Slow Running SQL results in Oracle performance degradation

ChunPei Feng & R. Wang

Environment

Oracle 9iR2 and Unix, production database and standby database

Circumstance

In the morning, routine daily database checking shows that the database has an unusual heavy load. As DBA, definitely, the first checking step is to monitor the top OS processes with command *TOP* or *PRSTAT*, which offer an ongoing look at processor activity in real time. In this case, however, a list of the most CPU-intensive processes on the system does not tell us anything special which might particularly cause the database performance degradation.

Next, information fetching about TOP SQL and long-running SQL also fail to figure out the possible reason of this performance problem.

Also, the team of application development confirms that no change has been made at the application level. And, application log doesn't show exception on heavy jobs and excessive user logon.

According to the information above, it can be concluded that the corrupt database performance is caused by issues relating to the database server.

Steps to diagnose:

1. Check and Compare Historical Statspack Reports

So far, no one is able to tell which job attributes to performance degradation because hundreds of processes, which reside on tens of Unix servers, make DBAs difficult to track process by process. Here, the more feasible action is to recur to Statspack, which provides a great deal of performance information about an Oracle database. By keeping historical Statspack reports, it makes possible to compare current Statspack report to the one in last week. The report, generated at peak period (9:00AM - 10:00AM), is sampled to compare to one of report created in last week at same period.

Upon comparison, the instant finding is that CPU time is increased by 1,200 (2341 vs. 1175) seconds. Usually, the significant increase on CPU time very likely attribute to the following two scenarios:

- More jobs loaded
- The execution plan of SQLs is changed

Top 5 Timed Events in Statspack Reports
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

- ***Current Stataspack Report***

Event	Waits	% Total Time(s)	Ela Time
CPU time		2,341	42.60
db file sequential read	387,534	2,255	41.04
global cache cr request	745,170	231	4.21
log file sync	98,041	229	4.17
log file parallel write	96,264	158	2.88

- ***Statspack Report in Last Week***

Event	Waits	% Total Time(s)	Ela Time
db file sequential read	346,851	1,606	47.60
CPU time		1,175	34.83
global cache cr request	731,368	206	6.10
log file sync	90,556	91	2.71
db file scattered read	37,746	90	2.66

2. Narrow down by examining SQL Part of Statspack Reports

Next, we examine the SQL part of Statspack report and find the following SQL statement (Query 1) is listed at the very beginning of "Buffer Gets" part. It tells us that this SQL statement is the consumer of 1161.27 seconds' CPU Time. In last week's report, no information about this SQL statement has been reported at the very beginning part. And, it only took 7.39 seconds to be finished. It's obvious that this SQL statement must be one of the attributors of performance degradation.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

SELECT login_id, to_char(gmt_create, 'YYYY-MM-DD HH24:MI:SS')
from IM_BlackList where black_id = :b1

Query 1: Query on table IM_BlackList with bind variable

SQL Part of Statspack Report
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

- ***Current Stataspack Report***

Buffer Gets	Executions	Gets per Exec	%Total	CPU Times (s)	Elapsd Times (s)	Hash Value
------------	-------------	---------------	--------	----------	----------	--------------
17,899,606	47,667	375.5	55.6	1161.27	1170.22	3481369999

Module: /home/oracle/AlitalkSrv/config/../../AlitalkSrv/
SELECT login_id, to_char(gmt_create, ‘YYYY-MM-DD HH24:MI:SS’)
from IM_BlackList where black_id = :b1

- ***Statspack Report in Last Week***

Buffer Gets	Executions	Gets per Exec	%Total	CPU Times (s)	Elapsd Times (s)	Hash Value
------------	------------	---------------	-------	----------	----------	--------------
107,937	47,128	2.3	0.8	7.39	6.94	3481369999

Module: /home/oracle/AlitalkSrv/config/../../AlitalkSrv/
SELECT login_id, to_char(gmt_create, 'YYYY-MM-DD HH24:MI:SS')
from IM_BlackList where black_id = :b1

Now, our investigation has been significantly narrowed down to single SQL statement. That is

SELECT login_id, to_char(gmt_create, 'YYYY-MM-DD HH24:MI:SS')
from IM_BlackList where black_id = :b1

This is a typical SQL query with binding variable and it should benefit from b-tree index created. But, the statistics show that it seems conduct full table scan rather than using proper index.

The following checking on index of field black_id in table IM_BlackList clearly demonstrates the availability of the index.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

```
SQL> select index_name,column_name from user_ind_columns where
table_name = 'IM_BLACKLIST';

IM_BLACKLIST_PK LOGIN_ID
IM_BLACKLIST_PK BLACK_ID
IM_BLACKLIST_LID_IND BLACK_ID
```

The question now is, how come full table scan replace usage of index for this SQL statement? In order to testify our supposition, we simply execute this SQL statement against production database and it's clear that full table scan is conducted rather than index access.

3. Go Check Histograms generated by Objects Analyzing

To figure out the problem, we then check histograms on the field of BLACK_ID against standby database. That's also a comparison practice between production database and standby database. Because the activity of gathering statistics does happen on production database, but not on standby database, we are hoping to find some difference between the histograms on the filed BLACK_ID and then to measure the impact of statistics collecting. We select histograms as criteria because histograms is cost-based optimizer (CBO) feature that allows Oracle to see the possible number of values of a particular column, which is known as data skewing, and histograms can track the number of occurrences of a particular data values when CBO decide on what type of index to use or even whether to use an index.

To gather histograms information against standby database, we run:

```
SQL> select COLUMN_NAME ,ENDPOINT_NUMBER, ENDPOINT_VALUE ,
from dba_histograms where table_name = 'IM_BLACKLIST' and
column_name = 'BLACK_ID';
```

Query 2:*gather histograms information from dba_hisrograms*

Then, we get:

```
COLUMN_NAME   ENDPOINT_NUMBER      ENDPOINT_VALUE
------------- -------------------- ------------------
---           -                    -
BLACK_ID      0                    2.5031E+35
BLACK_ID      1                    2.5558E+35
BLACK_ID      2                    2.8661E+35
BLACK_ID      3                    5.0579E+35
BLACK_ID      4                    5.0585E+35
BLACK_ID      5                    5.0585E+35
BLACK_ID      6                    5.0589E+35
BLACK_ID      7                    5.0601E+35
BLACK_ID      8                    5.1082E+35
BLACK_ID      9                    5.1119E+35
```

BLACK_ID	10	5.1615E+35
BLACK_ID	11	5.1616E+35
BLACK_ID	12	5.1628E+35
BLACK_ID	13	5.1646E+35
BLACK_ID	14	5.2121E+35
BLACK_ID	15	5.2133E+35
BLACK_ID	16	5.2155E+35
BLACK_ID	17	5.2662E+35
BLACK_ID	18	5.3169E+35
BLACK_ID	19	5.3193E+35
BLACK_ID	20	5.3686E+35
BLACK_ID	21	5.3719E+35
BLACK_ID	22	5.4198E+35
BLACK_ID	23	5.4206E+35
BLACK_ID	24	5.4214E+35
BLACK_ID	25	5.4224E+35
BLACK_ID	26	5.4238E+35
BLACK_ID	27	5.4246E+35
BLACK_ID	28	5.4743E+35
BLACK_ID	29	5.5244E+35
BLACK_ID	30	5.5252E+35
BLACK_ID	31	5.5252E+35
BLACK_ID	32	5.5272E+35
BLACK_ID	33	5.5277E+35
BLACK_ID	34	5.5285E+35
BLACK_ID	35	5.5763E+35
BLACK_ID	36	5.6274E+35
BLACK_ID	37	5.6291E+35
BLACK_ID	38	5.6291E+35
BLACK_ID	39	5.6291E+35
BLACK_ID	40	5.6291E+35
BLACK_ID	41	5.6305E+35
BLACK_ID	42	5.6311E+35
BLACK_ID	43	5.6794E+35
BLACK_ID	44	5.6810E+35
BLACK_ID	45	5.6842E+35
BLACK_ID	46	5.7351E+35
BLACK_ID	47	5.8359E+35
BLACK_ID	48	5.8887E+35
BLACK_ID	49	5.8921E+35
BLACK_ID	50	5.9430E+35
BLACK_ID	51	5.9913E+35
BLACK_ID	52	5.9923E+35
BLACK_ID	53	5.9923E+35
BLACK_ID	54	5.9931E+35
BLACK_ID	55	5.9947E+35
BLACK_ID	56	5.9959E+35
BLACK_ID	57	6.0428E+35
BLACK_ID	58	6.0457E+35
BLACK_ID	59	6.0477E+35

```
BLACK_ID          60                        6.0479E+35
BLACK_ID          61                        6.1986E+35
BLACK_ID          62                        6.1986E+35
BLACK_ID          63                        6.1994E+35
BLACK_ID          64                        6.2024E+35
BLACK_ID          65                        6.2037E+35
BLACK_ID          66                        6.2521E+35
BLACK_ID          67                        6.2546E+35
BLACK_ID          68                        6.3033E+35
BLACK_ID          69                        6.3053E+35
BLACK_ID          70                        6.3069E+35
BLACK_ID          71                        6.3553E+35
BLACK_ID          72                        6.3558E+35
BLACK_ID          73                        6.3562E+35
BLACK_ID          74                        6.3580E+35
BLACK_ID          75                        1.1051E+36
```

Output 1: Histograms data on standby database

Subsequently, then same command has been executed against production database. The output looks like followings:

```
COLUMN_NAME       ENDPOINT_NUMBER           ENDPOINT_VALUE
-----------------  ---------------------    --------------------
---               -                         -
BLACK_ID          0                         1.6715E+35
BLACK_ID          1                         2.5558E+35
BLACK_ID          2                         2.7619E+35
BLACK_ID          3                         2.9185E+35
BLACK_ID          4                         5.0579E+35
BLACK_ID          5                         5.0589E+35
BLACK_ID          6                         5.0601E+35
BLACK_ID          7                         5.1100E+35
BLACK_ID          8                         5.1601E+35
BLACK_ID          9                         5.1615E+35
BLACK_ID          10                        5.1624E+35
BLACK_ID          11                        5.1628E+35
BLACK_ID          12                        5.1642E+35
BLACK_ID          13                        5.2121E+35
BLACK_ID          14                        5.2131E+35
BLACK_ID          15                        5.2155E+35
BLACK_ID          16                        5.2676E+35
BLACK_ID          17                        5.3175E+35
BLACK_ID          18                        5.3684E+35
BLACK_ID          19                        5.3727E+35
BLACK_ID          20                        5.4197E+35
BLACK_ID          21                        5.4200E+35
BLACK_ID          22                        5.4217E+35
```

BLACK_ID	23	5.4238E+35
BLACK_ID	24	5.4244E+35
BLACK_ID	25	5.4755E+35
BLACK_ID	26	5.5252E+35
BLACK_ID	27	5.5252E+35
BLACK_ID	28	5.5252E+35
BLACK_ID	29	5.5283E+35
BLACK_ID	30	5.5771E+35
BLACK_ID	31	5.6282E+35
BLACK_ID	32	5.6291E+35
BLACK_ID	33	5.6291E+35
BLACK_ID	34	5.6291E+35
BLACK_ID	35	5.6299E+35
BLACK_ID	36	5.6315E+35
BLACK_ID	37	5.6794E+35
BLACK_ID	38	5.6798E+35
BLACK_ID	39	5.6816E+35
BLACK_ID	40	5.6842E+35
BLACK_ID	41	5.7838E+35
BLACK_ID	42	5.8877E+35
BLACK_ID	43	5.8917E+35
BLACK_ID	44	5.9406E+35
BLACK_ID	45	5.9909E+35
BLACK_ID	46	5.9923E+35
BLACK_ID	47	5.9923E+35
BLACK_ID	48	5.9946E+35
BLACK_ID	49	5.9950E+35
BLACK_ID	50	5.9960E+35
BLACK_ID	51	5.9960E+35
BLACK_ID	52	5.9960E+35
BLACK_ID	53	5.9960E+35
BLACK_ID	54	5.9960E+35
BLACK_ID	55	5.9960E+35
BLACK_ID	56	5.9960E+35
BLACK_ID	57	6.0436E+35
BLACK_ID	58	6.0451E+35
BLACK_ID	59	6.0471E+35
BLACK_ID	60	6.1986E+35
BLACK_ID	61	6.1998E+35
BLACK_ID	62	6.2014E+35
BLACK_ID	63	6.2037E+35
BLACK_ID	64	6.2521E+35
BLACK_ID	65	6.2544E+35
BLACK_ID	66	6.3024E+35
BLACK_ID	67	6.3041E+35
BLACK_ID	68	6.3053E+35
BLACK_ID	69	6.3073E+35
BLACK_ID	70	6.3558E+35
BLACK_ID	71	6.3558E+35
BLACK_ID	72	6.3558E+35

BLACK_ID	73	6.3558E+35
BLACK_ID	74	6.3580E+35
BLACK_ID	75	1.1160E+36

Output 2: Histograms data on production database

Comparing to the value of histograms derived from standby database, we find that the histograms values on production database is not distributed evenly as that on standby database. The exception occurred in range of line 50 -56 and line 70-73. That's important finding because histograms are used to predict cardinality and cardinality is the key measure in using B-tree index or bitmap index. The difference of histograms may be the most direct cause for this performance problem we're facing.

4. Trace with event 10053

We then analyze the "10053 event" and try to get more information to figure out this problem. And, this operation is also done against both standby database and production database.

To enable trace with event 10053, we run:

alter session set events '10053 trace name context forever';

And then, we rerun Query 1.

The comparison of these two 10053 trace files, as shown in color red in Output 3 and Output 4, shows that the cost of full table scan are all 38. The difference is that index access cost is jumped from 4 to 65 after conducting optimizer statistics. So far, it's very clear that this SQL statement is executed via path of full table scan rather than index access.

Event 10053 Trace files
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

- **Against Standby Database**

```
Table stats Table: IM_BLACKLIST Alias: IM_BLACKLIST
TOTAL :: CDN: 57477 NBLKS: 374 AVG_ROW_LEN: 38
-- Index stats
INDEX NAME: IM_BLACKLIST_LID_IND COL#: 2
TOTAL :: LVLS: 1 #LB: 219 #DK: 17181 LB/K: 1 DB/K: 2 CLUF: 44331
INDEX NAME: IM_BLACKLIST_PK COL#: 1 2
TOTAL :: LVLS: 1 #LB: 304 #DK: 57477 LB/K: 1 DB/K: 1 CLUF: 55141
_OPTIMIZER_PERCENT_PARALLEL = 0
***************************************
SINGLE TABLE ACCESS PATH
```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

```
Column: BLACK_ID Col#: 2 Table: IM_BLACKLIST Alias: IM_BLACKLIST
NDV: 17181 NULLS: 0 DENS: 5.8204e-05
NO HISTOGRAM: #BKT: 1 #VAL: 2
TABLE: IM_BLACKLIST ORIG CDN: 57477 ROUNDED CDN: 3 CMPTD CDN:
3
Access path: tsc Resc: 38 Resp: 38
Access path: index (equal)
Index: IM_BLACKLIST_LID_IND
TABLE: IM_BLACKLIST
RSC_CPU: 0 RSC_IO: 4
IX_SEL: 0.0000e+00 TB_SEL: 5.8204e-05
Skip scan: ss-sel 0 andv 27259
ss cost 27259
table io scan cost 38
Access path: index (no sta/stp keys)
Index: IM_BLACKLIST_PK
TABLE: IM_BLACKLIST
RSC_CPU: 0 RSC_IO: 309
IX_SEL: 1.0000e+00 TB_SEL: 5.8204e-05
BEST_CST: 4.00 PATH: 4 Degree: 1
***************************************
OPTIMIZER STATISTICS AND COMPUTATIONS
***************************************
GENERAL PLANS
***********************
Join order[1]: IM_BLACKLIST [IM_BLACKLIST]
Best so far: TABLE#: 0 CST: 4 CDN: 3 BYTES: 75
Final:
CST: 4 CDN: 3 RSC: 4 RSP: 4 BYTES: 75
IO-RSC: 4 IO-RSP: 4 CPU-RSC: 0 CPU-RSP: 0
```

Output 3*: Event 10053 Trace file on Standby database*

- **Against Production Database**

```
SINGLE TABLE ACCESS PATH
Column: BLACK_ID Col#: 2 Table: IM_BLACKLIST Alias: IM_BLACKLIST
NDV: 17069 NULLS: 0 DENS: 1.4470e-03
HEIGHT BALANCED HISTOGRAM: #BKT: 75 #VAL: 75
TABLE: IM_BLACKLIST ORIG CDN: 57267 ROUNDED CDN: 83 CMPTD CDN:
83
Access path: tsc Resc: 38 Resp: 38
Access path: index (equal)
Index: IM_BLACKLIST_LID_IND
TABLE: IM_BLACKLIST
RSC_CPU: 0 RSC_IO: 65
IX_SEL: 0.0000e+00 TB_SEL: 1.4470e-03
Skip scan: ss-sel 0 andv 27151
ss cost 27151
table io scan cost 38
```

```
Access path: index (no sta/stp keys)
Index: IM_BLACKLIST_PK
TABLE: IM_BLACKLIST
RSC_CPU: 0 RSC_IO: 384
IX_SEL: 1.0000e+00 TB_SEL: 1.4470e-03
BEST_CST: 38.00 PATH: 2 Degree: 1
***************************************
OPTIMIZER STATISTICS AND COMPUTATIONS
***************************************
GENERAL PLANS
***********************
Join order[1]: IM_BLACKLIST [IM_BLACKLIST]
Best so far: TABLE#: 0 CST: 38 CDN: 83 BYTES: 2407
Final:
CST: 38 CDN: 83 RSC: 38 RSP: 38 BYTES: 2407
IO-RSC: 38 IO-RSP: 38 CPU-RSC: 0 CPU-RSP: 0
```

Output 4*:Event 10053 Trace file on Production database*

5. Perform Statistics again without Analyzing index field

Our diagnosis demonstrates that the sort of skewed values on column BLACK_ID impacts the CBO optimizer in determining optimal execution plan. Thus, the next practice we'd like to do is to eliminate or overwrite histograms information on column BLACK_ID.

We run;

analyze table im_blacklist compute statistics;

And then, re-running Query 2 produces the following output:

```
COLUMN_NAME    ENDPOINT_NUMBER    ENDPOINT_VALUE
------------------ ------------------------ --------------------
GMT_CREATE     0                  2452842.68
GMT_MODIFIED   0                  2452842.68
LOGIN_ID       0                  2.5021E+35
BLACK_ID       0                  1.6715E+35
GMT_CREATE     1                  2453269.44
GMT_MODIFIED   1                  2453269.44
LOGIN_ID       1                  6.3594E+35
BLACK_ID       1                  1.1160E+36
```

Now, the column BLACK_ID has no skewed values displayed like other columns. The present statement analyze object table along with columns. The existing histograms information on columns has been overwritten and then we expect that CBO optimizer can make right decision in determining execution plan.

Therefore, we rerun SQL query 1 and is happy to see that this query is executed via index access instead of full table scan. The problem is eventually solved.

By reviewing the complete process to figure out problem, we realize that the cost of index access is dramatically increased and is even more costly than full table scan. The question now is, why?

The quick answer is that statistics made it.

Deeper Discussion

1. CBO what? How?

Firstly introduced to oracle 7, Cost-Based Optimizer (CBO) checks the execution plan from several possible options and selects the one with lowest cost. It's an extremely sophisticated component of oracle and it governs the execution of every oracle query. CBO is initialized by setting init parameter optimizer_mode. But, this parameter presents differently between oracle 9i and 10G. In Oracle 9i, it's still possible to have the database engine acted upon RBO (Rule-Based Optimizer). But, in Oracle 10g, we only have the choice to define how we benefit from CBO because RBO is de-supported oracle 10G onwards. Whatever oracle version we are on, the most important key is always properly preparing and presenting statistics on database objects. The built-in oracle package DBMS_STATS, which is also recommended from Oracle Corp. rather command "Analyze", can help us to gather statistics in pre-defined way.

We keep in mind that data statistics has been collected prior to the problem. The produced statistics may be improper for Query 1 and thus mislead the determination of execution plan.

2. SQL Query Internal

In order to present deeper understanding about the circumstance we are on, we'd like to examine this specific SQL query more internally.

The Query 1 is quite typical SQL statement with bind variable and it should naturally come with index access as long as index is available. But, the fact disappoints us. The only explanation, which can convince us, is that hard parse of Query 1 analyzes the distribution of histograms of column BLACK_ID and then makes decision to go with full table scan rather than index access because full table scan is less costly than index access at that point. And, it also should be lowest. The selection of full table scan will then dominate the execution of this SQL statement as long as Query 1 does not age out or is cluttered, which may happen if shared_pool_size is too small or non-reusable SQL (i.e. SQL that has literals "***where black_id = 321***) is introduced in the source.

3. How histograms impact CBO optimizer in Determining Execution Plan?

Next, it's time to know how histograms impact the selection of execution plan of SQL statement.

In Oracle, "the cost-based optimizer (CBO) can use histograms to get accurate estimates of the distribution of column data. A histogram partitions the values in the column into bands, so that all column values in a band fall within the same range. Histograms provided improved selectivity estimates in the presence of data skew, resulting in optimal plans with non-uniform data distributions."

In turn, histograms are used to predict cardinality and the number of rows returned to a query. And, cardinality of values of individual table column is also a key measure to define which index mechanism benefit oracle database performance.

In mathematics, the "cardinality" of a set is a measure of the "number of elements of the set". In Oracle, columns of tables with very few unique values, which are called low cardinality, are good candidates of bitmap indexes, other than B-tree index with which we are mostly familiar. Let's assume that we have a computer_type index and that 80 percent of the values are for the DESKTOP type. Whenever a query with clause "where computer_type = 'DESKTOP'" is specified, a full-table scan would be the fastest execution plan, while a query with clause "where computer_type='LAPTOP' "would be faster in using access via an index. That is, if the data values are heavily skewed so that most of the values are in a very small data range, the optimizer may avoid using the index for values in that range and manage to use the index for values outside that range.

Histograms, like all other oracle optimizer statistics, are static. They are useful only when they reflect the current data distribution of a given column. (The data in the column can change as long as the distribution remains constant.) If the data distribution of a column changes frequently, we must recompile its histograms frequently. Histograms will be used to determine the execution plan and thus affect performance. It's undoubted that it incur additional overhead during the parsing phase of a SQL statement. And, generally, histograms can be used effectively only when:

- A table column is referenced in one or more SQL statement query.—*Yes, a column may hold skewed values, but it never be referenced in a SQL statement. It's no need to analyze this column, which mistakenly create histograms on a skewed column.*
- A column's values cause the CBO to make an incorrect guess. — *For heavily skewed column, it's necessary to gather the histograms to aid CBO to choose the best plan.*

Histograms are not useful for columns with the following characteristics:

- All predicates on the column use bind variables. – *That's the circumstance we are on.*
- The column data is uniformly distributed. (Ideally, the clause AUTO SIZE of package DBMS_STATS determines if histograms are created.)
- The column is unique and is used only with equality predicates.

4. Analyze vs. DBMS_STATS

In this case, the typical statistics are collected by executing command ANALYZE. It looks like:

```
ANALYZE TABLE im_blacklist COMPUTE STATISTICS
FOR TABLE
FOR ALL INDEXES
FOR ALL INDEXED COLUMNS
```

The command above analyzes all of the table's columns, with index or not, by using ANALYZE command other than package DBMS_STATS, which is highly recommended by Oracle Corp. to be used to gather statistics information.(Identifying clause AUTO SIZE while issuing package DBMS_STATS will make database to automatically decide which columns need histograms.) The ANALYZE statement we issued above will create histograms for every columns in table IM_BLACKLIST. And, ideally, the histograms will appropriately present the distribution of columns values. The fact, shown in output 2, shows that the distribution of values of column BLACK_ID is sort of skewed (line 50-56 and line 70-73) and thus optimizer internally chooses full table scan instead of index access because at that point full table scan is considered lowest costly execution plan.

Is full table scan really fastest execution plan among possible options in this case? No, it's definitely not. That means, optimizer doesn't choose the optimal execution plan and mistakenly chooses improper one. What does that mean? Yes, oracle optimizer is not perfect. It is a piece of software. It was written by humans, and humans make mistakes. In this case, it's very clear that statistics is not gathered properly for column BLACK_ID because we use command ANALYZE instead of DBMS_STATS. And, the values of column BLACK_ID is very likely not skewed, at least not highly skewed. It may be a oracle bug in creating histograms when we issue command ANALYZE. It's also possible that CBO optimizer fail to choose most optimal execution plan (It may need to be enhanced in future version of Oracle).

Reproduce the happening of Problem internally

Now, we'd like to re-produce this case step by step from the very beginning and depict what happens internally.

1. Analyzing table generates un-uniform histograms on column BLACK_ID.

In this case, we use command ANALYZE to gather statistics of database objects rather than using package DBMS_STATS. As recommended, package DBMS_STATS lets you collect statistics in parallel, collect global statistics for partitioned objects, and fine tune your statistics collection in other ways. The exception of exclusively using command ANALYZE are listed for the following purpose:

- Validate the structure of an object by using the VALIDATE clause
- To list migrated or chained rows by using LIST CHAINED ROWS clause
- Collect statistics not used by the optimizer.
- To collect information on freelist blocks

Note: The testing in using package DBMS_STATS to gather statistics for this specific table (and columns) is not conducted because the circumstance is very likely an occasional event and can not be easily sampled again.

After issuing command ANALYZE, the distribution of histograms is created as sort of un-uniform, as shown in output 2. It should not be arbitrarily concluded that the values of column BLACK_ID are quite skewed because no significant data manipulation happens in production database. But, it's definitely possible that the values are sort of skewed unless analyzing doesn't do correctly to generate the histograms (it may happen). The un-uniform histograms of column BLACK_ID may correctly present the values distribution, or it is improperly created and couldn't present the correct data values range. We can't easily tell what really happen. But, here, we can expect CBO optimizer to make right decision in picking up optimal execution plan. Unfortunately, CBO optimizer fails to make right decision.

2. Parse in share pool

The Query 1, in this case, is a repeatable running SQL statement with bind variable. We don't know what kind of execution plan is created at the first time of running Query 1 after database started. But, at least, it could be concluded that the former execution plan of Query 1 is optimal (via index access) and this execution plan is kept in SQL area of share pool for reusing. There is no way to know how long it will stay there because it heavily depends on the database activity and the effect of that activity on the contents of the SQL area. The following events may happen on share pool.

- The share pool is flushed
- This execution plan may be aged out
- Heavy competition occurs on limited share pool

The happening of whatever events depicted above likely eliminates the execution plan of this SQL statement out of share pool. For this case, it indeed happens. Therefore, the first-time running of Query, right after collecting statistics, will causes the loading of SQL statement source code to share pool and subsequently parsing of SQL statement. During the parsing, oracle optimizer will check the histograms of column BLACK_ID and then compute costs of possible execution plans. Unfortunately, oracle optimizer eventually chooses full table scan rather than index access due to presented sort of skewed histograms of column BLACK_ID. Subsequently, we experience performance degradation and heavy load.

The scenario described above is only assumption and the most straightforward to explain the circumstance we are experiencing.

3. Impact of bind variable

Another possibility, which also makes CBO optimizer to choose sub-optimal execution plan, is presence of "bind variable" in Query 1. As discussed in former section, histograms are not useful while all predicates on the column use bind variables. Therefore, Query 1 is absolutely not candidate for using histograms to help CBO optimizer in determining execution plan.

Here, it's necessary to talk about an init parameter _optim_peek_user_binds, an undocumented session-level parameter. When set to TRUE (default), the CBO optimizer uses the values of the bind variable at the time the query is compiled, and proceeds as though the query has constants instead of bind variable. With 9i and onwards, Oracle picks the values of bind variable in the FIRST PARSE phase and generates execution plans according to the values in this first PARSE. If subsequent bind values are skewed, then execution plans may not be optimal for the subsequent bind variable.

Therefore, can we say Oracle optimizer act incorrectly? No, we can't. At the first-time running, full table scan may be the fastest and the lowest costly execution plan. It hundred percent depends on the value of bind variable **b1**. For repeatable calling of Query 1 with bind variable, the values of bind variable do not keep constant and thus determination of execution plan heavily depends on how the values of bind variable changes and if re-parsing happens due to the availability of identical SQL statement in share pool.

It, considerably, can be identified a bug in Oracle 9ir2. Similar problems have also been reported as in metalink with Doc ID 3668224 and 3765484.

3. New execution plan is sub-optimal

No matter what happens in step 2, re-running of Query 1 will keep using this execution plan regardless of changeable values of bind variable. And then, the performance degradation occurs because of using the expensive execution plan.

4. Re-analyzing table overwrite the histograms of columns

Once we figure out the problem, we then issue table-only ANALYZE command and that looks like:

```
        Analyze table im_blacklist compute statistics;
```

When we analyze a table with clause "COMPUTE STATISTICS", both table and column statistics are collected. But, the previous un-uniform histograms of column BLACK_ID in data dictionary table dba_histograms has been overwritten. Even though the values of column BLACK_ID is still skewed (or sort of skewed), the histograms of column BLACK_ID is not showed like Output 2 with default 75 buckets. Instead, it only shows two buckets (0 and 1) in table dba_histograms. Actually, this table-only analyze command acts as same as identifying clause SIZE 1 to analyze command used previously, like:

```
ANALYZE TABLE im_blacklist COMPUTE STATISTICS
FOR TABLE
FOR ALL INDEXES
FOR ALL INDEXED COLUMNS
SIZE 1;
```

Afterward, our next manual running of Query 1 (separated under SQL*PLUS and not with bind variable) with constant cause hard parse and then generate new execution plan according to current histograms of column BLACK_ID. At this time, the histograms do not present skewed and thus optimizer correctly chooses the execution plan via index access.

Furthermore, it's expected that the next real calling of Query 1 with bind variable will also cause hard parse and CBO optimizer can choose the correct execution plan via index access path because the histograms of column BLACK_ID is not shown as skewed any more.

More on CBO optimizer

As we already known, CBO optimizer heavily depends on the statistics to choose the right execution plan for very SQL statement. Namely, CBO optimizer wouldn't work well without the statistics. And, does that mean we need to collect statistics of oracle objects regularly? It's very hard to say. Generally, regular collecting statistics is necessary, but sometimes, such as in the case we are talking about, statistics collection may negatively harm the database performance. We say, oracle optimizer is not perfect.

In order make effective use of the CBO you should:

- Analyze all tables regularly
- Set the required OPTIMIZER_GOAL (FIRST_ROWS or ALL_ROWS)
- Use hints to help direct the CBO where required
- Use hints in PL/SQL to ensure the expected optimizer is used
- Be careful with the use of bind variables

Basically, CBO optimizer work well for ad-hoc queries. For hard coded, repeated SQL statements, and query with bind variables, these should be tuned to obtain a repeatable optimal execution plan.

Here, we'd like to re-discuss the methods to gather statistics in Oracle database. Rather than command ANALYZE, Oracle highly suggest to use PL/SQL package DBMS_STATS to do that. An innovate feature with DBMS_STATS, comparing to command ANALYZE, is that DBMS_STATS can automatically decide which columns need histograms. This is done by specifying clause SIZE AUTO. That is, for this case, the skewed histograms of column BLACK_ID may not be generated if we use DBMS_STATS with clause SIZE AUTO. Thus, the sub-optimal execution plan will not be chosen at all. That's very important point of DBMS_STATS. And, it's also ideal and could not be guaranteed.

The subsequent issue we'd like to talk about here is when we need to do collection of statistics. The followings are good candidates for re-gathering statistics.

- After large amounts of data changes (loads, purges, and bulk updates)
- After database upgrade or creations of new database
- Newly created database objects, such as tables
- After migration from RBO to CBO
- New high/low values for keys generated
- Upgrading CPUs and I/O subsystem (system statistics)

Besides appropriate statistics gathering, you should always monitor database performance over time. To achieve that, regularly creating and keeping Statspack reports are good for DBAs. Historical statspack reports offer DBAs useful reference to know how database performs. If some exceptional issues happens, DBAs can easily compare the statspack reports and thus figure out the problem.

About the Author.

Chunpei Feng has researched Oracle internals for a long time and understands the internals of block, transaction, undo, consistent reads, memory management, etc. His expertise is within Oracle performance tuning and troubleshooting. He is one of the chief administrators for ITPUB - China's largest Oracle technical community. He has co-authored two books, 'Oracle DBA Best Practices' and 'Oracle Database Performance Tuning', both in Chinese. Chunpei Feng was awarded 'China's 2006 Outstanding Database Engineers'.

R. Wang currently works as Oracle DBA in Canada. He is responsible for database performance tuning and high availability. With over 10 years experience in architecting and building oracle systems, Rui is an evangelist for oracle technology and products. Rui is OCP and received master degree in computer science from Simon Fraser University in Canada.
Visit Rui's blog at http://www.oraclepoint.com/oralife