IDERA® | Database Tools Whitepaper

READY, SET, LOAD TEST

BEST PRACTICES, TOOLS, AND TRICKS FOR DETERMINING DATABASE PERFORMANCE UNDER PRESSURE

TABLE OF CONTENTS

INTRODUCTION

In software engineering, developers use load testing to determine the speed of various aspects of a system as it executes typical transactions under a specified workload. Developers often use load testing to validate or verify certain system attributes or capabilities, including scalability and reliability.

However, today's complex applications execute hundreds of SQL procedures and maybe thousands of SQL statements during a typical workload. If the application is being used concurrently by multiple users beyond expected capacity, potentially untested conditions could occur, causing performance degradation. If the slow-performing application is a business-critical one used within an organization, performance issues can have a major impact on employee productivity. Underperforming applications are especially frustrating to users—especially customers.

Automated software quality (ASQ) assurance is a segment of the application lifecycle management (ALM) market that addresses some of these issues. ASQ assurance includes test management, automated functional and regression testing, and automated stress and load testing. In this paper, we'll talk about why stress and load testing are a critical part of the development lifecycle. We'll also talk about industry trends and outline some best practices for successful stress and load testing. Finally, we'll examine how load testing can be combined with profiling to provide greater insight into application performance during code development, allowing developers and DBAs to eliminate some of the costly SQL code errors that can ultimately impact time-to-market and a company's bottom line.

IS STRESS A BAD THING? NOT FOR PRE-PRODUCTION APPS

Developers perform load testing to determine how their code will behave under both normal and anticipated peak load conditions. In theory, such testing simulates possible scenarios that will occur when the application is in production and being used by users, illuminates bottlenecks or bad performing code, and gives the developer an opportunity to tune and improve the code before it causes problems in production.

Placing a load on a system or application that is higher than predicted normal usage patterns is called stress testing. During stress testing, developers expect to find problems. In other words, the more stress you put on a system, the more likely it is you'll find problems—and this is the intent. The tester can never be sure where the tipping point will be, or when the system will be over-taxed, so he must perform many tests with increasing load volumes to discover the application's true limitations.

For example, a developer may write a query to pull results back from a search performed with an online search engine. The search executes and seems to work well. However, the developer expects several people to be performing similar searches at the same time, perhaps for the same topic or item. The query may be fast if one person is executing it, but what if 100 people execute it simultaneously? Google found that even 500ms of extra latency reduced search traffic by 20%. Stress-testing the query before production helps to eliminate this risk.

Load testing is absolutely critical for applications that experience peak load times. For example, during the holidays, a popular retail store may need to process higher than average online transactions. If the code is inadequate, the system could slow to the point of frustrating shoppers, causing them to abandon their shopping carts and purchase elsewhere. This could lead to lost revenue and a damaged brand image.

SQL load and stress testing help developers identify potential bottlenecks and give them a chance to address them before the code goes into production. This shortens the development cycle, because individual lines or batches of code can be fixed before being integrated with the rest of the application, eliminating the time-consuming "needle-in-a-haystack" approach of going through thousands of lines of code to find a single statement that's causing the issue. Moreover, load testing during development creates accountability for code contributors and helps to remove some of the blame from DBAs. Typically, the database is the key suspect in slow application performance; load testing during development removes such finger-pointing and shifts the responsibility back to the folks developing the code. And, the results of pre-production load tests can be used by developers as proof that their code contributions are not causing performance problems later on.

HOW AND WHEN TO TEST

Clearly, load and stress testing are an important part of application development, but the question is, how do you do it? A stress test involves simulating a true production environment before the code is final, and pummeling your pre-production code with increasingly high transaction volumes; how do you do that when you don't have data, and when you don't have thousands of users in-house to run the application simultaneously? To make things even more complicated, some countries have data privacy rules that prevent exposing real data to the test environment.

Load or stress testing can be performed manually, and developers have done it this way for years. But manually testing a system's performance is tedious and costly, and consumes a lot of time and resources. For example, a developer may create a load test by writing a small code block, such as Oracle PL/SQL as follows:

```
declare
   v_start   date;
   v_stop   date;
begin
   v_start:=sysdate ;
   for i in 1..10000 loop
      SELECT ...
      FROM ... ;
   end loop;
   v_stop :=sysdate ;
   dbms_output.put_line('10000 iterations elapsed time:'||to_char(v_
stop - v_start)) ;
end ;
/
```

This simple code block would execute the SQL statement 10,000 times and capture the total elapsed time it takes to complete. However, this type of test isn't really a valid load; it's simply an iterative execution. To produce a load, you would have to run this code block in multiple, independent sessions— simultaneously.

Start by saving the code block with the file name "int_exec.sql". Then you could create some number of connected sessions and execute the code block in each at approximately the same time. Or considering automating the test by writing a shell script or some other "driver" program to create the session connections and execute the code. Then, you could add instrumentation to the code to get additional details about the code's performance during the test.

We would want to create a Shell “wrapper” script to ease automation of the code. To do this we would execute multiple sessions of the same shell script to create concurrency:

```
#!/bin/ksh
export sqlplus /nolog << EOF CONNECT username/password@db
SPOOL /u01/int_exec.lst
SET LINESIZE 100
SET PAGESIZE 50
@int_exec.sql;
SPOOL OFF
EXIT;
EOF
```

To perform the same on a Windows environment, we would use the preferred method of PowerShell as seen below. The script would pull the username and password from variables stored for the user’s environment. The script could be executed by as many processes as required to create the concurrency scenario.

```
$sqlQuery = @” int_exec.sql; exit”@
$SQLPrompt = @”$username/$password”@
$outputfile = “C:\int_exec.txt”

$sqlOutput = $sqlQuery | sqlplus -silent
$SQLPrompt
EXIT
```

While the developer could query views to monitor performance, such as:

- V$ACTIVE_SESSION_HISTORY
- V$SESSION
- V$SESSION_LONGOPS
- V$SQL
- V$SQL_PLAN

It can ease the demand and the manual processing by using a prebuilt package supplied by Oracle, such as DBMS_SQL_TUNE. To perform this from the command line if Enterprise Manager is unavailable for SQL Monitor access, the tester can gather the following as they are performing their test cases:

1. Gather the information collected during the testing process using DBMS_SQL_TUNE logged in with SQL*Plus:

```
SET LONG 1000000
SET LONGCHUNKSIZE 1000000
SET LINESIZE 1000
SET PAGESIZE 0
SET TRIM ON
SET TRIMSPOOL ON
SET ECHO OFF
SET FEEDBACK OFF
```

```
SPOOL <path>/report_sql_monitor_list.htm
SELECT DBMS_SQLTUNE.report_sql_monitor_list(
  type         => 'HTML',
  report_level => 'ALL') AS report
FROM dual;
SPOOL OFF
```

2. Gather the SQL_ID, the unique identifier for the query:

```
select sql_id, substr(sql_text,1,200) sql_text
   from v$sql
   where upper(sql_text) like '<insert unique text from query>' ;
```

Note** If a comment is used such as "/* initials_testing */" this can simplify identifying your test query.

Use the SQL_ID gathered in the query above and provide a detailed report regarding the test query:

```
SET LONG 1000000
SET LONGCHUNKSIZE 1000000
SET LINESIZE 1000
SET PAGESIZE 0
SET TRIM ON
SET TRIMSPOOL ON
SET ECHO OFF
SET FEEDBACK OFF
```

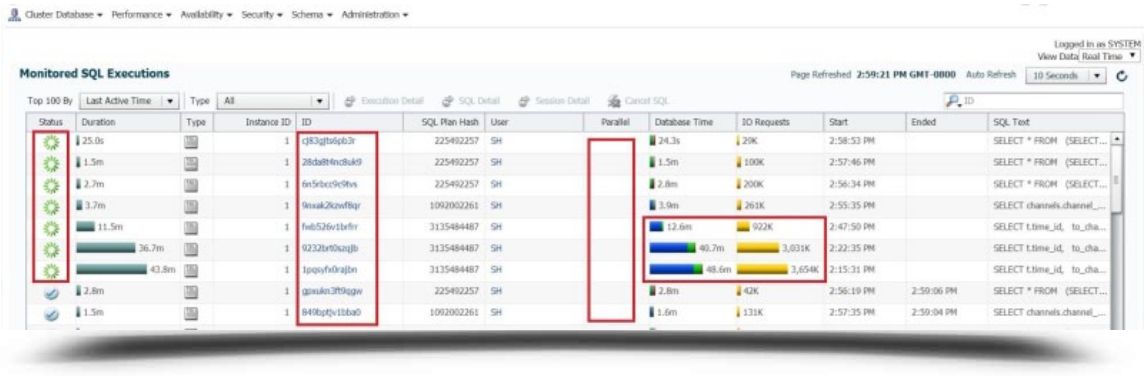
```
SPOOL /host/report_sql_detail.htm
SELECT DBMS_SQLTUNE.report_sql_detail(
  sql_id       => '<SQL_ID from query>',
  type         => 'ACTIVE',
  report_level => 'ALL') AS report
FROM dual;
SPOOL OFF
```

The report can be transferred to the tester's desktop with FTP or SCP and viewed with a browser to inspect the performance of the test and any issues.

ORACLE ENTERPRISE MANAGER AND SQL MONITOR

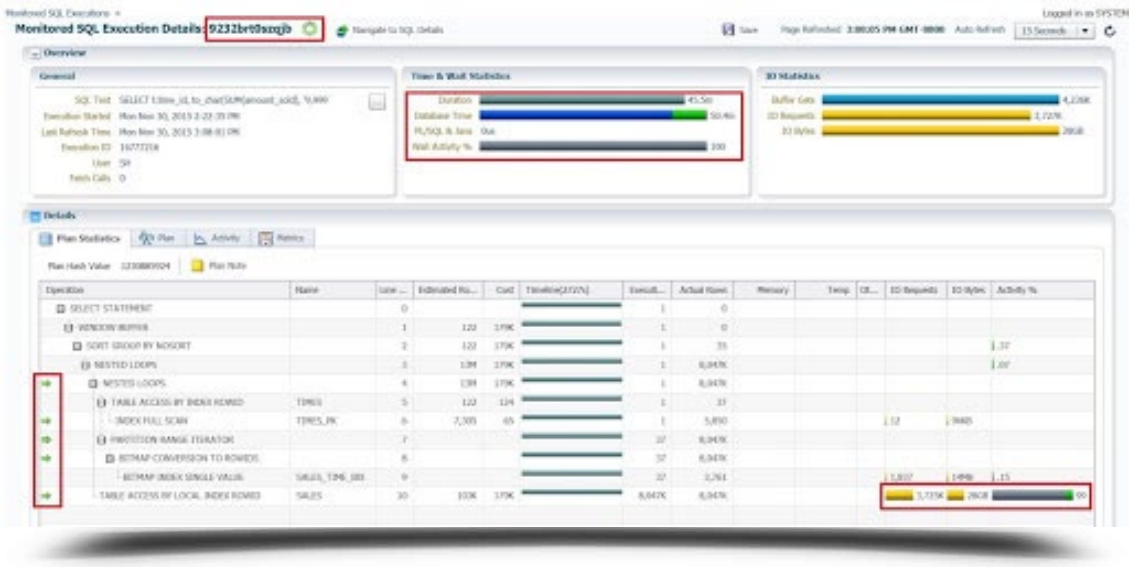
There are multiple profiling tools, but two offered from Oracle collect execution time and metrics surrounding the performance of the process. Oracle SQL Monitor requires the Oracle Database Tuning Management Pack, but if available, is the preferred method to collect execution time and summary of metric data.

Using Oracle's Enterprise Manager, direct access to SQL Monitor can be attained, once logged into the database, by clicking on Performance ➜ SQL Monitoring. To use SQL Monitor, ensure your user has the required privileges as a developer. Please refer to Oracle's documentation for more information on the requirements.



From the list of processes listed, you can see which ones are active, along with the execution time, the SQL IDs, if the process is utilizing any parallel and the amount of IO (blue), CPU (green), and further breakdown of IO Requests (reads and writes).

The SQL ID for each process is also a link to access deeper details about the process. Double click on your process, (not the SQL Text in the furthest right hand column) and the SQL Execution Monitored Details page will result.



For the SQL ID in question, information will be displayed for execution time, wait events, and the execution plan for the SQL in question.

LOAD TESTING AND THE DEVELOPER

Developers spend a lot of time testing features and functionality, and neglect performance testing altogether, handing off potentially poorly performing code to production. As a result, load testing is typically a step that occurs after they're done and before the production release—either by a QA team or designated tester. If the load testing reveals an issue with some code's performance, then the code must be given back to the developer for modification.

The DBA may also perform load testing using the same methods as the developer; however, he usually isn't responsible for it. Most organizations have a division of duties when it comes to application development and testing, and post-production code maintenance and performance monitoring. DBAs typically fall into the second category. They need tools that can identify code issues quickly—ones that gather diagnostic data about "bad" application code from their production environments. Such tools help them determine if a problem is caused by too much of the same thing running concurrently or by "bad code."

If a DBA finds "bad code," he may try to find a way to mitigate or circumvent the problem. For example, in Oracle, he can create a SQL profile or baseline to produce a better execution plan for a specific statement. In other words, he creates a temporary fix to stop the bleeding in production. While he's doing this, he may send the problem code back to the developer for a long-term fix—but that can take time. Often, developers have a hard time determining what needs to be done to fix poorly performing code, and it can be a guessing game. They try something, send it back into production and cross their fingers that it works. The closer the two groups can work together and collaborate, the more efficiently the problem can be solved.

The earlier performance testing is injected into the development lifecycle, the easier it is to modify poor design choices and correct them. Otherwise, it can cost a lot of time and money to rework code that's already been released—time and money that could have been saved, if only time had been taken early in the process for thorough performance testing.

A developer can test his own code, then move to integration testing to see if code works with other modules. Later in the development cycle, he may choose to test how his code performs when interacting with other functions. For example, in an insurance system, the refund and cancellation systems need to interact smoothly. Once the code is working as expected, the developer can then move on to testing various user scenarios to ensure the application performs the business function that was promised.

AN ALTERNATIVE TO MANUAL CODE TESTING

Automated load testing tools are designed to simulate the load of multiple users against a server-based application to understand performance and give developers an opportunity to tune code before it goes into production. Especially in the late stages of product development, automation is necessary for effective performance testing, because full-scale application load tests cannot be carried out properly or in a timely manner without it.

Efficient automated load testing tools will launch multiple transactions, or threads, through a single system at once. Some tools are complex and have modules that can test various aspects of a system, while others are focused on testing specific functionality, such as SQL code performance. SQL code developers may have more success with a focused tool that is built for evaluating and fine-tuning pre-production SQL queries.

Products from vendors like Microsoft and Oracle provide workload replay features that can enhance the testing process to ensure that production workloads are taken into consideration when testing performance in a non-production environment. The examples explained below provide guidelines for using these features and offer an opportunity for the tester to provide more complete test cases and performance analysis.

ORACLE DATABASE REPLAY

Oracle Database Replay provides the ability to capture database workloads and replay them in development, test, and other copies of the original database environment. This feature is part of the Real End-User Testing product from Oracle and requires added licensing costs.

If the tester has Oracle's Enterprise Manager at their disposal, the graphical interface allows the user to bypass the manual preliminary steps or command line execution to collect workload information.

If the tester is without Enterprise Manager, Oracle's Workload Analyzer must be run before the Database Replay to ensure all requirements are configured and to identify any code that won't be included in the capture. The target database the workload is to be run on, including the analyzer, must be a duplicate of the original database the workload was captured from. This can be done via a RMAN duplicate, a snapshot, import and export, etc.

Oracle's Workload Analyzer is a Java program used to identify sections of a captured workload that won't replay accurately due to errors during the capture or due to incorrect or insufficient

data. There are additional features that may not be supported by Database Replay that are also identified with this product and it's important to identify and isolate these areas before replaying the workload to understand what will be included in the replay, its impact and/or failure.

An example of a Workload Analyzer command, can be seen below. Note that the java path has been set as part of the environment settings beforehand.

```
java -classpath
$ORACLE_HOME/jdbc/lib/ojdbc6.jar:$ORACLE_HOME/rdbms/jlib/dbrparser.
jar:
$ORACLE_HOME/rdbms/jlib/dbranalyzer.jar:
oracle.dbreplay.workload.checker.CaptureChecker /scratch/capture

jdbc:oracle:thin:@myhost.mycompany.com:1521:orcl
```

With the execution of this command, the user will be required to insert the username and password for a user with execute privileges on DBMS_WORKLOAD_CAPTURE and the SELECT_CATALOG role.

To capture a workload on the production database, the DBA should have a workload directory set up to house the workload capture.

```
$mkdir /u01/app/oracle/admin/<directory>/capture
```

Log into the database the workload will be created in and begin by adding the directory at the database level:

```
$sqlplus / as sysdba
<create or replace directory capture_dir as'/u01/app/oracle/
admin/<directory>/capture';>
```

Once this is set up, enabling the workload capture is simple from the command line:

```
@$ORACLE_HOME/rdbms/admin/wrrenbl.sql;
```

The capture will then commence and once finished, the capture will need to be properly disabled by running the following script from the command line as the same user with the appropriate privileges as discussed previously:

```
@$ORACLE_HOME/rdbms/admin/wrrdsbl.sql;
```

Once completed, the binary workload file must be transferred via FTP, SCP or other method to the destination server and the replay directory.

Create the working directory on the new database, (if it doesn't already exist):

```
$sqlplus / as sysdba
<create or replace directory capture_dir as '/u01/app/oracle/
admin/<directory>/capture';>

$scp oracle@<ip address of host> /u01/app/oracle/admin/<directory>/
capture/
```

Once the file is transferred, then there are a number of preliminary steps, such as user and connection remapping that must be completed. This offers the ability to execute the same workload as is commonly performed in the production environment, allowing for time to analyze, fix issues and test repeatedly.

The simplest method of replaying the workload is to use Enterprise Manager which offers a simple user interface to access the file and run the replay. If the user doesn't have this available, an API can be utilized to define filters, timeouts, options and to start, pause and resume the replay as part of testing.

Run the workload and perform the next series of tests:

```
<exec dbms_workload_replay.initialize_replay  ('REPLAY_
DEMO_4','CAPTURE_DIR');>
<exec dbms_workload_replay.prepare_replay;>
<exec dbms_workload_replay.start_replay;>
```

The status of the database replay can be checked with the following command from SQLPlus:

```
<select id, name, status, duration_secs  from dba_workload_replays;>
    ID NAME          STATUS     DURATION_SECS
---------- -------------------- ----------- -------------
18 REPLAY_DEMO      IN PROGRESS      1089
```

When you have completed the workload replay, you can remove all this overhead from the system by running the following command from SQL Plus, using the Replay ID from the previous query:

```
<exec dbms_workload_capture.delete_replay_info(18);>
```

Oracle's Database Replay has significant features from the End User Testing interface and even more from the command line. For a more detailed description and examples from the product, reference the official documentation.

When you're using a more complex, multifunction tool, you might find yourself waiting until late in the development cycle, when more components are already integrated into the system. If a problem occurs, finding the source of the issue may require reverse engineering as you manually sift through individual SQL statements to find poor-performing code.

SQL SERVER DISTRIBUTED REPLAY

SQL Server also has an interesting feature called Distributed Replay. It helps to assess the impact of the hardware and operating system upgrades during the future SQL Server Upgrades. If you are into SQL Server performance tuning, you can also take advantage of this feature to gain lots of insight which is not always easily available.

Though SQL Server Distributed replay is a very feature-rich and powerful tool, it is not very widely used because of its similarity with SQL Server Profiler. However, many do not know that the biggest limitation of the SQL Server Profiler is that it can replay the workload from only a single computer, whereas SQL Server Distributed Replay actually overcomes that limitation.

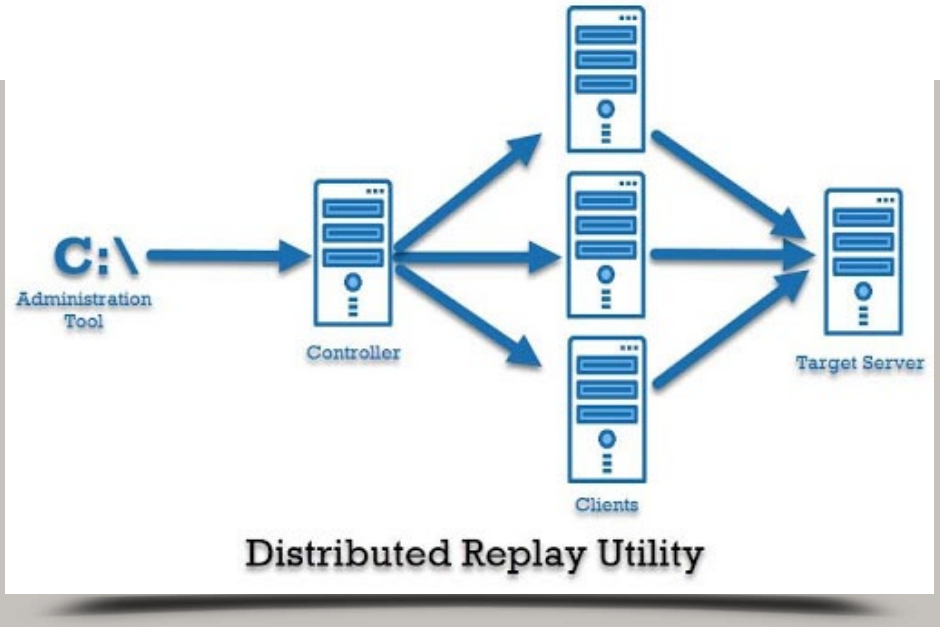
The biggest advantage of the Distributed Replay is that it can replay a workload from multiple computers and efficiently and easily. It is one of the most important tools to check compatibility of testing and capacity planning.

If you are a SQL Server performance tuning expert, you can use this tool to test operating system upgrades, hardware upgrades, or even index tuning scenarios. The biggest advantage of using

this tool is the concurrency in the captured test is so high that a single replay client cannot sufficiently simulate it.

The Distributed Replay administration tool, controller, and client can be installed on different computers as well as on the same computer. This gives the product robustness to scale to perform a larger load testing.

Here is a diagram of the architecture of SQL Server Distributed Replay Utility.

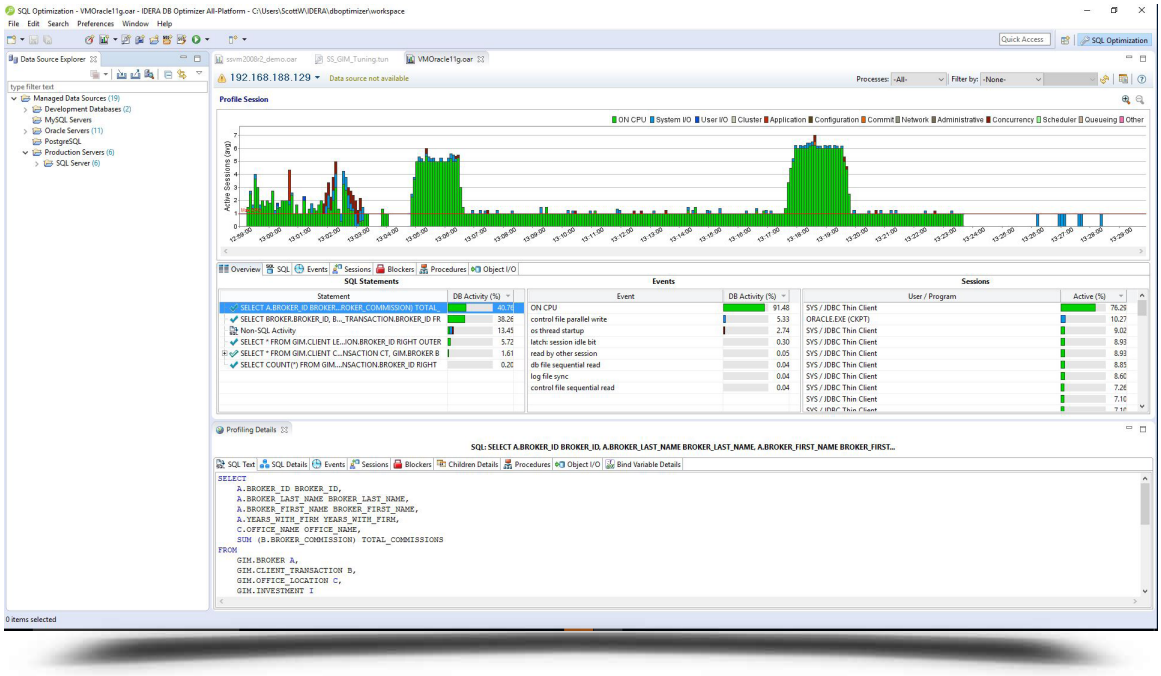


IDERA DB OPTIMIZER

IDERA offers a database query tuning tool called DB Optimizer, and it's the most powerful yet simplest database performance tool on the market. The Load Editor provides SQL stress testing that simulates a number of parallel users and executions over a specific period of time or execution cycle. The SQL code can be run in the Load Tester to test execution by multiple concurrent users. User load testing is very often done by just one single user, and then problems don't appear until production use with multiple concurrent users. Concurrent user testing is a breeze in DB Optimizer.

The integrated Profiler can be run while the Load Tester is executing to show clearly the impact on the database. You can continuously profile an entire data source within a configurable span of time. When fine tuning or testing SQL, you can profile the execution of a single stored routine when profiling an entire data source is not desired. Finally the Profiler can be run on any production database to clearly show load, bottlenecks and sources of bottlenecks or resource consumption.

Data is displayed in real-time while profiling is in progress. All data and metadata pertaining to a profile session can be saved as a single entity into an archive file. Profiles can be shared across multiple workspaces and machines for collaboration purposes.



BEST PRACTICES FOR SQL LOAD TESTING

With the extensive adoption of Agile development methodologies, numerous tools to both ease and automate SQL load testing have been introduced in recent years. The scope of the SQL load testing will decide which tool satisfies the requirements. Tools that focus on SQL development will often be easier to use as they are focused on this purpose, but many multi-functional load testing tools may offer broader opportunities for automation and other DevOps practices that are key to successful Agile development.

There are numerous methodologies on proper load testing; however, your methodology can affect time-to-market and overall project costs. Let's look at what the experts are doing to test and document the performance of their SQL, and consider some best practices for minimizing code changes late in the development cycle.

5 STEPS TO EFFECTIVE LOAD TESTING

- Start Early
- Automate deployment of test enviroments
- Automate the test cases and flow
- Consult the business
- Run, refine, repeat

START EARLY

Using a tool that's focused on SQL allows you to begin testing earlier in the development cycle. For example, you can determine a single query's maximum load by having it execute thousands of times, finding any problems, and then tuning it until it performs as intended. Then, you can produce a report to present to your DBA that shows the code passed performance testing and is production-ready. This eliminates finger-pointing, because you have proved your code, by itself, is healthy. Additionally, early testing eliminates potential problems later in the development cycle that may have been caused by poor-performing individual SQL queries.

AUTOMATE DEPLOYMENT OF TEST ENVIRONMENTS

Developers typically mock up their data for testing purposes, rather than testing performance using actual data. With the introduction of the General Data Protection Regulation (GDPR), there are significant laws in place to prohibit using authentic data for testing, but it's generally unwise to use Personally Identifiable Information (PII) for testing. It's important to have the right amount

of data for the test to yield viable results, but not enough to make a company liable to fines, which can impose up to 4% annual revenue. For example, if you're running selects against a large database, a bad select can cause more issues as SQL volume increases. If the database is too small, and you're testing update functionality, you may find fewer collisions occur than would actually occur in a production environment.

To produce relevant results, a good rule of thumb is to construct a test database that is roughly 25% to 50% the size of the target database, even better if it includes the ability to virtualize (offering the full data set, but without the storage requirements), and to utilize data masking to protect from critical data vulnerabilities.

SCRIPT OUT AND AUTOMATE TRANSACTION FLOWS

After you've selected your tool and built your test database, you must capture the activity on your system and review it to determine a typical workload. Then, create a script for performing the expected user transactions. Consider the following:

- What are your most critical business processes and how often are they processed? (e.g. number of sales activities per day, number of client requests per day, etc.)
- What is the typical user behavior for the application?
- What are the typical keystrokes a user will choose during the transaction?
- In what order will processes execute?
- What are acceptable response times for the application?
- How many concurrent users do you expect?
- Will there be peak usage times? What processes will be affected the most?

Many automated tools will capture this information for you and may even have an automated function for creating the script. All you have to do is execute the transaction, and the tool does the rest. You should be able to script a workload in numerous ways to mimic how different types of users may use your system.

CONSULT THE BUSINESS

The developer bridges the gap between the user and the business; he must understand the code and how it executes, what the user will experience, and whether the application meets the needs of the business. Much of the information he needs to build a test script for the application comes from the business. In other words, the business need informs the test method. If a retailer needs an order entry system to run optimally during the week before Christmas, and they're estimating 1000 orders per minute, the developer needs to incorporate those parameters into his test methodology. He'll want to make sure the system can handle 1000 transactions per minute without performance degradation.

RUN YOUR SCRIPT

Once you have the script, devise a testing strategy. Testing isn't simply running the script and recording performance; you must simulate the production environment. Vary the order of transactions and tasks, just as if multiple users were using the system without knowledge of the proper workflow. Inevitably, different users will have different approaches to completing transactions. Also, start with a small number and work up to larger transaction volume so you can determine at what point problems occur. Most load testers will "warm up" the system by testing one user, then 20, then 50, then 100 and so on. By ramping up a workload, you can mimic an increasing number of users doing a representative set of tasks and capture information about how long it took each task to execute. This approach makes it easier to identify the code that caused the system to slow down or crash. You can also determine which transactions will scale successfully as your workload increases, and which ones won't.

An efficient load testing tool will allow you to mark start and stop times for transactions, and create graphical representations of various metrics, such as average response time by transaction.

EXPECT PROBLEMS, AND TRY TO CAUSE THEM

Profiling helps to filter out well-performing light-weight SQL and collects information on heavy-weight SQL. SQL code that is heavy-weight is usually either long-running queries or queries that are short but run so often that they put load on the database. The idea is to look at the load on the database which can quickly indicate how the database is functioning.

The whole point of load and stress testing is to find the problems so they can be fixed. Be suspicious if no problems occur. Refine your test so that it runs a higher transaction volume, or continue to vary the order of the tasks in the script. You're not testing to make sure the system

works; you're stressing the system to find out where it breaks. Find the breaking point and decide whether your system will actually experience those conditions in a production environment. If it won't, then you're safe. If it will, you need to...

GO BACK TO THE DRAWING BOARD

Here's where the hard work begins. Using the data collected during testing, you must determine what needs to change. Some tools allow you to fine-tune SQL code without leaving the testing environment. This method is extremely efficient, since you can easily reference test results during tuning, then re-run the script against the new code.

PROFILING AND INSTRUMENTATION

An important decision to make is whether to build instrumentation into your code. Instrumentation refers to the ability to monitor or measure performance, diagnose errors, and write trace information using code instructions. Programmers use instrumentation to monitor specific components in a system.

Oracle systems, for example, have built-in procedures that allow for the labeling of sections of code (i.e. a single statement, a group of statements, or a transaction). Examples of labels in a retail application may include "order entry," "insert order," or "update quantity." After the test runs, the developer can apply filters to view the behavior of the labeled code. For instance, he can easily see how much time each "order entry" transaction consumed.

With instrumentation, you can build code tracing into the application and receive messages about how the application executes. It allows the developer to track down and fix programming errors. Other forms of instrumentation include performance counters that track application performance, and data logging, which helps track major events while the application runs.

Instrumentation is often used to perform profiling functions. Profiling technologies measure program behaviors during testing and gather information about what resources are used for various transactions within an application. Similar to instrumentation, a developer can use profiling to look at I/O times or CPU times, or to find where the application is wasting time. Profiling technologies use algorithms or other mathematical techniques to discover patterns or correlations in large quantities of data.

Another way to perform profiling is by using an automated tool called a code profiler to analyze the binary executable of an application's source code. After running a load test, you can examine the profiles to identify trends or anomalies—anything that might signify a problem with the code.

Profilers are often separate tools that must be used in conjunction with load test tools. It's important to have both capabilities—the load test tool can only see a single transaction, even though there may be hundreds of SQL statements executing behind the scenes. With a profiling tool, you have very granular insight into the code as it runs. The profiler monitors what's going on during the load test, so you can correlate slow response times with exact SQL statements and pinpoint the bottleneck. Filters within the profiler help you pull out the information you need quickly and efficiently. For example, you can ask to see transactions performed by a specific time period or action. If the peak load occurred at 2:30pm, you can filter the data for transactions that ran at 2:30, and see details about how the code performed.

COMBINING LOAD TESTING WITH PROFILING

In practice, load testing and profiling should go hand in hand. With Oracle's Real User Database Testing, much of the load testing and profiling is performed for the tester. IDERA DB Optimizer also provides integrated load testing and profiling capabilities that can be used with multiple database platforms.

If you're new to profiling, as a test runs, the profile measures the results. Without profiling, the best a load test can do is measure performance; it can only identify problematic transactions, not individual lines of code, and there's no real way to find the true culprit of the performance problem.

Combining load testing with profiling provides a holistic approach to performance testing, but previous manual methods could be resource and time intensive. Newer automation tools allows fewer testers to do more and to do it more effectively.

Full profiling with monitoring allows you to not only look at the performance of lines of code, but see how other processes may interact with code to cause longer wait times. For example, if you run a load test on a query that brings back product information, you can profile the database first and then run the load test. Watching the profiling session as the test is running allows you to see in real time what's going on during the transaction. You can also see if other actions

have repercussions on the query. Running the test without a profiler may give you a false sense of security. Profiling shows exactly how your code affects the database, and whether other processes running on the database are affecting the performance of the query.

With many companies' testing tools, like we've shown with IDERA DB Optimizer, Oracle Database Replay, SQL Server Distributed Replay, and SQL Monitor, these tools are incorporated into one. You can use the profiler function within the load test tool to watch the test while it runs. If you see performance start to degrade, you can capture the exact statement that's running in real time. Then you can pull that statement into a tuning session, improve the code, and, before putting it back into production, test it individually to verify that the new code will scale out. Once you complete the load test on the new code, you can integrate it into the application, and start over.

This method of code tuning is far superior to running a load test late in the development cycle with a complex tool, manually sifting through hundreds of transactions and lines of code to find the problem SQL statement, taking that code out of the test environment to fix it, then having to re-integrate the code, without any guarantee that you didn't introduce a new problem during the tuning session. Additionally, having both capabilities in a single tool eliminates integration costs and reduces the amount of capital outlay.

DO'S AND DON'TS

Do	understand the business needs before defining your test methodology.
Do	look for a tool that provides integrated load testing and profiling functionality.
Do	test often throughout development.
Do	work to add instrumentation and automation to all testing.
Do	use a test database that's as close to production size as possible and mask critical data to protect for GDPR, HIPAA, etc.
Don't	load test without profiling.
Don't	test the app to make sure it works; test it to find problems.
Don't	leave performance testing until the last minute.

CONCLUSION

The days of building an application and rolling the dice are over. Performing load and stress testing early, and often, in the development cycle, gives developers more control over the quality of their code, and confidence that it will perform as expected once it's integrated with the rest of the application.

However, before choosing a methodology for load testing, it's important to consider how the application will be used, what it will be used for, and how many concurrent users may stress the system at peak load. Especially for applications that must scale to serve thousands of users or more, automated load test tools save development teams time and money by eliminating manual tasks and providing greater accuracy. The best tools integrate both testing and profiling functions to simplify and streamline the process of identifying and tuning poorly performing code, helping to accelerate development and time-to-market with high-performing applications that can withstand even the most strenuous use cases.

ESSENTIALS OF AN EFFECTIVE LOAD TESTING TOOL	
Automated Load Testing	Simulate production environments by efficiently running SQL statements hundreds of times and in parallel.
Database Profiling	Locate database bottlenecks immediately and identify poor SQL code for immediate performance resolution.
Reporting	Create organizational transparency with analytical reports containing context on both performance issues and solution testing.
Integrated Capabilities	Eliminate integration costs of multiple tools and increase productivity with all essential components in one application.