Database Application Development
Oracle PL/SQL, part 2

CS430/630 Lecture 18b
Murach Chapter 14

How to manage transactions and locking
Objectives

Applied

- Given a set of SQL statements to be combined into a transaction, write a script that begins, commits, and rolls back the transaction.

Knowledge

- Describe the use of transactions.
- Describe the way concurrency control helps prevent concurrency problems.
- Describe the way the transaction isolation level affects concurrency problems and performance.
- Describe a deadlock.
A script that contains a transaction

BEGIN
  INSERT INTO invoices
  VALUES (115, 34, 'ZXA-080', '30-AUG-06',
          14092.59, 0, 0, 3, '30-SEP-06', NULL);

  INSERT INTO invoice_line_items
  VALUES (115, 1, 160, 4447.23, 'HW upgrade');

  INSERT INTO invoice_line_items
  VALUES (115, 2, 167, 9645.36, 'OS upgrade');

  COMMIT;
  DBMS_OUTPUT.PUT_LINE('The transaction was committed.');
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
    DBMS_OUTPUT.PUT_LINE('The transaction was rolled back.');
END;
/

Note: If you have COMMIT in a script, you need EXCEPTION handling and ROLLBACK to be sure to finish the transaction
**When to use transactions**

- When you code two or more INSERT, UPDATE, or DELETE statements that affect related data
- When you move rows from one table to another table by using INSERT and DELETE statements
- Whenever the failure of an INSERT, UPDATE, or DELETE statement would violate data integrity
Auto-commit and lack of it

• Auto-commit means automatic commit after each SQL statement
• Note the last paragraph of Murach pg. 452, which boils down to:
  • Oracle itself (the server) doesn’t have an “auto-commit” mode, so the programmer is expected to use commit and rollback as appropriate in PL/SQL code, or the code that calls it.
  • SQLPlus (the application) has auto-commit (by default), and this means it is calling COMMIT in the server after each statement
  • JDBC has auto-commit (by default), and this means that the JDBC driver is calling COMMIT in the server after each statement
  • So PL/SQL functions/procedures without COMMITs do get committed when they return to an auto-commit environment.
Two transactions that modify the same row

Transaction A

```
UPDATE invoices
    SET credit_total = credit_total + 100
    WHERE invoice_id = 6;
-- the SELECT statement in Transaction B
-- won't show the updated data
-- the UPDATE statement in Transaction B
-- will wait for A to finish
COMMIT;
-- the SELECT statement in Transaction B
-- will show the updated data
-- the UPDATE statement in Transaction B
-- will execute immediately
```

Transaction B

```
SELECT invoice_id, credit_total
    FROM invoices WHERE invoice_id = 6;
UPDATE invoices
    SET credit_total = credit_total + 200
    WHERE invoice_id = 6;
COMMIT;
```

Note these both need exception handling to be complete
The four types of concurrency problems that concurrency control can prevent

- Lost updates
- Dirty reads
- Nonrepeatable reads
- Phantom reads
The concurrency problems prevented by each transaction isolation level

<table>
<thead>
<tr>
<th>Isolation level</th>
<th>Problems prevented</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UNCOMMITTED</td>
<td>None</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>Dirty reads</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>Dirty reads, lost updates</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td>All</td>
</tr>
<tr>
<td>SNAPSHOT</td>
<td>All</td>
</tr>
</tbody>
</table>

- Oracle only provides read committed and serializable (AKA snapshot here)
- In fact, Oracle “serializable” has a few obscure anomalies
The SET TRANSACTION ISOLATION LEVEL statement

```
SET TRANSACTION ISOLATION LEVEL
    {READ COMMITTED|SERIALIZABLE}
```

A statement that sets the level to SERIALIZABLE

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

A statement that sets the level to Oracle’s default

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```
UPDATE statements that illustrate deadlocking

Transaction A

UPDATE savings SET balance = balance - :transfer_amount;
UPDATE checking SET balance = balance + :transfer_amount;
COMMIT;

Transaction B (possible deadlock)

UPDATE checking SET balance = balance - :transfer_amount;
UPDATE savings SET balance = balance + :transfer_amount;
COMMIT;

Transaction B (prevents deadlocks)

UPDATE savings SET balance = balance + :transfer_amount;
UPDATE checking SET balance = balance - :transfer_amount;
COMMIT;

These should have “where account_id = 1234” for example, to update a single account. Also exception handling.
How to prevent deadlocks

- Don’t allow transactions to remain open for long
- Use the lowest possible transaction isolation level, except not READ UNCOMMITTED.
- Make large changes when you can be assured of nearly exclusive access
- Consider locking when coding your transactions
Terms to know

- transaction
- commit
- rollback
- concurrency
- locking
- transaction isolation level
- deadlock