CSCI317 Database Performance Tuning

SQL Tuning (3)

Dr Janusz R. Getta

School of Computing and Information Technology -
University of Wollongong

SQL Tuning (3)

Outline

Improving efficiency of WHERE and SET clauses

Elimination of redundant computations from WHERE and SET clauses with temporary tables

UPDATE statement with redundant access to ORDERS table

```
UPDATE LINEITEM
SET L_EXTENDEDPRICE = ( SELECT MAX(O_TOTALPRICE)
                        FROM ORDERS )
WHERE L_EXTENDEDPRICE = ( SELECT MIN(O_TOTALPRICE)
                          FROM ORDERS );
```

```
-------------------------------------------------------------------------------
| Id  | Operation            | Name     | Rows | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | UPDATE STATEMENT     |          |    6 |    36 | 12678   (1)| 00:00:01|
|   1 |  UPDATE              | LINEITEM |      |       |            |         |
|*  2 |   TABLE ACCESS FULL  | LINEITEM |    6 |    36 |  8781   (1)| 00:00:01|
|   3 |    SORT AGGREGATE    |          |    1 |     6 |            |         |
|   4 |     TABLE ACCESS FULL| ORDERS   |  450K|  2636K|  1949   (1)| 00:00:01|
|   5 |    SORT AGGREGATE    |          |    1 |     6 |            |         |
|   6 |     TABLE ACCESS FULL| ORDERS   |  450K|  2636K|  1949   (1)| 00:00:01|
-------------------------------------------------------------------------------
   2 - filter("L_EXTENDEDPRICE"= (SELECT MIN("O_TOTALPRICE") FROM
              "ORDERS" "ORDERS"))
```

Improving efficiency of WHERE and SET clauses

Elimination of redundant computations from WHERE and SET clauses with temporary tables

Creating a temporary table

```
CREATE GLOBAL TEMPORARY TABLE MAX_MIN
 AS SELECT MAX(O_TOTALPRICE) MX, MIN(O_TOTALPRICE) MN
    FROM ORDERS;
```

UPDATE statement using a temporary table

```
UPDATE LINEITEM
SET L_EXTENDEDPRICE = ( SELECT MX
                        FROM MAX_MIN )
WHERE L_EXTENDEDPRICE = ( SELECT MN
                          FROM MAX_MIN );
```

```
-------------------------------------------------------------------------------
| Id  | Operation           | Name     | Rows | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | UPDATE STATEMENT    |          |    6 |    36 | 8786    (1)| 00:00:01 |
|   1 |  UPDATE             | LINEITEM |      |       |            |          |
|*  2 |   TABLE ACCESS FULL | LINEITEM |    6 |    36 | 8781    (1)| 00:00:01 |
|   3 |   TABLE ACCESS FULL |  MAX_MIN |    1 |    13 |    2    (0)| 00:00:01 |
|   4 |   TABLE ACCESS FULL |  MAX_MIN |    1 |    13 |    2    (0)| 00:00:01 |
-------------------------------------------------------------------------------
   2 - filter("L_EXTENDEDPRICE"= (SELECT "MN" FROM "MAX_MIN" "MAX_MIN"))
```

SQL Tuning (3)

Outline

- Improving efficiency of `WHERE` and `SET` clauses
- Impact of index maintenance on performance
- Impact of integrity constraints on performance
- Improving efficiency with virtual columns
- `INSERT` specific optimizations
- `DELETE` specific optimizations
- `UPDATE/MERGE` specific optimizations
- `COMMIT` specific optimizations

Impact of index maintenance on performance

Indexing is a very efficient way to improve performance of `SELECT` statements

Indexing is always adds additional workload to processing of `DELETE` and `INSERT` statements

Indexing may add additional workload to processing of `UPDATE` statement

Creating an index and updating a table

```
SQL> CREATE INDEX IDX ON ORDERS(O_TOTALPRICE);
Index created.
SQL>
SQL> SET TIMING ON
SQL> UPDATE ORDERS
  2 SET O_TOTALPRICE = O_TOTALPRICE + 1;
450000 rows updated.
Elapsed: 00:00:15.36
SQL>
SQL> SET TIMING OFF
SQL> DROP INDEX IDX;
Index dropped.
```

Impact of index maintenance on performance

Indexing may add additional workload to processing of `UPDATE` statement

Creating and index and making it unusable

```
SQL> CREATE INDEX IDX ON ORDERS(O_TOTALPRICE);
Index created.
SQL>
SQL> ALTER INDEX IDX UNUSABLE;
Index altered.
SQL>
SQL> SET TIMING ON
SQL> UPDATE ORDERS
  2 SET O_TOTALPRICE = O_TOTALPRICE + 1;
450000 rows updated.
Elapsed: 00:00:09.96
SQL>
SQL> ALTER INDEX IDX REBUILD;
Index altered.
Elapsed: 00:00:02.44
SQL>
SQL> DROP INDEX IDX;
Index dropped.
```

SQL Tuning (3)

Outline

- Improving efficiency of `WHERE` and `SET` clauses
- Impact of index maintenance on performance
- Impact of integrity constraints on performance
- Improving efficiency with virtual columns
- `INSERT` specific optimizations
- `DELETE` specific optimizations
- `UPDATE/MERGE` specific optimizations
- `COMMIT` specific optimizations

Impact of integrity constraints on performance

The presence of integrity constraint always forces database system to evaluate it

Evaluation of integrity constraint always adds additional workload to processing of `INSERT` statements

Evaluation of integrity constraint may add additional workload to processing of `UPDATE` statement

Impact of integrity constraints on performance

Evaluation of integrity constraint may add additional workload to processing of `UPDATE` statement

Processing of UPDATE statement with and without evaluation of consistency constraint

```
SQL> SET TIMING ON
SQL>
SQL> UPDATE ORDERS
  2 SET O_TOTALPRICE = O_TOTALPRICE + 1;
450000 rows updated.
Elapsed: 00:00:10.93
SQL>
SQL> ALTER TABLE ORDERS MODIFY CONSTRAINT ORDER_CHECK1 DISABLE;
Table altered.
Elapsed: 00:00:01.25
SQL>
SQL> UPDATE ORDERS
  2 SET O_TOTALPRICE = O_TOTALPRICE + 1;
450000 rows updated.
Elapsed: 00:00:09.30
SQL>
SQL> ALTER TABLE ORDERS MODIFY CONSTRAINT ORDER_CHECK1 ENABLE;
Table altered.
Elapsed: 00:00:00.26
```

SQL Tuning (3)

Outline

- Improving efficiency of `WHERE` and `SET` clauses
- Impact of index maintenance on performance
- Impact of integrity constraints on performance
- Improving efficiency with virtual columns
- `INSERT` specific optimizations
- `DELETE` specific optimizations
- `UPDATE/MERGE` specific optimizations
- `COMMIT` specific optimizations

Improving efficiency with virtual columns

Virtual columns are defined by the expressions on other columns within a table

Virtual columns can be used to implement the derived values in relational tables with lower overhead than database triggers

Creating a virtual column

```
ALTER TABLE LINEITEM ADD L_TOTAL GENERATED ALWAYS AS
 (L_EXTENDEDPRICE - L_DISCOUNT + L_TAX);
```

SELECT statement with a virtual column

```
SELECT L_ORDERKEY, L_LINENUMBER, L_TOTAL
FROM LINEITEM;
```

```
-------------------------------------------------------------------------------------
| Id  | Operation          | Name     | Rows | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |          | 1800K|    37M| 8780    (1)| 00:00:01 |
|   1 |  TABLE ACCESS FULL| LINEITEM | 1800K|    37M| 8780    (1)| 00:00:01 |
-------------------------------------------------------------------------------------
```

SQL Tuning (3)

Outline

`INSERT` specific optimizations

Array processing enables multiple rows to be inserted in a single operation

Direct path inserts perform insert I/O directly to database files bypassing a buffer cache, for example with `APPEND` hint or using `SQL*Loader`

Multitable inserts enable multiple tables to be involved in a single `INSERT` statement

Multiple free lists avoid contention for the data blocks that receive inserts

Parallel DML can be used with bulk insert operations

SQL Tuning (3)

Outline

- Improving efficiency of `WHERE` and `SET` clauses
- Impact of index maintenance on performance
- Impact of integrity constraints on performance
- Improving efficiency with virtual columns
- `INSERT` specific optimizations
- `DELETE` specific optimizations
- `UPDATE/MERGE` specific optimizations
- `COMMIT` specific optimizations

`DELETE` specific optimizations

Optimize `WHERE` clause with indexes, hash antijoin hints and efficient implementation of correlated queries

Eliminate the indexes that have a significant negative impact on performance of `DELETE` statements

Whenever it is possible use `TRUNCATE TABLE` statement instead of `DELETE` statement

Consider deactivation of the referential integrity constraints

SQL Tuning (3)

Outline

- Improving efficiency of `WHERE` and `SET` clauses
- Impact of index maintenance on performance
- Impact of integrity constraints on performance
- Improving efficiency with virtual columns
- `INSERT` specific optimizations
- `DELETE` specific optimizations
- `UPDATE/MERGE` specific optimizations
- `COMMIT` specific optimizations

UPDATE/MERGE specific optimizations

Eliminate the repetitions of identical operations in SET and WHERE clauses of UPDATE statement through temporary tables or correlated updates

Use MERGE instead of UPDATE and INSERT statements

Create a relational table CUSTOMER_UPDATE that contains changes to C_ACCTBAL that suppose to be propagated to CUSTOMER table

Creating a relational table

```
CREATE TABLE CUSTOMER_UPDATE AS ( SELECT C_CUSTKEY, C_ACCTBAL
                                  FROM CUSTOMER
                                  WHERE MOD(C_CUSTKEY, 5) = 0);
```

Creating an index

```
CREATE UNIQUE INDEX IDX ON CUSTOMER_UPDATE(C_CUSTKEY);
```

Updating a column

```
UPDATE CUSTOMER_UPDATE
SET C_ACCTBAL = 1.1* C_ACCTBAL;
```

UPDATE/MERGE specific optimizations

Add C_ACCTBAL from CUSTOMER_UPDATE to C_ACCTBAL in a relational table CUSTOMER

Updating a relational table

```
UPDATE CUSTOMER
SET C_ACCTBAL = C_ACCTBAL + ( SELECT C_ACCTBAL
                              FROM CUSTOMER_UPDATE
                              WHERE CUSTOMER.C_CUSTKEY =
                                    CUSTOMER_UPDATE.C_CUSTKEY )
WHERE CUSTOMER.C_CUSTKEY IN ( SELECT C_CUSTKEY
                              FROM CUSTOMER_UPDATE );
```

```
-----------------------------------------------------------------------------------------------------
| Id  | Operation                      | Name           | Rows | Bytes| Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | UPDATE STATEMENT               |                | 9000 |  140K| 27285  (33)| 00:00:02|
|   1 |  UPDATE                        | CUSTOMER       |      |      |            |          |
|   2 |   NESTED LOOPS                 |                | 9000 |  140K|   285  (2)| 00:00:01 |
|   3 |    TABLE ACCESS FULL           | CUSTOMER       | 45000|  483K|   282  (0)| 00:00:01 |
|*  4 |    INDEX UNIQUE SCAN           | IDX            |     1|    5 |     0  (0)| 00:00:01 |
|   5 |     TABLE ACCESS BY INDEX ROWID | CUSTOMER_UPDATE|     1|   10 |     2  (0)| 00:00:01 |
|*  6 |      INDEX UNIQUE SCAN         | IDX            |     1|      |     1  (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
  4 - access("CUSTOMER"."C_CUSTKEY"="C_CUSTKEY")
  6 - access("CUSTOMER_UPDATE"."C_CUSTKEY"=:B1)
```

UPDATE/MERGE specific optimizations

Add C_ACCTBAL from CUSTOMER_UPDATE to C_ACCTBAL in a relational table CUSTOMER

Updating a relational table

```
UPDATE (SELECT C_CUSTKEY, CUSTOMER.C_ACCTBAL ACC, CUSTOMER_UPDATE.C_ACCTBAL ACC_UPD
        FROM CUSTOMER JOIN CUSTOMER_UPDATE
                      USING (C_CUSTKEY) )
SET ACC = ACC + ACC_UPD;
```

```
-------------------------------------------------------------------------------------
| Id  | Operation           | Name            | Rows | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | UPDATE STATEMENT    |                 | 9000 |   184K|  291    (1)| 00:00:01|
|   1 |  UPDATE             | CUSTOMER        |      |       |            |         |
|*  2 |   HASH JOIN         |                 | 9000 |   184K|  291    (1)| 00:00:01|
|   3 |    TABLE ACCESS FULL| CUSTOMER_UPDATE | 9000 |  90000|    8    (0)| 00:00:01|
|   4 |    TABLE ACCESS FULL| CUSTOMER        | 45000|   483K|  282    (0)| 00:00:01|
-------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
  2 - access("CUSTOMER"."C_CUSTKEY"="CUSTOMER_UPDATE"."C_CUSTKEY")
```

SQL Tuning (3)

Outline

- Improving efficiency of `WHERE` and `SET` clauses
- Impact of index maintenance on performance
- Impact of integrity constraints on performance
- Improving efficiency with virtual columns
- `INSERT` specific optimizations
- `DELETE` specific optimizations
- `UPDATE/MERGE` specific optimizations
- `COMMIT` specific optimizations

`COMMIT` specific optimizations

Do `COMMIT` less frequently (`we shall return to this problem later ...`)

References

G. Harrison, Oracle Performance Survival Guide, A Systematic Approach to Database Optimization, Prentice Hall Professional Oracle Series, 2010