Introduction

Good performance of database applications is essential: no DBA, developer, manager or end-user would disagree that fast response times and high throughput are vital for any application to be successful. Interestingly however, this commonly shared understanding doesn't always seem to translate into sufficient attention to performance issues during system design and development. As a result, performance problems occur all too often: many, if not most, IT professionals have seen examples of applications turning into a failure because of their unacceptably bad performance and slow response times, even though it may have worked correctly from a functional point of view.

Somehow, the concept of "performance" doesn't always appear to be as straightforward as it may seem. After all, performance can be expressed in simple and clear figures, such as "1 second response time", or "100 transactions per second", suggesting performance is really an objective issue. However, the only time when there is anything easy about such figures is when they represent actual performance, which can be determined by simply firing off a query and measuring the response time. When those deceptively simple figures represent a performance target to be achieved, things are suddenly not obvious or straightforward at all, because this involves a complex set of interacting factors which all affect performance in some way. Dealing with this complexity is the real challenge in performance tuning.

This article will discuss various performance-related issues, both technical and non-technical, and provide some recommendations to help avoid problems. Due to the large number of issues related to performance in one way or another, it is impossible to cover all of these in this article. Therefore, a (partly arbitrary) selection has been made of some issues that occur regularly in practical situations.

Different angles on performance

Bad performance of ASE-based applications can have many different technical reasons. I prefer to distinguish the following main categories of performance-related issues:

- **Efficiency of algorithms.** Processing algorithms used in applications should be designed with performance in mind. If this is not done properly, performance is likely to suffer. Problems in this area are usually the result of improper software design and development.
- **Efficiency of SQL queries.** To write fast and efficient SQL queries, it is important to have a good understanding of the underlying data model and to be aware of ASE-specific aspects of query processing. Insufficient attention to these issues can lead to very inefficient SQL queries, resulting in severe performance degradation. Problems of this type are often the result of insufficient ASE knowledge or insufficient testing during system development.
- **ASE server configuration issues.** This involves the efficiency of ASE's internal resource usage, with issues such as memory allocation, data cache partitioning (named caches and buffer pools), internal data structures, etcetera. The system stored procedure `sp_sysmon`can be used for diagnosing potential problems in this area.
- **Host system resources.** Because every ASE application is running on some computer system, the hardware capacity (CPU, memory, disk, network) and operating system (OS)-level resources constitute limits to the performance that can be achieved by ASE. OS-level tools, like the Unix commands "iostat", "vmstat", "sar" and "top", or the NT "perfmon" utility, should be used to determine whether there are any performance bottlenecks in this area. Solutions to performance problems in this category may involve purchasing additional hardware.

Bottlenecks in any of these categories can have a significant impact on performance. However, in my experience as a performance & tuning consultant, the most common performance problems are those caused by inefficient application algorithms and inefficient SQL queries. These are usually the result of sub-optimal software engineering decisions during system development, which could have been avoided with very little effort if only the right choices had been made. What's worse is that fixing this type of problem tends to be expensive because it usually requires additional software engineering effort. Some examples of these two categories will be discussed later in this article.

There are some further points with respect to performance which are worth mentioning:

- **Performance testing.** When specific performance levels must be achieved, special attention should be paid to performance-specific testing, so that problems can be identified and fixed during development. In practice however, this is often done only superficially or omitted altogether, because performance testing is both expensive and complex.
- **Performance politics.** Whether performance is considered "good" is not just a technical issue. Non-technical factors, such as the end-user's perception of the system and their experiences with other applications, may also play a role. Political issues between the IT department and other departments may also influence discussions about performance. Most technology-oriented software engineers would probably prefer to stay away from non-technical issues like these. Unfortunately, the powers of politics, perception and prejudice can be strong, and technical

measures alone are usually not sufficient to counter them. Therefore, best not ignore this aspect of performance problems.

Let's now take a look at some specific recommendations to help avoid performance problems.

Recommendation #1: Avoid classical "one-by-one" algorithms

A common cause of bad performance lies in a wrong choice of application algorithms, namely those based on what I'd call "one-by-one" processing logic. This is the classical type of processing algorithm that has been used successfully for decades in applications written in programming languages such as Fortran or COBOL. Basically, it consists of a program loop which iteratively processes a "next" item until some end condition is reached. This approach was so efficient in those legacy systems because it closely matched the data storage structures, such as sequential files, that were used in those systems: a fundamental characteristic of these data structures is the concept of a "next" record and an "end" of the data.

In contrast, ASE, being a relational DBMS, does not offer a data storage concept, but rather emphasises retrieval of data through relational operations using the SQL language. ASE has been designed to yield best performance when accessing data relationally, through SQL operations such as "select" and "join". It is therefore strongly recommended to follow this relational approach because it tends to be significantly faster than the classic method: the difference is typically one or more orders of magnitude.

In database literature, the classic type of algorithm is also referred to as "record-oriented" processing indicating its storage-oriented nature, while the relational approach is known as "set-oriented" processing, after the underlying mathematical set theory.

As an example of what a wrong processing algorithm can lead to, I recall an application that could produce a list of the names of all customers named "Smith" (there were about 300 of them) in just one second, but once you'd want to see their phone numbers as well, this took more than a minute. The reason for this slow response was that, after retrieving the customer names very efficiently through a single SELECT statement, a stored procedure was executed for every individual customer to obtain the phone number. There was nothing wrong with this stored procedure itself, as it was perfectly optimised, but executing it some 300 times in a row caused the overall response time to slow down. If you were interested in both the customer's phone numbers and their addresses, an additional stored procedure would also be executed 300 times to retrieve each customer's address, slowing down response time further.
Ironically, the phone number and address data could also have been retrieved as part of the initial SELECT query at no extra performance cost because it was stored in the same database table as the customer's name. However, the client application, written in C++, was designed in such a way that the processing algorithms could not be changed without a complete redesign of the application.

This case is an interesting example of a fundamentally wrong design decision: the algorithm will work efficiently only when selecting individual customers or very small groups of customers. In practice however, retrieval of large groups of customers appeared to be an essential function, which the application couldn't do very well. For performance problems like these, faster hardware, server configuration changes or creation of additional indexes will do little to alleviate the problem: the only effective solution is to implement a better application algorithm.

Recommendation #2: Avoid loops and cursors in SQL

In SQL code, such as stored procedures, it is technically possible to implement "one-by-one" algorithms to retrieve data from database tables by using the "cursor" feature of Transact-SQL. A cursor offers the possibility to retrieve individual rows one by one, as opposed to normal SELECT statements which produce a (conceptual) table as a result. Thus, a cursor can be used to artificially create the concept of a "next" row, and in combination with the Transact-SQL "while" statement, the classical "one-by-one" algorithms can be implemented.

The difference with the "one-by-one" algorithms discussed in the previous section, is that those algorithms are running at the client side, and are usually written in a programming language other than SQL. The cursor-based loops meant here are written in SQL and executed at the server side (for clarity, let's limit ourselves to "server cursors" and "language cursors" here, as these are the types of cursors most commonly used by developers). Note that loops can also be created without cursors by using some incremental loop control logic, possibly in combination with the "set rowcount 1" statement; however, this type of loop tends to be even slower than cursor-based loops.

While Transact-SQL offers the functionality to create loop-based programs, it is best to avoid this in the interest of performance: as a rule of thumb, a SELECT statement is about a factor 10 faster when run in its plain, standard form, than when executed through a cursor. In addition, cursors often have side effects like lock contention in a multi-user environment, causing further performance degradation.

Only in very special circumstances may the use of cursors be justified from a performance point of view, for example in case of extremely complex relations between entities which cannot really be implemented efficiently using standard SQL.

In my experience however, such exceptions are rare: since I started working with Sybase SQL Server in 1989, I have seen only a handful of cases where the cursor-based approach was indeed the best option. Therefore, best avoid cursors and loop-based SQL code altogether, and accept their use only after careful analysis.

One of the more interesting examples I've seen of how not to program in SQL, was a stored procedure containing three nested cursor-based loops, which essentially implemented a three-table join. In this case, various pages of SQL code could be replaced with a single SELECT statement performing this join, running more than 500 times faster than the original.

Despite these disadvantages, software developers often choose to use loops and cursors in an SQL environment because it allows them to continue programming according to the classical "one-by-one" approach which they are so familiar with. Such preferences often come from insufficient understanding of the concepts of SQL: as a non-procedural language, SQL requires programmers to think in a different way than when using the more traditional programming languages such as C, Basic, COBOL, etc. Ensuring that software developers have been sufficiently educated and trained in the use of SQL will help to avoid this type of performance problem.

Recommendation #3: Use stored procedures as the client-server interface

In a client-server architecture, the client application communicates with the server by sending it the SQL commands to be executed. There are two main methods to implement this interface: in the first method, the client generates the complete SQL queries for every function the client needs to perform, and sends these to the server. In the other method, those SQL statements have been encapsulated in stored procedures which reside on the server; the client then just executes those stored procedures.

While both methods can be used to implement the same functionality, there is an important performance-related aspect to be considered when choosing between these two methods. Because stored procedures are developed as separate objects, they can be optimised in advance as the particular functionality is already known during development. In contrast, when a client application dynamically generates a text string containing the full SQL command to be executed, it usually doesn't pay any attention to performance aspects. As a result, it may well be possible that such a query won't execute very efficiently, for example because there appear not to be any indexes to support the query. When using stored procedures, such problems can be avoided by optimising the code, and creating the proper indexes, in advance.

These problems often occur with the type of windows-based applications with a graphical interface where the end-user can specify the data to retrieve by clicking on the desired tables and columns and indicate relations by drawing lines between the different tables' attributes. The application then dynamically composes the corresponding SQL query and sends it to the server for execution. However, it is not unusual for this type of application to perform badly due to the reasons described above.

Another situation where free-text SQL queries may cause problems is when certain users, such as supervisors or managers, have full SQL access to the database, probably as a result of political factors ("I am the boss so I need unrestricted access"). However, because such users normally don't excel in SQL skills or knowledge of the technical details of the database structure, this often results in terribly inefficient queries featuring Cartesian products, non-indexed data access, etcetera. Such ad-hoc queries are known in database literature as "queries from hell", because of the devastating performance effects they can cause. Therefore, this capability should be restricted to knowledgable users in order to minimise risk.

In short, it should preferably be avoided for clients to issue complete SQL queries, because their contents are as variable and unpredictable as the users who, directly or indirectly, generate these queries. Instead, stored procedures should form the interface between client and server. Note that this recommendation also applies to the new "execute immediate" feature in version 12.0 of ASE. This feature allows queries to be built and executed dynamically, even inside stored procedures. In the interest of performance, this should best be avoided, for the reasons described above.

One reason why application developers sometimes prefer to avoid stored procedures is that it simplifies the software development process: when the client issues the SQL queries, all functionality is concentrated in one place. When using stored procedures, the application functionality is split into two parts, making development more complex due to the extra interfaces. This way, developers are effectively trading the efficiency of their development against the efficiency of the application, but this is not likely to be a conscious and approved decision by those developers.

Another way of looking at the performance aspects of client-level SQL statements, is that this represents a tradeoff between maximum functionality and flexibility for end-users and the performance that can be guaranteed to those users. When both aspects are equally important, this may mean a lot of effort has to be invested in the development of optimised stored procedures to cover as many practical situations as possible. As always with tradeoffs, a choice will have to be made.

Recommendation #4: Have ASE-specific knowledge available

When developing SQL queries in an ASE environment, a good understanding of how ASE will process those queries is important for achieving optimal performance. Although ASE will automatically execute most SQL queries very efficiently, the universal 80/20 rule applies also here: there is always a small category of queries which don't perform very well and require additional tuning effort in order to run faster.

Optimising SQL queries requires understanding of rather technical topics such as the way ASE's query optimizer works, the different available indexing techniques and topics such as table partitioning and locking. Fortunately, the ASE Performance & Tuning Guide contains extensive information covering the ins and outs of these issues in the chapters "Understanding the Query Optimizer", "How indexes work" and "Indexing for Performance". These chapters should be considered mandatory reading for all SQL developers who will have anything to do with ASE query performance (note: like all ASE documentation, the Performance & Tuning Guide can be downloaded from the Sybase website as a printable PDF document. Go tohttp://sybooks.sybase.com/ , click on "Database Servers", then on "Adaptive Server Enterprise"; in the left-hand pane, select a "Generic" collection, then click the desired manual in the right-hand pane. A PDF icon now appears at the bottom of the page - click it to download).

Performance problems caused by inefficient SQL queries can be severe: it is not uncommon to see response times improve by a factor 1000 or more after making only small changes to the query or add or modify indexes. In one of the more spectacular cases I've seen, response times dropped from 10 hours to 5 seconds after a join clause with incompatible datatypes was fixed.

In an ideal world, your development project would be staffed with experienced software engineers, having plenty of this ASE-specific knowledge. From their previous experience with similar ASE-based applications, they would know which big and small traps to avoid. However, back down here in reality, things are often a little less ideal. As skilled and experienced staff is one of the most expensive resources involved in system development, there is often not enough of it available, which can lead to performance problems. Education and training are the obvious measures to take in this context, but this may not always be feasible. Still, many problems can be avoided by having the more experienced staff share their technical ASE experience by creating a checklist of ASE-specific DOs and DON'Ts for the benefit of the less experienced developers (due to space limitations, it is unfortunately not possible to include such a checklist in this article). Such a checklist could be part of the development or coding standards already in use in your organisation.

Recommendation #5: Understand the limitations of "sp_sysmon"

The system stored procedure `sp_sysmon` was introduced in ASE version 11.0 as a tool to obtain information about the behaviour and resource usage of internal server processing, primarily for performance optimisation purposes. Prior to version 11.0, there was no insight into these aspects of ASE at all, and `sp_sysmon` consequently received a lot of attention as a new angle on ASE performance tuning. However, because the `sp_sysmon` output is very technical, it can be quite difficult to interpret and translate into specific actions to improve performance (in fact, last year in a presentation about the new Q Diagnostics tool, Peter Thawley mentioned the need for "rocket scientist DBAs" as a disadvantage of `sp_sysmon`). In combination, these aspects have helped to create an image where mastering `sp_sysmon` is seen as a kind of black art, holding the key to solving all performance problems.

However, this seems to be an overvaluation of `sp_sysmon`'s capabilities, as it provides little or no information to help diagnose and pinpoint very common performance problems caused by inefficient algorithms or queries, such as those discussed earlier in this article. For example, an inefficient query performing unnecessary table scans on the inner table of a join, might result in `sp_sysmon` statistics showing a high disk I/O rates, a less than optimal cache hit rate and high CPU utilisation. From such figures alone, one might wrongly conclude that more memory, CPUs or disk bandwidth is needed because the problem is in the availability of OS-level resources, rather than in badly written queries.

Because `sp_sysmon` simply does not provide any information that can be related to individual queries or tables, it does not make sense to draw conclusions about such performance issues just from the `sp_sysmon` output. `sp_sysmon` can be a very useful tool for tuning certain aspects on the level of the ASE server, but it will be most valuable when application-level performance issues are known to have been resolved first.

These misconceptions about `sp_sysmon` may be easier to understand with some of ASE's background in mind: historically, Sybase has always been strongly focused on the needs of high-end transaction processing environments. In these types of systems, application logic and query efficiency usually did not cause performance problems, as these were custom developed and their optimisation received sufficient attention. The remaining bottlenecks used to be on the level of ASE's resource usage, and `sp_sysmon` does indeed provide the necessary information to help configure the server more optimally.

Recommendation #6: Enforce systematic performance testing

Building an application which doesn't perform is one thing, but discovering this only after it has gone into production, is quite another. Yet, this is suprisingly often the way things go, because proper performance testing seems to be the

exception rather than the rule.

One reason for this omission is that performance testing is more difficult than "normal" software testing, which is mainly about proving the correctness of individual pieces of functionality, and usually takes place in a small test environment. For performance tests to have any predictive value, a test environment is required which models the future production environment as realistically as possible. This involves aspects like:

- A test database of real-life size and contents. On a small database, even inefficient algorithms and queries are still fast enough, but with increasing database size, performance can deteriorate quickly.
- Realistic workload simulation. This is a difficult aspect to model in a test environment as it involves simulating the activity of end-users and other existing applications. Still, when the production environment can be matched closely, this can deliver very valuable information. For example, concurrency problems can be revealed, which would be very difficult to identify otherwise.

Performance testing should ideally be done at two levels. First, during development, every piece of SQL code should undergo performance-oriented tests, which involves inspection of the "query plan" and the amount of I/O that has been performed. The aim should be to determine whether the code is unreasonably slow for the function it performs. With some simple rules of thumb, developers can quickly be taught what symptoms to look for, so that they can hand over suspected problem cases to performance experts. Tests like these will require realistic database size and contents.

The second phase of performance testing should be on the level of the completed application. This is where workload generation would play a role, and concurrency issues may be discovered.

Altogether, thorough performance testing may take a significant amount of time and resources, making it an expensive operation. For this reason, it is often skipped partially or completely, effectively postponing performance testing until the application goes into production. Needless to say, this may not be a cost-effective strategy.

When it comes to performance testing, there often is a strong feeling that problems are not likely to occur. However, unless realistic performance tests have been performed, such optimism is likely to be unjustified.

Recommendation #7: Data modelling software: use with care

Many software development projects use software tools to create a logical data model (i.e. the entity-relationship diagram) for the database that's being designed. When it comes to the implementation stage, such tools can automatically generate the physical data model, in the form of the required DDL (Data Definition Language) statements. While this may certainly save a lot of manual typing, it is often forgotten that the translation from the logical into the physical data model involves some important performance-related decisions which the data modelling tool cannot be aware of. As a result, the generated physical data model may lead to less than optimal performance.

A typical example is the choice of indexes: by default, data modelling tools almost always create a clustered index for a primary key. It is however not at all obvious that this will be an optimal choice; a nonclustered index would often perform better, especially for artificial keys. Some data modelling tools also generate the implementation of the referential integrity identified in the logical data model by littering the database with RI constraints and triggers, leading to performance problems later.

The bottom line is that data modelling tools are best suited for modelling the logical data model. It can be useful to generate a first version of the physical data model to get started, but for the rest this should be done by knowledgeable developers.

Recommendation #8: Don't believe the sales rep

There are many third-party applications which use ASE for database functionality. The application vendors often create the impression that the underlying RDBMS will be a "black box" which you won't have to worry about. However, just as often this appears not to be the case, and additional DBA effort may be required to keep the system operational. Also, performance may leave much to be desired: applications like these often use the type of "one-by-one" processing and inefficient SQL queries which lead to inferior performance.

The reason for such problems is often that companies making, let's say, ERP software, are good at implementing a payroll program, but tend to be less accomplished in the area of RDBMS-specific software development. Another factor is that this type of application can often run on a number of different RDBMS products, depending on the customer's preference. This means that the database functionality is often kept generic and ANSI-complaint for portability reasons, thus not taking advantage of the existing optimisation techniques for each of those different databases.

A common problem with this type of application is that it may not be possible to make any improvements to the situation,

even when it is clear which parts of SQL code can be optimised or which indexes should be changed. It is not uncommon that the vendor's license terms prohibit the customer from touching anything in the database, with the consequence of losing support or being in breach of contract. In such cases, you sometimes see very powerful hardware being brought in as this may be one of the few, yet expensive, options to alleviate the performance problems a little. Obviously, this type of solution is not ideal.

While this is not really an ASE-specific or a technical topic, the consequences of an unfortunate choice can be significant. Therefore, don't fall for the slick sales talk, but pay attention to RDBMS-specific issues, specifically performance aspects. Ask for reference sites with database sizes and performance figures comparable to your situation and check out the support agreement - and give your sales person a good run for his money. Effort spent in this area may prove to be very cost-effective in the end.

Conclusion

Achieving well-performing applications may seem easy, but often appears difficult in practice. Because performance involves many different aspects, there is no single success factor holding the key to avoiding all performance problems. However, some common problem areas are the use of inefficient algorithms and queries, and a lack of proper performance testing. The recommendations in this article may help to avoid many performance problems.