The Oracle Database Upgrade Motivators: Why You Should Move to 11g.
SQL Plan Management, Baselining
Presented by John Watson, Oracle Certified Master
Moderated by Dave Anderson
Watch the Complete Tutorial at SkillBuilders.com/sql-plan-management-tutorial

John Watson:
What I want to talk about now is SQL Plan Management. [also known as baselining]

Now, I hope you were all with us a couple of weeks ago in our first session, because in our first session, Dave [Anderson] and I presented to you fantastic features, the general thrust of which is - if you let us upgrade your database - we would enable, we would configure a couple of features; specifically, the result cache and some optimizing enhancements to parallel practicing. We will enable those features and like magic, your database will run fantastically compared to the way it ran before.

Now, the question that I was expecting to get from one of you but didn't was someone saying something along the lines of "Hey, guys, the last time I upgraded my database, it was awfully disastrous. How can you guarantee it ain't going to be disastrous again?"

Well, I took a gamble. None of you asked me that, so I get to ask the question myself. So how can we guarantee that after upgrade things actually will get better or, more importantly, won't get worse? Because this is a major problem with the Oracle environment and always has been? How do you ensure stability? That's where SQL Plan Management really comes into its own.

The issues of change control, Oracle databases have always been plagued with problems of wildly erratic performance and that works in any sense, that people are absolutely terrified to making any changes whatsoever.

Now, SQL Plan Management is meant to be the tool that will never - [actually] - it's not meant to be - it *is* the tool that means that [statement regression] will never be a problem again. It

will solve the major issues of change control. We can give you an absolute guarantee, if you configure this as part of your upgrade, that SQL statement execution will never regress.

What sorts of problems, what sort of issues, can cause SQL statement plans to regress, performance to degrade?

New statistics: Well, after we upgrade your database, you're going to be getting new statistics every day and they're calculated in a very different way and calibrated in ways unlike they were previously. The usage of new statistics should be a good thing. They should be able to optimize their best possible chance of developing a good execution and plan. Sometimes they are disastrous.

Schema changes: Things like new indexes, materialized views, again should be good, can be bad. Parameter changes, all the tuning you do. You fiddle around with parameters to improve performance. It can be good, can be bad. Upgrade, of course, is the most obvious issue, even platform operation. Any of these changes are meant to be for the better, but they can be seriously bad.

I was just looking at one as an example. I have to be very careful with showing you examples, by the way, because of reference sites, reference and industry sites. I can't show you anything from our current customers. So this is the best example I have of new statistics causing a most extraordinary change in performance.

This is a Statspack report. This is going back a while. It's a fairly old Statspack report from an E-Business Suite database. Some of you – well, some of you may even recognize this query. Basically, you could look at E-Business suite. This query here selects vendor name, item description, vendor products and so on and so forth.

At the time I generated this report, we had 19 executions of past statements and the total executions line 1,015 seconds. So that's about , what, 12 seconds per execution, just on the edge of being acceptable because it's a pretty horrible statement, but you're getting an answer in

12 seconds.

I got a phone call one morning basically saying, “Our whole database is completely broken.” Why? Because this statement went from 12 seconds to three hours, and here is the proof of it in this collected stats..

I generated another Statspack report on the current version of the database to see what on earth was going on, and we found that that one statement has gone to – one execution of the same statement took 8,845 seconds. And that was an overnight change; so one day 12 seconds, next day 8,800 seconds. No changes at all, or so the end user said. That’s the sort of issue we want to prevent.

Now, why did that particular case happen? Just for completeness, that was new statistics; had just analyzed the database. The new statistics meant a new execution plan and that was disastrous, and that is the sort of thing that we intend to prevent.

So the idea then, in order to make sure that plan changes, if they work at all, will work as planned. So often, you do everything when you’re testing, development testing. You do it on live and it’s different. We want to make sure the production really will not have a degrade and of course unplanned changes like the one I just gave you, they’ve got to be stopped completely.

The issue here is largely Oracle’s dynamic parsing, and this is where the second topic we hoped to have time to do today but [we will do in next few weeks].

Oracle’s dynamic parsing does work against stability and the adaptive cursor-sharing feature, a very powerful new feature, can sometimes make the situation even worse because the parsed statements possibly, through every execution that’s most likely every two or three days and the nature of the application, the repars can produce a drastically different plan and dramatically different performance.

The alternative is to force stability. If you force stability by using the old feature of [Oracle]

stored outlines, then you totally freeze the system. You will never get any improvement at all. So what we want is a mechanism whereby changes are controlled in a fashion that means a change will be accepted if it's for the better but rejected if it's for the worse, and that's done through what we call "Managed plan evolution."

I'm going to go to a demonstration at this point to give you an idea of what we're trying to achieve.

So what we do is done through a couple of optimizer of parameter changes. Now, this is why I so like this facility. I, as DBA, can enable this declaratively. As database administrator, I ensure that performance will never degrade. You don't have to do anything to your software, not even having to work your program. You need do nothing at all. I can do it all declaratively.

So I'll just create a table. [demonstration] We want a decent size table. Well, that's going to have about 70,000 rows. Now, our critical parameters that we enable to do this.... [there are] two new parameters with oracle 11g, optimizer_capture_SQL_plan baselines and optimizer_use_sql_plan_baselines.

The concept of a baseline: A baseline is a history of the plan execution. By default, this facility is disabled, so you get more dynamic parsing and possibly quite dramatic changes from one day to another.

I'll separate the list of parameters to true. Capture SQL plan baselines equals true. This instructs the database to capture copies of all execution plans that are ever used. This must be controlled. It can take up a considerable amount of space, but we would plan for that in the space budget. But from now on, I'm going to capture all execution plans.

The second parameter I'll change will be use them [i.e. use the plans we've captured]. Once I enable the use of the baselines, Oracle will now monitor execution plans and if a new plan comes in, it will not be used. A plan will not be used until it has been proven to be better than the plan that was used previously.

That's the critical point. Changes are simply not permitted unless they are known to be better. If it [the SQL plan] turns out to be worse, it will not be accepted at all. So look at this table I've got here, you know, "create table tab1 select * from all_objects". There's no index on that table. There is no option whatsoever about how to run that statement [i.e. a full table scan must be used].

I'll just set autotrace on and I'll explain so we can see the execution plan we are going to get. But I know what plan I'm going to get. I know the table (no index], there is only one option and that would be full table scans. I'll select - I'm going to add a comment. I'll put my initials in there so that I can identify the statement easily later.

If I select the MAX object ID from table called tab1. There's only one way to run that statement - all table scan. So back on our 70,000, the highest locator ID is 77,000. All table scan, the only way to do it.

All right. What do I have in the baselines? Well, the execution plan for that statement should indeed have been captured. There is a view, dba_sql_plan_baselines, and that will show - for each SQL statements - identified by columns sql_text or by plan_name.

We see the plan is being saved under a certain name and we see that the plan has been accepted for use. Only plans that have been accepted will ever be used. So I'll take SQL_text and plan_name and whether it's been accepted, from the view [dba_sql_plan_baselines] and I'll put in text to get just the statements I want and leave the text blank. That's my identifying string..

[demonstration at 10:55]

John Watson, 11:22
There's my statement and there's the SQL plan that's been saved and it's been accepted for use. The first plan ever developed into a statement must be accepted because you have to run the plan sometime.

If I make a change – I'm actually going to make a change that will – any change at all will do in principal, but I'll make change and we'll see what happens when I rerun the statement, does the change I make have any affect at all. Now, the change I will make will be a fairly basic one. I'll just create an index from previous table [on column object_id].

Well, that index really ought to help that statement.. Well, is it going to be used? Well, let's just find out. I'll set autotrace on this file.

I'll re-run my statements now. There we go, full table scan. So my statements are identical. I've created a new index. I am still using the old execution plan. Why? Because I've enabled SQL plan baseline. The baseline was used.

So, you can see the execution plan is absolutely frozen because at this precise moment, Oracle doesn't know if the new version, the improvement, is for the better or for the worse. Take a look at what's in the SQL base, the SQL management base and see that we now have two copies. That's the execution plan that was first used and is still being used repeatedly with just the full table scan. Then there's another one there, and is this plan is accepted? No. So Oracle [database] has realized, "Hey, man, if I come up with a new plan, I ain't gonna use it because I don't know if it's better or worse."

What do we do next? We have to test it. What we would do at this stage is create an automatic job that would test whether new plans are better or worse than old plans. How can we do that? We use a package, dbms_spm, SQL Plan Management. This package is how we administer this entire system and in particular, we test whether new execution plans are better or worse, and I'm going to [demonstrate] this interactively.

Of course, for a real customer, we do a set of jobs for this automatically. We use the dbms_spm.evolve_SQL_plan_baseline procedure. It's overloaded. We evolve a plan giving it one statement or you can give it a set of statements or we can alternate the whole process. The evolved process tests new plans to see if they are better or worse than the plans that were used

previously.

[demonstration at 14:34]

I'm doing this interactively now but of course, in the real world, we would always do it through an automatic job that selects dbms_spm-dot. The test procedure is to evolve SQL plan baseline.

Dave Anderson at 15:00
DAVE: This may come clear as part of your presentation. You may want to do it that way, but what matrix is used to determine if a statement plan is better?

JOHN: Wall clock time [i.e. elapsed time]. It's nothing to do with – it's the most basic metric that you've got. It doesn't go by things like I/O.. It doesn't go by CPU usage. It goes by how much time from the moment the call was received by the server process to the moment the fetch back to the user process is completed. That's wall clock time, which is almost certainly what you would want.

So, to test this one – but you'll see actually use the statistics it uses coming in the results of this – that was a good question, actually. It's very timely. It fits perfectly with this.

Now, to test a plan, we use the evolve_SQL_plan_baseline procedure and I'll test this new one here. This is the new plan that was developed after I created the index and has not been accepted for use. So we test this and see what we get out. [demonstration continues]

All right. So that was the plan that was developed and here's the report. Plan name, you can give it a time limit of course, because normally you might find you've thousands of different plans and if you do something as drastic as create an index, it will just materialize even that quickly hundreds or thousands of statements where you get new columns.

Well, the results are pretty dramatic here. Time used, .327 seconds; 520 times better than the baseline plan. Therefore, the plan's been accepted for use and you can see here the elapsed time

coming through.

DAVE: Excellent. I'm sorry, John. I thought you were pausing.
I have another question in the queue when you're ready.

JOHN: I'll take it now.

DAVE: Does the SQL text have to match exactly to make use...?

JOHN: Yes, it does. It's an exact match, because we're talking about reusing cursors. It needs to be a precise match. So, if you're running ad-hoc SQL, well, you're not going to get much benefit. Ad-hoc SQL is almost certainly going to be down on re-parsed every time, so this would not.

This will be in stored procedures and that's the sort of thing, like that example I gave you, in the * Statspack reports. It's coded in stored procedures. It's really the only *.

DAVE: I think this next question will bump into our adaptive cursor sharing discussion in the next session. What if you have skewed data and you want different plans based on different bind-variable values?

JOHN: This is why I had hoped to talk about this at the same time as adaptive cursor sharing. The mechanisms work together and I think probably next time, we'll need to come back to that. In some circumstances, they can work together. In some circumstances, they can work against each other.

To give a very short answer at this point, if you enable this mechanism that I'm describing here, adaptive cursor sharing is in effect postponed until tomorrow because for the first day, you will get one plan and one plan only. The new plans will be developed due to adaptive cursor sharing but not used. The alternative plans wouldn't be tested until probably your run an overnight job, the equivalent to what I've just done interactively. And the next day, you would have a set

of plans available which could be used interactively. It's quite a complex topic, the way they interact, but they do interact and I'll ask that you trust me at his point.

Okay. So back to this - you can see why the topics go together here - we've never proved, or Oracle has tested this and found that my index does indeed give a better plan. Note that this report here doesn't say why the plan is better. You know, there's nothing about it's better because I used the index on material that's new or that bind variables values are difference or whatever.

All it knows is it checked out the statement and found the improvements. As a result of that, if you go back to the view that's showing what's going on, we now find that hey, both plans were accepted. In the case of adaptive cursor sharing, you might find about 20 accepted plans, ones that all have been tested. And from now on, Oracle knows that there are two plans, both of which are available for use and depending on the circumstances, it would choose which one is best, but it can only choose plans to be proven to be superior.

The final check, what's happening now, let's set autotrace on and see if it really does work. I'll rerun that same query and [the query] uses the index. Why? Because there's an accepted plan. And this is a really, really powerful technique.

So coming back to my slide here, just by assessing two parameters to capture plans, enable their use, no plan will ever be used if it hasn't been proven and accepted already to be better. So initially, you get no changes. In effect, you freeze it, but freezing is no good because that means you'll never get improvements. Things won't get worse, but you won't get an improvement either.

That's where we have to evolve the plans. Now, I did it interactively but what we'll normally work towards is a completely automatic process. So we enable the parameters [optimizer_capture_sql_plan_baselines and optmizer_use_sql_plan_baselines] so all plans are captured and saved in SYSAUX tablespace, in the workload repository. New plans - not used. It reverts to the best previous accepted plan, the best in terms of bind variables and so on, you

know, what the circumstances happen to be.

Then we can figure - typically - a daily job. I won't go into details on that [configuring an Oracle job]. But we have to configure automatic testing of new plans. The end result of that is that any statements that have deteriorated will not be accepted, will never be used. Improved plans then become available to for use, but accepted doesn't mean they're going to be used. It could be it becomes possible to use it. You still have dynamic parsing. The end result: Changes are held back until they are proven to be for the better and the end result of all that is we can absolutely guarantee that your performance will never degrade.

With regard to upgrade, of particular importance, this is backwardly compatible with previous releases - up to a point. It's possible to capture all your code on say, release 10g database, then test it on Release 11 [Oracle] database. That way, you can ensure - which is the point of this particular series of seminars – that upgrade will never cause performance problems.

Generally speaking, it's equally powerful and possibly a lot more useful for day-to-day management; parameter changes, statistics changes. The whole process can be automated for guaranteeing that things will only ever get better, and that's why I would strongly recommend that if we upgrade your databases, you'll let us do this.

Back to you, Dave.