Lecture 2 & 3:
Oracle 12c Database SQL Server  Data Concurrency: Transactions and Locking
B+ Trees

By David Itshak
shaked19@gmail.com

[Links to websites and YouTube videos]
Reference and Credits

**Oracle® Database** Concepts
12c Release 1 (12.1)
E41396-13
https://docs.oracle.com/database/121/CNCPT/toc.htm

**Oracle® Database** Performance Tuning Guide
12c Release 1 (12.1)
E49058-06
https://docs.oracle.com/database/121/TGDBA/toc.htm

**Oracle® Database SQL Language Reference**
12c Release 1 (12.1)
E41329-20

Oracle Essentials(Oracle Database 12c), 5th; O'Reilly, 2013

Oracle OCA Oracle Database 12c Administrator Certified Associate Study Guide Exam

Pro Oracle Database 12c Administration, 2 edition ISBN 1430257288 2013

Apress Oracle Database Transactions and Locking Revealed (2014)

Oracle Learning Library

Pro SQL Server Internals 2014  Apress
Agenda: Data Concurrency and Consistency

- Transactions
- Undo
- Data Concurrency and Consistency
- Pessimistic Locking, Optimistic Locking
- Multiversion Read Consistency
- Locking Mechanisms
- ANSI/ISO Transaction Isolation Levels
- Overview of Oracle Database Transaction Isolation Levels
- Read Committed Isolation Level
- Serializable Isolation Level
- Read-Only Isolation Level
- Overview of the Oracle Database Locking Mechanism
Agenda: Data Concurrency and Consistency

- Summary of Locking Behavior
- Use of Locks
- Lock Modes
- Lock Conversion and Escalation
- Lock Duration
- Locks and Deadlocks
- Overview of Automatic Locks
- DML
- DDL Locks
- System Locks
- Overview of Manual Data Locks
- Overview of User-Defined Locks
- Review questions
Transactions

• A transaction is a set of DML statements executed sequentially by a session.
• 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.
• Starts with the first of the following statements executed by the session:
  – INSERT
  – UPDATE
  – DELETE
  – MERGE
  – SELECT FOR UPDATE
  – LOCK TABLE
• Ends with either a COMMIT or ROLLBACK
Transactions

- **Atomicity**: A transaction is treated as a single unit of work. Either it completes entirely, or the system has no "memory" of it happening at all.

- **Consistency**: A transaction will leave data in a meaningful state when it completes. In RDBMS, all constraints will be applied to the transaction's modifications to maintain data integrity. Internal data structures, such as the trees and linked lists used for maintaining indexes, will be correct at the end of a transaction.

- **Isolation**: Effects of a transaction may not be visible to other transactions until the transaction has committed.

- **Durability**: Once a transaction completes, its effects are permanent and recoverable.
Transactions Control Statements

1. **COMMIT**
   
2. **ROLLBACK**
   
3. **SAVEPOINT:** Allows you to create a marked point within a transaction. You may have multiple SAVEPOINTs within a single transaction.
   
4. **ROLLBACK TO **<SAVEPOINT>:** This statement is 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.

- You should always explicitly terminate your transactions with a COMMIT or ROLLBACK
- A transaction implicitly begins with the first statement that modifies data (the first statement that gets a TX lock)
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:

```
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;
```
Undo

• Undo is conceptually the opposite of redo.
• Undo information is generated by the database as you make modifications to data so that the data can be put back the way it was before the modifications took place.

• Support of multiversioning, or in the event the transaction or statement you are executing fails for any reason, or if we request it with a ROLLBACK statement.

• Redo is used to replay a transaction in the event of failure—to recover the transaction—Undo is used to reverse the effects of a statement or set of statements.

• Undo, unlike redo, is stored internally in the database in a special set of segments known as undo segments.
Undo Segments and Transactions

- When a transaction starts, DB assigns transaction to an undo segment, and to a transaction table, in current undo tablespace.
- Multiple active transactions can write concurrently to the same undo segment or to different segments.
- EX: transactions T1 and T2 can both write to undo segment U1, or T1 can write to U1 while T2 writes to undo segment U2.
- Undo segment form a ring. Transactions write to one undo extent, and then to the next extent in the ring, and so on in cyclical fashion.
## Data Concurrency

| Time: 09:00:00 | Transaction 1 | UPDATE hr.employees  
|               |              | SET salary=salary+100  
|               |              | WHERE employee_id=100;|
| Transaction 2 |              | UPDATE hr.employees  
|               |              | SET salary=salary+100  
|               |              | WHERE employee_id=101;|
| Transaction 3 |              | UPDATE hr.employees  
|               |              | SET salary=salary+100  
|               |              | WHERE employee_id=102;|
| ...           | ...          | ...          |
| Transaction x |              | UPDATE hr.employees  
|               |              | SET salary=salary+100  
|               |              | WHERE employee_id=xxx;|
Concurrency and Transactions

- The ANSI (American National Standards Institute) SQL Standard defines three phenomena: **dirty reads, non-repeatable reads and phantom reads.**
  - Can be allowed or prevented, depending on the ANSI-standard transaction isolation level in use: **READ UNCOMMITTED, READ COMMITTED (the default), REPEATABLE READ, or SERIALIZABLE**

- **Lost updates** – One session accidentally overwrites modifications performed by another

- **Excessive blocking** – A "queue" of blocked processes forms, causing pressure on the resource and unacceptable delays to end-users

- **Deadlocks** – Mutual blocking between sessions such that further progress is impossible. Oracle will choose one of the deadlocked sessions as the "victim," roll it back, and issue an error message to the affected client.
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.

This process is referred to as a *lost update* because all of the changes made in Step 3 will be lost.

\[ s = r_1(x)r_2(x)w_1(x)c_1w_2(x)c_2 \]
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:

```
SCOTT@ORA12CR1> 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.

```
SCOTT@ORA12CR1> variable empno number
SCOTT@ORA12CR1> variable ename varchar2(20)
SCOTT@ORA12CR1> variable sal number
SCOTT@ORA12CR1> exec :empno := 7934; :ename := 'MILLER'; :sal := 1300;
PL/SQL procedure successfully completed.
```
Pessimistic Locking- Cont

• we are going to lock the row using FOR UPDATE NOWAIT

```
SCOTT@ORA12CR1> select empno, ename, sal
2 from emp
3 where empno = :empno
4 and decode( ename, :ename, 1 ) = 1
5 and decode( sal, :sal, 1 ) = 1
6 for update nowait
7 /
```

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
<th>SAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>1300</td>
</tr>
</tbody>
</table>
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
2 set ename = :ename, sal = :sal
3 where empno = :empno;
1 row updated.
SCOTT@ORA12CR1> 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

```
Update table
Set column1 = :new_column1, column2 = :new_column2, ....
Where primary_key = :primary_key
And decode( column1, :old_column1, 1 ) = 1
And decode( column2, :old_column2, 1 ) = 1
```

Other Options :
• Optimistic Locking Using a Version Column (systimestamp column)
• Optimistic Locking Using a Checksum
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 $w_2(x)$ changes $x$ in a way that $x$ satisfies $P$.

- schedule $S = r_1(P)w_2(x)c_2r_1(P)c_1$
Data Concurrency and Consistency

- In a multiuser DB, statements within multiple simultaneous transactions may update the same data.

- Transactions executing simultaneously must produce meaningful and consistent results.

- A multiuser database must provide the following:
  - **Data concurrency** - Users can access data at the same time.
  - **Data consistency** - User sees a consistent view of the data visible changes made by user's own transactions and committed transactions of other users

- **Serializability** - A serializable transaction operates in an environment that makes it appear as if no other users were modifying data in the database.
Data Concurrency and Consistency

• Complete isolation of concurrently running transactions could mean that one transaction cannot perform an insertion into a table being queried by another transaction.

• In real-world trade of between perfect transaction isolation and performance.

• Oracle DB maintains data consistency by using a multi version consistency model and various types of locks and transactions.

• Oracle can present a view of data to multiple concurrent users, with each view consistent to a point in time.

• Because different versions of data blocks can exist simultaneously, transactions can read the version of data committed at the point in time required by a query and return results that are consistent to a single point in time.
Oracle Multiversion Read Consistency

• **Read-consistent queries**
  – Data returned by a query is committed and consistent for a single point in time
  – **Note**: Oracle DB never permits a dirty read, which occurs when a transaction reads uncommitted data in another transaction.

• **Nonblocking queries**
  – Readers and writers of data do not block one another.
Read Consistency and Undo Segments

• To manage the multiversion read consistency model, the database must create a read consistent set of data when a table is simultaneously queried and updated. Oracle Database achieves this goal through undo data.

• Guarantees that data returned by a single query is committed and consistent for a single point in time.
  • Depends on the transaction isolation level and the query:

• In the read committed isolation level, point is time at which the statement was opened.
  – EX: if a SELECT statement opens at SCN 1000, then this statement is consistent to SCN 1000.

• In a serializable or read-only transaction, this point is the time the transaction began.
  – EX: if a transaction begins at SCN 1000, and if multiple SELECT statements occur in this transaction, then each statement is consistent to SCN 1000

• In a Flashback Query operation (SELECT ... AS OF), SELECT statement explicitly specifies the point in time.
  – EX: you can query a table as it appeared last Thursday at 2 p.m.
SCN coordination

• An Oracle DB uses the System Change Number (SCN) to keep track of transactions.
  • For every commit, a new **SCN** is assigned.
    – The data changes and **SCN** are written to DB’s online redo logs.
    – Oracle requires these logs for crash recovery, which allows the committed transactions to be recovered (uncommitted transactions are rolled back).
System Change Numbers (SCNs)

- A logical, internal time stamp used by Oracle Database.
- SCNs order events that occur within the database: satisfy the ACID properties of a transaction.

- Oracle Database can use an SCN like a clock because an observed SCN indicates a logical point in time. Several events may share the same SCN, which means that they occurred at the same time in the database.

- Every transaction has an SCN. Ex: if a transaction updates a row, then the database records the SCN at which this update occurred. Other modifications in this transaction have the same SCN.

- When a transaction commits, the database records an SCN for this commit.

- Oracle Database increments SCNs in the system global area (SGA). When a transaction modifies data, the database writes a new SCN to the undo data segment assigned to the transaction. The log writer process then writes the commit record of the transaction immediately to the online redo log. The commit record has the unique SCN of the transaction.

- Oracle Database also uses SCNs as part of its instance recovery and media recovery mechanisms.
Examples

• Examples in this presentation are based on cricket

◆ The following table has been used in all examples in this presentation

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TEAM</td>
<td>VARCHAR2(30)</td>
</tr>
<tr>
<td>RUNS</td>
<td>NUMBER</td>
</tr>
<tr>
<td>WICKETS</td>
<td>NUMBER</td>
</tr>
</tbody>
</table>

◆ The table has no indexes
Flashback Query and SCN

**Session 1**

```
SELECT runs
FROM score
WHERE team = 'ENG';
Runs
137
```

```
UPDATE team
SET runs = 141
WHERE team = 'ENG';
COMMIT;
```

```
SELECT runs
FROM score
WHERE team = 'ENG';
```

```
SELECT dbms_flashback.get_system_change_number FROM dual;
SCN
3494824
```

**Session 2**

```
SELECT dbms_flashback.get_system_change_number FROM dual;
SCN
3494833
```

```
SELECT team, runs, wickets FROM score AS OF SCN 3494824;
```

```
Team  Runs  Wickets
ENG  141   1
```

```
SELECT team, runs, wickets FROM score
WHERE team = 'ENG';
```

```
Team  Runs  Wickets
ENG  137   1
```

```
SELECT dbms_flashback.get_system_change_number FROM dual;
SCN
3494833
```

```
SELECT team, runs, wickets FROM score AS OF SCN 3494833;
```

```
Team  Runs  Wickets
ENG  141   1
```
Flashback Query

• Can specify **AS OF** clause:
  – Returns single-row
  – Syntax is

  \[
  \text{AS OF [ SCN <scn> | TIMESTAMP <timestamp> ]}
  \]

  ◆ For example:

  ```sql
  SELECT team, runs, wickets
  FROM score AS OF SCN 3506431 WHERE team = 'ENG';
  ```

  ```sql
  SELECT team, runs, wickets
  FROM score AS OF TIMESTAMP to_timestamp('01-OCT-2004 10:53:47.000');
  ```
Flashback Query

• Can also specify `VERSIONS` clause:

  – Returns multiple rows

  VERSIONS BETWEEN SCN [ <scn> | MINVALUE ] AND [ <scn> | MAXVALUE ]

  VERSIONS BETWEEN TIMESTAMP [ <timestamp> | MINVALUE ] AND [ <timestamp> | MAXVALUE ]

  ◆ For example:

  ```sql
  SELECT team, runs, wickets
  FROM score
  VERSIONS BETWEEN SCN 3503511 AND 3503524
  WHERE team = 'ENG';
  ```
Read Consistency in the Read Committed Isolation Level

1. Client A sends a SQL SELECT
2. Server process obtains an SCN for the statement. If the server finds a transaction with a later SCN than the current SELECT statement, server process uses data in the UNDO segments to create a “consistent read”
3. Client B sends a SQL UPDATE that has not yet been read by Client A’s SELECT statement.
   - Server process gets an SCN for the statement and begins the operation 3.
4. Client B commits his changes.
   - Server records information in the data block that contains the modified row that allows Oracle to determine the SCN for the update transaction.
5. Server process for Client A’s read operation comes to the newly modified block. UNDO segment uses the old version of the data to create a version of the block as it existed when the SELECT statement started.
Reading with multiversion read consistency

- Client A is reading rows from the EMP table, while Client B modifies a row before it is read by Client A, but after Client A begins her transactions.
- 1. Client A sends a SQL SELECT statement.
- 2. 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, 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.
  - Multiversion read consistency (MVRC): avoid 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.
Reading with multiversion read consistency

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.
   - Server process gets an SCN for the statement and begins the operation 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.

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. 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. 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.
   - 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.
Read Consistency in the Read Committed Isolation Level

- DB retrieves data blocks on behalf of a query, DB ensures that the data in each block reflects the contents of the block when the query began.
  - DB rolls back changes to the block as needed to reconstruct the block to the point in time the query started processing.
- DB determines the SCN recorded at the time the query began executing. In example SCN is 10023.
- The query only sees committed data with respect to SCN 10023.
- DB creates two CR clones: one block consistent to **SCN 10006** and **SCN 10021**
ANSI/ISO Transaction Isolation Levels

- **Dirty reads**
  A transaction reads data that has been written by another transaction that has not been committed yet.

- **Nonrepeateable (fuzzy) reads**
  - A transaction rereads data it has previously read and finds that another committed transaction has modified or deleted the data. Ex: a user queries a row and then later queries the same row, only to discover that the data has changed.

- **Phantom reads**
  - A transaction reruns a query returning a set of rows that satisfies a search condition and finds that another committed transaction has inserted additional rows that satisfy the condition.

<table>
<thead>
<tr>
<th>Preventable Read Phenomena by Isolation Level</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Isolation Level</strong></td>
</tr>
<tr>
<td>---------------------</td>
</tr>
<tr>
<td>Read uncommitted</td>
</tr>
<tr>
<td>Read committed</td>
</tr>
<tr>
<td>Repeatable read</td>
</tr>
<tr>
<td>Serializable</td>
</tr>
</tbody>
</table>
Overview of Oracle Database
Transaction Isolation Levels

Oracle Database provides the transaction isolation levels:

• Read Committed Isolation Level
• Serializable Isolation Level
• Read-Only Isolation Level
Read Committed Isolation

- Oracle Default
- Every query executed by a transaction sees only data committed before the query—not the transaction—began.
- Appropriate for DB environments in which few transactions are likely to conflict.
- A query in a read committed transaction avoids reading data that commits while the query is in progress.
- If a query is halfway through a scan of a million row table, and if a different transaction commits an update to row 950,000, then the query does not see this change when it reads row 950,000.
  - However, DB not prevent other transactions from modifying data read by a query, other transactions may change data between query executions.
  - A transaction that runs the same query twice may experience fuzzy reads and phantoms.
Read Consistency in the Read Committed Isolation Level

• DB provides a consistent result set for every query, guaranteeing data consistency, with no action by the user.
• An implicit query, such as a query implied by a WHERE clause in an UPDATE statement, is guaranteed a consistent set of results.
• Each statement in an implicit query does not see the changes made by the DML statement itself, but sees the data as it existed before changes were made.
• If a SELECT list contains a PL/SQL function, DB applies statement-level read consistency at the statement level for SQL run within the PL/SQL function code, rather than at the parent SQL level.
  – For example, a function could access a table whose data is changed and committed by another user. For each execution of the SELECT in the function, a new read-consistent snapshot is established.
Session 1 queries the salaries for Banda, Greene, and Hintz. No employee named Hintz is found.

```
SQL> SELECT last_name, salary FROM employees
WHERE last_name IN ('Banda', 'Greene', 'Hintz');

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banda</td>
<td>6200</td>
</tr>
<tr>
<td>Greene</td>
<td>9500</td>
</tr>
</tbody>
</table>
```

Session 1 begins a transaction by updating the Banda salary. The default isolation level for transaction 1 is READ COMMITTED.

```
SQL> UPDATE employees
SET salary = 7600 WHERE last_name = 'Banda';
```

Session 2 begins transaction 2 and sets the isolation level explicitly to READ COMMITTED.

```
SQL> SET TRANSACTION
ISOLATION LEVEL
READ COMMITTED;
```

Transaction 2 queries the salaries for Banda, Greene, and Hintz. Oracle Database uses read consistency to show the salary for Banda before the uncommitted update made by transaction 1.

```
SQL> SELECT last_name, salary FROM employees
WHERE last_name IN ('Banda', 'Greene', 'Hintz');

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banda</td>
<td>6200</td>
</tr>
<tr>
<td>Greene</td>
<td>9500</td>
</tr>
</tbody>
</table>
```

Transaction 2 updates the salary for Greene successfully because transaction 1 locked only the Banda row (see “Row Locks (TX)”).

```
SQL> UPDATE employees
SET salary = 9900
WHERE last_name = 'Greene';
```

Transaction 1 inserts a row for employee Hintz, but does not commit.

```
SQL> INSERT INTO employees (employee_id, last_name, email, hire_date, job_id)
VALUES (210, 'Hintz', 'JHINTZ', SYSDATE, 'SH_CLERK');
```
<table>
<thead>
<tr>
<th>Session 1</th>
<th>Session 2</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL&gt; SELECT last_name, salary FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz');</td>
<td></td>
<td>Transaction 2 queries the salaries for employees Banda, Greene, and Hintz.</td>
</tr>
<tr>
<td>LAST_NAME</td>
<td>SALARY</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>----------</td>
<td>----------------</td>
</tr>
<tr>
<td>Banda</td>
<td>6200</td>
<td>Transaction 2 sees its own update to the salary for Greene.</td>
</tr>
<tr>
<td>Greene</td>
<td>9900</td>
<td>Transaction 2 does not see the uncommitted update to the salary for Banda or the insertion for Hintz made by transaction 1.</td>
</tr>
<tr>
<td>SQL&gt; UPDATE employees SET salary = 6300 WHERE last_name = 'Banda';</td>
<td></td>
<td>Transaction 2 attempts to update the row for Banda, which is currently locked by transaction 1, creating a conflicting write.</td>
</tr>
<tr>
<td>-- prompt does not return</td>
<td></td>
<td>Transaction 2 waits until transaction 1 ends.</td>
</tr>
<tr>
<td>SQL&gt; COMMIT;</td>
<td></td>
<td>Transaction 1 commits its work, ending the transaction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The lock on the Banda row is now released, so transaction 2 proceeds with its update to the salary for Banda.</td>
</tr>
<tr>
<td>SQL&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL&gt; SELECT last_name, salary FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz');</td>
<td></td>
<td>Transaction 2 queries the salaries for employees Banda, Greene, and Hintz. The Hintz insert committed by transaction 1 is now visible to transaction 2.</td>
</tr>
<tr>
<td>LAST_NAME</td>
<td>SALARY</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>----------</td>
<td>----------------</td>
</tr>
<tr>
<td>Banda</td>
<td>6300</td>
<td>Transaction 2 sees its own update to the Banda salary.</td>
</tr>
<tr>
<td>Greene</td>
<td>9900</td>
<td></td>
</tr>
<tr>
<td>Hintz</td>
<td>9900</td>
<td></td>
</tr>
<tr>
<td>COMMIT;</td>
<td></td>
<td>Transaction 2 commits its work, ending the transaction.</td>
</tr>
</tbody>
</table>
(Cont.) Conflicting Writes and Lost Updates in a READ COMMITTED Transaction

<table>
<thead>
<tr>
<th>Session 1</th>
<th>Session 2</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL&gt; SELECT last_name,</td>
<td></td>
<td>Session 1 queries the rows for Banda, Greene, and Hintz. The salary for</td>
</tr>
<tr>
<td>salary FROM employees</td>
<td></td>
<td>Banda is 6300, which is the update made by transaction 2. The update of</td>
</tr>
<tr>
<td>WHERE last_name IN</td>
<td></td>
<td>Banda's salary to 7000 made by transaction 1 is now &quot;lost.&quot;</td>
</tr>
<tr>
<td>('Banda', 'Greene',</td>
<td></td>
<td></td>
</tr>
<tr>
<td>'Hintz');</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banda</td>
<td>6300</td>
</tr>
<tr>
<td>Greene</td>
<td>9900</td>
</tr>
<tr>
<td>Hintz</td>
<td></td>
</tr>
</tbody>
</table>
Serializable Isolation Level

- A transaction sees only changes committed at the time the transaction—not the query—began and changes made by the transaction itself.
- A serializable transaction operates in an environment that makes it appear as if no other users were modifying data in the database.
- Serializable isolation is suitable for environments:
  - With large databases and short transactions that update only a few rows
  - Where the chance that two concurrent transactions will modify the same rows is relatively low
- Where relatively long-running transactions are primarily read only
Serializable Isolation Level

• In serializable isolation, the read consistency normally obtained at the statement level extends to the entire transaction.

• **Any row read by the transaction is assured to be the same when reread.**

• Any query is guaranteed to return the same results for the duration of the transaction, so changes made by other transactions are not visible to the query regardless of how long it has been running.

• **Serializable transactions do not experience dirty reads, fuzzy reads, or phantom reads.**
Serializable Isolation Level

• **Oracle DB** permits a serializable transaction to modify a row only if changes to the row made by other transactions were *already* committed when the serializable transaction began.

• DB generates an error when a serializable transaction tries to update or delete data changed by a different transaction that committed *after the serializable transaction began*:
  – **ORA-08177**: Cannot serialize access for this transaction

• Application can take several actions:
  – Commit the work executed to that point
  – Execute additional (but different) statements, perhaps after rolling back to a save point established earlier in the transaction
  – Roll back entire transaction.
## Serializable Transaction

<table>
<thead>
<tr>
<th>Session 1</th>
<th>Session 2</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SQL&gt; SELECT last_name, salary</code></td>
<td></td>
<td>Session 1 queries the salaries for Banda, Greene, and Hintz. No employee named Hintz is found.</td>
</tr>
<tr>
<td>FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz');</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAST_NAME</td>
<td>SALARY</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td><code>SQL&gt; UPDATE employees</code></td>
<td></td>
<td>Session 2 begins transaction 2 and sets it to the SERIALIZABLE isolation level.</td>
</tr>
<tr>
<td>SET salary = 7000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WHERE last_name='Banda';</td>
<td></td>
<td>Transaction 2 queries the salaries for Banda, Greene, and Hintz. Oracle Database uses read consistency to show the salary for Banda before the uncommitted update made by transaction 1.</td>
</tr>
<tr>
<td><code>SQL&gt; SET TRANSACTION</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ISOLATION LEVEL SERIALIZABLE;</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>SQL&gt; SELECT last_name, salary</code></td>
<td></td>
<td>Transaction 2 updates the Greene salary successfully because only the Banda row is locked.</td>
</tr>
<tr>
<td>FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz');</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAST_NAME</td>
<td>SALARY</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td><code>SQL&gt; UPDATE employees</code></td>
<td></td>
<td>Transaction 1 inserts a row for employee Hintz.</td>
</tr>
<tr>
<td>SET salary = 9900</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WHERE last_name = 'Greene';</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>SQL&gt; INSERT INTO employees</code></td>
<td></td>
<td>Transaction 1 commits its work, ending the transaction.</td>
</tr>
<tr>
<td>(employee_id, last_name, email, hire_date, job_id) VALUES (210, 'Hintz', 'JHINTZ', SYSDATE,'SR_CLERK');</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>SQL&gt; COMMIT;</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Serializable Transaction

<table>
<thead>
<tr>
<th>Session 1</th>
<th>Session 2</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL: SELECT last_name, salary</td>
<td>SQL: SELECT last_name, salary</td>
<td>Session 1 queries the salaries for employees Banda, Greene, and Hintz and sees changes committed by transaction 1. Session 1 does not see the uncommitted Greene update made by transaction 2. Transaction 2 queries the salaries for employees Banda, Greene, and Hintz. Oracle Database read consistency ensures that the Hintz insert and Banda update committed by transaction 1 are not visible to transaction 2. Transaction 2 sees its own update to the Greene salary.</td>
</tr>
<tr>
<td>FROM employees</td>
<td>FROM employees</td>
<td></td>
</tr>
<tr>
<td>WHERE last_name IN ('Banda', 'Greene', 'Hintz');</td>
<td>WHERE last_name IN ('Banda', 'Greene', 'Hintz');</td>
<td></td>
</tr>
<tr>
<td><strong>LAST_NAME</strong></td>
<td><strong>SALARY</strong></td>
<td><strong>LAST_NAME</strong></td>
</tr>
<tr>
<td>--------</td>
<td>---------</td>
<td>--------</td>
</tr>
<tr>
<td>Banda</td>
<td>7000</td>
<td>Banda</td>
</tr>
<tr>
<td>Greene</td>
<td>9500</td>
<td>Greene</td>
</tr>
<tr>
<td>Hintz</td>
<td></td>
<td>Hintz</td>
</tr>
<tr>
<td>COMMIT;</td>
<td></td>
<td>COMMIT;</td>
</tr>
<tr>
<td>SQL: SELECT last_name, salary</td>
<td>SQL: SELECT last_name, salary</td>
<td></td>
</tr>
<tr>
<td>FROM employees</td>
<td>FROM employees</td>
<td></td>
</tr>
<tr>
<td>WHERE last_name IN ('Banda', 'Greene', 'Hintz');</td>
<td>WHERE last_name IN ('Banda', 'Greene', 'Hintz');</td>
<td></td>
</tr>
<tr>
<td><strong>LAST_NAME</strong></td>
<td><strong>SALARY</strong></td>
<td><strong>LAST_NAME</strong></td>
</tr>
<tr>
<td>--------</td>
<td>---------</td>
<td>--------</td>
</tr>
<tr>
<td>Banda</td>
<td>7000</td>
<td>Banda</td>
</tr>
<tr>
<td>Greene</td>
<td>9900</td>
<td>Greene</td>
</tr>
<tr>
<td>Hintz</td>
<td></td>
<td>Hintz</td>
</tr>
</tbody>
</table>
| SQL> UPDATE employees SET salary = 7100 WHERE last_name = 'Hintz'; | SQL> UPDATE employees SET salary = 7100 WHERE last_name = 'Hintz'; | Session 1 begins transaction 3 by updating the Hintz salary. The default isolation level for transaction 3 is READ COMMITTED. Session 2 begins transaction 4 and sets it to the 
<p>| SQL&gt; SET TRANSACTION ISOLATION LEVEL SERIALIZABLE; | SQL&gt; SET TRANSACTION ISOLATION LEVEL SERIALIZABLE; | SERIALIZABLE isolation level. Transaction 4 attempts to update the salary for Hintz, but is blocked because transaction 3 locked the Hintz row (see “Row Locks”). Transaction 4 queues behind transaction 3. |
| SQL&gt; COMMIT; | SQL&gt; COMMIT; | Transaction 3 commits its update of the Hintz salary, ending the transaction. |</p>
<table>
<thead>
<tr>
<th>Session 1</th>
<th>Session 2</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UPDATE employees</strong>&lt;br&gt;SET salary = 7200&lt;br&gt;WHERE last_name = 'Hintz' *&lt;br&gt;ERROR at line 1:&lt;br&gt;ORA-08177: can’t serialize access for this transaction</td>
<td></td>
<td>The commit that ends transaction 3 causes the Hintz update in transaction 4 to fail with the ORA-08177 error. The problem error occurs because transaction 3 committed the Hintz update after transaction 4 began.</td>
</tr>
<tr>
<td>SQL&gt; ROLLBACK;</td>
<td></td>
<td>Session 2 rolls back transaction 4, which ends the transaction.</td>
</tr>
<tr>
<td>SQL&gt; SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;</td>
<td></td>
<td>Session 2 begins transaction 5 and sets it to the SERIALIZABLE isolation level.</td>
</tr>
<tr>
<td>SQL&gt; SELECT last_name, salary FROM employees&lt;br&gt;WHERE last_name IN ('Banda', 'Greene', 'Hintz');</td>
<td></td>
<td>Transaction 5 queries the salaries for Banda, Greene, and Hintz. The Hintz salary update committed by transaction 3 is visible.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAST_NAME</td>
<td>SALARY</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>Banda</td>
<td>7000</td>
<td></td>
</tr>
<tr>
<td>Greene</td>
<td>9500</td>
<td></td>
</tr>
<tr>
<td>Hintz</td>
<td>7100</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL&gt; UPDATE employees&lt;br&gt;SET salary = 7200&lt;br&gt;WHERE last_name='Hintz';</td>
<td></td>
<td>Transaction 5 updates the Hintz salary to a different value. Because the Hintz update made by transaction 3 committed before the start of transaction 5, the serialized access problem is avoided. <strong>Note:</strong> If a different transaction updated and committed the Hintz row after transaction 5 began, then the serialized access problem would occur again.</td>
</tr>
<tr>
<td>1 row updated.</td>
<td></td>
<td>Session 2 commits the update without any problems, ending the transaction.</td>
</tr>
</tbody>
</table>
Read-Only Isolation Level

- Similar to the serializable isolation level, but do not permit data to be modified in the transaction unless the user is SYS.
- Not susceptible to the ORA-08177 error.
- Useful for generating reports in which the contents must be consistent with respect to the time when the transaction began.
- Oracle DB achieves read consistency by reconstructing data as needed from the undo segments.
- Undo segments are used in a circular fashion
  - DB can overwrite undo data.
- Long-running reports run the risk that undo data required for read consistency may have been reused by a different transaction, raising a snapshot too old error.
- Solution: Setting an undo retention period, which is the minimum amount of time that the database attempts to retain old undo data before overwriting it, appropriately
Overview of the Oracle Database Locking Mechanism

- Summary of Locking Behavior
- Use of Locks
- Lock Modes
- Lock Conversion and Escalation
- Lock Duration
- Locks and Deadlocks
locking behavior of Oracle Database for readers and writers

• **A row is locked only when modified by a writer.**
  – When a statement updates one row, the transaction acquires a lock for this row only. By locking table data at the row level, DB minimizes contention for the same data.
  – Under normal circumstances the database does not escalate a row lock to the block or table level.

• **A writer of a row blocks a concurrent writer of the same row.**
  – If one transaction is modifying a row, then a row lock prevents a different transaction from modifying the same row simultaneously.
locking behavior of Oracle Database for readers and writers

- **A reader never blocks a writer.**
  - Because a reader of a row does not lock it, a writer can modify this row. The only exception is a `SELECT ... FOR UPDATE` statement, which is a special type of SELECT statement that *does* lock the row that it is reading.

- **A writer never blocks a reader.**
  - When a row is being changed by a writer, the database uses undo data to provide readers with a consistent view of the row.
Overview of the Oracle Database Locking Mechanism

• Summary of Locking Behavior

• **Use of Locks**

• Lock Modes

• Lock Conversion and Escalation

• Lock Duration

• Locks and Deadlocks
Locks in Oracle

• In a single-user database, locks are not necessary.
• Locks are used in the database to permit concurrent access to shared resources, while at the same time providing data integrity and consistency.
• 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.**
Locks in Oracle
– Are automatically obtained at the lowest possible level for a given statement
– Do not escalate
– There are many types of locks used by the Oracle instance to maintain internal consistency. I will focus on locking used to protect rows and tables.

Transaction 1

```
SQL> 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;
```
Locking Mechanism

– High level of data concurrency:
  • Row-level locks for inserts, updates, and deletes
  • No locks required for queries
– Automatic queue management
– Locks held until the transaction ends (with the \texttt{COMMIT} or \texttt{ROLLBACK} operation)

Example
Assume that the rows for \texttt{employee_id} 100 and 101 reside in the same block:

\begin{verbatim}
Transaction 1
\texttt{SQL> UPDATE employees}
\texttt{2  SET salary=salary+100}
\texttt{3  WHERE employee_id=100;}

Transaction 2
\texttt{SQL> UPDATE employees}
\texttt{2  SET salary=salary*1.1}
\texttt{3  WHERE employee_id=101;}
\end{verbatim}
Locks in Oracle

• Oracle locks table data at the row level, but it also uses locks at many other levels to provide concurrent access to various resources.
  • EX: while a stored procedure is executing, the procedure itself is locked in a mode that allows others to execute it, but it will not permit another user to alter that instance of that stored procedure in any way.

• There are as many ways to implement locking in a database as there are many RDBMS vendors.

• Before one or more rows can be changed, user execute DML must obtain lock on row or rows.

• Lock gives use exclusive control until user commits or rollback transaction.

• In Oracle 12C, transaction can lock row, multiple rows or entire table.

• You can lock rows manually but Oracle automatically locks the rows needed at Lowes possible level to ensure data integrity and minimize conflict with other transactions.
Locks in Oracle

- The rule is commit when you must, and not before. Your transactions should only be as small or as large as your business logic dictates.

- You should hold locks on data as long as you need to. Locks may not be scarce, but they can prevent other sessions from modifying information.

- There is no overhead involved with row-level locking in Oracle—none. Whether you have 1 row lock or 1,000,000 row locks, the number of resources dedicated to locking this information will be the same. Number of resources needed to lock 1,000,000 rows is the same as for 1 row; it is a fixed constant.

- Never escalate a lock (use a table lock instead of row locks) it will save no resources.
  - Use a table lock to ensure you can gain access to all of the resources your batch program needs..
  - Ex: batch process, when you know you will update the entire table and you do not want other sessions to lock rows on you

- Concurrency and consistency can be achieved simultaneously.
  - Readers of data are not blocked by writers of data.
  - Writers of data are not blocked by readers of data.
  - Main differences between Oracle and most other RDBMS.
Understanding Locks and transaction

- Both updates to EMPLOYEE table return immediately after update because locks are on different rows.
- Neither session is waiting for the other lock to be released.

<table>
<thead>
<tr>
<th>Session 1</th>
<th>Time</th>
<th>Session 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>update employees set salary =</td>
<td>11:29</td>
<td>update employees set manager =</td>
</tr>
<tr>
<td>salary * 1.2 where employee_id = 102;</td>
<td></td>
<td>100 where employee_id = 109;</td>
</tr>
<tr>
<td>commit;</td>
<td>11:30</td>
<td>commit;</td>
</tr>
</tbody>
</table>
Need for locks example

- Concurrent update of a single row.
- APP uses an UPDATE statement to modify data:

```sql
UPDATE employees
SET email = ?, phone_number = ?
WHERE employee_id = ?
AND email = ?
AND phone_number = ?
```

This update

- Ensures that the row that the application modifies was not changed after the application last read and displayed it to the user.
- Avoids lost update problem in which one user overwrites changes made by another user.
### Row Locking Example

<table>
<thead>
<tr>
<th>T</th>
<th>Session 1</th>
<th>Session 2</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td><code>SELECT employee_id as ID, email, phone_number</code></td>
<td><code>SELECT employee_id as ID, email, phone_number</code></td>
<td>In session 1, the hr1 user queries hr.employees for the Himuro record and displays the employee_id (118), email (GHIMURO), and phone number (515.127.4565) attributes.</td>
</tr>
<tr>
<td></td>
<td><code>FROM hr.employees</code></td>
<td><code>FROM hr.employees</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>WHERE last_name='Himuro'</code></td>
<td><code>WHERE last_name='Himuro'</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ID EMAIL PHONE_NUMBER</td>
<td>ID EMAIL PHONE_NUMBER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>--- -----------</td>
<td>--- -----------</td>
<td></td>
</tr>
<tr>
<td></td>
<td>118 GHIMURO 515.127.4565</td>
<td>118 GHIMURO 515.127.4565</td>
<td></td>
</tr>
<tr>
<td>t1</td>
<td><code>SELECT employee_id as ID, email, phone_number</code></td>
<td><code>SELECT employee_id as ID, email, phone_number</code></td>
<td>In session 2, the hr2 user queries hr.employees for the Himuro record and displays the employee_id (118), email (GHIMURO), and phone number (515.127.4565) attributes.</td>
</tr>
<tr>
<td></td>
<td><code>FROM hr.employees</code></td>
<td><code>FROM hr.employees</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>WHERE last_name='Himuro'</code></td>
<td><code>WHERE last_name='Himuro'</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ID EMAIL PHONE_NUMBER</td>
<td>ID EMAIL PHONE_NUMBER</td>
<td></td>
</tr>
<tr>
<td></td>
<td>--- -----------</td>
<td>--- -----------</td>
<td></td>
</tr>
<tr>
<td></td>
<td>118 GHIMURO 515.127.4565</td>
<td>118 GHIMURO 515.127.4565</td>
<td></td>
</tr>
</tbody>
</table>
### (Cont.) Row Locking Example

<table>
<thead>
<tr>
<th>T</th>
<th>Session 1</th>
<th>Session 2</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t2</td>
<td>UPDATE hr.employees SET phone_number='515.555.1234' WHERE employee_id=118 AND email='GHIMURO' AND phone_number = '515.127.4565';</td>
<td></td>
<td>In session 1, the hr1 user updates the phone number in the row to 515.555.1234, which acquires a lock on the GHIMURO row.</td>
</tr>
<tr>
<td></td>
<td>1 row updated.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t3</td>
<td>UPDATE hr.employees SET phone_number='515.555.1235' WHERE employee_id=118 AND email='GHIMURO' AND phone_number = '515.127.4565';</td>
<td></td>
<td>In session 2, the hr2 user attempts to update the same row, but is blocked because hr1 is currently processing the row. The attempted update by hr2 occurs almost simultaneously with the hr1 update.</td>
</tr>
<tr>
<td></td>
<td>-- SQL*Plus does not show -- a row updated message or -- return the prompt.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(Cont.) Row Locking Example

<table>
<thead>
<tr>
<th>T</th>
<th>Session 1</th>
<th>Session 2</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t4</td>
<td><code>COMMIT;</code></td>
<td></td>
<td>In session 1, the hr1 user commits the transaction.</td>
</tr>
<tr>
<td></td>
<td>Commit complete.</td>
<td></td>
<td>The commit makes the change for Himuro permanent and unblocks session 2, which has been waiting.</td>
</tr>
<tr>
<td>t5</td>
<td></td>
<td>0 rows updated.</td>
<td>In session 2, the hr2 user discovers that the GHIMURO row was modified in such a way that it no longer matches its predicate.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Because the predicates do not match, session 2 updates no records.</td>
</tr>
</tbody>
</table>
(Cont.) Row Locking Example

<table>
<thead>
<tr>
<th>T</th>
<th>Session 1</th>
<th>Session 2</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t6</td>
<td><code>UPDATE hr.employees SET</code>&lt;br&gt;<code>phone_number='515.555.1235'</code>&lt;br&gt;<code>WHERE employee_id=118</code>&lt;br&gt;<code>AND email='GHIMURO' AND</code>&lt;br&gt;<code>phone_number='515.555.1234';</code>&lt;br&gt;<code>1 row updated.</code></td>
<td></td>
<td>In session 1, the hr1 user realizes that it updated the GHIMURO row with the wrong phone number. The user starts a new transaction and updates the phone number in the row to 515.555.1235, which locks the GHIMURO row.</td>
</tr>
<tr>
<td>t7</td>
<td><code>SELECT employee_id as ID,</code>&lt;br&gt;<code>email, phone_number</code>&lt;br&gt;<code>FROM hr.employees</code>&lt;br&gt;<code>WHERE last_name='Himuro';</code>&lt;br&gt;<code>ID EMAIL PHONE_NUMBER</code>&lt;br&gt;<code>--- ------- ---------------</code>&lt;br&gt;<code>118 GHIMURO 515.555.1234</code></td>
<td></td>
<td>In session 2, the hr2 user queries hr.employees for the Himuro record. The record shows the phone number update committed by session 1 at t4. Oracle Database read consistency ensures that session 2 does not see the uncommitted change made at t6.</td>
</tr>
</tbody>
</table>
(Cont.) Row Locking Example

<table>
<thead>
<tr>
<th>T</th>
<th>Session 1</th>
<th>Session 2</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t8</td>
<td>UPDATE hr.employees SET phone_number='515.555.1235'</td>
<td>UPDATE hr.employees SET phone_number='515.555.1235'</td>
<td>In session 2, the hr2 user attempts to update the same row, but is blocked because hr1 is currently processing the row.</td>
</tr>
<tr>
<td></td>
<td>WHERE employee_id=118</td>
<td>WHERE employee_id=118</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AND email='GHIMURO'</td>
<td>AND email='GHIMURO'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AND phone_number = '515.555.1234';</td>
<td>-- SQL*Plus does not show</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>-- a row updated message or</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>-- return the prompt.</td>
<td></td>
</tr>
<tr>
<td>t9</td>
<td>ROLLBACK;</td>
<td></td>
<td>In session 1, the hr1 user rolls back the transaction, which ends it.</td>
</tr>
<tr>
<td></td>
<td>Rollback complete.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### (Cont.) Row Locking Example

<table>
<thead>
<tr>
<th>T</th>
<th>Session 1</th>
<th>Session 2</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t10</td>
<td></td>
<td>1 row updated.</td>
<td>In session 2, the update of the phone number succeeds because the session 1 update was rolled back. The GHIMURO row matches its predicate, so the update succeeds.</td>
</tr>
<tr>
<td>t11</td>
<td></td>
<td>COMMIT; Commit complete.</td>
<td>Session 2 commits the update, ending the transaction.</td>
</tr>
</tbody>
</table>
Overview of the Oracle Database Locking Mechanism

- Summary of Locking Behavior
- Use of Locks
- **Lock Modes**
- Lock Conversion and Escalation
- Lock Duration
- Locks and Deadlocks
Lock Modes

• 2 modes of locking in a multiuser database:
  
  • **Exclusive lock mode**
    
    – Prevents the associated resource from being shared.
    
    – A transaction obtains an exclusive lock when it modifies data.
    
    – The first transaction to lock a resource exclusively is the only transaction that can alter the resource until the exclusive lock is released.
Lock Modes

• **Share lock mode**
  – Allows the associated resource to be shared.
  – Multiple users reading data can share the data, each holding a share lock to prevent concurrent access by a writer who needs an exclusive lock.
  – Multiple transactions can acquire share locks on the same resource.
Lock Modes

- Assume that a transaction uses a SELECT ... FOR UPDATE statement to select a single table row.
  - Transaction acquires an exclusive row lock and a row share table lock.
  - Row lock allows other sessions to modify any rows other than the locked row, while the table lock prevents sessions from altering the structure of the table.
  - Thus, DB permits as many statements as possible to execute.

```sql
select * from employee for update nowait;
select * from employee for update wait 10;
```
Note: the above commands will abort if the lock is not release in the specified time period.

<table>
<thead>
<tr>
<th>RS (table) and RX (row)</th>
<th>select ... for update; lock table ... in row share mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>RX (table) and RX (row)</td>
<td>any insert, update or delete</td>
</tr>
</tbody>
</table>
Overview of the Oracle Database Locking Mechanism

• Summary of Locking Behavior
• Use of Locks
• Lock Modes
• **Lock Conversion and Escalation**
• Lock Duration
• Locks and Deadlocks
Lock Escalation : Microsoft SQL Server Lock Hierarchy

- Always have a Shared Lock (S) on DB level.

- When your query is connected to a DB (USE MyDatabase), Shared Lock prevents the dropping of the DB, or that backups are restored over that database.

- You have locks on the table, on the pages, and the records when you are performing an operation.
SQL Server Lock Escalation

- In DML: Intent Exclusive or Update Lock (IX or IU) on the table and page level, and a Exclusive or Update Lock (X or U) on the changed records.

- SQL Server always acquires locks from top to bottom to prevent Race Conditions, when multiple threads trying to acquire locks concurrently within the locking hierarchy.
SQL Server Lock Escalation

- A **DELETE** operation on a table against **20,000** rows.

- Let’s assume that a row is **400 bytes** long, means that **20** records fit onto one page of **8kb**:

```
1 S lock on the database

1 IX lock on the table

1,000 IX locks on the pages

20,000 X locks on the rows
```
SQL Server Lock Escalation

- one S Lock on the database, 1 IX Lock on the table, 1.000 IX locks on the pages (20,000 records are spread across 1,000 pages), and you have finally 20,000 X locks on the records itself.
- In sum you have acquired **21,002** locks for the **DELETE** operation.
- Every lock needs in SQL Server **96 bytes of memory**, so we look at **1.9 MB of locks just for 1 simple query**.
- This will not scale indefinitely when you run multiple queries in parallel.
- For that reason SQL Server implements now the so-called **Lock Escalation**.
SQL Server Lock Escalation

- For more than 5,000 locks on one level in your locking hierarchy, SQL Server escalates into a simple **coarse-granularity lock**.
- SQL Server will by default *always* escalate directly to the table level.
- An escalation policy to the page level just doesn’t exist.
- One Exclusive Lock (X) on the table level. **Concurrency of your database in a very negative way**
- No other session is able any more to access that table – every other query will just block.

![Diagram](image)
SQL Server Lock Escalation

Since SQL Server 2008 you can also control how SQL Server performs the Lock Escalation – through the `ALTER TABLE` statement and the property `LOCK_ESCALATION`.

3 different options:

- **TABLE**: `Always` performs the Lock Escalation to the table level
- **AUTO**: Lock Escalation is performed to the partition level, if the table is partitioned, and otherwise to the table level.
- **DISABLE**: Disable the Lock Escalation for that specific table. Lock Manager of SQL Server can then consume a huge amount of memory. **Not Recommended** !!!!

```sql
-- Controllling Lock Escalation
ALTER TABLE Person.Person
SET
 (LOCK_ESCALATION = AUTO -- or TABLE or DISABLE
GO
```
Lock Escalation

- System is decreasing the granularity of your locks
- Ex: DB turning your 100 row-level locks against a table into a single table-level lock.

- **Oracle will never escalate a lock. Never.**

- The terms lock conversion and lock promotion are synonymous.
Lock Conversion and Escalation

- **Oracle Database performs lock conversion as necessary.** DB automatically converts a table lock of lower restrictiveness to one of higher restrictiveness.
- Oracle will take a lock at the lowest level possible and convert that lock to a more restrictive level if necessary.

Example:
A transaction issues a SELECT ... FOR UPDATE for an employee and later updates the locked row(s)

1. One lock is placed on the row(s) you selected: **An exclusive lock; no one else can lock that specific row in exclusive mode.**
2. Other lock, a **ROW SHARE TABLE lock**, is placed on the table itself. This will prevent other sessions from placing an exclusive lock on the table and thus prevent them from altering the structure of the table.

- Another session can modify any other row in this table without conflict.
- As many commands as possible that could execute successfully given there is a locked row in the table will be permitted.
Lock Conversion and Escalation

- **Lock conversion** occurs when numerous locks are held at one level of granularity (for example, rows) and a DB raises the locks to a higher level of granularity (for example, table).

- If a user locks many rows in a table, then some databases automatically escalate the row locks to a single table. The number of locks decreases, but the restrictiveness of what is locked increases.

- **When lock escalation occurs, the system is decreasing the granularity of your locks.**

- **Lock escalation is not a database “feature.” It is not a desired attribute.**
  - Overhead in RDBMS locking mechanism and significant work is performed to manage hundreds of locks.

- **In Oracle, the overhead to have 1 lock or 1 million locks is the same: none Oracle will never escalate a lock. Never.**
Overview of the Oracle Database Locking Mechanism

- Summary of Locking Behavior
- Use of Locks
- Lock Modes
- Lock Conversion and Escalation
- **Lock Duration**
- Locks and Deadlocks
Lock Duration

- Oracle DB automatically releases a lock when some event occurs so that the transaction no longer requires the resource.
- Usually, DB holds locks acquired by statements within a transaction for the duration of the transaction.
- These locks prevent destructive interference such as dirty reads, lost updates, and destructive DDL from concurrent transactions.
- **Oracle DB releases all locks acquired by the statements within a transaction when it commits or rolls back.**
- **Oracle DB also releases locks acquired after a savepoint when rolling back to the savepoint.**
- **However, only transactions not waiting or the previously locked resources can acquire locks on the now available resources.**
- **Waiting transactions continue to wait until after the original transaction commits or rolls back completely.**
### Rollback to Savepoint Example

<table>
<thead>
<tr>
<th>T</th>
<th>Session 1</th>
<th>Session 2</th>
<th>Session 3</th>
<th>Explanation</th>
</tr>
</thead>
</table>
| t0 | **UPDATE** employees  
   SET salary = 7000  
   WHERE last_name = 'Banda'; | | | Session 1 begins a transaction. The session places an exclusive lock on the Banda row (TX) and a subexclusive table lock (SX) on the table. |
| t1 | SAVEPOINT after_banda_sal; | | | Session 1 creates a savepoint named after_banda_sal. |
| t2 | **UPDATE** employees  
   SET salary = 12000  
   WHERE last_name = 'Greene'; | | | Session 1 locks the Greene row. |
| t3 | **UPDATE** employees  
   SET salary = 14000  
   WHERE last_name = 'Greene'; | | | Session 2 attempts to update the Greene row, but fails to acquire a lock because session 1 has a lock on this row. No transaction has begun in session 2. |
| t4 | ROLLBACK TO SAVEPOINT after_banda_sal; | | | Session 1 rolls back the update to the salary for Greene, which releases the row lock for Greene. The table lock acquired at t0 is not released. At this point, session 2 is still blocked by session 1 because session 2 enqueues on the session 1 transaction, which has not yet completed. |
(Cont.) Rollback to Savepoint Example

<table>
<thead>
<tr>
<th>T</th>
<th>Session 1</th>
<th>Session 2</th>
<th>Session 3</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>t5</td>
<td></td>
<td></td>
<td>UPDATE employees SET salary = 11000 WHERE last_name = 'Greene';</td>
<td>The Greene row is currently unlocked, so session 3 acquires a lock for an update to the Greene row. This statement begins a transaction in session 3.</td>
</tr>
<tr>
<td>t6</td>
<td>COMMIT;</td>
<td></td>
<td></td>
<td>Session 1 commits, ending its transaction. Session 2 is now enqueued for its update to the Greene row behind the transaction in session 3.</td>
</tr>
</tbody>
</table>
Lock enqueue Mechanism

- Queries never require a lock. A query always succeeds using a pre lock image of data stored in undo tablespace.

- If multiple users require a lock first user obtains the lock. Others users are enqueued using FIFO.

- 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.
Lock enqueue Mechanism

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

• Summary of Locking Behavior
• Use of Locks
• Lock Modes
• Lock Conversion and Escalation
• Lock Duration
• Locks and Deadlocks
Locks and Deadlocks

- A deadlock is a situation in which two or more users are waiting for data locked by each other.
- 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.**
# Locks and Deadlocks

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

<table>
<thead>
<tr>
<th></th>
<th>Session 1</th>
<th>Session 2</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td>SQL&gt; UPDATE employees SET salary = salary*1.1 WHERE employee_id = 100;</td>
<td>SQL&gt; UPDATE employees SET salary = salary*1.1 WHERE employee_id = 200;</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>1 row updated.</td>
<td>1 row updated.</td>
<td></td>
</tr>
<tr>
<td>t1</td>
<td>SQL&gt; UPDATE employees SET salary = salary*1.1 WHERE employee_id = 200;</td>
<td>SQL&gt; UPDATE employees SET salary = salary*1.1 WHERE employee_id = 100;</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>-- prompt does not return</td>
<td>-- prompt does not return</td>
<td></td>
</tr>
</tbody>
</table>
(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>&quot;UPDATE employees *&quot;</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>ERROR at line 1:</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>ORA-00060: deadlock detected</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>while waiting for resource</td>
<td></td>
<td>SQL&gt;</td>
</tr>
</tbody>
</table>
### (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><code>SQL&gt; COMMIT;</code></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><code>1 row updated.</code></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></td>
<td><code>SQL&gt;</code></td>
<td></td>
</tr>
<tr>
<td>t5</td>
<td><code>SQL&gt; COMMIT;</code></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>
Overview of Automatic Locks

• **DML locks**
  – DML: Data Manipulation Language
  – SELECT, INSERT, UPDATE, MERGE, and DELETE statements
  – Allows for concurrent data modifications
  – Protect data.
    • 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.
Overview of Automatic Locks Cont.

- **System Locks**
  - Latches, mutexes, and internal locks are entirely automatic.
  - 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.
Locks Overview

• Oracle places a **TM lock** on it to prevent other sessions from dropping that table (or performing most DDL, in fact).

• **TX locks** that are left on the various blocks we modify so others can tell what data we own.

• Database employs **DDL locks** to protect objects from change while we ourselves are changing them.

• It uses **latches and locks internally** to protect its own structure.
DML Locks

• A DML lock, also called a data lock, guarantees the integrity of data accessed concurrently by multiple users.
  – For example, a DML lock prevents two customers from buying the last copy of a book available from an online bookseller.

• DML locks prevent destructive interference of simultaneous conflicting DML or DDL operations.

• DML statements automatically acquire the following types of locks:

  1. **Row Locks (TX)**
  2. **Table Locks (TM)**
DML Locks

• Each DML transaction must acquire two locks:
  – **EXCLUSIVE** row lock on the row or rows being updated
  – Table lock (TM) in **ROW EXCLUSIVE** (RX) mode on the table containing the rows
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.
Row Locks (TX)

- If a transaction obtains a lock for a row, then the transaction also acquires a lock for the table containing the row.
- The table lock prevents conflicting DDL operations that would override data changes in a current transaction.
- An update of the third row in a table. Oracle Database automatically places an exclusive lock on the updated row and a sub exclusive lock on the table.

**Row and Table Locks**

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
<th>EMAIL</th>
<th>HIRE_DATE</th>
<th>JOB_ID</th>
<th>MANAGER_ID</th>
<th>DEPARTMENT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>King</td>
<td>SKING</td>
<td>17-JUN-87</td>
<td>AD_PRES</td>
<td>100</td>
<td>90</td>
</tr>
<tr>
<td>101</td>
<td>Kochhar</td>
<td>NKOCHHR</td>
<td>21-SEP-89</td>
<td>AD_VP</td>
<td>100</td>
<td>90</td>
</tr>
<tr>
<td>102</td>
<td>De Hann</td>
<td>LDEHANN</td>
<td>13-JAN-93</td>
<td>AD_VP</td>
<td>100</td>
<td>90</td>
</tr>
<tr>
<td>103</td>
<td>Hunold</td>
<td>AHUNOLD</td>
<td>03-JAN-90</td>
<td>IT_PROG</td>
<td>102</td>
<td>60</td>
</tr>
</tbody>
</table>

- Table lock acquired
- Exclusive row lock (TX) acquired
- Row being updated
Row Locks and Concurrency

• This scenario illustrates how Oracle Database uses row locks for concurrency.

• Three sessions query the same rows simultaneously.

• Session 1 and 2 proceed to make uncommitted updates to different rows, while session 3 makes no updates.

• Each session sees its own uncommitted updates but not the uncommitted updates of any other session.
### Data Concurrency Example

<table>
<thead>
<tr>
<th>T</th>
<th>Session 1</th>
<th>Session 2</th>
<th>Session 3</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td>SELECT employee_id, salary, FROM employees, WHERE employee_id IN (100, 101);</td>
<td>SELECT employee_id, salary, FROM employees, WHERE employee_id IN (100, 101);</td>
<td>SELECT employee_id, salary, FROM employees, WHERE employee_id IN (100, 101);</td>
<td>Three different sessions simultaneously query the ID and salary of employees 100 and 101. The results returned by each query are identical.</td>
</tr>
<tr>
<td></td>
<td>EMPLOYEE_ID</td>
<td>SALARY</td>
<td>EMPLOYEE_ID</td>
<td>SALARY</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>512</td>
<td>100</td>
<td>512</td>
</tr>
<tr>
<td></td>
<td>101</td>
<td>600</td>
<td>101</td>
<td>600</td>
</tr>
<tr>
<td>t1</td>
<td>UPDATE hr.employees</td>
<td>SET salary = salary + 100, WHERE employee_id = 100;</td>
<td></td>
<td>Session 1 updates the salary of employee 100, but does not commit. In the update, the writer acquires a row-level lock for the updated row only, thereby preventing other writers from modifying this row.</td>
</tr>
</tbody>
</table>
(Cont.) Data Concurrency Example

<table>
<thead>
<tr>
<th>T</th>
<th>Session 1</th>
<th>Session 2</th>
<th>Session 3</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>t2</td>
<td>SELECT employee_id, salary FROM employees WHERE employee_id IN (100, 101);</td>
<td>SELECT employee_id, salary FROM employees WHERE employee_id IN (100, 101);</td>
<td>SELECT employee_id, salary FROM employees WHERE employee_id IN (100, 101);</td>
<td>Each session simultaneously issues the original query. Session 1 shows the salary of 612 resulting from the t1 update. The readers in session 2 and 3 return rows immediately and do not wait for session 1 to end its transaction. The database uses multiversion read consistency to show the salary as it existed before the update in session 1.</td>
</tr>
<tr>
<td></td>
<td>EMPLOYEE_ID</td>
<td>SALARY</td>
<td>EMPLOYEE_ID</td>
<td>SALARY</td>
</tr>
<tr>
<td>----</td>
<td>-------------</td>
<td>-----------</td>
<td>-------------</td>
<td>--------</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>612</td>
<td>100</td>
<td>512</td>
</tr>
<tr>
<td></td>
<td>101</td>
<td>600</td>
<td>101</td>
<td>600</td>
</tr>
</tbody>
</table>
(Cont.) **Data Concurrency Example**

<table>
<thead>
<tr>
<th>T</th>
<th>Session 1</th>
<th>Session 2</th>
<th>Session 3</th>
<th>Explanation</th>
</tr>
</thead>
</table>
| t3 | UPDATE hr.employee  
| | SET salary =  salary + 100  
| | WHERE  employee_id = 101;  
| |  
| t4 | SELECT employee_id,  
| | salary  
| | FROM employees  
| | WHERE employee_id IN (100, 101);  
| |  
| | EMPLOYEE ID | SALARY  
| | 100 | 612  
| | 101 | 600  
| |  
| | SELECT employee_id,  
| | salary  
| | FROM employees  
| | WHERE employee_id IN (100, 101);  
| |  
| | EMPLOYEE ID | SALARY  
| | 100 | 512  
| | 101 | 700  
| |  
| | SELECT employee_id,  
| | salary  
| | FROM employees  
| | WHERE employee_id IN (100, 101);  
| |  
| | EMPLOYEE ID | SALARY  
| | 100 | 512  
| | 101 | 600  

Session 2 updates the salary of employee 101, but does not commit the transaction. In the update, the writer acquires a row-level lock for the updated row only, preventing other writers from modifying this row.

Each session simultaneously issues the original query. Session 1 shows the salary of 612 resulting from the t1 update, but not the salary update for employee 101 made in session 2. The reader in session 2 shows the salary update made in session 2, but not the salary update made in session 1. The reader in session 3 uses read consistency to show the salaries before modification by session 1 and 2.
Storage of Row Locks : Non Oracle

• In a DB with a traditional memory-based lock manager, the process of locking a row

1. **Find the address of the row you want to lock.**
2. **Get in line at the lock manager (which must be serialized, as it is a common in-memory structure).**
3. **Lock the list.**
4. **Search through the list to see if anyone else has locked this row.**
5. **Create a new entry in the list to establish the fact that you have locked the row.**
6. **Unlock the list.**

• Now that you have the row locked, you can modify it. Later, as you commit your changes, you must continue the procedure as follows:

1. **Get in line again.**
2. **Lock the list of locks.**
3. **Search through the list and release all of your locks.**
4. **Unlock the list.**

As you can see, the more locks acquired, the more time spent on this operation, both before and after modifying the data.
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, not in a big list somewhere (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.
- 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.
Read Consistency and Interested Transaction Lists (ITL)

- The block header of every segment block contains an interested transaction list (ITL).
- The database 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

- The interesting thing is that the 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 that transaction ID 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, it 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.
Locks Type 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. If you already have them in your schema, replace them with the following definitions:
Locks Type Example 1

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.
Locks Type Example 2

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;

<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>
Locks Type Example 4

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>

• 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.
Locks Type Example 5

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;

<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>
</table>
Locks Type Example 6

• Now we’ll start another session using the same username, update some rows in EMP, and then try to update DEPT:

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.
Locks Type Example 7

If we run the V$ queries again, we see the following:

```
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;
```

<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.
Locks Type Example 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>
Locks Type Example 9

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
## Locks Type Example 10

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;

<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>8</td>
<td>17</td>
<td>21403</td>
<td>6</td>
<td>0</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:
Now, if we commit our original transaction, SID=22, and rerun our lock query, we find that the request row has gone:
# Lock Conflicts

<table>
<thead>
<tr>
<th>Transaction 1</th>
<th>Time</th>
<th>Transaction 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPDATE employees SET salary=salary+100 WHERE employee_id=100; 1 row updated.</td>
<td>9:00:00</td>
<td>UPDATE employees SET salary=salary+100 WHERE employee_id=101; 1 row updated.</td>
</tr>
</tbody>
</table>
| UPDATE employees SET COMMISION_PCT=2 WHERE employee_id=101; Session waits enqueued due to lock conflict. | 9:00:05 | SELECT sum(salary) FROM employees; SUM(SALARY) 
692634 |
| Session still waiting! | 16:30:00 | Many selects, inserts, updates, and deletes during the last 7.5 hours, but no commits or rollbacks! |
| 1 row updated. Session continues. | 16:30:01 | commit; |
Possible Causes of Lock Conflicts

– Uncommitted changes
– Long-running transactions
– Unnecessarily high locking levels
Detecting Lock Conflicts

- Select Blocking Sessions on the Performance page.

<table>
<thead>
<tr>
<th>Select</th>
<th>Username</th>
<th>Sessions Blocked</th>
<th>Session ID</th>
<th>Serial Number</th>
<th>SQL ID</th>
<th>Wait Class</th>
<th>Wait Event</th>
<th>P1 Value</th>
<th>P2 Value</th>
<th>P3 Value</th>
<th>Seconds in Wait</th>
</tr>
</thead>
<tbody>
<tr>
<td>☐</td>
<td>Blocking Sessions</td>
<td>1</td>
<td>114</td>
<td>33091</td>
<td>Idle</td>
<td>SQL*Net message from client</td>
<td></td>
<td>1650815232</td>
<td>1</td>
<td>0</td>
<td>89</td>
</tr>
<tr>
<td>☐</td>
<td>BERNST</td>
<td>1</td>
<td>124</td>
<td>46897</td>
<td>0tqktcyyhr56af</td>
<td>Application</td>
<td>enq: TX - row lock contention</td>
<td></td>
<td>1415053318</td>
<td>10954</td>
<td>3085</td>
</tr>
</tbody>
</table>

- Click the Session ID link to view information about the locking session, including the actual SQL statement.
Resolving Lock Conflicts

• To resolve a lock conflict:
  – Have the session holding the lock commit or roll back
  – Terminate the session holding the lock (in an emergency)
Resolving Lock Conflicts with SQL

- SQL statements can be used to determine the blocking session and kill it.

1. SQL> select SID, SERIAL#, USERNAME
from V$SESSION where SID in
(select BLOCKING_SESSION from V$SESSION)

Result:

<table>
<thead>
<tr>
<th>SID</th>
<th>SERIAL#</th>
<th>USERNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>144</td>
<td>8982</td>
<td>HR</td>
</tr>
</tbody>
</table>

2. SQL> alter system kill session '144,8982' immediate;
DML Locks

• DML statements automatically acquire the following types of locks:

1. **Row Locks (TX)**

2. **Table Locks (TM)**

• Oracle places a TM lock on it to prevent other sessions from dropping that table (or performing most DDL, in fact).
• TX locks that are left on the various blocks we modify so others can tell what data we own
• The database employs DDL locks to
• protect objects from change while we ourselves are changing them. It uses latches and locks internally to protect its
• own structure.
Table Locks (TM)

- A table lock (TM lock) is acquired by a transaction when a table is modified by an INSERT, UPDATE, DELETE, MERGE, SELECT with the FOR UPDATE clause, or LOCK TABLE statement.

- DML operations require table locks to prevent DDL operations that would conflict with the transaction.
Table Locks (TM)

• TM locks are used to ensure that the structure of a table is not altered while you are modifying its contents.
• Example:
  • if you have updated a table, you will acquire a TM lock on that table. This will prevent another user from executing DROP or ALTER commands on that table.
  • If another user attempts to perform DDL on the table while you have a TM lock on it, he’ll receive the following error message:

```sql
drop table dept
```

* ERROR at line 1:

ORA-00054: resource busy and acquire with NOWAIT specified

• **Note** In Oracle 11g Release 2 and above, you may set DDL_LOCK_TIMEOUT in order to have DDL wait.
  • Example: ALTER SESSION SET DDL_LOCK_TIMEOUT=60;
Table Locks (TM) Example

EODA@ORA12CR1> create table t1 ( x int );
Table created.
EODA@ORA12CR1> create table t2 ( x int );
Table created.
EODA@ORA12CR1> insert into t1 values ( 1 );
1 row created
EODA@ORA12CR1> insert into t2 values ( 1 );
1 row created.
Table Locks (TM) Example 1

```
EODA@ORA12CR1> select (select username
2 from v$session
3 where sid = v$lock.sid) username,
4 sid,
5 id1,
6 id2,
7 lmode,
8 request, block, v$lock.type
9 from v$lock
10 where sid = sys_context('userenv','sid');

<table>
<thead>
<tr>
<th>USERNAME</th>
<th>SID</th>
<th>ID1</th>
<th>ID2</th>
<th>LMODE</th>
<th>REQUEST</th>
<th>BLOCK</th>
<th>TY</th>
</tr>
</thead>
<tbody>
<tr>
<td>EODA</td>
<td>22</td>
<td>133</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>AE</td>
</tr>
<tr>
<td>EODA</td>
<td>22</td>
<td>244271</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>TM</td>
</tr>
<tr>
<td>EODA</td>
<td>22</td>
<td>244270</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>TM</td>
</tr>
<tr>
<td>EODA</td>
<td>22</td>
<td>1966095</td>
<td>152</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>TX</td>
</tr>
</tbody>
</table>
```

- Whereas we get only one TX lock per transaction, we can get as many TM locks as the objects we modify.
- ID1 column for the TM lock is the object ID of the DML-locked object, so it is easy to find the object on which the lock is being held.
### Table Locks (TM) Example 2

EODA@ORA12CR1> select object_name, object_id
2 from user_objects
3 where object_id in (244271,244270);

<table>
<thead>
<tr>
<th>OBJECT_NAME</th>
<th>OBJECT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2</td>
<td>244271</td>
</tr>
<tr>
<td>T1</td>
<td>244270</td>
</tr>
</tbody>
</table>

- Whereas we get only one TX lock per transaction, we can get as many TM locks as the objects we modify.

- ID1 column for the TM lock is the object ID of the DML-locked object, so it is easy to find the object on which the lock is being held.
Table Locks Modes

Row Share (RS)

• Also called a *subshare table lock (SS)*, indicates that the transaction holding the lock on the table has locked rows in the table and intends to update them.

• The least restrictive mode of table lock, offering the highest degree of concurrency for a table.
Table Locks Modes

Row Exclusive Table Lock (RX)

- Also called a *subexclusive table lock (SX)*
- Generally indicates that the transaction holding the lock has updated table rows or issued SELECT ... FOR UPDATE.
- An SX lock allows other transactions to query, insert, update, delete, or lock rows concurrently in the same table.
- SX locks allow multiple transactions to obtain simultaneous SX and subshare table locks for the same table.
Table Locks Modes

Share Table Lock (S)

• A share table lock held by a transaction allows other transactions to query the table (without using SELECT ... FOR UPDATE)

• Updates are allowed only if a single transaction holds the share table lock.

• Because multiple transactions may hold a share table lock concurrently, holding this lock is not sufficient to ensure that a transaction can modify the table.
Table Locks Modes

**Share Row Exclusive Table Lock (SRX)**

- Also called a *share-subexclusive table lock (SSX)*
- More restrictive than a share table lock.
- Only one transaction at a time can acquire an SSX lock on a given table.
- An SSX lock held by a transaction allows other transactions to query the table (except for `SELECT ... FOR UPDATE`) but not to update the table.
Table Locks Modes

Exclusive Table Lock (X)

- The most restrictive, prohibiting other transactions from performing any type of DML statement or placing any type of lock on the table.
Locks and Foreign Keys

• Oracle DB maximizes the concurrency control of parent keys in relation to dependent foreign keys.
• Locking behavior depends on whether foreign key columns are indexed.
• 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.
• The only exception is when the matching unique or primary key is never updated or deleted.
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 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 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.
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.
DDL Locks

• A data dictionary (DDL) lock protects the definition of a schema object while an ongoing DDL operation acts on or refers to the object.

• Only individual schema objects that are modified or referenced are locked during DDL operations.

• DB never locks the whole data dictionary.

• **Oracle DB acquires a DDL lock automatically on behalf of any DDL transaction requiring it.**

• **Users cannot explicitly request DDL locks.**
  
  – EX : User creates a stored procedure, then Oracle DB automatically acquires DDL locks for all schema objects referenced in the procedure definition.

  – The DDL locks prevent these objects from being altered or dropped before procedure compilation is complete.
Exclusive DDL Locks

• Exclusive DDL locks last for the duration of DDL statement execution and automatic commit.

• During the acquisition of an exclusive DDL lock, if another DDL lock is held on the schema object by another operation, then the acquisition waits until the older DDL lock is released and then proceeds.

• Exclusive DDL locks last for the duration of DDL statement execution and automatic commit.
  – During the acquisition of an exclusive DDL lock, if another DDL lock is held on the schema object by another operation, then the acquisition waits until the older DDL lock is released and then proceeds.
Share DDL Locks

• A share DDL lock for a resource prevents destructive interference with conflicting DDL operations, but allows data concurrency for similar DDL operations.

• Ex: when a CREATE PROCEDURE statement is run, the containing transaction acquires share DDL locks for all referenced tables. Other transactions can concurrently create procedures that reference the same tables and acquire concurrent share DDL locks on the same tables, but no transaction can acquire an exclusive DDL lock on any referenced table.

• A share DDL lock lasts for the duration of DDL statement execution and automatic commit.
  
  – A transaction holding a share DDL lock is guaranteed that the definition of the referenced schema object remains constant during the transaction.
Breakable Parse Locks

• Parse lock is held by a SQL statement or PL/SQL program unit for each schema object that it references.

• Acquired so that the associated shared SQL area can be invalidated if a referenced object is altered or dropped.

• Called a breakable parse lock because it does not disallow any DDL operation and can be broken to allow conflicting DDL operations.

• Acquired in the shared pool during the parse phase of SQL statement execution.

• Lock is held as long as the shared SQL area for that statement remains in the shared pool.
**Internal locks and latches**: System Locks

- **Internal locks and latches**: Latches, Mutexe and Internal Locks

- Oracle DB uses various types of system locks to protect internal database and memory structures.
  - EX: Oracle parses a query and generates an optimized query plan, it will latch library cache to put that plan in there for other sessions to use.

- **Users have no control over their occurrence or duration**

- A latch is a lightweight, low-level serialization device employed by Oracle, similar in function to a lock latches are a common cause of contention in the database, as you will see. They are lightweight in their implementation, but not their effect.
Review Questions

1. Changes made with an UPDATE statement in a transaction are permanent in the database and visible to other users after what occurs?
   A. DBWR flushes the changes to disk.
   B. You issue a SAVEPOINT statement.
   C. You issue a COMMIT statement.
   D. A checkpoint occurs.

2. Which of the following commands returns an error if the transaction starts with SET TRANSACTION READ ONLY?
   A. ALTER SYSTEM
   B. SET ROLE
   C. ALTER USER
   D. None of the above

3. Guaranteed undo retention can be specified for which of the following objects?
   A. A tablespace
   B. A table
   C. The database
   D. A transaction
   E. The instance
Review Questions

4. Which of the following lock modes permits concurrent queries on a table but prohibits updates to the locked table?
A. ROW SHARE
B. ROW EXCLUSIVE
C. SHARE ROW EXCLUSIVE
D. All of the above

5. Select the statement that is not true regarding undo tablespaces.
A. Undo tablespaces will not be created if they are not specified in the CREATE DATABASE command.
B. Two undo tablespaces can be active if a new undo tablespace was specified and the old one contains pending transactions.
C. You can switch from one undo tablespace to another while the database is online.
D. UNDO_MANAGEMENT cannot be changed dynamically while the instance is running.

6. To resolve a lock conflict, which of the following methods can you use? Choose all that apply.
A. Oracle automatically resolves the lock after a short but predefined time period by killing the session that is holding the lock.
B. The DBA can kill the session holding the lock.
C. The user can either roll back or commit the transaction that is holding the lock.
D. Oracle automatically resolves the lock after a short but predefined period by killing the session that is requesting the lock.
Review Questions

7. Two transactions occur at the wall clock times in the following table. What happens at 10:05?

<table>
<thead>
<tr>
<th>Session 1</th>
<th>Session 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>update customer set region = 'H' where state='WI' and county='GRANT';</td>
<td>Time 9:51 update customer set mgr=201 where state='IA' and county='JOHNSON';</td>
</tr>
<tr>
<td>9:59 update customer set region='H' where state='IA' and county='JOHNSON';</td>
<td>10:01 update customer set mgr=201 where state='WI' and county='GRANT';</td>
</tr>
<tr>
<td>10:05 update customer set mgr=201 where state='WI' and county='GRANT';</td>
<td></td>
</tr>
</tbody>
</table>

A. Session 2 will wait for session 1 to commit or roll back.
B. Session 1 will wait for session 2 to commit or roll back.
C. A deadlock will occur, and both sessions will hang unless one of the users cancels their statement or the DBA kills one of the sessions.
D. A deadlock will occur, and Oracle will cancel one of the statements.
E. Neither session is updating the same column, so no waiting or deadlock will occur.
Review Questions

8. If all extents in an undo segment fill up, which of the following occurs next? Choose all that apply.
   A. A new extent is allocated in the undo segment if all existing extents still contain active transaction data.
   B. Other transactions using the segment are moved to another existing segment with enough free space.
   C. A new undo segment is created, and the transaction that filled up the undo segment is moved in its entirety to another undo segment.
   D. The first extent in the segment is reused if the undo data in the first extent is not needed.
   E. The transaction that filled up the undo segment spills over to another undo segment.

9. Which of the following commands returns control to the user immediately if a table is already locked by another user?
   A. LOCK TABLE HR.EMPLOYEES IN EXCLUSIVE MODE WAIT DEFERRED;
   B. LOCK TABLE HR.EMPLOYEES IN SHARE MODE NOWAIT;
   C. LOCK TABLE HR.EMPLOYEES IN SHARE MODE WAIT DISABLED;
   D. LOCK TABLE HR.EMPLOYEES IN EXCLUSIVE MODE NOWAIT DEFERRED;

10. Undo information falls into all the following categories except for which one?
    A. Uncommitted undo information
    B. Undo information required in case an instance crash requires a roll forward operation when the instance is restarted
    C. Committed undo information required to satisfy the undo retention interval
    D. Expired undo information that is no longer needed to support a running transaction
12. The EM Database Express Undo Advisor uses what to recommend the new size of the undo tablespace?

A. The value of the parameter UNDO_RETENTION
B. The number of Snapshot too old errors
C. The current size of the undo tablespace
D. The desired amount of time to retain undo data
E. The most recent undo generation rate

13. Choose the option that is true regarding locks in Oracle Database 12c.

A. When session 1 has a table locked using the LOCK TABLE...EXCLUSIVE MODE statement, all DML statements and queries wait until session 1 does a COMMIT or ROLLBACK.
B. When SELECT...FOR UPDATE is performed, the table is locked.
C. The DDL_LOCK_TIMEOUT parameter can be set to TRUE to not return the ORA-00054 error.
D. The LOCK TABLE statement can include the WAIT clause to specify the number of seconds to wait for acquiring the lock.
Review Questions
14. The following table shows the timestamp and actions by two users. Choose the best option that describes the outcome of the actions.

<table>
<thead>
<tr>
<th>Time</th>
<th>John</th>
<th>Sara</th>
</tr>
</thead>
<tbody>
<tr>
<td>10:14</td>
<td>Select * from hr.employees</td>
<td>Update hr.employees set salary = 100 where employee_id = 206</td>
</tr>
<tr>
<td>10:15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10:16</td>
<td>Commit;</td>
<td>Commit;</td>
</tr>
<tr>
<td></td>
<td>Select * from hr.employees</td>
<td></td>
</tr>
<tr>
<td>10:18</td>
<td>Commit;</td>
<td></td>
</tr>
<tr>
<td>10:20</td>
<td>Select * from hr.employees;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Commit;</td>
<td></td>
</tr>
</tbody>
</table>

A. John’s query will return the same results all three times it is executed as they are run in the same session.
B. John’s queries run at 10:16 and 10:20 produce the same result, which is different from the one at 10:14.
C. John’s query run at 10:16 waits until 10:18 to produce results, waiting for the commit to happen.
D. John’s queries run at 10:14 and 10:16 produce the same result, which is different from the one at 10:20.
Review Questions

15. Which statement is true regarding the locking behavior of Oracle Database 12c?
A. Readers block writers.
B. Writers block readers.
C. Readers block writers.
D. Writers do not block readers.

16. Identify the operation that does not generate redo.
A. An INSERT statement reading from a global temporary table into a persistent table
B. An INSERT statement reading from a persistent table into a global temporary table
C. Roll back an UPDATE operation
D. Writing undo records during a DML operation

17. User Maria just called to let you know that the long-running query she runs every week just received a Snapshot Too Old error. What is the best action you can take?
A. Tell Maria to rerun the query.
B. Increase Undo Retention.
C. The materialized view used in the query is stale and needs to be refreshed.
D. Increase the undo tablespace size.
Review Questions

18. Which two statements regarding undo and transactions are true?
A. Multiple active transactions can write concurrently to the same extent in an undo segment.
B. Multiple active transactions can write concurrently to the same undo segment.
C. Each transaction acquires an extent in the undo segment and does not share the extent.
D. Each transaction acquires a segment in the undo tablespace and does not share the segments.

19. Which statement ends a transaction?
A. UPDATE
B. ALTER TABLE
C. ALTER SESSION
D. ALTER SYSTEM

20. Which statement regarding lock is true?
A. A developer must lock the row before performing an update to prevent others from changing the same row.
B. When a row in a table is locked, the table is locked and no other transactions can update the table.
C. When two sessions try to update the same row at the same time, both sessions fail.
D. When a session tries to update the row already updated by another session, it waits until the other session does a commit or rollback.
21. The lock mechanism defaults to a fine-grained, row-level locking mode.
   1. True
   2. False

22. When the deadlock occurs, Oracle database automatically:
   1. Waits 300 seconds before terminating both sessions
   2. Terminates one statement with an error in one session
   3. Terminates the statements with an error in both sessions
   4. Takes no action by default and leaves it to DBA
Summary

• In this lesson, you should have learned how to:
  – Describe the locking mechanism and how Oracle manages data concurrency
  – Monitor and resolve locking conflicts