Lecture 4 :
Oracle 12c Database Data Concurrency :
Transactions and Locking – Part 2
By David Yitzhak , 4.28.2018
shaked19@gmail.com

12c DATABASE

• http://www.iloug.org.il/DBA_NorthForum.php
• http://www.ildba.co.il/author/cimid/
• http://www.sqlserver.co.il/?cat=940
• 3rd Israeli Conference on Software Architecture
• http://www.iltam.org/sw-arch2016/arch2016_page#OpenSource
• Global Hebrew Virtual PASS Chapter :
• https://www.youtube.com/watch?v=x4hGjYGBfkC
• https://www.youtube.com/watch?v=eJO8G9if3EY
• Sqlsaturday Israel 2016 :
• Sqlsaturday Israel 2017 :
• http://www.sqlsaturday.com/623/Sessions/Detail
• Sqlsaturday Israel 2018
• Course 236510 : Technion - Israel Institute of Technology
Reference and Credits

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

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

- Concurrency and Multiversioning
- Redo and Undo
- Flashback Query and SCN
- ANSI/ISO Transaction Isolation Levels
- Read Committed Isolation
- Serializable Isolation Level
- Read-Only Isolation Level
- DML Locks
- TX (Transaction) Locks
- TM (DML Enqueue) Locks
- Exclusive DDL Locks
- Share DDL locks
- Breakable parse lock
- Latches & Latch “Spinning
- Review questions
Multiversioning

- Related to concurrency control. Ability to simultaneously maintain multiple versions of the data in DB which provides:

  1. *Read-consistent queries*: Queries that produce consistent results with respect to a point in time.
  2. *No blocking queries*: Queries are never blocked by writers of data, as they are in other databases.
Redo and Undo

• Key to Oracle’s recovery mechanism is redo, and core to multiversioning (read consistency) is undo.

• Oracle uses redo to capture how the transaction changed the data to replay the transaction in the event of an instance crash or a media failure.

• Oracle uses undo to store the before image of a modified block. This allows you to reverse or rollback a transaction.

• Undo is conceptually the opposite of redo.

• 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.

• Developers do not need to understand the details. Dba should need to manage and measure this resources consumption.
Undo Segments and Transactions

• When a transaction starts, DB assigns transaction Undo Segments and Transactions 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.
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).
Examples

- Examples in this presentation are based on cricket

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

<table>
<thead>
<tr>
<th>SCORE</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEAM</td>
</tr>
<tr>
<td>RUNS</td>
</tr>
<tr>
<td>WICKETS</td>
</tr>
</tbody>
</table>
# Flashback Query and SCN

## Session 1

<table>
<thead>
<tr>
<th>SQL Statement</th>
<th>Output</th>
</tr>
</thead>
</table>
| ```sql
SELECT runs
FROM score
WHERE team = 'ENG';
``` | Runs 137 |
| ```sql
UPDATE team
SET runs = 141
WHERE team = 'ENG';
COMMIT;
``` |  |

## Session 2

<table>
<thead>
<tr>
<th>SQL Statement</th>
<th>Output</th>
</tr>
</thead>
</table>
| ```sql
SELECT dbms_flashback.get_system_change_number FROM dual;
SCN
3494824
``` |  |
| ```sql
SELECT team, runs, wickets FROM score AS OF SCN 3494824;
WHERE team = 'ENG';
``` | Team Runs Wickets
ENG 141 1 |
| ```sql
SELECT dbms_flashback.get_system_change_number FROM dual;
SCN
3494833
``` |  |
| ```sql
SELECT team, runs, wickets FROM score
WHERE team = 'ENG';
``` | Team Runs Wickets
ENG 137 1 |
Flashback Query

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

```
AS OF [ SCN <scn> | TIMESTAMP <timestamp> ]
```

◆ For example:

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

```
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.**
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.

• **Nonrepeatable (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.

*Preventable Read Phenomena by Isolation Level*

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Dirty Read</th>
<th>Nonrepeatable Read</th>
<th>Phantom Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read uncommitted</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Read committed</td>
<td>Not possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Repeatable read</td>
<td>Not possible</td>
<td>Not possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Serializable</td>
<td>Not possible</td>
<td>Not possible</td>
<td>Not possible</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.
### 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, salary FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz');</td>
<td></td>
<td>Session 1 queries the salaries for Banda, Greene, and Hintz. No employee named Hintz is found.</td>
</tr>
<tr>
<td>LAST_NAME</td>
<td>SALARY</td>
<td></td>
</tr>
<tr>
<td>Banda</td>
<td>6200</td>
<td></td>
</tr>
<tr>
<td>Greene</td>
<td>9500</td>
<td></td>
</tr>
<tr>
<td>SQL&gt; UPDATE employees SET salary = 7000 WHERE last_name = 'Banda';</td>
<td></td>
<td>Session 1 begins a transaction by updating the Banda salary. The default isolation level for transaction 1 is READ COMMITTED.</td>
</tr>
<tr>
<td>SQL&gt; SET TRANSACTION ISOLATION LEVEL READ COMMITTED;</td>
<td></td>
<td>Session 2 begins transaction 2 and sets the isolation level explicitly to READ COMMITTED.</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 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>LAST_NAME</td>
<td>SALARY</td>
<td></td>
</tr>
<tr>
<td>Banda</td>
<td>6200</td>
<td></td>
</tr>
<tr>
<td>Greene</td>
<td>9500</td>
<td></td>
</tr>
<tr>
<td>SQL&gt; UPDATE employees SET salary = 9000 WHERE last_name = 'Greene';</td>
<td></td>
<td>Transaction 2 updates the salary for Greene successfully because transaction 1 locked only the Banda row (see “Row Locks (TX)”).</td>
</tr>
<tr>
<td>SQL&gt; INSERT INTO employees (employee_id, last_name, email, hire_date, job_id) VALUES (210, 'Hintz', 'JHINTZ', SYSDATE, 'SH_CLERK');</td>
<td></td>
<td>Transaction 1 inserts a row for employee Hintz, but does not commit.</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>SQL&gt; SELECT last_name,</td>
<td>Transaction 2 queries the salaries for employees Banda, Greene,</td>
</tr>
<tr>
<td>salary FROM employees</td>
<td>salary FROM employees</td>
<td>and Hintz.</td>
</tr>
<tr>
<td>WHERE last_name IN</td>
<td>WHERE last_name IN</td>
<td>Transaction 2 sees its own update to the salary for Greene.</td>
</tr>
<tr>
<td>('Banda', 'Greene',</td>
<td>('Banda', 'Greene',</td>
<td>Transaction 2 does not see the uncommitted update to the salary for Banda or</td>
</tr>
<tr>
<td>'Hintz')</td>
<td>'Hintz')</td>
<td>the insertion for Hintz made by transaction 1.</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>6200</td>
<td>Transaction 2 attempts to update the row for Banda, which is currently locked by transaction 1, creating a conflicting write. Transaction 2 waits until transaction 1 ends.</td>
</tr>
<tr>
<td>Greene</td>
<td>9900</td>
<td></td>
</tr>
<tr>
<td>Hintz</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQL&gt; COMMIT;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 row updated.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQL&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQL&gt; COMMIT;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SQL&gt; SELECT last_name, salary FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz');</td>
</tr>
<tr>
<td></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. Transaction 2 sees its own update to the Banda salary.</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></td>
</tr>
<tr>
<td>Greene</td>
<td>9900</td>
<td></td>
</tr>
<tr>
<td>Hintz</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>COMMIT;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transaction 2 commits its work, ending the transaction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></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, salary FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz');</td>
<td></td>
<td>Session 1 queries the rows for Banda, Greene, and Hintz. The salary for Banda is 6300, which is the update made by transaction 2. The update of Banda's salary to 7000 made by transaction 1 is now &quot;lost.&quot;</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
  - Rollback entire transaction.
### Serializable 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, salary FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz');</td>
<td></td>
<td>Session 1 queries the salaries for Banda, Greene, and Hintz. No employee named Hintz is found.</td>
</tr>
<tr>
<td><strong>LAST_NAME</strong></td>
<td><strong>SALARY</strong></td>
<td></td>
</tr>
<tr>
<td>Banda</td>
<td>6200</td>
<td></td>
</tr>
<tr>
<td>Greene</td>
<td>9500</td>
<td></td>
</tr>
<tr>
<td>SQL&gt; UPDATE employees SET salary = 7000 WHERE last_name='Banda';</td>
<td></td>
<td>Session 1 begins transaction 1 by updating the Banda salary. The default isolation level is READ COMMITTED.</td>
</tr>
<tr>
<td>SQL&gt; SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;</td>
<td></td>
<td>Session 2 begins transaction 2 and sets it to the SERIALIZABLE isolation level.</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 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><strong>LAST_NAME</strong></td>
<td><strong>SALARY</strong></td>
<td></td>
</tr>
<tr>
<td>Banda</td>
<td>6200</td>
<td></td>
</tr>
<tr>
<td>Greene</td>
<td>9500</td>
<td></td>
</tr>
<tr>
<td>SQL&gt; UPDATE employees SET salary = 9900 WHERE last_name = 'Greene';</td>
<td></td>
<td>Transaction 2 updates the Greene salary successfully because only the Banda row is locked.</td>
</tr>
<tr>
<td>SQL&gt; INSERT INTO employees (employee_id, last_name, email, hire_date, job_id) VALUES (210, 'Hintz', 'JHINTZ', SYSDATE, 'SR_CLERK');</td>
<td></td>
<td>Transaction 1 inserts a row for employee Hintz.</td>
</tr>
<tr>
<td>SQL&gt; COMMIT;</td>
<td></td>
<td>Transaction 1 commits its work, ending the transaction.</td>
</tr>
<tr>
<td>Session 1</td>
<td>Session 2</td>
<td>Explanation</td>
</tr>
<tr>
<td>-----------</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>SQL&gt; SELECT last_name, salary FROM employees WHERE last_name IN ('Banda', 'Greene', 'Hintz');</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>LAST_NAME</td>
<td>SALARY</td>
<td>LAST_NAME</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>
</tbody>
</table>

COMMIT;

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>7000</td>
</tr>
<tr>
<td>Greene</td>
<td>9900</td>
</tr>
<tr>
<td>Hintz</td>
<td></td>
</tr>
</tbody>
</table>

SQL> UPDATE employees SET salary = 7100 WHERE last_name = 'Hintz';

SQL> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

SQL> UPDATE employees SET salary = 7200 WHERE last_name = 'Hintz';

-- prompt does not return

SQL> COMMIT;

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>7000</td>
</tr>
<tr>
<td>Greene</td>
<td>9900</td>
</tr>
<tr>
<td>Hintz</td>
<td>9900</td>
</tr>
</tbody>
</table>

Both sessions query the salaries for Banda, Greene, and Hintz. Each session sees all committed changes made by transaction 1 and transaction 2.

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 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 (TX)”): Transaction 4 queues behind transaction 3.

Transaction 3 commits its update of the Hintz salary, ending the transaction.
(Cont.) Serializable Transaction

<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;<strong>ERROR at line 1:</strong>&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>
</tbody>
</table>

```sql
SQL> ROLLBACK;
```

| **SQL> SET TRANSACTION**<br>**ISOLATION LEVEL SERIALIZABLE;** | | Session 2 rolls back transaction 4, which ends the transaction. |

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

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banda</td>
<td>7000</td>
</tr>
<tr>
<td>Greene</td>
<td>9500</td>
</tr>
<tr>
<td>Hintz</td>
<td>7100</td>
</tr>
</tbody>
</table>

Transaction 5 queries the salaries for Banda, Greene, and Hintz. The Hintz salary update committed by transaction 3 is visible.

```sql
SQL> UPDATE employees<br>    SET salary = 7200<br>WHERE last_name = 'Hintz';
```

1 row updated. 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.

**Note:** If a different transaction updated and committed the Hintz row after transaction 5 began, then the serialized access problem would occur again.

```sql
SQL> COMMIT;
```

Session 2 commits the update without any problems, ending the transaction.
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
Data Concurrency and Consistency

• **Concurrency** = Many users can access data simultaneously
• **Consistency** = Each user has a consistent view of that data
• **Isolation levels and locks** help support data concurrency and consistency

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Dirty Read</th>
<th>Nonrepeatable Read</th>
<th>Phantom Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read uncommitted</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Read committed</td>
<td>Not possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Repeatable read</td>
<td>Not possible</td>
<td>Not possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Serializable</td>
<td>Not possible</td>
<td>Not possible</td>
<td>Not possible</td>
</tr>
</tbody>
</table>
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 = salary * 1.2 where employee_id = 102;</td>
<td>11:29</td>
<td>update employees set manager = 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.
TM (DML Enqueue) Locks

- 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.
- This will prevent another user from executing DROP or ALTER commands on that table.
- TM locks are used to ensure that the structure of a table is not altered while you are modifying its contents.
- If you have updated a table, you will acquire a TM lock 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.
DDL Locks

- DDL locks are automatically placed against objects during a DDL operation to protect them from changes by other sessions.
- Example, when performing DDL operation ALTER TABLE T, table T will have an exclusive DDL lock placed against it, preventing other sessions from getting DDL locks and TM locks on this table.
- DDL locks are held for the duration of the DDL statement and are released immediately afterward.
- DDL always commits in Oracle. Every CREATE, ALTER, and so on statement is executed as shown in this pseudo-code:

```sql
BEGIN
  COMMIT;
  DDL-STATEMENT
  COMMIT;
EXCEPTION
  WHEN OTHERS THEN ROLLBACK;
END;
```

So, DDL will always commit, even if it is unsuccessful. DDL starts by committing; be aware of this. It commits first so that if it has to roll back, it will not roll back your transaction. If you execute DDL, it will make permanent any outstanding work you have performed, even if the DDL is not successful. If you need to execute DDL, but you do not want it to commit your existing transaction, you may use an autonomous transaction.
DDL Locks

1. **Exclusive DDL locks**: prevent other sessions from gaining a DDL lock or TM (DML) lock themselves. This means that you may query a table during a DDL operation, but you may not modify it in any way.

2. **Share DDL locks (dictionary)**: Protect the structure of the referenced object against modification by other sessions, but allow modifications to the data.

3. **Breakable parse locks**: These allow an object, such as a query plan cached in the shared pool, to register its reliance on some other object. If you perform DDL against that object, Oracle will review the list of objects that have registered their dependence and invalidate them. Hence, these locks are breakable—they do not prevent the DDL from occurring.
Exclusive DDL Locks

• Most DDL takes an exclusive DDL lock.

• EX : *Alter table t move*;
  – Table T will be unavailable for modifications during the execution of that statement. The table may be queried using SELECT during this time, but most other operations will be prevented, including all other DDL statements.

• In Oracle, some DDL operations may take place without DDL locks. Ex: *Create index t_idx on t(x) ONLINE*;

• The ONLINE keyword modifies the method by which the index is actually built. Instead of taking an exclusive DDL lock, preventing modifications of data, Oracle will only attempt to acquire a low-level (mode 2) TM lock on the table. This will effectively prevent other DDL from taking place, but it will allow DML to occur normally.

• Oracle accomplishes this feat by keeping a record of modifications made to the table during the DDL statement and applying these changes to the new index as it finishes the CREATE action.
Exclusive DDL Locks

EODA@ORA12CR1> create table t as select * from all_objects;
Table created.
EODA@ORA12CR1> select object_id from user_objects where object_name = 'T';

<table>
<thead>
<tr>
<th>OBJECT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>244277</td>
</tr>
</tbody>
</table>

- Then run the create index against that table:

EODA@ORA12CR1> create index t_idx on t(owner,object_type,object_name) ONLINE;

While at the same time running this query in another session to see the locks taken against that newly created table (ID1=244277 is specific to my example).

> select (select username from v$session where sid = v$lock.sid) username,
sid, id1, id2, lmode, request, block, v$lock.type
from v$lock
where id1 = 244277

<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>244277</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>DL</td>
</tr>
<tr>
<td>EODA</td>
<td>22</td>
<td>244277</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>DL</td>
</tr>
<tr>
<td>EODA</td>
<td>22</td>
<td>244277</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>TM</td>
</tr>
<tr>
<td>EODA</td>
<td>22</td>
<td>244277</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>OD</td>
</tr>
</tbody>
</table>
## Exclusive DDL Locks

```sql
> select (select username from v$session where sid = v$lock.sid) username,
  sid,
  id1,
  id2,
  lmode,
  request, block, v$lock.type
from v$lock
where id1 = 244277
/
```

### USERNAME SID ID1 ID2 LMODE REQUEST BLOCK TY

<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>244277</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>DL</td>
</tr>
<tr>
<td>EODA</td>
<td>22</td>
<td>244277</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>DL</td>
</tr>
<tr>
<td>EODA</td>
<td>22</td>
<td>244277</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>TM</td>
</tr>
<tr>
<td>EODA</td>
<td>22</td>
<td>244277</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>OD</td>
</tr>
</tbody>
</table>

- Two DL locks are **direct load locks**. They are used to prevent a direct path load into our base table while the index creation is taking place.
  - you cannot directly path load the table AND create the index simultaneously.

- OD (Online DDL) lock is a lock type permits truly online DDL which allows modifications of the base table data. Used internally to allow truly online DDL operations. Starting with 11g, the **CREATE INDEX ONLINE** command is completely online; it does not require exclusionary locks at the beginning/end of the command.

---

### Database Writes on SQL*Loader

**Direct Path and Conventional Path**

Insert /*+ parallel(t1) */ into t1

```sql
select * from t2;
```

### Database Writes on SQL*Loader

**Direct Path**

- Write Database
- Generate SQL Commands
- User Processes

**Conventional Path**

- Direct Path
- Conventional Path

### Diagram

- Oracle Server
- SQL*Loader
- User Processes

- Space Management
  - Get new extents
  - Adjust high-water mark
  - Find partial blocks
  - Fill partial blocks

- Buffer Cache Management
  - Manage grants
  - Receive connection

- Read Database Blocks
- Write Database Blocks
Share DDL locks.

• The following DDL statements need a shared DDL lock on the objects that they refer to: audit, noaudit, comment, create [or replace] view/ procedure/ package/package body/function/ trigger, create synonym. Also, create table when the cluster clause isn’t specified.

• Shared DDL lock prevents other users altering or dropping the object, but will not prevent similar DDL statements or any DML

• Ex : If you execute the following, share DDL locks will be placed against both EMP and DEPT (dependent objects) while the CREATE VIEW command is being processed:

  Create view MyView
  as
  select emp.empno, emp.ename, dept.deptno, dept.dname
  from emp, dept
  where emp.deptno = dept.deptno;
Breakable parse lock in action

- An SQL statement or PL/SQL object in the library cache holds a breakable parse lock for each object that it references, until the statement is aged out of the shared pool.
- Breakable parse lock is used to check if the statement should be invalidated if the object changes.
- This lock will never cause waits or contention.

```
EODA@ORA12CR1> create or replace procedure p
2 as
3 begin
4 null;
5 end;
6 /
```

Procedure created.

```
EODA@ORA12CR1> exec p
PL/SQL procedure successfully completed.
```

- Procedure, P, will now show up in the DBA_DDL_LOCKS view. We have a parse lock on it:

```
EODA@ORA12CR1> select session_id sid, owner, name, type,
2 mode_held held, mode_requested request
3 from dba_ddl_locks
4 where session_id = (select sid from v$mystat where rownum=1)
5 /
```
Breakable parse lock in action

- We then recompile our procedure and query the view again:

```
EODA@ORA12CR1> alter procedure p compile;
Procedure altered.
EODA@ORA12CR1> select session_id sid, owner, name, type,
2 mode_held held, mode_requested request
3 from dba_ddl_locks
4 where session_id = (select sid from v$mystat where rownum=1)
5/
```

- We find that P is now missing from the view. Our parse lock has been broken
Latches

• lightweight serialization devices used to coordinate multiuser access to shared data structures, objects, and files.
• Latches are locks designed to be held for extremely short periods of time. Ex: the time it takes to modify an in-memory data structure.
• They are used to protect certain memory structures, such as the database block buffer cache or library cache in the shared pool and Synchronizing Access to the Shared SQL Area.

• If the latch is not available, the requesting session will sleep for a short period of time and retry the operation later.

When a Process Cannot Acquire the Latch

Latch Spinning: Process will loop trying over and over to acquire the latch
Sleep: After a few thousand iterations, the process will sleep for a short duration
Latch Spinning: Upon waking, the process will resume looping trying to acquire the latch
### 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>SELECT employee_id as ID, email, phone_number</td>
<td></td>
<td>In session 1, the hr1 user queries hr.employees for the Himuro record and</td>
</tr>
<tr>
<td></td>
<td>FROM hr.employees</td>
<td></td>
<td>displays the employee_id (118), email (GHIMURO), and phone number (515.127.4565) attributes.</td>
</tr>
<tr>
<td></td>
<td>WHERE last_name='Himuro';</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ID    EMAIL    PHONE_NUMBER</td>
<td></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></td>
<td></td>
</tr>
<tr>
<td>t1</td>
<td></td>
<td>SELECT employee_id as ID, email, phone_number</td>
<td>In session 2, the hr2 user queries hr.employees for the Himuro record and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FROM hr.employees</td>
<td>displays the employee_id (118), email (GHIMURO), and phone number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WHERE last_name='Himuro';</td>
<td>(515.127.4565) attributes.</td>
</tr>
<tr>
<td></td>
<td></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></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>
</table>
| t2 | **UPDATE hr.employees SET**  
    **phone_number='515.555.1234'**  
    **WHERE employee_id=118**  
    **AND email='GHIMURO'**  
    **AND phone_number =**  
    **'515.127.4565';**  
    
    1 row updated. | **UPDATE hr.employees SET**  
    **phone_number='515.555.1234'**  
    **WHERE employee_id=118**  
    **AND email='GHIMURO'**  
    **AND phone_number =**  
    **'515.127.4565';**  
    
    **-- SQL*Plus does not show**  
    **-- a row updated message or**  
    **-- return the prompt.** | 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.  
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. |
(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>COMMIT;</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>0 rows updated.</td>
<td></td>
<td>In session 2, the hr2 user discovers that the GHI MURO 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>UPDATE hr.employees SET phone_number='515.555.1235'</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></td>
<td>WHERE employee_id=118 AND email='GHIMURO' AND phone_number='515.555.1234';</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 row updated.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t7</td>
<td>SELECT employee_id as ID, email, phone_number</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>
<tr>
<td></td>
<td>FROM hr.employees</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WHERE last_name='Himuro';</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ID  EMAIL  PHONE_NUMBER</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>---  ------  ---------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>118  GHIMURO  515.555.1234</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>t8</td>
<td>UPDATE hr.employees SET phone_number='515.555.1235' WHERE employee_id=118 AND email='GHIMURO' AND phone_number = '515.555.1234';</td>
<td>ROLLBACK; Rollback complete.</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>t9</td>
<td>-- SQL*Plus does not show -- a row updated message or -- return the prompt.</td>
<td></td>
<td>In session 1, the hr1 user rolls back the transaction, which ends it.</td>
</tr>
</tbody>
</table>
(Cont.) Row Locking Example

<table>
<thead>
<tr>
<th></th>
<th>Session 1</th>
<th>Session 2</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>t10</td>
<td></td>
<td>1 row updated.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></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></td>
<td>t11</td>
<td></td>
<td>COMMIT;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Session 2 commits the update, ending the transaction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Commit complete.</td>
</tr>
<tr>
<td>T</td>
<td>Session 1</td>
<td>Session 2</td>
<td>Session 3</td>
</tr>
<tr>
<td>---</td>
<td>-----------</td>
<td>-----------</td>
<td>-----------</td>
</tr>
</tbody>
</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>
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>
</table>
| t0 | `SELECT employee_id, salary`  
FROM employees  
WHERE employee_id IN (100, 101); | `SELECT employee_id, salary`  
FROM employees  
WHERE employee_id IN (100, 101); | `SELECT employee_id, salary`  
FROM employees  
WHERE employee_id IN (100, 101); | Three different sessions simultaneously query the ID and salary of employees 100 and 101. The results returned by each query are identical. |
| | ----------- | ----------- | ----------- | ----------- |
| | **EMPLOYEE_ID** | **SALARY** | **EMPLOYEE_ID** | **SALARY** | **EMPLOYEE_ID** | **SALARY** |
| | 100 | 512 | 100 | 512 | 100 | 512 |
| | 101 | 600 | 101 | 600 | 101 | 600 |
| t1 | `UPDATE hr.employees`  
SET salary =  
 salary+100  
WHERE employee_id=100; | | | 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. |
## 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,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>salary,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FROM employees</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WHERE employee_id</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IN ( 100, 101 );</td>
<td></td>
<td></td>
<td></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>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>

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.
### 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>t3</td>
<td></td>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>UPDATE hr.employee</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SET salary =</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>salary + 100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WHERE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>employee_id = 101;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| 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  |
| 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. |
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>Time</th>
<th>Session 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>9:51</td>
<td>update customer set mgr=201</td>
</tr>
<tr>
<td></td>
<td></td>
<td>where state='IA' and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>county='JOHNSON';</td>
</tr>
<tr>
<td></td>
<td>9:59</td>
<td>update customer set region='H'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>where state='WI' and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>county='GRANT';</td>
</tr>
<tr>
<td></td>
<td>10:01</td>
<td>update customer set region='H'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>where state='IA' and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>county='JOHNSON';</td>
</tr>
<tr>
<td></td>
<td>10:05</td>
<td>update customer set mgr=201</td>
</tr>
<tr>
<td></td>
<td></td>
<td>where state='WI' and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>county='GRANT';</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
Review Questions

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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>where employee_id = 206</td>
</tr>
<tr>
<td>10:15</td>
<td>Commit;</td>
<td></td>
</tr>
<tr>
<td>10:16</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>Commit;</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.
Review Questions

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