Obtaining Data from Multiple Tables

EMPLOYEES

	EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
1	200	Whalen	10
2	201	Hartstein	20
3	202	Fay	20

. . .

18	174	Abel	80
19	176	Taylor	80
20	178	Grant	(null)

DEPARTMENTS

	DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
1	10	Administration	1700
2	20	Marketing	1800
3	50	Shipping	1500
4	60	IT	1400
5	80	Sales	2500
6	90	Executive	1700
7	110	Accounting	1700
8	190	Contracting	1700

	EMPLOYEE_ID	DEPARTMENT_ID	DEPARTMENT_NAME
1	200	10	Administration
2	201	20	Marketing
3	202	20	Marketing
4	124	50	Shipping

. . .

18	205	110	Accounting
19	206	110	Accounting

Types of Joins

Joins that are compliant with the SQL:1999 standard include the following:

- Natural joins:
    - `NATURAL JOIN` clause
    - `USING` clause
    - `ON` clause
- `OUTER` joins:
    - `LEFT OUTER JOIN`
    - `RIGHT OUTER JOIN`
    - `FULL OUTER JOIN`
- Cross joins

Joining Tables Using SQL:1999 Syntax

Use a join to query data from more than one table:

```
SELECT    table1.column, table2.column
FROM      table1
[NATURAL JOIN table2] |
[JOIN table2 USING (column_name)] |
[JOIN table2
  ON (table1.column_name = table2.column_name)]|
[LEFT|RIGHT|FULL OUTER JOIN table2
  ON (table1.column_name = table2.column_name)]|
[CROSS JOIN table2];
```

Creating Natural Joins

- The `NATURAL JOIN` clause is based on all the columns in the two tables that have the same name.

- It selects rows from the two tables that have equal values in all matched columns.

- If the columns having the same names have different data types, an error is returned.

Retrieving Records with Natural Joins

```
SELECT department_id, department_name,
       location_id, city
FROM   departments
NATURAL JOIN locations ;
```

	DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID	CITY
1	60	IT	1400	Southlake
2	50	Shipping	1500	South San Francisco
3	10	Administration	1700	Seattle
4	90	Executive	1700	Seattle
5	110	Accounting	1700	Seattle
6	190	Contracting	1700	Seattle
7	20	Marketing	1800	Toronto
8	80	Sales	2500	Oxford

Creating Joins with the `USING` Clause

- If several columns have the same names but the data types do not match, use the `USING` clause to specify the columns for the equijoin.

- Use the `USING` clause to match only one column when more than one column matches.

- The `NATURAL JOIN` and `USING` clauses are mutually exclusive.

Joining Column Names

EMPLOYEES

DEPARTMENTS

Foreign key

Primary key

Using Table Aliases with the `USING` Clause

- Do not qualify a column that is used in the `USING` clause.
- If the same column is used elsewhere in the SQL statement, do not alias it.

```
SELECT l.city, d.department_name
FROM   locations l JOIN departments d
USING (location_id)
WHERE d.location_id = 1400;
```

Error encountered

An error was encountered performing the requested operation:

ORA-25154: column part of USING clause cannot have qualifier
25154. 00000 - "column part of USING clause cannot have qualifier"
*Cause: Columns that are used for a named-join (either a NATURAL join
          or a join with a USING clause) cannot have an explicit qualifier.
*Action: Remove the qualifier.
Vendor code 25154Error at Line:4 Column:6

OK

Creating Joins with the ON Clause

- The join condition for the natural join is basically an equijoin of all columns with the same name.

- Use the `ON` clause to specify arbitrary conditions or specify columns to join.

- The join condition is separated from other search conditions.

- The `ON` clause makes code easy to understand.

Retrieving Records with the ON Clause

```
SELECT e.employee_id, e.last_name, e.department_id,
       d.department_id, d.location_id
FROM   employees e JOIN departments d
ON     (e.department_id = d.department_id);
```

	EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID_1	LOCATION_ID
1	200	Whalen	10	10	1700
2	201	Hartstein	20	20	1800
3	202	Fay	20	20	1800
4	144	Vargas	50	50	1500
5	143	Matos	50	50	1500
6	142	Davies	50	50	1500
7	141	Rajs	50	50	1500
8	124	Mourgos	50	50	1500
9	103	Hunold	60	60	1400
10	104	Ernst	60	60	1400
11	107	Lorentz	60	60	1400

...

Creating Three-Way Joins with the ON Clause

```
SELECT  employee_id, city, department_name
FROM    employees e
JOIN    departments d
ON      d.department_id = e.department_id
JOIN    locations l
ON      d.location_id = l.location_id;
```

	EMPLOYEE_ID	CITY	DEPARTMENT_NAME
1	100	Seattle	Executive
2	101	Seattle	Executive
3	102	Seattle	Executive
4	103	Southlake	IT
5	104	Southlake	IT
6	107	Southlake	IT
7	124	South San Francisco	Shipping
8	141	South San Francisco	Shipping
9	142	South San Francisco	Shipping

…

Applying Additional Conditions to a Join

Use the `AND` clause or the `WHERE` clause to apply additional conditions:

```
SELECT  e.employee_id, e.last_name, e.department_id,
        d.department_id, d.location_id
FROM    employees e JOIN departments d
ON      (e.department_id = d.department_id)
AND     e.manager id = 149 ;
```

Or

```
SELECT  e.employee_id, e.last_name, e.department_id,
        d.department_id, d.location_id
FROM    employees e JOIN departments d
ON      (e.department_id = d.department_id)
WHERE   e.manager_id = 149 ;
```

Self-Joins Using the ON Clause

```
SELECT worker.last_name emp, manager.last_name mgr
FROM    employees worker JOIN employees manager
ON      (worker.manager_id = manager.employee_id);
```

	EMP	MGR
1	Hunold	De Haan
2	Fay	Hartstein
3	Gietz	Higgins
4	Lorentz	Hunold
5	Ernst	Hunold
6	Zlotkey	King
7	Mourgos	King

...

Retrieving Records with Nonequijoins

```
SELECT  e.last_name, e.salary, j.grade_level
FROM    employees e JOIN job_grades j
ON      e.salary
        BETWEEN j.lowest_sal AND j.highest_sal;
```

	LAST_NAME	SALARY	GRADE_LEVEL
1	Vargas	2500	A
2	Matos	2600	A
3	Davies	3100	B
4	Rajs	3500	B
5	Lorentz	4200	B
6	Whalen	4400	B
7	Mourgos	5800	B
8	Ernst	6000	C
9	Fay	6000	C
10	Grant	7000	C

...

LEFT OUTER JOIN

```
SELECT e.last_name, e.department_id, d.department_name
FROM    employees e LEFT OUTER JOIN departments d
ON     (e.department_id = d.department_id) ;
```

	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
1	Whalen	10	Administration
2	Fay	20	Marketing
3	Hartstein	20	Marketing
4	Vargas	50	Shipping
5	Matos	50	Shipping

...

16	Kochhar	90	Executive
17	King	90	Executive
18	Gietz	110	Accounting
19	Higgins	110	Accounting
20	Grant	(null)	(null)

RIGHT OUTER JOIN

```
SELECT e.last_name, d.department_id, d.department_name
FROM    employees e RIGHT OUTER JOIN departments d
ON      (e.department_id = d.department_id) ;
```

	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
1	Whalen	10	Administration
2	Hartstein	20	Marketing
3	Fay	20	Marketing
4	Davies	50	Shipping
5	Vargas	50	Shipping
6	Rajs	50	Shipping
7	Mourgos	50	Shipping
8	Matos	50	Shipping

...

18	Higgins	110	Accounting
19	Gietz	110	Accounting
20	(null)	190	Contracting

FULL OUTER JOIN

```
SELECT e.last_name, d.department_id, d.department_name
FROM    employees e FULL OUTER JOIN departments d
ON      (e.department_id = d.department_id) ;
```

	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
1	Whalen	10	Administration
2	Hartstein	20	Marketing
3	Fay	20	Marketing
4	Higgins	110	Accounting

...

17	Zlotkey	80	Sales
18	Abel	80	Sales
19	Taylor	80	Sales
20	Grant	(null)	(null)
21	(null)	190	Contracting

Creating Cross Joins

- The `CROSS JOIN` clause produces the cross-product of two tables.

- This is also called a Cartesian product between the two tables.

```
SELECT last_name, department_name
FROM    employees
CROSS JOIN departments ;
```

	LAST_NAME	DEPARTMENT_NAME
1	Abel	Administration
2	Davies	Administration
3	De Haan	Administration
4	Ernst	Administration
5	Fay	Administration

...

158	Vargas	Contracting
159	Whalen	Contracting
160	Zlotkey	Contracting

Using a Subquery to Solve a Problem

Who has a salary greater than Abel's?

Main query:

Which employees have salaries greater than Abel's salary?

Subquery:
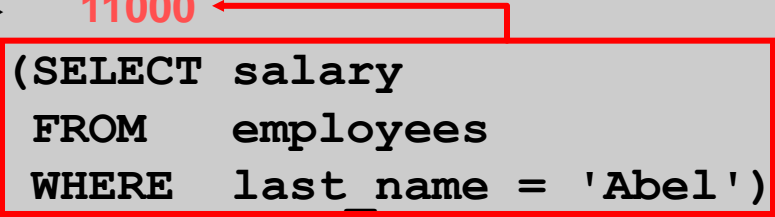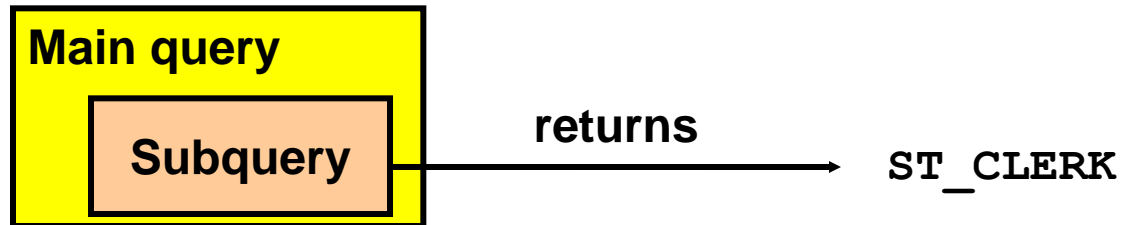
What is Abel's salary?

Using a Subquery

```
SELECT last_name, salary
FROM    employees
WHERE   salary >   11000
                (SELECT salary
                 FROM    employees
                 WHERE   last_name = 'Abel');
```
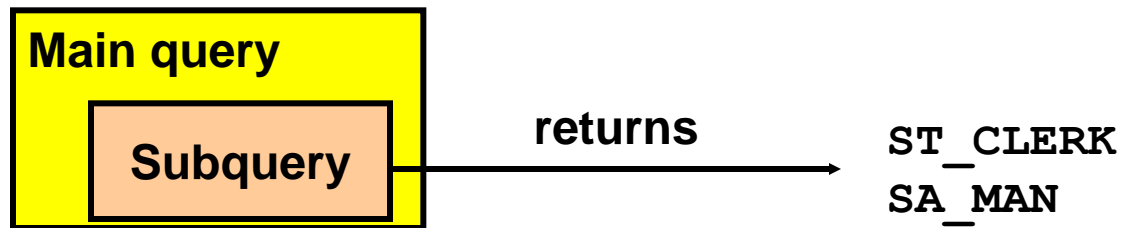
	LAST_NAME		SALARY
1	Hartstein		13000
2	Higgins		12000
3	King		24000
4	Kochhar		17000
5	De Haan		17000

Types of Subqueries

- Single-row subquery

Main query		
	Subquery	

returns → `ST_CLERK`

- Multiple-row subquery

Main query		
	Subquery	

returns → `ST_CLERK`
`SA_MAN`

Single-Row Subqueries

- Return only one row
- Use single-row comparison operators

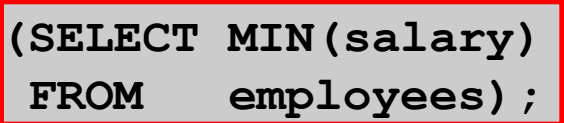| Operator	Meaning
=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<>	Not equal to

Executing Single-Row Subqueries

```
SELECT  last_name, job_id, salary
FROM    employees
WHERE   job_id =                          SA_REP

                (SELECT job_id
                 FROM   employees
                 WHERE  last_name = 'Taylor')
AND     salary >                          8600
                (SELECT salary
                 FROM   employees
                 WHERE  last_name = 'Taylor');
```

LAST_NAME	JOB_ID	SALARY
1 Abel	SA_REP	11000

Using Group Functions in a Subquery

```
SELECT last_name, job_id, salary
FROM    employees
WHERE   salary =                2500
                (SELECT MIN(salary)
                 FROM    employees);
```
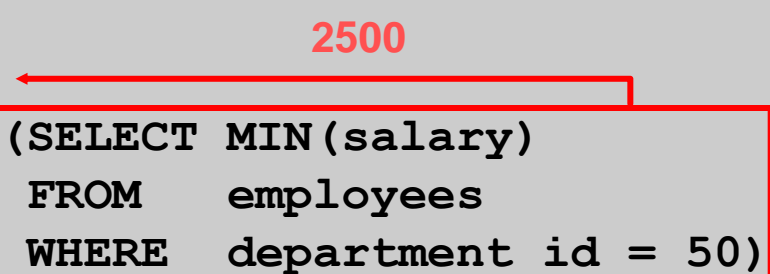
	LAST_NAME	JOB_ID	SALARY
1	Vargas	ST_CLERK	2500

HAVING Clause with Subqueries

- The Oracle server executes the subqueries first.
- The Oracle server returns results into the HAVING clause of the main query.

```
SELECT     department_id, MIN(salary)
FROM       employees
GROUP BY department_id
HAVING     MIN(salary) >                    2500

                        (SELECT MIN(salary)
                         FROM    employees
                         WHERE   department_id = 50);
```

	DEPARTMENT_ID	MIN(SALARY)
1	(null)	7000
2	20	6000
3	90	17000
4	110	8300
5	80	8600
6	10	4400
7	60	4200

No Rows Returned by the Inner Query

```
SELECT last_name, job_id
FROM   employees
WHERE  job_id =
              (SELECT job_id
               FROM   employees
               WHERE  last_name = 'Haas');
```

0 rows selected

Subquery returns no rows because there is no employee named "Haas."

Multiple-Row Subqueries

- Return more than one row
- Use multiple-row comparison operators

Operator	Meaning
IN	Equal to any member in the list
ANY	Must be preceded by =, !=, >, <, <=, >=. Compares a value to each value in a list or returned by a query. Evaluates to FALSE if the query returns no rows.
ALL	Must be preceded by =, !=, >, <, <=, >=. Compares a value to every value in a list or returned by a query. Evaluates to TRUE if the query returns no rows.

Using the ANY Operator in Multiple-Row Subqueries

```
SELECT  employee_id, last_name, job_id, salary
FROM    employees        9000, 6000, 4200
WHERE   salary < ANY
                    (SELECT  salary
                     FROM    employees
                     WHERE   job_id = 'IT_PROG')
AND     job_id <> 'IT_PROG';
```

	EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
1	144	Vargas	ST_CLERK	2500
2	143	Matos	ST_CLERK	2600
3	142	Davies	ST_CLERK	3100
4	141	Rajs	ST_CLERK	3500
5	200	Whalen	AD_ASST	4400

...

	EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
9	206	Gietz	AC_ACCOUNT	8300
10	176	Taylor	SA_REP	8600

Using the ALL Operator
in Multiple-Row Subqueries

```
SELECT  employee_id, last_name, job_id, salary
FROM    employees       9000, 6000, 4200
WHERE   salary < ALL
                        (SELECT salary
                         FROM    employees
                         WHERE   job_id = 'IT_PROG')
AND     job_id <> 'IT_PROG';
```

	EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
1	141	Rajs	ST_CLERK	3500
2	142	Davies	ST_CLERK	3100
3	143	Matos	ST_CLERK	2600
4	144	Vargas	ST_CLERK	2500

Using the `EXISTS` Operator

```
SELECT * FROM departments
WHERE NOT EXISTS
(SELECT * FROM employees
 WHERE employees.department_id=departments.department_id);
```

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	190	Contracting	(null)	1700

Set Operator Guidelines

- The expressions in the `SELECT` lists must match in number.

- The data type of each column in the second query must match the data type of its corresponding column in the first query.

- Parentheses can be used to alter the sequence of execution.

- `ORDER BY` clause can appear only at the very end of the statement.

Oracle Server and Set Operators

- Duplicate rows are automatically eliminated except in `UNION ALL`.

- Column names from the first query appear in the result.

- The output is sorted in ascending order by default except in `UNION ALL`.

Using the `UNION` Operator

Display the current and previous job details of all employees. Display each employee only once.

```
SELECT  employee_id, job_id
FROM    employees
UNION
SELECT  employee_id, job_id
FROM    job_history;
```

	EMPLOYEE_ID	JOB_ID
1	100	AD_PRES
2	101	AC_ACCOUNT

...

	EMPLOYEE_ID	JOB_ID
22	200	AC_ACCOUNT
23	200	AD_ASST

...

	EMPLOYEE_ID	JOB_ID
27	205	AC_MGR
28	206	AC_ACCOUNT

Using the **UNION ALL** Operator

Display the current and previous departments of all employees.

```
SELECT employee_id, job_id, department_id
FROM    employees
UNION ALL
SELECT employee_id, job_id, department_id
FROM    job_history
ORDER BY  employee_id;
```

	EMPLOYEE_ID	JOB_ID	DEPARTMENT_ID
1	100	AD_PRES	90

...

	EMPLOYEE_ID	JOB_ID	DEPARTMENT_ID
17	149	SA_MAN	80
18	174	SA_REP	80
19	176	SA_REP	80
20	176	SA_MAN	80
21	176	SA_REP	80
22	178	SA_REP	(null)
23	200	AD_ASST	10

...

	EMPLOYEE_ID	JOB_ID	DEPARTMENT_ID
30	206	AC_ACCOUNT	110

Using the `INTERSECT` Operator

Display the employee IDs and job IDs of those employees who currently have a job title that is the same as their previous one (that is, they changed jobs but have now gone back to doing the same job they did previously).

```
SELECT  employee_id, job_id
FROM    employees
INTERSECT
SELECT  employee_id, job_id
FROM    job_history;
```

	EMPLOYEE_ID	JOB_ID
1	176	SA_REP
2	200	AD_ASST

Using the MINUS Operator

Display the employee IDs of those employees who have not changed their jobs even once.

```
SELECT  employee_id
FROM    employees
MINUS
SELECT  employee_id
FROM    job_history;
```

	EMPLOYEE_ID
1	100
2	103
3	104

...

13	202
14	205
15	206

Matching the `SELECT` Statements

- Using the `UNION` operator, display the location ID, department name, and the state where it is located.

- You must match the data type (using the `TO_CHAR` function or any other conversion functions) when columns do not exist in one or the other table.

```
SELECT location_id, department_name "Department",
   TO_CHAR(NULL) "Warehouse location"
FROM departments
UNION
SELECT location_id, TO_CHAR(NULL) "Department",
   state_province
FROM locations;
```

Data Manipulation Language

- A DML statement is executed when you:
  - Add new rows to a table
  - Modify existing rows in a table
  - Remove existing rows from a table
- A *transaction* consists of a collection of DML statements that form a logical unit of work.

Inserting New Rows

- Insert a new row containing values for each column.

- List values in the default order of the columns in the table.

- Optionally, list the columns in the `INSERT` clause.

```
INSERT INTO departments(department_id,
        department_name, manager_id, location_id)
VALUES (70, 'Public Relations', 100, 1700);
```
1 rows inserted

- Enclose character and date values within single quotation marks.

Inserting Rows with Null Values

- Implicit method: Omit the column from the column list.

```
INSERT INTO   departments (department_id,
                           department_name)
VALUES        (30, 'Purchasing');
1 rows inserted
```

- Explicit method: Specify the `NULL` keyword in the `VALUES` clause.

```
INSERT INTO   departments
VALUES        (100, 'Finance', NULL, NULL);
1 rows inserted
```

Inserting Specific Date and Time Values

- Add a new employee.

```
INSERT INTO employees
VALUES         (114,
                'Den', 'Raphealy',
                'DRAPHEAL', '515.127.4561',
                TO_DATE('FEB 3, 1999', 'MON DD, YYYY'),
                'SA_REP', 11000, 0.2, 100, 60);
1 rows inserted
```

- Verify your addition.

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT
1	114 Den	Raphealy	DRAPHEAL	515.127.4561	03-FEB-99	SA_REP	11000	0.2

Creating a Script

- Use the & substitution in a SQL statement to prompt for values.
- & is a placeholder for the variable value.

```
INSERT INTO departments
            (department_id, department_name, location_id)
VALUES      (&department_id, '&department_name', &location);
```

Copying Rows
from Another Table

- Write your `INSERT` statement with a subquery:

```
INSERT INTO sales_reps(id, name, salary, commission_pct)
   SELECT employee_id, last_name, salary, commission_pct
   FROM    employees
   WHERE   job_id LIKE '%REP%';
```

4 rows inserted

- Do not use the `VALUES` clause.

- Match the number of columns in the `INSERT` clause to those in the subquery.

- Inserts all the rows returned by the subquery in the table, `sales_reps`.

Updating Rows in a Table

- Values for a specific row or rows are modified if you specify the `WHERE` clause:

```
UPDATE  employees
SET     department_id = 50
WHERE   employee_id = 113;
```
`1 rows updated`

- Values for all the rows in the table are modified if you omit the `WHERE` clause:

```
UPDATE      copy_emp
SET         department_id = 110;
```
`22 rows updated`

- Specify `SET column_name= NULL` to update a column value to `NULL`.

Updating Two Columns with a Subquery

Update employee 113's job and salary to match those of employee 205.

```
UPDATE     employees
SET        job_id  = (SELECT   job_id
                      FROM     employees
                      WHERE    employee_id = 205),
           salary  = (SELECT   salary
                      FROM     employees
                      WHERE    employee_id = 205)
WHERE      employee_id   =   113;
```
1 rows updated

Updating Rows Based on Another Table

Use the subqueries in the UPDATE statements to update row values in a table based on values from another table:

```
UPDATE  copy_emp
SET     department_id  =  (SELECT department_id
                            FROM employees
                           WHERE employee id = 100)
WHERE   job_id         =  (SELECT job_id
                            FROM employees
                           WHERE employee_id = 200);
1 rows updated
```

Deleting Rows from a Table

- Specific rows are deleted if you specify the `WHERE` clause:

```
DELETE FROM departments
WHERE   department_name = 'Finance';
```
1 rows deleted

- All rows in the table are deleted if you omit the `WHERE` clause:

```
DELETE FROM   copy_emp;
```
22 rows deleted

Deleting Rows Based on Another Table

Use the subqueries in the `DELETE` statements to remove rows from a table based on values from another table:

```
DELETE FROM employees
WHERE   department_id =
                  (SELECT department_id
                   FROM    departments
                   WHERE   department_name
                           LIKE '%Public%');
```
`1 rows deleted`

TRUNCATE Statement

- Removes all rows from a table, leaving the table empty and the table structure intact

- Is a data definition language (DDL) statement rather than a DML statement; cannot easily be undone

- Syntax:

```
TRUNCATE TABLE table_name;
```

- Example:

```
TRUNCATE TABLE copy_emp;
```
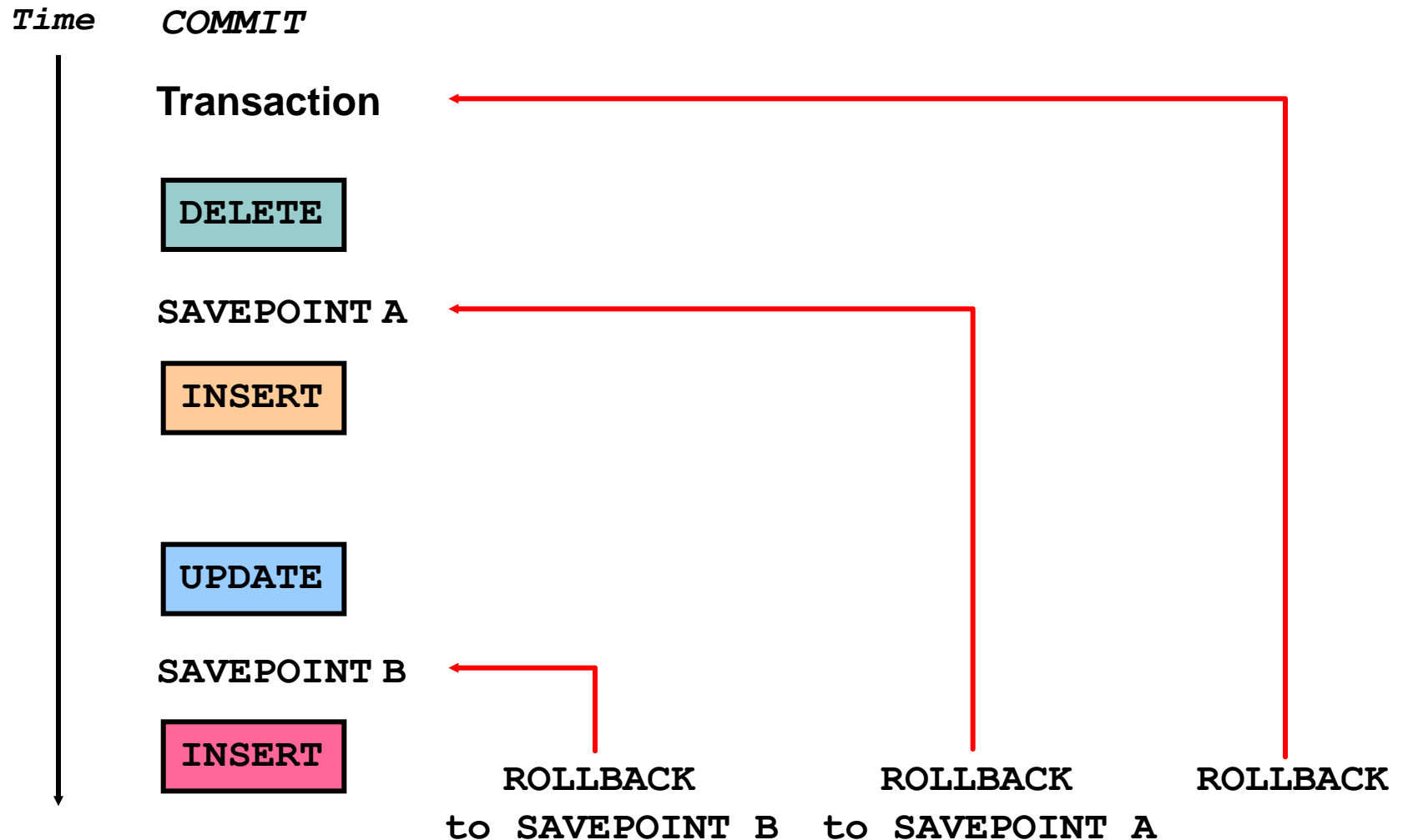
Database Transactions

A database transaction consists of one of the following:

- DML statements that constitute one consistent change to the data

- One DDL statement

- One data control language (DCL) statement

Database Transactions: Start and End

- Begin when the first DML SQL statement is executed.

- End with one of the following events:
  - A `COMMIT` or `ROLLBACK` statement is issued.
  - A DDL or DCL statement executes (automatic commit).
  - The user exits SQL Developer or SQL*Plus.
  - The system crashes.

Explicit Transaction Control Statements

Time   *COMMIT*

Transaction

DELETE

SAVEPOINT A

INSERT

UPDATE

SAVEPOINT B

INSERT

ROLLBACK
to SAVEPOINT B

ROLLBACK
to SAVEPOINT A

ROLLBACK

Rolling Back Changes to a Marker

- Create a marker in the current transaction by using the `SAVEPOINT` statement.
- Roll back to that marker by using the `ROLLBACK TO SAVEPOINT` statement.

```
UPDATE...
SAVEPOINT update_done;
```
SAVEPOINT update_done succeeded.
```
INSERT...
ROLLBACK TO update_done;
```
ROLLBACK TO succeeded.

Committing Data

- Make the changes:

```
DELETE  FROM employees
WHERE   employee_id = 99999;
```
1 rows deleted

```
INSERT  INTO departments
VALUES (290, 'Corporate Tax', NULL, 1700);
```
1 rows inserted

- Commit the changes:

```
COMMIT;
```
COMMIT succeeded.

State of the Data After ROLLBACK

Discard all pending changes by using the ROLLBACK statement:

- Data changes are undone.
- Previous state of the data is restored.
- Locks on the affected rows are released.

```
DELETE FROM copy_emp;
ROLLBACK ;
```

State of the Data After `ROLLBACK`: Example

```
DELETE FROM test;
25,000 rows deleted.


ROLLBACK;
Rollback complete.


DELETE FROM test WHERE  id = 100;
1 row deleted.


SELECT * FROM   test WHERE  id = 100;
No rows selected.


COMMIT;
Commit complete.
```
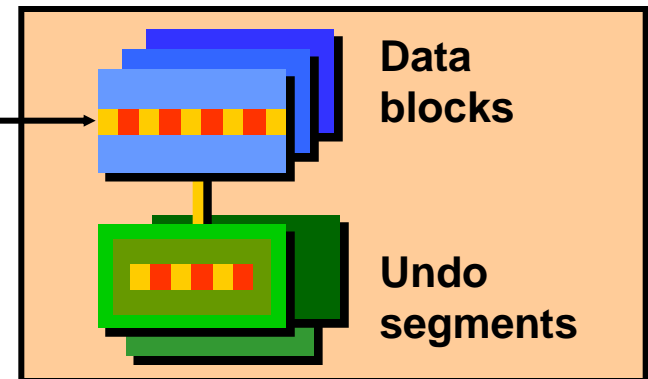
Read Consistency

- Read consistency guarantees a consistent view of the data at all times.

- Changes made by one user do not conflict with the changes made by another user.

- Read consistency ensures that, on the same data:
  – Readers do not wait for writers
  – Writers do not wait for readers
  – Writers wait for writers
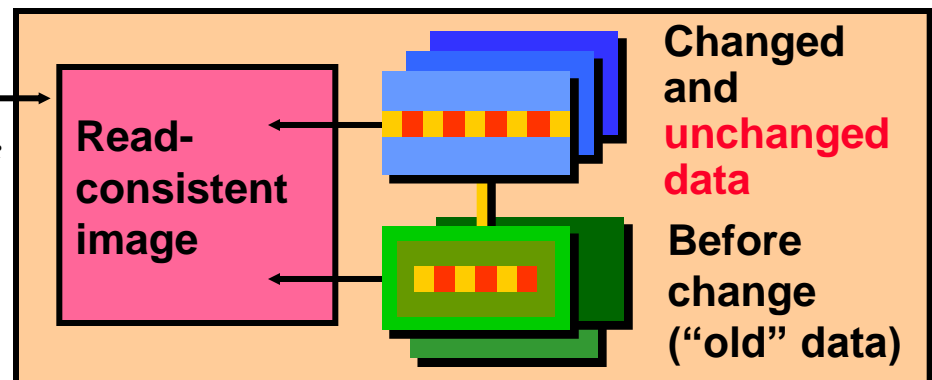
Implementing Read Consistency

User A

```
UPDATE employees
SET    salary = 7000
WHERE  last_name = 'Grant';
```

Data blocks

Undo segments

User B

```
SELECT  *
FROM userA.employees;
```

Read-consistent image

Changed and unchanged data

Before change ("old" data)

FOR UPDATE Clause in a SELECT Statement

- Locks the rows in the EMPLOYEES table where job_id is SA_REP.

```
SELECT employee_id, salary, commission_pct, job_id
FROM employees
WHERE job_id = 'SA_REP'
FOR UPDATE
ORDER BY employee_id;
```

- Lock is released only when you issue a ROLLBACK or a COMMIT.

- If the SELECT statement attempts to lock a row that is locked by another user, the database waits until the row is available, and then returns the results of the SELECT statement.