Performance and Tuning

Purpose

The purpose of this page is to discuss diagnosing application performance issues with SQL Anywhere.

Overview

We present a method for troubleshooting performance problems in SQL Anywhere. We describe the nature of performance problems and classify them into categories of CPU-bound, I/O-bound, and concurrency-bound problems. We describe the steps a DBA should take to classify a performance problem, the tools provided in the product to examine the problem, and the analysis that must be done to resolve the issue.

Introduction

In this paper, we will describe the steps you should take, as an application developer or DBA, when you begin to hear users reporting that your database application is performing slowly.

Given this question as our starting point, let's consider what we mean by "database application." At an architectural level, the database server can be thought of as a black-box query machine: queries go in, data comes out. When designing an application, it is convenient to treat the database in this way - as an appendage of the application. However, the database server is much more closely integrated into a typical application than this. For many database applications, it is not possible to consider the database server in isolation; rather, we must consider the entire application, end to end. This means considering not only the resource consumption and timing on the database server side, but also the patterns of use of the database server by the application.

We should also consider what is meant by the other half of our question: "performing slowly." Since slowly is a relative term, we need to be aware of what expectations make up our comparison. These expectations are often legitimate, but can be hard to quantify. In practice, what often happens is that a rough mental comparison is being made to some other similar application, or similar set of operations, on a different system. The validity of such a comparison depends on just how similar the other system is.

It is easier to quantify what poor performance means when you have another installation of the application that performs well. This will often be a test instance of the application, or perhaps the previous behavior of the application whose performance was historically good, but has now degraded. It will be much easier to troubleshoot application performance if you can nd such an instance - you will be able to focus your efforts on the parts of the application that have degraded, and see how they differ from the original instance.

There are several common reasons that may cause an application to degrade in performance over time or in comparison to another instance. The most common is that the application has scaled beyond its original size. This might be in terms of

- the absolute data volume (size of the database instance);
- the number of clients/connections to the database; or,
- the volume of requests.

A more subtle form of scaling problem is that the data distributions inside the database no longer match those used in the test database or original application; this is a type of selective scaling, and can have just as significant an impact on performance as an absolute increase in database size.

Other possibilities are that the application itself does not require more resources, but that it has to compete with other resource consumers. For example, perhaps other applications on the same machine are restricting the memory, i/o bandwidth, or cpu usage of the database server. Another possible cause of application slowdown is increased network congestion - this is especially true for applications where the client and server interact with numerous short requests.

Steps in Diagnosing Application Performance Problems

The process we will outline in this paper has four steps:

1. Broadly characterize the nature of the performance problem and the constraining resources.
2. Collect relevant information about the database application performance, based on the category of problem identified in Step 1.
3. Analyze the collected data to identify the specific cause(s) of the slowdown.
4. Implement and test possible solutions.

See the WIKI links for walkthroughs for how to capture performance data in a tracing database and how to analyze the tracing database afterwards.

Characterize the Problem

When troubleshooting a performance problem for a large and complex database application, it can be difficult to know where to begin. The best way to start is to gather information about the problem that is both high-level and easy to obtain. The simple process of laying out the problem can
often lead to insights about contributing factors to performance difficulties.

High-Level Characteristics

The idea of gathering high-level information is to look for patterns in the application performance. For example:

- Does the problem affect all users? An identifiable group of users? Just one user?
- Is there a specific operation that triggers the appearance of the problem?
- Is there a temporal pattern to the performance? For example, is the performance adequate in the morning but inadequate in the afternoon, as the business day progresses? Keep in mind that for distributed applications, the busy time of day will shift as oces in different time zones become active.

This kind of information will be helpful because it will let you know which portions of your application must be examined in detail.

Characterizing Resource Usage

Database performance problems can be broadly classified into two categories - either the server has run out of one of the basic computational resources (cpu cycles, ram, disk i/o bandwidth, network i/o bandwidth), or the server has these resources available, but is unable to use them fully. The rst problem we will call the resource exhaustion problem, the second we will call the contention problem.

We need to determine which category of problem we are looking at early in the resolution process. Fortunately, it is often quite easy to characterize the basic nature of our performance problem using tools provided by both SQL Anywhere and the underlying operating system. In this paper, when giving specific examples, we will assume the database server is running on top of Microsoft Windows. The examples are readily transferrable to other operating systems - for example, instead of using Performance Monitor (perfmon) on Windows, you would use utilities such as iostat and top on Linux.

The following performance counters, exposed by the os, will help to clarify what category of problem the server is experiencing. In Windows, these are viewable with either the Task Manager (taskman) or with perfmon.

- CPU usage
  - Is the database server using all cpus made available to it? This may be the number of cpus on the system, but may be fewer depending on the server configuration and licensing. If yes, your application may be cpu-bound on the database server. See the CPU-Bound Applications section for more information.
  - Is another process on the machine using the cpus and preventing the database server from using them? If such a process exists and it is part of your application, your application may be bound in the client-tier. See the Client-Tier Performance Issues section for more information.
- Disk activity
  - How active are the disks used by the database server? On a Windows server, a very useful statistic is the % Idle Time, which can be found in perfmon for all Physical Disk objects. If this idle time is low, your application may be disk-bound. See the I/O-Bound Applications section for more information.
  - A very quick and dirty estimate of disk activity is to simply observe the machine and see whether any of its disks appear to be stressed (LED fully lit or almost-fully lit, and the disk making lots of noise). This kind of direct observation of the disk does not replace monitoring disk activity through the tools provided by the OS, but it can confirm that a low reported disk idle time is connected to obvious physical disk activity.
- Active Requests
  - The number of active requests (see the Concurrency-Bound Applications section for how to obtain this value) can show whether the server has work to do or not. If this number is non-zero, the cpu usage of the machine is low or relatively low (up to about 30%), and the disk idle time for all disks that are part of the server is above 5%, it is likely that the performance you are observing is constrained by concurrency bottlenecks.

Get an Application Profile

In general terms, a profile is a summary that provides enough information for you to extrapolate about the properties of the object summarized. For example, the profile of a person's face only contains information about the contour of that face, but from it, you can guess a lot about the general form of the face.

In the context of database applications, an application profile is a concise representation of the work performed by the database system on the application's behalf. From this profile, many of the complex behaviors of the applications can be extrapolated and analyzed.

Once you know what kind of problem you are dealing with (see the Characterize the Problem section), you will know what kind of data to collect (profile) to resolve it. See the detailed description of the category of performance problem that you have identied (in the Types of Performance Problems section) for advice on what specific information should be collected to analyze the problem.

Application Profiling Concepts: Architecture

Application Profiling is a term used in multiple contexts in the SQL Anywhere product. In its most general form, it describes the action of gathering and analyzing an application profile as described above. This is also the name of the mode in the SQL Anywhere plug-in for Sybase Central that facilitates this analysis, integrating tools added to version 10, as well as legacy tools as discussed in the Alternatives to Application Profiling section. Accessible from within this mode is the Application Profiling Wizard, which automates much of this process for simple configurations.
Diagnostic tracing is a term for the support in the SQL Anywhere database server for capturing, in an integrated way:

- sql statements,
- user contexts,
- blocking and deadlock events,
- query plans, and
- performance statistics.

When performing diagnostic tracing, the server establishes a special connection, over the tcp/ip protocol, to a database server running a tracing database. This tracing database may even be the database being profiled (often called the production database), or it may be another database running on the same server as the production database. In theory, any database can be used as a tracing database, but some of the profiling analysis tools will only work if the tracing database has been specially created (see the Application Profiling Concepts: Configuration section).

The tracing data is stored in global shared temporary tables during tracing capture. When the tracing session is closed, the data is moved to base tables, where it is indexed and readied for analysis.

Application Profiling Wizards

The Application Profiling Wizard will run the first time you switch to profiling mode in Sybase Central. If you have a relatively small application, or want the wizard to make many of the choices discussed in this paper for you, you will find it easiest to collect an application profile using this wizard. It will guide you through the steps of creating an application profile, and will perform some of the analysis discussed in the Types of Performance Problems section for you.

You will still be able to analyze the results manually. The main tradeoff in using the wizard is that you will have less ability to restrict the profiling process to specific parts or users of your application. If you use the Application Profiling wizard, you don't need to concern yourself with the details of tracing configuration as discussed in the Application Profiling Concepts: Configuration section.

For the analysis described in this paper, you will want to choose at least the second option in the wizard ("Performance of your database application"). You will also find it useful to choose the third option ("Overall database performance based on the database schema"), which will produce some additional recommendations for you with negligible additional overhead.

Application Profiling Concepts: Configuration

The diagnostic tracing features of the SQL Anywhere server are highly configurable. You will be able to request a trace of only those data items that are relevant to the type and scope of performance problem you are looking at.

You will be able to limit the tracing information you capture in several dimensions:

- **levels of data** - sql statements, query plans, deadlocks, etc.
- **scopes of data collection** - tracing events for the whole database, for a given user, for a given table, etc.
- **conditions** under which data will be collected - collect data only for statements that have a specified duration or that differ from their estimated run time by a specified percentage.

Detailed information about tracing configuration is available in the Related Documents section.

Starting and Stopping Application Profiling

There are four steps to acquiring an application profile. You must create or locate a database to store the tracing data; configure the tracing; start tracing and run your application; and save the tracing session.

Creating a Tracing Database

Any database can be used as a tracing database, but it is best to either create a tracing database specifically for tracing or to use the local database.

The local database is often the most convenient database, especially during application development, because it requires no extra work to set up. However, the size of data captured during tracing can bloat the local database. Since database files cannot be shrunk, you may want to avoid growing your local database by the large volume of data captured by tracing.

Instead, you can either send tracing data to a copy of your local database, or create a tracing database specifically for the purpose of storing an application profile. The advantage of creating a database specially is that it does not contain any unnecessary data.

If you are configuring tracing with the Database Tracing Wizard, you will be prompted to create the tracing database - if you are planning to host it on your local machine, it will also offer to start it for you. If you are going to run the tracing database on another machine, you must copy it and start the server on that machine before proceeding with tracing.

To create a tracing database manually, use the `dbunload` command with the `-n` and `-k` options.

Setting Tracing Levels

You can either set up tracing levels by using the Sybase Central Database Tracing Wizard (available by right-clicking the Database object in the SQL Anywhere plug-in tree), or by calling the `sa_set_diagnostic level()` procedure and then manually modifying the rows in the `sa_diag`
nomic_tracing_level table. It is recommended that you use the wizard, because it will ensure that only valid combinations of 
level/scope/condition are specified.

Attaching Tracing

Once you have tailored the tracing levels you are interested in to the specific circumstances of your performance investigation (see the Types of 
Performance Problems section for the particular type of problem you are investigating), you are ready to start a tracing session.

A tracing connection is ATTACHED to a tracing database to start tracing, and is DETACHED from it to end the session. The Database Tracing 
Wizard will prompt you for the connection string for the tracing database.

To attach tracing manually, issue the ATTACH TRACING statement. For example:

```
ATTACH TRACING TO `uid=dba;pwd=sql;eng=mytracingeng;dbn=mytracingdb`
```

For more information, see the Related Documents section.

Detaching Tracing

Once the behavior you are interested in has occurred, you can detach your tracing session. Ideally, you want to keep tracing as focussed as 
possible, by detaching as soon as a representative sample of the poor performance has been observed. For example, a tracing configuration that 
is capturing everything in a busy server should not run for more than a minute or two, while a much more selective con guration can comfortably 
run for hours.

Once you have captured a tracing session, you must detach the tracing connection, and choose whether to save the session information into the 
database for later analysis, or allow it to be thrown away. Normally, you will want to save tracing information | it would only be thrown away if you 
are sure that the performance behavior you were trying to troubleshoot didn't happen during that tracing session.

In Sybase Central, tracing is stopped by right-clicking the Database object and choosing to "Stop tracing with save" or "Stop tracing without 
save". To detach tracing manually, use the DETACH TRACING WITH SAVE statement (or DETACH TRACING WITHOUT SAVE to throw away 
what you have captured).

A detailed walk-through on Setting Database Tracing Using Database Tracing Wizard

Alternatives to Application Profiling

There are several technologies that are complementary to the Application Profiling/Diagnostic Tracing tools. In SQL Anywhere versions 9 and 
earlier, you will be limited to using the technologies discussed below.

Request Logging

The most direct analogue of Application Profiling (diagnostic tracing) in versions prior to version 10 was the request logging feature. This was a 
feature that allowed the text of all requests (statements) processed by the server to be recorded to a plain text log le in chronological order. In 
addition, some information could be recorded optionally, such as the values of host variables and the plans of queries executed by the server. The advantages of this approach are

- Its simplicity, in terms of configuration, operation, and analysis; and
- Its resilience, even to server crashes or loss of power on the machine. The log is flushed at every write, and so it is always current.

However, the data captured by request logging is limited in comparison to the much larger set of data that is archived through Application 
Profiling. Additionally, especially on a production server, the performance overhead of request logging is substantial, since all requests processed 
by the server are serialized into a single le. Also, the request logging feature has limited abilities to focus on specific parts of your application, 
while the Application Profiling tools allow you to focus the profiling activities on a very specific part of your application. For example, with 
application profiling you can focus on a specific user, connection, set of tables, set of procedures, or cost of queries, which can greatly reduce the 
impact of such profiling on a production system. For these reasons, it is recommended that Application Profiling be used in all versions where it is supported.

For information about how to enable request logging, see the Related Documents section.

Procedure Profiling

A second older technology that is also used for Application Profiling is the procedure profiling tool. This is very like a standard code profiling tool, 
in that it records the number of times a statement in a stored procedure or user function was executed and the total cost accrued to that 
statement. This functionality can be used to identify programming improvements that could be made to procedure code, as well as to find 
statements within procedures or (procedure call-stacks) that are expensive. While this approach does not give as much information as the 
Application Profiling tool, it is quite easy to use and can give a high-level overview of resource usage in procedure-heavy applications. In version 
10 and later, it is accessed from Profiling mode in Sybase Central.

In version 10 and later, procedure profiling is still useful for analyzing flow control in stored procedure code.

Index Consultant
The Index Consultant was a stand-alone tool in version 9; it is fully integrated into the Application Profiling feature in version 10 and later. This tool allows you to have the server recommend indexes to support the execution of either a single query or a set of queries (a workload). In version 9, this tool captured its own workload, but in version 10 and later, it is able to use the much more detailed information captured by the diagnostic tracing feature. See the Missing Indexes section for more information about how to use the Index Consultant.

**SQL Anywhere Console Utility**

dbconsole is another tool that is complementary to the Application Profiling tool for diagnosing performance issues. Whereas diagnostic tracing is a session-based tool that captures a detailed set of performance data over a period of time, dbconsole is a monitoring tool that gives you a picture of the current state of the database server. dbconsole can be used to quickly check the state of performance counters, such as the I/O counters discussed in the Database Cache is Too Small section and the ActiveReq counter used to detect concurrency problems.

**Analyze the Data**

Once you have collected data relevant to the type of problem you are experiencing, you will want to analyze it to find the exact cause of your performance trouble, and to suggest a solution.

See the details described in the Types of Performance Problems for each category of problem to see what to look for in the analysis.

In general, you will want to examine:

- expensive statements and their plans,
- indications of concurrency problems, and
- performance counters.

**Analysis Using Sybase Central**

The best way to analyze an application profile is from the Profiling mode of the SQL Anywhere plug-in for Sybase Central. This browser will provide a relatively simple, but powerful, view of the data, allowing you to find the information from it to support the analysis described in the Types of Performance Problems section.

A tracing session can be opened from Sybase Central by switching to Profiling mode, cancelling the wizard if it starts, and choosing “Open an analysis file or connect to a tracing database”. Choose to connect to a tracing database, and then give the connection information for that database (even if it is stored in the database you are already connected to, the profiling analysis mode opens a separate connection to it). See Demonstration on Opening the Profiling Information on the Tracing Database

**Status Pane**

When you first open a tracing database in Profiling mode, you are shown the Status pane. If you have saved multiple tracing sessions in this database, you will be able to pick which one to analyze by looking at the timestamp and size of the trace.

From here, you will want to examine the Summary, Details, and Connection Blocks panes, all contained within the Database Tracing Data pane.

**Summary Pane**

The Summary pane displays a high-level view of all statements that were captured during the trace. It groups “similar” statements together, so that you can see the net effect of different statement types. The server computes a signature for each statement captured, and groups statements with a similar signature together. A representative statement for each group is shown; it is important to remember that not all statements in the group are identical to this representative statement.

For queries (SELECT, INSERT, UPDATE, or DELETE statements), a signature is computed by looking at all tables and columns referenced by that statement. Variable and constant values are ignored when computing the signature. As a result, the following two statements would appear aggregated into the same group in the Summary view:

```sql
SELECT * FROM product WHERE id = 1;
SELECT * FROM product WHERE id = 2;
```

All non-query statements are assigned a single signature. For example, all CREATE INDEX statements have the same signature. One thing to watch out for is that all CALL statements also have the same signature (although this is not ideal and may be fixed in a future version). The individual statements that make up procedures will have their own entries.

You can click any column to sort the results by that column; clicking the column again reverses the sort order (from ascending to descending, or vice-versa). You can also right-click any statement in the Summary pane to see the details for each invocation of that statement.

The chevron in the top-right of the Summary pane allows you to open the filtering properties for the view. If you want to limit the time window of your view, or search for a particular statement, you can enter filtering information into this pane and Apply it. If you ever appear to have less data displayed than you expect, Reset the filter. (Similar filtering panes are available in the Details and Connection Blocks panes as well.)
The main purpose of the Summary view is to allow you to sort the statement groups by Total Time (given in milliseconds), in order to find the kinds of statements that, in aggregate, are the most time-consuming in your application. This could be either because they are expensive statements in their own right - that is, a single execution of a statement is expensive. Alternatively, it could be because the statement, although cheap in isolation, is executed many times, and is thus expensive in aggregate. Any improvements that can be made to these statements will have significant benefits for the application as a whole.

Details Pane

The Details pane is similar in layout to the Summary pane, and provides the same sorting and filtering capabilities. It displays a single line for every recorded execution of a SQL statement. Among the details recorded are the time the statement was executed, the total time (Duration) that the server spent actively processing the request, and the SQLCODE (if non-zero) for the statement.

For queries that opened a cursor, the cursor open and close times will be recorded. Note that the difference between the cursor close and open times may be much larger than the reported Duration, since the server may have quickly processed the answer to the query, but the client may have waited a long time before closing the cursor. This is something to look for, especially when troubleshooting concurrency problems (see the Application-Based Concurrency Limitations section).

From this view, you can right-click to open the SQL Statement Details window to see more details about any individual statement.

SQL Statement Details Window

This window shows all of the information displayed in the Details pane, but in a more readable way (since space is less constrained). It will also show additional details, including the values of any variables used in the statement, if the trace was configured to capture them.

If the SQL statement is a query, an extra tab will be available to show the plan information for that statement. Plan information is especially useful when troubleshooting CPU- and I/O-bound applications. The short text plan shown for a query is the actual plan that was used by that query in every case. If tracing was configured to capture graphical plans (either with or without statistics), the captured graphical plan will also be shown. If
the server did not capture graphical plans, it will attempt to reoptimize the query and show the plan it would have used, simulating the state of the server at the time the statement was originally executed. You must compare this "Best guess plan" with the short text plan before relying on it for analysis.

<table>
<thead>
<tr>
<th>Start Time</th>
<th>Duration</th>
<th>Query ID</th>
<th>Connection ID</th>
<th>Connection Info</th>
<th>SQL ID</th>
<th>Predicate Or Operation</th>
<th>Procedure Or Trigger</th>
<th>Statement Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>2010-01-01</td>
<td>0.01 s</td>
<td>123456</td>
<td>A123456ABC</td>
<td>NULL</td>
<td>NULL</td>
<td></td>
<td>NULL</td>
<td>(select column1 from table1 where column2 = 'value')</td>
</tr>
<tr>
<td>2010-01-01</td>
<td>0.02 s</td>
<td>654321</td>
<td>B654321DEF</td>
<td>NULL</td>
<td>NULL</td>
<td></td>
<td>NULL</td>
<td>(select column2 from table2 where column3 = 'value')</td>
</tr>
<tr>
<td>2010-01-01</td>
<td>0.03 s</td>
<td>789012</td>
<td>C789012GHI</td>
<td>NULL</td>
<td>NULL</td>
<td></td>
<td>NULL</td>
<td>(select column3 from table3 where column4 = 'value')</td>
</tr>
</tbody>
</table>

For analysis.
SELECT sa_split_list.line_num, sa_split_list.row_value FROM sa_split_list ORDER BY sa_split_list.line_num ASC
Connection Blocks Pane

This pane shows all blocking events that were captured (if the Blocking level was specified). These events will be particularly relevant for investigating application-based concurrency problems (see the Application-Based Concurrency Limitations section). For each block, the statement that was blocked, and the blocked and blocking connections will be shown. Right-clicking on any blocking event will allow you to see more details about the context in which it was blocked.

It is not possible to tell directly what statement the blocking connection was running; however, by checking the time at which the block occurred, you can go to the Details pane to see what statement that connection was running (or what cursor it had open) at the time that the block occurred.

Deadlocks Pane

If you configured your trace to include deadlock events, this pane shows you all deadlock cycles that were detected during the trace. In each case, it will show you all connections that were part of the deadlock, who blocked on each connection, and what connection was picked as the deadlock victim. If you are sending tracing data to the local database or to a copy of the local database, it will also show you the primary key value, if available, for the row on which each connection blocked. If you have large numbers of deadlocks, especially if they involve more than two connections, it is an indication that you have fundamental concurrency problems in your application; see the Application-Based Concurrency Limitations section.

Statistics Pane

If you captured statistics (numerical performance counters), then you will be able to view them graphically in the Statistics pane. Non-volatile counters (those that do not change from second to second) will be displayed at the same rate as the faster, volatile counters. This will cause the graph for non-volatile counters to have a stair-step appearance.

Although there is a wide variety of performance counters captured by the server, there are relatively few counters that will be useful for general performance analysis; most of the counters are useful only for internal SQL Anywhere development, or for troubleshooting very specific
performance circumstances beyond the scope of this paper.

Performance counters that count events are cumulative, monotonically non-decreasing counters. For performance analysis, you are often interested in the rate of growth of these counters, rather than the absolute value. For example, pick two points in time, say 60 seconds apart, and mouse over those points on the graph to get the counter values at each point. Subtracting the two counter values will give you the average growth per minute.
Performance counters that count objects are cardinal values, taken at the time the counter was captured. Examples include the number of file fragments, number of heap pages, and total cache size. For these counters, the rate of change of the counter is less important than the absolute value. If the absolute value jumps suddenly at some point during a trace, it indicates that something interesting happened to the server at that point.

The specific counters of interest for each category of performance problem are discussed in the individual descriptions in the Types of Performance Problems section.

**Manual Analysis with SQL Queries**

All of the analysis exposed by the Profiling mode in Sybase Central can be performed manually. There are a few cases where you may find it helpful to perform this analysis by querying the `sa_diagnostic_*` tables directly:

- If you have an extremely large trace to analyze (500,000 requests or more), you may find the GUI browser too slow to work with comfortably.
- If you are looking for a very specific circumstance (for example, a query issued from a specific user at a specific time that contained a specific variable value), you will need to query the tables directly.
- If you want to automate the process of application analysis (for example, to generate a custom report of expensive queries), you will need to generate such reports through SQL queries.

Additional information on the `sa_diagnostic_*` tables can be found in the Related Documents section. For the most part, the join between these tables should be self-explanatory.

Each tracing session is identified by a `logging_session_id`. This value is used to distinguish the session to which a given row belongs, and it is included in all diagnostic tracing tables. It is important to make sure that manual queries against each table are limited to a single `logging_session_id` column, either through joins or additional predicates, so that results from multiple sessions are not confused.

For example, to see the 5 most expensive individual SQL statements executed in a logging session with a `logging_session_id` of 2, issue the following query:
SELECT TOP 5 *
FROM sa_diagnostic_request R, sa_diagnostic_statement S
WHERE R.statement_id = S.statement_id
AND R.logging_session_id = S.logging_session_id
AND R.logging_session_id = 2
ORDER BY R.duration_ms DESC;

Analyzing Applications with Other Tools

If you are using SQL Anywhere version 9 or earlier, or for some other reason don't want to use the integrated Application Profiling tools, you can perform application performance analysis using the tools described in the Alternatives to Application Profiling section. Included here is brief description of how to use the data captured with these tools for performance analysis.

Request Logging

The request logging feature saves all statements received by the server from external clients in a flat text file. This file is human-readable, but procedures are also provided in the server to make it easier to analyze. The data captured by request logging is a subset of that captured by application profiling. The procedure `sa_get_request_profile()` takes the name of a request log file and parses it, populating the `satmp_request_profile` temporary table with the list of unique statements executed by the server and the total, average, and maximum execution times of each. This is similar to the Summary view in the Application Profiling analysis, except that statements are considered to be "similar" in this view only when they have identical SQL text.

Similarly, the procedure `sa_get_request_times()` takes the name of a request log file and parses it, populating the `satmp_request_time` temporary table with the statements and their execution times. This is analogous to the Detail view in the Application Profiling analysis.

In both cases, these procedures accept optional parameters that allow you to filter the results by connection (similar to the filtering pane accessible in the Application Profiling viewer in Sybase Central).

Procedure Profiling

Information recorded by the procedure profiling tool is best viewed from within Sybase Central. By switching to Profiling mode and opening a procedure, you will see the source code of the procedure annotated with extra columns, showing details about the number of times each statement was executed and the execution length. This view is similar to the information shown by traditional source-code profilers. Note that although the procedure profiling tool is viewed from within Profiling mode of Sybase Central, it is independent of with the Application Profiling features of the database server.

Tune and Test Your Application

Depending on the type of performance problem you are experiencing, the solutions for poorly performing applications may include:

- Changing the database configuration (command line options, physical file placement, etc.);
- Changing the database structure (adding new indexes, regenerating statistics, etc.);
- Adding more hardware;
- Recoding portions of your application.

The process of tuning and testing your application is likely to be iterative. It is best to proceed slowly when making changes to your database, especially for a production server. You want to avoid unintentionally destabilizing any other aspect of a production system by making configuration changes, and minimize unnecessary code changes or hardware purchases, which can be costly.

Types of Performance Problems

In this section, we consider some of the specific causes of performance bottlenecks. Using the method described in the Steps in Diagnosing Application Performance Problems section, you should identify which of the following categories your performance problem falls into, and then collect the data and perform the analysis described for that category.

I/O-Bound Applications

Database performance may be constrained simply by I/O bandwidth. This has historically been the major performance limitation on many database systems. You are more likely to see it if your database is much larger than your cache size (for example, in the tens or hundreds of gigabytes); however, I/O bandwidth can be a limiting factor even for much smaller databases.

If your database is I/O-bound, you will notice that the idle time on the disks storing your database files will be very low - less than a few percent, and perhaps as low as 0.0 percent. As an additional check, you should be able to see the disks working very hard if you go and physically observe the machine.

There are many reasons that your database application might be I/O-bound. Some of these reasons can be solved or worked-around simply by changing the configuration of your server. Often, though, if your application is constrained this way, you will have to add more disks. The good news is that this is one of the types of performance problem where simply adding more resources (more disks in this case) will give you an
increase in performance proportional to what you have added.

**Database Cache is Too Small**

Before going out and buying new disks, though, there are some things to check. It is possible that you have a database cache size that is set too small. This can lead to the phenomenon known as *thrashing*. The database server keeps whatever pages it thinks will be necessary to answer future requests in its cache (buffer pool), so that it doesn’t have to go read them from disk when requested. However, if there are too many pages that it thinks will need to be kept in cache to answer future requests, it will have to write some of them back out to disk (in order to stay within its resource limit), and then bring them back into cache at a later time. If the server is constantly moving the same pages back and forth between cache memory and disk, the server is said to be thrashing. It doesn’t sound good, and it isn’t | this can be a significant source of performance trouble.

You can get an idea whether thrashing is happening by looking at some of the counters that the server provides. You can look at these counters in real time in `perfmon`, or you can look at them as they are captured in an Application Profiling diagnostic trace. In particular, the counters that we’re interested in when we suspect that the cache is not large enough are the counters related to table pages and index pages. These are the two kinds of pages that will make up the vast bulk of the data that is used by your application. There are other page types that are used internally in the server for things like resource allocation, lock maintenance, and so on. Your data, however, is stored on a mix of table and index pages.

The statistics to look at are the `CacheReadTable` and `DiskReadTable` counters. The `CacheReadTable` counter will be incremented every time the server attempts to read a table page (whether from cache or disk). This could be because, for example, the server is trying to fetch a row, to update a row, or to index a row. The `DiskReadTable` will be incremented every time that the server has to go to disk to bring a table page into memory. These two numbers are both absolute counts since the database was started. To get an idea of the steady-state behavior of the server, pick two fixed points in time, roughly 60 seconds apart, and compare the increase in the `CacheReadTable` to the increase in the `DiskReadTable` counter. In an adequately-sized buffer, the `CacheReadTable` counter should be growing at least 10 times as fast as the `DiskReadTable`.

Note that the server must be at steady state (that is, it was not recently started and has been serving the same application for at least twenty minutes or so).

You can make the same comparison between the index leaf counters (`CacheReadIndLeaf` and `DiskReadIndLeaf`), and between the index internal page counters (`CacheReadIndInt` and `DiskReadIndInt`). For these counters, the `CacheReadTable` counter should grow even faster relative to the `DiskReadTable` counters than the `CacheReadTable` counters did. For the `IndInt` counters, there should be a difference of at least a factor of 100 in a properly sized cache at steady state.

In a properly configured buffer pool, the non-leaf pages of indexes should almost always be in cache.

In looking at these three sets of statistics, if you see that the disk reads are close to the cache reads on any of them (especially the index pages), and your server appears to be I/O-bound, it’s an indication that the server has a problem with thrashing, and you need to find a way to increase the server cache size. This may be as simple as increasing the maximum and minimum cache size parameters (`-ch` and `-ci`, respectively) on the database server command line. However, if there is not more physical ram available on the machine (either because it is used by other processes or because the database server is using the largest possible cache size) then the server is likely to benefit from the addition of physical RAM to the machine. For more information, see the Related Documents section.

**Missing Indexes**

An application that is driving the I/O subsystem to its limit may simply need more indexes in the database. Properly tuned indexes can greatly reduce the amount of I/O necessary for certain types of queries because they enable the server to be selective about which pages it reads from disks to answer queries. The downside of indexes is that each secondary index added to a table takes up more space (both disk space and cache space, when it's read into memory). Each secondary index also adds a maintenance penalty to all statements that update the data in the table.

To help you determine which indexes are relevant for your application, SQL Anywhere provides an Index Consultant (IC). The IC can analyze individual queries or a set of queries in an application profile, and recommend a set of indexes that it believes will increase the throughput of your application. If you are running the IC over a full application profile, it will take the maintenance penalty of data updates into account when making its recommendations. If you have an update-heavy workload, it will be more conservative in recommending indexes.

There are four ways to run the IC:

- **An entire workload can be analyzed from within the Profiling mode of Sybase Central. This happens automatically when using the Profiling Wizard, or can be started from the Profiling menu when browsing a saved trace.**
- **A single query can be analyzed from within Profiling mode: if you have identified a particular query or a small group of queries in an application trace whose performance you care about at all costs, you can select them individually in the Details view and choose to run the IC from the Profiling menu on these queries individually.**
- **The IC is embedded within `dbisql`: you can get a recommendation for any indexes that would help the single query that is active in the `dbisql` window; choose Index Consultant from the Tools menu.**
- **You can access the underlying functionality of the IC manually to perform what-if analysis on the possible creation of indexes. The CREATE VIRTUAL INDEX statement is a variant of the CREATE INDEX statement that allows you to see whether the optimizer thinks it would be able to make use of one or more of the indexes you suggest with that statement if those indexes were to exist physically in the database.**

Related Documents section for more information.

Keep in mind when manually tuning indexes that the penalties associated with queries that update the database are not automatically taken into account.
Memory Available for Query Processing

A third possible cause of excessive disk I/O is that the query processing engine does not have enough memory to process the queries submitted to the server using cheap, in-memory strategies. As a result, it has to revert to low-memory strategies, which are designed to minimize the in-memory footprint of a cursor at the expense of increased I/O requirements; or, it means that the in-memory algorithms have to spill their memory contents to disk temporarily during query processing and read them back in later, to avoid occupying too much of the buffer pool. In SQL Anywhere version 10, the server determines the amount of memory available to a given connection for processing queries by considering how much of the entire cache is available for query memory, and then dividing that by the server's multiprogramming level, which is configured by the --gn command line option.

If you have a very high --gn (where high is anything above 50), then make sure that you do in fact expect your server to have to concurrently process a large number of simultaneous requests. The server can have more active connections than the number specified by the --gn option; the --gn option merely indicates how many requests the server will process concurrently.

Suboptimal File Placement

Another possible reason for poor I/O performance characteristics is that the placement of the database files used by your database application is suboptimal. For instance, it is best practice to have the transaction log on a separate drive from the main database files. In a very high I/O intensive application, or for applications that have very large databases, it's useful to push the data into a secondary dbspace, and to keep the system dbspace (the .db file) relatively free of application data, because this is where the checkpoint log is stored. Both the transaction log and the checkpoint log see a lot of disk activity whenever transactions that insert, update, or delete data are being processed by the server. For the best performance characteristics, if at all possible, move these files to a separate drive. The temporary file should be on its own drive if at all possible. In every case, you especially want to avoid putting the temporary dbspace or the transaction log on a RAID 1 device. If you need the peace of mind that a redundant RAID configuration provides, a small RAID 5 drive is recommended for the transaction log file.

CPU-Bound Applications

If your application is CPU-bound on the database server, the good news is that you'll be able to tell this right away. Simply opening taskman or perfmon in Windows, or their equivalents in other operating systems, will show you that the database server is using all of the CPUs that have been allotted to it, and any value above 90% of the CPUs that are allotted to a server indicates the possibility of CPU-boundedness in an application. In a truly cpu-bound application, you'll see the server using 99 or 100% of the cpus allotted to it.

The bad news in a CPU-bound application is that there are many causes for this behavior, and investigating and identifying the specific causes of your cpu-bound application will take some time and effort. We will consider three broad categories of tunable, fixable problems that can cause CPU-boundedness, and they are related to each other along a continuum:

- Suboptimal plans | the query optimizer has chosen a plan that is much more expensive than the best possible plan for a given query;
- Suboptimal queries | the query itself has been composed in a way that makes it very difficult for the server to execute it in an efficient way;
- Suboptimal query patterns | each individual query that is part of the database application may be fine in isolation, but the queries may be issued in a pattern that, as a whole, is suboptimal for the database server.

Suboptimal Query Plans

Data Distribution Statistics

Suboptimal query plans are generated for queries that the optimizer has difficulty optimizing. The optimization process is a complicated phase of query processing because the range of possible access plans, especially for non-trivial queries, is enormous. For a query of medium complexity, there are potentially tens of thousands of query plans that could be used. For very complex queries involving dozens of tables, there may be hundreds of thousands of possible plans or more. As a result, the query optimizer has to decide which plan among all of these possibilities is going to provide acceptable performance characteristics. The query optimizer has to rely on estimates of the cost of each portion of a query plan in order to make its decisions. These estimates are based on the statistics that SQL Anywhere automatically collects about the data stored in its tables. Over time, especially in a database application that sees a lot of churn (large quantities of data being deleted and new data being inserted to replace it), these statistics can become outdated. This is one of the main causes for suboptimal query plan selection by the optimizer.

Optimization Goal Option

Another possible reason that the optimizer selects a suboptimal plan is that the optimization goal option is improperly set. For most queries, the All-rows setting (which is the default in version 8.0.2 and later) is the one that makes sense. This means that the server may not return the rst row immediately, but it will return all the rows in the shortest possible time (according to its estimates). However, it is sometimes the case that your application is interested in getting the rst row, or the first few rows, of a query as soon as possible, so that it can start processing, and you're less concerned about time required to fetch the entire result set. If you have ever set this option, it can make sense to double-check that setting for each of the queries that you have identified as slow.

Analyzing Sub-Optimal Plans

To detect a suboptimal query plan, you want to find the expensive plans and look at the graphical plans that were captured. If the graphical plans have been captured with statistics, they will show both the estimates that the optimizer expected for the number of rows and the cost of executing each part of the plan, as well as the actual number of rows fetched and the actual time spent fetching these rows. If you see large variations between the estimate and actual values, it's an indication that the statistics that the optimizer has relied on to make its estimates may be out of date. The graphical query plans with statistics will highlight operators where there is a substantial difference between the estimated and the actual
values; when you open a graphical query plan in the profiling analyzer, you should be able to see whether there are any significant anomalies, just by looking for the red boxes. When looking at the query plans, you also want to look for the tables that supply the bulk of the rows, because any inaccurate estimates on these tables will have significant effects on the plan chosen, and on the run time of that plan. You can find which are the most expensive and heavily-weighted portions of a query plan by looking for thick black boxes and thick lines connecting operators. Any table that has a thick line on it is a table that is important for that query; it’s a table where you’ll want to double-check that the estimates are in line with the actual values in the table. You’ll want to do this analysis for queries that have been identified as expensive.

It’s very rare that the optimizer is going to accurately predict the exact run time of an operator, or the entire query. It’s often sufficient for the optimizer to estimate costs within a factor of 3 (5 times the actual run time because the optimizer only cares about the relative cost of different plans. The optimizer wants to be able to compare plans so that it can pick the cheapest one but it doesn’t care about the absolute cost of the plan. However, if you see that the actual and estimated cost of a plan differs by a factor of well over 20 times on a relatively simple query, or if the estimated number of rows differs by a factor of 10 times on a relatively simple query, then it’s an indication that there may be an underlying problem with statistics. For more complex queries, especially those involving subqueries or UDFs, the difference must be several orders of magnitude before it definitively indicates a problem.

Suboptimal Queries

The optimizer may do the best that it can and rely on very accurate statistics, but queries are sometimes written in a way that makes it very hard for the optimizer to build a good access plan. An example of this would be a query that asks for unnecessary columns. For instance, consider an application that wanted to retrieve a list of orders along with the customer’s name:

```
SELECT *
FROM orders, customer
WHERE orders.o_customer_id = customer.id;
```

The application only cares about the columns from the `ORDERS` table (plus the customer name id), but it has asked the server to compute the join for all columns from both tables. In this case, especially if the `CUSTOMER` table is very wide, the server has to do a lot of extra work moving those columns around. It also uses extra memory to store those extra columns in any intermediate results. The client, too, needs extra space and cpu cycles to fetch and store these results.

In this example, if the application is not actually using these extra columns, it would be better to write the query as:

```
SELECT orders.*, customer.name
FROM orders, customer
WHERE orders.o_customer_id = customer.id;
```

It is time-consuming to detect unnecessary columns within a result set, because you have to examine both the expensive queries and the application code that calls them, but it can pay off to look for such cases among the most expensive queries. If you do identify unbound/unused columns in your application, the optimizer may be able to select a more efficient access plan. But, because of the time-consuming nature of this analysis, it only makes sense to do it for the most expensive queries that show up in the Application Profiling trace.

User-Defined Functions

Another type of query that is difficult for the database server to process efficiently is one that makes many calls to user-defined functions (UDFs). UDFs can be very powerful and can allow you to push a lot of logic into the query, but because they are not native server code and instead are written in interpreted SQL (or another stored-procedure language supported by the server), there is a performance penalty associated with their execution. In particular, it is wise to avoid a UDF call on every row of a result set if that result set contains thousands of rows or more. UDFs are most efficiently used when they are called a rite and well-regulated number of times. This is not an absolute rule, and you can successfully use UDFs on large result sets; however, if the most expensive queries in your application contain UDFs, you may want to consider rewriting the queries in a way that does not rely on that construct. This may involve rewriting the query with additional joins.

There are many other possibilities, beyond the above examples, of suboptimal queries. If you have identified a set of queries that appear to be very slow, but you can’t understand why, consider whether it is possible to experiment with the query - a semantically equivalent rewrite may result in significant performance improvements for complex queries.

Suboptimal Query Patterns

As noted above, it may be that individual queries are very fast in their own right, but they still appear among the most expensive requests at the top of the Profiling mode summary view because the queries are executed many times. Such queries may be part of a suboptimal query pattern within the application. This happens when you issue the same basic query many times, with slight differences in the parameters that are passed. For example, you may issue a sequence of statements like the following:

```
SELECT * FROM orders WHERE o_custname = 'Alice';
SELECT * FROM orders WHERE o_custname = 'Bob';
...
```

It is almost certainly more efficient to issue this pattern of queries as a single query. If the list of customers is only available in the application, it could be written as a single statement like this:

```
SELECT * FROM orders
WHERE o_custname IN ('Alice', 'Bob', ...);
```

If the list of customers was stored in the database, the query could be written as a join:
SELECT orders.* FROM orders, customer
WHERE order.o_custname = customer.name;

Because a single query doesn't involve more than one round trip, and because it doesn't involve parsing and reopening a cursor for each individual customer, the server has the ability to deliver the required results more quickly. Unfortunately, this strategy requires changing your application code, but it may result in a significant performance boost. This will be especially true if there is a network connection between the portion of the application that is issuing the queries and the server.

A variant of this kind of suboptimal pattern that is easier to fix is a query that fetches the same data repeatedly. If it is convenient and possible to cache data in your client, rather than executing the same query over and over again, you can reduce the overall latency and execution time of your application by avoiding the unnecessary work of repeatedly executing and fetching the same data. The Summary pane in the Profiling mode is the place where you will find indications of poorly performing query patterns.

Concurrency-Bound Applications

If your application is concurrency-bound, it is in fact still resource-bound, but the problem is that the resources that the database server is able to use are restricted by other considerations. What this means is that the server is not able to take advantage of additional CPUs or additional disks. If the application is concurrency-bound, it is likely still I/O-bound or CPU-bound, but the problem is one of scalability, and not one of absolute boundedness.

A concurrency-bound application is one in which the server is waiting to process queries, but it can't. The easiest way to determine that the application is concurrency-bound within the server is to look at the ActiveReq performance counter. This can be seen in many places, including:

- viewing the counter exposed for the SQL Anywhere 10 Server object in perfmon;
- querying the property manually:
  ```sql
  SELECT PROPERTY('ActiveReq');
  ```
  or,
- by looking at the performance counter that was captured in the Application Profiling trace (see the Statistics pane).

Determining the Type of Contention

If the number of active requests is high (at least 10-20, and perhaps much higher) but the server does not appear to be using substantial CPU or I/O resources, it's a good indication that your application is concurrency-bound. A concurrency-bound application may be server-related or it may be application-related.

You can tell whether contention problems are server- or application-based by:

- looking at dbconsole. If connections are reported as blocked on other connections, the contention is application (row-lock) related; if the connections are not reported as blocked on other connections, internal server contention is a possibility.
- looking at the blocking events tab when examining an Application Profiling trace in Sybase Central (if you have a very large trace). If the number of blocking events is a significant fraction (20-30% or higher) of the number of database requests, there is likely an application-based concurrency bottleneck. You can also view this information by querying the sa_diagnostic_blocking table:
  ```sql
  SELECT COUNT(*) FROM sa_diagnostic_blocking WHERE logging session id = 1;
  ```
  compared to
  ```sql
  SELECT COUNT(*) FROM sa_diagnostic_request WHERE logging session id = 1;
  ```

Server-Based Concurrency Limitations

Server-related concurrency bottlenecks happen because the server maintains internal locks to protect its data structure and to throttle access to the machine resources it uses. For example, only one transaction can write to a given portion of the transaction log, and so the server has to restrict the reservation process for portions of the transaction log. In prior versions of SQL Anywhere, internal server contention was a significant source of server performance issues on highly parallel machines. For example, on an eight-way machine, version 8 experienced significant contraints on its performance. Improvements were made to versions 9 and 10 that reduced the number of contention points in the server.

There are very few possible points of contention in the version 10.0.1 server. In a version 10.0.1 server, the most likely points of server contention are:

- the transaction log;
- the checkpoint log; and,
- the lock table.

These types of server contention are usually possible only when the number of simultaneous client connections is well above 50.

Transaction Log Contention

Transaction log contention is a possibility if you have large numbers of connections doing insert, update, or delete transactions, where these transactions are very small. For example, if you have hundreds of connections, each of which repeatedly inserts one row and then commits, then
transaction log contention is a possibility. You'll see this as high disk activity on the disk drive containing the transaction log, even though the rest of the drives may be quite idle. If you suspect this is what's happening, the first solution is to simply pregrow the transaction log. You can do that by issuing the following statement (choosing a size that is your circumstance, but probably at least 200 mb for a high-volume application):

```
ALTER DBSPACE TRANSACTION LOG ADD 200MB;
```

Additionally, if you suspect this is a problem, it will be very helpful to move your transaction log to its own disk as discussed in the comments on I/O-bound servers; see the Suboptimal File Placement section.

**Checkpoint Log Contention**

If your application is doing updates or deletes selectively and sparsely, you may experience checkpoint log contention. The server has to store a clean copy (preimage) for any page that it modifies between checkpoints. If your application has many connections that are updating small amounts of data on widely scattered pages, then the number of pages that become dirty rises very quickly, and as a result, the number of clean preimages that have to be stored in the checkpoint log also rises very quickly. This can lead to contention on the checkpoint log. If you suspect that this is what is happening because it matches the data update patterns of your application, your options in version 10.0.1 and previous are limited. The best thing to do is to try and move your data into a secondary dbspace and ensure that the system dbspace (the main .db file) is on a fast drive by itself.

**Lock Table Contention**

If neither of the above cases seems to fit your application, but the server appears to be CPU-bound on a single CPU (or perhaps two CPUs), then there is a possibility that the lock table is a source of contention. Again, there is not much that can be done in this case in version 10.0.1 or earlier. However, this is only likely to be visible under extremely high database loads that are not constrained by I/O - typically, only on a server hosting thousands of connections, where the database size is very small relative to the cache memory.

**Insufficient I/O Concurrency**

Another form of concurrency bottleneck is that the server multiprogramming level (-gn option) is set too low. The circumstance here is that the database file is on a fast RAID 0 disk with many spindles that can retrieve random data very quickly. In this case, the server is driving the disk as hard as it can, but it could be that the disk harder if it was processing more concurrent connections. You'll be able to tell if this is the case by looking at the Current Disk Queue Length in perfmon. For a large RAID disk, a fully-loaded disk may have a counter of 50 or higher. If this counter is below 10-20, it's an indication that the disk could be driven harder, but the server is not issuing I/O requests fast enough to do this.

The solution here is to increase the multiprogramming level in your server. Try increasing the -gn option (which defaults to 20) in increments of 30% and observe the effect on database throughput and disk queue length. If I/O concurrency was the original problem, you will see the disk queue length increase and the disk idle time decrease on the drive after making this change. Again, this type of bottleneck is only possible on heavily loaded servers with large, parallel I/O capabilities. You cannot be too aggressive with the multiprogramming setting or you risk running into problems with query execution as described in the Memory Available for Query Processing section.

**Application-Based Concurrency Limitations**

The basic reason for an application-based concurrency problem is that connections hold locks on rows when other connections want them. This is necessary to ensure consistent and predictable behavior for your application. However, tradeoffs may have to be made between ease of programming and performance.

**Hot Rows**

One common case where this arises is the hot row problem. This is the case where many connections want to update a single row (or depending on the isolation level, it may be that a few connections update the row, but many connections want to read the row). A hot row often appears when one row is used to store a value for a new primary key. There are more effective ways to do this - for example, using global autoincrement.

**Long Term Holding of Read Locks**

It may also be that although there's no single row (or small set of rows) that are eagerly sought by all connections, that individual connections hold read locks for a very long time on rows. See the Related Documents section for more details on locking. The problem is that read locks are often held for the duration of a transaction. Read locks may be held for a long time at isolation level 1 or higher if a cursor is sitting on a row and has been held open for a long time. If you're running connections at isolation level 1 or higher, you should make sure that if you keep a cursor open, there is a good reason for doing so. As long as a cursor at isolation level 1 or higher is opened and is sitting on a row, it is preventing other connections from updating that row. At isolation level 2 or higher, it is even more restrictive, since locks are held on all rows the cursor has touched until the cursor is closed.

At isolation level 3, a lock is acquired for every row your transaction visits, and it is held until the end of the transaction. If you must run a transaction at isolation level 3, try to do everything possible to keep the transaction short and to visit the minimum number of rows possible. You should avoid doing queries at isolation level 3 that scan an entire table; if the query must search for specific rows, create an index to facilitate this process. Sequential scans of large or important tables at isolation level 3 have very serious concurrency implications if you also need to update such tables.

You can see the length of time a cursor stays open, either in the Details tab in the Sybase Central Profiling mode, or by querying the sa_diagnostic_cursor table for cursor open and close times. You can see the isolation level used for a cursor in the SQL Statement Details wind
ow (viewable by right-clicking a query in the Details view), or by querying the sa_diagnostic_cursor table. If you are not using Application Profiling, you must refer to your application or to your database option settings to see what isolation level your cursors are configured to use.

**Long Term Holding of Write Locks**

Your application may be such that it contains large write transactions that take a long time. Those transactions are holding locks on all of their underlying rows for the duration of the transaction. If it is necessary for you to have long-running transactions, you should do what you can to ensure that the rows that are held open by a transaction are not likely to be sought, either for update or for read at isolation level 1 or higher, by other connections.

Multiple write transactions that update the same rows can lead to concurrency problems, but these problems will be most serious when a mix of write transactions and high-isolation read transactions contend for the same rows.

**Snapshot Isolation**

Snapshot isolation is an alternative to the traditional isolation levels for providing consistent and predictable semantics for concurrent transactions. A transaction running at one of the snapshot isolation levels (there are actually several snapshot levels that have subtle behavior differences) sees the entire database as it was at the time the snapshot transaction started. This is done by keeping copies (preimages) of all rows that change after the transaction starts. For rows that are not updated during the transaction, no extra work needs to be done. Thus, from the application's point of view, this is like opening a transaction at isolation level 3, visiting all rows that will ever be used by the transaction, and then starting the transaction proper.

The snapshot isolation levels provide highly specified semantics for transactions and have the potential to support a high concurrency level for a mix of read and write transactions, largely because readers do not block writers. Depending on the exact snapshot level chosen, conflict detection for writers may also be deferred.

However, there are tradeoffs to this approach. Long-running transactions will not prevent other transactions from running, but will require all other transactions to make backup copies of rows they touch. In an application where transactions update large numbers of rows (even if these rows are completely disjoint between transactions), a large amount of additional work will have to be done to make these copies. This overhead is incurred even if there are only a few snapshot transactions active at a time.

Snapshot isolation will improve the throughput of applications that need the predictable semantics of a consistent view of the data, provided that the application is primarily read-only (that is, the total volume of rows inserted/updated/deleted is much smaller than the total number of rows read). Although there won't be a concurrency penalty for write-heavy workloads, there will be extra CPU (and potentially disk I/O) costs.

For more information, see the Related Documents section.

**Performance Impact of Deadlocks**

You are also more likely to see deadlocks in an application that has concurrency limitations. Deadlocks are an inconvenience because you have to write error-handling code that either gives up on a transaction that has deadlocked, or retries the transaction at a later time. Deadlocks can also represent a source of performance difficulties: a lot of work may have been done for a transaction, all of which has to be rolled back and reapplied later if that transaction is selected as a victim of a deadlock cycle.

**Application Design**

Unfortunately, if application bottlenecks are revealed by an analysis of the application trace, the server is limited in what it is able to do. The server is likely resource-bound in this case, either on the CPU or the disk used by a single connection, but it's not able to take advantage of any additional CPUs or additional disks that you add to it. Application redesign is often necessary in this circumstance. This is why it's important to think about concurrency and scalability from the earliest stages of application design: database applications that have high lock contention will frequently work well on small test instances, but become unworkable once the workload grows.

**Client-Tier Performance Issues**

If none of the circumstances we've considered appear to explain the performance problems in your application, it's likely that the primary problem lies within the client portion of your application and is not connected to the database server at all. You can verify this by looking at the **ActiveReqPerf** counter, either in perfmon or in a saved application profile, and also look at the resource consumption on the server. The CPU and the disk I/O will be low, and the **ActiveReq** counter will also be low (probably well below 20). If this is the case, the server is simply not receiving enough work to keep it busy. The fix is to examine your application code, perhaps with the aid of a traditional source-code profiler, to see why the queries are not being issued to the server fast enough. This is perhaps most likely in a multi-tiered application where the client tier(s) are doing significant processing - that would be the place to look at improving performance.

**Summary**

Investigating database application performance problems can be a time-consuming process. However, the tools and techniques discussed in this paper can make it relatively easy to identify the causes of performance problems. We have discussed the benefits of beginning with a broad characterization of application performance, and then using the profiling tools included in SQL Anywhere to identify the specific type of
performance issue you are seeing. The Application Profiling tools included in version 10 and later provide the most comprehensive and integrated set of features to date to assist in this process, but the tools shipped in earlier versions (see the Analyzing Applications with Other Tools section) can also be used.

Unfortunately, there is often little correlation between the amount of time and effort needed to identify the cause of a performance problem, and the time and effort needed to resolve it. Some problems can be resolved by simply changing isolated configuration options, or by adding additional hardware. However, some problems (especially those discussed in the Concurrency-Bound Applications section) may require substantial changes to the database application to perform acceptably on larger workloads.

In the Types of Performance Problems section, we have discussed the broad categories of database application performance problems, and the most common individual types of problems within these categories. For each, we have suggested the approach most likely to resolve the problem. The recommendations made in this paper are, in the experience of the SQL Anywhere development team, the ones that have the broadest applicability. However, a large database application is a complex piece of software, and no finite set of recommendations will be sufficient to resolve every possible performance problem. By iteratively applying this process, you will be better able to accurately characterize any performance problems you are experiencing, allowing recommendations from outside sources to be tailored to your precise situation.

**Related Content**

**Related Documents**

Diagnostic tracing configuration
ATTACH TRACING statement
Request logging
Diagnostic table tracing
Tip: Use the cche to improve performance
CREATE INDEX statement
How locking works
Snapshot isolation

**Related SAP Notes/KBAs**