQL queries submitted by the application in ASE 15 (again, see page 32), and analyze these further (see below).

In general, it is particularly important to ensure the following types of queries are included in your tests, even if they are not executed frequently (for example, for periodic reporting). Application developers may be able to provide some of these queries. This concerns queries involving:

- multi-table joins,
- multiple views/unions,
- star-schema joins
- several subqueries
- derived tables
- outer joins

Analyzing performance differences between ASE 12.x and ASE 15

Once you have captured the SQL text of actual production queries, you will need to analyze these to determine which queries perform worse in ASE 15 than in 12.x. This process can be performed with simple command-line

tools, for example as described below. Note that this approach is well suited for identifying query plan problem with longer execution times (i.e. multiple seconds), but less suitable for queries with very short execution times (e.g. 100 milliseconds).

First, create a file, named, say, `sql.in`, with the following layout, containing the SQL text of your captured queries (in batches, since this is how applications submit queries to the ASE server):

```
set plan optgoal allrows_xxx  -- only for 15; this raises an error in
12.x
go
set statistics plancost on    -- only for 15; this raises an error in
12.x
go
use your_db                   -- your default database
go
set statistics io on
set statistics time on
set showplan on
go
<batch 1>                     -- captured SQL query batches go here
go
<batch 2>
go
[...]
<batch n>
go
```

Now run this file both in ASE 12.x and in 15 and capture the output:

```
isql -U username -P passwd -S YOUR_12X_SERVER -e -i sql.in >
sql.12x.out
isql -U username -P passwd -S YOUR_15_SERVER  -e -i sql.in > sql.15.out
```

Now you need to analyze the differences in performance between execution in 12.x and in 15.
This can be done by searching for the string "Server elapsed time:" in both files (for example, with **grep**), and comparing the results (for example, by importing the list of execution times for both cases into a spreadsheet and looking for those cases that are more than, say, 50% slower in ASE 15).
Here is an example of such a line, indicating that the query ran in little less than 16 minutes in ASE 15:

```
Adaptive Server cpu time: 27300 ms. Adaptive Server elapsed time: 945953 ms.
```

Another criterion to identify queries is to compare the amount of I/O the queries spent in 12.x and 15.0. This can be done by examining output lines like the following (note that each query may produce a number of such lines, one for each table referenced by the query). The number to look for is the number of regular logical reads (shown in bold below):

```
Table: Accounts scan count 365, logical reads: (regular=833370 apf=16
total=833386), physical reads: (regular=1927 apf=101769 total=103696),
apf IOs used=99762
```

Once you have identified specific queries this way, you can study those particular cases in more detail by looking at their query plan, and the Lava Tree output resulting from the `plancost` option (see page 30).

Analyzing short-duration queries

To compare the performance difference of very short queries, you may want to run such a query a predefined number of times, e.g.

```
declare @i int, @d datetime
select @i = 1000    -- run the query 1000 times
declare select @d = getdate()
while @i > 0
begin
   select @i = @i -1
   ...your query...
end
select DurationInMilliSec=datediff(ms, @d, getdate())
go
```

With this approach, you should compare the number of milliseconds for 1000 executions in 12.x and 15. Repeating a query 1000 times will eliminate jitter due to external circumstances, which could affect the response time of one such individual, short execution quite drastically.

Server-level performance aspects

Server-level configuration settings can have a big impact on query performance, though it may not be possible to predict which queries will be affected most.

As part of your testing around the ASE 15 migration, capture the following server-level data:

- Run `sp_sysmon` regularly and capture its output; recommend is running it every 10-15 minutes with a sample interval time of 1 minute.
  It is critical that data is captured during all the significant periods of the application processing. For example: daily peak periods, nightly / weekly / monthly batch cycles, Monday morning Market Open's etc.
  When analyzing the `sp_sysmon` output, also pay attention to the following:

  - Data Cache usage: Review the cache hits, spinlock contention and the buffer pool usage.
    High counts for cache hits and/or high spinlock contention may be an indication of poorly performing queries.

  - CPU utilization: Review the CPU characteristics of the load i.e. how busy are the various engines, how much time is spent in I/O Busy, Idle v/s CPU Busy etc.

  - Device IO characteristics: Review the I/Os outstanding , I/Os completed/sec to get a rough idea of the level of I/Os generated by the application/module. It would also be interesting to note how the I/Os are distributed across the various database devices. If the layout in the target ASE 15.0 is expected to change, this information may come in handy.

- Run `sp_monitorconfig 'all'` regularly (every 10-15 minutes), and look for configuration parameters showing a non-zero 'reuse', in particular procedure cache. You may also run `sp_monitorconfig` with a table name as an additional parameter: this will cause the result values from `sp_monitorconfig` to be stored in a table (which you have to create first; see the ASE Reference manual for details).

Example Of ASE 15 Query Plan And Lava Tree

ASE 15 Query Plans

Each SQL statement is compiled into a query plan. In ASE 15, a query plan is built as an upside-down tree of operators: the operator at the top has one or more child operators. Each of these, in turn, can have one or more child operators themselves, and so on, thus building a tree of operators. The exact shape of the tree and the operators in it are chosen by the ASE query optimizer.

The query plan can be represented in different ways:

- `set showplan on` shows the query plan in the well-known format, showing the join order, indexes used and I/O strategy.
- `set statistics plancost on` (a new command in ASE 15) shows the query plan in a format more closely resembling the internal "Lava Tree".
- Lastly, the GUI tool DBISQL can display the query plan in a graphical tree format. This is not discussed further in this document.

Example: set showplan

To display the query plan in the classic ASE format, use `set showplan on`:

```
set showplan on
go

select A.au_fname, A.au_lname
from authors A, titleauthor TA
where A.au_id = TA.au_id
go
```

The following shows some highlights from the resulting query plan output:

```
 The type of query is SELECT.

 3 operator(s) under root

 |ROOT:EMIT Operator (VA = 3)
 |
 |   |NESTED LOOP JOIN Operator (VA = 2) (Join Type: Inner Join)
 |   |
 |   |   |SCAN Operator (VA = 0)
 |   |   |  FROM TABLE
 |   |   |  authors
 |   |   |  A
 |   |   |  Table Scan.
[...]
 |   |   |SCAN Operator (VA = 1)
```

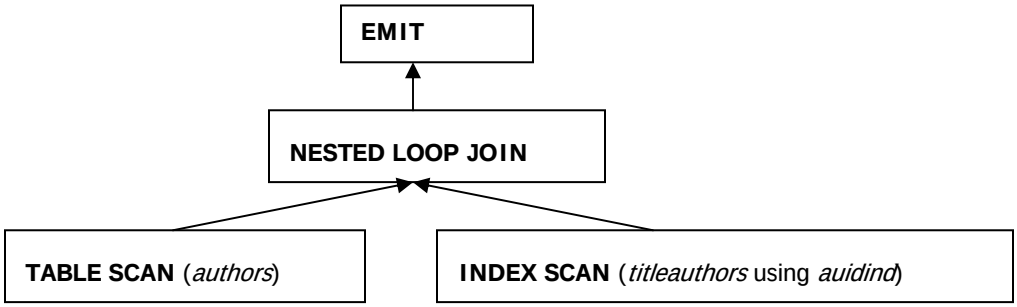
```
     |    |    |  FROM TABLE
     |    |    |  titleauthor
     |    |    |  TA
     |    |    |  Index : auidind
     |    |    |  Forward Scan.
     |    |    |  Positioning by key.
     |    |    |  Index contains all needed columns. Base table will not be
    read.
     |    |    |  Keys are:
     |    |    |    au_id ASC
    [...]
```

The query plan output as displayed by `set showplan on` consists of two main components:

- The names of the query plan operators, along with some additional information, to show which operations are being executed in the query plan. Note that this output is very similar to that in ASE 12.x, though ASE 15 has various new operators.
- Vertical bars (the "|" symbol) with indentation are new in ASE 15. These are added to visually clarify the structure of the query plan operator tree.

The query plan from the example above indicates that the ASE query optimizer has chosen a plan consisting of 3 operators under the root operator EMIT. In ASE 15, the EMIT operator is always the top operator (called the 'root' operator) that is responsible for routing the result of the query to the client.

The plan contains a NESTED LOOP JOIN operator, with the *authors* tables as the outer table (since it comes first in the join order) and the *titleauthor* table as an inner table (second in the join order). The first indented operator appearing below the NESTED LOOP JOIN operator always represents the outer side of the join and the second operator at the same level of indentation is the inner side of the join.
The scan strategy for the *authors* table uses a table scan, while the inner table *titleauthor* will be scanned by index *auidind.* Both these operators are shown in the `showplan` output as a SCAN operator. Graphically, this query plan can be represented by the following operator tree:

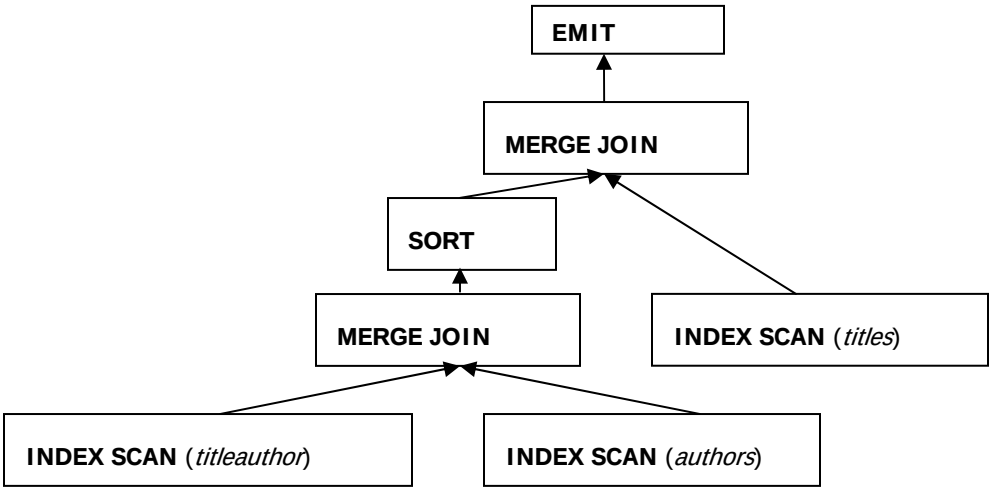


A query plan can have an arbitrarily deep tree of query plan operators. Each operator receives requests for the next row from its parent (top-down control flow), processes the request and returns the requested row to its parent (bottom-up data flow). The order of execution of an operator depends thus on its parent's algorithm and, recursively, on the order of execution of its parent.

For instance, a "left-deep" tree of nested-loop join operators will start by scanning the outermost table, when the "next row" request, which is passed down on the outer side of each nested-loop join, reaches the outermost scan. Then, as a nested-loop join receives a row from its outer child, it will do a corresponding scan of its inner side based on that row. The nested-loop join will return, each time its parent asks for, its own row based on its current

outer and inner child rows.
The following example shows a more complex query plan:

```
select A.au_fname, A.au_lname, T.title
from authors A, titleauthor TA, titles T
where A.au_id = TA.au_id and T.title_id = TA.title_id
```



In this case, EMIT asks its MERGE JOIN child for a row, which in turn asks its 2 children for rows attempting to satisfy the merge clause.
The upper MERGE JOIN's outer child, SORT, is a blocking operator. This means it needs to consume its entire input before returning its first row (since the first row in the sorted result could be last row returned by its child). SORT will thus ask its child, the lower MERGE JOIN, one by one for all of its rows and will sort them (if possible in memory, overflowing if needed to a worktable) and only then will it return its first row to the uppermost MERGE JOIN.

To produce these rows at each request of its SORT parent, the lower MERGE JOIN will ask for new rows from its outer and/or inner children, depending on the current values in the merge columns, until the equijoin clause is satisfied and a MERGE JOIN row is returned.

As these children are INDEX SCANs using indexes on the equijoin columns, the rows are returned ordered, as required by the MERGE JOIN algorithm, and no sort is deeded.
The upper MERGE JOIN will have the same behavior, asking SORT and INDEXSCAN(titles) for rows and return at each EMIT request a row where the equijoin clause is satisfied.
Finally, the root operator EMIT sends each final result set row back to the client application that invoked the query.

Note that it is not guaranteed that each operator in the tree is executed. For example, a filter condition may evaluate in such a way that no rows can ever result from a subtree, whose operators may therefore not be executed at all. For more information about the operators and tree structure, see the ASE Performance and Tuning Guide, volume "Query Processing and Abstract Plans".

Example: plancost and Lava Tree

A useful new feature in ASE 15 is the `plancost` option, which displays the query plan in a semi-graphical tree format more closely resembling the internal representation, known as the so-called "Lava Tree" (which got its name since data rows flow upward through the tree operators to a single point, similar to lava in a volcano).

The `plancost` option can be turned on using the following command:

```
set statistics plancost on
go

--- Run your query here
```

One application of this command is to compare the estimated and actual costs in a query plan. It can be a useful tool for diagnosing query performance problems. For each operator, the Lava Tree option shows the estimated (el:) and actual (l:) numbers of logical I/O, estimated (`ep:`) and actual (`p:`) physical I/O, estimated (`er:`) and actual (r:) row counts and actual CPU ticks (`cpu:`). In addition, execution operators that do a sort or a hash-based operation will also report the number of private buffers used for that operation (`bufct:`, not shown in the example below). Since not all of these quantities apply to each type of operator, a subset may be shown. For sub-optimal query plans, this information can be used to verify that the optimizer's estimates are sound.

If the join query from the previous example is run, you get output that looks like the operator tree shown in the previous diagram with additional annotations showing the actual versus estimated numbers. The estimated numbers are those computed by the query optimizer and the actual numbers are the result of the actual query execution. This is a good way to validate how close the optimizer's cost model is with respect to the actual numbers.

```
==================== Lava Operator Tree ====================

                                        Emit
                                        (VA = 6)
                                        r:25 er:342
                                        cpu: 0

                                      /
                                    MergeJoin
                                    Inner Join
                                    (VA = 5)
                                    r:25 er:342

                            /                       \
                          Sort                      IndexScan
                          (VA = 3)                  titles_672002394 (T)
                          r:25 er:25                (VA = 4)
                          l:6 el:6                  r:18 er:18
                          p:0 ep:0                  l:2 el:3
                          cpu: 0 bufct: 24          p:0 ep:3

                    /
                  MergeJoin
                  Inner Join
                  (VA = 2)
                  r:25 er:25
          /                   \
```

```
IndexScan                IndexScan
auidind (TA)             authors_57600205 (A)
(VA = 0)                 (VA = 1)
r:25 er:25               r:23 er:23
l:1 el:2                 l:1 el:2
p:0 ep:2                 p:0 ep:2
```

From the above output, some interesting observations can be made. The bottom-left `IndexScan` operator (on the *titleauthors* table) has an estimated row count (er:) of 25 and the actual row count (r:) is 25 as well; this means that the optimizer's estimate was correct.

However, the row count estimates for the top `MergeJoin` (VA = 5) are completely off, as the optimizer's estimate is 342 but the actual row count is 25. It may be possible to rectify some of the inaccuracies in the optimizer's estimates by making sure that the statistics are up to date, or by increasing the number of steps in the histogram. If no histograms exist on the join columns, as indicated by `set option show_missing_stats on`, creating those histograms may improve the optimizer's estimates.

Note that comparing the optimizer estimates and actuals is not an exact science: when the estimated rowcount is 25 and the actual rowcount is 30, that should not necessarily be taken as an indication that the optimizer's estimates are incorrect. Order-of-magnitude differences, such as 25 vs. 342 in the example above, are what you should look for.

Also, note that the table name is not displayed for `IndexScan` operator nodes, but only the name of the index. If it is unclear which table is associated with an operator node, there are various ways to find this out:

- The name of the index may uniquely identify the table
- If a correlation name was used in the query, this is shown in the operator node. For the bottom-left `IndexScan` operator, "(TA)" refers to "`titleauthor TA`" from the original SQL query.

The indication "(VA=*n*)", where *n* = 0, 1, 2,…, is shown both in the Lava Tree and in the `showplan` output, and uniquely identifies each operator node.

Capturing Application SQL

In the context of monitoring or troubleshooting performance issues, it is often useful to determine the exact SQL queries an application is submitting to the ASE server, along with some performance metrics such as elapsed time or I/O consumption. This information can be useful both before and after migrating to ASE 15, for example in the context of performance testing around the ASE 15 upgrade (see page 22).
From the side of the ASE server, this can be done in a number of ways, depending on your ASE version:

- Auditing
- MDA tables (12.5.0.3 or later)
- Application tracing (15.0.2 or later)
- Abstract plan capture (12.0 or later) or query metrics capture (15.0 or later)
- Monitor Server
- SQL interception tools, like Ribo (see `$SYBASE/jutils-x_y`) or 3rd-party products

We will look at the first four in a little more detail below.

Auditing

With auditing, you can enable the `'cmdtext'` auditing option (with `sp_audit`). This will capture the client-submitted SQL text in the `sybsecurity` database. You will need to extract the captured text from this database. No performance metrics related to the SQL are captured.
See the ASE System Administration Guide for details on setup and configuration of auditing, and for information about accessing the captured data.

MDA tables (12.5.0.3)

In ASE 12.5.0.3 or later, the MDA tables (also known as 'monitoring tables') can be used to capture, among other things, the SQL text submitted by client applications. For this, use the MDA table `master..monSysSQLText`. Associated execution metrics(execution time, I/O counts, and more) can be found in `master..monSysStatement`.

When looking for slow-running or resource-intensive queries, you can filter the contents of `monSysStatement` by, for example, looking for:

- queries running longer than, say, 10 seconds (i.e. 10000 milliseconds):
  filter on `datediff(ms, StartTime, EndTime) > 10000`

- queries using more than a certain number of logical I/Os, say, 20000:
  filter on `LogicalReads > 20000`

Note that `monSysSQLText` and `monSysStatement` have a configurable but limited size, so they should be queried regularly to extract the captured information and store this in a permanent, regular table. If this is not done frequently enough, or if the number of submitted queries is very high, some of the submitted SQL may not be captured through these MDA tables.
For help with installing and using the MDA tables please refer to either of the following:

- the ASE Performance and Tuning Guide, volume "Monitoring and Analyzing", chapter "Monitoring Tables"

- members of ISUG can use this information available at the ISUG website**:**

```
http://my.isug.com/p/do/sd/sid=13&fid=9&req=direct
```

Application Tracing (15.0.2)

In ASE 15.0.2, you can intercept the SQL queries sent to the ASE server by the client application by means of a new feature named application tracing. This can be used to identify queries with long running times or high I/O consumption.

Suppose an application has connected to ASE as session #24, and you want to see the queries submitted through this connection. This can be done as follows (assuming ASE runs on Unix/Linux):

```
set tracefile '/tmp/my_dir/apptrace.24.out' for 24
go
set show_sqltext on
set statistics io, time, plancost on
set showplan on
go
```

As of this moment, all SQL submitted by session 24 will be written to the file `'/tmp/my_dir/apptrace.24.out'`, accompanied by the output from `set showplan` and `set statistics`. Note that the `set` commands above apply to session 24, and not to the session executing these statements, as would be the normal semantics.

While your session has initiated application tracing on another session, it is important to observe the following:

- Do not run any other commands in this session until you have disabled the tracing again (see below).
- Once application tracing is enabled, do not run other `set` commands than those shown above.
- Do not disconnect your session until you have disabled the tracing again; this may cause `showplan` output to be sent to session 24, even though that session did not run `set showplan on` itself.
- Do not leave tracing enabled for long periods, to avoid disconnects like mentioned in the previous bullet.

To disable application tracing, first switch all `set` commands off again and then stop the tracing:

```
set show_sqltext off
set statistics io, time, plancost off
set showplan off
go
set tracefile off
go
```

You can now access the file `'/tmp/my_dir/apptrace.24.out'` and analyze its contents.

In practice, it may be most practical to use application tracing for short intervals rather than prolonged periods of time, since the amount of data written to the tracefile can be significant. For more information about application tracing, please see a separate whitepaper available on the Sybase website at http://www.sybase.com/detail?id=1052835.

Abstract Plan Capture/Query Metrics Capture

Since ASE 12.0, abstract plan capture can be used to intercept client-submitted SQL text. This text is stored in the `sysqueryplans` table along with its abstract query plan. The captured query text can be retrieved directly from this table. In ASE 15, the new feature of query metrics capture uses the same underlying mechanism to capture execution metrics such as I/O counts and execution times, along with the SQL text. These metrics are also stored in `sysqueryplans`, but should be accessed through a view named `sysquerymetrics`.

A potential disadvantage of these features is that the captured data is stored in the `sysqueryplans` table in the session's current database at the moment the query was executed. This means that the captured query text could end up in, for example, `tempdb`, if that was the session's current database, even though all queried tables may reside in other databases.

For more information about query metrics capture, see the ASE Performance and Tuning Guide, volume "Query Processing and Abstract Plans".

Information To Capture Before Contacting Sybase TechSupport

Before contacting Tech Support, it is often useful to gather additional diagnostics. This is especially true when the problem is reproducible. This section will discuss some potential problem areas and document the collection procedures.

701 Errors

When a regular query (i.e. not `update index statistics`) runs into a 701 error, this indicates that ASE exhausted the procedure cache space. If you are running with the default procedure cache size, you should increase this and try again. The general guideline is to initially take 2-3 times the size of your 12.5.x procedure cache, though in some cases, especially when using the optimization goal `allrows_dss`, your procedure cache may need to be larger.

If increasing the procedure cache has not resolved the 701 error and you cannot isolate the problem, then you should setup a Configurable Shared Memory Dump (CSMD). The following instructions are used.

```
sp_configure 'dump on conditions', 1
go
sp_shmdumpconfig 'add', 'error', 701, 1, '/opt/my_dir'  --Unix
go
sp_shmdumpconfig 'add', 'error', 701, 1, 'D:\my_dir'    -- Windows
go
```

The second instruction adds the error 701 condition to initiate a memory dump. The fourth parameter indicates the number of memory dumps to capture, in this case 1. ASE will not capture additional memory dumps on this condition until ASE is recycled or the counter is manually reset. The last parameter shown is the name of a directory to hold the memory dump. Note that the file system on which the directory resides should have enough free space to hold the memory dump file, which can be large.

You can verify the dump conditions currently defined by running `sp_shmdumpconfig` without any parameters. This will also show an estimated size of the memory dump to be captured. You should verify that the disk space available in the directory is larger than the estimated memory dump file size.

A file name will automatically be generated that includes the date and time of the memory dump. Once the memory dump has been captured, you can remove this condition using

```
sp_shmdumpconfig 'drop', 'error', 701
go
```

If you wish to stop all CSMDs, configure `'dump on conditions'` to 0.
Once the memory dump has been captured, you should open a case with Tech Support and upload the memory dump to the ftp site (for instructions, see page 39).

You should also include the output from the following SQL statements. These use the MDA tables within ASE, which is automatically setup when running the `installmaster` script in ASE 15.0.2 or later. Earlier ASE versions need to run the `installmontables` script. See existing documentation for configuring the MDA tables.

```
select * from master..monProcedureCacheMemoryUsage
```

```
select * from master..monProcedureCacheModuleUsage
go
```

Note: the contents of the MDA tables shown above do not contain any useful information for customers.

By default, ASE will send the 701 error message to the client. You should consider getting this message also reported in the errorlog, so that you can track how often it is happening.

This can be done as follows:

```
sp_altermessage 701,'with_log',true
```

Performance Problem with a Limited Number of Queries

If a limited number of queries are not performing well due to suboptimal query plans or suboptimal resource consumption, you may want to consider installing the latest ASE 15.0.x release on your development server to see if the problem still exists. If the problem still exists, or if you cannot test the latest ASE release, then a reproduction should be collected and submitted to Sybase TechSupport. If you cannot send a reproduction to TechSupport, you should use the following steps to gather diagnostics:

1. Create a script file (here, named **sql.txt**) containing these commands:

```
select @@version
go
select @@optgoal
go
sp_cacheconfig
go
sp_configure 'nondefault'  -- only if you're running 15.0.2 or later!
go
dbcc traceon(3604)
set showplan on
set statistics time, io, plancost on
set option show long
go
<your query text>
go
```

2. Run `sql.out` using `isql` and capture the output in a file (WARNING: the output file can potentially be very large -- potentially Gigabytes for complex queries with many tables, under `allrows_dss`):

```
isql –Usa –P yourpassword -S YOUR_SERVER_NAME –i sql.txt –o sql.out
```

The following information should also be included so that Tech Support can attempt a reproduction:

- The files `sql.txt` and `sql.out`. If applicable, include a case with the 'fast' query plan (`sql.fast.txt`) and one with the 'slow' query plan (`sql.slow.txt`) and corresponding files

`sql.fast.out`, `sql.slow.out`

- DDL for the base table(s) and index(es); this can be generated with the `ddlgen` utility
- Simulate statistics output for the base table(s) via `optdiag`:

```
optdiag statistics simulate <fully-qualified-table-name> -Usa
-P yourpassword -S YOUR_SERVER_NAME -o <output-file>
```

- A copy of the ASE configuration file, or, in 15.0.2, the output of `sp_configure 'nondefault'`
- If the query is using view(s) or stored procedure(s), then also include their SQL source code as obtained by `defncopy` or `ddlgen`.
- The output of `sp_monitorconfig 'all'`.

System-Wide Performance Problems or High CPU Usage: step 1

If the performance of ASE, at a server-wide level, is not acceptable, and you are running 15.0.2 ESD#3 or later, please follow the steps below. This is also required when ASE is experiencing unusually high levels of CPU usage without a clear cause.

This remedy is relevant mostly when running a multi-engine ASE server; for a single-engine ASE server this procedure may not yield as much effect. Please enable this traceflag only as described below:

- Shut down ASE and restart it with traceflag 757 in the **RUN_***server* file (i.e. add **-T757**)
- If ASE cannot be rebooted, you can try running the following `dbcc` commands instead. It should however be noted that this may have less effect than an ASE reboot:

```
dbcc traceon(757)
go
dbcc proc_cache(free_unused)
go
```

Broadly speaking, the effect of this traceflag is that allocation/deallocation algorithms in the procedure cache behave differently on some points.

Please report to TechSupport (by opening a case) whether these steps make a difference to the situation in your ASE server. When reporting this information, please refer to this whitepaper and request your case to be linked to CR484362; please also indicate whether you rebooted ASE or not.

Note that this traceflag should not be used in any ASE versions earlier than 15.0.2 ESD#3.

System-Wide Performance Problems or High CPU Usage: step 2

If the performance of ASE, at a system-wide level, is still not acceptable after the step above, Sybase TechSupport needs more information about how ASE is spending its processing time.

Since, in cases like these, there is no error condition to configure a memory dump for, you will need to capture a Manual Shared Memory Dump. To do this, you need to use SybMon, which is an undocumented diagnostic tool.

NOTE: THE INFORMATION OBTAINED BY THE FOLLOWING PROCEDURE IS USEFUL FOR SYBASE TECHSUPPORT ONLY AND DOES NOT CONTAIN ANY INFORMATION THAT CAN BE USED BY CUSTOMERS. STRICTLY NO OTHER COMMANDS SHOULD BE USED BY CUSTOMERS OTHER THAN AS INSTRUCTED BY SYBASE TECHSUPPORT AND THIS DOCUMENT. PUBLIC DISCUSSIONS ABOUT SYBMON, SUCH AS IN NEWSGROUPS, SHOULD BE AVOIDED SINCE INCORRECT INFORMATION MAY BE COMMUNICATED THAT MAY BE HARMFUL TO YOUR ASE SERVER! USING SYBMON OTHER THAN AS INSTRUCTED BY SYBASE TECHSUPPORT AND THIS DOCUMENT, OR FOR DIFFERENT PURPOSES, IS ENTIRELY AT YOUR OWN RISK.

SybMon is started as shown below. You must use the same OS login that was used to start ASE in order to avoid permission problems. Note that the *password* is not the **'sa'** login password, but a special password that can only be obtained from Sybase TechSupport. Please contact TechSupport if you need to run this step.

Please note: it is essential that the following procedure be performed when performance problems are actually occurring, since only at that moment can the right information be gathered that is needed to diagnose the issue.

On Unix:

```
dataserver –X –P password –D<path to directory with .krg file>
```

On Windows:

```
sqlsrvr.exe –X –P password -D<path to directory with .krg file>
```

If you have only 1 krg file in the directory, ASE will automatically attach to the shared memory region and, if the server is named "PROD1", you will see a prompt like this:

```
PROD1:active>
```

If you have multiple .krg files, you will need to attach to the correct server's shared memory using the command:

```
attach <servername>
```

Once you have attached to the shared memory region, run the following commands:

```
log on <not-yet-existing file>
set display off
sample count=150 interval=200
set display on
log close
quit
```

The output will be in the file specified in the "log on" command.
In addition, you should run sp_sysmon (for a 1- or 2-minute interval). Submit the output from both SybMon and sp_sysmon to Tech Support.

Uploading diagnostics to Sybase TechSupport through FTP

Once the required diagnostics have been captured, first open a case with Sybase TechSupport. Then, use the following instructions to upload diagnostics to the Sybase ftp site:

```
ftp ftp.sybase.com
user: anonymous
pwd: <your email address>
cd /pub/incoming/wcss
mkdir <your case number>
cd <your case number>
bin
put <memory dump filename>
quit
```

Conclusions And Recommendations

The main conclusions and recommendations from this document are the following. Please refer to the mentioned pages for details:

- The query optimizer in ASE 15 is significantly different from the optimizer in prior ASE releases, offering great performance benefits, but also introducing new tuning aspects (see pages 3 - 21).

- For customers who prefer to limit the required testing effort for upgrading to ASE 15, 'Compatibility Mode' may be used in ASE 15.0.3 ESD#1. However, this may result in not being able to benefit from the query processing enhancements in ASE 15 (see page 12).

- Do not initially use parallel processing when upgrading to ASE 15, even when parallelism was used in ASE 12.x. ASE 15 may well deliver better overall performance in serial mode than pre-15 in parallel mode, especially for DSS-type queries with optimization goal `allrows_dss` (see page 8).

- More procedure cache will be needed in ASE 15 than in 12.5; expect to require 2 to 3 times as much procedure cache as in pre-15 (see page 14).

- Up-to-date statistics are more critical for good performance in ASE 15 than ever before. Preferably, run update index statistics, and if needed, configure additional histogram steps (see page 21).

- Sybase recommends testing your applications and queries extensively before upgrading your production system to ASE 15. For best results, test on a realistic production database with actual production queries, using realistic workload levels and real-life concurrency (see page 22).

- When encountering unexpected query processing issues, upgrade to the latest ESD available; Sybase is making continuous improvements and your issue may already have been fixed (see page 12).

Enjoy Sybase ASE 15 !

Reference Documents

- Technical whitepaper *"Changes to scope and semantics of session-level optimization settings in ASE 15.0.2"* : http://www.sybase.com/detail?id=1056206

- Technical whitepaper *" Using 'Compatibility Mode' In ASE 15.0.3 ESD#1 For ASE 15 Migration"* : http://www.sybase.com/detail?id=1063556

- Technical whitepaper *"Required SQL code changes when migrating to ASE 15"* : http://www.sybase.com/detail?id=1063534

CONTACT INFORMATION

For Europe, Middle East,
or Africa inquiries:
+(31) 34 658 2999

For Asia-Pacific inquiries:
+852 2506 8900 (Hong Kong)

For Latin America inquiries:
+770 777 3131 (Atlanta, GA)

SYBASE, INC.
WORLDWIDE HEADQUARTERS
ONE SYBASE DRIVE
DUBLIN, CA 94568-7902 USA
Tel: 1 800 8 SYBASE

www.sybase.com

SYBASE®