Database Performance Tuning Methods
The DBA's Perspective

Vladimir Andreev
Semantec GmbH
Benzstr. 32, 71083 Herrenberg

Summary

As a long-time database developer and DBA, I have come to understand that database performance tuning approaches and methods available to the DBA are substantially different from those available to the developer. For example, while developers and designers operate mostly in test environments and have full access to the application's source code, DBAs usually deal with a "frozen" application in a production environment. Thus, developers typically apply proactive tuning, while DBAs mostly do reactive tuning.

In reactive tuning mode it is crucially important to quickly organise the work and find a solution for the performance problem. In order to achieve this, I usually apply a response time profiling method, which allows me to quickly find the location of the problem from an architectural perspective, and then use "classical" response time analysis [2] to find the causes of the problem, if its location happens to be within the Oracle Database.

Using these simple methods, I have been able to identify causes and, on many occasions, to propose solutions for acute performance problems in a matter of minutes.

Developers and DBAs

The performance of a database application is in the hands of these two distinct groups of individuals. Each of them operates in a different environment and under different constraints. Following are the differences that I find most important from a performance tuning perspective. The list is based on my experience only and does not attempt to be exhaustive or to take a representative number of companies into consideration. In other words, things may look very different in your shop.

Developers:

- Determine and can change the design and implementation of the application
- Operate in a sandbox where no real damage can be done to the business of the end users. Experiments are allowed and encouraged in the sandbox, but the building material is, well, sand: data volumes are scaled down, application usage patterns do not even remotely resemble real usage, the hardware is less powerful (best case) and/or completely different (worst case) from the production machines, the system architecture is often emulated or even simulated using stubs, and so on.
- Work based on a predefined schedule

DBAs:

- Cannot see the application's source code or SQL. It is usually hidden in compiled executables or wrapped program units.[1]
- Are primarily responsible for the availability of the application; its performance comes at most second, usually behind a considerable number of other tasks. In everyday priorities, that is.
- Deal with real production systems, changes to which are heavily regulated and strictly controlled. This means that DBAs typically have very little means for influencing the application's SQL, and absolutely no way to change the application's design or architecture.
- Sit around all day playing computer games while waiting for a disaster to strike. The good DBAs, at any rate.

Tuning Modes

These differences determine the most common modes in which developers and DBAs tune. Developers generally apply *proactive* tuning, whose goal is to prevent performance and scalability problems from occurring by designing and implementing the whole system with performance in mind. It also includes adequately sizing the hardware, adjusting business processes, and much more. Ideally, proactive tuning can eliminate the need for *reactive* tuning, but is much more difficult to apply in full.

Proactive tuning is also a planned activity, which reduces the time pressure on the developers executing it.

Reactive tuning is the most common mode in which DBAs operate. A reactive tuning effort starts when a performance problem is detected, so it cannot be planned in advance. It ends when the problem is resolved. After the (successful) end of a reactive tuning session, a proactive tuning effort should be executed in order to make sure that the same or similar problem would not occur again.

In mission-critical systems, bad performance cost may get very high very fast, which translates directly into urgency and pressure applied to the tuner.

[1] Of course, DBAs **can** actually see the SQL in the library cache (via v$sqlarea). What they cannot see is the application code that generates this SQL, or the context in which the SQL is used. They are usually forced to reverse-engineer the program logic from trace files or execution statistics. (Thanks to Craig Shallahamer for pointing out this distinction to me.)

The following table summarises the characteristics of proactive and reactive tuning modes:

Proactive	**Reactive**
Is planned	Cannot be planned
Low time pressure	Extremely high time pressure
No constraints on the methods and tools used	Analysis and tuning measures constrained by dealing with a packaged application and live production data
No specific scope and target	Scope usually limited to the problem at hand; target loosely defined, maybe, but always present

As explained above, reactive tuning is almost always associated with high time pressure. It is therefore very important that the tuner has a clear and simple plan (because complex plans tend to be forgotten under stress) on how to address the problem. In the following sections I will make an attempt at outlining the major steps in the plan I use.

Reproduce the problem

The first thing to do is to get a first-hand description of the problem, and reproduce it. This may involve interviewing the users who are observing the problem. It is important to keep in mind at this point that the users can be trusted to describe the effects of the problem, but **not** the reasons for it, or possible solutions, even if the users are themselves developers, and as such, appear to be better positioned to propose solutions. One example:

An application developer calls the DBA saying that a FETCH they are issuing hangs because of locking. They ask the DBA to find the session that is holding the lock and kill it.
The effect is clear and valid – a FETCH call doesn't return for a long time. The reason for it is not. However, it is quite easy for the DBA to trust the developer for the reason, because they know the application logic, and the DBA doesn't. The DBA can then spend valuable time feverishly chasing a lock that isn't there before it dawns on them that a FETCH can't lock anything, and the reason must be something else. In their desperate quest for a quick solution, some DBAs in similar situation might even decide to bounce the database to make sure the phantom session gets killed.

The best way, of course, is for the DBA to be able to reproduce the problem. This might not be easy to achieve, however, even if the actions leading to the problem are clear. It might be a problem in a batch program that runs for 20 hours instead of the usual 5, or it might happen only when certain irreversible actions are carried on in the application, such as transferring money from one account to another, or it might require all other users to stop working with the system. In such cases the best solution would be to reproduce the problem on an adequate test instance, perhaps scaled down so that the "problem" batch job runs for 20 minutes instead of hours. Such downscaling might be dangerous, though, because if a statement poses a performance problem during production use, this usually means the execution plan is not scalable, and while performing acceptably on 1000 rows, it can hang on 10000. For these reasons, it is necessary to validate the tuning results on the production system before declaring the job to be done.

Define the target

The most frequent performance complaint is "It's too slow". While it may be true, it doesn't give the tuner an "exit criterion". It is important to define when to stop tuning, so that no time and effort is used to achieve marginal improvement. The tuning target should be defined in measurable terms, e.g., response time in seconds, total elapsed time, total or peak memory usage in bytes, total CPU time, etc.

Keep Baselines

No tuning effort should make the performance **worse** than it was before, so you need to be able to quickly undo all the tuning you have applied to the database, and achieve the same (poor) performance as before the effort started.

How you achieve this depends pretty much on the situation, but it is worth the effort. Keeping initialisation files and application source code under change control might help. Keeping notes on everything you change might do the rest. Maintaining an undo script whenever you modify a database object will probably prove extremely beneficial. Sometimes reverting the database to its initial state using only scripts is not possible at all – for example, if you rebuild an index, there is no way save point-in-time recovery to bring it back to its fragmented state.

Find the bottleneck component

Often it is not very clear which system component is responsible for a performance problem. End users and their managers are often very quick to blame the database for any performance mishap, but more often than not, the problem lies elsewhere.

Only when it occurs in a batch job executed completely within the database (i.e., via DBMS_JOB), a slowdown may be attributed entirely to the database component without any analysis. In the more common situation of an n-tier (n>1) system, there are many other suspects, namely each of the tiers, and the network between the tiers. The tuner must keep in mind that even when executing statements using SQL*Plus running on the same machine as the database, they still have 2 tiers – the server and the sqlplus client – and possibly a (loop back) network connection between them. Example:

> For reporting purposes, a SQL*Plus script is executed on the server from time to time to generate a detailed text-based report. The script takes too long to complete. The cause might lay in the performance of the queries executed in the database, or it might lay in the size of the SQL*Plus array buffer.
> *If a query returns lots of long rows, it puts a high load on the network and on the client processing capabilities. Concentrating on tuning the execution of the queries might not yield the desired results in this case.*

A common error when testing the execution time of individual SQL statements is to take the statement out of the application and execute it in SQL*Plus with timing switched on (*set timing on*). The time reported by SQL*Plus in this case includes:

- Statement transfer time. This is the time needed to transfer the text of the statement from the client to the server. Happens over the network.
- Statement parse time. Happens in the server.
- Statement execution time. Happens in the server.
- Results transfer time. Happens over the network.
- Results rendering time (alignment, formatting, adding column headers, etc.). Happens in the client.
- Results display time. Happens in the client OS/hardware.

When the statement or its result set are very large, or the network is slow, the transfer times might become significant and skew the results of the test. Instead of executing the (select) statement itself, it is better to wrap it in an anonymous PL/SQL block along the lines of:

```
Declare i integer:=0; start_time date:=sysdate;
Begin
        Dbms_output.Enable(50000);
        For x in (select * from problem_statement) loop
                i:=i+1;
        End loop;
        Dbms_output.Put_Line('Count: '||i);
        Dbms_output.Put_Line('Elapsed: '||to_char(trunc(sysdate)+
                (sysdate-start_time), 'HH24:MI:SS');
End;
/
```

When more tiers are involved (e.g., a web server, or a distributed database), things get even more complicated. Different tiers require different tuning approaches; so demonstrating that the bottleneck is **not** in the database server might be all that the DBA needs to do.

It is therefore vitally important to determine the system component containing the bottleneck before making any attempt at fixing it. It might be tricky to achieve this in all environments, or to devise a set of guidelines applicable in all situations.

Tier Walk

One way to identify the tier containing the bottleneck is to measure the response time at every tier of an n-tier application, starting with the user perspective.
Here is what I usually do when dealing with a web application:
I first click the link in the client browser and measure the response time, then connect with SQL*Plus from the web server to the database server and measure the execution time of the database call(s) resulting from the user click. Then I log on to the database server machine and measure the execution time of the same database calls from SQL*Plus.

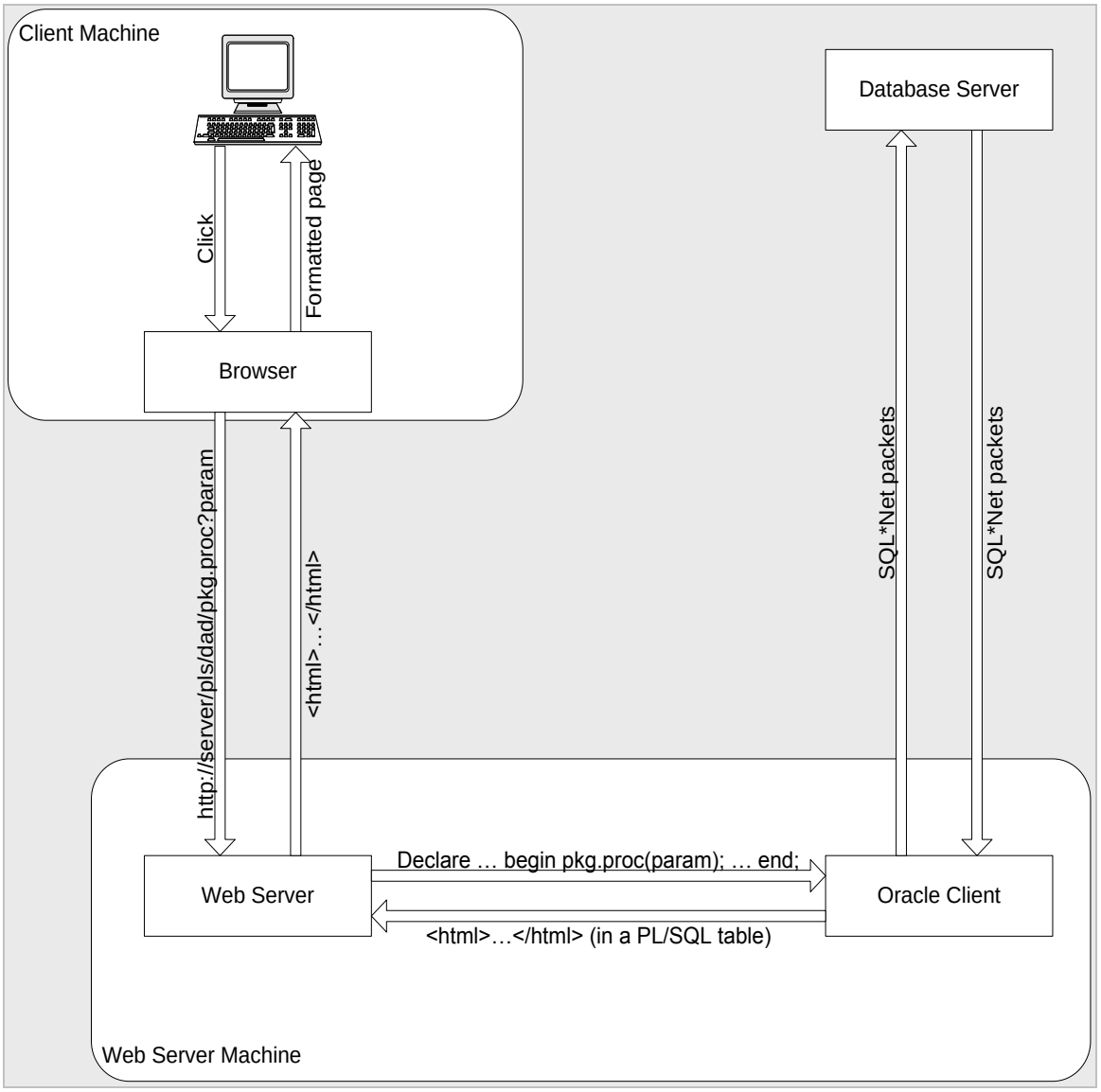

Figure 1: Communication between tiers in a Web application

If you find out that the database responds in only a small fraction of the overall application response time, then the problem is clearly in some of the other tiers or the connections between them. For example, if the user's web page gets displayed in 45sec, yet a simple HTTP client can retrieve it in only 5 seconds on the same machine, then the bottleneck occurs in the web browser, maybe because the HTML is too complex. An additional indication in that direction might be a high CPU usage on the side of the browser process combined with low network activity during the processing.

This method has the advantage of identifying the tier responsible for the delay. It might be hard to apply though, because it is not always trivial to translate user actions to tier-specific calls on every step.

Inside Out

Think about the simple client-server architecture. The server's role in it is to sit idle and wait for requests from the client. A request comes; the server processes it, sends the response back to the client and is idle once again. Much like its DBA.

Let us – for simplicity – eliminate end user interaction by considering a batch job that runs on the client and communicates with the server. In this case "server idle time" actually means "client processing time", and "server processing time" means "client idle time", that is, the roles of the "client" and the "server" are symmetrical – we can say that the batch job is the server for the corresponding server session! Indeed, when the server process sends its response to the batch job, it is actually *requesting* it to do its part of the work; when the batch job is ready with its processing, it *responds* to the server process by sending another request.

Now, let us see what this change of perspective can give us: It allows us to measure the client's performance from the server! So, it turns out that the Oracle database server is fully equipped to measure not only its own performance, but also the performance of its clients.

I believe two clarifications are necessary at this point. First, people report response time problems when their application locks them out, so it is always valid to exclude user interaction when solving a response time problem. Second, the client response time measured from the server necessarily includes processing time at all tiers as well as transfer time.

These considerations show that it is enough to perform response time analysis on the server session in order to prove that the bottleneck is or is not in the database: If client processing (= server idle) time constitutes the majority of the elapsed time for the session, then the bottleneck is **not** in the database.

Session Level Response Time Analysis

Response Time Analysis is a powerful extension of the wait event-based bottleneck identification method. Craig Shallahamer discusses both in great detail in excellent papers [1][2] available on his company's web site http://www.orapub.com.

Here is only a brief discussion on how I apply that method for bottleneck identification:

Response time consists of two components: service time and wait time. Service time translates to CPU time used by the Oracle server process serving our session (the "CPU used by this session" statistic in V$Sesstat, in centiseconds), and wait time is everything else. Wait time may be further divided into idle and non-idle. Again, all the time the server process spends waiting for a request from the client application (shown in V$Session_Event for the "SQL*Net message from client" event, in centiseconds, plus any time recorded in V$Session_Wait for the same event if the process is currently waiting on that event) is idle. Non-idle wait time comprises of waits for disk operations to complete, latches to be acquired, etc. Analysing this component might be very

helpful when tuning, but at this point you should know enough in order to determine whether the bottleneck is in the database, or not.

Here is a sample statement that may be used to get the time breakdown. It is by far not perfect, but illustrates the basic idea:

```
select  s.sid,
                to_char(
                        trunc(sysdate)
                        + ss.value / 100 / 3600 / 24,
                        'HH24:MI:SS') cpu_time,
                to_char(
                        trunc(sysdate)
                        + nvl(se.waited, 0) / 100 / 3600 / 24,
                        'HH24:MI:SS') non_idle_waits,
                to_char(
                        trunc(sysdate)
                        + nvl(si.idle, 0) / 100 / 3600 / 24,
                        'HH24:MI:SS') idle_waits,
                to_char(trunc(sysdate) + (sysdate - s.logon_time), 'HH24:MI:SS')
elapsed_time
        from v$sesstat ss,
                v$session s,
                (select   sid, sum(time_waited) waited
                         from (select sid, event, time_waited
                                        from v$session_event
                                      union all
                                      select sid, event,
                                                seconds_in_wait * 100
                                        from v$session_wait
                                       where state = 'WAITING')
                               where event not in ('SQL*Net message from client', 'pipe
get')
                 group by sid) se,
                (select   sid, sum(time_waited) idle
                         from (select sid, event, time_waited
                                        from v$session_event
                                      union all
                                      select sid, event,
                                                seconds_in_wait * 100
                                        from v$session_wait
                                       where state = 'WAITING')
                               where event in ('SQL*Net message from client', 'pipe get')
                 group by sid) si
   where ss.statistic# = 12
         and s.sid = ss.sid
         and s.sid = se.sid(+)
         and s.sid = si.sid(+)
         and s.username is not null
group by s.sid, ss.value, s.logon_time, se.waited, si.idle;
```

If the biggest component of the database response time is idle wait time ("SQL*Net message from client"), then the bottleneck must be somewhere else. If, however, most of the time is CPU time or non-idle wait time, then you'll need to dive deeper.

Of course, this method, and any other sensible methods for performance analysis and tuning depend heavily on timed statistics. Following the widespread but never proven belief that switching on timed statistics introduces a significant performance overhead, they are generally switched off on production databases. So before you start diagnosing a performance problem, you need to ensure that timed statistics are on at least for the session that you use for the diagnosis. If that is not possible (most likely, since you probably need to modify the application to be able to

issue an alter session command), then you can switch them on for the entire instance using the alter system command, or the timed_statistics initialisation parameter.

Close in on the problem

If you have determined that it is the database that spends most of the response time, then you need to find out why it happens. Look at the response time breakdown again. If you have reached this point, then the idle wait time should be negligible. So it must be either service (CPU) time, or non-idle wait time.

CPU-Bound databases

If CPU time constitutes a large portion of the processing time, this usually means one of two things: heavy (re-)parsing, or multiple scans of the same set of cached blocks (too many logical reads).

Look at V$Sesstat to compare the CPU time used for parsing (the "parse time cpu" statistic) to the total CPU usage. If this is high (say, more than 10%), you might have an unsharable SQL problem. Look in the SQL area for statements that specify literals instead of bind variables.
If there is not much unsharable SQL, your shared pool might be too small.

If the high CPU usage is not due to hard parsing, the most likely cause is inefficient SQL.
Isolate all the statements issued to satisfy the user request and determine the heaviest few of them in terms of logical reads (V$SQLArea.BUFFER_GETS). An alternative approach would be to break down the database processing time by statement and concentrate on the most time- and resource-consuming few. Tools that can help in this direction are:

- StatsPack.
- DBMS_Proflier to answer the question "Where the time is spent?" if you deal with PL/SQL code.
- Switching sql_trace on for the ill-performing session (using *alter session* or *dbms_system.set_sql_trace_in_session*) or the entire instance and then using *tkprof* to parse the trace file.

A less likely reason for high CPU usage causing performance problems is the usage of PL/SQL functions in SQL statements. Any SQL executed in these functions appears as recursive SQL in all statistics. Again, V$SQLArea, DBMS_Profiler and *tkprof* can help you in identifying such statements and the functions they use. Some hints about tuning this case is to look for possible deterministic functions that are not declared as such. A deterministic function's result depends only on its arguments, and it does not modify its environment in any way except via its result. Declaring a PL/SQL function deterministic allows Oracle to calculate its result only once for every distinct combination of argument values. A bit of warning, though: *Any* function may be declared deterministic, even if it does not comply with the above definition.

I/O-Bound databases

The rest of the response time must be non-idle waits. If their share in the total response time is considerable, then you have some sort of resource contention. Most often, the contended resource

is the I/O subsystem. So it makes sense to split the non-idle wait time into I/O waits and other waits.

If the I/O-related waits are responsible for most of the non-idle wait time, then you have an I/O-bound database.

Each time an Oracle process sends an I/O request to the Operating System, the process waits until that request is satisfied. The event it waits on depends on the type of request issued. For example, when a user process waits for a multiblock read operation (as part of a full table scan) to complete, it registers the time waited against the "db file scattered read" wait event. The list of disk I/O-related wait event names varies from version to version, mostly because new events are added. The most common ones are:

```
db file scattered read
db file sequential read
direct path read
direct path write
log file parallel write
```

You can drill down further to the tablespace and datafile responsible for most of the disk I/O by using v$datafile and v$filestat (and v$tempfile and v$tempstat, if you use tempfiles).

The disk I/O-related performance problems may arise from various reasons. Most often the reason is either inefficient SQL combined with insufficient buffer memory (if the memory is sufficient, inefficient SQL binds the database to CPU), or unbalanced I/O putting an extremely high load on one or a few of the available disk drives, or an inefficient application design resulting in too much data processing on the client. Another indication of such an inefficient design is the overwhelming presence of simple SQL statements, and absence of complex join and aggregate queries.

Another category of I/O waits are the network waits. Their names start with "SQL*Net", and one of them is the "SQL*Net message from client" event which collects the database idle wait time.
A client-server dialog consists of (typically, in that order on the server side):

- Request from the client to the server
- More chunks of the same request if it is larger than the network data unit
- Reply from the server to the client
- More chunks of the same reply if it is larger than the network data unit

Additionally, synchronisation messages may be exchanged between the client and the server.
Each of the above steps requires the server process to wait for the network layer to process them. The wait event names are as follows:

- "**SQL*Net message from client**" – idle wait.
- "**SQL*Net more data from client**" – includes client processing time and network latency.
- "**SQL*Net message to client**" – the time needed for the OS "send()" call to return.
- "**SQL*Net more data to client**" – same as above for all subsequent chunks of the result.

It must be noted that a large number of short "**SQL*Net message from client**" waits indicates heavy processing within the client between calls to the database. This might mean the same type of inefficient application design is in place (see the discussion on disk I/O above), or that some off-the-shelf reporting tools are used which are not very good at translating user input into

efficient SQL and hence must do more work on the client side. This might be confirmed by comparing the amount of data send to the client (the "***bytes sent via SQL*Net to client***" statistic) versus the amount of data displayed to the end user.

Internal Contention

This category covers all of the remaining non-idle wait time. In a multi-session environment, access to certain resources must be serialised, so that only one session at a time modifies the resource. All such resources are protected by latches or enqueues. Enqueues and latches have different implementations, but they serve the same purpose and can cause contention. For example, putting a new statement in the SQL area is protected by a single latch. A session must hold the latch while it searches the SQL area for a matching statement. When the shared pool (and hence the SQL area) is increased, the sessions must spend more time searching, especially when there is no matching statement at all, and hold the latch for longer time. This of course prevents other sessions from doing the same type of search and they must wait on the "**latch free**" event. That is why increasing the shared pool makes an un-sharable SQL problem worse.

Another point of contention are the so called "hot blocks". Only one session at a time may modify a block buffer, so when multiple sessions try to modify the same block (e.g., a rollback segment header, the leading block of an index filled with monotonically increasing values, and so on), they must wait on the "**buffer busy waits**" event.

There are many more sources for internal contention. The causes for each non-idle wait event, and the possible remedies are well sufficiently covered in the Oracle documentation as well as in MetaLink notes and various resources on the Internet

Conclusion

DBAs routinely perform reactive tuning under enormous time pressure in live production environment. That is why they need a straightforward method to quickly find the source of any performance problem. I hope that the method outlined in this paper will enable them to quickly identify performance bottlenecks, as it has enabled me.

References

1. Shallahamer, Craig A.: "Direct Contention Identification Using Oracle's Session Wait Event Views", http://www.orapub.com
2. Shallahamer, Craig A.: "Response Time Analysis for Oracle Based Systems", http://www.orapub.com
3. Adams, Steve (Ixora Pty Ltd): http://ixora.com.au
4. Kyte, Thomas: http://asktom.oracle.com

Contact:

Vladimir Andreev
E-Mail andreev@itdepends.eu

First published in *Vortragsband zur 16. Deutschen ORACLE-Anwenderkonferenz*, 2003, ISBN 3-928490-13-3; ISSN 0939-1541