# ARE WE DONE YET?

## CERTIFYING APPLICATION PERFORMANCE UNDER NEW DATABASE VERSIONS OR ARCHITECTURES

*David A Haas, The Nielsen Company*

## OVERVIEW:

Fear of the unknown can cripple a cautious development team and prevent them from making upgrades to the database or architecture necessary to maintain technological and business relevance. Lack of knowledge on the impact of changes can empower a reckless development team with the false confidence to make upgrades that render their application unusable. In the end neither of these options is viable in today's fast changing world.

Application Development teams require proven techniques to evaluate the impact of a change, empowering them to make the calculated risks necessary for success and continued relevance.

This paper will provide insight into the issues associated with testing application performance on a complex decision support system. Illustrate the methods used to capture real-world test cases insuring that the results are relevant to the business. Provide examples of a proven statistics based approach used to compare performance before/after an Oracle 8i to 10g upgrade combined with underlying partition structure and block size changes, as well as a radical shift in database architecture isolating static and dynamic data content into separate instances joined into a single virtual instance using DBlinks.

## OUTLINE:

1. Starting Assumptions
   A. Test database and web/app server separately
   B. Execute test cases serially in a fixed order
   C. Utilize an isolated server/network
2. Test Case Selection
   A. Use "real world" test cases captured from actual users
   B. Create a sub-set insuring full coverage of conditional paths
3. Benchmark Process
   A. Perform initial iterations of individual test cases and capture total time for each
   B. Evaluate the confidence interval of the average performance for each test case
   C. Continue iterations until you achieve the desired confidence interval
4. Proposed Environment
   A. Perform initial iterations of individual test cases and capture total time for each
   B. Compare to current benchmark and focus on extreme outliers
   C. Evaluate individual queries within the outlier test cases to mitigate issues
   D. Repeat these steps until performance is within defined service levels
5. Final Comparison
   A. Regenerate both benchmarks with new code
   B. Summarize results for the business

## STARTING ASSUMPTIONS

There is nothing more frustrating than completing a long database conversion project, then having the implementation deferred, waiting for confirmation that the end-user performance is comparable to the current implementation. The common sense principles illustrated below were developed in response to that exact situation, only after the database team took responsibility for testing and validating the application performance under the new architecture, were we able to deliver on the promises that had been made to the business. The lessons learned from the first attempt, which ended without a successful delivery to production, were overcome the following year with the successful deployment of the database upgrade and validated the year after that in another major renovation to the database architecture. The team could only be successful once we took a test first approach; validating the performance before starting the data conversion process and preemptively removing this delivery barrier.

### TEST DATABASE AND WEB/APP SERVER SEPARATELY

While ultimately the end user perception of the entire system is the only thing that matters, it is impossible to capture and validate performance benchmarks from this point of view. Instead it is imperative to only exercise and measure performance on those parts of the system which are impacted by the proposed change. In our case, this meant running Java method calls directly on the database server, without exercising the actual front end components which were not impacted by the proposed architecture change.

### EXECUTE TEST CASES SERIALLY IN A FIXED ORDER

In order to come up with a dependable benchmark, a statistically reliable estimate of the average performance for each individual test case is required. This meant that the tests needed to execute in consistent order and serial fashion to prevent differences in performance due to caching from prior or concurrent test cases. If the test cases were run in parallel one test running shorter or longer would significantly impact the environment of all remaining test cases. Only by running the individual cases serially can limit performance impact from external sources.

### UTILIZE AN ISOLATED SERVER/NETWORK

The use of a dedicated database server avoids introducing variance from the network or application server from entering into the benchmark statistics. Some variance in a limited number of test cases that do not converge towards a statistically reliable estimate of their average performance may be inevitable. In our case, we were not able to eliminate some variance in the results generated within the storage environment, as impact from processes on other servers within the production and development environments were introduced via the SAN array.

## TEST CASE SELECTION

In order to sell the results of the testing to the business stakeholders, the test cases need to be representative of typical usage and not randomly generated. At the same time, full coverage must be maintained exercising all queries within the application, including the conditional paths which impact the amount of data retrieved and the calculations performed.

### USE "REAL WORLD" TEST CASES CAPTURED FROM ACTUAL USERS

The methods used to define the set of tests will depend on the application which is being updated; the test cases selected need to simulate as closely as possible the type of load that will be placed on the new architecture once it goes into production. In our case, we met this need by extracting the Java method calls submitted by actual users, available in logging tables on the live site. This provided a known good set of prompt values which worked well together producing results which were business relevant.

- The sample was taken from reports run within the past three months, against the most current data version, in order to insure that the data used in the report was still available and avoid any potential with application changes.
- The sample was limited to clients who were hand selected to guarantee a good cross section of client types. Clients with extensive custom data and without, covering the discrete usage patterns of the Manufacturer, Retailer and Broker clients, who rely on diverse aspects within the Spectra suite of applications.
- The sample was limited to reports whose run time fell within typical usage ranges. This avoided issues with reports whose execution time was too short, where it would be impossible to detect performance changes; or too long, that would be impractical to execute in multiple iterations.

### CREATE A SUB-SET INSURING FULL COVERAGE OF CONDITIONAL PATHS

Once the sample has been defined, a representative sub-set of test cases should be selected that maintains the level of coverage achieved from the sample. We met this need by implementing a process to analyze the contents of both the sample and the a sub-set which assured full coverage of each report type and conditional paths within each client, along with a second process which generated a representative sub-set of tests that could be used for analysis. The statistics for both the sample and the sub-set were summarized in an XML document, allowing us to understand the level of coverage and adjust the set of clients or date range as needed to generate a complete set of test cases.

```
- <LIVE>
  - <STATS QUERY="SELECT USERID, CLT_ID, APP_ID, TIME_RUNNING, MOD_ID, STATUS, FILE_SIZE,
      PKG_ID, PROMPTIDS, ID FROM REPORTS WHERE ERROR_CODE=0 AND DATE_CREATED BETWEEN
      TO_DATE('03/01/2006', 'MM/DD/YYYY') AND TO_DATE('05/30/2006', 'MM/DD/YYYY') AND
      CLT_ID IN ('AHB', 'INFCLIENT', 'GALLO', 'KRAFT', 'PEPSI') AND MOD_ID='ATHENA' AND PKG_ID LIKE
      '19%' AND TIME_RUNNING BETWEEN 15 AND 1800 ORDER BY CLT_ID, PKG_ID">
    + <APP_ID CARD="49" TOTAL="1633">
    - <CLT_ID CARD="5" TOTAL="1633">
      + <V VALUE="AHB" COUNT="142">
      + <V VALUE="GALLO" COUNT="59">
      + <V VALUE="INFCLIENT" COUNT="472">
      + <V VALUE="KRAFT" COUNT="471">
      + <V VALUE="PEPSI" COUNT="489">
      </CLT_ID>
      <FILE_SIZE AVG="644002.838334354" COUNT="1633.0" />
    + <MOD_ID CARD="1" TOTAL="1633">
    + <PKG_ID CARD="8" TOTAL="1633">
    - <PROMPT_COMPONENTS>
      + <CL_ID CARD="217" TOTAL="1364">
      + <DS_ID CARD="21" TOTAL="1573">
      + <GTL_ID CARD="102" TOTAL="3443">
      + <MSR_ID CARD="8" TOTAL="339">
      + <SC_ID CARD="14" TOTAL="322">
      + <SGM_ID CARD="14" TOTAL="943">
      </PROMPT_COMPONENTS>
    + <PARAMS>
    + <STATUS CARD="3" TOTAL="1633">
      <TIME_RUNNING AVG="111.83710961420698" COUNT="1633.0" />
    + <USERID CARD="196" TOTAL="1633">
    </STATS>
```

```
+ <APP_ID CARD="49" TOTAL="261">
- <CLT_ID CARD="5" TOTAL="261">
  + <V VALUE="AHB" COUNT="29">
  + <V VALUE="GALLO" COUNT="18">
  + <V VALUE="INFCLIENT" COUNT="81">
  + <V VALUE="KRAFT" COUNT="59">
  + <V VALUE="PEPSI" COUNT="74">
  </CLT_ID>
  <FILE_SIZE AVG="395741.5134099617" COUNT="261.0" />
+ <MOD_ID CARD="1" TOTAL="261">
+ <PKG_ID CARD="8" TOTAL="261">
- <PROMPT_COMPONENTS>
  + <CL_ID CARD="106" TOTAL="274">
  + <DS_ID CARD="17" TOTAL="213">
  + <GTL_ID CARD="55" TOTAL="569">
  + <MSR_ID CARD="5" TOTAL="32">
  + <SC_ID CARD="10" TOTAL="37">
  + <SGM_ID CARD="10" TOTAL="108">
  </PROMPT_COMPONENTS>
+ <PARAMS>
+ <STATUS CARD="3" TOTAL="261">
  <TIME_RUNNING AVG="103.38697318007662" COUNT="261.0" />
+ <USERID CARD="116" TOTAL="261">
```

*An example of the XML documents for the sample and sub-set.*

## BENCHMARK PROCESS

Like the selection of the test cases, the process used to execute those tests is also highly dependant on the application whose performance is being evaluated. In our case, in order to execute the Java method calls directly against the database server; we utilized a JMeter framework to execute the test cases, capture the execution time for each test case and loop through a predetermined number of iterations. The input to this framework is a text file containing all of the Java method calls for a given test case, along with unique identifying information; including the report type, a unique report id and a client designation. The output is a text file containing the timestamp at the start of execution, the elapsed execution time in milliseconds, the identifying information for the test case, followed by the result code and any error messages which may have been generated. We considered having JMeter return more detailed information about the execution time for each method within a test case or even each SQL statement within the various methods. However, this information was not necessary until an actual performance issue had been identified and would have greatly complicated the analysis of the performance of the individual test cases within the sub-set.

*SAMPLE INPUT:*

```
Report Builder by Stores:3232207:AHB;call SP_CREATE_CLEANUP_TABLE('0003232207')call
GeotradeList.getListExpansion('0003232207','19','27^20','STR.AHB^TRADE.GEO')call
GeotradeList.getTradeReferenceList('0003232207','19','1','STR.AHB^TRADE.GEO','','30')call
GeotradeFacts.getGTFacts('0003232207','19','STR.AHB^TRADE.GEO','NAME')call
GeotradeFacts.factRowsToColumns('0003232207','19','T_GT_ID_LIST_','T_GT_FACTS_','T_R2C_','
NAME')call
GeotradeSegmentation2.getDefaultBaseData('0003232207','19','STR.AHB^TRADE.GEO','1','T
_GT_COUNTS_TEMP_')call
GeotradeFacts.getGTFacts('0003232207','19','STR.AHB^TRADE.GEO','27275^LST.MEM^LST.MEM^LST.
MEM','T_MY_FACTS_','T_CELLS_')call
GeotradeFacts.getGTFacts('0003232207','19','STR.AHB^TRADE.GEO','LATITUDE^STR.LOC^TXT^DBL.F
CL^LONGITUDE^STR.LOC^TXT^DBL.FCL','T_MAP_FACTS_','T_MCELLS_')call
GeotradeFacts.getGTParentsByGTL('0003232207','19','STR.AHB^TRADE.GEO','CLT.NONE','0')
```

*SAMPLE OUTPUT:*

```
timeStamp,elapsed,label,responseCode,responseMessage,dataType,bytes
08/18/06 18:12:17,142498,AHB^3232207^Report Builder by Stores,,,,0
08/18/06 18:14:39,265381,AHB^3233202^Retail Interaction by Stores,,,,0
08/18/06 18:19:05,19983,AHB^3234834^Target Snapshot,,,,0
08/18/06 18:19:25,71435,AHB^3233654^Store Proximity,,,,0
08/18/06 18:20:36,205324,AHB^3236346^Target Ranking by Stores,,,,0
08/18/06 18:24:01,2237246,AHB^3237368^Report Builder by Stores,,,,0
08/18/06 19:01:19,194689,AHB^3237782^Account vs Market (Demographics),,,,0
08/18/06 19:04:33,88784,AHB^3238600^Account vs Market (Demographics),,,,0
08/18/06 19:06:02,117789,AHB^3241288^Target Ranking by Stores,,,,0
```

### PERFORM INITIAL ITERATIONS OF INDIVIDUAL TEST CASES AND CAPTURE TOTAL TIME FOR EACH

With all of the preparations completed, it's time to execute the sub-set of test cases. Run a limited number of iterations first in order to generate some basic data about the performance of the sub-set and determine the stability of the test cases. The results are aggregated for each individual test case across all of the iterations; calculating the number of executions for a given test case and the standard deviation of the execution time in milliseconds, along with the average, minimum and maximum execution time. In our case, we did this by importing the text file into Microsoft Excel and building pivot tables to generate the aggregate data. This allowed the stats to be updated in a couple of mouse clicks, one to refresh the data from the log and a second to refresh the pivot table. There were some performance issues within Excel once the number of iterations increased and the size of the spreadsheet went over 5mb; leaving plenty of time to get a cup of coffee or cruise the web while the stats were updating.

## EVALUATE THE CONFIDENCE INTERVAL OF THE AVERAGE PERFORMANCE FOR EACH TEST CASE

Using the aggregated performance data, calculate a 95% Confidence Interval for each test case using the number of observations along with the standard deviation computed from the available executions. Without getting deep in the statistics; this is basically the amount of time +/- between the current computed average and the *actual true average performance* for a given test case. To increase the relevance of the results to the business stakeholders, the times in milliseconds were converted to "hh:mi:ss" format to increase readability. Since the confidence intervals are reported in actual time, another measure of the interval relative to the average execution time for the test case is required. A variation of several minutes may be acceptable on a large test case running more than an hour, but be totally unacceptable on a small test case running under two minutes. In order to quantify this difference, divide the confidence interval by the average execution time for the test, to provide a relative measure of how out of confidence the current estimate is in relation to total time required for that test case. While neither of these measures on their own provides enough information to evaluate the accuracy of the average performance, the combination of the two allows the tuning efforts to focus on the actual outliers within the sub-set.

| Report Id | Sample | Average | Minimum | Maximum | StdDev | 95% Conf. | Relative |
|---|---|---|---|---|---|---|---|
| 4389084 | 30 | 5,595 | 2,517 | 11,883 | 2,471 | 884 | 15.80% |
| 4449490 | 30 | 27,124 | 5,257 | 45,759 | 14,269 | 5,106 | 18.83% |
| 4258152 | 30 | 307,397 | 148,857 | 393,427 | 76,007 | 27,198 | 8.85% |
| 4380454 | 30 | 22,089 | 12,892 | 33,693 | 6,886 | 2,464 | 11.15% |

*An example of the aggregate data and derived calculations (done in milliseconds) for each test case across all iterations.*

| Report Id | Sample | Average | Minimum | Maximum | StdDev | 95% Conf. | Relative |
|---|---|---|---|---|---|---|---|
| 4389084 | 30 | 0:00:06 | 0:00:03 | 0:00:12 | 0:00:02 | 0:00:01 | 15.80% |
| 4449490 | 30 | 0:00:27 | 0:00:05 | 0:00:46 | 0:00:14 | 0:00:05 | 18.83% |
| 4258152 | 30 | 0:05:07 | 0:02:29 | 0:06:33 | 0:01:16 | 0:00:27 | 8.85% |
| 4380454 | 30 | 0:00:22 | 0:00:13 | 0:00:34 | 0:00:07 | 0:00:02 | 11.15% |

*The same data expressed in human readable format for increased clarity.*

## CONTINUE ITERATIONS UNTIL YOU ACHIEVE THE DESIRED CONFIDENCE INTERVAL

When deciding if additional iterations are required, identify the number of test cases without a reliable estimate of average performance. In our case, this meant tests with confidence intervals greater than 30 seconds with variance in average performance of more than 10% of the average execution time. Since the report generation in the system was queued, we decided that a difference in performance of 30 seconds or less would not be detected by end users and that larger variations in execution time were okay, as long as the variation in execution time was less than 10% of the average performance. The proper settings for these thresholds need to be determined based on the environment being tested and the usage of the application. In our case, at least one third of the sub-set did not have a reliable average of the estimated performance after the initial runs. However, as the number of iterations approached 20, the exceptions had stabilized at less than five and those tests did not appear to be converging toward a statistically reliable average. At that point, we had produced the best benchmark possible, given the shared SAN and further executions would only extend the timeline for the project without providing any tangible quality benefit. There were also a couple of occasions along the way when an external event caused an entire iteration to take longer than normal, we found that it was better to include these observations in the benchmark, rather than suppressing them, since the confidence interval was significantly impacted by the number of observations.

| # of Outlier Reports | 3 | Cycle Time | 9:47:08 (hr:mi:se) | | | | | |
|---|---|---|---|---|---|---|---|---|
| Confidence > 30 secs | | | Average | Minimum | Maximum | StdDev | 95% Conf. | Relative |
| Relative > 10% | | Average | 0:02:21 | 0:01:48 | 0:03:14 | 0:00:19 | 0:00:05 | 3.46% |
| | | Minimum | 0:00:03 | 0:00:02 | 0:00:04 | 0:00:00 | 0:00:00 | 1.10% |
| | | Maximum | 1:01:08 | 0:42:40 | 1:15:02 | 0:08:07 | 0:02:04 | 18.45% |
| | | StdDev | 0:05:32 | 0:04:07 | 0:07:13 | 0:00:49 | 0:00:12 | 2.19% |

*An example of the aggregate data used to look across all test cases and determine if additional iterations were required.*

## PROPOSED ENVIRONMENT

With a solid benchmark for the current environment, it's time to implement the proposed change and generate a second benchmark for comparison. This phase of the validation can require a great deal of time to complete, depending on the scope of the changes being evaluated and their impact on the application, doing this in an agile way by generating statistics which are "good enough" allows for a quicker release of the architecture improvements. If possible, validate the performance before converting all of the data into the new structure, eliminating the manual effort required to keep both architectures in synch. When this approach was initially developed; we were upgrading the database behind the suite of applications from Oracle 8i to 10g, changed the partitioning structure and increasing the underlying block size for the largest data tables. Given the difficulty in switching between versions within the testing environment, a solid benchmark under 8i was crucial before starting the benchmark under the 10g version. In contrast, while evaluating the static/dynamic database split with the latest upgrade; all that was needed to flip back and forth between environments was a JDBC URL change within the JMeter framework. This allowed us to spend a lot less time upfront on the initial benchmark and begin the tuning effort sooner. Both approaches can be valid; it all depends on the specifics of the architecture change and the difficulty in switching between environments.

### PERFORM INITIAL ITERATIONS OF INDIVIDUAL TEST CASES AND CAPTURE TOTAL TIME FOR EACH

Perform a limited number of iterations initially in the proposed environment; then work through the performance issues and make the required application changes. While a few iterations may not provide a statistically reliable estimate of the average performance, it only takes a couple of runs to shake out the worst performance issues. The statistics calculated for each test case are identical to those calculated above for the test cases within the benchmark; including the confidence interval and relative performance. Depending on the scope of the change, the first iteration may not complete within a reasonable time, cancel it if necessary, tune the longest running test cases and start again. Basically, the number of iterations depends on how many should complete overnight; setting the stage for another day of looking at the test cases with the worst performance increases, identifying and tuning the problem queries, then checking in the application changes and starting the process again.

### COMPARE TO CURRENT BENCHMARK AND FOCUS ON EXTREME OUTLIERS

Take the two benchmarks that are *good enough* and lay them next to one another to look at the performance differences; focusing on the absolute difference +/- in average execution times across both architectures, along with the delta "(new-old)/old" to provide a relative measure of the difference in performance. Focus on the most extreme differences within the sub-set, selecting a number that's large enough to provide some variety, since a fix to one test case may resolve several others, without having so many tests to review that it delays more forward with a new build and generating a new benchmark. In our case, that meant starting with an actual difference of more than 5 minutes with at least a 10% relative change and working down from there. Reducing the absolute time threshold until it approached the 30 second mark; continuing to adjust the cutoff to highlight the dozen test cases with the worst performance. In addition to the comparison above, review any test cases where the percentage difference in performance is more than one standard deviation above the average of all test cases; this helps to identify tests with very large deltas, even if the absolute time difference is below the threshold. Another aggregate statistic which is useful, is the comparison of the confidence intervals for the individual test cases across both benchmarks, determining if the interval from the proposed environment is less than or equal to the one from the current environment, providing another measure of the similarity in performance.

| | | | | |
|---|---|---|---|---|
| # of Outlier Reports | 6 | # of Reports within Confidence | | 68 |
| # of High Deltas | 17 | | | |
| (more than 1 StdDev above average) | | | Difference | Delta |
| Difference > 300 secs | | Average | 0:01:42 | 131.94% |
| Delta > 10% | | Minimum | 0:00:00 | -90.77% |
| | | Maximum | 0:45:09 | 2144.27% |
| # of Improved Reports | 10 | StdDev | 0:04:52 | 215.23% |

*An example of the aggregate data used to look at the differences across all test cases to determine which cases required further analysis*

### EVALUATE INDIVIDUAL QUERIES WITHIN THE OUTLIER TEST CASES TO MITIGATE ISSUES

With a single test case under consideration, it was now possible to execute the test again in both architectures and capture the execution time for each SQL statement generated by the test case. Compare the two logs side by side by cut/paste them into

Excel or some other tool, then calculate the difference in time on a statement by statement basis. In our case, there were generally only a handful of statements within the hundreds that were generated which were at the root of the performance difference. Once the problem statements have been identified, compare the execution plans and actual performance for these queries within the two environments to identify the optimizer hints or other changes required to achieve comparable performance.

| | | | | | | |
|---|---|---|---|---|---|---|
| TRUNCATE TABLE G_GEOTRADE_<br>-- ^63^20 | 63 | TRUNCATE TABLE G_GEOTRADE_<br>-- ^78^20 | 78 | 15 | 23.81% | 0:00:00 |
| INSERT INTO T_VALUE_LIST_80042<br>-- 1SELECT DISTINCT S.BASE_CL_<br>-- ^297^21 | 297 | INSERT INTO T_VALUE_LIST_80042<br>-- 1SELECT DISTINCT S.BASE_CL_<br>-- ^219^21 | 219 | -78 | -26.26% | 0:00:00 |
| INSERT INTO G_GEOTRADE_DATA<br>UNION ALL SELECT /*+ INDEX(B)*/<br>-- 2372^26563^22 | 26563 | INSERT INTO G_GEOTRADE_DATA<br>UNION ALL SELECT /*+ USE_NL(L E<br>-- 2372^31454^22 | 31454 | 4891 | 18.41% | 0:00:05 |
| INSERT INTO G_GEOTRADE_DATA<br>UNION ALL SELECT /*+ INDEX(B)*/<br>-- 1^18578^23 | 18578 | INSERT INTO G_GEOTRADE_DATA<br>UNION ALL SELECT /*+ USE_NL(L E<br>-- 1^194801^23 | 194801 | 176223 | 948.56% | 0:02:56 |

*An example of the side by side comparison of the SQL log's to identify which queries required tuning.*

## REPEAT THESE STEPS UNTIL PERFORMANCE IS WITHIN DEFINED SERVICE LEVELS

Keep looping here until all of the issues have been addressed and the performance is comparable across both architectures. Avoid the temptation to focus successive iterations on specific tests with past performance issues. In order to continue to compare the times across the benchmarks, the entire sub-set of test cases need to continue to be executed to maintain a consistent environment. As an added benefit, this provides a way to quickly identify tests which are negatively impacted by the tuning efforts. In our case, we generally found that the fixes which addressed the worst issues within one test case, also corrected issues within other test cases, even those which were well under the detection threshold. However, the opposite is also true, when fixes to one test case with a given set of prompts actually causes issues with other tests, which is another reason that more frequent executions under the updated code base, is essential to stay on track and avoid having to rollback too many changes once an issue surfaces.

## FINAL COMPARISON

Once the tuning is finished and the performance of the application in the proposed environment is comparable to the current environment. It's time to summarize all of the information available to report back to the business stakeholders seeking approval to proceed with the proposed change. This can be difficult, since no one has the time to sit down and go through all of the numbers; the results need to be boiled down to a couple of PowerPoint slides that anyone can understand.

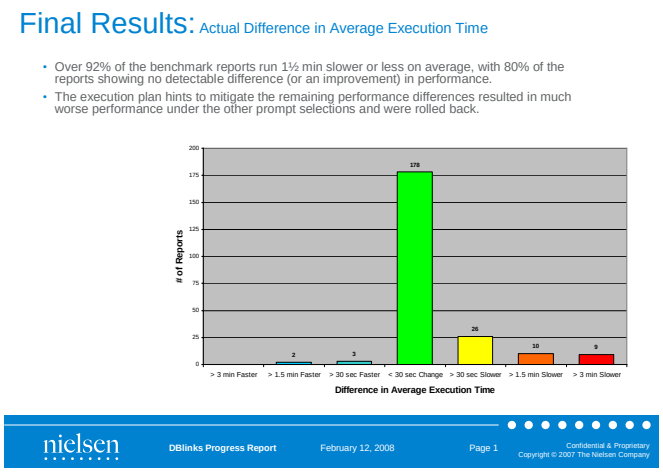
### REGENERATE BOTH BENCHMARKS WITH NEW CODE

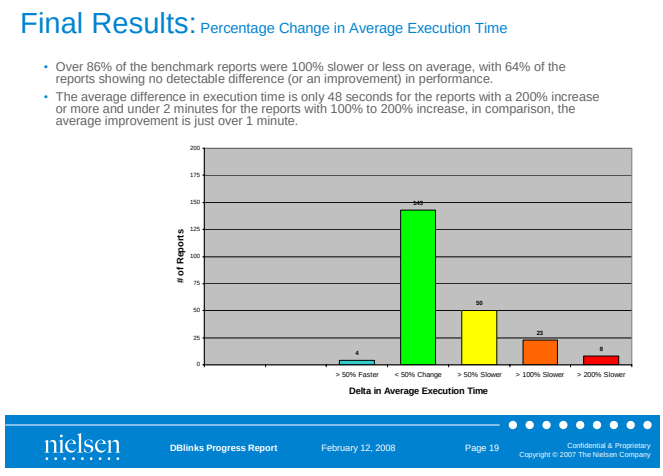
First make sure that there are two solid benchmarks available for comparison, since the application was changing along the way it's best to repeat the initial benchmark with the latest code base, if possible. The benchmark in the proposed environment needs to be strengthened beyond the small number of iterations used for tuning. Reliable estimates of average performance in both environments are essential to the success of the final comparison. A benchmark is solid once the majority of the test cases fall into the desired confidence interval and the remaining exceptions shows no sign of converging toward a reliable estimate of the average performance.

### SUMMARIZE THE RESULTS FOR THE BUSINESS

This is probably the most difficult part of the entire process; finding ways to summarize the performance differences in as few slides as possible without obscuring the risks associated with the architecture change. One way to summarize the results of the individual test cases is to group them into ranges by both the actual time difference and the percentage change; showing a normal distribution of the test cases with the majority of the cases on the center line, with no detectable difference in performance.

For the actual differences in average execution times, we chose ranges of plus or minus 30 seconds, 30 seconds to 1.5 minutes, 1.5 to 3 minutes and more than 3 minutes. For the percentage changes in average execution time, we settled on plus or minus 50 percent, 50-100 percent, 100-200 percent and more than 200 percent. Different ranges may be required, depending on the range of the performance differences within the application under evaluation and the type of change being implemented, the key is to attempt to show a normal distribution with a small number of exceptions at the outside edges.



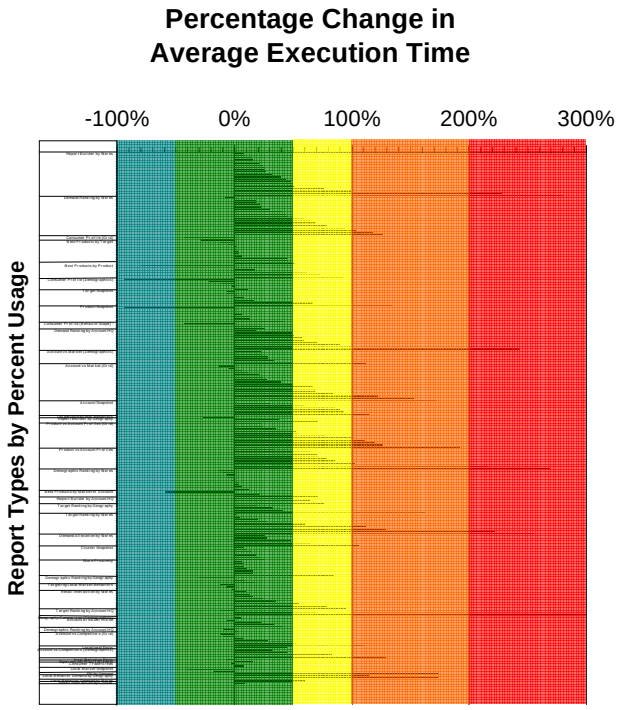




*Examples of two of the slides from our latest project showing the overall impact of the architecture change.*

In addition to these two summary charts, group the individual test cases in order of their usage by end users and plot the individual percentage change in average execution time on a chart to show that the differences in performance impact within a given set of tests. This confirms that the worst performing test cases are not indicative of a general problem, but related to differences in the ways that the current and proposed architecture deals with various loads, based on the end user selections.

# Performance across Report Types

- Looking at all 228 individual prompt sets across the 43 report types tested; 64% are contained within the green or blue zones.
- The average difference in execution time is only 48 seconds for the 8 entries stretching into the red zone.
- The average difference in execution time is under 2 minutes for the 23 entries within the orange zone.
- The average improvement in execution time just over 1 minute for the 4 entries in the blue zone.
- The 9 remaining outliers running 3 minutes longer or more under the DB links are split evenly within the yellow and orange zones with an average percentage change of just over 100%.







*An example of the third slide showing the percentage change across the 228 individual test cases.*