

Does the Optimizer Need a Clue?

Ask the Oracles!



Gaja Krishna Vaidyanatha: Let me start with a life-altering philosophical question to my male readers! When your wife—or significant other—gives you a hint, what does it really mean? If you are a smart man, you will answer, “A hint from her is a directive.” If you have answered the above question correctly, you probably understand the Oracle Optimizer very well. A *Hint* in a SQL statement is—in no uncertain terms—a directive to the Optimizer.

But wait—do Hints always work? Let us add some real-life perspective. It is Super Bowl Sunday and you are ensconced in your comfy leather couch, watching the game on TV and eating out of a bag without consciously tasting its contents. You are witnessing the dying moments of the grand finale of the NFL season. Your beautiful wife, who is fixing dinner while keeping tabs on the game, gives you a hint: “Honey, the trash is full!” You hear the hint, but choose to ignore it. Given that the context was inappropriate, you firmly believe that the hint is *invalid*! Your significant other—to your absolute dismay—then plants herself in front of the television and asks, “Honey, did you hear what I just said?”

When your wife or significant other gives you a hint, what does it really mean?

What your wife demonstrated was a real-life example of how to “Explain Plan.” She gave you a hint, but followed up by asking you to “Explain Plan”! The moral of the story: if you introduce a Hint into your SQL, follow up by asking Oracle to “Explain Plan” and verify that the Optimizer is using it.

You may be interested in knowing that the Oracle Optimizer possesses “selective hearing” or “filtering” just as you and I do. There are many situations where the Optimizer registers the Hint that you give it, but chooses to ignore it even if the Hint is syntactically accurate. You can—and should—confirm this by asking it to “Explain Plan.” To flash back to the

final quarter of Super Bowl Sunday—taking the trash out during the dying moments of that game just did not seem to add up. It did not make sense. It was *invalid*! Thus you ignored it.

In a perfect world, we and the Oracle Optimizer may not require Hints. This is because, in a perfect world, we will always possess accurate statistics and the perfect context. The point to note here is that the Optimizer has to make split-second decisions based on previously collected statistics. Nine out of ten times, when an Optimizer picks the “wrong plan,” it does so because of insufficient or inappropriate statistics. And the single most relevant reason when the statistics go bad is in

The Optimizer and we do not live in a perfect world!

the cardinality of column values.

So you may ask, “What if I always computed statistics—using a sample size of 100%—on all of my objects and gave the Optimizer 100% accurate statistics? Will that be the perfect place to be?” The answer is no, not always. In fact, during a recent performance tuning engagement, we found that deleting the statistics on one of the tables and its associated indexes actually made a certain query run faster and consume fewer resources. The bottom line in that particular case was that the Optimizer chose an inefficient join method even when it had access to 100% accurate statistics, and that it did a significantly better job with default cardinality assumptions and dynamic sampling methods. Note that computing statistics using a sample size of 100% may not even be feasible if some objects are very large.

So then—does the Optimizer need a Hint? It sure does! Not always—just every now and then. Why? It is because the Optimizer and we do not live in a perfect world!

Gaja Krishna Vaidyanatha is the principal of DBPerfMan LLC (www.dbperfman.com), an independent consulting firm specializing in Oracle Database Performance Diagnostics & Management.

In recent years, he has worked in a product management role, providing strategic and technical direction to applica-

tion performance and storage management solutions for companies like Veritas Corporation, Oracle Corporation, and Quest Software.

He is the primary author of Oracle Performance Tuning 101 and one of the co-authors of Oracle Insights: Tales of the Oak Table. He has presented many papers at regional, national, and international Oracle conferences. He can be reached at gaja@dbperfman.com.



Guy Harrison: Introduced in Oracle 7, Hints allow the SQL developer to embed instructions in SQL statements that influence the Optimizer's choice of execution plan. When introduced, Hints were a significant improvement on the existing methods of changing execution plans, which essentially involved tricking the Optimizer into choosing a particular plan.

Hints can be either to advise the Optimizer or to override it. In the former case, we tell the Optimizer something about the desired behavior of the SQL. For instance, the `FIRST_ROWS` Hint allows the developer to advise Oracle to optimize for a certain number of rows—for example, for the first “page” of data in a query screen. However, most Hints allow the developer to override the Optimizer—by

specifying a particular index, join order or method, or other optimization technique.

Hints that advise the Optimizer of the developer's expectations regarding the number of rows for optimization or the desired degree of parallelism

are fairly benign, since they do not prevent the Optimizer from choosing any particular plan. Instead they provide the Optimizer with additional factors to consider when choosing between candidate plans.

However, Hints that directly request particular plans often cause problems over time, since they prevent the Optimizer from choosing a new plan when circumstances change. Execution plans will normally change in response to changes in table sizes or data distribution, changes in available indexes, or upgrades to the Oracle software. Hints might prevent the Optimizer from choosing a better plan as circumstances change or—more seriously—prevent the Optimizer from picking a good plan when a change (such as dropping an index) renders the original plan impossible.

In early incarnations of the cost-based Optimizer, these concerns were relatively unimportant, since the Optimizer's success rate was sufficiently low as to require Hints in SQL statements in a wide variety of circumstances. However, with today's Optimizer there are far fewer situations in which the Optimizer will produce a poor execution plan. Furthermore,

The developer has the advantage of true—not artificial—intelligence.

alternatives to Hints now exist—we can use SQL profiles (10g) or stored outlines (9i and subsequent versions) to directly influence or even “freeze” a plan without having to modify the text of that SQL—which allows us to change the execution plan even when we don't have access to the application code.

Despite these advances I continue to believe that the developer must take responsibility for the performance of his or her code and—if necessary—may need to include an

Optimizer Hint to do so. Although the Optimizer is an increasingly sophisticated piece of software, it is not aware of all the application requirements, data semantics, and other relevant information available to the developer. Furthermore, the developer has the advantage of being able to try alternative approaches and has the benefits of true—not artificial—intelligence. In many cases, the developer will be completely justified in directing the Optimizer to follow a specific execution plan.

In conclusion, the increasing sophistication of the Optimizer and the availability of alternatives such as stored outlines and SQL profiles make the use of Hints less of a necessity than in the past. However, Hints are still an important tool in your SQL tuning toolbox.

Guy Harrison is chief architect for database solutions at Quest Software. He is the author of Oracle SQL High Performance Tuning, Oracle Desk Reference, and MySQL Stored Procedure Programming, as well as numerous shorter articles and presentations. Guy can be reached at guy.harrison@quest.com.



Jonathan Lewis: If you design your application perfectly, make all the smart choices with data structures, create all the relevant constraints, generate suitable statistical information about your data, and write carefully crafted code, then you will still find that there are a few statements that need Hints before the Optimizer follows the “best” execution path.

There are many reasons why the Optimizer may need help—including bugs, simple deficiencies in the Optimizer model, and the inherent problems of collecting, storing, and processing the complex statistics needed to describe real-world data—thoroughly described by other authors in this compendium.

Counterintuitively, another reason why the Optimizer may need help is that there are some extremely cunning strategies built into the runtime engine. Consider, for example, the following simple query.

There are many reasons why the Optimizer may need help.

```
select outer.*
from emp outer
where outer.sal > (
  select avg(inner.sal)
  from emp inner
  where inner.dept_no = outer.dept_no
);
```

There are two major strategies that the Optimizer could adopt for this query—create a result set with the structure (deptno, avg_sal) and do a join to the driving table (an un-nest operation), or scan the driving table and execute the sub query whenever necessary (a filter operation). If you want the un-nesting to happen, you could include the UNNEST Hint in the sub query; for the filter operation you could include the NO_UNNEST Hint.

Of course, you might try to avoid using Hints by manually un-nesting—rewriting the query as follows.

```
select outer.*
from emp outer,
(
  select dept_no, avg(sal) avg_sal
  from emp
  group by dept_no
) inner
where outer.dept_no = inner.dept_no
and outer.sal > inner.avg_sal;
```

Alas, if you do this in recent versions of Oracle you might then need to stop the Optimizer from doing a cunning—but possibly catastrophically inefficient—piece of complex view merging by including the NO_MERGE Hint in what is now the inline view (or a NO_MERGE (INNER) in the main query).

But why might you want to control the strategy that the Optimizer chooses for the original query anyway? Because there is a clever trick, known as scalar sub query caching, that can occur at runtime to minimize the number of times the filter sub query is executed—but it is impossible for the Optimizer to know how many times the filter sub query will actually run. (The Optimizer may be able to work out the minimum number of times the filter sub query has to run,

There are many reasons why the Optimizer may need help.

but that's not necessarily a good estimate of the actual runtime activity.)

In this specific example it is likely that un-nesting will be the better option; in other cases it will be less obvious. And if the Optimizer chooses the wrong option, you have to give it a Hint or rewrite the query to make it do the right thing.

But look at the comment I made about rewriting this query. In Oracle 8i, my rewrite with the inline view could be a good idea—in Oracle 9i the inline view might get merged, with disastrous effects on performance.

The same type of issue appears with Hinting—you find a

Hint that seems to solve a problem in one version of Oracle and causes a problem when you upgrade to the next version. (The ORDERED Hint is a good example of this in 8i, and the PUSH_SUBQ Hint is a good example in 9i). The biggest problem with Hints is that they are badly documented; it is almost impossible to find out exactly what each specific Hint is supposed to do, and if you don't know what a particular Hint does, how can you work out why it seems to solve a particular problem?

Hints can be very useful to solve urgent problems, but don't use them as a first resort.

Hints can be very useful to solve urgent problems—but my general advice is (a) don't use them as a first resort, (b) check whether the real problem is in the statistics, (c) if you really need to hint your SQL, you probably need an average of at least one Hint per table to lock in the execution path you expect, and (d) assume that you're going to have to revisit and retest any hinted SQL on the next upgrade.

Jonathan Lewis is well known to the Oracle community as a consultant, author, and speaker, with more than 18 years' experience in designing, optimizing, and troubleshooting on Oracle database systems. His latest book is Cost Based Oracle—Fundamentals, which is the first of three volumes on understanding and using the cost-based Optimizer.



Cary Millsap: I like Tom Kyte's idea that there are good Hints and there are bad Hints (asktom.oracle.com/pls/ask/f?p=4950:8:::::F4950_P8_DISPLAYID:7038986332061). Good Hints, like FIRST_ROWS and ALL_ROWS, give the Optimizer additional information about your intentions that can help it to make better decisions. Bad Hints, like USE_NL and HASH, constrain the Optimizer from choosing an execution plan that might actually be ideal for you.

Bad Hints are like morphine for your database. They're just right for a very specific purpose, but if your application needs them to run, it's a problem.

Bad Hints are just right for experimenting with your SQL to see how different plans perform. Hints let you tell the Optimizer exactly what to do, so you can measure how it acts when it does it.

But habitual bad-Hint use is a bad thing. Especially since version 9.2, the Oracle Optimizer generally makes excellent decisions based upon the information you (or your DBA) give it. Having

Good Hints give the Optimizer additional information; bad Hints constrain it.

When your Optimizer does something “crazy,” don’t reach for the Hints; find out why a good decision-maker has made a bad decision.

the Optimizer is a lot like having a really smart SQL performance expert in your office.

And here’s where I think a lot of people mess up. Imagine that a really smart SQL performance expert is saying that your query’s most efficient execution plan is something you find utterly ridiculous. What would you do? If it really were a smart person in your office, you might at least listen respectfully. But with Oracle, a lot of people just roll their eyes and slam a Hint onto their SQL to constrain the Optimizer from choosing the apparently dumb plan.

The problem is that the Optimizer probably made a smart decision based on the data you gave it. Maybe it chose a stupid plan because you accidentally told it that your 10,000,000-row table has only 20 rows in it. If you just tape your Optimizer’s mouth shut with an INDEX Hint, you may never find the 26 other queries that also use bad plans because of this bad assumption.

So when your Optimizer does something “crazy,” don’t reach for the Hints; find out why a good decision-maker has made a bad decision. The less you can rely on bad Hints, the less time and effort you’ll have to spend hand-tuning individual SQL statements, and the better your odds will be of having stable performance after your next database upgrade.

You can cure your addiction to bad Hints. Start by visiting the AskTom URL that I listed earlier. The full prescription is to read Jonathan Lewis’s outstanding book Cost-Based Oracle—Fundamentals. It covers a tremendous range of information that will help make the Oracle Optimizer your friend.

Cary Millsap is the chief technology officer of Hotsos Enterprises, Ltd., where he serves as an author, teacher, consultant, designer, and developer within the company’s portfolio of software products and educational services. He wrote Optimizing Oracle Performance, for which he and co-author Jeff Holt were named Oracle Magazine’s 2004 Oracle Author of the Year.

Prior to co-founding Hotsos in 1999, he served for ten years at Oracle Corporation as one of the company’s leading system performance experts. At Oracle, he founded and served as vice president of the 80-person System Performance Group.

He has educated thousands of Oracle consultants, support analysts, developers, and customers in the optimal use of Oracle technology through writing, teaching, and speaking at public events.



Chris Lawson: I consider facility with SQL Hints to be of crucial importance to the serious performance specialist. When I help a client screen prospective performance DBAs, I always ask questions about common SQL Hints, such as USE_HASH, or ORDERED. Here’s why: As the size of database tables grow,

it becomes more and more likely that good performance will require a SQL Hint. It’s not that the Optimizer works badly for big databases; rather, the consequences just become much more severe for being a little bit wrong.

Not knowing SQL Hints will seriously handicap anyone trying to improve performance—especially in large databases. For instance, when joining huge tables in a data warehouse, a small mistake in join order can add hours to the runtime of a batch job. As the number of tables in the join increase, the importance of join order (as well as the join method) magnifies.

Not knowing SQL Hints will seriously handicap anyone trying to improve performance.

There’s another reason that Hints are important. For critical batch jobs, it’s important that we achieve consistency in runtimes. Our customers simply can’t tolerate huge variations in runtimes. Adding a SQL Hint is one way to achieve reasonably consistent runtimes. So, for example, if you know that a full table scan is the right choice most of the time, it’s reasonable to lock in that execution plan.

There are a few Hints that I use frequently: PARALLEL, USE_HASH, FULL, NO_MERGE, LEADING, and ORDERED. Note that most of these are related to joins. This is where the Optimizer often needs help. Of course, some Hints don’t really correct the Optimizer, but are used for special purposes. The Parallel Hint is one example. As your database grows, you will become very familiar with the Parallel Hint.

A SQL Hint is often appropriate when you know something special about the data distribution that gives you inside information on the best join order. Even though the Optimizer concludes that you are choosing an inferior join order, you know better, and instruct the Optimizer to choose your plan—even if the Optimizer doesn’t really like your choice.

No Hint is better than the wrong Hint!

Remember—SQL Hints are powerful and oftentimes necessary, but using them incorrectly can be devastating. On one application I tuned, almost all the SQL Hints were inappropriate, and my first step in tuning was simply to remove the Hints! When applying a SQL Hint, remember that you are assuring the Optimizer that you know better than it does. The Optimizer

is willing to trust you, but you take on full responsibility for the consequences—good or bad. So, understand what a Hint does before you slap it on. No Hint is better than the wrong Hint.

Chris Lawson is an Oracle DBA consultant in the San Francisco Bay Area, where he specializes in performance tuning of data warehouse and financial applications. He is a frequent speaker at NoCOUG, and has written for a number of publications such as Oracle Internals, Exploring Oracle, SELECT, Oracle Informant, and Intelligent Enterprise. Chris has held a variety of positions in the IT field—ranging from systems engineer to department manager—and is an instructor for the University of Phoenix. He can be contacted via www.oraclemagician.com.



Dan Tow: Some say that Oracle's cost-based Optimizer has ended the need for manual SQL tuning. This may sound plausible in theory, yet somehow manual SQL tuning has provided 75% of my livelihood for years. There are two common root causes for SQL performance problems:

First, someone may have made a mistake. This might be bad application or database design, missing indexes, a subtle error in the SQL functionality, or a DBA mistake, among others. No Optimizer can solve these problems.

Second, even with a perfect configuration and statistics, the Optimizer is sometimes unable to choose the right plan without a functionality-neutral change to the SQL—usually Hints.

These are both common, real-world causes of poorly performing SQL, with about 20% being of Type 2. These problems happen in the real world, and manual tuning of the worst-offending SQL is a highly efficient means to resolve both types of problems, without wasting time on the 99% of the SQL that already runs well. However, only Type 2 problems bear directly on the question “Does the Optimizer need a Hint?”—so it usually does not.

SQL tuning has provided 75% of my livelihood for years.

There are many reasons why even an excellent Optimizer may need Hints to deliver a good enough plan. Suffice it to say that a human tuner working at length on a short list of the worst-performing SQL may know much more than the Optimizer can see or consider during a sub-second parse step. To ask the Optimizer

to entirely replace human tuning even for just the Type 2 problems described above is to ask it to win—or tie—every time, even against well-trained, experienced human tuners who enjoy enormous built-in advantages.

The uncontroversial “intent” Hints, such as FIRST_ROWS and ALL_ROWS, help even a hypothetical perfect Optimizer, conveying the developers' objective without limiting the

Optimizer's choices. In contrast, the more controversial plan-limiting Hints may help today, but they can prevent future releases of the Optimizer from making even better choices.

Furthermore, a plan constrained by Hints has less freedom to adapt to changes to the data distributions that may make today's good plan terrible a year from now. Plan-limiting Hints are a two-edged sword, but if you limit Hints as follows they do far more good than harm:

First, don't tune with Hints just for minor improvements to the runtime. A fix that yields at least a twofold runtime improvement is a big, juicy “bird in the hand,” compared to the possibility of further improvements representing some fractional “bird” in some future “bush.” It is likely that the evolution of the application will result in changes to the SQL before those improvements materialize.

Hint carefully, but, when justified, by all means, Hint!

Second, when performing manual tuning—which generally is only necessary for the worst SQL—tune only with the intention of creating robust execution plans.

It is easy to choose an execution plan that is sensitively dependent on a dozen assumptions and data points being precisely correct—the Optimizer does it often! This plan may degrade horribly if any of those assumptions or datapoints are even moderately wrong. Such plans are not robust, although they may be technically “optimal”—i.e., fastest—for now. However, in my whole career, I haven't once needed a non-robust plan to get good enough performance. In practice, robust plans usually follow well-indexed paths and nested-loops joins to the larger tables, in a well-chosen join order, and these robust plans almost never need to change!

Hint carefully, but, when justified, by all means, Hint! ▲

Dan Tow has 16 years of experience focused on performance and tuning, beginning at Oracle Corporation from 1989 to 1998, where he headed the performance and tuning group for all of Oracle applications and invented a systematic, patented method (U.S. Patent #5761654) to tune any query efficiently. This method is extended and elaborated in his book, SQL Tuning. Dan lives in Palo Alto, California, and is reachable at dantow@singingsql.com.