|  |
| --- |
| EPAM Systems, RD Dep. |
| MTN.BI.07 Oracle Join Methods |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| REVISION HISTORY | | | | | |
| Ver. | Description of Change | Author | Date | Approved | |
| Name | Effective Date |
| 1.0 | Initial status | [Kiryl Bucha](mailto:Kiryl_Bucha@epam.com) | 12-JAN-2012 |  |  |
| 2.0 | Updated in accordance with renewed content | [Elias Nema](mailto:Elias_Nema@epam.com) | 20-JAN-2014 |  |  |

Contents

[1. SQL plus session tool: Auto Trace Description 3](#_Toc384509971)

[2. SQL Execution Plan 4](#_Toc384509972)

[2.1. Using Explain Plan 4](#_Toc384509973)

[2.2. Understanding How EXPLAIN PLAN can Miss the Mark 4](#_Toc384509974)

[3. Join Methods 4](#_Toc384509975)

[3.1. Nested Loops Joins 4](#_Toc384509976)

[3.2. Sort-Merge Joins 5](#_Toc384509977)

[3.3. Hash Joins 5](#_Toc384509978)

[3.4. Cartesian Joins 5](#_Toc384509979)

[3.5. Outer Joins 6](#_Toc384509980)

[3.5.1. Left/Right Outer Joins 6](#_Toc384509981)

[3.5.2. Full Outer Join 6](#_Toc384509982)

[3.6. Semi Joins 6](#_Toc384509983)

[3.7. Anti Joins 7](#_Toc384509984)

[4. Oracle at Work 8](#_Toc384509985)

[4.1. Oracle and Transactions 8](#_Toc384509986)

[4.2. Flashback 9](#_Toc384509987)

[4.3. A Transaction, Step by Step 9](#_Toc384509988)

[4.4. Oracle and Concurrent User Access 10](#_Toc384509989)

[4.5. How Oracle Handles Locking 11](#_Toc384509990)

[4.5.1. A Simple Write Operation 11](#_Toc384509991)

[4.5.2. A Conflicting Write Operation 12](#_Toc384509992)

[4.5.3. A Read Operation 12](#_Toc384509993)

[5. Source Books and Articles 13](#_Toc384509994)

# SQL plus session tool: Auto Trace Description

Below all possible variants of SQL plus utilities autotrace:

Example:

set autotrace off

set autotrace on

set autotrace traceonly

set autotrace on explain

set autotrace on statistics

set autotrace on explain statistics

set autotrace traceonly explain

set autotrace traceonly statistics

set autotrace traceonly explain statistics

set autotrace off explain

set autotrace off statistics

set autotrace off explain statistics

Setting autotrace allows displaying some statistics and/or a [query execution plan](http://www.adp-gmbh.ch/ora/sql/execution_plan.html) for DML statements.

|  |  |
| --- | --- |
| set autotrace on: | Shows the execution plan as well as statistics of the statement. |
| set autotrace on explain: | Displays the execution plan only. |
| set autotrace on statistics: | Displays the statistics only. |
| set autotrace traceonly: | Displays the execution plan and the statistics (as set autotrace on does), but doesn't print a query's result. |
| set autotrace off: | Disables all autotrace |

If autotrace is enabled with statistics, then the following statistics are displayed:

* recursive calls
* db\_block gets
* consistent gets
* physical reads
* redo size
* bytes sent via SQL\*Net to client
* bytes received via SQL\*Net from client
* SQL\*Net roundtrips to/from client
* sorts (memory)
* sorts (disk)

**NOTE:** If you received next error: Check PLUSTRACE role is enabled. Please make next steps:

1. Run next script connected as sysdba:

# @ $ORACLE\_HOME/sqlplus/admin/plustrce.sql;

1. Grant role PLUSTRACE to $UserName$

# grant plustrace to $UserName$;

# SQL Execution Plan

The EXPLAIN PLAN statement is used to display the plan operations chosen by the optimizer for a SQL statement. The first thing I want to clarify is that when you have EXPLAIN PLAN output, you have the estimated execution plan that should be used when the SQL statement is actually executed. You do not have the actual execution plan and its associated row source execution statistics. You have estimates only—not the real thing. Throughout this chapter, I will make the distinction between actual and estimated plan output by referring to estimated information as explain plan output and terming actual information as execution plan output.

## Using Explain Plan

When using EXPLAIN PLAN to produce the estimated execution plan for a query, the output will show:

* Each of the tables referred to in the SQL statement.
* The access method used for each table.
* The join methods for each pair of joined row sources.
* An ordered list of all operations to be completed.
* A list of predicate information related to steps in the plan.
* For each operation, the estimates for number of rows and bytes manipulated by that step.
* For each operation, the computed cost value.
* If applicable, information about partitions accessed.
* If applicable, information about parallel execution.

## Understanding How EXPLAIN PLAN can Miss the Mark

One of the most frustrating things about EXPLAIN PLAN output is that it may not always match the plan that is used when the statement is actually executed. There are three things to keep in mind about using EXPLAIN PLAN that make it susceptible to producing plan output that won’t match the actual execution plan:

* EXPLAIN PLAN produces plans based on the environment at the moment you use it.
* EXPLAIN PLAN doesn’t consider the datatype of bind variables (all binds are VARCHAR2).
* EXPLAIN PLAN doesn’t “peek” at bind variable values.

For these reasons, it is very possible that EXPLAIN PLAN will produce a plan that won’t match the plan that is produced when the statement is actually executed.

# Join Methods

## Nested Loops Joins

Nested loops joins use each row of the query result reached through one access operation to drive into another table. These joins are typically most effective if the result set is limited in size and indexes are present on the columns used for the join. With nested loops, the cost of the operation is based on reading each row of the outer row source and joining it with the matching row of the inner row source.

These kinds of joins are quite robust in that they use very little memory. Since row sets are built one row at a time, there is little overhead required. For that reason, they are actually good for huge result sets except for the fact that building a huge result set one row at a time can take quite a longtime.

Example Nested Loop Join:

SQL> conn scott;

SQL> select empno, ename, dname, loc

from emp, dept

where emp.deptno = dept.deptno;

----------------------------------------------------------------------------

| Id | Operation | Name | Rows | Bytes | Cost (%CPU)|

----------------------------------------------------------------------------

| 0 | SELECT STATEMENT || 14 | 462 | 4 (0)|

| 1 | NESTED LOOPS | | 14 | 462 | 4 (0)|

| 2 | TABLE ACCESS FULL| EMP | 14 | 182 | 3 (0)|

| 3 | TABLE ACCESS BY INDEX ROWID| DEPT | 1 | 20| 1 (0)|

|\* 4 | INDEX UNIQUE SCAN | PK\_DEPT | 1 || 0 (0)|

----------------------------------------------------------------------------

## Sort-Merge Joins

Sort-merge joins read the two tables to be joined independently, sorts the rows from each table (but only those rows that meet the conditions for the table in the WHERE clause) in order by the join key, and then merges the sorted row sets. The sort operations are the expensive part for this join method. For large row sources that won’t fit into memory, the sorts will end up using temporary disk space to complete. This can be quite memory and time-consuming to complete. But once the row sets are sorted, the merge happens quickly. To merge, the database alternates down the two lists, compares the top rows, discards rows that are earlier in the sort order than the top of the other list, and only returns matching rows.

SQL> conn scott;

SQL>select /\*+ USE\_MERGE \*/ empno, ename, dname, loc

from scott.dept, scott.emp

where emp.deptno = dept.deptno;

-----------------------------------------------------------------------------

| Id | Operation | Name | Rows | Bytes | Cost (%CPU)|

-----------------------------------------------------------------------------

| 0 | SELECT STATEMENT | | 14 | 462 | 6 (17)|

| 1 | MERGE JOIN || 14 | 462 | 6 (17)|

| 2 | TABLE ACCESS BY INDEX ROWID| DEPT | 4 | 80 | 2 (0)|

| 3 |INDEX FULL SCAN | PK\_DEPT | 4 | | 1 (0)|

|\* 4 | SORT JOIN | | 14 | 182 | 4 (25)|

| 5 | TABLE ACCESS FULL | EMP | 14 | 182 |3 (0)|

-----------------------------------------------------------------------------

## Hash Joins

Hash joins, like sort-merge joins, first reads the two tables to be joined independently and applies the criteria in the WHERE clause. Based on table and index statistics, the table that is determined to return the fewest rows will be hashed in its entirety into memory. This hash table includes all the row data for that table and is loaded into hash buckets based on a randomizing function that converts the join key to a hash value.

The next step is for the other larger table to be read and the hash function is applied to the join key column. That hash value is then used to probe the smaller in memory hash table for the matching hash bucket where the row data for the first table resides. Each bucket has a list (represented by a bitmap) of the rows in that bucket. That list is checked for matches with the probing row. If a match is made, the row is returned; otherwise it is discarded.

Example of Hash Join plan:

SQL> conn scott;

SQL>select /\*+ USE\_HASH \*/ empno, ename, dname, loc

from scott.dept, scott.emp

where emp.deptno = dept.deptno;

----------------------------------------------------------------

| Id | Operation | Name | Rows | Bytes | Cost (%CPU)|

----------------------------------------------------------------

| 0 | SELECT STATEMENT | | 14 | 462 | 7 (15)|

|\* 1 | HASH JOIN | | 14 | 462 | 7 (15)|

| 2 | TABLE ACCESS FULL| DEPT | 4 | 80 | 3 (0)|

| 3 | TABLE ACCESS FULL| EMP |14 | 182 | 3 (0)|

----------------------------------------------------------------

## Cartesian Joins

Cartesian joins occur when all the rows from one table are joined to all the rows of another table. Therefore, the total number of rows resulting from the join equals the number of rows from one table(A) multiplied by the number of rows in the other table (B) such that A x B = total rows in the result set. Cartesian joins often occur when a join condition is overlooked or left out such that there isn’t a specified join column so the only operation possible is to simply join everything from one row source to everything from the other.

SQL> conn scott;

SQL>select /\*+ USE\_HASH \*/ empno, ename, dname, loc

from scott.dept, scott.emp

------------------------------------------------------------------

| Id | Operation | Name | Rows | Bytes | Cost (%CPU)|

------------------------------------------------------------------

| 0 | SELECT STATEMENT | | 56| 1568 |9 (0)|

| 1 | MERGE JOIN CARTESIAN| | 56 | 1568 | 9 (0)|

| 2 | TABLE ACCESS FULL | DEPT | 4 | 72 | 3 (0)|

| 3 | BUFFER SORT | | 14 | 140 | 6 (0)|

| 4 | TABLE ACCESS FULL| EMP | 14 | 140 | 2 (0)|

------------------------------------------------------------------

## Outer Joins

### Left/Right Outer Joins

An outer join returns all rows from one table and only those rows from the joined table where the join condition is met. Example of Outer Join is below:

---------------------------------------------------------------------------------

| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |

---------------------------------------------------------------------------------

PLAN\_TABLE\_OUTPUT

---------------------------------------------------------------------------------

| 0 | SELECT STATEMENT | | | | 7 (100)| |

| 1 | HASH GROUP BY | | 168 | 3192 | 7 (29)| 00:00:01 |

|\* 2 | HASH JOIN OUTER | | 318 | 6042 | 6 (17)| 00:00:01 |

|\* 3 | TABLE ACCESS FULL| CUSTOMERS | 260 | 3900 | 3 (0)| 00:00:01 |

|\* 4 | TABLE ACCESS FULL| ORDERS | 105 | 420 | 2 (0)| 00:00:01 |

---------------------------------------------------------------------------------

### Full Outer Join

A full outer join will join two tables from left-to-right and right-to-left. Records that join in both directions are output once to avoid duplication. The full outer join will return all the rows from both tables that match plus the rows that are unique to each table.

SQL> conn scott;

SQL>SELECT d.department\_id, e.employee\_id

FROM employees e FULL OUTER JOIN departments d

ON e.department\_id = d.department\_id

ORDER BY d.department\_id;;

---------------------------------------------------------------------------------------

| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |

---------------------------------------------------------------------------------------

| 0 | SELECT STATEMENT | | 122 | 4758 | 6 (34)| 00:0 0:01 |

| 1 | SORT ORDER BY | | 122 | 4758 | 6 (34)| 00:0 0:01 |

| 2 | VIEW | VW\_FOJ\_0 | 122 | 4758 | 5 (20)| 00:0 0:01 |

|\* 3 | HASH JOIN FULL OUTER | | 122 | 1342 | 5 (20)| 00:0 0:01 |

| 4 | INDEX FAST FULL SCAN| DEPT\_ID\_PK | 27 | 108 | 2 (0)| 00:0 0:01 |

| 5 | TABLE ACCESS FULL | EMPLOYEES | 107 | 749 | 2 (0)| 00:0 0:01 |

---------------------------------------------------------------------------------------

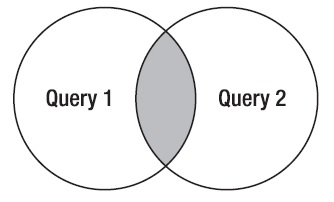
Predicate Information (identified by operation id):

---------------------------------------------------

3 - access("E"."DEPARTMENT\_ID"="D"."DEPARTMENT\_ID")

## Semi Joins

A semi-join is a join between two sets of data (tables) where rows from the first set are returned, based on the presence or absence of at least one matching row in the other set. We'll come back to the "absence" of a matching row later—that is a special case of the semi-join called an anti-join.



**Figure 1 Semi Join**

The main difference between a normal inner join and a semi-join is that with a semi-join, each record in the first set (Query 1 in the diagram) is returned only once, regardless of how many matches there are in the second set (Query 2 in the diagram). This definition implies that the actual processing of the query can be optimized by stopping Query 2 as soon as the first match is found.

SQL> conn scott;

SQL>SELECTDName

FROM SCOTT.dept dept

WHERE deptno IN

(SELECT deptnoFROM scott.emp );

-------------------------------------------------------------------------------

|Id| Operation | Name |Rows|Bytes|Cost (%CPU)| Time |

-------------------------------------------------------------------------------

| 0| SELECT STATEMENT | | 10| 190| 3 (0)| 00:00:01|

| 1| NESTED LOOPS SEMI | | 10| 190| 3 (0)| 00:00:01|

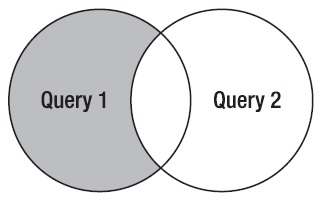
| 2| TABLE ACCESS FULL| DEPARTMENTS | 27| 432| 3 (0)| 00:00:01|

|\*3| INDEX RANGE SCAN | EMP\_DEPARTMENT\_IX | 41| 123| 0 (0)| 00:00:01|

-------------------------------------------------------------------------------

## Anti Joins

Anti-joins are basically the same as semi-joins in that they are an optimization option that can be applied to nested loop, hash, and merge joins. However, they are the opposite of semi-joins in terms of the data they return. Those mathematician types familiar with relational algebra would say that antijoin scan be defined as the complement of semi-joins.



**Figure 2 Anti Join**

SQL> conn scott;

SQL>SELECTDName

FROM SCOTT.dept dept

WHERE deptno NOT IN

(SELECT deptnoFROM scott.emp );

----------------------------------------------------------------------

|Id|Operation |Name |Rows|Bytes|Cost (%CPU)|Time |

----------------------------------------------------------------------

| 0|SELECT STATEMENT| | 17| 323| 6 (17)|00:00:01|

| 1| MERGE JOIN ANTI NA | | 17| 323| 6 (17)|00:00:01|

| 2| SORT JOIN | | 27| 432| 2 (0)|00:00:01|

| 3| TABLE ACCESS BY |DEPARTMENTS| 27| 432| 2 (0)|00:00:01|

| 4| INDEX FULL SCAN |DEPT\_ID\_PK | 27| | 1 (0)|00:00:01|

|\*5| SORT UNIQUE | | 107| 321| 4 (25)|00:00:01|

| 6| TABLE ACCESS FULL|EMPLOYEES | 107| 321| 3 (0)|00:00:01|

----------------------------------------------------------------------

# Oracle at Work

To help you truly understand how all the disparate pieces of the Oracle Database work together, this section walks through an example of the steps taken by the Oracle Database to respond to a user request. This example examines the work of a user who is adding new information to the database—in other words, executing a transaction.

## Oracle and Transactions

A transaction is a work request from a client to retrieve, insert, update, or delete data. (The remainder of this section will focus on transactions that modify data, as opposed to retrieving data.) The statements that change data are a subset of the SQL language called Data Manipulation Language (DML). Transactions must be handled in a way that guarantees their integrity.

* **Transactions are logical and complete.** In database terms, a transaction is a logical unit of work composed of one or more data changes. A transaction may consist of multiple INSERT, UPDATE, and/or DELETE statements affecting data in multiple tables. The entire set of changes must succeed or fail as a complete unit of work. A transaction starts with the first DML statement and ends with either a commit or a rollback.

Oracle also supports autonomous transactions—transactions whose work is committed or rolled back, but that exist within the context of a larger transaction. Autonomous transactions are important because they can commit work without destroying the context of the larger transaction.

* **Commit or rollback.** Once a user enters the data for his transaction, he can either commit the transaction to make the changes permanent or roll back the transaction to undo the changes.
* **System Change Number (SCN).** A key factor in preserving database integrity is an awareness of which transaction came first. For example, if Oracle is to prevent a later transaction from unwittingly overwriting an earlier transaction’s changes, it must know which transaction began first. The mechanism Oracle uses is the System Change Number, a logical timestamp used to track the order in which events occurred. Oracle also uses the SCN to implement multiversion read consistency.
* **Rollback segments.** Rollback segments are structures in the Oracle Database used to store “undo” information for transactions, in case of rollback. This undo information restores database blocks to the state they were in before the transaction in question started. When a transaction starts changing some data in a block, it first writes the old image of the data to a rollback segment. The information stored in a rollback segment is used for two main purposes: to provide the information necessary to roll back a transaction and to support multiversion read consistency. A rollback segment is not the same as a redo log. The redo log is used to log all transactions to the database and to recover the database in the event of a system failure, while the rollback segment provides rollback for transactions and read consistency. Blocks of rollback segments are cached in the SGA just like blocks of tables and indexes. If rollback segment blocks are unused for a period of time, they may be aged out of the cache and written to the disk.Oracle9i introduced automatic management of rollback segments. In previous versions of the Oracle Database, DBAs had to explicitly create and manage rollback segments. Since Oracle9i, you have the ability to specify automatic management of all rollback segments through the use of an undo tablespace. With automatic undo management, you can also specify the length of time that you want to keep undo information; this feature is very helpful if you plan on using flashback queries, discussed in the following section. Oracle Database 10g added an undo management retention time advisor.

Oracle’s method for concurrency management is multiversion read consistency. This method uses rollback segments to retrieve earlier versions of changed rows. If the required blocks are no longer available, Oracle delivers a “snapshot too old” error.

* **Fast commits.** Because redo logs are written whenever a user commits an Oracle transaction, they can be used to speed up database operations. When a user commits a transaction, Oracle can do one of two things to get the changes into the database on the disk:
  + Write all the database blocks the transaction changed to their respective datafiles.
  + Write only the redo information, which typically involves much less I/O than writing the database blocks. This recording of the changes can be replayed to reproduce all the transaction’s changes later, if they are needed due to a failure.

To provide maximum performance without risking transactional integrity, Oracle writes out only the redo information. When a user commits a transaction, Oracle guarantees that the redo for those changes writes to the redo logs on disk. The actual changed database blocks will be written out to the datafiles later. If a failure occurs before the changed blocks are flushed from the cache to the datafiles, the redo logs will reproduce the changes in their entirety. Because the slowest part of a computer system is the physical disk, Oracle’s fast-commit approach minimizes the cost of committing a transaction and provides maximum risk-free performance.

## Flashback

In Oracle9i, rollback segments were also used to implement a feature called Flashback Query. Remember that rollback segments are used to provide a consistent image of the can direct Oracle to return the results for a SQL query at a specific point in time. For instance, you could ask for a set of results from the database as of two hours ago. Flashback provided extra functionality by leveraging the rollback feature that was already a core part of the Oracle architecture.

Since Flashback uses rollback segments, you can only flash back as far as the information in the current rollback segment. This requirement typically limits the span of Flashback to a relatively short period of time—you normally would not be able to roll back days, since your Oracle Database doesn’t keep that much rollback information around. Despite this limitation, there are scenarios in which you might be able to use a Flashback Query effectively, such as going back to a point in time before a user made an error that resulted in a loss of data.

The use of Flashback has increased as Oracle has added more flashback capabilities to the database. Oracle Database 10g greatly expanded the flashback capabilities available to include:

* Flashback Database, to roll back the entire database to a consistent state
* Flashback Table, to roll back a specific table
* Flashback Drop, to roll back a DROP operation
* Flashback Versions Query, to retrieve changes to one or more rows

Oracle Database 11g continued this expansion with the Flashback Transaction feature, which can be used to reverse the effect of a transaction and any other transactions that are dependent on it. Oracle Database 11g R2 added Flashback Data Archive, part of the Advanced Compression option. With this feature, older versions of rows are stored in shadow tables, allowing Flashback Queries against very old data without having to have to keep all of the versions in the undo tablespace. In Oracle Database 12c, Flashback Queries are extended to support queries on temporal validity dimensions (for valid time periods). Additionally, with Oracle Database 12c you can recover an individual table from a database backup.

## A Transaction, Step by Step

This simple example illustrates the complete process of a transaction. The example uses the EMP table of employee data, which is part of the traditional test schema shipped with Oracle Databases. In this example, an HR clerk wants to update the name of an employee. The clerk retrieves the employee’s data from the database, updates the name, and commits the transaction.

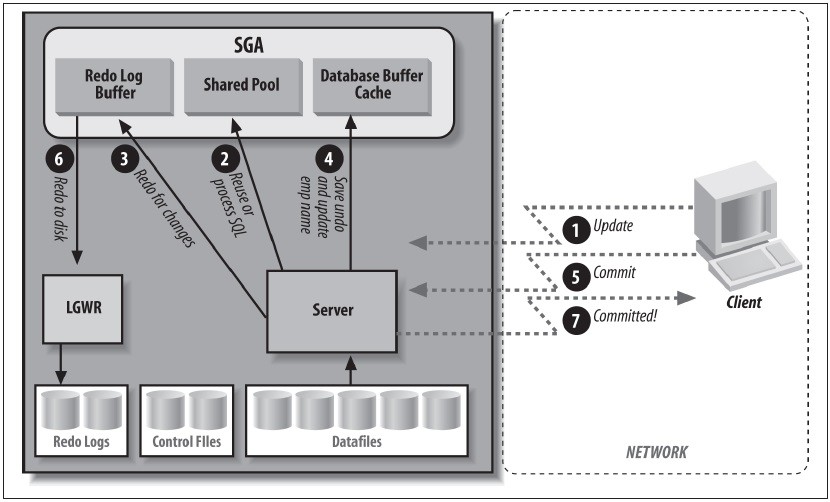
The example assumes that only one user is trying to update the information for a row in the database. Because of this assumption, it won’t include the steps normally taken by Oracle to protect the transaction from changes by other users.

The HR clerk already has the employee record on-screen and so the database block containing the row for that employee is already in the database buffer cache. The steps from this point would be:

1. The user modifies the employee name on-screen and the client application sends a SQL UPDATE statement over the network to the server process.
2. The server process looks for an identical statement in the shared SQL area of the shared pool. If it finds one, it reuses it. Otherwise, it checks the statement for syntax and evaluates it to determine the best way to execute it. This processing of the SQL statement is called parsing and optimizing. Once the processing is done, the statement is cached in the shared SQL area.
3. The server process copies the old image of the employee data about to be changed notes the changes in a rollback segment and to a redo segment. The rollback segment changes are part of the redo. This may seem a bit odd, but remember that redo is generated for all changes resulting from the transaction. The contents of the rollback segment have changed because the old employee data was written to the rollback segment for undo purposes. This change to the contents of the rollback segment is part of the transaction and therefore part of the redo for that transaction.
4. Once the server process has completed this work, the process modifies the database block to change the employee name. The database block is stored in the database cache at this time. Control is passed back to the client process.
5. The HR clerk commits the transaction.
6. The Log Writer (LGWR) process writes the redo information for the entire transaction from the redo log buffer to the current redo logfile on disk. When the operating system confirms that the write to the redo logfile has successfully completed, the transaction is considered committed.
7. The server process sends a message to the client confirming the commit.

The user could have canceled or rolled back the transaction instead of committing it, in which case the server process would have used the old image of the employee data in the rollback segment to undo the change to the database block.

Figure 3 shows the steps described here. Network traffic appears as dotted lines.



**Figure 3 Oracle Transaction Steps**

## Oracle and Concurrent User Access

Oracle solves the problems created by concurrent access through a technology called multiversion read consistency, sometimes referred to as MVRC. Multiversion read consistency guarantees that a user sees a consistent view of the data she requests. If another user changes the underlying data during the query execution, Oracle maintains a version of the data as it existed at the time the query began. If there were transactions underway but uncommitted at the time the query began, Oracle will ensure that the query ignores the changes made by those transactions. The data returned to the query will reflect all committed transactions at the time the query started.

This feature has two dramatic effects on the way queries impact the database. First, Oracle doesn’t place any locks on data for read operations. This means that a read operation will never block a write operation. Even where the database places a single lock on a single row as part of a read operation, that single lock can still cause contention in the database, especially since most database tables tend to concentrate update operations around a few “hot spots” of active data.

Second, a user gets a complete “snapshot” view of the data, accurate at the point in time that the query began. Other databases may reduce the amount of contention in the database by locking an individual row only while it’s being read, rather than over the complete duration of the row’s transaction. A row that’s retrieved at the end of a result set may have been changed since the time the result set retrieval began. Because rows that will be read later in the execution of the query weren’t locked, they could be changed by other users, which would result in an inconsistent view of the data.

## How Oracle Handles Locking

Let’s walk through three scenarios: a simple write to the database, a situation in which two users attempt to write to the same row in the same table, and a read that takes place in the midst of conflicting updates.

For the purposes of these examples, we’ll use the scenario of one or two users modifying the EMP table, a part of the standard sample Oracle schema that lists data about employees via a form.

### A Simple Write Operation

This example describes a simple write operation, in which one user is writing to a row in the database. In this example, an HR clerk wants to update the name for an employee. Assume that the HR clerk already has the employee record on-screen. The steps from this point are as follows:

1. The client modifies the employee name on the screen. The client process sends a SQL UPDATE statement over the network to the server process.
2. The server process obtains a System Change Number and reads the data block containing the target row.
3. The server records row lock information in the data block.
4. The server writes the old image of the data to the redo buffers in memory, and then writes the changes to an UNDO segment and modifies the employee data, which includes writing the SCN to the ORA\_ROWSCN pseudocolumn in Oracle Database 10g or newer database releases.
5. The server process writes the redo buffers to disk, and then writes the UNDO segments and the changed data to disk. The UNDO segment changes are part of the redo, since the redo log stores all changes coming from the transaction.
6. The HR clerk commits the transaction.
7. Log Writer (LGWR) writes the redo information for the entire transaction, including the SCN that marks the time the transaction was committed, from the redo log buffer to the current redo logfile on disk. When the operating system confirms that the write to the redo logfile has successfully completed, the transaction is considered committed.
8. The server process sends a message to the client confirming the commit.

Oracle Database 10g Release 2 introduced the ability to have the server process return control to the client without waiting for all the redo information to be written. The plus side of this enhancement is that high-volume OLTP applications may benefit from improved performance. The downside of this feature is that it opens a window of vulnerability—the database could crash after a transaction had been committed, but before the redo was written, which would make it impossible to recover the committed transaction, so this feature should be used with caution.

Oracle Database 12c introduces a new feature called Transaction Guard. If you followed the description above closely, you can see that a transaction could be between step 7 and step 8 when some type of failure occurs that prevents the message relaying the successful commit to the application. (This failure could have nothing to do with the database and be quite short-lived—such as a network failure.)This series of events would mean that the Oracle Database has committed data, but the application does not know whether the commit failed or succeeded. Transaction Guard provides an API that allows an application to specifically check on the outcome of a potentially failed transaction.

### A Conflicting Write Operation

The write operation previously described is a little different if there are two users, Client A and Client B, who are trying to modify the same row of data at the same time. The steps are as follows:

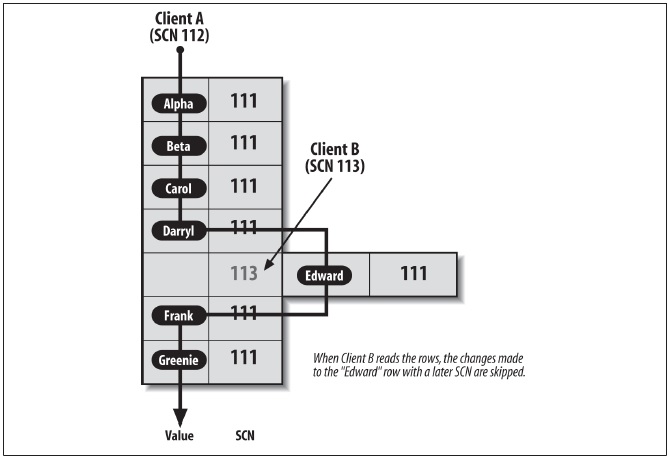
1. Client A modifies the employee name on the screen. Client A sends a SQL UPDATE statement over the network to the server process.
2. The server process obtains an SCN for the statement and reads the data block containing the target row.
3. The server records row lock information in the data block.
4. The server process writes the changes to the redo log buffer.
5. The server process copies the old image of the employee data about to be changed to an UNDO segment. Once the server process has completed this work, the process modifies the employee data, which includes writing the SCN to the ORA\_ROWSCN pseudocolumn in Oracle Database 10g or newer database releases.
6. Client B modifies the employee name on the screen and sends a SQL UPDATE statement to the server.
7. The server process obtains an SCN and reads the data block containing the target row.
8. The server process sees that there is a lock on the target row from the information in the header of the data block, so it takes one of two actions. If the isolation level on Client B’s transaction is READ COMMITTED, the server process waits for the blocking transaction to complete. If the isolation level for Client B’s transaction is SERIALIZABLE, an error is returned to the client.
9. Client A commits the transaction, the server process takes the appropriate action, and the server sends a message to Client A confirming the commit.
10. If Client B executed the SQL statement with the READ COMMITTED isolation level, the SQL statement then proceeds through its normal operation.

The previous example illustrates the default behavior of Oracle when it detects a problem caused by a potential lost update. Because the SERIALIZABLE isolation level has a more drastic effect when it detects a write conflict than the READ COMMITTED isolation level, many developers prefer the latter level. They can avoid some of the potential conflicts by either checking for changes prior to issuing an update (by comparing values in a row or using the Oracle Database 10g or later row SCN) or using the SELECT FOR UPDATE syntax in their SQL to avoid the problem altogether.

### A Read Operation

You can really appreciate the beauty of Oracle’s read consistency model by looking at the more common scenario of one user reading data and one user writing to the same row of data. In this scenario, Client A is reading a series of rows from the EMP table, while Client B modifies a row before Client A reads it, but after Client A begins her transaction:

1. Client A sends a SQL SELECT statement over the network to the server process.
2. The server process obtains an SCN for the statement and begins to read the requested data for the query. For each data block that it reads, it compares the SCN of the SELECT statement with the SCNs for any transactions for the relevant rows of the data block. If the server finds a transaction with a later SCN than the current SELECT statement, the server process uses data in the UNDO segments to create a “consistent read” version of the data block, current as of the time the SELECT was issued. This is what provides the multiversion read consistency (MVRC) and avoids the need for Oracle to use read locks on data. If a row has been updated since the transaction started, Oracle simply gets the earlier version of the data for a consistent view.
3. Client B sends a SQL UPDATE statement for a row in the EMP table that has not yet been read by Client A’s SELECT statement. The server process gets an SCN for the statement and begins the operation.
4. Client B commits his changes. The server process completes the operation, which includes recording information in the data block that contains the modified row that allows Oracle to determine the SCN for the update transaction.
5. The server process for Client A’s read operation comes to the newly modified block. It sees that the data block contains changes made by a transaction that has an SCN that is later than the SCN of the SELECT statement. The server process looks in the data block header, which has a pointer to the UNDO segment that contains the data as it existed when Client A’s transaction started. The UNDO segment uses the old version of the data to create a version of the block as it existed when the SELECT statement started. Client A’s SELECT statement reads the desired rows from this consistent version of the data block.



**Figure 4 Oracle MVRC**

# Source Books and Articles

1. Kyte T. Expert Oracle Database Architecture: Oracle Database 9i, 10g, and 11g Programming Techniques and Solutions, Second Edition. Apress, 2010.
2. Morton K., & Osborne K., & Sands R., & Shamsudeen R., & Still J. Pro Oracle SQL. Apress, 2013.