Indexing and Tuning

12 min. read · View original

One of the great dividends of investing in an RDBMS is that you don't have to think too much about the computer's inner life. You're the programmer and say what kinds of data you want. The computer's job is to fetch it and you don't really care how.

Maybe you'll start caring after that $500,000 database server has been grinding away on one of your queries for two solid hours...

While software is being developed, it is rare for tables to contain more than a handful of rows. Nobody wants to sit in SQL*Plus or at Web forms and type in test data. After the application launches and tables begin to fill up, people eventually notice that a particular section of the site is slow. Here are the steps that you must take

1. Find a URL that is running too slowly.
2. If possible, enable query logging from your Web or application server. What you want is for the Web server to write every SQL query and transaction into a single file so that you can see

exactly what the database management system was told to do and when. This the kind of feature that makes a Web programming environment truly productive that it is tough to advertise it to the Chief Technology Officer types who select Web programming environment (i.e., if you're stuck using some closed-source Web connectivity middleware/junkware you might not be able to do this).

With AOLserver, enable query logging by setting `Verbose=On` in the `[ns/db/pool/**poolname**]` section of your .ini file. The queries will show up in the error log ("/home/nsadmin/log/server.log" by default).

3. Request the problematic URL from a Web browser.
4. fire up Emacs and load the query log into a buffer; spawn a shell and run sqlplus from the shell, logging in with the same username/password as used by the Web server
5. you can now cut (from server.log) and paste (into sqlplus) the queries performed by the script backing the slow URL. However, first you must turn on tracing so that you can see what Oracle is doing.

```
SQL> set autotrace on
Unable to verify PLAN_TABLE format or
existence
Error enabling EXPLAIN report
```

Oops! It turns out that Oracle is unhappy about just writing to standard output. For each user that wants to trace queries, you need to feed sqlplus the file

$ORACLE_HOME/rdbms/admin/utlxplan.sql which contains a single table definition:

```
create table PLAN_TABLE (
        statement_id    varchar2(30),
        timestamp       date,
        remarks         varchar2(80),
        operation       varchar2(30),
        options         varchar2(30),
        object_node     varchar2(128),
        object_owner    varchar2(30),
        object_name     varchar2(30),
        object_instance numeric,
        object_type     varchar2(30),
        optimizer       varchar2(255),
        search_columns  number,
        id              numeric,
        parent_id       numeric,
        position        numeric,
        cost            numeric,
        cardinality     numeric,
        bytes           numeric,
        other_tag       varchar2(255),
        partition_start varchar2(255),
        partition_stop  varchar2(255),
        partition_id    numeric,
        other           long);
```

6. Type "set autotrace on" again (it should work now; if you get an error about the PLUSTRACE role then tell your dbadmin to run $ORACLE_HOME/sqlplus/admin/plustrce.sql as SYS then GRANT your user that role).
7. Type "set timing on" (you'll get reports of elapsed time)
8. cut and paste the query of interest.

Now that we're all set up, let's look at a few examples.

A simple B-Tree Index

Suppose that we want to ask "Show me the users who've requested a page within the last few minutes". This can support a nice "Who's online now?" page, like what you see at

http://www.photo.net/shared/whos-online. Here's the source code to find users who've requested a page within the last 10 minutes (600 seconds):

```
select user_id, first_names, last_name, email
from users
where last_visit > sysdate - 600/86400
order by upper(last_name), upper(first_names),
upper(email)
```

We're querying the users table:

```
create table users (
        user_id                 integer primary
key,
        first_names             varchar(100) not
null,
        last_name               varchar(100) not
null,
        ...
        email                   varchar(100) not
null unique,
        ...
        -- set when user reappears at site
        last_visit              date,
        -- this is what most pages query against
(since the above column
        -- will only be a few minutes old for
most pages in a session)
        second_to_last_visit    date,
        ...
);
```

Suppose that we ask for information about User #37. Oracle need not scan the entire table because the declaration that `user_id` be the table's primary key implicitly causes an index to be constructed. The `last_visit` column, however, is not constrained to be unique and therefore Oracle will not build an index on its own. Searching for the most recent visitors at photo.net will require scanning all 60,000 rows in the `users` table. We can add a B-Tree index, for many years the only kind available in any database management system, with the following statement:

```
create index users_by_last_visit on users
(last_visit);
```

Now Oracle can simply check the index first and find pointers to rows in the `users` table with small values of `last_visit`.

Tracing/Tuning Case 1: did we already insert the message?

The SQL here comes from an ancient version of the bulletin board system in the ArsDigita Community System (see http://www.photo.net/bboard/ for an example). In the bad old days when we were running the Illustra relational database management system, it took so long to do an INSERT that users would keep hitting "Reload" on their browsers. When they were all done, there were three copies of a message in the bulletin board. So we modified the insertion script to check the `bboard` table to see if there was already a message with exactly the same values in the `one_line` and `message` columns. Because `message` is a CLOB column, you can't just do the obvious "=" comparison and need to call the PL/SQL function `dbms_lob.instr`, part of Oracle's built-in DBMS_LOB package.

Here's a SQL*Plus session looking for an already-posted message with a subject line of "foo" and a body of "bar":

```
SQL> select count(*) from bboard
where topic = 'photo.net'
and one_line = 'foo'
and dbms_lob.instr(message,'bar') > 0 ;

  COUNT(*)
----------
         0



Execution Plan
----------------------------------------------------
--------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (BY INDEX ROWID) OF
```

```
'BBOARD'
   3    2       INDEX (RANGE SCAN) OF
'BBOARD_BY_TOPIC' (NON-UNIQUE)


Statistics
----------------------------------------------------
--------
          0  recursive calls
          0  db block gets
      59967  consistent gets
      10299  physical reads
          0  redo size
        570  bytes sent via SQL*Net to client
        741  bytes received via SQL*Net from
client
          4  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

Note the "10,299 physical reads". Disk drives are very slow. You don't really want to be doing more than a handful of physical reads. Let's look at the heart of the query plan:

```
   2    1     TABLE ACCESS (BY INDEX ROWID) OF
'BBOARD'
   3    2       INDEX (RANGE SCAN) OF
'BBOARD_BY_TOPIC' (NON-UNIQUE)
```

Looks as though Oracle is hitting the `bboard_by_topic` index for the ROWIDs of "just the rows that have a topic of 'photo.net'". It is then using the ROWID, an internal Oracle pointer, to pull the actual rows from the BBOARD table. Presumably Oracle will then count up just those rows where the ONE_LINE and MESSAGE columns are appropriate. This might not actually be so bad in an installation where there were 500 different discussion groups. Hitting the index would eliminate 499/500 rows. But BBOARD_BY_TOPIC isn't a very selective index. Let's investigate the selectivity with the query `select topic, count(*) from bboard group by topic order by count(*) desc`:

topic	count(*)
photo.net	14159

Nature Photography	3289
Medium Format Digest	1639
Ask Philip	91
web/db	62

The `bboard` table only has about 19,000 rows and the photo.net topic has 14,000 of them, about 75%. So the index didn't do us much good. In fact, you'd have expected Oracle not to use the index. A full table scan is generally faster than an index scan if more than 20% of the rows need be examined. Why didn't Oracle do the full table scan? Because the table hadn't been "analyzed". There were no statistics for the cost-based optimizer so the older rule-based optimizer was employed. You have to periodically tell Oracle to build statistics on tables if you want the fancy cost-based optimizer:

```
SQL> analyze table bboard compute statistics;

Table analyzed.

SQL> select count(*) from bboard
where topic = 'photo.net'
and one_line = 'foo'
and dbms_lob.instr(message,'bar') > 0 ;

  COUNT(*)
----------
         0

Execution Plan
----------------------------------------------------
--------
   0      SELECT STATEMENT Optimizer=CHOOSE
(Cost=1808 Card=1 Bytes=828)
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (FULL) OF 'BBOARD'
(Cost=1808 Card=1 Bytes=828)

Statistics
----------------------------------------------------
--------
          0  recursive calls
          4  db block gets
      74280  consistent gets
```

```
      12266  physical reads
          0  redo size
        572  bytes sent via SQL*Net to client
        741  bytes received via SQL*Net from
client
          4  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

The final numbers don't look much better. But at least the cost-based optimizer has figured out that the topic index won't be worth much. Now we're just scanning the full `bboard` table. While transferring 20,000 rows from Illustra to Oracle during a photo.net upgrade, we'd not created any indices. This speeded up loading but then we were so happy to have the system running deadlock-free that we forgot to recreate an index that we'd been using on the Illustra system expressly for the purpose of making this query fast.

```
SQL> create index bboard_index_by_one_line on
bboard ( one_line );

Index created.
```

Bboard postings are now indexed by subject line, which should be a very selective column because it is unlikely that many users would choose to give their question the same title. This particular query will be faster now but inserts and updates will be slower. Why? Every INSERT or UPDATE will have to update the `bboard` table blocks on the hard drive and also the `bboard_index_by_one_line` blocks, to make sure that the index always has up-to-date information on what is in the table. If we have multiple physical disk drives we can instruct Oracle to keep the index in a separate tablespace, which the database administrator has placed on a separate disk:

```
SQL> drop index bboard_index_by_one_line;

SQL> create index bboard_index_by_one_line
     on bboard ( one_line )
     tablespace philgidx;

Index created.
```

Now the index will be kept in a different tablespace (`philgidx`) from the main table. During inserts and updates, data will be written on two separate disk drives in parallel. Let's try the query again:

```
SQL> select count(*) from bboard
where topic = 'photo.net'
and one_line = 'foo'
and dbms_lob.instr(message,'bar') > 0 ;

  COUNT(*)
----------
         0

Execution Plan
----------------------------------------------------
--------
   0      SELECT STATEMENT Optimizer=CHOOSE
(Cost=2 Card=1 Bytes=828)
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (BY INDEX ROWID) OF
'BBOARD' (Cost=2 Card=1 Bytes=828)
   3    2       INDEX (RANGE SCAN) OF
'BBOARD_INDEX_BY_ONE_LINE' (NON-UNIQUE) (Cost=1
Card=1)

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

We've brought physical reads down from 12266 to 3. Oracle is checking the index on `one_line` and then poking at the main table using the ROWIDs retrieved from the index. It might actually be better to build a concatenated index on two columns: the user ID of the person posting and the subject line, but at this point you might make the engineering decision that 3 physical reads is acceptable.

Tracing/Tuning Case 2: new questions

At the top of each forum page, e.g., http://www.photo.net/bboard/q-and-a.tcl?topic=photo.net, the ArsDigita Community System shows questions asked in the last few days (configurable, but the default is 7 days). After the forum filled up with 30,000 messages, this page was perceptibly slow.

```
SQL> select msg_id, one_line, sort_key, email,
name
from bboard
where topic = 'photo.net'
and refers_to is null
and posting_time > (sysdate - 7)
order by sort_key desc;

...

61 rows selected.

Execution Plan
----------------------------------------------------
--------
   0      SELECT STATEMENT Optimizer=CHOOSE
(Cost=1828 Card=33 Bytes=27324)

   1    0   SORT (ORDER BY) (Cost=1828 Card=33
Bytes=27324)
   2    1     TABLE ACCESS (FULL) OF 'BBOARD'
(Cost=1808 Card=33 Bytes=27324)


Statistics
----------------------------------------------------
--------
          0  recursive calls
```

```
          4  db block gets
      13188  consistent gets
      12071  physical reads
          0  redo size
       7369  bytes sent via SQL*Net to client
       1234  bytes received via SQL*Net from
client
          8  SQL*Net roundtrips to/from client
          2  sorts (memory)
          0  sorts (disk)
         61  rows processed
```

A full table scan and 12,071 physical reads just to get 61 rows! It was time to get medieval on this query. Since the query's WHERE clause contains topic, refers_to, and posting_time, the obvious thing to try is building a concatenated index on all three columns:

```
SQL> create index bboard_for_new_questions
     on bboard ( topic, refers_to, posting_time )
     tablespace philgidx;

Index created.

SQL> select msg_id, one_line, sort_key, email,
name
from bboard
where topic = 'photo.net'
and refers_to is null
and posting_time > (sysdate - 7)
order by sort_key desc;

...

61 rows selected.

Execution Plan
----------------------------------------------------
--------
   0      SELECT STATEMENT Optimizer=CHOOSE
(Cost=23 Card=33 Bytes=27324)

   1    0   SORT (ORDER BY) (Cost=23 Card=33
```

```
Bytes=27324)
   2    1     TABLE ACCESS (BY INDEX ROWID) OF
'BBOARD' (Cost=3 Card=33 Bytes=27324)
   3    2       INDEX (RANGE SCAN) OF
'BBOARD_FOR_NEW_QUESTIONS' (NON-UNIQUE) (Cost=2
Card=33)

Statistics
----------------------------------------------------
--------
          0  recursive calls
          0  db block gets
         66  consistent gets
         60  physical reads
          0  redo size
       7369  bytes sent via SQL*Net to client
       1234  bytes received via SQL*Net from
client
          8  SQL*Net roundtrips to/from client
          2  sorts (memory)
          0  sorts (disk)
         61  rows processed
```

60 reads is better than 12,000. One bit of clean-up, though. There is no reason to have a BBOARD_BY_TOPIC index if we are going to keep this BBOARD_FOR_NEW_QUESTIONS index, whose first column is TOPIC. The query optimizer can use BBOARD_FOR_NEW_QUESTIONS even when the SQL only restricts based on the TOPIC column. The redundant index won't cause any services to fail, but it will slow down inserts.

```
SQL> drop index bboard_by_topic;

Index dropped.
```

We were so pleased with ourselves that we decided to drop an index on `bboard` by the `refers_to` column, reasoning that nobody ever queries `refers_to` without also querying on `topic`. Therefore they could just use the first two columns in the `bboard_for_new_questions` index. Here's a query looking for unanswered questions:

```
SQL> select msg_id, one_line, sort_key, email,
name
```

```
from bboard bbd1
where topic = 'photo.net'
and 0 = (select count(*) from bboard bbd2 where
bbd2.refers_to = bbd1.msg_id)
and refers_to is null
order by sort_key desc;

...

57 rows selected.

Execution Plan
----------------------------------------------------
--------
   0      SELECT STATEMENT Optimizer=CHOOSE
(Cost=49 Card=33 Bytes=27324)

   1    0   SORT (ORDER BY) (Cost=49 Card=33
Bytes=27324)
   2    1     FILTER
   3    2       TABLE ACCESS (BY INDEX ROWID) OF
'BBOARD' (Cost=29 Card=33 Bytes=27324)
   4    3         INDEX (RANGE SCAN) OF
'BBOARD_FOR_NEW_QUESTIONS' (NON-UNIQUE) (Cost=2
Card=33)
   5    2       INDEX (FULL SCAN) OF
'BBOARD_FOR_NEW_QUESTIONS' (NON-UNIQUE) (Cost=26
Card=7 Bytes=56)

Statistics
----------------------------------------------------
--------
          0  recursive calls
          0  db block gets
     589843  consistent gets
     497938  physical reads
          0  redo size
       6923  bytes sent via SQL*Net to client
       1173  bytes received via SQL*Net from
client
          7  SQL*Net roundtrips to/from client
          2  sorts (memory)
```

```
          0  sorts (disk)
         57  rows processed
```

Ouch! 497,938 physical reads. Let's try it with the index in place:

```
SQL> create index bboard_index_by_refers_to
     on bboard ( refers_to )
     tablespace philgidx;

Index created.

SQL> select msg_id, one_line, sort_key, email,
name
from bboard bbd1
where topic = 'photo.net'
and 0 = (select count(*) from bboard bbd2 where
bbd2.refers_to = bbd1.msg_id)
and refers_to is null
order by sort_key desc;

...

57 rows selected.

Execution Plan
----------------------------------------------------
--------
   0      SELECT STATEMENT Optimizer=CHOOSE
(Cost=49 Card=33 Bytes=27324)
   1    0   SORT (ORDER BY) (Cost=49 Card=33
Bytes=27324)
   2    1     FILTER
   3    2       TABLE ACCESS (BY INDEX ROWID) OF
'BBOARD' (Cost=29 Card=33 Bytes=27324)
   4    3         INDEX (RANGE SCAN) OF
'BBOARD_FOR_NEW_QUESTIONS' (NON-UNIQUE) (Cost=2
Card=33)
   5    2       INDEX (RANGE SCAN) OF
'BBOARD_INDEX_BY_REFERS_TO' (NON-UNIQUE) (Cost=1
Card=7 Bytes=56)

Statistics
```

```
------------------------------------------------
--------
          0  recursive calls
          0  db block gets
       8752  consistent gets
       2233  physical reads
          0  redo size
       6926  bytes sent via SQL*Net to client
       1173  bytes received via SQL*Net from
client
          7  SQL*Net roundtrips to/from client
          2  sorts (memory)
          0  sorts (disk)
         57  rows processed
```

This is still a fairly expensive query, but 200 times faster than before and it executes in a fraction of a second. That's probably fast enough considering that this is an infrequently requested page.

Tracing/Tuning Case 3: forcing Oracle to cache a full table scan

You may have a Web site that is basically giving users access to a huge table. For maximum flexibility, it might be the case that this table needs to be sequentially scanned for every query. In general, Oracle won't cache blocks retrieved during a full table scan. The Oracle tuning guide helpfully suggests that you include the following cache hints in your SQL:

```
select /*+ FULL (students) CACHE(students) */
count(*) from students;
```

You will find, however, that this doesn't work if your buffer cache (controlled by db_block_buffers; see above) isn't large enough to contain the table. Oracle is smart and ignores your hint. After you've reconfigured your Oracle installation to have a larger buffer cache, you'll probably find that Oracle is *still* ignoring your cache hint. That's because you also need to

```
analyze table students compute statistics;
```

and then Oracle will work as advertised in the tuning guide. It makes sense when you think about it because Oracle can't realistically start stuffing things into the cache unless it knows roughly how large the table is.

If it is still too slow

If your application is still too slow, you need to talk to the database administrator. If you *are* the database administrator as well as the programmer, you need to hire a database administrator ("dba").

A professional dba is great at finding queries that are pigs and building indices to make them faster. The dba might be able to suggest that you partion your tables so that infrequently used data are kept on a separate disk drive. The dba can make you extra tablespaces on separate physical disk drives. By moving partitions and indices to these separate disk drives, the dba can speed up your application by factors of 2 or 3.

A factor of 2 or 3? Sounds pretty good until you reflect on the fact that moving information from disk into RAM would speed things up by a factor of 100,000. This isn't really possible for database updates, which must be recorded in a durable medium (exception: fancy EMC disk arrays, which contain write caches and batteries to ensure durability of information in the write cache). However, it is relatively easy for queries. As a programmer, you can add indices and supply optimizer hints to increase the likelihood that your queries will be satisfied from Oracle's block cache.

The dba can increase the amount of the server's RAM given over to Oracle. If that doesn't work, the dba can go out and order more RAM!

> *In 1999, Oracle running on a typical ArsDigita server gets 1 GB of RAM.*

Reference

Next: data warehousing

---

philg@mit.edu

Add a comment