Lecture 3:
Oracle 12c Database Data Concurrency:
Transactions and Locking – Part 1

By David Yitzhak
shaked19@gmail.com

http://www.ildba.co.il/author/cimid/
http://www.sqlserver.co.il/?cat=940

Global Hebrew Virtual PASS Chapter:
https://www.youtube.com/watch?v=x4hGjYGBfkc
https://www.youtube.com/watch?v=eJO8G9if3EY

SqlSaturday Israel 2016:
Reference and Credits

Apress Oracle Database Transactions and Locking Revealed (2014)
Chapter 1: Getting Started
Chapter 2: Locking and Issues
Chapter 3: Lock Types
Chapter 4: Concurrency and Multiversioning
Chapter 5: Transactions
Chapter 6: Redo and Undo

Oracle® Database Concepts
12c Release 1 (12.1)
E41396-13
https://docs.oracle.com/database/121/CNCPT/toc.htm
Part III Oracle Transaction Management
9 Data Concurrency and Consistency
10 Transactions
Agenda

• Transactions
• Data Concurrency and Consistency
• Locks in Oracle
• Oracle DB Locking Rules
• Lost Updates
• Non-repeatable reads
• Pessimistic Locking, Optimistic Locking
• Locks and Foreign Keys
• Locks and Deadlocks
• Lock Escalation\Conversion
• Locks Types
• DML Locks
• Storage of Row Locks: Oracle
• Row Locks (TX)
Transactions

- Set of DML statements executed sequentially by a session.
- Implicitly begins with the first statement that modifies data (first statement that gets a TX lock) executed by the session:
  - UPDATE
  - DELETE
  - MERGE
  - SELECT FOR UPDATE
  - LOCK TABLE
- Ends with either a COMMIT or ROLLBACK
- ACID characteristics of Oracle transactions: When you issue a COMMIT, you are assured that all of your changes have been successfully saved and that any data integrity checks and rules have been validated
  - Atomicity: Either all of a transaction happens or none of it happens.
  - Consistency: A transaction takes the database from one consistent state to the next.
  - Isolation: The effects of a transaction may not be visible to other transactions until the transaction has committed.
  - Durability: Once the transaction is committed, it is permanent.
Transactions Control Statements

- **COMMIT**: Ends transaction and makes any changes permanent (durable).
- **ROLLBACK**: ends transaction and undoes any uncommitted change.
- **SAVEPOINT**: Allows you to create a marked point within a transaction. You may have multiple SAVEPOINTs within a single transaction.
- **ROLLBACK TO <SAVEPOINT>**: Used with the SAVEPOINT command. You can roll back your transaction to that marked point without rolling back any of the work that preceded it.
- **SET TRANSACTION**: Allows you to set various transaction attributes: Isolation level and whether it is read-only or read-write.
- **You should always explicitly terminate your transactions with a COMMIT or ROLLBACK**
Creating Savepoints: Example

- To update the salary for Banda and Greene in the sample table hr.employees, check that the total department salary does not exceed 314,000, then reenter the salary for Greene:

```sql
UPDATE employees SET salary = 7000 WHERE last_name = 'Banda';
SAVEPOINT banda_sal;

UPDATE employees SET salary = 12000 WHERE last_name = 'Greene';
SAVEPOINT greene_sal;

SELECT SUM(salary) FROM employees;

ROLLBACK TO SAVEPOINT banda_sal;

UPDATE employees SET salary = 11000 WHERE last_name = 'Greene';
COMMIT;
```
Set Transaction READ_ONLY Example

- All subsequent queries in transaction see only changes that were committed before the transaction began.
- Useful for reports that run multiple queries against one or more tables while other users update these same tables

```sql
COMMIT;
SET TRANSACTION READ ONLY NAME 'Toronto';
SELECT product_id, quantity_on_hand FROM inventories WHERE warehouse_id = 5 ORDER BY product_id;
COMMIT;
```

- First COMMIT statement ensures that SET TRANSACTION is the first statement in the transaction.
- Last COMMIT statement does not actually make permanent any changes to the database. It simply ends the read-only transaction.
Locks in Oracle

- In a single-user database, locks are not necessary.
- Locks are used in DB to permit concurrent access to **shared resources**, while at the same time providing data integrity and consistency.
- Oracle locks table data at the row level but also uses locks to provide concurrent access to various resources like stored procedure.
- Locks achieve the following important DB requirements:
  - **Consistency**
    - The data a session is viewing or changing must not be changed by other sessions until the user is finished.
  - **Integrity**
    - The data and structures must reflect all changes made to them in the correct sequence.
- Oracle DB provides data concurrency, consistency, and integrity among transactions through its locking mechanisms.
- **Locking occurs automatically and requires no user action!**
locking behavior of Oracle DB for readers and writers

- Oracle locks data at the row level on Modification. There is no lock escalation to a block or table level.
- Oracle never locks data just to read it. There are no locks placed on rows of data by simple reads.
- A writer of data does not block a reader of data. This is fundamentally different from many other databases, where reads are blocked by writes.
- writer of data is blocked only when another writer of data has already locked the row. A reader of data never blocks a writer of data.

Transaction 1

```
UPDATE hr.employees
2  SET salary=salary*1.2
3 WHERE employee_id=512;
```

Transaction 2

```
SQL> UPDATE hr.employees
2  SET salary=salary+400
3  WHERE employee_id=512;
```
Oracle DB Locking Rules & Lock enqueue

• No exclusive lock needed for SELECTs.
  – A query always succeeds using a pre lock image of data stored in undo tablespace.

• All locks are released when transaction is completed
  – Rollback or Commit

• Row-level locks are used for INSERT, UPDATE and DELETE

• The session enqueue method is automatic and handled by Oracle
  – At SQL> command prompt, DML statement (INSERT, UPDATE, DELETE, MERGE) is waiting for a lock resource seems to hang, unless NOWAIT is used in Lock statement.

• The enqueue mechanism keeps track of:
  – Sessions waiting for locks
  – Requested lock mode
  – Order in which sessions requested the lock

• You can always pass locking hints in your SQL and/or PL/SQL Code
  
  SELECT * FROM hr.emp where emp_id=115 FOR UPDATE
Lost Updates

1. A transaction in Session1 retrieves (queries) a row of data into local memory and displays it to an end user, User1.

2. Another transaction in Session2 retrieves that same row, but displays the data to a different end user, User2.

3. User1, using the application, modifies that row and has the application update the database and commit. Session1’s transaction is now complete.

4. User2 modifies that row also, and has the application update the database and commit. Session2’s transaction is now complete.

A lost update because all of the changes made in Step 3 will be lost.

Classic DB Problem in problem in all multiuser computer environments

\[ s = r_1(x)r_2(x)w_1(x)c_1w_2(x)c_2 \]
Non-repeatable reads

• Called inconsistent analysis.

• A read is non-repeatable if a query might get different values when reading the same data in two separate reads within the same transaction.

• This can happen when a separate transaction updates the same data, after the first read but before the second read.

• Let $P$ be a predicate and $x$ is not in $P$.

• Operation $w2(x)$ changes $x$ in a way that $x$ satisfies $P$.

• schedule $S = r1(P)w2(x)c2r1(P)c1$
Pessimistic Locking

- A row lock would be placed as soon as the user indicates his intention to perform an update on a specific row that he has selected.

Example:

```sql
> select empno, ename, sal from emp where deptno = 10;

EMPNO  ENAME  SAL
----------  -------  -----
7782    CLARK   2450
7839    KING    5000
7934    MILLER  1300
```
Pessimistic Locking- Cont

• Bind the values the user selected so we can query the database and make sure the data hasn’t been changed yet.

```sql
> variable empno number
> variable ename varchar2(20)
> variable sal number
> exec :empno := 7934; :ename := 'MILLER'; :sal := 1300;
PL/SQL procedure successfully completed.
• we are going to lock the row using FOR UPDATE NOWAIT

> select empno, ename, sal from emp
where empno = :empno
and decode( ename, :ename, 1 ) = 1
and decode( sal, :sal, 1 ) = 1
for update nowait
/
EMPNO ENAME SAL
---------- ---------- ----------
7934 MILLER 1300
```

[Shorthand way of expressing where (column = :bind_variable OR (column is NULL and :bind_variable is NULL)"

since NULL = NULL is never true (nor false!)"
Pessimistic Locking- Cont

- If the underlying data has not changed, we will get our MILLER row back, and this row will be locked from updates (but not reads) by others.
- If another user is in the process of modifying that row, we will get an ORA-00054 resource busy error. We must wait for the other user to finish with it.
- If, in the time between selecting the data and indicating our intention to update, someone has already changed the row, then we will get zero rows back.

```
SCOTT@ORA12CR1> update emp
set ename = :ename, sal = :sal
where empno = :empno;
row updated.
> commit;
Commit complete.
```
Optimistic Locking

- Defers all locking up to the point right before the update is performed.
- One popular implementation of optimistic locking is to keep the old and new values in the application, and upon updating the data, use an update like

\[ \text{Update table} \]
\[ \text{Set } \text{column1} = \text{:new_column1}, \text{column2} = \text{:new_column2}, \ldots \]
\[ \text{Where } \text{primary_key} = \text{:primary_key} \]
\[ \text{And decode( column1, :old_column1, 1 ) = 1} \]
\[ \text{And decode( column2, :old_column2, 1 ) = 1} \]

Other Options:
- Optimistic Locking Using a Version Column (systimestamp column)
- Optimistic Locking Using a Checksum
Optimistic Locking Using a Version Column

- Insert a copy of the DEPT data into this table:

```sql
EODA@ORA12CR1> create table dept
  2  ( deptno number(2),
  3    dname varchar2(14),
  4    loc varchar2(13),
  5  last_mod timestamp with time zone
  6    default systimestamp
  7    not null,
  8    constraint dept_pk primary key(deptno)
  9  )
10 /
Table created.

EODA@ORA12CR1> insert into dept( deptno, dname, loc )
  2  select deptno, dname, loc
  3  from scott.dept;
4 rows created.

EODA@ORA12CR1> commit;
Commit complete.
```

- LAST_MOD column NOT NULL. It must be populated with default value: current system time.

- TIMESTAMP data type has the highest precision available in Oracle (microsecond)
Optimistic Locking Using a Version Column

EODA@ORA12CR1> variable deptno number
EODA@ORA12CR1> variable dname varchar2(14)
EODA@ORA12CR1> variable loc varchar2(13)
EODA@ORA12CR1> variable last_mod varchar2(50)
EODA@ORA12CR1>
EODA@ORA12CR1> begin
2   :deptno := 10;
3   select dname, loc, to_char( last_mod, 'DD-MON-YYYY HH.MI.SSXFF AM TZR' )
4     into :dname,:loc,:last_mod
5     from dept
6   where deptno = :deptno;
7 end;
8 /
PL/SQL procedure successfully completed.

• Currently

EODA@ORA12CR1> select :deptno dno, :dname dname, :loc loc, :last_mod lm
2   from dual;

<table>
<thead>
<tr>
<th>DNO</th>
<th>DNAME</th>
<th>LOC</th>
<th>LM</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
<td>15-APR-2014 07.04.01.147094 PM -06:00</td>
</tr>
</tbody>
</table>
Optimistic Locking Using a Version Column

- Update row by primary key (DEPTNO). Verify LAST_MOD column had not been modified by any other session between the time we read it first and the time we did the update.

```
EODA@ORA12CR1> update dept
2    set dname = initcap(:dname),
3        last_mod = systimestamp
4    where deptno = :deptno
5    and last_mod = to_timestamp_tz(:last_mod, 'DD-MON-YYYY HH.MI.SSXFF AM TZR');

1 row updated.
```

- Run same update again. Notice how 0 rows updated because the predicate on LAST_MOD was not satisfied.

```
EODA@ORA12CR1> update dept
2    set dname = upper(:dname),
3        last_mod = systimestamp
4    where deptno = :deptno
5    and last_mod = to_timestamp_tz(:last_mod, 'DD-MON-YYYY HH.MI.SSXFF AM TZR');

0 rows updated.
```
Optimistic or Pessimistic Locking?

• Pessimistic locking works very well in Oracle
  – Has many advantages over optimistic locking
  – Requires a stateful connection to DB, like a client/server connection => unrealistic in many cases today

• Optimistic concurrency recommend for most applications with version column approach with a timestamp
  – less computationally expensive than a hash or checksum on large columns like LONG, LONG RAW, CLOB, BLOB.
Locks and Deadlocks

- *Deadlocks* occur when you have two sessions, each of which is holding a resource that the other want.
- Deadlocks prevent some transactions from continuing to work.
- Oracle DB automatically detects deadlocks and resolves them by rolling back one statement involved in the deadlock, releasing one set of the conflicting row locks.
- DB returns a corresponding message to the transaction that undergoes statement-level rollback.
- The statement rolled back belongs to the transaction that detects the deadlock.
- Deadlocks most often occur when transactions explicitly override the default locking of Oracle Database.
- Because Oracle Database does not escalate locks and does not use read locks for queries, but does use row-level (rather than page-level) locking, deadlocks occur infrequently.

<table>
<thead>
<tr>
<th>Transaction 1</th>
<th>Transaction 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UPDATE employees</strong></td>
<td><strong>UPDATE employees</strong></td>
</tr>
<tr>
<td><strong>SET salary = salary x 1.1</strong></td>
<td><strong>SET manager = 1342</strong></td>
</tr>
<tr>
<td><strong>WHERE employee_id = 1000;</strong></td>
<td><strong>WHERE employee_id = 2000;</strong></td>
</tr>
<tr>
<td>9:00</td>
<td>9:00</td>
</tr>
<tr>
<td><strong>UPDATE employees</strong></td>
<td><strong>UPDATE employees</strong></td>
</tr>
<tr>
<td><strong>SET salary = salary x 1.1</strong></td>
<td><strong>SET manager = 1342</strong></td>
</tr>
<tr>
<td><strong>WHERE employee_id = 2000;</strong></td>
<td><strong>WHERE employee_id = 1000;</strong></td>
</tr>
<tr>
<td>9:15</td>
<td>9:15</td>
</tr>
<tr>
<td>ORA-00060: Deadlock detected while waiting for resource</td>
<td>9:16</td>
</tr>
</tbody>
</table>
## Locks and Deadlocks

### Deadlocked Transactions

<table>
<thead>
<tr>
<th>T</th>
<th>Session 1</th>
<th>Session 2</th>
<th>Explanation</th>
</tr>
</thead>
</table>
| t0 | SQL> UPDATE employees      
    SET salary = salary*1.1 
    WHERE employee_id = 100; | SQL> UPDATE employees      
    SET salary = salary*1.1 
    WHERE employee_id = 200; | Session 1 starts transaction 1 and updates the salary for employee 100. Session 2 starts transaction 2 and updates the salary for employee 200. No problem exists because each transaction locks only the row that it attempts to update. |
|   | 1 row updated. | 1 row updated. | |
| t1 | SQL> UPDATE employees      
    SET salary = salary*1.1 
    WHERE employee_id = 200; | SQL> UPDATE employees      
    SET salary = salary*1.1 
    WHERE employee_id = 100; | Transaction 1 attempts to update the employee 200 row, which is currently locked by transaction 2. Transaction 2 attempts to update the employee 100 row, which is currently locked by transaction 1. |
|   | -- prompt does not return | -- prompt does not return | |
## Locks and Deadlocks

### (Cont.) Deadlocked Transactions

<table>
<thead>
<tr>
<th>T</th>
<th>Session 1</th>
<th>Session 2</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>t2</td>
<td><strong>UPDATE employees</strong></td>
<td></td>
<td>A deadlock results because neither transaction can obtain the resource it needs to proceed or terminate. No matter how long each transaction waits, the conflicting locks are held.</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td></td>
<td>Transaction 1 signals the deadlock and rolls back the UPDATE statement issued at t1. However, the update made at t0 is not rolled back. The prompt is returned in session 1.</td>
</tr>
<tr>
<td></td>
<td>ERROR at line 1:</td>
<td></td>
<td><strong>Note:</strong> Only one session in the deadlock actually gets the deadlock error, but either session could get the error.</td>
</tr>
<tr>
<td></td>
<td>ORA-00060: deadlock detected while waiting for resource</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SQL&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Locks and Deadlocks

#### (Cont.) Deadlocked Transactions

<table>
<thead>
<tr>
<th>T</th>
<th>Session 1</th>
<th>Session 2</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>t3</td>
<td>SQL&gt; COMMIT;</td>
<td></td>
<td>Session 1 commits the update made at t0, ending transaction 1. The update unsuccessfully attempted at t1 is not committed.</td>
</tr>
<tr>
<td></td>
<td>Commit complete.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t4</td>
<td></td>
<td>1 row updated.</td>
<td>The update at t1 in transaction 2, which was being blocked by transaction 1, is executed. The prompt is returned.</td>
</tr>
<tr>
<td></td>
<td>SQL&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t5</td>
<td>SQL&gt; COMMIT;</td>
<td></td>
<td>Session 2 commits the updates made at t0 and t1, which ends transaction 2.</td>
</tr>
<tr>
<td></td>
<td>Commit complete.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Locks and Unindexed Foreign Keys

- The number one cause of deadlocks in the Oracle database, in my experience, is unindexed foreign keys.
- If foreign keys are not indexed, then child table will probably be locked more frequently, deadlocks will occur, and concurrency will be decreased.
- For this reason foreign keys should almost always be indexed.
  1. If you update the parent table’s primary key the child table will be locked in the absence of an index on the foreign key.
  2. If you delete a parent table row, the entire child table will be locked.

```sql
> create table p ( x int primary key );
Table created.
> create table c ( x references p );
Table created.
> insert into p values ( 1 );
1 row created.
> insert into p values ( 2 );
1 row created.
> commit;
Commit complete.
> insert into c values ( 2 );
> delete from p where x = 1;
> insert into c values ( 2 );
```
- But if we go into another session and attempt to delete the first parent record, we’ll find that session gets immediately blocked. The sessions that had already started may continue, but no new sessions may start to modify C.
- It is attempting to gain a full table lock on table C before it does the delete. Now no other session can initiate a DELETE, INSERT, or UPDATE of any rows in C
- This blocking would happen with an update of the primary key value as well.
Locks and Unindexed Foreign Keys

1. DB acquires a full table lock on employees during the primary key modification of department 60.

2. This lock enables other sessions to query but not update the employees table.

3. EX: sessions cannot update employee phone numbers. The table lock on employees releases immediately after the primary key modification on the departments table completes.

4. If multiple rows in departments undergo primary key modifications, then a table lock on employees is obtained and released once for each row that is modified in departments.

Note: DML on a child table does not acquire a table lock on the parent table.
Locks and Unindexed Foreign Keys

• When both of the following conditions are true, DB acquires a full table lock on the child table:

1. **No index exists on the foreign key column of the child table.**

2. A session modifies a primary key in the parent table (for example, deletes a row or modifies primary key attributes) or merges rows into the parent table.

Note:

• Inserts into the parent table do **not** acquire blocking table locks that prevent DML on the child table.

• In the case of inserts, DB acquires a lock on the child table that prevents structural changes, but not modifications of existing or newly added rows.
Locks and indexed Foreign Keys

1. Child table employees with an indexed department_id column.
2. A transaction deletes department 280 from departments.
3. This deletion does not cause DB to acquire a full table lock on the employees table.
4. If the child table specifies ON DELETE CASCADE, then deletions from the parent table can result in deletions from the child table. EX: deletion of department 280 can cause the deletion of records from employees for employees in the deleted department.
5. In this case, waiting and locking rules are the same as if you deleted rows from the child table after deleting rows from the parent table.
Locks and indexed Foreign Keys

- When both of the following conditions are true, DB does *not* acquire a full table lock on the child table:
  1. A foreign key column in the child table is indexed.
  2. A session modifies a primary key in the parent table (Ex: deletes a row or modifies primary key attributes) or merges rows into the parent table.

- Lock on the parent table prevents transactions from acquiring exclusive table locks, but does not prevent DML on the parent or child table during the primary key modification.

- This situation is preferable if primary key modifications occur on the parent table while updates occur on the child table.
Lock Escalation/Conversion

- System is decreasing the granularity of your locks
- Ex: A DELETE operation on a table against **20,000** rows. For more than 5,000 locks on one level in your locking hierarchy, SQL Server escalates into a simple **coarse-granularity lock**.

- Oracle will never escalate a lock. Never.
Locks Types

• **DML locks**
  – DML: SELECT, INSERT, UPDATE, MERGE, and DELETE statements
  – Allows for concurrent data modifications
  – Example locks on a specific row of data or a lock at the table level to lock every row in the table.

• **DDL locks**
  – *Data Definition Language*: CREATE and ALTER statements....
  – *Protect the structure of schema objects for example, the dictionary definitions* of tables and views.

• **System locks**
  – Protect Oracle internal data structures.
  – EX: Oracle parses a query and generates an optimized query plan, it will latch the library cache to put that plan in there for other sessions to use.
  – A latch is a light weight, low-level serialization device employed by Oracle, similar in function to a lock.
  – Lightweight in their implementation, but not their effect
  – **Users have no control over their occurrence or duration.**
DML Locks

- A DML lock (a data lock) guarantees the integrity of data accessed concurrently by multiple users.
  - Ex: a DML lock prevents two customers from buying the last copy of a book available from an online bookseller.
- Each DML transaction must acquire two locks:
  1. Row Locks (TX)
  2. Table Locks (TM)
- Oracle places a TM lock to prevent other sessions from dropping that table (or performing most DDL, in fact).
- TX locks EXCLUSIVE row lock on the row or rows being updated

Transaction 1

```
SQL> UPDATE employees
2  SET salary=salary*1.1
3  WHERE employee_id= 107;
1 row updated.
```

Transaction 2

```
SQL> UPDATE employees
2  SET salary=salary*1.1
3  WHERE employee_id= 106;
1 row updated.
```
Row Locks (TX)

- A row lock, also called a **TX lock**, is a lock on a single row of table.
- A *transaction* acquires a row lock for each row modified by an INSERT, UPDATE, DELETE, MERGE, or SELECT ... FOR UPDATE statement.
- Row lock exists until the transaction commits or rolls back.
- Row locks primarily serve as a queuing mechanism to prevent two transactions from modifying the same row.
- DB always locks a modified row in exclusive mode so that other transactions cannot modify the row until the transaction holding the lock commits or rolls back.
- Row locking provides the finest grain locking possible and so provides the best possible concurrency and throughput.
- Note: If a transaction terminates because of database instance failure, then block level recovery makes a row available before the entire transaction is recovered.
Read Consistency and Interested Transaction Lists (ITL)

- Block header of every segment block contains an ITL.
- DB uses the ITL to determine whether a transaction was uncommitted when the database began modifying the block.
- Entries in the ITL describe which transactions have rows locked and which rows in the block contain committed and uncommitted changes.
- The ITL points to the transaction table in the undo segment, which provides information about the timing of changes made to the database.
- Block header contains a recent history of transactions that affected each row in the block.
- The INITRANS and MAXTRANS: parameters of the CREATE TABLE and ALTER TABLE statements controls the amount of transaction history
Read Consistency and Interested Transaction Lists (ITL)

1. Typical data block right after the creation of the table.

2. We inserted three rows into the table.

3. Transaction Txn1 updates Row1, but does not commit. This locks Row1, and the transaction places the lock in the slot number one in the ITL.

4. Another transaction, Txn2, updates the row Row2 and wants to lock the row. The maxtrans entry is 11, meaning the ITL can grow up to 11 slots and the block has empty space.

3. Transaction Txn1 updates Row1, but does not commit. This locks Row1, and the transaction places the lock in the slot number one in the ITL.
Storage of Row Locks : Oracle

- Oracle DB stores lock information in data block that contains the locked row.

Oracle’s process looks like this:
- 1. Find the address of the row you want to lock.
- 2. Go to the row.
- 3. Lock the row right there, right then—at the location of the row, (waiting for the transaction that has it locked to end if it is already locked, unless you are using the NOWAIT option).
- **Oracle does not need a traditional lock manager**  Like other RDBMs
- The transaction will simply go to the data and lock it (if it is not locked already).
- DB uses a queuing mechanism for acquisition of row locks.
- If a transaction requires a lock for an unlocked row, then the transaction places a lock in the data block.
- Each row modified by this transaction points to a copy of the transaction ID stored in the block header
Storage of Row Locks: Oracle

- Data may appear locked when you get to it, even if it’s not.
- When you lock rows of data in Oracle, the row points to a copy of the transaction ID that is stored with the block containing the data.
- When the lock is released, the `transactionID` is left behind.
- This `transaction ID` is unique to your transaction and represents the undo segment number, slot, and sequence number.
- You leave that on the block that contains your row to tell other sessions that you own this data (not all of the data on the block—just the one row you are modifying).
- When another session comes along, it sees the transaction ID and can quickly see if the transaction holding the lock is still active.
  - If the lock is not active, the session is allowed access to the data.
  - If the lock is still active, that session will ask to be notified as soon as the lock is released.
- Queuing mechanism: the session requesting the lock will be queued up waiting for that transaction to complete, and then it will get the data.
TX Locks 1

Example:

- **V$TRANSACTION**, which contains an entry for every active transaction.
- **V$SESSION**, which shows the sessions logged in.
- **V$LOCK**, which contains an entry for all enqueue locks being held as well as for sessions that are waiting on locks.

First, let's get a copy of the EMP and DEPT tables.

```
EODA@ORA12CR1> create table dept
2 as select * from scott.dept;
Table created.
EODA@ORA12CR1> create table emp
2 as select * from scott.emp;
Table created.
EODA@ORA12CR1> alter table dept
2 add constraint dept_pk
3 primary key(deptno);
Table altered.
EODA@ORA12CR1> alter table emp
2 add constraint emp_pk
3 primary key(empno);
Table altered.
```
EODA@ORA12CR1> alter table emp
2  add constraint emp_fk_dept
3  foreign key (deptno)
4  references dept(deptno);
Table altered.
EODA@ORA12CR1> create index emp_deptno_idx
2  on emp(deptno);
Index created.
•  Let’s start a transaction now:
EODA@ORA12CR1> update dept
2  set dname = initcap(dname);
4 rows updated.
EODA@ORA12CR1> select username,
2 v$lock.sid,
3 trunc(id1/power(2,16)) rbs,
4 bitand(id1,to_number('ffff','xxxx'))+0 slot,
5 id2 seq,
6 lmode,
7 request
8 from v$lock, v$session
9 where v$lock.type = 'TX'
10 and v$lock.sid = v$session.sid
11 and v$session.username = USER;

USERNAME    SID   RBS   SLOT   SEQ   LMODE   REQUEST
-----------  -----  -----  ------  ----  ------  -------
EODA        22     2    27     21201  6      0        
V$LOCK table : LMODE=6 is an exclusive lock. A value of 0 in the request means you have the lock.

Only one row in this table. This V$LOCK table is more of a queuing table than a lock table. Many people expect four rows in V$LOCK since we have four rows locked. Remember, however, that Oracle does not store a master list of every row locked anywhere. To find out if a row is locked, we must go to that row.

I took the ID1 and ID2 columns and performed some manipulation on them.

RBS, SLOT, and SEQ values match the V$TRANSACTION information. This is my transaction ID.
EODA@ORA12CR1> select username, sid, trunc(id1/power(2,16)) rbs, bitand(id1,to_number('ffff','xxxx'))+0 slot, id2 seq, lmode, request from v$lock, v$session where v$lock.type = 'TX' and v$lock.sid = v$session.sid and v$session.username = USER;

<table>
<thead>
<tr>
<th>USERNAME</th>
<th>SID</th>
<th>RBS</th>
<th>SLOT</th>
<th>SEQ</th>
<th>LMODE</th>
<th>REQUEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>EODA</td>
<td>22</td>
<td>2</td>
<td>27</td>
<td>21201</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

EODA@ORA12CR1> select XIDUSN, XIDSLOT, XIDSQN from v$transaction;

<table>
<thead>
<tr>
<th>XIDUSN</th>
<th>XIDSLOT</th>
<th>XIDSQN</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>27</td>
<td>21201</td>
</tr>
</tbody>
</table>
• Now we’ll start another session using the same username, update some rows in EMP, and then try to update DEPT:

```sql
EODA@ORA12CR1> update emp set ename = upper(ename);
14 rows updated.
EODA@ORA12CR1> update dept set deptno = deptno-10;
```

• We’re now blocked in this session.
If we run the V$ queries again, we see the following:

```
EODA@ORA12CR1> select username, v$lock.sid, trunc(id1/power(2,16)) rbs, bitand(id1,to_number('ffff','xxxx'))+0 slot, id2 seq, lmode, request from v$lock, v$session where v$lock.type = 'TX' and v$lock.sid = v$session.sid and v$session.username = USER;
```

<table>
<thead>
<tr>
<th>USERNAME</th>
<th>SID</th>
<th>RBS</th>
<th>SLOT</th>
<th>SEQ</th>
<th>LMODE</th>
<th>REQUEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>EODA</td>
<td>17</td>
<td>2</td>
<td>27</td>
<td>21201</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>EODA</td>
<td>22</td>
<td>2</td>
<td>27</td>
<td>21201</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>EODA</td>
<td>17</td>
<td>8</td>
<td>17</td>
<td>21403</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

A new transaction has begun, with a transaction ID of (8,17,21403).

Our new session, SID=17, has two rows in V$LOCK this time. One row represents the locks that it owns (where LMODE=6).

It also has a row that shows a REQUEST with a value of 6. This is a request for an exclusive lock.

The interesting thing to note here is that the RBS/SLOT/SEQ values of this request row are the transaction ID of the holder of the lock.
TX Locks 8

EODA@ORA12CR1> select XIDUSN, XIDSLOT, XIDSQN from v$transaction;

<table>
<thead>
<tr>
<th>XIDUSN</th>
<th>XIDSLOT</th>
<th>XIDSQN</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>27</td>
<td>21201</td>
</tr>
<tr>
<td>8</td>
<td>17</td>
<td>21403</td>
</tr>
</tbody>
</table>
EODA@ORA12CR1> select
2 (select username from v$session where sid=a.sid) blocker,
3 a.sid,
4 ' is blocking ',
5 (select username from v$session where sid=b.sid) blockee,
6 b.sid
7 from v$lock a, v$lock b
8 where a.block = 1
9 and b.request > 0
10 and a.id1 = b.id1
11 and a.id2 = b.id2;

<table>
<thead>
<tr>
<th>BLOCKER</th>
<th>SID</th>
<th>'ISBLOCKING'</th>
<th>BLOCKEE</th>
<th>SID</th>
</tr>
</thead>
<tbody>
<tr>
<td>EODA</td>
<td>22</td>
<td>is blocking</td>
<td>EODA</td>
<td>17</td>
</tr>
</tbody>
</table>

Transaction with
SID=22 is blocking the transaction with SID=17 a self-join of V$LOCK
EODA@ORA12CR1> select username,
2 v$lock.sid,
3 trunc(id1/power(2,16)) rbs,
4 bitand(id1,to_number('ffff','xxxx')+0 slot,
5 id2 seq,
6 lmode,
7 request
8 from v$lock, v$session
9 where v$lock.type = 'TX'
10 and v$lock.sid = v$session.sid
11 and v$session.username = USER;

Now, if we commit our original transaction, SID=22, and rerun our lock query, we find that the request row has gone:

USERNAME  SID  RBS  SLOT  SEQ  LMODE REQUEST
----------  ----  -----  -----  -----  ------  -------
EODA       17    8    17    21403  6    0
EODA@ORA12CR1> select XIDUSN, XIDSLOT, XIDSQN from v$transaction;

<table>
<thead>
<tr>
<th>XIDUSN</th>
<th>XIDSLOT</th>
<th>XIDSQN</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>17</td>
<td>21403</td>
</tr>
</tbody>
</table>

Now, if we commit our original transaction, SID=22, and rerun our lock query, we find that the request row has gone: