When to Use an Index

Cary Millsap/*Hotsos Enterprises, Ltd.*

Executive Summary

When should you use an index? For well over a decade, Oracle application developers have used a simple rule of thumb to help them decide whether to use a non-unique index. However, in our field work, we encounter performance problems caused by using these rules of thumb. In this paper, I explain the following research results:

- A rule of thumb is unreliable if it relies on a break-even row-selectivity percentage to determine whether or not to create an index.
- An index can significantly improve the efficiency even for queries of a one-row table.
- The factor that should dominate your decision about whether to create an index is *block* selectivity, not *row* selectivity.
- You can determine a given `where`-clause predicate's block selectivity by using SQL provided in this paper.
- Column values are often either naturally clustered or naturally uniform. You can use this knowledge to make better decisions about whether or not to create an index.
- Many new Oracle features simplify your ability to store data in a physical order that facilitates excellent performance.

When to Use an Index: the Traditional Advice

In one form or another, the standard advice for whether to use an index has been around at least since Oracle version 5:

Use an index when a query will return less than x percent of a table's rows.

Figure 1 illustrates the notion that some percentage *x* acts as the break-even point for the performance of Oracle index range scan and full-table scan access paths. This plot relates response time *R* (typically expressed in seconds) to the proportion p_r of the table's rows that a given query operation returns.

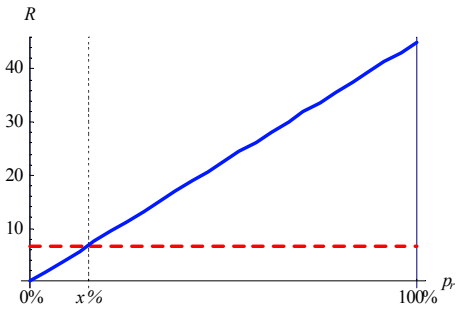

Figure 1. *Response time R in seconds as a function of the percentage p_r of the table's rows that are being returned. The dashed line at R = 6.75 (the red line, if you are viewing this in color) is the response time of a full-table scan. The solid line (blue) is the response time of an index range scan that returns p_r percent of the given table's rows.*

The response time of an execution plan returning r rows via a full-table scan is roughly a constant, no matter whether r is one or the total number of rows in the table. However, the response time of an index range scan increases as the size of the resulting row source[1] increases. The percentage $p_r = x$ is the break-even p_r value at which the response time of a full-table scan and an index range scan are identical. For values $p_r < x$, the index range scan produces better performance. For values $p_r > x$, the full-table scan produces better performance.

There is a big problem, though, with this line of reasoning. Any rule of thumb about indexes is unreliable if it relies on the existence of a break-even percentage like x.

Why the Rule of Thumb is Unreliable

A rule of thumb like "Use an index when a query will return less than x percent of a table's rows" is based on the following observations:

1. If a query operation's resulting row source contains only one row, then an index range scan is more efficient than a full-table scan.
2. If a query operation's resulting row source contains all the rows from a table, then a full-table scan is more efficient than an index range scan.
3. Therefore, there must be some break-even percentage of a table's total row count at which the cost of obtaining the row source via an index range scan is equal to obtaining the row source via a full-table scan. For query operations returning fewer rows than the break-even, the index range scan will be more efficient. For query operations returning more rows than the break-even, the full-table scan will be more efficient.

Our testing and field experience has shown that observation 1 is true, even for very small tables. A query returning one row is more efficient when executed via an index than via a full-table scan, even if the table contains only one row. A lot of people with whom we've discussed this have found this result surprising. The result also contradicts Oracle's specific recommendation that, "Small tables do not require indexes" [Oracle 2001a]. Small tables may not *require* indexes, but indexes on small tables can make your system enormously more efficient and, hence, more scalable.[2]

So, we agree with part 1, but there are big problems starting with part 2. It is sometimes *much* cheaper to read 100% of a table's rows through an index than it is to read them with a full-table scan.

> ***Example:*** Imagine a table called `interface` whose high-water mark is 10,000 blocks deep into the table. Although at some point in history, `interface` had hundreds of thousands of rows in it, today the table contains only 100 rows. These rows are scattered throughout 30 of the table's blocks. Suppose that the table has a primary key column called `id`, which of course has an index (called `id_u1`). And suppose that we needed to execute the following query:
>
> ```
> select id, date, status from interface i
> ```
>
> If we execute this query via a full-table scan, it will require 10,000 Oracle LIO calls. We can rewrite this query slightly to enable Oracle to fulfill the query through an index. If `id` is a number column and all `id` values are non-negative integers, then the following query will produce the desired row set via an index:
>
> ```
> select /*+ index(i id_u1) */ id, date, status from interface i where id>-1
> ```

[1] A *row source* is simply some subset of a table's rows (potentially all of a table's rows, such as in the case of a full-table scan). The output of a query operation is often referred to as the operation's *resulting row source*. In a complicated Oracle execution plan, a child execution plan operation will pass its resulting row source to its parent for further processing (such as joining or filtering).

[2] In Oracle8*i*, we have found that creating and using an index on `system.dual` significantly reduces the number of LIOs required to select its single row. We have observed about a 10:1 associated improvement in response time for queries of `dual`. If your application makes, for example, millions of LIO calls per day against one-row reference tables, then our research suggests that you can conserve millions of LIOs per day and about 90% less CPU for those queries by indexing them.

This query will require fewer than 40 Oracle LIO calls. The response time will be on the order of 10,000/40 = 250 times better if we select 100% of this table's rows via an index instead of via a full-table scan.

There are all sorts of hooks and crooks that we could examine in this example. For instance, if the select clause had contained only `id` or `count(id)` (which can be obtained from information in the index without even visiting the data segment), then the indexed scan would have been even faster still.

So, for any percentage-of-rows-based indexing rule of thumb to be useful in a case like this, it must admit the possibility that using an index might be more efficient than a full-table scan even for queries that return as many as 100% of a table's rows. Figure 2 depicts this phenomenon.

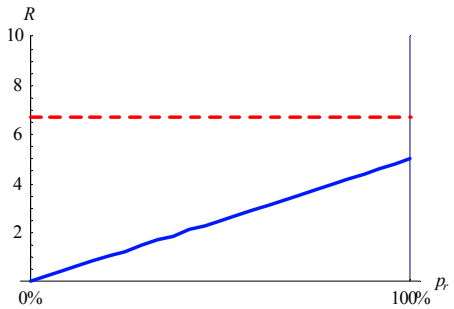

Figure 2. *This plot depicts the situation that exists when a table contains a large number of empty blocks. The index range scan (blue solid line) is faster than the full-table scan (red dashed line), even for a query returning 100 percent of the table's rows.*

There are many cases in which percentage-based rules of thumb are unreliable. There's even a bigger problem with part 3 of the observations listed earlier. This problem will reveal itself as we continue.

The Oddly Evolutionary Character of *x*

A big problem with the indexing rule of thumb is that it's not really clear which value of *x* you should use. If you follow the evolution of recommendations for *x* throughout the history of Oracle documentation, you'll find the following:[3]

Oracle version	Oracle documentation's recommended value of *x*
5	10–15
6	10–15
7	2–4
8	2–4
8*i*	15
9*i*	15

It's worse even than the table lets on. If my memory serves me correctly, an early release of production Oracle7 documentation listed its recommendation of *x* as "1–15 percent." I got a kick out of how wide the range was. To further complicate matters, some of my friends in Oracle Applications development argued very persuasively that in their applications, they observed many *x* values in excess of 40.

Many people presume that the reason that *x* keeps wiggling is that Oracle keeps changing the way the optimizer works. But this isn't actually the reason at all. The reason that *x* has been such a moving target is that the authors of the recommendations have failed to notice the true parameters that drive the break-even value.

[3] Please do *not* confuse this table with charts that ascribe changes in *x* to Oracle kernel modifications. For example, [Niemiec 1999 (38, 318)] indicates that the dominant parameter in the changing *x* values is the Oracle version number. My goal is to explain that the actual break-even is heavily dependent upon other parameters and virtually independent of Oracle version number.

The critical parameter is this: it is the number of Oracle blocks below the table high-water mark that can be skipped by using an index.[4] The way to construct an index creation rule of thumb that makes life easier is to ask the question, "Which execution plan will require the smaller number of Oracle blocks to be visited?"

For any row source with more than one row in it, an index helps to the extent that it enables you to eliminate PIO calls. The number of data-block PIOs that an index helps you ignore depends upon the following:

- How many blocks beneath the table's high-water mark contain at least *one* row that satisfies your query's `where` clause? If your "interesting" rows are distributed uniformly throughout your table, then you might find that using an index is inefficient even for incredibly "good" row selectivity values.

  Example: We wish to optimize the following query:

  ```
  select id, date from shipment where flag='x'
  ```

  The `shipment` table contains 1,000,000 rows stored in 10,000 Oracle blocks. Only 10,000 rows match the criterion `flag='x'`. The row selectivity on the `flag` column for the value `x` is therefore a very "good" 1%. However, the physical distribution of the rows in `shipment` is such that every single block in the table has exactly one row for which `flag='x'`. Hence, we must visit every single block in `shipment` to satisfy this query, whether we use an index on `flag` or not. Therefore, a full-table scan will be more efficient than an index range scan for this query, even though the query returns only 1% of the table's rows.

- Can Oracle fulfill the requirements of a query's `select` clause solely by using the data stored in the index? If so, then the index can eliminate the need to visit the table at all. The columns in an index are usually a small subset of the columns in the indexed table. Hence, the number of leaf blocks in an index is usually much less than the number of blocks below the high-water mark in its associated table. Therefore, scanning even a whole index can be cheaper than scanning a range of a table.

A Parable for Indexers

Let's discover the importance of a concept called *block selectivity* through the use of a story. Here goes…

> Imagine a book called *Brief History of Humanity*, a 1,000-page summary of virtually everything our species has done since we gained the ability to tell about it. Imagine that you're interested in reading about Alexander the Great from this great book. How will you do it? Through the book's index, of course.
>
> The index will tell you on exactly which pages you'll find information about Alexander the Great. You'll probably bookmark the index and then go off on a direct-access search by page number for "Alexander." When you finish one section, you'll refer back to your bookmarked index page to learn where you'll need to turn next to find more. In the end, you'll make one more reference to the index to learn that you have exhausted your list of numbers of pages that contain information that you want.
>
> Now imagine that, unlike regular books, every single word of this book is in the index. In this book's index, you can find the locations of even such words as "the." Let's say now that you're interested in finding the complete list of words in the *Brief History of Humanity* that follow the word "the."By requiring the word that *follows* the word "the," we won't be able to find everything we're looking for in the index; we'll have to visit the actual text to find what we're looking for.

---

[4] Craig Shallahamer, Micah Adler, and I wrote about the identity of the true parameters that influence *x* for *Oracle Magazine* back in 1993 [Millsap et. al 1993]. Craig, Micah, and I may actually be the only people in the world who have read it. After looking at it again recently, I've reached a clearer understanding of why it wasn't one of the more popular papers I've ever worked on. One line of the paper actually said the following:

$$\text{Formula 2: } B > H + \lceil pR/\beta' \rceil + \alpha\left(1-\left(1-p\right)^{\beta}\right.$$

…Not exactly easy reading. Not surprisingly, nobody has ever written me to ask where the closing parenthesis was supposed to have been. (I am pretty sure that our original manuscript was syntactically complete, at least.) The article is also virtually impossible to find, which makes it even more difficult to read. See the *References* section for details.

The extraordinary frequency of the word "the" will probably make the job absolutely miserable, even with an index. "Let's see, 'the'… Ah yes, 'the': page one." Okay, so you'll bookmark the first "the" page in the index. Then you'll flip from the index over to page one. You'll locate the word after the first "the." Then you'll return to the index to find that the next page on which "the" appears is also page one. Back and forth you'll go until you've visited every page in the book dozens of times, and you'll have flipped the book back and forth so many times that you'll probably wear out the binding.

Now, imagine a *Reader's Digest* Large Print for Easier Reading edition of *Brief History of Humanity*. For the sake of illustration, imagine that the entire book is set in 72-point type. *Brief History of Humanity* now has only 20–30 words per page. Although the word "the" is common enough to have appeared on virtually every single page in the regular book, the word "the" is not common enough to appear on every page in the Large Print edition. In this new scheme, the index now has immensely greater utility for our little "find the word after the 'the'" project, because the index now allows us to skip more pages.

Chapter 1

This is 72-point type. The Large Print for Easier Reading edition of Brief History of Humanity would have far fewer words on each standard-sized page.

Mystery Solved

The parameters that influence the usefulness of an index in a range scan that requires a `table access by rowid` are:

- *Oracle data block size*. The bigger the data block size, the more "interesting" a data block will tend to be to more `where` clause predicates, and therefore the less useful an index will be. Bigger data blocks will tend to drive smaller *x* values.
- *Oracle index block size*. The bigger the index block size (which, of course is always the same as the data block size prior to Oracle9*i*), the fewer LIO and PIO operations will be required upon the index, and therefore the more useful an index will be. Bigger index blocks will tend to drive larger *x* values.
- *Row size*. In our parable, our index increased in utility when we made our words bigger relative to the page size. This is analogous to having larger rows in a fixed-size Oracle block. Bigger rows will tend to drive larger *x* values.
- *Data distribution*. Had the text of *Brief History of Humanity* been sorted alphabetically by word, our index would have had extraordinarily good utility, even in the small-print edition and even for the word "the." All of the "the"s would have been clumped together about two-thirds deep into the book.[5] Better physical clustering of table data will tend to drive the value of *x* for your query to smaller values.

Understanding the parameters of index utility solves the mystery of why people can't decide on a good value for *x*.

- When the Oracle documentation researchers wrote the Oracle version 6 tuning guide, they probably used tables like the `scott/tiger` schema's `emp` and `dept` in an Oracle database with 2KB blocks. When they wrote the Oracle7 documentation, they probably tested the same queries as before. But they probably used the "new" 4KB Oracle block size came into fashion with the release of Oracle7. Since the bigger blocks stored more rows than before, the observed value of *x* was lower. Indexes were apparently less useful than they were in Oracle6. The published break-even dropped from 10–15 to 2–4.

[5] Of course, sorting the text of *Brief History of Humanity* alphabetically by word would have destroyed the meaning of the text and, with it, the meaning of any query such as "find the word that follows the word 'the.'"

- The Oracle8*i* and Oracle9*i* documentation is much better on the topic of index utility. Oracle now uses $x = 15$ as a general guideline but mentions that the value "varies greatly." They mention clustering and full-scan speeds as parameters to the variance, but they do not mention block size or row size as parameters of clustering [Oracle 2001a].
- Do you remember our good friends at Oracle Applications development who claimed good results with $x > 40$? Why were they convinced of a number so dramatically different from anything that the official Oracle documentation ever said? It's not difficult to understand their perspective if you think about their environment. First, their tables have huge rows. Many applications tables have in excess of 200 columns per row. Next, for a variety of reasons, the Oracle Applications are a "little slow" to adopt new technologies emanating from the kernel group. Throughout the mid-1990s, they used 2KB-block databases almost exclusively. In fairness, of course, changing the Oracle block size of Oracle's big Applications databases would have been a huge job, not to mention the seemingly insurmountable job of validating SQL execution plans. As it happened, the combination of big rows and small blocks drove the observed break-even x value higher than many other groups had observed.

Now What?

My advice to you is this:

Forget about any *percentage-based indexing rule of thumb.*

There simply is no range of percentages that will give you reliable results. There are queries returning 1% or less of a table's rows that are more efficient with a full-table scan than with an index. And there are queries returning a full 100% of a table's rows that are more efficient through an index. So if you insist upon choosing a value for x, I recommend that you find one that is simultaneously less than 1% and greater than or equal to 100%. Since no such number exists, I recommend that you divert your focus from percentage-based indexing rules of thumb entirely.

Oracle optimizer technology has come a long way since the introduction of the Oracle cost-based optimizer (it's actually quite nice in Oracle8*i*). But it's still up to you to determine which indexes to create. The Oracle kernel will attempt to use indexes that you've created only when it's efficient to do so. But it is a waste of both space and time to create an index that *never* has good utility.[6] So, how should you decide whether to create an index or not? The answer is to look at *block selectivity*.

Block Selectivity

You are probably already familiar with the concept of *row selectivity*. You can define the row selectivity of a given `where`-clause predicate as the number of rows returned by the predicate (r) divided by the total number of rows in the table (R):

$$p_r = \frac{r}{R} \qquad \text{definition of } \textit{row selectivity}$$

You can define the *block selectivity* of a given `where`-clause predicate analogously, as the number of data blocks containing at least one row matching the predicate condition (b) divided by the total number of data blocks below the high-water mark (B):

$$p_b = \frac{b}{B} \qquad \text{definition of } \textit{block selectivity}$$

The distinction between *row* selectivity and *block* selectivity is critical, because block selectivity is almost always worse—often much worse—than row selectivity. As we saw earlier in the `shipment` table `flag='x'` example, it is possible for a predicate to have row selectivity of 1% but a block selectivity of 100%.

You can calculate row selectivity and block selectivity using SQL, as in the following example that we call `hds.sql` [Holt 2002].

[6] …As long as the index is not required to fulfill a constraint definition.

```
rem $Header: /usr/local/hotsos/RCS/hds.sql,v 1.8 2002/01/07 18:12:27 hotsos Exp $
rem  Copyright (c) 2000-2002 by Hotsos Enterprises, Ltd. All rights reserved.
rem  Author: jeff.holt@hotsos.com
rem  Notes: Hotsos data selectivity using a full table scan for the row count.

define v_substr7 = 'substr(rowid,15,4)||substr(rowid,1,8)'
define v_substr8 = 'substr(rowid,7,9)'
define v_over    = 'substr(''&_O_RELEASE'',1,1)'

col dummy new_value v_substr

set termout off heading on pause off

select decode(&v_over, '7', '&v_substr7', '&v_substr8') dummy
  from dual;

set termout on verify off feedback off pages 10

accept p_town  prompt 'TableOwner : '
accept p_tname prompt 'TableName  : '
accept p_clst  prompt 'ColumnList : '
accept p_where prompt 'WhereClause: '
accept p_pgs   prompt 'PageSize   : '

variable fblks number

declare
  tblks  number;
  tbytes number;
  ublks  number;
  ubytes number;
  luefid number;
  luebid number;
  lublk  number;
begin
  sys.dbms_space.unused_space(
    upper('&p_town'), upper('&p_tname'), 'TABLE',
    tblks, tbytes, ublks, ubytes, luefid, luebid, lublk, null
  );
  :fblks := tblks - ublks;
end;
/

col blks  form   9,999,999,999 heading 'Table blocks below hwm|(B)' just c
col nrows form 999,999,999,999 heading 'Table rows|(R)'             just c new_value v_nrows

select :fblks blks, count(*) nrows
  from &p_town..&p_tname;

col bs    form             a17 heading 'Block selectivity|(pb = b/B)' just c
col nblks form   9,999,999,999 heading 'Block count|(b)' just c
col rs    form             a17 heading 'Row selectivity|(pr = r/R)' just c
col nrows form 999,999,999,999 heading 'Row count|(r)' just c

set pause on pause 'More: ' pages &p_pgs

select &p_clst,
       lpad(to_char(count(distinct &v_substr)/:fblks*100,'990.00')||'%',17) as bs,
       count(distinct &v_substr) nblks,
       lpad(to_char(count(*)/&v_nrows*100,'990.00')||'%',17) rs,
       count(*) nrows
  from &p_town..&p_tname &p_where
group by &p_clst
order by bs desc;
```

Using `hds.sql` is reasonably straightforward. However, generating perfect data distribution information for a table can be very expensive. Depending upon your data, this query may run for minutes or hours. This expense is why the

Oracle cost-based optimizer relies on stored statistics instead of analyzing the data itself when it computes or validates an execution plan. The following example illustrates how we use `hds.sql` data.

Example: A system has a table called `po.cs_ec_po_items`. Our goal is to optimize several query suboperations that use `ec_po_id=:v` as a `where`-clause predicate. Should we create an index on `ec_po_id`? We can use `hds.sql` to produce perfect data distribution information about the various values of `ec_po_id`:

```
SQL> @hds
TableOwner : po
TableName  : cs_ec_po_item
ColumnList : ec_po_id
WhereClause:
PageSize   : 20

Table blocks below hwm    Table rows
         (B)                 (R)
--------------------- ----------------
                1,655          299,960
More:

           Block selectivity  Block count    Row selectivity     Row count
  EC_PO_ID    (pb = b/B)          (b)         (pr = r/R)          (r)
---------- ----------------- ------------- ----------------- ----------------
         8            63.50%         1,051             0.54%            1,606
         0            61.81%         1,023             0.52%            1,572
         1            61.27%         1,014             0.49%            1,470
         2            60.66%         1,004             0.52%            1,555
         4            60.24%           997             0.51%            1,529
         6            59.94%           992             0.52%            1,552
         7            59.94%           992             0.50%            1,514
         5            59.70%           988             0.51%            1,536
         9            59.58%           986             0.49%            1,459
         3            57.95%           959             0.49%            1,480
        45            10.76%           178             0.06%              190
        37            10.63%           176             0.06%              183
        .1            10.39%           172             0.06%              180
       2.5            10.33%           171             0.06%              177
       2.3            10.27%           170             0.06%              175
       5.8            10.21%           169             0.06%              178
More: ^C
```

Output of `hds.sql` is sorted in descending block selectivity order. The output routinely has thousands of rows in it, but all the worst-case data—the interesting part of the whole output—is at the top of the report. Therefore, we normally terminate the output of `hds.sql` after a page or two.

Notice that for this table, the row selectivity is excellent for every value of `ec_po_id`. The "worst" row selectivity value is only 0.54%.[7] This means that the only one-half of one percent of the table's rows have the value `ec_po_id='8'`. However, the block selectivity column tells a completely different story. The block selectivity of `ec_po_id='8'` is 63.50%. This means that almost two-thirds of the table's blocks contain at least one row for which `ec_po_id='8'`.

Should we create an index on `ec_po_id`? We could spend half a day or more computing a "back of the envelope" answer by trying to calculate formulas to predict execution plan costs. But the Oracle optimizer can do the work for you. The most accurate and ultimately least time-consuming method for determining the answer is to construct tests on an actual Oracle system. The best way to determine the relative costs of two execution plans is to execute them upon some test data, with `sql_trace=true`. If you want more

[7] You cannot determine that 0.54% is the worst row selectivity for the table from looking only at the output excerpted here. Output from `hds.sql` is ordered by descending *block* selectivity. This does *not* imply a descending order of row selectivity. (For proof, notice that row selectivity values in the output shown here do not appear in strictly descending order.) To determine the worst row selectivity value in an `hds.sql` output set, you must examine the entire output.

detail about the non-CPU work that Oracle performed during the operation, then trace the executions with Oracle 10046 level-8 tracing [Hotsos 2002]. If you want more data about why the optimizer chose the plan that it did, then trace with Oracle event 10053 [Lewis 2001].

With `hds.sql` output, we know the boundary conditions that we need to test. For example, we now know that our tests need to answer the following questions:

- Is `select foo from cs_ec_po_item where ec_po_id='8'` any faster with an index on `ec_po_id`?
- Is the query any faster with an index for `ec_po_id='45'`?
- Is the query any faster with an index for values of `ec_po_id` that have block selectivity less than 1%? (Since the report is sorted in descending block selectivity order, the values with the best block selectivity values are not shown here.)

Your ultimate decision about whether to build the index of course relies on whether the benefit of having the index exceeds the cost of having it. Those costs can include:

- Accidental degradation of other queries' execution plans. In applications that still rely on the Oracle rule-based optimizer, this is a prominent risk. Creating an index to optimize statement *A* can accidentally degrade the performance of some other statement *B*. With cost-based optimization, especially with histograms, thankfully this phenomenon is becoming rarer.
- Increased response time for DML upon the table. However, I have seen people drastically overestimate the importance of this factor. Don't guess about it; profile your DML operations' trace data to find out the true cost.
- Increased space consumption of the index. Once upon a time, the amount of space consumed by an index was a financially important factor in determining whether to build an index. With today's disk prices, it is almost irrelevant.

When you use a tool like `hds.sql`, you will observe one of three patterns:

1. Every value's block selectivity is so good that you definitely do want to create an index for the column.
2. Every value's block selectivity is so poor that you definitely do *not* want to create an index for the column.
3. Block selectivity is poor for some values, but good for others. In this case, you will have to decide whether the utility of the index in the good cases is sufficient to compensate for the cost of having it.

The decisions in cases 1 and 2 are obvious. Of course, situation 3 is probably the one in which you'll find yourself the most often. Users of Oracle's cost-based optimizer prior to release 7.3 faced a tough decision. If they didn't create the index, they risked poor performance for some `where`-clause values; if they did create the index, then they risked poor performance for others. Newer versions of Oracle's cost-based optimizer make life much simpler. These days, if you do your duties with respect to statistics collection[8] then it's much less likely that erroneously creating a low-utility index will inflict torture upon your users.

Example: Imagine a `division` table containing a `id` column with the following data distribution:

[8] This means regularly running `fnd_stats` if you're an Oracle Applications manager, or `dbms_stats` otherwise.

id value	Row selectivity (p_r)	Block selectivity (p_b)
01	89.87%	100%
02	2.34%	7%
03	2.16%	6%
04	1.86%	2%
05	1.42%	1%
06	1.17%	1%
07	0.75%	1%
08	0.18%	< 1%
09	0.14%	< 1%
10	0.11%	< 1%
TOTAL	100.00%	N/A

The data distribution shown here is highly skewed. Now, imagine the following query upon this table:

```
select name from division d where id=:a1
```

Without histograms, a cost-based optimizer might assume that since there are ten distinct `id` values, each id accounts for about 1/10 of the table's rows. This assumption makes it look like a good idea to use an index on the `id` column. And it would be—as long as `:a1!='01'`.

The power of histogram-based optimization is that, properly implemented,[9] a histogram-based optimizer will notice when `:a1='01'` and *not* attempt to use the `id` index. Without histogram-based optimization, the application developer must either (1) optimize the query so that it is efficient if `:a1='01'`, but horribly inefficient otherwise;[10] or (2) you must write procedural logic that uses one SQL statement for common values and another statement for rare values. Oracle General Ledger generates dynamic SQL using method 2 for its Financial Statement Generator functions. It's clever, but it's a mess.

[9] It is a long story, but Oracle histogram-based optimization has a history of painful design tradeoffs. Prior to Oracle9*i*, the use of bind variables prohibits histogram optimization. This is a pity because, in general, an application that does not use bind variables cannot scale to large user counts. In Oracle9*i*, the optimizer does almost what we want: it "peeks" at the content bound to a bind variable and makes a valid histogram-based decision. However, on subsequent executions of a shared SQL statement after the first, each execution gets the plan produced by the session's first optimization. So if the first execution of the query uses `:a1='01'`, then the second (and every other) execution will use the optimal plan for `:a1='01'`, even if the subsequent execution searches for `:a1='07'`.

The good news is that it is not really necessary to use bind variables for `where`-clause predicates that don't have a large value domain. For example, using literal values in a `sex='m'` predicate is okay, because it makes only up to two copies of an otherwise sharable SQL statement in the library cache. However, using literal values in something like `order_id='1289942'` would be catastrophic, because it would run potentially thousands of nearly identical but nonetheless distinct SQL statements through the library cache.

[10] One old-fashioned way to optimize such a query is to rewrite it like this:

```
select name from division d where nvl(id,id)=:a1
```

This technique prevents Oracle from using any index with `id` as its prefix. A more modern way to do it would be to specify a hint, such as:

```
select /*+ full(d) */ name from division d where id=:a1
```

Beginning with Oracle8*i*, we can go one step better: we can optimize the query by manipulating its *stored outline*. This wonderful new feature allows us to optimize the statement *without requiring access to the statement's source code*.

Values are Often not Distributed Randomly

The recent trend in Oracle documentation is to mention the assumption that "rows in the table are randomly ordered with respect to the column on which the query is based" [Oracle 2001b]. This assumption makes Oracle documentation a little simpler to write, but it makes Oracle's advice less useful than it could have been.

You will see, after gaining experience with `hds.sql`, that sometimes a column's values will cluster themselves quite naturally and remain clustered forever.

> ***Example:*** A `shipment` table has a status column called `shipped`, which is `'y'` if and only if an order item has been shipped. Because orders tend to be shipped in roughly the same sequence as they were entered, the `shipment` table has good natural clustering of `shipped='n'` values over time, as is shown in Figure 3. The clustering of `shipped='n'` rows improves the usefulness of an index when searching for `shipped='n'` rows.

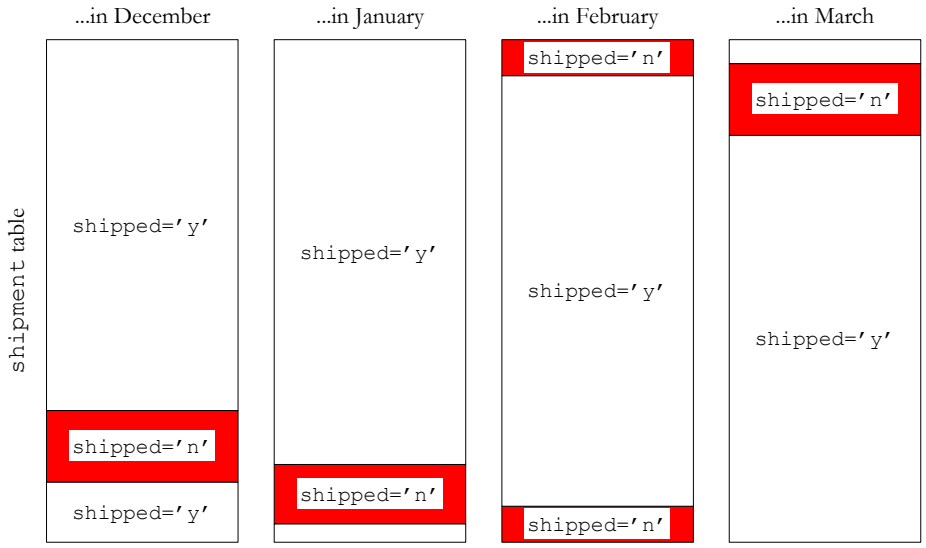

Figure 3. *Status column values can tend to cluster themselves naturally.*

The opposite of a *clustered* distribution is a *uniform* distribution. If a column's value has a true uniform distribution within a table, then the instances of the given value are physically equidistant from each other.

> ***Example:*** An `address` table has a type column called `state`, which contains a customer's two-letter state or province code. In this table's application, there is no apparent relationship between the time at which a customer's row is inserted and the customer's `state` value. Hence, the physical distribution of each `state` value is virtually uniform. Although `state='TX'` is true for only perhaps one row in 30, there are very few blocks in the table that have no `state='TX'` rows. Figure 4 depicts the situation.

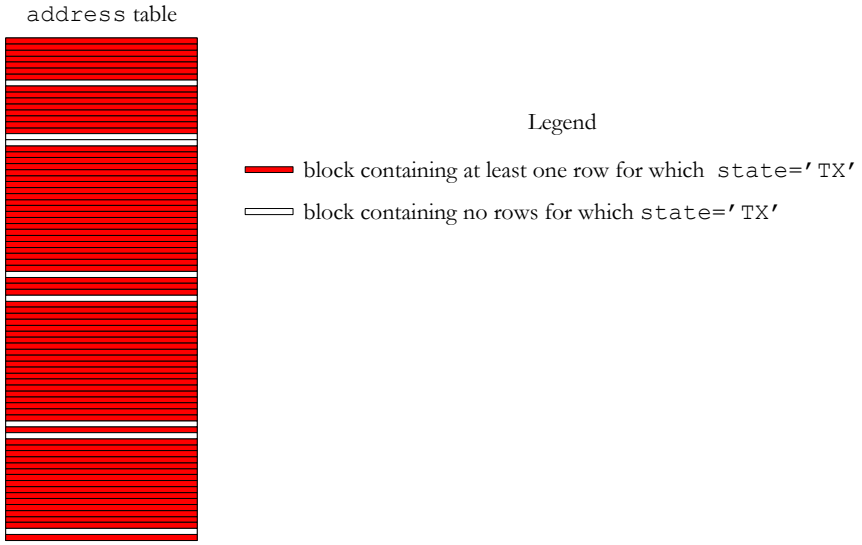

Figure 4. An index on `state` has low utility for `state='TX'`.

Here, using an index on `state` would likely be inefficient for the search of any "popular" `state` value. But if, for example, there were one or more states with far fewer rows than there are blocks in the `address` table, *and* if you search for these states frequently, *and* you are using histograms, then creating an index on `state` will probably help your application.

Status columns can sometimes cluster themselves naturally. But in the absence of any artificial outside influence, *type* columns tend to have more uniform physical distribution. You have several types of influence over a table's physical data storage. You can enforce a specific physical ordering upon your data by using:

- Oracle table and index *partitioning*
- Oracle index-organized tables
- Periodic operational maintenance procedures to delete your rows and reinsert them in a preferred physical order
- Oracle *cluster* segments instead of *table* segments

Don't helplessly assume that your data distribution is random. Find out with `hds.sql`. Each technique to enforce physical ordering brings both benefits and costs to your business. If changing your physical data distribution helps you to simultaneously maximize your company's net profit, cash flow, and return on investment, then do it [Goldratt 1992].

Conclusion

Many sources teach that you should base indexing decisions upon an analysis of a `where`-clause predicate's *row selectivity*. Worse yet, some sources discuss indexing decisions in terms of row selectivity for a whole *column*, which completely disregards the possibility of skew. However, row selectivity is an unreliable basis for index creation decisions. The best way to mitigate your risks is to test actual SQL performance on valid test data. A tool like `hds.sql`, which reports *block selectivity* information, improves the reliability and efficiency of your testing, by revealing the critical column values for which you should test performance.

The Oracle cost-based optimizer simplifies answering the question of whether to build an index because it makes more intelligent index-use decisions than a rule-based optimizer can. But for implementations that still rely upon the Oracle rule-based optimizer, understanding the importance of block selectivity can be vital to Oracle application performance. Once you understand your data's block selectivity characteristics, you needn't take a passive approach

to the physical ordering of your data. Many features added to Oracle since release 7.3 simplify your ability to store data in a physical order that facilitates excellent performance.

References

GOLDRATT, E. M.; COX, J. 1992. *The Goal: A Process of Ongoing Improvement*. 2d ed. Great Barrington MA: North River Press.

GORMAN, T. 2001. *The Search for Intelligent Life in the Cost-Based Optimizer.* Evergreen Database Technologies: *http://www.evdbt.com/library.htm.*

HOLT, J. 2000. *Predicting Multi-Block Read Call Sizes.* Hotsos: *http://www.hotsos.com/catalog/catalog.html*

HOLT, J. 2002. *Hotsos Clinic Tools Pack*. Available to Hotsos Clinic students: *http://www.hotsos.com/catalog/catalog.html*

HOTSOS. 2002. *Sparky™ Data Collector*. Hotsos: *http://www.hotsos.com/products/sparky.html*

LEWIS, J. 2001. *Practical Oracle8i: Building Efficient Databases*. Upper Saddle River NJ: Addison-Wesley.

MILLSAP, C; SHALLAHAMER, C.; ADLER, M. 1993. "Predicting the Utility of the Nonunique Index." *Oracle Magazine* Vol. VII, No. 2 (Spring 1993): 48–53. This article was also printed and distributed within Oracle Corporation from 1992 through 1995. Unfortunately, there is no electronic copy of this article available publicly anywhere. The only way the author knows to view this article is to locate a physical copy of the Spring 1993 issue of *Oracle Magazine*.

NIEMIEC, R. 1999. *Oracle Performance Tuning Tips & Techniques*. Berkeley CA: Osborne/McGraw Hill.

ORACLE. 2001a. *Oracle9i Database Administrator's Guide Release 1 (9.0.1)*. Oracle: *http://download-west.oracle.com/otndoc/oracle9i/901_doc/server.901/a90117/indexes.htm#5640*

ORACLE. 2001b. *Oracle9i Database Performance Guide and Reference Release 1 (9.0.1)*. Oracle: *http://download-west.oracle.com/otndoc/oracle9i/901_doc/server.901/a87503/stats.htm*

Acknowledgments

Once again, enormous credit for this work belongs to Jeff Holt of Hotsos for his research, testing, proof-reading, fact-checking, and actual field use of the principles documented here. Nine years after our card table discussion at the Buckhorn Exchange in Denver, I'd like to thank Craig Shallahamer and Micah Adler again for the good conversations about the principles of index selectivity. Thank you also to Steve Adams, Jonathan Lewis, Bjørn Engsig, Dominic Delmolino, Mogens Nørgaard, and Zach Friese for their constructive feedback during review. Finally, I am grateful to the students of our Hotsos Clinic who have helped us refine the message about index selectivity.

About the Author

Cary Millsap is a limited partner of Hotsos Enterprises, Ltd., a company dedicated to improving the self-reliance of Oracle system performance managers worldwide through classroom education; information, software, and services delivered via *www.hotsos.com*, and consulting services. He is the inventor of the Optimal Flexible Architecture, creator of the original APS toolkit, a Hotsos software designer and developer, editor of *hotsos.com*, and the creator and principal instructor of the *Hotsos Clinic on Oracle® System Performance Problem Diagnosis and Repair*.

Mr. Millsap served for ten years at Oracle Corporation as a leading system performance expert, where he founded and served as vice president of the System Performance Group. He has educated thousands of Oracle consultants, support analysts, developers, and customers in the optimal use of Oracle technology through commitment to writing, teaching, and public speaking. While at Oracle, Mr. Millsap improved system performance at over 100 customer sites, including several escalated situations at the direct request of the president of Oracle. He served on the Oracle Consulting global steering committee, where he was responsible for service deployment decisions worldwide.

Revision History

9 January 2002: Original revision produced to accompany live presentation at *IOUG-A Live! 2002*.

29 January 2002: Reformatting and minor editing.

21 July 2002: Minor editing.