




## Chapter 10

# Insert, Update and Delete

---



## Referencing Another User's Objects (nib)

---

- Use the owner's **schema** name as a prefix to the object name
  - if you've been granted privileges to do so

```
SQL> SELECT *  
      FROM ttrollen.writer
```



# Data Manipulation Language

- Use DML statements to:
  - **INSERT** new rows to a table
  - **UPDATE** existing rows in a table
  - **DELETE** existing rows from a table

3



# Inserting an Individual Row

```
INSERT INTO table [(column [, column...])]  
VALUES (value [, value...]);
```

```
SQL> INSERT INTO writer  
VALUES ('A123', 'Along', 'Bob', Null, Null, Null, Null, Null)
```

- if you provide a value for every field you are not required to specify field names; values are **positionally** matched to fields

```
SQL> INSERT INTO writer(ln, fn, writerid)  
VALUES ('New', 'Writer', 'N999')
```

- you must specify field names if you don't provide a value for each field or if you specify the values in a different order

4



## Inserting Multiple Rows

- can insert rows selected from existing table(s)
  - the field names from the two tables are positionally matched

```
SQL> INSERT INTO student
(STUDENT_ID, LAST_NAME, FIRST_NAME, ZIP,
REGISTRATION_DATE, CREATED_BY, CREATED_DATE,
MODIFIED_BY, MODIFIED_DATE)
SELECT student_id_seq.NEXTVAL, LAST_NAME, FIRST_NAME,
ZIP, SYSDATE, USER, SYSDATE, USER, SYSDATE
FROM instructor
WHERE zip IS NOT NULL
```

```
SQL> ROLLBACK;
```

5



## Transaction Control

- A **transaction** consists of a collection of DML statements that form a logical unit of work
  - used to group a series of changes into a single **batch**
  - help assure **data integrity**
  - classic example: transferring money between accounts (next slide)
  - if each and every operation in the batch succeeds, use the **COMMIT** command write the data to the database
  - if any of the individual operations in the batch fail, use **ROLLBACK** command to undo pending (uncommitted) DML statements
  - create a marker in a current transaction by using the **SAVEPOINT** statement
  - can rollback to that marker using the **ROLLBACK TO SAVEPOINT** statement

6



## Transaction Control: Example

- An ATM user transfers \$500 from her savings account to her checking account.

```
SQL> UPDATE account SET balance = balance - 500
      WHERE account_number = 44533277658;

SQL> UPDATE account SET balance = balance + 500
      WHERE account_number = 44533277650;

SQL> INSERT INTO transfer_log
      VALUES (logseq.nextval, 44533277658, 44533277650,
              500, SYSDATE);

SQL> COMMIT;
```

7



## Transaction

- Begins when the first DML statement is executed
- Ends with one of the following events:
  - **COMMIT** is explicitly issued
  - **ROLLBACK** is explicitly issued
  - DDL statement executes (automatic **COMMIT**)
    - egs: CREATE, ALTER, DROP (Ch11)
  - DCL statement executes (automatic **COMMIT**)
    - egs: GRANT, REVOKE (Ch14)
  - User exits SQL\*Plus normally (automatic **COMMIT**)
    - SQL> exit
  - Abnormal termination of SQL\*Plus (automatic **ROLLBACK**)
    - Close button
    - Task Manager to end task
  - Server system crash (automatic **ROLLBACK**)

8



## State of the Data Before a COMMIT or ROLLBACK

---

- The previous state of the data can be recovered
  - rollback segments have the old values
- The current user can review the results of their own DML operations by using a SELECT statement
- Other users **cannot** view the results of the DML statements by the current user (**read consistency**)
- The affected rows are **locked**
  - other users cannot change the data within the affected rows

9



## State of the Data

---

- After a **COMMIT**
  - Data changes are written to disk
  - Locks on the affected rows are released
  - Other users can now view the results
- After a **ROLLBACK**
  - Pending data changes are undone
  - The previous state of the data is restored from the rollback segments
  - Locks on the affected rows are released

10



## Read Consistency

---

- As User A modifies data during a transaction, he can use SELECT statements to see his own changes
- User B will see the previous state of the data until User A commits his transaction
  - User B does not see the User A's **uncommitted** changes
- Oracle stores the old values in the Undo Tablespace
  - old values are available in case the transaction is rolled back
  - old values are available to present other users a read consistent view of data
  - previous versions of Oracle called them Rollback Segments

11

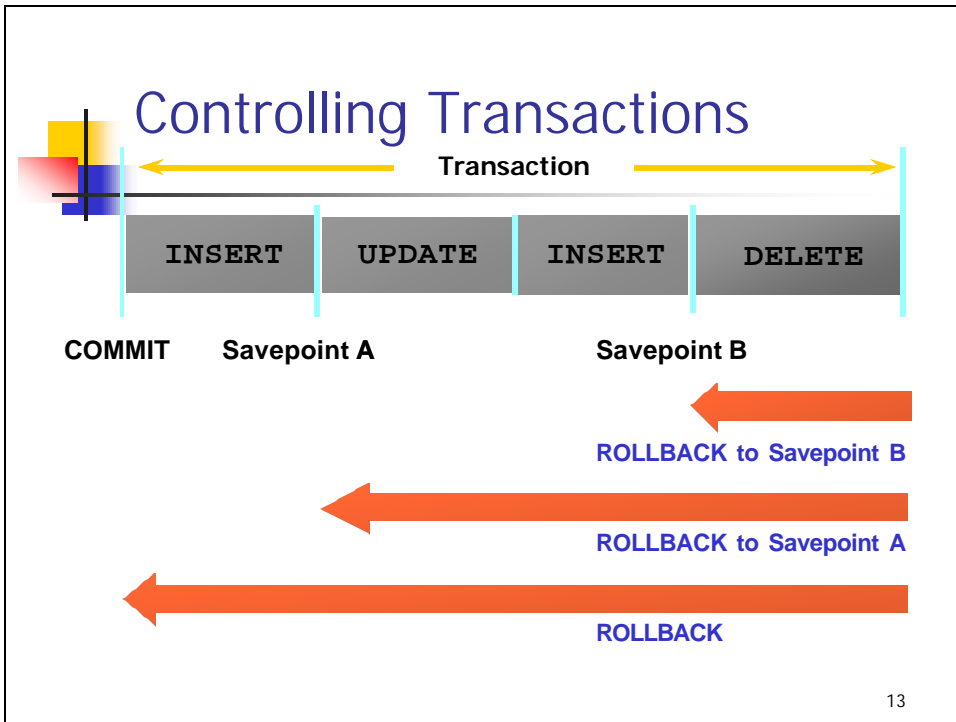


## Statement-Level Rollback (p459+)

---

- A misnomer
- If an individual DML statement fails during execution, that statement (only) is automatically rolled back
- All other (previous) Insert/Update/Delete changes are retained in the pending transaction
- The user should terminate transactions explicitly by executing either a **COMMIT** or **ROLLBACK**

12



- ## Using Special Characters in SQL Statements
- Text literals must be enclosed in single quotes
    - 'Burger King'
  - What if the string data contains a single quote
    - 'Wendy's'
    - 'Wendy''s'
- 14

## Using Special Characters in SQL Statements (p465)

- In SQL\*Plus, **&** indicates that what follows is a **substitution variable** (Ch13)

```
SQL> SELECT title, length
      FROM article
      WHERE title = 'AT&T Antitrust Settlement'
Enter value for t:
```

- SET DEFINE OFF
  - suspends the use of substitution variables
  - from then on, **&** isn't interpreted as a substitution variable

```
SQL> set define off
SQL> SELECT title, length FROM article
      WHERE title = 'AT&T Antitrust Settlement';
TITLE                                     LENGTH
-----
AT&T Antitrust Settlement                 1600
SQL> set define on
```

15

## UPDATING Rows

```
UPDATE table
SET column = value [, column = value, ...]
[WHERE condition];
```

- All rows in the table are modified if you omit the WHERE clause
- Only specific row(s) are modified when you specify a WHERE clause

```
SQL> UPDATE writer
      SET amount = amount + 200, lastcontact=SYSDATE
      WHERE writerid = 'J525';
```

```
SQL> ROLLBACK;
```

16



## UPDATING Rows and CASE

- Can use a searched CASE expression to specify conditional logic that is evaluated before column value is updated

```
SQL> UPDATE writer
      SET amount =
          CASE WHEN contact IS NOT NULL
              THEN amount + 200
              WHEN contact IS NULL
              THEN amount + 500
          END
      WHERE lastcontact > (SYSDATE - 30);
```

17



## DELETING Rows

```
DELETE [FROM] table
[WHERE condition];
```

- All rows in the table are deleted if you omit the WHERE clause!
- If the table is the One side of a One-to-Many relationship and the row(s) to be deleted have matching rows on the Many side
  - if the foreign key constraint uses the ON DELETE CASCADE option the related rows in the Many side of the relationship will automatically be deleted without warning
  - if the foreign key constraint does not use the ON DELETE CASCADE option you won't be able to delete the requested row(s)
    - ORA -02292 error

```
SQL> DELETE FROM writer
      WHERE writerid in ('A123', 'N999')
SQL> DELETE writer WHERE writerid = 'J525';
SQL> ROLLBACK;
```

18



# TRUNCATE Statement

```
TRUNCATE TABLE tablename
```

- Deletes all rows from a table and immediately issues an **automatic** COMMIT
  - can't use ROLLBACK!... be careful!
- What are you left with?
- Does not allow a WHERE clause

```
SQL> TRUNCATE TABLE junk
```

19



# Locks

- Oracle sets and releases **row-level** locks as transactions start/commit/rollback
- Oracle generates **table-level** locks during DDL operations (Chapters 11, 12)
  - egs: ALTER TABLE, CREATE INDEX

20

```
SELECT...FROM...
FOR UPDATE [OF column] [{WAIT n | NOWAIT}]
```

## FOR UPDATE Clause

- Locks the SELECTed rows so no other user can lock or update or delete them until you end your transaction
  - also prevents other users who also use SELECT... FOR UPDATE from seeing the existing values
  - pessimistic locking
- Use **OF ... column** clause to lock the rows only for a particular table. The column in the OF clause is not significant... it only indicates which table to lock.
  - if you omit OF... Oracle locks the selected rows from **each** table
- NOWAIT/WAIT** clause tells Oracle how to proceed if the SELECT attempts to lock a row that is already locked
  - specify **NOWAIT** to return control to you immediately if a lock already exists
  - specify **WAIT** to instruct the database to wait **n** seconds for the row to become available and then return control to you

21

## Lost Update (pg485)

- Occurs when an update made by one user is overwritten by an update of another

|    |  |   |
|----|--|---|
| T1 | <pre>SELECT qty_on_hand FROM inventory WHERE partnum = 4552; QTY_ON_HAND -----           307</pre> |   |
| T2 | <pre>UPDATE inventory SET qty_on_hand = 304 WHERE partnum = 4552;</pre>                            |   |
|    | <pre>SELECT qty_on_hand FROM inventory WHERE partnum = 4552; QTY_ON_HAND -----           307</pre> | T3  |
|    | <pre>COMMIT;</pre>   | <pre>UPDATE inventory SET qty_on_hand = 306 WHERE partnum = 4552;</pre> |
|    |  | T4  |

- What `qty_on_hand` is stored?
- What `qty_on_hand` should be stored?

22

# Lost Update (pg485)

## 3 Solutions

```
SQL> UPDATE inventory
      SET qty_on_hand = 306
      WHERE partnum = 4552 AND qty_on_hand =307; (T4)
No rows updated.
```

```
SQL> UPDATE inventory
      SET qty_on_hand = qty_on_hand - 1
      WHERE partnum = 4552; (T4)
1 row updated.
```

```
SQL> SELECT qty_on_hand
      FROM inventory
      WHERE partnum = 4552 FOR UPDATE NOWAIT; (T1)
```

# Using SELECT... FOR UPDATE to Prevent a Lost Update

