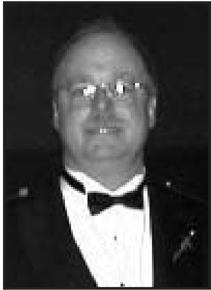
Understanding Indexes



By Tim Gorman, SageLogix, Inc.

Oracle's implementation of indexes works very well and is quite clever, improving on the basic designs set down decades ago. There seems to be an impression that they don't work very well or they need constant maintenance, and that is just not true.

For certain, they are not a panacea for all performance problems, as there are indeed situations where the use of indexes will actually result in poorer performance. The purpose of this article is to help you anticipate and correctly diagnose situations where indexes are (or are not) working to the benefit of application performance, and will describe several situations where indexes are not appropriate or are ineffective.

Please be aware that Oracle does not leave us stranded in bad situations. For each of the conditions where a problem can exist, Oracle usually has developed an alternative to help. What's more, there are unambiguous methods to detect the problems so you can apply these alternatives.

There are a fair number of myths about indexes in popular circulation, some of which include:

- Over time, indexes tend to become "unbalanced," requiring that they be rebuilt on a regular basis for optimal performance
- Indexes supporting monotonically ascending data values (i.e.,, sequences and timestamps) tend to deteriorate over time, requiring that they be rebuilt on a regular basis for optimal performance
- One of the reasons for separating tables and indexes to separate tablespaces is to permit I/O to each to be processed in parallel

In particular, there is altogether too much rebuilding of indexes going on, fed partly by the fact that Oracle has made enhancements that make it easy to do this, but also fed by the belief that it is necessary.

This article will debunk each of these myths in turn, during the course of explanation. But, more than mere critique, the intent of this article is to provide well-rounded understanding, so that one can not only put knowledge into application, but also diagnose adverse results.

So, we will start with a brief description of the architecture, construction, usage, and maintenance of indexes, culminating in a list of weaknesses. Then, the remainder of the article will detail each particular shortcoming, including how the particular weakness is detected, with appropriate alternatives for resolution.

One important note: When I use the term "index" by itself, I am referring to B*Tree indexes, which is the default indexing mechanism used by Oracle. Bitmap indexes, as an alternate form of index, are referred to as "bitmap" indexes.

Index Architecture in Oracle

Indexes are optional data structures, associated with tables and clusters, that are used for speeding access to rows of data and, as a side-effect, can be used as a mechanism to enforce uniqueness of data values in one or more columns.

Most of the benefit of an index comes during data retrieval, by improving response time for a query. The rest of the benefit comes when data is being changed and an index that is created to enforce unique data values prevents non-unique values from being entered to the table.

Otherwise, as an optional data structure, indexes represent additional processing and storage overhead when performing inserts, updates, or deletes to the data in the table. Data can always be retrieved from a table by simply scanning the table until the desired rows are found. Indexes take up additional space in the database, separate from the table with which they are associated. When indexed columns are changed, then the associated index entries are also changed, thus making the change more expensive. So, when is this additional processing and storage overhead worthwhile?

Think in terms of a problem set, the table to be searched, and a result set, the specific rows to be returned. Indexes work best when retrieving a small result set from a large problem set. Cut out all the fancy terminology and lengthy explanations, and that is the basis of all insight.

So, what does that mean? Well, given the fact that we are discussing two items: a result set and a problem set, each having two attributes, small and large. We can create a matrix to quickly summarize the four possible alternatives:

	Small result set	Large result set
Small problem set	Indexes GOOD	Indexes BAD
Large problem set	Indexes EXCELLENT	Indexes BAD

Roots, Branches, and Leaves

The hierarchical tree-structure of B*Tree indexes consists of:

- "Leaf" nodes, which contain entries that refer directly to rows within tables;
- "Branch" or "decision" nodes, which contain entries that refer either to "leaf" nodes or to other "branch" nodes;
- A single "root" node, which is a "branch" node at the top or root of the tree structure.

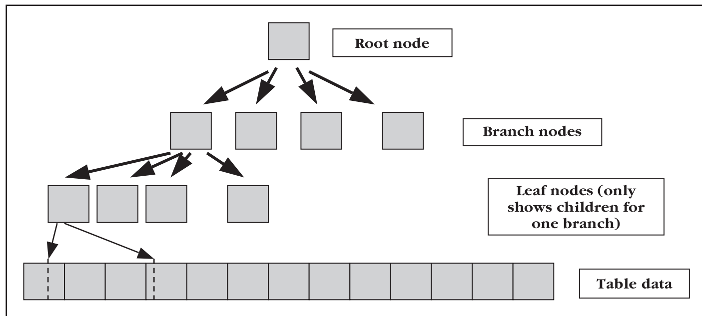


Figure 1: The logical representation of B*Tree structure

Each entry in a "branch" block (including the "root") consists of a row with two columns. The first column is the maximum data value of the indexed data values in the child block. The second column is a four-byte data block address or DBA, which points to the next block in the next lower layer.

The number of entries in a "branch" block is determined partly by the Oracle block size (i.e., parameter DB_BLOCK_SIZE) as well as the length of the partial data values. Like so many things in Oracle that are variable-length, there is no formula to accurately derive this number and the only way to know for certain is to dump an index that already exists and simply count.

Each entry in a "leaf" block also consists of a row with two columns. The first column is the entire data value for the index entry, for one column or a concatenated value from multiple columns. The second column is a full ROWID, which is the physical address of the data row within a table. If the index is created on a non-partitioned table or if the index is a LOCAL partitioned index, then the form of ROWID used is the old six-byte "restricted" ROWID left over from Oracle7. If the index is a GLOBAL index (i.e., a non-partitioned index created on a partitioned table or a non-equi-partitioned index), then the form of ROWID used is the newer ten-byte "extended" ROWID introduced with Oracle8.

So, to get a rough idea of how many rows an index might be able to reference, let's use the example of an 8,192-byte (i.e., 8k) database block, which is common. Roughly speaking, assume that roughly 8,000 or those 8,192 bytes are available for storing index entries in each block.

Let's assume that we are indexing a single DATE column on a table, and that this is a very large non-partitioned table. DATE datatypes are one of the very few fixed-length datatypes in Oracle, so this will make our example a little easier to calculate. DATE datatypes are always seven bytes in length: one byte each for century, year within century, month within year, day within month, hour within day, minute within hour, and second within minute.

So, starting from the bottom of the tree-structure, in the leaf nodes, each index entry occupies a slot or row within the database block. Each row or index entry has two to three bytes used for a row header (i.e., flag and lock bytes), and each column within the row has one byte for data length, and then the data itself. So, it may look like this:

- Row header (three bytes)
    - ✧ Column 0 length (one byte to store length value of "7")
    - ✧ Column 0 data (seven bytes to store a DATE datatype value)
    - ✧ Column 1 length (one byte to store length value of "6")
    - ✧ Column 1 data (six bytes to store restricted ROWID value)

So, each index entry is about 18 bytes in length. We had about 8,000 bytes of space in the block, but the storage parameter of PCTFREE is default to 10 percent, so we can only insert into 90 percent of the 8,000 bytes or about 7,200 bytes. At 18 bytes per index entry, we can store about 400 index entries in each leaf block.

So, if our table has 100,000,000 rows, then we'll need about 250,000 leaf blocks to store all those index entries.

Now, how big does the supporting branch/root structure above those 250,000 leaf blocks have to be?

Each branch entry stores a maximum data value and a data block address (DBA), which is a four-byte subset of a ROWID. In branch blocks, we only need to point to entire blocks, not individual rows within blocks, so a DBA is sufficient.

So, we might have:

- Row header (three bytes)
    - ✧ Column 0 length (one byte to store length value of "7")
    - ✧ Column 0 data (seven bytes to store a maximum DATE datatype value)
    - ✧ Column 1 length (one byte to store length value of "4")
    - ✧ Column 1 data (four bytes to store DBA value)

So, each branch index entry is 16 bytes. Using the same math as above, each branch block can store about 450 branch index entries. So, that means we need 556 branch blocks above the 250,000 leaf blocks. So, our index has its first BLEVEL.

Now, can a single root block reference all 556 branch blocks? Apparently not, since a branch block can only reference 450 children. So, two more branch blocks are needed above the first BLEVEL of 556 branch blocks, comprising our second BLEVEL. Now, we can have a single root block manage those two branch blocks, so the single root block becomes our third BLEVEL. So, to support our index of a DATE column on a non-partitioned table of 100 million rows, we can use three branch levels:

- BLEVEL 1 = one branch block
    - BLEVEL 2 = two branch blocks
        - BLEVEL 3 = 556 branch blocks
            - Leaf level = 250,000 leaf blocks
                - Table rows = 100,000,000 rows

A Familiar Example

The everyday telephone book is an excellent example of a B*Tree index.

Consider the full range of numbers in the 10-digit telephone dialing system currently in use in the US, Canada, and several Caribbean countries to be rows in a "table" in our analogy.

In order to call a specific terminal in this 10-digit telephone dialing system, you could start at 000-000-0000 and call each number, one number at a time, until the highest number of 999-999-9999, eventually finding the person or persons you are seeking.

However, for everyday use, this would make the phone system unusable. Of course, telemarketers have high-speed dialing systems for performing just this type of FULL table-scan, so some people actually can use this method.

But for the rest of us, an index is necessary. That index is the everyday phone book.

The "root" node is the alphabetic index on the outer spine of the book,

continued on page 34

Understanding Indexes continued from page 33

while the first (and only) level of "branch" nodes are the partial data values at the top of each page. Finally, the entries on each page represent a "leaf" node.

We will refer back to this example time and again though out the paper, as many situations present with Oracle's implementation of B*Tree indexes are illustrated perfectly in the phone book example.

Another Familiar Example

When Oracle is accessing the blocks in an index, how is this done?

There are two types of read I/O performed by Oracle server processes when accessing datafiles:

- Random-access, single-block reads (represented by the wait-event "db file sequential read")
- Sequential-access, multi-block reads (represented by the wait-event "db file scattered read")

The sequential-access multi-block I/O is used for FULL table-scan access, when there is a need to gulp down large numbers of blocks very quickly, often in parallel.

However, that is not the case with indexed access.

Recall please the diagram of the tree-structure of an index. Obviously, the index is not stored in that hierarchical representation. Instead, the structure is flattened out into a series of blocks, just like anything else in an Oracle datafile. So, the sequence-pattern of blocks stored might look something like this:

|root|branch|branch|leaf|…|leaf|branch|leaf|leaf|…|leaf|branch|leaf|leaf|…|l eaf|branch|…

…and so on. So, there is a need to hop back and forth in this structure, randomly accessing individual blocks, one at a time.

In many ways, the I/O used to access indexes is like a children's treasure hunt. Come again? How is that? First, some background…

Sequential or Scattered

Why is a *random-access* operation known as "db file *sequential* read" and a *sequential-access* operation known as "db file *scattered* read"? Isn't that backwards?

Perhaps, when it is looked at from this perspective…

But the wait-events are named more for the way that the I/O calls themselves are made, rather than for their footprint on the I/O subsystem. The random-access single-block reads are performed *sequentially* one right after another, synchronously. This is because the descent, from root to branch to leaf, of a tree-structure hierarchy is a performed by reading a block in order to find out where to go next. Thus, we cannot read several blocks together, because it is possible that the next node we will visit will be *behind us*. Hence, the naming with the word *sequential*, since we have to progress sequentially from one block to another.

On the other hand, when performing sequential-access multi-block reads for a FULL table-scan, there is no unpredictability involved. We know exactly which blocks we are going to access, so the only question is getting them all read quickly enough. To do that, we can employ parallel processing and asynchronous operations. Hence the naming with the word *scattered*, since we can scatter dozens or hundreds of I/O requests to the operating system, and just sort the results when we get them back.

A Treasure Hunt

So, remember I/O on indexes to be like a children's treasure hunt. You have directions to the first set of directions, perhaps hidden under a rock in the woods. You follow the directions to that location, pick up the directions stashed there, and read them. They will direct you to another set of directions, perhaps stashed in the fork of a tree. You follow the directions, find the next set of directions, read it, and they will direct you somewhere else.

There is no chance for parallel processing here. You cannot send ten people into the woods and expect them to get anywhere. You have to move from spot to spot to reach your goal.

Likewise with indexed access in Oracle. When you want to use a particular index, then the data dictionary table SYS.IND$ has two columns which locate the segment header block of the index. The information in the SYS.IND$ table is typically cached into the "row cache" in the Oracle system global area (SGA). Reading the segment header block will yield the location of the root block of the index. Reading the root block of the index will yield the correct branch block to go to. Reading that branch block will yield the location of the correct leaf block to go to. And so on…

> **Myth-buster:** Separating tables and indexes to different tablespaces is a good idea, because tables are more important than indexes from a database recovery perspective. You can rebuild indexes, but a lost table is lost data.
>
> But over the years, an idea has arisen of somehow optimizing performance by enabling "parallel reads" to blocks in the table and in the index. As you can tell from the treasure hunt analogy, this is not possible for individual sessions. Many sessions, maybe.

Index Maintenance

So, accessing them is one thing, but how are the danged things built? Essentially, they are either created against an already-populated table or they are built row-by-row as the table is populated.

Building/rebuilding an index on a populated table. This is much easier to describe.

When an index is built (using a CREATE INDEX command) or rebuilt (using an ALTER INDEX REBUILD command), the process performs the following major steps:

1. Query the source data and populate leaf blocks first, fully populated and packed
2. Build the tree-structure above the leaf blocks, branch level by branch level, upwards toward the root

Plain and simple…

Index maintenance during transactional activity. This is more difficult to describe. Have you ever watched a super-highway being repaired or rebuilt as it is being used? Many of the challenges are the same.

Starting off empty. When an index is first created upon an empty table, it consists of a single leaf block.

As rows are inserted into the table, corresponding index entries are populated into the single leaf block of the index. When that block becomes full, then the index must grow to add another block. But, you can't have two leaf blocks side-by-side in an index tree-structure. So, the single leaf block morphs itself into a single branch block, which now becomes the first BLEVEL and becomes the root block. The leaf index entries are then moved to two other blocks, which become leaves.

Why does the single leaf block become the root, forcing all of its index entries to be moved to other blocks? I believe that it is because the location of the root of the index is stored in the data dictionary and it is easier to move the leaf index entries to another block than to update the location of the root block in the data dictionary.

Split or overflow? Even from the filling of the first block in the index, a decision has to be made: shall we split the contents of the current block, or shall we overflow into another block?

Let's say we fill the index block to its default maximum of 90 percent full. When we wish to add one more index entry, we don't have room.

In a table, we would simply *overflow* to the next block, starting to add rows there. That is OK to do in a *heap-organized* structure, where there is no inherent order to the rows.

But an index has to be managed differently. It is not heap-organized, but rather the rows in an index are sorted by their data value, in ascending order. So, depending on the value of the next index entry, it might need to be inserted somewhere into the middle of the list of sorted entries in the block. Insertion into the middle of a sorted list does not lend itself naturally to *overflowing* into the next block. Instead, first *splitting* the contents of the block, keeping the lower half of the sorted list in the current block, and then moving the upper half of the sort list into the new block, and then inserting the new index entry, makes more sense.

However, if the new index entry to be inserted into the block is to be the new highest value in the block, then the whole operations of first *splitting* the block and then inserting the new row does not make sense. In this case, simply allowing the new high value entry to simply *overflow* into the next empty block makes more sense.

So, the algorithm is pretty easy to understand. When an index block fills and the next entry to be inserted would be the highest data value in the block, then simply allow that new entry to *overflow* into the next higher block. However, if the next entry to be inserted would not be the highest value in the block, then first *split* the contents of the current block, sending the upper half of values to the next block, then insert the new index entry into whichever block is appropriate.

Managing Space Using Splits and Overflows

This algorithm has an important downstream benefit for space management in the index.

As indexes are not heap-organized structures, but instead can be thought of as a sorted list when you traverse across all of the leaf blocks, the nature of the data values in the index can determine whether the index is populated evenly or not.

If the data values in the index are randomly generated, ranging back and forth unpredictably across the entire range of sorted values, then only allowing index entries to *overflow* would leave behind only fully-populated blocks. This would not work well with randomly generated values, as random data would force those fully-populated blocks to be split up anyway, to make room for lesser data values inserted later. In fact, the data does not even have to be randomly generated. It only needs to not be monotonically ascending in nature. Under any data conditions other than monotonically ascending data values, performing block *splits* allows the back-filling of data into the middle of the body of the sorted list of the index, with minimal maintenance.

When the data is monotonically ascending, like a system-generated sequence number or a timestamp, then and only then do *overflows* from one block to another become advantageous. If the data is completely predictable, and we know for certain that the next data value will be higher than the last, then and only then is it safe to use the simple *overflow* to another block. That is because we know that we do not have to worry about back-filling data into previous blocks, in the middle of the sorted list. When we are constantly pushing to the right-hand side of the ascending-sorted list, then it is safe to use the *overflow* method.

So, the algorithm for *split-vs-overflow*, besides making immediate sense for determining where to insert the next index entry and how exactly to go about it, also makes perfect sense based on the probable nature of the data. Of course, there is no guarantee that, just because the next index entry to be inserted was the highest value in one block, so all data values are monotonically-ascending. It could be a fluke. But when you insert hundreds of entries into a block, and the last one inserted just happens to have the highest data value, then it is fairly probable that overflowing to the next block is the right choice, otherwise splitting is definitely the right choice.

> **Myth-buster:** So, indexes built upon columns with monotonically-ascending data values, such as sequences and timestamps, do not need to be rebuilt frequently in order to keep them from becoming sparsely populated.

Sparseness

As Jonathan Lewis noted in his paper about "unbalanced indexes" (see "Sources and References" section at the end of this paper), B*Tree indexes never become unbalanced. But the phrase "unbalanced" itself is ambiguous, so he goes on to explain exactly what this means.

"Balance" in this context means that every data value is the same distance from root node to leaf node. So, as one descends the tree structure from root to leaf, it is always the same number of "hops."

However, this is not the only measure of balance.

What about the distribution of entries in the leaf nodes? Instead of looking at how the index is structured from top to bottom (or vertically), how about looking at the picture horizontally, across the leaf nodes?

Oracle indexes grow efficiently, but they do not shrink at all. When a row is deleted from a table, the corresponding index entry is not removed. Instead, the ROWID portion of the index entry is merely set to a NULL value. This allows the index entry to be reused easily.

However, if a leaf node fills with such NULLed index entries, shouldn't it be removed from the index and released for reuse? Logically, that makes sense, but Oracle has not chosen to do this. Instead, as mass deletions occur, associated indexes simply become sparser and sparser.

A sparse index is an inefficient index. Indexes exist to reduce the number of I/Os. By definition, a sparsely-populated index consumes more blocks than it needs to, forcing sessions to perform more I/Os to use it.

> **Myth-buster:** Indexes can deteriorate or become less-efficient over time, but this would be due to sparseness due to data deletions, not some inherent flaw in the way Oracle manages or indexes.

The Problem

So, the effectiveness of an index can deteriorate over time, but not due to it becoming "unbalanced." Rather, the deterioration can happen because the index is becoming more and more sparsely populated. Most often, this sparseness occurs due to row deletions from the associated table, resulting in un-reused index entries in the index.

continued on page 36

Understanding Indexes continued from page 35

Detecting the Problem

Fortunately, the problem is relatively easy to detect.

The ANALYZE INDEX … VALIDATE STRUCTURE command populates a view named INDEX_STATS. The contents of this view are visible only from the session from which the ANALYZE INDEX … VALIDATE STRUCTURE command was executed, and the view never has more than one row. This one row shows information from the most recent ANALYZE INDEX … VALIDATE STRUCTURE command executed.

Of particular interest are two columns in the INDEX_STATS view:

- BTREE_SPACE – the total number of bytes used by the index (excluding block header and overhead)
- USED_SPACE – the total number of bytes used by valid branch entries and index entries (excluding block header and overhead)
- PCT_USED – the ratio of USED_SPACE divided by BTREE_SPACE

The key is to watch the value of PCT_USED. For most indexes where the PCTFREE storage parameter is allowed to remain at the default of 10, then the maximum initial value of PCT_USED is 100 percent - PCTFREE (i.e., 100-10) or 90.

```
SQL> create index xx1
  2         on stats$sqltext(hash_value, piece);
SQL>
SQL> analyze index xx1 validate structure;
SQL>
SQL> select btree_space, used_space, pct_used, blocks, lf_blks, br_blks
  2  from index_stats;

BTREE_SPACE USED_SPACE   PCT_USED     BLOCKS    LF_BLKS    BR_BLKS
----------- ---------- ---------- ---------- ---------- ----------
   12540252   11236231         90        416        384          1
Now, if we reset the PCTFREE storage parameter from the default of 10 to a
value of 5, we see a difference:
SQL> create index xx1
  2         on stats$sqltext(hash_value, piece)
  3         pctfree 5;
SQL>
SQL> analyze index xx1 validate structure;
SQL>
SQL> select btree_space, used_space, pct_used, blocks, lf_blks, br_blks
  2  from index_stats;

BTREE_SPACE USED_SPACE   PCT_USED     BLOCKS    LF_BLKS    BR_BLKS
----------- ---------- ---------- ---------- ---------- ----------
   11888812   11235868         95        384        364          1
Please note that the value of PCT_USED has increased from 90 to 95,
corresponding to the decrease in PCTFREE from 10 to 5.
SQL> create index xx1
  2         on stats$sqltext(hash_value, piece)
  3         pctfree 30;
SQL>
SQL> analyze index xx1 validate structure;
SQL>
SQL> select btree_space, used_space, pct_used, blocks, lf_blks, br_blks from
index_stats;

BTREE_SPACE USED_SPACE   PCT_USED     BLOCKS    LF_BLKS    BR_BLKS
----------- ---------- ---------- ---------- ---------- ----------
   16123172   11238289         70        512        494          1
```

Correspondingly, increasing the PCTFREE storage parameter to 30 leaves only 70 percent of the block available for storage, reflected by the new value of PCT_USED.

In an index in which sparseness is a problem, the value of PCT_USED would be significantly less than 100 percent - PCTFREE. Using a default value of PCTFREE of 10, leaving a maximum initial value for PCT_USED of 90, we might see PCT_USED descend toward a value of 50 or even to single digits.

However, a warning about using ANALYZE INDEX … VALIDATE STRUCTURE: While it is running, it obtains a "DML Enqueue" (i.e., lock type = 'TM') on the table itself, not just the index being analyzed. The reason for locking the table is because the original purpose of the ANALYZE INDEX … VALIDATE STRUCTURE command is not merely to populate the INDEX_STATS table with useful information. That is actually a side effect. Instead, the primary purpose of the command is to validate that all of the index entries correspond to valid rows in the table, as well as validating that all rows in the table have corresponding index entries. In order to perform this validation, the command locks the table against any changes.

Thus, running ANALYZE INDEX … VALIDATE STRUCTURE blocks any INSERT, UPDATE, or DELETE statements against the table.

It seems a shame that we cannot obtain the information in INDEX_STATS any other way, but c'est la vie…

Resolving the Problem

There are two things to do if PCT_USED drops toward 50 or less. You can:

- Drop the index and then re-create it
- Rebuild the index (became available in Oracle7 v7.3.3)
- Coalesce the index (became available in Oracle8*i* v8.1.5)

Rebuilding an index. Since the ALTER INDEX … REBUILD command became available in Oracle7 v7.3.3, there is not much need to ever drop an index and then recreate it.

The ALTER INDEX … REBUILD command does several things more efficiently than CREATE INDEX:

- It uses the leaf blocks of the existing index as the source for the new index.
  - ✧ The leaf blocks of the existing index are a much smaller data structure than the table, so there is less I/O to perform
  - ✧ A *fast full scan* operation is performed against the existing index to retrieve the leaf blocks. This fast scan is much like a FULL table-scan, and while it does not guarantee to retrieve the leaf blocks in sorted order, then contents of the leaf blocks themselves are already sorted. Therefore, the amount of sorting by an ALTER INDEX … REBUILD command is much less than the amount of sorting by a CREATE TABLE command
- Since Version 8.1.6, an ALTER INDEX … REBUILD command can include the ONLINE clause, allowing the index to be rebuilt while changes are being made to the existing index. This capability simply does not exist in the CREATE INDEX command

There are several similarities between the ALTER INDEX … REBUILD and the CREATE INDEX command, however:

- Both can utilize parallel execution and direct-path (a.k.a. bulk-load) insertions using the PARALLEL clauses, improving performance by enabling more CPU and memory resources to be utilized simultaneously
- When using the parallel clause, both can minimize the generation of redo logging using the NOLOGGING clause, improving performance by cutting out this additional processing
- Since version 8.1.5, both commands can use the COMPUTE STATISTICS clause to generate computed cost-based optimizer statistics, which

eliminates the need for the additional processing needed by an ANALYZE or DBMS_STATS command.

Coalescing an index. The ALTER INDEX … COALESCE is a purely transactional operation, which means that it does not interfere with other transactions, so it is implicitly an ONLINE operation. The command does not reclaim space for use by other objects, but it does coalesce index entries in the leaf blocks so that they consume less of the blocks already allocated to the index. Therefore, COALESCE improves query performance by packing leaf blocks (as well as their supporting branch blocks) tighter, so that the index is no longer sparse.

Like the CREATE INDEX and ALTER INDEX … REBUILD commands, ALTER INDEX … COALESCE can use the COMPUTE STATISTICS clause to generated computed statistics for the cost-based optimizer.

However, the COALESCE command cannot use the PARALLEL or NOLOGGING clauses. The ONLINE clause is also verboten, but unnecessary because COALESCE is already implicitly on "online" operation.

Summary: Dealing with Sparseness in an Index

An index can deteriorate over time due to sparseness, not due to "unbalancing." The most common reason for sparseness is row deletion when deleted data values are not reused, which is always the case with monotonically ascending data, but can also be the case with other types of data.

The condition of sparseness is easily detected by running the ANALYZE INDEX … VALIDATE STRUCTURE command and then querying the INDEX_STATS view, but be aware that the ANALYZE command locks not only the index but also the entire table against any changes, so use it carefully on busy systems.

If sparseness is detected, it can be resolved by either rebuilding or coalescing the index.

Selectivity, Distribution, and Clustering of Data Values

Remember the earlier point about indexes being most efficient when they are retrieving a small result set from a *large problem set*?

Well, what happens when the data forces a *large result set* to be retrieved?

In other words, what if there are only two data values in an indexed column, each with 50 percent of the data in the table? No matter which of the two data values you specify, the index is going to have to retrieve a large result set, and that just won't do.

The Problem

Why is a large result set such a problem?

The best illustration might be our two analogies: the *telephone book* and the *treasure hunt*.

Poor selectivity of data values. Let's say we're trying to use a phone book for any city in the U.S. to look for the name of John Smith. "Smith" is a popular surname in most U.S. cities, easily spanning several dozen pages. Even worse, "John" is also a popular given name in most U.S. cities. The numbers of "Smith, John" and "Smith, J" entries in any large U.S. city are liable to number in the hundreds or thousands.

Using the phone book to find the first entry for "Smith, John" will be easy. But choosing amongst the pages and pages of similar entries could be difficult. Unless you know his street address as well, you are almost guaranteed to end up scanning a lot of wrong numbers before you find the right one. Think about this problem as an example of a *large result set* from a *large problem set*. Even from a *small problem set* (i.e., a small town or village), we would still have a problem.

In contrast, most U.S. cities are unlikely to have even one person named Zbigniew Zhang, much less more than one. So, retrieving this *small result set* from either a *small* or a *large problem set* (i.e., from a phone book in a small town or a large city) is likely to be an easy matter

OK, enough of the phone books for now. To further illustrate the evils of a *large result set*, consider the analogy of the treasure hunt…

An index with three branch levels (i.e., BLEVEL = 3) means that the typical initial probe for a table row is going to take five logical reads: one read on the root block, one read on the next level branch block, and one read on the lowest level branch block. Finally, one more read on a leaf block, and then at last a read of the table block containing the row.

Think of that as a treasure hunt with five steps: root, branch, branch, leaf, and table row.

Now, when retrieving a huge result set, let's say 500,000 rows, it's like a treasure hunt with 1,000,005 steps. First, we probe to the first row as described earlier. That's five steps. Then, to get the next row, we have to go back to the leaf block to find the pointer to the next row, then follow that pointer to the table row itself. Two more steps, for a total of seven. And so, for 499,998 more rows. Talk about two-stepping!

In contrast, simply performing a FULL table scan on the table, which might consist of 4,000 blocks, could be performed in only 500 read operations (assuming eight blocks per read operation). 1,000,005 read operations versus 500 read operations.

When it comes to large result sets, indexes are just additional processing overhead, even on queries.

Uneven distribution of data values. This problem is just as serious as poor selectivity, but it can be more difficult to detect. After all, the results may be inconsistent, depending on which data value is popular, resulting in a large result set, and which data values are unpopular, resulting in a small result set.

When data values in the indexed columns are unevenly distributed, it is bad to use the index to retrieve popular data values (those associated with many rows) while it is good to use the index to retrieve unpopular data values (those associated with few rows).

A good example might be a column GENDER in the student database at the Virginia Military Institute. The student body is overwhelmingly male, but there are a few female students. Using an index when retrieving the records for female cadets, a small result set, might be useful, while using an index when retrieving the records for all male cadets, a large result set, would be disastrous.

Clustering. Tables are heap-organized data structures. Rows are inserted into heaps willy-nilly, with no regard for organization. This is OK, because we expect indexes to impose some kind of sorted order.

However, we might experience some extremely bad luck in how the data in the table is located. A test case probably works best on illustrating this problem, as opposed to a prose description…

First, we create a simple table for our test case:

continued on page 38

Understanding Indexes continued from page 37

```
SQL> create table t1
  2  (
  3          c1      number,
  4          c2      number,
  5          c3      number
  6  )       tablespace tools;
```

Table created.

Then, we populate the table with 100,000 rows.

```
SQL> begin
  2          for i in 1..100000 loop
  3                  insert into t1 values(mod(i,187), i, mod(i,187));
  4          end loop;
  5          commit;
  6  end;
  7  /
```

PL/SQL procedure successfully completed.

Please note that the modulus function "MOD()" is used on the data values. The purpose of this function in this context is to impart a particular characteristic on the inserted data, namely a cycling nature. Since we are feeding the MOD function an ascending sequence number (from one to 100,000), the remainder it returns will cycle repeatedly from 0 to 186, providing the following data pattern:

0,1,2,3,4,5,…,184,185,186,0,1,2,3,…,184,185,186,0,1,2,3,…,184,185,186,0,1,2,3,…

So, as rows are inserted into table blocks, it is likely that each block will receive at least one occurrence of each of the 187 possible data values. So, now let's create an index on column C1 and analyze everything:

```
SQL> create index i1 on t1(c1) tablespace tools;

Index created.

SQL>
SQL> analyze table t1 compute statistics;

Table analyzed.
```

So, let's see how it performs during a query:

```
SQL> select count(c2) from t1 where c1 = 100;

 COUNT(C2)
----------
       535

Elapsed: 00:00:00.58

Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=37 Card=1 Bytes=7)
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (FULL) OF 'T1' (Cost=37 Card=535 Bytes=3745)

Statistics
----------------------------------------------------------
          5  recursive calls
          0  db block gets
        240  consistent gets
          0  physical reads
          0  redo size
        381  bytes sent via SQL*Net to client
        503  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          2  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

[Now, that is odd! Why was a FULL table scan chosen instead of using the index? We performed an ANALYZE … COMPUTE STATISTICS, so the cost-based optimizer isn't lacking any information!

Ah, this is interesting…

```
SQL> select num_rows, blocks from user_tables where table_name = 'T1';

  NUM_ROWS     BLOCKS
---------- ----------
    100000        232

SQL> select num_rows, distinct_keys, avg_leaf_blocks_per_key,
avg_data_blocks_per_key
  2  from user_indexes where index_name = 'I1';

  NUM_ROWS DISTINCT_KEYS AVG_LEAF_BLOCKS_PER_KEY AVG_DATA_BLOCKS_PER_KEY
---------- ------------- ----------------------- -----------------------
    100000           187                       1                     231
```

Querying the statistics populated by the ANALYZE command shows that the cost-based optimizer is aware that there are 100,000 rows in 232 blocks on the table. Likewise, there are 100,000 rows in the associated index, with 187 distinct data values as we planned (by using the "MOD(I, 187)" expression). But here is something else that is interesting: The average number of distinct data blocks per data value (key) is 231 blocks! There are only 232 blocks in the table altogether, but this statistic is telling us that each data value will cause us to visit every single one of the blocks in that table.

If that is the case, why bother with using the index at all? Why not just scan those table blocks by themselves, and leave the additional I/O necessary to utilize the index out of it altogether.

In point of fact, that is exactly what the cost-based optimizer did. On first glance, it seems crazy not to use an index when one is available, but since we have to visit every single block anyway, what is the sense?

Just to prove the point, let's force the use of the index using SQL hints:

```
SQL> select /*+ index(t1 i1) */ count(c2) from t1 where c1 = 100;

 COUNT(C2)
----------
       535

Elapsed: 00:00:00.55

Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=234 Card=1 Bytes=7)
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=234 Card=535
Bytes=3745)
   3    2       INDEX (RANGE SCAN) OF 'I1' (NON-UNIQUE) (Cost=2 Card=535)

Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        249  consistent gets
          0  physical reads
          0  redo size
        381  bytes sent via SQL*Net to client
        503  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

Notice that the value of the statistic *consistent gets* has actually increased, from 240 for the FULL table scan to 249 when using the index. The two statistics *db block gets* and *consistent gets* together summarize to the more familiar statistic *logical reads*, which is the number of I/Os made against the Buffer Cache in the Oracle SGA.

What this means is that the indexed access plan tried to do more work than the FULL table-scan access plan, 249 to 240. If we had created the table with more rows, the difference would have been more pronounced…

Now, let's drop everything and reload the table, this time using a different data pattern:

```
SQL> truncate table t1;

Table truncated.

SQL> drop index i1;

Index dropped.

SQL>
SQL> begin
  2         for i in 1..100000 loop
  3                 insert into t1 values(round(i/187,0), i,
round(i/187,0));
  4         end loop;
  5         commit;
  6  end;
  7  /

PL/SQL procedure successfully completed.

SQL>
SQL> create index i1 on t1(c1) tablespace tools;

Index created.

SQL>
SQL> analyze table t1 compute statistics;

Table analyzed.
```

We truncated the table and dropped the index. Now, in the PL/SQL procedure that is populating the table with another 100,000 rows, we are using the divisor of the counter divided by 187, instead of the remainder resulting from a modulus operation. So now, the pattern of data inserted into the table will be vastly different. Instead of before, when we had:

0,1,2,3,4,5,…,184,185,186,0,1,2,3,…,184,185,186,0,1,2,3,…,184,185,186,0,1,2,3,…

We will now instead have a pattern of:

0,0,0,…(repeat 182 more times)…,0,0,0,1,1,1,…(repeat 182 more times)…,1,1,1,2,2,2,…

The first pattern of data is a poor clustering factor, with data values scattered over many blocks in the table. The new pattern of data is a good clustering factor, where all of the like data values are clustered together into a small number of blocks in the table.

A good clustering factor is good for index performance. Watch…

```
SQL> set autotrace on timing on
SQL>
SQL> select count(c2) from t1 where c1 = 100;

 COUNT(C2)
----------
       187

Elapsed: 00:00:00.20

Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=1 Bytes=7)
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'T1' (Cost=2 Card=187
Bytes=1309)
   3    2       INDEX (RANGE SCAN) OF 'I1' (NON-UNIQUE) (Cost=1 Card=187)

Statistics
----------------------------------------------------------
          5  recursive calls
          0  db block gets
          7  consistent gets
          0  physical reads
          0  redo size
        381  bytes sent via SQL*Net to client
        503  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          2  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

That's a big difference from before! With the previous data pattern, the FULL table-scan was the most cost-effective with 240 logical reads. Using the index put additional overhead on accessing every block in the table, resulting in 249 logical reads.

But now, there are only seven logical reads! That's more what we expect from an index.

Here's the reason why:

continued on page 40

```
SQL> select num_rows, blocks from user_tables where table_name = 'T1';

  NUM_ROWS     BLOCKS
---------- ----------
    100000        242

1 row selected.

SQL> select num_rows, distinct_keys, avg_leaf_blocks_per_key,
avg_data_blocks_per_key
  2  from user_indexes where index_name = 'I1';

  NUM_ROWS DISTINCT_KEYS AVG_LEAF_BLOCKS_PER_KEY AVG_DATA_BLOCKS_PER_KEY
---------- ------------- ----------------------- -----------------------
    100000           536                       1                       1
```

Similar to before, we have 100,000 rows in both the table and the index. This time around, the number of blocks in the table is greater, 242 instead of 232 previously. But the big difference is in the average number of data blocks per distinct data value (key): 1 block, instead of 231 as before.

For that reason, the cost-based optimizer correctly chose to use the index this time. It chose the right thing to do in each situation.

Detecting the Problem

So, we have three major problems:

- Selectivity of data values;
- Uneven distribution of data values;
- Poor clustering of data values.

Detecting these conditions can generally be handled by querying analyzed information from the data dictionary views DBA_TABLES and DBA_INDEXES, using the following queries:

```
SELECT NUM_ROWS, BLOCKS FROM DBA_TABLES WHERE TABLE_NAME = 'table-name';
SELECT NUM_ROWS, DISTINCT_KEYS, AVG_LEAF_BLOCKS_PER_KEY,
AVG_DATA_BLOCKS_PER_KEY
FROM DBA_INDEXES WHERE INDEX_NAME = 'index-name'
```

Of course, you don't have to use the "DBA_" views; you can use the "ALL_" or "USER_" views as well. They all have these columns.

Detecting poor selectivity. Poor selectivity can be detected using the ratio of DISTINCT_KEYS/NUM_ROWS. When the ratio tends toward zero, then on average selectivity is pretty good. When the ratio tends toward one, then there might be something to be worried about.

Detecting uneven distribution of data. But that simple ratio can be fooled by uneven data distribution. In fact, any average value can miss a serious problem caused by uneven data distribution. So, to detect uneven data distribution, I often have to perform another query, this time against the table itself. From the ratio of DISTINCT_KEYS/NUM_ROWS, I'm going to want to calculate the average expected number of rows per data value, using (DISTINCT_KEYS/NUM_ROWS)*NUM_ROWS. Then, I'm going to want to run a query like this:

```
SELECT <indexed-column-list>, COUNT(*)
FROM   table-name
GROUP BY <indexed-column-list>
HAVING COUNT(*) > (4 * average-expected-rows-per-key);
So, for example, let's say that a table named SALES_FACT has an index on a
column named PRODUCT_ID. The NUM_ROWS for the SALES_FACT table is 1,000,000
and the DISTINCT_KEYS on the index on PRODUCT_ID is 2,000. So, the calculated
value for average-expected-rows-per-key is (2000/1000000)/1000000 or 2,000
rows. So the query would look like:
SELECT PRODUCT_ID, COUNT(*)
FROM   SALES_FACT
GROUP BY PRODUCT_ID
HAVING COUNT(*) > 8000;

PRODUCT_ID       COUNT(*)
------------ ------------
BEANYBABY1A8       297888
TICKLEELMO3Z       274009
```

So, while we have a selectivity value that is close to zero (i.e., 2000/1000000) = 0.002, we nonetheless have a serious problem with uneven data distribution. Of the 1,000,000 rows in the table in total, 571,897 rows or 57 percent of all rows are in just two of those 2,000 data values. So, people using the index on PRODUCT_ID would want to use that index for any data value except those two.

Detecting poor clustering. In the end, the only thing that really matters is: how much I/O is using the index going to require. In the test case illustrating the effects of poor clustering, the following query pointed directly at the problem:

```
SELECT NUM_ROWS, DISTINCT_KEYS, AVG_LEAF_BLOCKS_PER_KEY,
AVG_DATA_BLOCKS_PER_KEY
FROM DBA_INDEXES WHERE INDEX_NAME = 'index-name'
```

The value in AVG_LEAF_BLOCKS_PER_KEY can be used to detect both poor selectivity as well as uneven distribution of data. Ideally, this value should be one or a low single-digit number. Remember, each leaf block can hold hundreds of index entries, so whenever this value exceeds low single-digits, there is a problem.

The value in AVG_DATA_BLOCKS_PER_KEY points directly at the poor clustering problem illustrated earlier. You just don't want to scan too many data blocks for each key value.

Resolving the Problem

In this situation, there is nothing you can do to the index to resolve this problem. In each of these three problem situations, the problem resides entirely with the data in the table. Most likely, it is not a "problem" per se—after all, the data is what it is!

The best way to deal with these problems is to provide the cost-based optimizer with enough information to make the right decision, as it did in the test case. If the data is not conducive to using an index, then don't use an index.

Bitmap indexes. The exception to that advice is the case of poor selectivity, the alternative indexing mechanism of bitmap indexes become viable. Describing precisely how bitmap indexes help in this situation is beyond the scope of this version of this paper, but I will expand upon this at a future time. At the present time, I would just advise readers to consult Jonathan Lewis's excellent article entitled "Understanding Bitmap Indexes" available online at www.dbazine.com/jlewis3.shtml. The most basic thing to understand about bitmap indexes is that they are not magic, and they are truly suited only for a small number of applications. They are useful because

they are much more compact than B*Tree indexes, resulting in less I/O during query processing. Moreover, their use of bitmaps makes the use of efficient merging operations using bitmasking, so bitmap indexes are best used in sets, not individually. Last, the advantages conferred by their compactness also results in serious disadvantages during INSERTs, UPDATEs, and DELETEs, so the upshot is that you don't want to perform these operations on bitmap indexes if you don't have to. For this reason alone, bitmap indexes are generally restricted to reporting and data warehouse applications only.

Column-level statistics. Since most of the basic statistics used by the cost-based optimizer are averages (i.e., average selectivity, average number of leaf blocks per key, average number of data blocks per key, etc), it can be fooled by uneven data distribution.

In this case, if you detect uneven data distribution as shown, then you may want to gather *column-level* statistics on the column in question. This additional level of ANALYZE or DBMS_STATS.GATHER_xxx_STATS gathers enough data to allow the cost-based optimizer to detect *popular* data values and avoid using the index. Likewise, when a SQL statement is utilizing an *unpopular data* value on the indexed column(s), the cost-based optimizer can detect the situation and utilize the index.

Summary: Dealing with Non-selective or Unevenly Distributed Data in Indexed Columns

Overall, the three situations described here are situations where using indexes is not always appropriate. The key here is to detect the condition when query performance is poor and understand why performance is poor. If the data conditions are such that using an index is inappropriate, then you can take whatever steps are necessary to avoid the use of the index in question. Use bitmap indexes if the problem is poor selectivity and if the application will permit it. If the problem is uneven data distribution, gather column-level statistics.

Otherwise, trust the cost-based optimizer a little more and be aware of why it might reject the use of an index. It might not be as buggy as you think…

Conclusion

We addressed two major causes of problems with indexes: Sparseness and data problems. We discussed what the problems were, how to detect them, how to resolve them.

This paper did not discuss the problems with block contention when inserting monotonically ascending data values from multiple concurrently executing sessions, nor did it discuss the resolution of this problem using REVERSE indexes.

Nor did the paper discuss the alternative indexing solutions of:

- Function-based indexes
- Descending indexes
- Index-organized tables
- Bitmap-join indexes
- Index compression

I estimate that such discussion would require at least another 15-20 pages. But I'm working on them, and hope to provide a newer, more complete version of this paper soon.

Sources and References

Oracle9i Database Administrator's Guide - Release 2 (9.2) – Oracle Corporation, part number A96521-01 (March 2002), chapter 16 on "Managing Indexes."

Oracle9i Database Concepts – Release 2 (9.2) – Oracle Corporation, part number A96524-01 (March 2002), chapter 10 on "Schema Objects."

Oracle9i Database Performance Tuning Guide and Reference - Release 2 (9.2) – Oracle Corporation, part number A96533-02 (October 2002), chapter 4 on "Understanding Indexes and Clusters."

Knuth, Donald - *The Art of Computer Programming – Volume 3: Sorting and Searching – 2nd edition* (Addison-Wesley Publishing, April 1998).

Lewis, Jonathan – *Unbalanced Indexes?* – www.dbazine.com/jlewis13.shtml.

Lewis, Jonathan – *When should you rebuild an index?* – www.dbazine.com/jlewis14.shtml.

Lewis, Jonathan - *Understanding bitmap indexes* – www.dbazine/jlewis3.shtml.

…and lots and lots of testing on my laptop viewing dumps from "ALTER SYSTEM DUMP DATAFILE n BLOCK MIN nnn BLOCK MAX nnn" and "ALTER SESSION SET EVENTS 'immediate trace name treedump level <obj#>'"

About the Author

Tim Gorman can be reached at SageLogix via e-mail at **tim@sagelogix.com**. He lives in Evergreen, Colo., on a couple acres of heaven with his family and menagerie. He enjoys tuning databases and applications, trouble-shooting, database recovery, snowboarding, and bicycling.