ORACLE®

甲骨文

ORACLE

甲骨文

Oracle SQL & PL/SQL

Huiyun Mao
Yolanda.mao@oracle.com

SQL Overview

SQL Statements

SELECT	Data retrieval language (DRL)
INSERT UPDATE DELETE	Data manipulation language (DML)
CREATE ALTER DROP RENAME TRUNCATE	Data definition language (DDL)
COMMIT ROLLBACK SAVEPOINT	Transaction control
GRANT REVOKE	Data control language (DCL)

Tables Used in the Course

- Three main tables are used in this course:
  - EMP table
  - DEPT table

The EMP Table

EMP

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	KING	PRESIDENT		17-NOV-81	5000		10
7698	BLAKE	MANAGER	7839	01-MAY-81	2850		30
7782	CLARK	MANAGER	7839	09-JUN-81	1500		10
7566	JONES	MANAGER	7839	02-APR-81	2975		20
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	30
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0	30
7900	JAMES	CLERK	7698	03-DEC-81	950		30
7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	30
7902	FORD	ANALYST	7566	03-DEC-81	3000		20
7369	SMITH	CLERK	7902	17-DEC-80	800		20
7788	SCOTT	ANALYST	7566	09-DEC-82	3000		20
7876	ADAMS	CLERK	7788	12-JAN-83	1100		20
7934	MILLER	CLERK	7782	23-JAN-82	1300		10

Primary key

Foreign key

Foreign key

DEPT Tables

DEPT

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

Primary key

Writing Basic SQL Statements

Capabilities of SQL SELECT Statements

Restriction

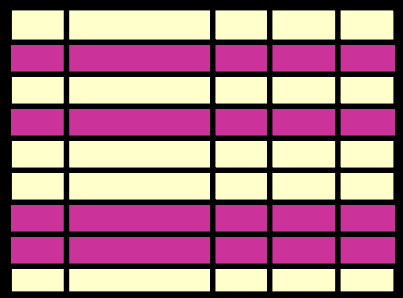


Table 1

Projection

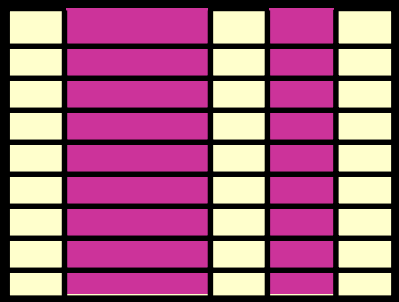


Table 1

Join

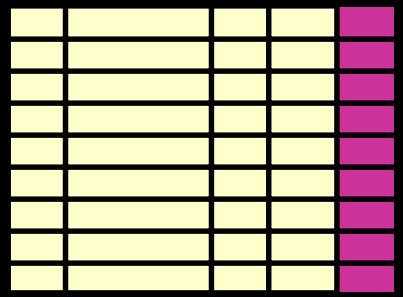


Table 1

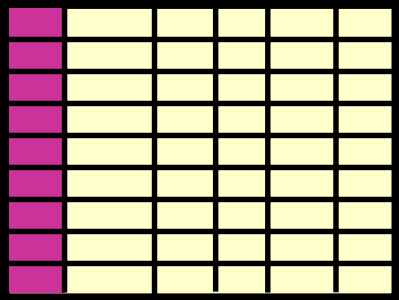


Table 2

Basic SELECT Statement

```
SELECT        [DISTINCT] {*, column [alias],...}
FROM          table
[WHERE        condition(s)]
[GROUP BY     group_by_expression]
[ORDER BY     column];
```

- SELECT identifies the columns to be displayed.
- FROM identifies the table that contains the columns.

Writing SQL Statements

- SQL statements are not case sensitive.
- SQL statements can be on one or more lines.
- Keywords cannot be abbreviated or split across lines.
- Clauses are usually placed on separate lines.
- Tabs and indents are used to enhance readability.

Retrieving All Columns from a Table

DEPT

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

Retrieve all columns from the DEPT table

DEPT

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

All columns are displayed

Selecting All Columns

```
SQL> SELECT *
  2  FROM   dept;
```

```
    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
```

Creating a Projection on a Table

DEPT

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

Retrieve DEPTNO and LOC columns from the DEPT table

DEPT

DEPTNO	LOC
10	NEW YORK
20	DALLAS
30	CHICAGO
40	BOSTON

Only two columns are displayed

Selecting Specific Columns

```
SQL> SELECT deptno, loc
  2  FROM   dept;
```

```
    DEPTNO LOC
---------- -------------
        10 NEW YORK
        20 DALLAS
        30 CHICAGO
        40 BOSTON
```

Default Column Justification

Character
left justified

Date
left justified

Number
right justified

EMP

ENAME	HIREDATE	SAL
KING	17-NOV-81	5000
BLAKE	01-MAY-81	2850
CLARK	09-JUN-81	2450
JONES	02-APR-81	2975
MARTIN	28-SEP-81	1250
ALLEN	20-FEB-81	1600

...
14 rows selected.

Arithmetic Expressions

- Create expressions on NUMBER and DATE data types by using arithmetic operators.

Operator	Description
+	Add
-	Subtract
*	Multiply
/	Divide

Using Arithmetic Operators

```
SQL> SELECT ename, sal, sal+300
  2  FROM   emp;
```

```
ENAME             SAL   SAL+300
---------- ---------- ----------
KING             5000       5300
BLAKE            2850       3150
CLARK            2450       2750
JONES            2975       3275
MARTIN           1250       1550
ALLEN            1600       1900
...
14 rows selected.
```

Using Arithmetic Operators on Multiple Columns

```
SQL> SELECT grade, hisal-losal
  2  FROM   salgrade;
```

```
     GRADE HISAL-LOSAL
---------- -----------
         1         500
         2         199
         3         599
         4         999
         5        6998
```

Operator Precedence

* / + –

- Multiplication and division take priority over addition and subtraction.
- Operators of the same priority are evaluated from left to right.
- Parentheses are used to force prioritized evaluation and to clarify statements.

Operator Precedence

```
SQL> SELECT ename, sal, 12*sal+100
  2  FROM   emp;
```

```
ENAME             SAL 12*SAL+100
---------- --------- -----------
KING             5000      60100
BLAKE            2850      34300
CLARK            2450      29500
JONES            2975      35800
MARTIN           1250      15100
ALLEN            1600      19300
...
14 rows selected.
```

Using Parentheses

```
SQL> SELECT ename, sal, 12*(sal+100)
  2  FROM   emp;
```

```
ENAME             SAL 12*(SAL+100)
---------- --------- ------------
KING             5000        61200
BLAKE            2850        35400
CLARK            2450        30600
JONES            2975        36900
MARTIN           1250        16200
...
14 rows selected.
```

Defining a Column Alias

- Renames a column heading
- Is useful with calculations
- Immediately follows column name; optional AS keyword between column name and alias
- Requires double quotation marks if it is case sensitive or contains spaces or special characters

Using Column Aliases

```
SQL> SELECT ename AS name, sal salary
  2  FROM   emp;
```

```
NAME                 SALARY
---------- ---------
KING                   5000
BLAKE                  2850
CLARK                  2450
JONES                  2975
...
14 rows selected.
```

Using Column Aliases

```
SQL> SELECT ename "Name",
  2         sal*12 "Annual Salary"
  3  FROM   emp;
```

```
Name           Annual Salary
-------------- -------------
KING                   60000
BLAKE                  34200
CLARK                  29400
...
14 rows selected.
```

Concatenation Operator

- Concatenates columns or character strings to other columns
- Is represented by two vertical bars ||
- Creates a result column that is a character expression

Using the Concatenation Operator

```
SQL> SELECT   ename||job AS "Employees"
  2  FROM     emp;
```

```
Employees
--------------------
KINGPRESIDENT
BLAKEMANAGER
CLARKMANAGER
JONESMANAGER
MARTINSALESMAN
ALLENSALESMAN
...
14 rows selected.
```

Literals

- A literal is a constant value of character, expression, or number that can be included in the SELECT list.
- Date and character literal values must be enclosed in single quotation marks.
- Each character string is output once for each row returned.

Using Literal Character Strings

```
SQL> SELECT ename||' is a '||job AS
  2  "Employee Details"
  3  FROM   emp;
```

```
Employee Details
----------------------------
KING is a PRESIDENT
BLAKE is a MANAGER
CLARK is a MANAGER
JONES is a MANAGER
MARTIN is a SALESMAN
...
14 rows selected.
```

Duplicate Rows

- The default display of queries is all rows, including duplicate rows.

```
SQL> SELECT   deptno
  2  FROM     emp;
```

```
    DEPTNO
---------
        10
        30
        10
        20
        ..
14 rows selected.
```

Eliminating Duplicate Rows

- Eliminate duplicate rows by using the DISTINCT keyword in the SELECT clause.

```
SQL> SELECT DISTINCT deptno
  2  FROM   emp;
```

```
    DEPTNO
---------
        10
        20
        30
```

Restricting and Sorting Data

Limiting Rows by Using a Restriction

EMP

EMPNO	ENAME	JOB	...	DEPTNO
7839	KING	PRESIDENT		10
7698	BLAKE	MANAGER		30
7782	CLARK	MANAGER		10
7566	JONES	MANAGER		20
...				

Retrieve all employees in department 10

EMP

EMPNO	ENAME	JOB	...	DEPTNO
7839	KING	PRESIDENT		10
7782	CLARK	MANAGER		10
7934	MILLER	CLERK		10

Using the WHERE Clause

```
SQL> SELECT ename, job, deptno
  2  FROM   emp
  3  WHERE  deptno=10;
```

```
ENAME      JOB         DEPTNO
---------- --------- ---------
KING       PRESIDENT        10
CLARK      MANAGER          10
MILLER     CLERK            10
```

Character Strings and Dates

- Character strings and date values are enclosed in single quotation marks.
- Character values are case sensitive and date values are format sensitive.
- Default date format is DD-MON-YY.

```
SQL> SELECT   ename, job, deptno, hiredate
  2  FROM     emp
  3  WHERE    ename = 'JAMES';
```

Comparison Operators

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

Using the Comparison Operators with Another Column

```
SQL> SELECT ename, sal, comm
  2  FROM   emp
  3  WHERE  sal<=comm;
```

```
ENAME             SAL       COMM
---------- ---------- ----------
MARTIN           1250       1400
```

Using the Comparison Operators with Characters

```
SQL> SELECT ename, mgr
  2  FROM   emp
  3  WHERE  ename='SMITH';
```

```
ENAME                MGR
---------- ---------
SMITH               7902
```

Other SQL Comparison Operators

Operator	Meaning
BETWEEN ...AND...	Between two values (inclusive)
IN(list)	Match any of a list of values
LIKE	Match a character pattern
IS NULL	Is a null value

Using the BETWEEN Operator

- Use the BETWEEN operator to display rows based on a range of values.

```
SQL> SELECT   ename, sal
  2  FROM     emp
  3  WHERE    sal BETWEEN 1000 AND 1500;
```

Lower limit

Higher limit

```
ENAME              SAL
---------- ---------
MARTIN            1250
TURNER            1500
WARD              1250
ADAMS             1100
MILLER            1300
```

Using the IN Operator

- Use the IN operator to test for values in a list.

```
SQL> SELECT   empno, ename, sal, mgr
  2  FROM     emp
  3  WHERE    mgr IN (7902, 7566, 7788);
```

```
     EMPNO ENAME             SAL        MGR
--------- ---------- --------- ----------
      7902 FORD             3000       7566
      7369 SMITH             800       7902
      7788 SCOTT            3000       7566
      7876 ADAMS            1100       7788
```

Using the IN Operator with Strings

- Use the IN operator to test for values in a list of strings.

```
SQL> SELECT ename, deptno, hiredate
  2  FROM   emp
  3  WHERE  ename IN ('BLAKE','MARTIN');
```

```
ENAME          DEPTNO HIREDATE
---------- --------- ---------
BLAKE          30 01-MAY-81
MARTIN         30 28-SEP-81
```

Using the LIKE Operator

- Use the LIKE operator to perform wildcard searches of valid search string values.
- Search conditions can contain either literal characters or numbers.
  - % denotes zero or many characters
  - _ denotes one character

```
SQL> SELECT   ename
  2  FROM     emp
  3  WHERE    ename LIKE 'S%';
```

Using the LIKE Operator

- You can combine pattern matching characters.

```
SQL> SELECT   ename
  2  FROM     emp
  3  WHERE    ename LIKE '_A%';
```

```
ENAME
----------
MARTIN
JAMES
WARD
```

- Use the ESCAPE identifier to search for % or _.

Using the IS NULL Operator

- Test for null values with the IS NULL operator.

```
SQL> SELECT  ename, mgr
  2  FROM    emp
  3  WHERE   mgr IS NULL;
```

```
ENAME      MGR
---------- ---------
KING
```

Logical Operators

Operator	Meaning
AND	Returns TRUE if *both* component conditions are TRUE
OR	Returns TRUE if *either* component condition is TRUE
NOT	Returns TRUE if the following condition is FALSE

Using the AND Operator

- AND requires both conditions to be TRUE.

```
SQL> SELECT empno, ename, job, sal
  2  FROM   emp
  3  WHERE  sal>=1100
  4  AND    job='CLERK';
```

```
     EMPNO ENAME      JOB          SAL
---------- ---------- --------- ---------
      7876 ADAMS      CLERK        1100
      7934 MILLER     CLERK        1300
```

Using the AND Operator

- AND requires both conditions to be TRUE.

```
SQL> SELECT ename, mgr, sal,deptno
  2  FROM   emp
  3  WHERE  sal>1000
  4  AND    deptno = 10;
```

ENAME	MGR	SAL	DEPTNO
KING		5000	10
CLARK	7839	2450	10
MILLER	7782	1300	10

Using the OR Operator

OR requires either condition to be TRUE.

```
SQL> SELECT empno, ename, job, sal
  2  FROM   emp
  3  WHERE  sal>=2000
  4  OR     job='CLERK';
```

```
     EMPNO ENAME      JOB         SAL
--------- ---------- --------- ----------
      7839 KING       PRESIDENT     5000
      7698 BLAKE      MANAGER       2850
      7782 CLARK      MANAGER       2450
      7566 JONES      MANAGER       2975
      7900 JAMES      CLERK          950
      7902 FORD       ANALYST       3000
      ...
10 rows selected.
```

Using the OR Operator

OR requires either condition to be TRUE.

```
SQL> SELECT ename, deptno, mgr
  2  FROM   emp
  3  WHERE  deptno = 10
  4  OR     mgr = 7839;
```

ENAME	DEPTNO	MGR
KING	10	
BLAKE	30	7839
CLARK	10	7839
JONES	20	7839
MILLER	10	7782

Using the NOT Operator

```
SQL> SELECT ename, job
  2  FROM   emp
  3  WHERE  job NOT IN ('CLERK','MANAGER','ANALYST');
```

```
ENAME      JOB
---------- ---------
KING       PRESIDENT
MARTIN     SALESMAN
ALLEN      SALESMAN
TURNER     SALESMAN
WARD       SALESMAN
```

Using the NOT Operator

```
SQL> SELECT empno,ename,deptno,mgr
  2  FROM   emp
  3  WHERE mgr NOT LIKE '78%';
```

```
 EMPNO    ENAME          DEPTNO        MGR
--------- ---------- --------- ---------
     7654 MARTIN             30      7698
     7499 ALLEN              30      7698
     ...
     ...
     7902 FORD               20      7566
     7369 SMITH              20      7902
     ...
10 rows selected.
```

Using the NOT Operator

```
SQL> SELECT empno, sal, mgr
  2  FROM   emp
  3  WHERE  sal NOT BETWEEN 1000 AND 1500;
```

EMPNO	SAL	MGR
7839	5000	
7698	2850	7839
7782	2450	7839
7566	2975	7839
7499	1600	7698
7900	950	7698
7902	3000	7566
7369	800	7902
7788	3000	7566

9 rows selected.

Using the NOT Operator

```
SQL> SELECT ename, sal AS "Salary Before Commission",
  2  comm
  3  FROM   emp
  4  WHERE  comm IS NOT NULL;
```

```
ENAME      Salary Before Commission       COMM
---------- ------------------------- ----------
MARTIN                        1250       1400
ALLEN                         1600        300
TURNER                        1500          0
WARD                          1250        500
```

Rules of Precedence

Order Evaluated	Operator
1	All comparison operators
2	NOT
3	AND
4	OR

- Use parentheses to override rules of precedence.

Rules of Precedence

```
SQL> SELECT ename, job, sal
  2  FROM   emp
  3  WHERE  job='SALESMAN'
  4  OR     job='PRESIDENT'
  5  AND    sal>1500;
```

ENAME	JOB	SAL
KING	PRESIDENT	5000
MARTIN	SALESMAN	1250
ALLEN	SALESMAN	1600
TURNER	SALESMAN	1500
WARD	SALESMAN	1250

Rules of Precedence

Use parentheses to force priority.

```
SQL> SELECT   ename, job, sal
  2  FROM     emp
  3  WHERE    (job='SALESMAN'
  4  OR       job='PRESIDENT')
  5  AND      sal>1500;
```

ENAME	JOB	SAL
KING	PRESIDENT	5000
ALLEN	SALESMAN	1600

ORDER BY Clause

- Sort rows with the ORDER BY clause:
  - ASC: ascending order, default
  - DESC: descending order
- The ORDER BY clause comes last in the SELECT statement.

```
SQL> SELECT ename, job, deptno
  2  FROM   emp
  3  ORDER BY deptno;
```

```
ENAME      JOB           DEPTNO
---------- --------- ---------
KING       PRESIDENT        10
CLARK      MANAGER          10
...
JONES      MANAGER          20
SCOTT      ANALYST          20
...
14 rows selected.
```

Sorting in Descending Order

```
SQL> SELECT   ename, job, deptno, sal
  2  FROM     emp
  3  ORDER BY sal DESC;
```

```
ENAME      JOB           DEPTNO        SAL
---------- --------- ---------- ----------
KING       PRESIDENT         10       5000
FORD       ANALYST           20       3000
SCOTT      ANALYST           20       3000
JONES      MANAGER           20       2975
BLAKE      MANAGER           30       2850
CLARK      MANAGER           10       2450
ALLEN      SALESMAN          30       1600
...
14 rows selected.
```

Sorting by Column Alias

```
SQL> SELECT   empno, ename, sal*12 annsal
  2  FROM     emp
  3  ORDER BY annsal;
```

```
     EMPNO ENAME          ANNSAL
--------- ---------- ---------
      7369 SMITH          9600
      7900 JAMES         11400
      7876 ADAMS         13200
      7654 MARTIN        15000
      7521 WARD          15000
      7934 MILLER        15600
      7844 TURNER        18000
      ...
14 rows selected.
```

Sorting by Multiple Columns

- The order of an ORDER BY list is the order of the sort.

```
SQL> SELECT    ename, deptno, sal
  2  FROM      emp
  3  ORDER BY  deptno, sal DESC;
```

```
ENAME          DEPTNO        SAL
---------- --------- ---------
KING           10       5000
CLARK          10       2450
MILLER         10       1300
FORD           20       3000
...
14 rows selected.
```

Sorting by a Column Not in the SELECT List

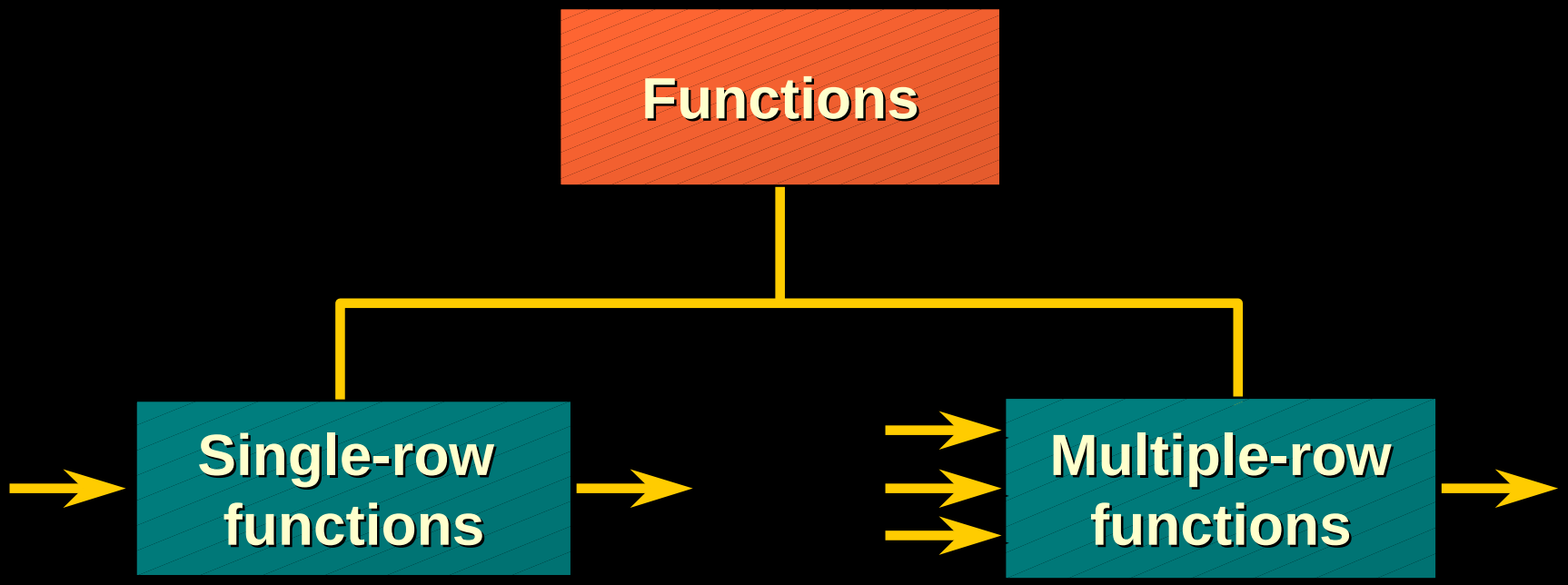
```
SQL> SELECT   ename, deptno
  2  FROM     emp
  3  ORDER BY sal;
```

```
ENAME          DEPTNO
---------- ---------
SMITH              20
JAMES              30
ADAMS              20
MARTIN             30
WARD               30
MILLER             10
...
14 rows selected.
```

Single-Row Number and Character Functions

How a Function Works

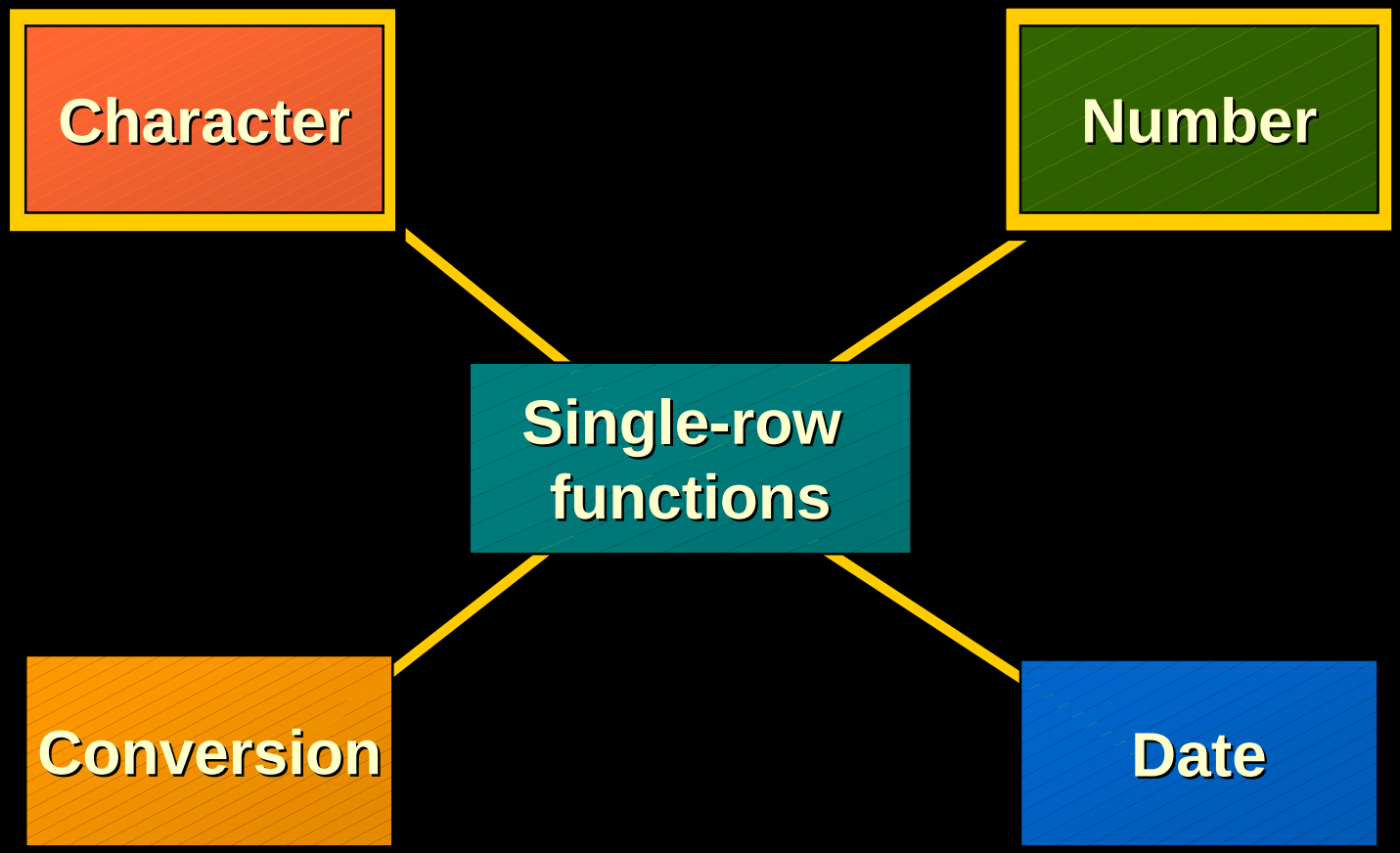
Two Types of SQL Functions



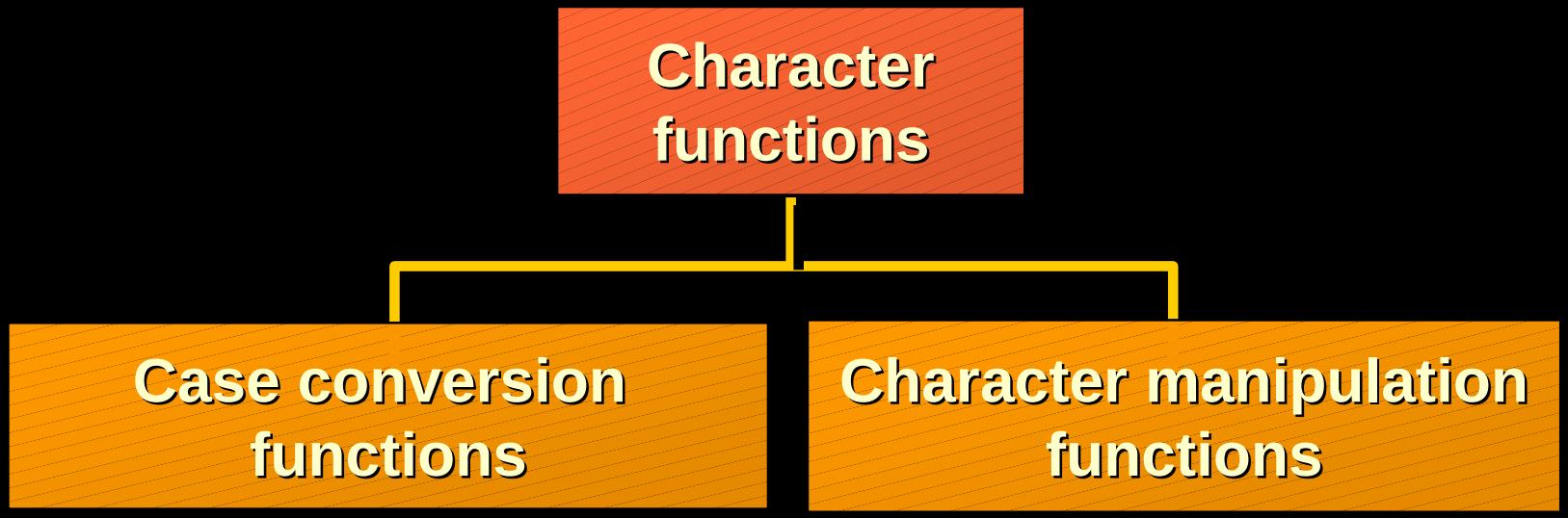
Single-Row Functions

- Manipulate data items
- Accept arguments and return one value
- Act on each row returned
- Return one result per row
- Can modify the data type
- Can be nested

Single-Row Functions



Character Functions



LOWER

UPPER

INITCAP

Case Conversion Functions

- Convert the case for character strings

Function	Result
LOWER('SQL Course')	sql course
UPPER('SQL Course')	SQL COURSE
INITCAP('SQL Course')	Sql Course

Using Case Conversion Functions

- Display the employee number, name, and department number for employee Blake.

```
SQL> SELECT   empno, ename, deptno
  2  FROM     emp
  3  WHERE    ename = 'blake';
no rows selected
```

```
SQL> SELECT   empno, ename, deptno
  2  FROM     emp
  3  WHERE    ename = UPPER('blake');
```

```
     EMPNO ENAME          DEPTNO
--------- ---------- ---------
      7698 BLAKE             30
```

Using Case Conversion Functions

- Display the employee name for all employees with an initial capital.

```
SQL> SELECT INITCAP(ename) as EMPLOYEE
  2  FROM   emp;
```

```
EMPLOYEE
----------
King
Blake
Clark
Jones
Martin
...
14 rows selected.
```

Number Functions

- ROUND: Rounds value to specified decimal

  ROUND(45.926, 2)

  45.93

- TRUNC: Truncates value to specified decimal

  TRUNC(45.926, 2)

  45.92

- MOD: Returns remainder of division

  MOD(1600, 300)

  100

Defining a Null Value

- A null is a value that is unavailable, unassigned, unknown, or inapplicable.
- A null is not the same as zero or a blank space.

```
SQL> SELECT   ename, job, comm
  2  FROM     emp;
```

```
ENAME      JOB          COMM
---------- --------- ---------
KING       PRESIDENT
BLAKE      MANAGER
...
TURNER     SALESMAN         0
...
14 rows selected.
```

Null Values in Arithmetic Expressions

- Arithmetic expressions that contain a null value evaluate to null.

```
SQL> SELECT ename NAME, 12*sal+comm
  2  FROM   emp;
```

```
NAME       12*SAL+COMM
---------- -----------
KING
BLAKE
CLARK
JONES
MARTIN           16400
...
14 rows selected.
```

Using the NVL Function

```
NVL (expr1, expr2)
```

- Use the NVL function to force a value where a null would otherwise appear:
  - NVL can be used with date, character, and number data types.
  - Data types must match. For example:
    - NVL(comm,0)
    - NVL(hiredate,'01-JAN-97')
    - NVL(job,'no job yet')

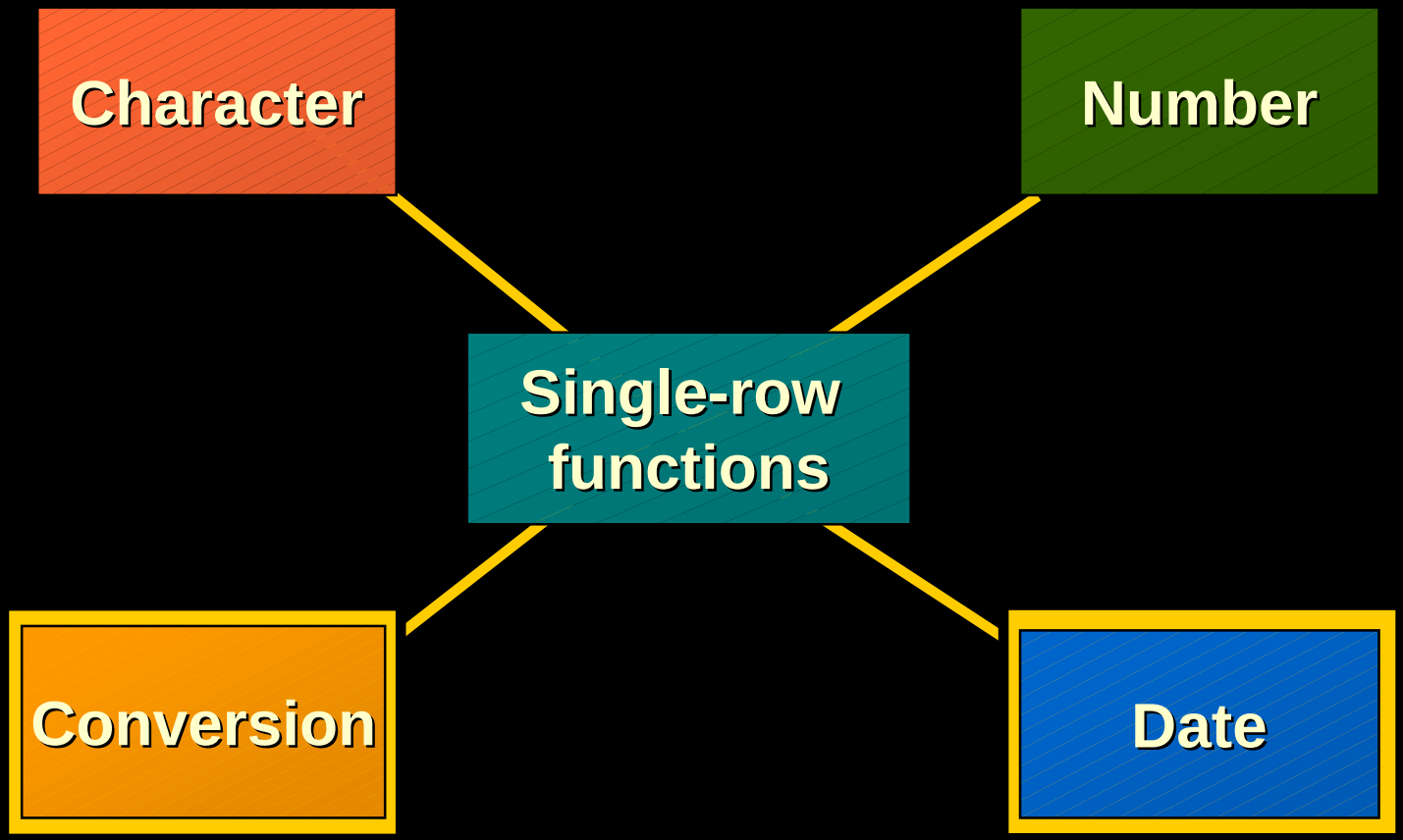
Using the NVL Function to Handle Null Values

```
SQL> SELECT ename, job, sal * 12 + NVL(comm,0)
  2  FROM   emp;
```

```
ENAME      JOB       SAL*12+NVL(COMM,0)
---------- --------- ------------------
KING       PRESIDENT              60000
BLAKE      MANAGER                34200
CLARK      MANAGER                29400
JONES      MANAGER                35700
MARTIN     SALESMAN               16400
ALLEN      SALESMAN               19500
TURNER     SALESMAN               18000
...
14 rows selected.
```

Single-Row Date and Conversion Functions

Single-Row Functions



Working with Dates

- Oracle stores dates in an internal 7 byte numeric format: century, year, month, day, hours, minutes, seconds.
- The default date format is DD-MON-YY.

SYSDATE

- Use SYSDATE to display the current date and time.
- DUAL is a one-column, one-row table that is used as a dummy table.

```
SQL> SELECT SYSDATE
  2  FROM   DUAL;
```

```
SYSDATE
---------
26-JAN-98
```

Default Date Formats

Columns that are defined as DATE are displayed as DD-MON-YY by default.

```
SQL> SELECT ename, hiredate
  2  FROM   emp
  3  WHERE  ename='SMITH';
```

```
ENAME      HIREDATE
---------- ---------
SMITH      17-DEC-80
```

Arithmetic with Dates

- Add or subtract a number to or from a date to obtain a *date* value
- Subtract two dates to find the *number* of days between those dates

Using Arithmetic Operators with Dates

```
SQL> SELECT ename, hiredate, hiredate+30 "NEW DATE"
  2  FROM   emp
  3  WHERE  ename='SMITH';
```

```
ENAME      HIREDATE  NEW DATE
---------- --------- ---------
SMITH      17-DEC-80 16-JAN-81
```

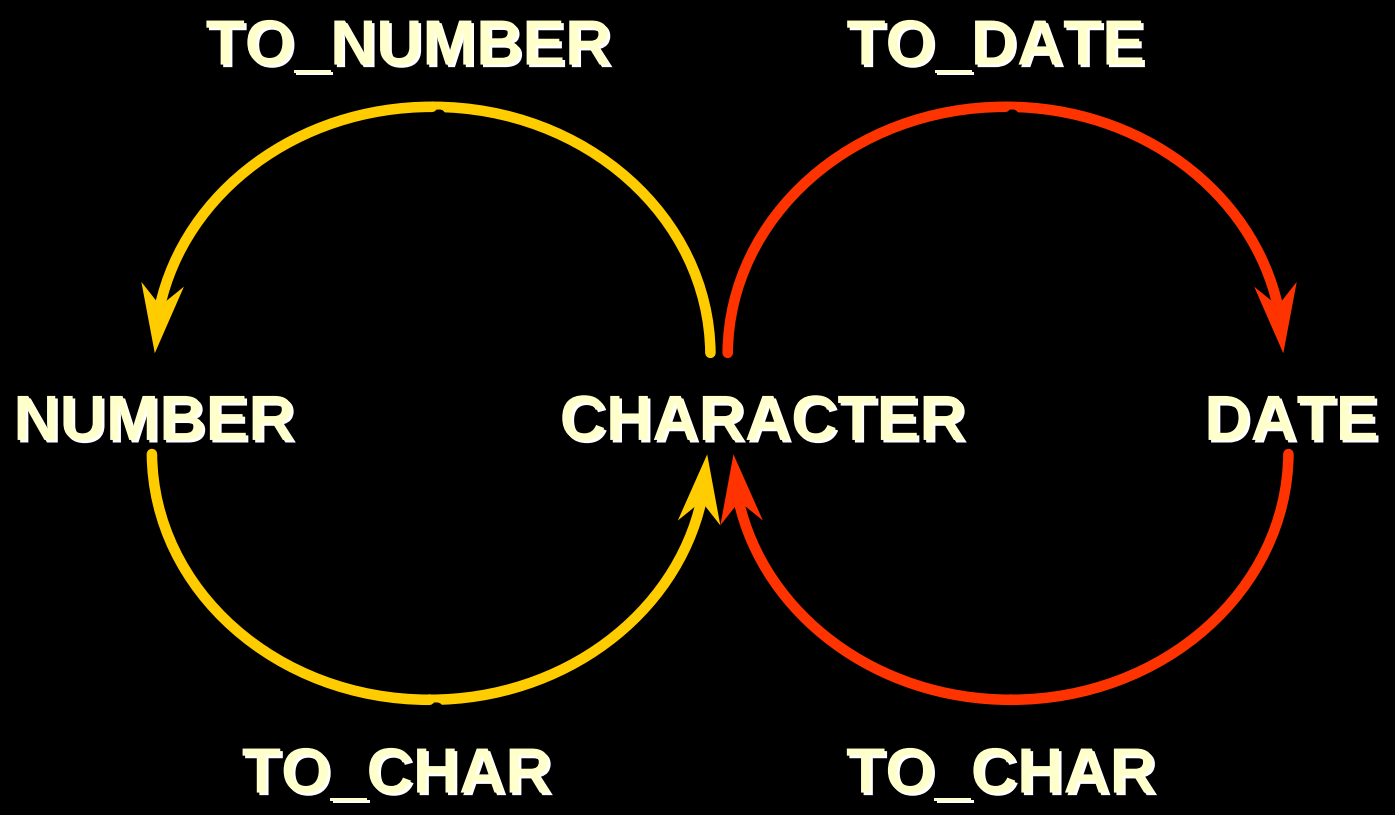
Using SYSDATE in Calculations

- Determine for how many weeks employees have worked

```
SQL> SELECT ename, (SYSDATE-hiredate)/7
  2 "WEEKS AT WORK"
  3  FROM  emp
  4  WHERE deptno=10;
```

```
ENAME      WEEKS AT WORK
---------- -------------
KING           844.94617
CLARK          867.94617
MILLER         835.37474
```

Explicit Data Type Conversion



Modifying the Display Format of Dates

27-JAN-98 → Tuesday the 27th of January, 1998

27-JAN-98 → January 27, 1998

27-JAN-98 → 01/27/98

TO_CHAR Function with Dates

```
TO_CHAR(date, 'fmfmt')
```

- The format model:
  - Is case sensitive and must be enclosed in single quotation marks
  - Can include any valid date format element
  - Has an *fm* element to remove padded blanks or suppress leading zeros
  - Is separated from the date value by a comma

Date Format Model Elements

YYYY	Full year in numbers
YEAR	Year spelled out
MM	2-digit value for month
MONTH	Full name of the month
DY	3-letter abbreviation of the day of the week
DAY	Full name of the day

Using the TO_CHAR Function with Dates

```
SQL> SELECT ename, TO_CHAR(hiredate, 'Month DDth, YYYY')
  2  AS HIREDATE
  3  FROM   emp
  4  WHERE job='MANAGER';
```

```
ENAME      HIREDATE
---------- --------------------
BLAKE      May       01st, 1981
CLARK      June      09th, 1981
JONES      April     02nd, 1981
```

Using the TO_CHAR Function with Dates

```
SQL> SELECT   empno, TO_CHAR(hiredate, 'MM/YY') AS MONTH
  2  FROM     emp
  3  WHERE    ename='BLAKE';
```

```
     EMPNO MONTH
--------- -----
      7698 05/81
```

Using the TO_CHAR Function with Dates

```
SQL> SELECT ename,
  2  TO_CHAR(hiredate, 'fmDD Month YYYY') AS HIREDATE
  3  FROM   emp;
```

```
ENAME      HIREDATE
---------- -----------------
KING       17 November 1981
BLAKE      1 May 1981
CLARK      9 June 1981
JONES      2 April 1981
MARTIN     28 September 1981
ALLEN      20 February 1981
...
14 rows selected.
```

Using the TO_CHAR Function with Dates

```
SQL> SELECT ename, mgr, sal,TO_CHAR(hiredate,'YYYY-MON-DD')
  2  AS     HIREDATE
  3  FROM   emp
  4  WHERE  sal<1000
  5  AND hiredate like '%80';
```

```
ENAME              MGR        SAL HIREDATE
---------- --------- --------- -----------
SMITH           7902        800 1980-DEC-17
```

Using the TO_CHAR Function with Dates

```
SQL> SELECT empno,ename,deptno,TO_CHAR(hiredate,'MM-DD-YYYY')
  2  AS     HIREDATE
  3  FROM   emp
  4  WHERE hiredate NOT LIKE '%81';
```

```
     EMPNO ENAME          DEPTNO HIREDATE
  -------- ---------- --------- ----------
      7369 SMITH             20 12-17-1980
      7788 SCOTT             20 12-09-1982
      7876 ADAMS             20 01-12-1983
      7934 MILLER            10 01-23-1982
```

Using the TO_CHAR Function with Dates

```
SQL> SELECT    ename, job, deptno,
  2  TO_CHAR(hiredate,'DD-MON-YYYY') AS HIRE_DATE
  3  FROM      emp
  4  ORDER BY hiredate DESC;
```

```
ENAME      JOB           DEPTNO HIRE_DATE
---------- --------- --------- -----------
ADAMS      CLERK             20 12-JAN-1983
SCOTT      ANALYST           20 09-DEC-1982
MILLER     CLERK             10 23-JAN-1982
JAMES      CLERK             30 03-DEC-1981
FORD       ANALYST           20 03-DEC-1981
KING       PRESIDENT         10 17-NOV-1981
MARTIN     SALESMAN          30 28-SEP-1981
...
14 rows selected.
```

Date Format Model Elements

- Time elements format the time portion of the date.

HH24:MI:SS AM	15:45:32 PM
DD "of" MONTH	12 of OCTOBER
ddspth	fourteenth

Using Format Models to Display Time

```
SQL> SELECT TO_CHAR(SYSDATE,'HH24:MI:SS') TIME
  2  FROM   DUAL;
```

```
TIME
--------
13:55:46
```

TO_CHAR Function with Numbers

- `TO_CHAR(n,'fmt')`

  display a number value as a character:

9	Represents a number
0	Forces a zero to be displayed
$	Places a floating dollar sign
L	Uses the floating local currency symbol
.	Prints a decimal point
,	Places a thousand indicator

Using the TO_CHAR Function with Numbers

```
SQL> SELECT   TO_CHAR(sal,'$99,999') SALARY
  2  FROM     emp
  3  WHERE    ename = 'SCOTT';
```

```
  SALARY
--------
  $3,000
```

Thousand indicator

Dollar sign

Using the TO_NUMBER and TO_DATE Functions

- Convert a character string to a number data type using the TO_NUMBER function

```
TO_NUMBER(char)
```

- **Convert a character string to a date data type using the TO_DATE function**

```
TO_DATE(char[, 'fmt'])
```

Using the TO_NUMBER Function

```
SQL> SELECT    TO_NUMBER('1000')+sal AS NEW_SALARY
  2  FROM      emp
  3  WHERE     ename = 'SCOTT';
```

```
NEW_SALARY
----------
      4000
```

Date Functions

FUNCTION	DESCRIPTION
MONTHS_BETWEEN	Number of months between two dates
ADD_MONTHS	Adds calendar months to date
NEXT_DAY	Next day following the date specified
LAST_DAY	Last day of the month
ROUND	Round off date
TRUNC	Truncate date

Using Date Functions

- Use the ADD_MONTHS function to add months to a date.

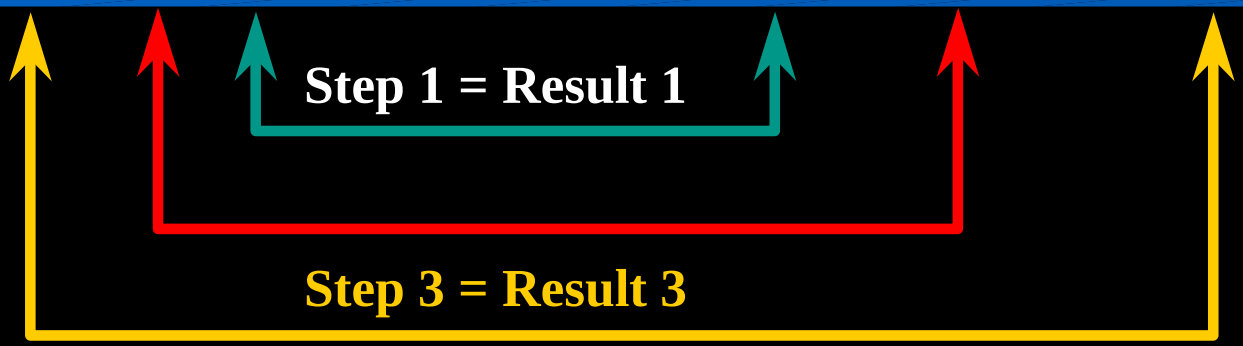
```
SQL> SELECT ename, hiredate, ADD_MONTHS(hiredate, 6)
  2  AS      "+6 MONTHS"
  3  FROM    emp
  4  WHERE ename='BLAKE';
```

```
ENAME      HIREDATE  +6 MONTHS
---------- --------- ---------
BLAKE      01-MAY-81 01-NOV-81
```

Nesting Functions

- Single-row functions can be nested to any level.
- Nested functions are evaluated from the innermost level to the outermost level.

```
F3(F2(F1(col,arg1),arg2),arg3)
```



Nesting Functions

Result 1

```
SQL> SELECT   ename,
  2           NVL(TO_CHAR(mgr),'No Manager')
  3  FROM     emp
  4  WHERE    mgr IS NULL;
```

```
ENAME      NVL(TO_CHAR(MGR),'NOMANAGER')
---------- -----------------------------
KING       No Manager
```

Nesting Functions

```
SQL> SELECT MONTHS_BETWEEN
   2        (TO_DATE('02-02-1995','MM-DD-YYYY'),
   3         TO_DATE('01-01-1995','MM-DD-YYYY'))
   4        "Months"
   5  FROM DUAL;
```

```
 Months
 ----------
 1.03225806
```

Displaying Data from Multiple Tables

Obtaining Data from Multiple Tables

EMP

EMPNO	ENAME	...	DEPTNO
7839	KING	...	10
7698	BLAKE	...	30
...			
7934	MILLER	...	10

DEPT

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

EMPNO	DEPTNO	LOC
7839	10	NEW YORK
7698	30	CHICAGO
7782	10	NEW YORK
7566	20	DALLAS
7654	30	CHICAGO
7499	30	CHICAGO

Joining Tables

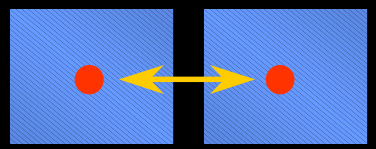
- Use a join to query data from more than one table:

```
SELECT   table1.column1, table2.column2
FROM     table1, table2
WHERE    table1.column1 = table2.column2;
```

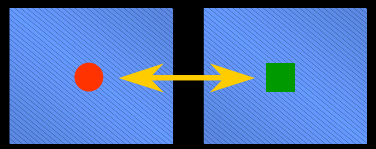
  – Prefix the column name with the table name when the same column name appears in more than one table.

Types of Joins

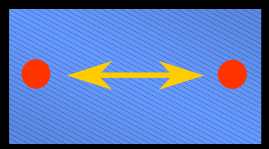
Equijoin



Nonequijoin



Self join



What Is an Equijoin?

EMP

EMPNO	ENAME	DEPTNO
...		
7782	CLARK	10

DEPT

DEPTNO	DNAME	LOC
...		
10	ACCOUNTING	NEW YORK
...		

Links rows that satisfy a specified condition

WHERE emp.deptno=dept.deptno

Equijoin

EMP

EMPNO	ENAME	DEPTNO
7839	KING	10
7698	BLAKE	30
7782	CLARK	10
7566	JONES	20
7654	MARTIN	30
7499	ALLEN	30
7844	TURNER	30
7900	JAMES	30
7521	WARD	30
7902	FORD	20
7369	SMITH	20

...
14 rows selected.

DEPT

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
30	SALES	CHICAGO
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
30	SALES	CHICAGO
30	SALES	CHICAGO
30	SALES	CHICAGO
30	SALES	CHICAGO
20	RESEARCH	DALLAS
20	RESEARCH	DALLAS

...
14 rows selected.

Foreign key **Primary key**

Retrieving Records with an Equijoin

```
SQL> SELECT   emp.empno, emp.ename, emp.deptno,
  2           dept.deptno, dept.loc
  3  FROM     emp, dept
  4  WHERE    emp.deptno=dept.deptno;
```

```
EMPNO ENAME   DEPTNO DEPTNO LOC
----- ------ ------ ------ ----------
 7839 KING       10     10 NEW YORK
 7698 BLAKE      30     30 CHICAGO
 7782 CLARK      10     10 NEW YORK
 7566 JONES      20     20 DALLAS
...
14 rows selected.
```

Qualifying Ambiguous Column Names

- Use table prefixes to qualify column names that are in multiple tables.
- Use table prefixes to improve performance.

Additional Search Conditions Using the AND Operator

EMP

EMPNO	ENAME	DEPTNO
7839	KING	10
7698	BLAKE	30
7782	CLARK	10
7566	JONES	20
7654	MARTIN	30
7499	ALLEN	30
7844	TURNER	30
7900	JAMES	30

...
14 rows selected.

DEPT

NAME	LOC
CCOUNTING	NEW YORK
SALES	CHICAGO
CCOUNTING	NEW YORK
ESEARCH	DALLAS
LES	CHICAGO
LES	CHICAGO
LES	CHICAGO
LES	CHICAGO

lected.

WHERE emp.deptno=dept.deptno AND ename='KING'

Using Additional Search Conditions with a Join

```
SQL> SELECT   emp.empno, emp.ename, emp.deptno, dept.loc
  2  FROM     emp, dept;
  3  WHERE    emp.deptno = dept.deptno
  4  AND      emp.ename = 'KING';
```

```
     EMPNO ENAME          DEPTNO LOC
---------- ---------- ---------- -------------
      7839 KING               10 NEW YORK
```

Using Additional Search Conditions with a Join

```
SQL> SELECT emp.ename, emp.job, dept.deptno, dept.dname
  2  FROM   emp, dept
  3  WHERE  emp.deptno=dept.deptno
  4  AND    emp.job IN ('MANAGER','PRESIDENT');
```

ENAME	JOB	DEPTNO	DNAME
KING	PRESIDENT	10	ACCOUNTING
BLAKE	MANAGER	30	SALES
CLARK	MANAGER	10	ACCOUNTING
JONES	MANAGER	20	RESEARCH

Table Aliases

- Simplify queries by using table aliases.

```
SQL> SELECT emp.empno, emp.ename, emp.deptno,
  2         dept.deptno, dept.loc
  3  FROM   emp, dept
  4  WHERE  emp.deptno=dept.deptno;
```

... can be written as ...

```
SQL> SELECT e.empno, e.ename, e.deptno,
  2         d.deptno, d.loc
  3  FROM   emp e, dept d
  4  WHERE  e.deptno=d.deptno;
```

Using Table Aliases

```
SQL> SELECT e.empno, e.ename, e.deptno,
  2         d.deptno, d.loc
  3  FROM   emp e, dept d
  4  WHERE  e.deptno=d.deptno;
```

```
     EMPNO ENAME          DEPTNO    DEPTNO LOC
--------- ---------- --------- --------- -----------
      7839 KING               10        10 NEW YORK
      7698 BLAKE              30        30 CHICAGO
      7782 CLARK              10        10 NEW YORK
      7566 JONES              20        20 DALLAS
      7654 MARTIN             30        30 CHICAGO
      7499 ALLEN              30        30 CHICAGO
...

14 rows selected.
```

Nonequijoins

EMP

EMPNO	ENAME	SAL
7839	KING	5000
7698	BLAKE	2850
7782	CLARK	2450
7566	JONES	2975
7654	MARTIN	1250
7499	ALLEN	1600
7844	TURNER	1500
7900	JAMES	950

...
14 rows selected.

SALGRADE

GRADE	LOSAL	HISAL
1	700	1200
2	1201	1400
3	1401	2000
4	2001	3000
5	3001	9999

Salary in the EMP table is between low salary and high salary in the SALGRADE table.

Retrieving Records with Nonequijoins

```
SQL>  SELECT   e.ename, e.sal, s.grade
   2  FROM     emp e,   salgrade s
   3  WHERE    e.sal
   4  BETWEEN  s.losal AND s.hisal;
```

```
ENAME          SAL      GRADE
---------- --------- ---------
JAMES          950          1
SMITH          800          1
ADAMS         1100          1
...
14 rows selected.
```

Joining More Than Two Tables

EMP

ENAME	SAL	DEPTNO
JAMES	950	30
SMITH	800	20
ADAMS	1100	20
MARTIN	1250	30
WARD	1250	30
MILLER	1300	10

…

14 rows selected.

DEPT

DEPTNO	DNAME
10	ACCOUNTING
20	RESEARCH
30	SALES
40	OPERATIONS

SALGRADE

LOSAL	HISAL	GRADE
700	1200	1
1201	1400	2
1401	2000	3
2001	3000	4
3001	9999	5

WHERE emp.sal BETWEEN salgrade.losal AND salgrade.hisal AND emp.deptno = dept.deptno

Using Multiple Joins

```
SQL> SELECT e.ename, e.deptno, d.dname, e.sal, s.grade
  2  FROM   emp e,    dept d,    salgrade s
  3  WHERE  e.deptno=d.deptno
  4  AND    e.sal BETWEEN s.losal and s.hisal;
```

ENAME	DEPTNO	DNAME	SAL	GRADE
JAMES	30	SALES	950	1
SMITH	20	RESEARCH	800	1
ADAMS	20	RESEARCH	1100	1
MARTIN	30	SALES	1250	2
WARD	30	SALES	1250	2
MILLER	10	ACCOUNTING	1300	2
ALLEN	30	SALES	1600	3

...
14 rows selected.

Selfjoins

EMP (WORKER)			EMP (MANAGER)	
EMPNO	ENAME	MGR	EMPNO	ENAME
---	---	---	---	---
7839	KING			
7698	BLAKE	7839	7839	KING
7782	CLARK	7839	7839	KING
7566	JONES	7839	7839	KING
7654	MARTIN	7698	7698	BLAKE
7499	ALLEN	7698	7698	BLAKE

MGR in the WORKER table is equal to EMPNO in the MANAGER table.

Joining a Table to Itself

```
SQL> SELECT worker.ename||' works for '||manager.ename
  2  AS      WHO_WORKS_FOR_WHOM
  3  FROM    emp worker, emp manager
  4  WHERE   worker.mgr = manager.empno;
```

```
WHO_WORKS_FOR_WHOM
------------------------------
BLAKE works for KING
CLARK works for KING
JONES works for KING
MARTIN works for BLAKE
...
13 rows selected.
```

Aggregating Data by Using Group Functions

What Are Group Functions?

- Group functions operate on sets of rows to give one result per group.

EMP

DEPTNO	SAL
10	2450
10	5000
10	1300
20	800
20	1100
20	3000
20	3000
20	2975
30	1600
30	2850
30	1250
30	950
30	1500
30	1250

maximum salary in the EMP table

MAX(SAL)
5000

Types of Group Functions

- AVG
- COUNT
- MAX
- MIN
- SUM

Guidelines for Using Group Functions

Many aggregate functions accept these options:

- DISTINCT
- ALL
- NVL

Using the AVG and SUM Functions

- You can use AVG and SUM for numeric data.

```
SQL> SELECT   AVG(sal), SUM(sal)
  2  FROM     emp
  3  WHERE    job LIKE 'SALES%';
```

```
AVG(SAL)  SUM(SAL)
--------- ----------
     1400       5600
```

Using the MIN and MAX Functions

- You can use MIN and MAX for any data type.

```
SQL> SELECT   TO_CHAR(MIN(hiredate),'DD-MON-YYYY'),
  2           TO_CHAR(MAX(hiredate),'DD-MON-YYYY')
  3  FROM     emp;
```

```
TO_CHAR(MIN TO_CHAR(MAX
----------- -----------
17-DEC-1980 12-JAN-1983
```

Using the MIN and MAX Functions

- You can use MIN and MAX for any data type.

```
SQL> SELECT   MIN(sal) AS "Lowest Salary",
  2  MAX(sal) AS "Highest Salary"
  3  FROM     emp;
```

```
Lowest Salary Highest Salary
------------- --------------
          800           5000
```

Using the COUNT Function

- COUNT(*) returns the number of rows in a query.

```
SQL> SELECT   COUNT(*)
  2  FROM     emp
  3  WHERE    deptno = 30;
```

```
  COUNT(*)
----------
         6
```

Using the COUNT Function

- COUNT(*expr*) returns the number of nonnull rows.

```
SQL> SELECT   COUNT(comm)
  2  FROM     emp
  3  WHERE    deptno = 30;
```

```
COUNT(COMM)
-----------
          4
```

Group Functions and Null Values

- Group functions ignore null values in the column.

```
SQL> SELECT AVG(comm)
  2  FROM   emp;
```

```
 AVG(COMM)
---------
       550
```

Using the NVL Function with Group Functions

- The NVL function forces group functions to include null values.

```
SQL> SELECT AVG(NVL(comm,0))
  2  FROM   emp;
```

```
AVG(NVL(COMM,0))
----------------
       157.14286
```

Using the NVL Function with Group Functions

- Average commission for *all* people hired in 1981

```
SQL> SELECT  AVG(NVL(comm,0))
  2  FROM    emp
  3  WHERE   hiredate
  4  BETWEEN TO_DATE('01-JAN-1981','DD-MON-YYYY')
  5  AND     TO_DATE('31-DEC-1981','DD-MON-YYYY');
```

```
AVG(NVL(COMM,0))
----------------
             220
```

Creating Groups of Data

EMP

DEPTNO	SAL
10	2450
10	5000
10	1300
20	800
20	1100
20	3000
20	3000
20	2975
30	1600
30	2850
30	1250
30	950
30	1500
30	1250

2916.6667

2175

1566.6667

average salary in EMP table for each department

DEPTNO	AVG(SAL)
10	2916.6667
20	2175
30	1566.6667

Creating Groups of Data: GROUP BY Clause

- Use the GROUP BY clause to divide rows in a table into smaller groups.

```
SELECT        column, group_function
FROM          table
[WHERE        condition]
[GROUP BY     group_by_expression]
[ORDER BY     column];
```

Using the GROUP BY Clause

- All columns in the SELECT list that are not in group functions must be in the GROUP BY clause.

```
SQL> SELECT   deptno, AVG(sal)
  2  FROM     emp
  3  GROUP BY deptno;
```

```
    DEPTNO   AVG(SAL)
---------- ---------
        10 2916.6667
        20      2175
        30 1566.6667
```

Using the GROUP BY Clause

- The GROUP BY column does not have to be in the SELECT list.

```
SQL> SELECT    AVG(sal)
  2  FROM      emp
  3  GROUP BY deptno;
```

```
 AVG(SAL)
---------
2916.6667
     2175
1566.6667
```

Using the GROUP BY Clause

- Display the number of people in each department.

```
SQL> SELECT   deptno, COUNT(*) AS "Dept Employees"
  2  FROM     emp
  3  GROUP BY deptno;
```

```
    DEPTNO Dept Employees
--------- --------------
        10              3
        20              5
        30              6
```

Using a Group Function in the ORDER BY Clause

```
SQL> SELECT    deptno, AVG(sal)
  2  FROM      emp
  3  GROUP BY deptno
  4  ORDER BY AVG(sal);
```

```
    DEPTNO     AVG(SAL)
---------- -------------
        30    1566.6667
        20         2175
        10    2916.6667
```

Illegal Queries Using Group Functions

- Any column or expression in the SELECT list that is not an aggregate function must be in the GROUP BY clause.

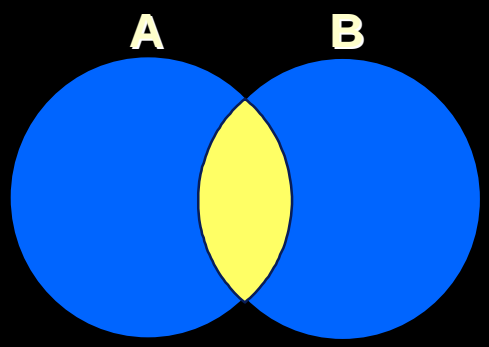
```
SQL> SELECT   deptno, COUNT(ename)
  2  FROM     emp;
```

```
SELECT deptno, COUNT(ename)
       *
ERROR at line 1:
ORA-00937: not a single-group group function
```

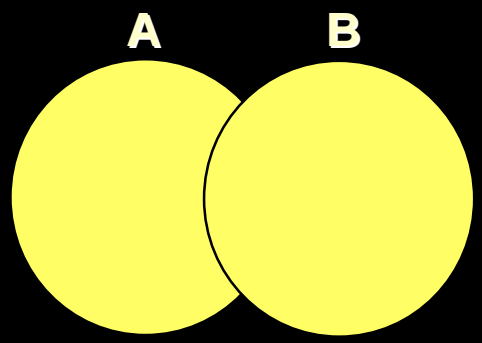
Column missing in the GROUP BY clause

Using Set Operators

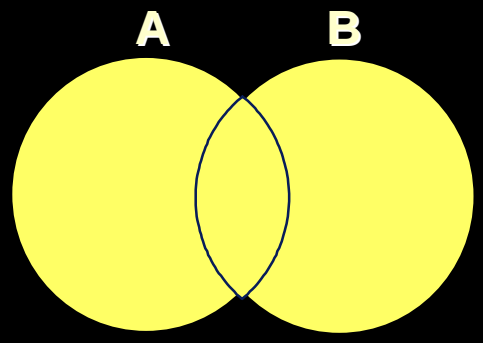
The Set Operators

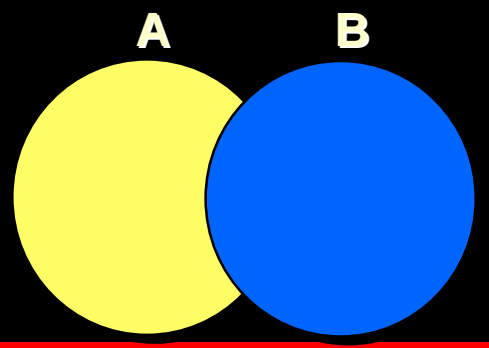


Intersect



Union / Union All





Minus

Tables Used in This Lesson

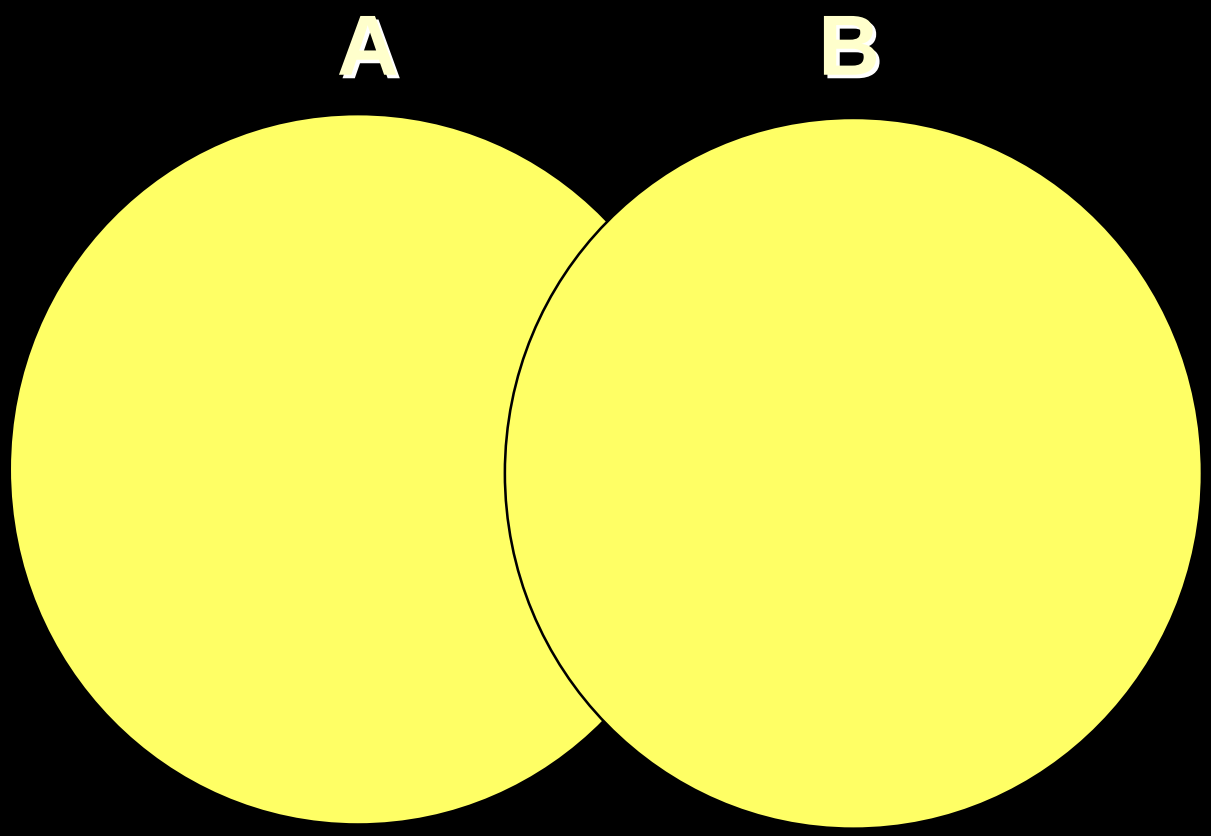
EMP

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	KING	PRESIDENT		17-NOV-81	5000		10
7698	BLAKE	MANAGER	7839	01-MAY-81	2850		30
7782	CLARK	MANAGER	7839	09-JUN-81	1500		10
7566	JONES	MANAGER	7839	02-APR-81	2975		20
7654	MARTIN	SALESMAN					30
7499	ALLEN	SALESMAN					30

EMP_HISTORY

EMPID	NAME	TITLE	DATE_OUT	DEPTID
6087	SPENCER	OPERATOR	27-NOV-81	20
6185	VANDYKE	MANAGER	17-JAN-81	10
6235	BALFORD	CLERK	22-FEB-80	20
7788	SCOTT	ANALYST	05-MAY-81	20
7001	JEWELL	ANALYST	10-JUN-81	20

UNION



Using the UNION Operator

- Display the name, employee number, and job title of all employees. Display each employee only once.

```
SQL> SELECT ename, empno, job
  2  FROM   emp
  3  UNION
  4  SELECT name, empid, title
  5  FROM   emp_history;
```

```
ENAME           EMPNO JOB
---------- --------- ---------
ADAMS            7876 CLERK
ALLEN            7499 SALESMAN
BALFORD          6235 CLERK
...
20 rows selected.
```

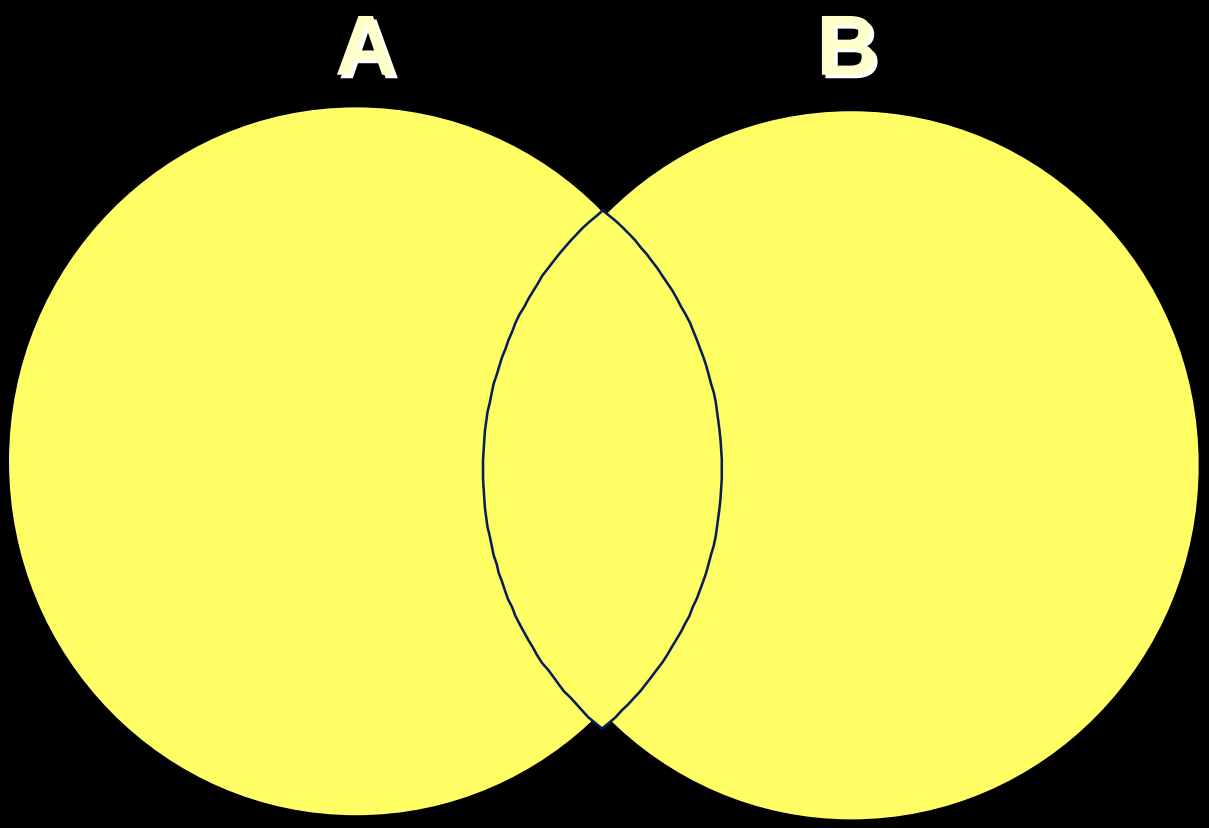
Using the UNION Operator

- Display the name, job title, and salary of all employees.

```
SQL> SELECT ename, job, sal
  2  FROM   emp
  3  UNION
  4  SELECT name, title, 0
  5  FROM   emp_history;
```

```
ENAME      JOB              SAL
---------- --------- ---------
ADAMS      CLERK           1100
ALLEN      SALESMAN           0
ALLEN      SALESMAN        1600
BALFORD    CLERK              0
...
23 rows selected.
```

UNION ALL



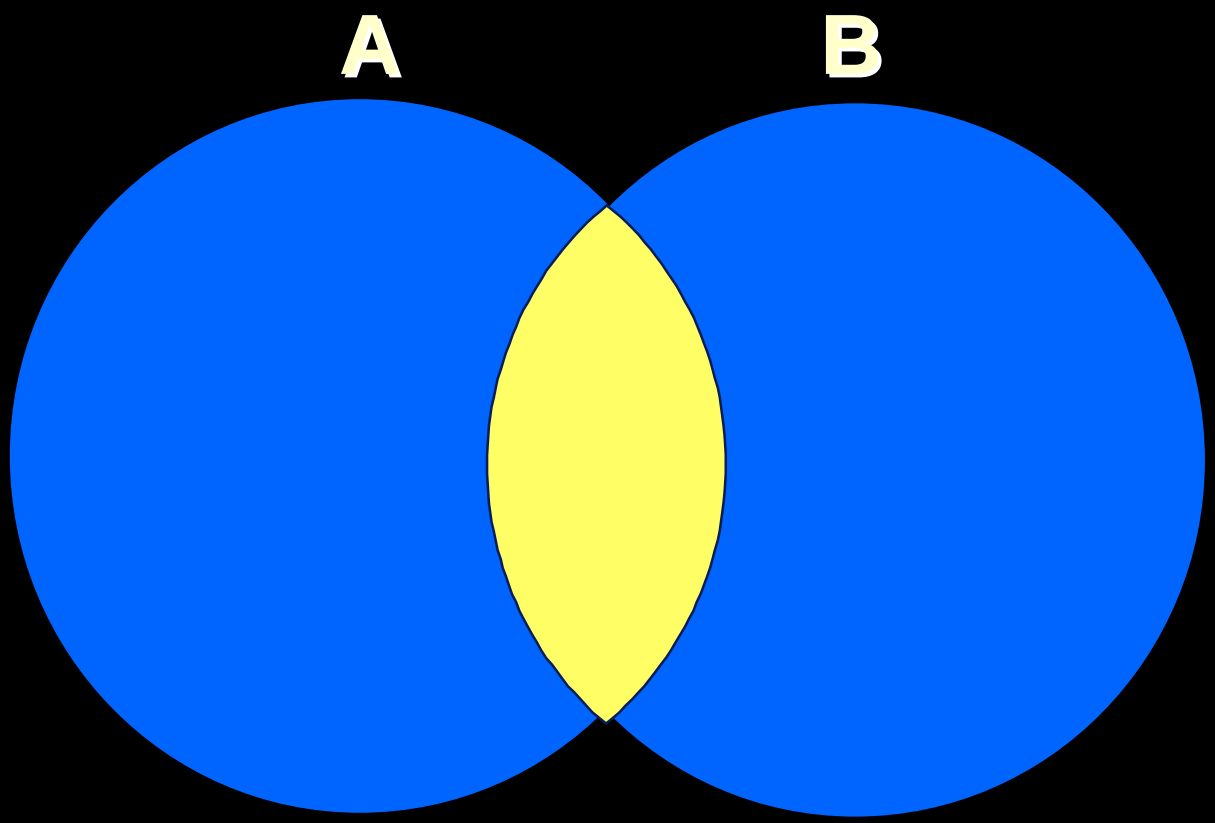
Using the UNION ALL Operator

- Display the names, employee numbers, and job titles of all employees.

```
SQL> SELECT ename, empno, job
  2  FROM   emp
  3  UNION ALL
  4  SELECT name, empid, title
  5  FROM   emp_history;
```

```
ENAME           EMPNO JOB
---------- --------- ---------
KING             7839 PRESIDENT
BLAKE            7698 MANAGER
CLARK            7782 MANAGER
CLARK            7782 MANAGER
...
23 rows selected.
```

INTERSECT



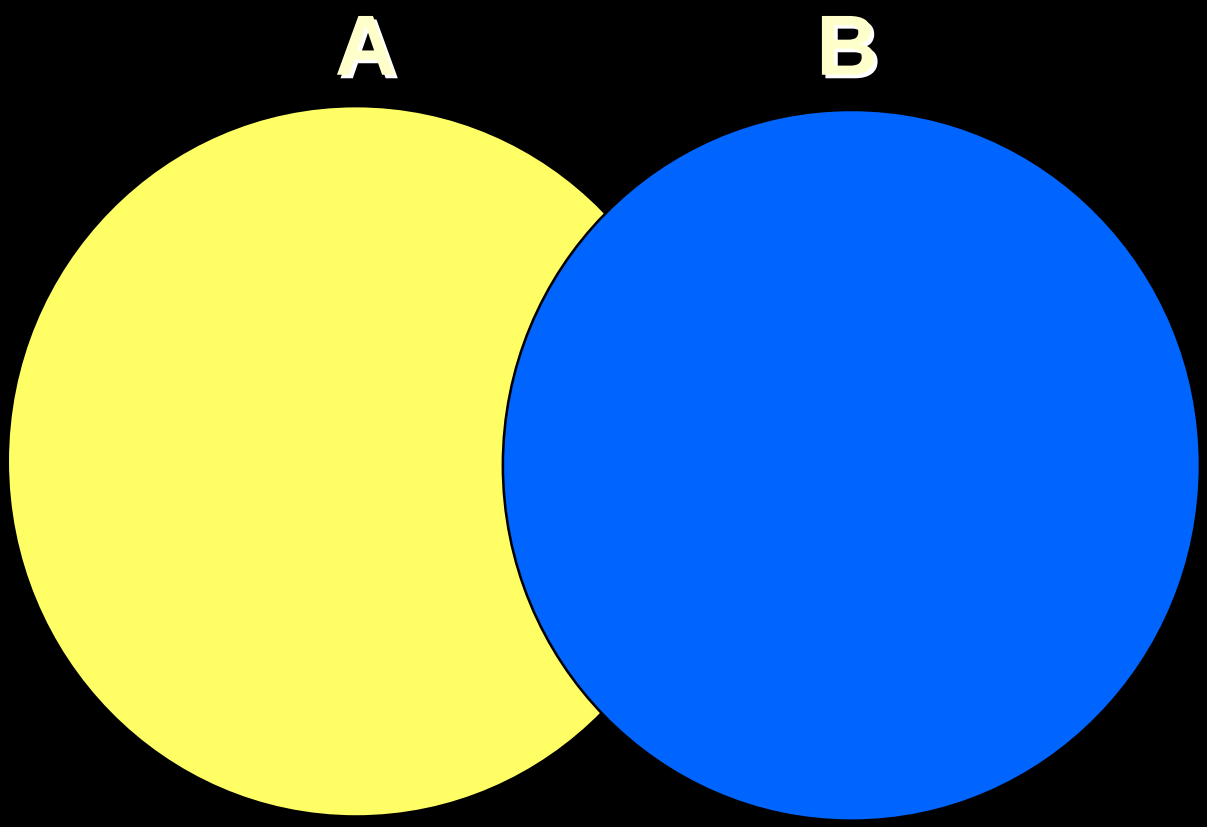
Using the INTERSECT Operator

- Display the distinct names, employee numbers, and job titles of employees found in both the EMP and EMP_HISTORY tables.

```
SQL> SELECT ename, empno, job
  2  FROM   emp
  3  INTERSECT
  4  SELECT name, empid, title
  5  FROM   emp_history;
```

```
ENAME           EMPNO JOB
---------- --------- ---------
ALLEN            7499 SALESMAN
CLARK            7782 MANAGER
SCOTT            7788 ANALYST
```

MINUS



MINUS

Display the names, employee numbers, and job titles for all employees who have left the company.

```
SQL> SELECT name, empid, title
  2  FROM   emp_history
  3  MINUS
  4  SELECT ename, empno, job
  5  FROM   emp;
```

```
NAME           EMPID TITLE
---------- --------- ---------
BALFORD         6235 CLERK
BRIGGS          7225 PAY CLERK
...
6 rows selected.
```

SET Operator Rules

- The expressions in the SELECT lists must match in number and datatype.
- 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.
- Parentheses can be used to alter the sequence of execution.

Matching the SELECT Statement

Display the department numbers, locations, and hiredates for all employees.

```
SQL> SELECT deptno, null location, hiredate
  2  FROM   emp
  3  UNION
  4  SELECT deptno, loc, TO_DATE(null)
  5  FROM   dept;
```

Controlling the Order of Rows

Produce an English sentence using two UNION operators.

```
SQL> COLUMN a_dummy NOPRINT
SQL> SELECT 'sing' "My dream", 3 a_dummy
  2  FROM dual
  3  UNION
  4  SELECT 'I''d like to teach', 1
  5  FROM dual
  6  UNION
  7  SELECT 'the world to', 2
  8  FROM dual
  9  ORDER BY 2;
```

```
My dream
-------------------------
I'd like to teach
the world to
sing
```

Writing Subqueries

Using a Subquery to Solve a Problem

- Who has a salary greater than Jones's?

Main Query

Which employees have a salary greater than Jones's salary?

Subquery

What is Jones's salary?

Subqueries

```
SELECT    select_list
FROM      table
WHERE     expr operator
                    (SELECT      select_list
                    FROM         table);
```

- The subquery (inner query) executes once before the main query.
- The result of the subquery is used by the main query (outer query).

Using a Subquery

"Who has a salary greater than Jones'?"

```
SQL> SELECT ename
  2  FROM   emp
  3  WHERE  sal >
  4                (SELECT sal
  5                 FROM   emp
  6                 WHERE  ename='JONES');
```

2975

```
ENAME
----------
KING
FORD
SCOTT
```

Guidelines for Using Subqueries

- Enclose subqueries in parentheses.
- Place subqueries on the right side of the comparison operator.
- Do not add an ORDER BY clause to a subquery.
- Use single-row operators with single-row subqueries.

Types of Subqueries

- Single-row subquery

- uery

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

Who works in the same department as King?

```
SQL> SELECT ename, deptno
  2  FROM   emp
  3  WHERE  deptno =
  4                  (SELECT deptno
  5                  FROM   emp
  6                  WHERE  ename='KING');
```

10

```
ENAME          DEPTNO
---------- ---------
KING               10
CLARK              10
MILLER             10
```

Executing Single-Row Subqueries

Who has the same manager as Blake?

```
SQL> SELECT ename, mgr
  2  FROM   emp
  3  WHERE  mgr =
  4                (SELECT mgr
  5                 FROM   emp
  6                 WHERE  ename='BLAKE');
```

7839

```
ENAME                MGR
---------- ---------
BLAKE             7839
CLARK             7839
JONES             7839
```

Executing Single-Row Subqueries

Who has the same job as employee 7369 *and* earns a higher salary than employee 7876?

```
SQL> SELECT    ename, job
  2  FROM      emp
  3  WHERE     job =
  4                  (SELECT      job
  5                  FROM         emp
  6                  WHERE        empno = 7369)
  7  AND       sal >
  8                  (SELECT      sal
  9                  FROM         emp
 10                  WHERE        empno = 7876);
```

CLERK

1100

```
ENAME      JOB
---------- ---------
MILLER     CLERK
```

Using Group Functions in a Subquery

Display all employees who earn the minimum salary.

```
SQL> SELECT   ename, job, sal
  2  FROM     emp
  3  WHERE    sal =
  4                (SELECT      MIN(sal)
  5                FROM         emp);
```

800

```
ENAME      JOB              SAL
---------- --------- ---------
SMITH      CLERK            800
```

What Is Wrong with This Statement?

```
SQL> SELECT empno, ename
  2  FROM   emp
  3  WHERE  sal =
  4                    (SELECT   MIN(sal)
  5                    FROM      emp
  6                    GROUP BY  deptno);
```

```
ERROR:
ORA-01427: single-row subquery returns more than
one row

no rows selected
```

Single-row operator with multiple-row subquery

Will This Statement Work?

```
SQL> SELECT ename, job
  2  FROM   emp
  3  WHERE  job =
  4                (SELECT job
  5                FROM    emp
  6                WHERE   ename='SMYTHE');
```

```
no rows selected
```

Subquery returns no values

Multiple-Row Subqueries

- Return more than one row
- Use the IN multiple-row comparison operator to compare an expression to any member in the list that a subquery returns

Using Group Functions in a Multiple-Row Subquery

Display all employees who earn the same salary as the minimum salary for each department.

```
SQL> SELECT   ename, sal, deptno
  2  FROM     emp
  3  WHERE    sal IN
  4                  (SELECT   MIN(sal)
  5                   FROM     emp
  6                   GROUP BY deptno);
```

800, 950, 1300

ENAME	SAL	DEPTNO
SMITH	800	20
JAMES	950	30
MILLER	1300	10

Using Group Functions in a Multiple-Row Subquery

Display the employees who were hired on the same date as the longest serving employee in any department.

```
SQL> SELECT  ename, sal, deptno,
  2  TO_CHAR(hiredate,'DD-MON-YYYY')HIREDATE
  3  FROM    emp
  4  WHERE   hiredate  IN
  5                     (SELECT   MIN(hiredate)
  6                      FROM     emp
  7                      GROUP BY deptno);
```

ENAME	SAL	DEPTNO	HIREDATE
SMITH	800	20	17-DEC-1980
ALLEN	1600	30	20-FEB-1981
CLARK	2450	10	09-JUN-1981

Controlling Transactions

Data Manipulation Language

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

Database Transactions

- Database transactions can consist of:
  - DML statements that make up one consistent change to the data

    Example: UPDATE
  - One DDL statement

    Example: CREATE
  - One DCL statement

    Example: GRANT and REVOKE

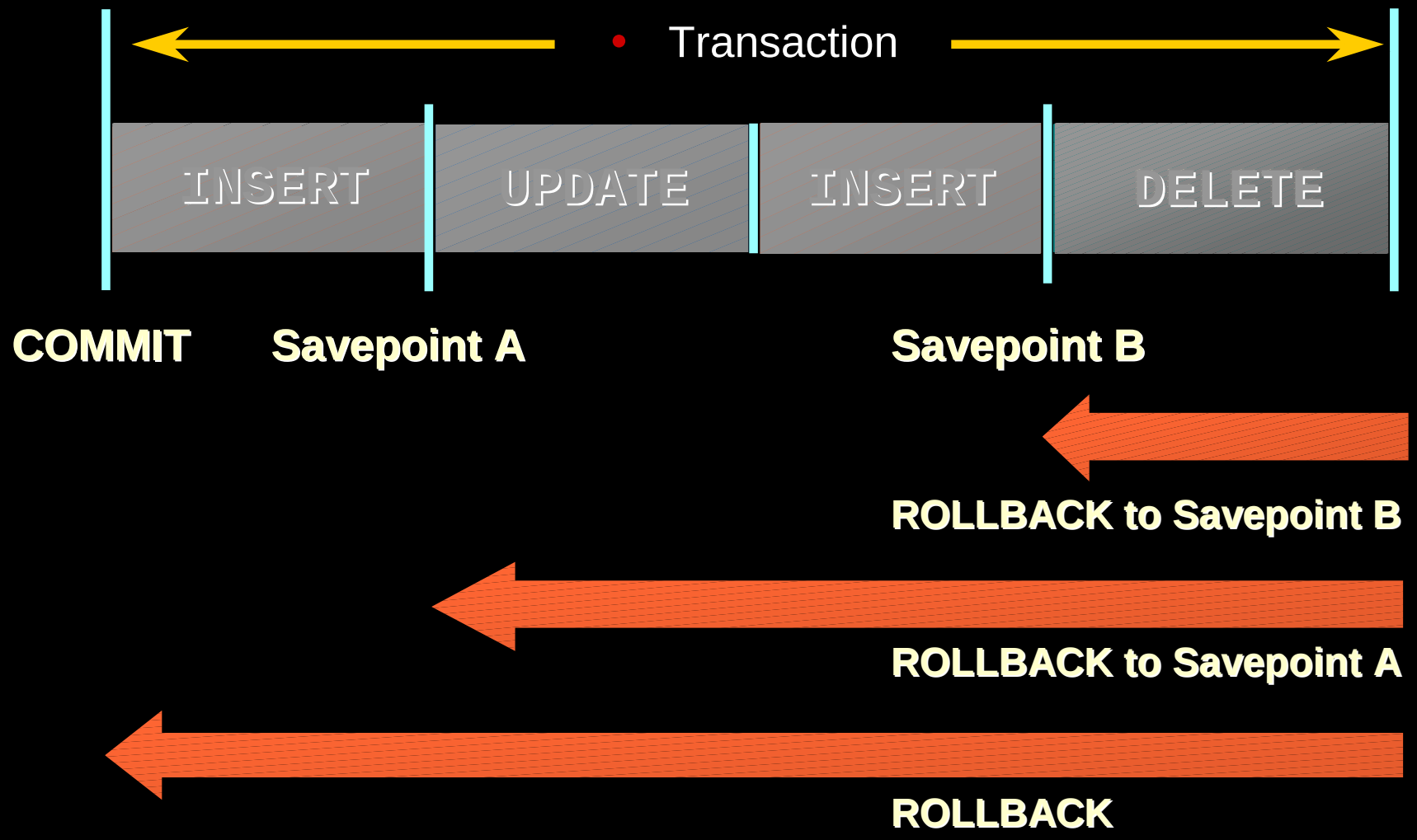
Database Transactions

- Begin when the first executable SQL statement is executed
- End with one of the following events:
  - COMMIT or ROLLBACK
  - DDL or DCL statement executes (automatic commit)
  - User exits
  - System crashes

Advantages of COMMIT and ROLLBACK

- COMMIT and ROLLBACK ensure data consistency.
- Users can preview data changes before making changes permanent.
- Users can group logically related operations.

Controlling Transactions



Implicit Transaction Processing

- An automatic commit occurs under the following circumstances:
  - A DDL statement is issued, such as CREATE
  - A DCL statement is issued, such as GRANT
  - A normal exit from SQL*Plus occurs without an explicitly issued COMMIT or ROLLBACK statement
- An automatic rollback occurs under an abnormal termination of SQL*Plus or a system failure.

State of the Data Before COMMIT or ROLLBACK

- The previous state of the data can be recovered.
- The current user can review the results of the DML operations by using the SELECT statement.
- Other users cannot view the results of the DML statements by the current user.
- The affected rows are locked; other users cannot change the data within the affected rows.

Committing Data

- Change the department number of an employee (Clark) identified by a employee number.
  - Make the changes.

```
SQL> UPDATE   emp
  2  SET      deptno = 10
  3  WHERE    empno = 7782;
1 row updated.
```

- **Commit the changes.**

```
SQL> COMMIT;
Commit complete.
```

State of the Data After COMMIT

- Data changes are made permanent in the database.
- The previous state of the data is permanently lost.
- All users can view the results.
- Locks on the affected rows are released; those rows are available for other users to manipulate.
- All savepoints are erased.

ORACLE
甲骨文

State of the Data After ROLLBACK

- Discard all pending changes by using the ROLLBACK statement. Following a ROLLBACK:
  - Data changes are undone.
  - The previous state of the data is restored.
  - Locks on the affected rows are released.

```
SQL> DELETE FROM employee;
14 rows deleted.
SQL> ROLLBACK;
Rollback complete.
```

Rolling Back Changes to a Marker

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

```
SQL> UPDATE...
SQL> SAVEPOINT update_done;
Savepoint created.
SQL> INSERT...
SQL> ROLLBACK TO update_done;
Rollback complete.
```

Statement-Level Rollback

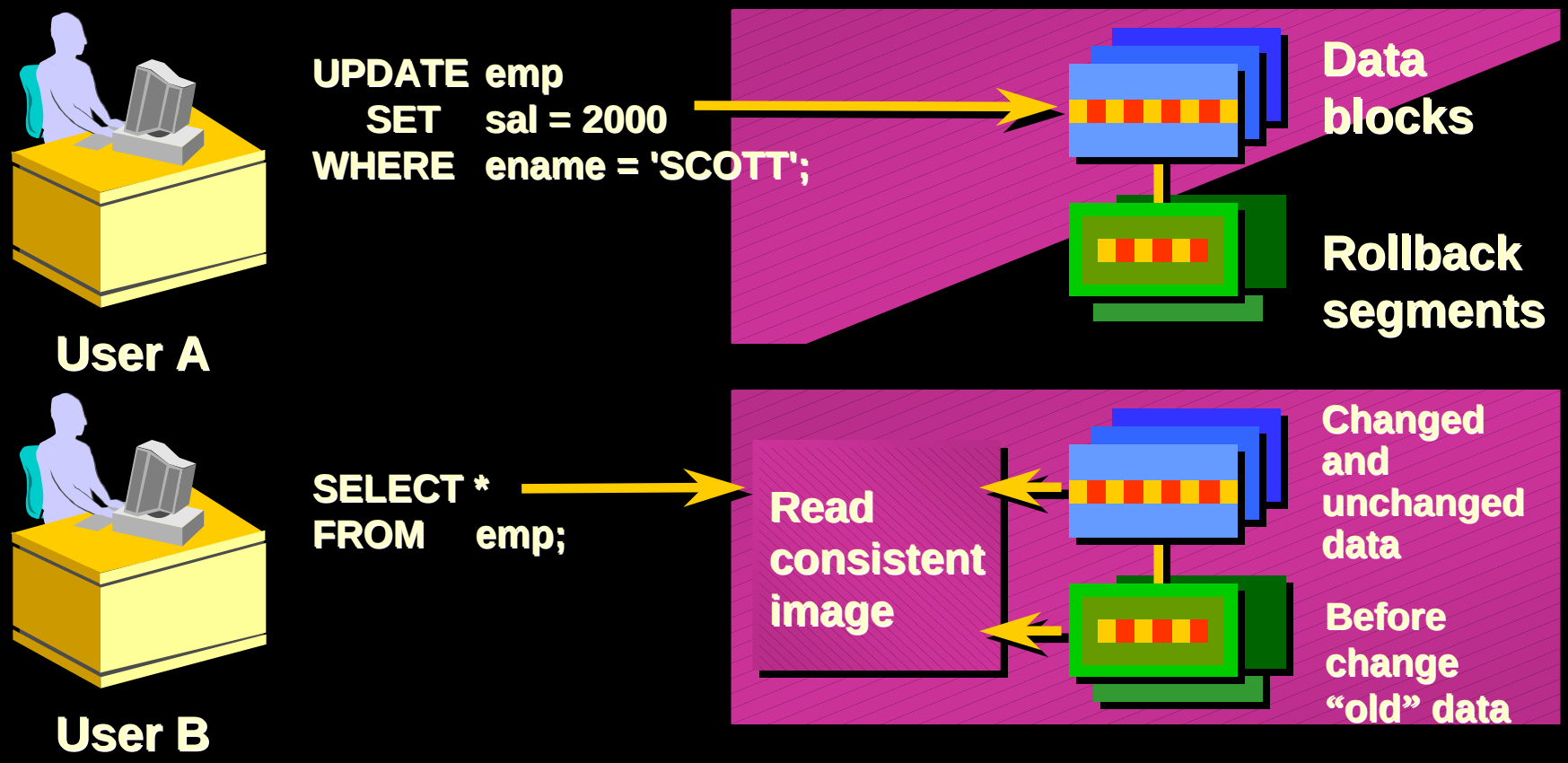
- If a single DML statement fails during execution, only that statement is rolled back.
- Oracle implements an implicit savepoint.
- All other changes are retained.
- The user should terminate transactions explicitly by executing a COMMIT or ROLLBACK statement.

Read Consistency

- Read consistency guarantees a consistent view of the data at all times.
- Changes made by one user do not conflict with changes made by another user.
- Read consistency ensures that on the same data:
  - Readers do not wait for writers or other readers
  - Writers do not wait for readers

ORACLE
甲骨文

Implementation of Read Consistency



Locking

- The Oracle Server locks:
  - Prevent destructive interaction between concurrent transactions
  - Require no user action
  - Automatically use the lowest level of restrictiveness
  - Are held for the duration of the transaction
  - Have two basic modes:
    - Exclusive
    - Share

Locking Modes

Lock Mode	Description
Exclusive lock	Prevents a resource from being shared. The first transaction to lock a resource exclusively is the only transaction that can alter the resource until the exclusive lock is released.
Share	Allows the resource to be shared. Multiple users reading data can share the data, holding share locks to prevent concurrent access by a writer (who needs an exclusive lock). Several transactions can acquire share locks on the same resource.

Implicit Locking

User Action	Row-Level Lock	Table-Level Lock
SELECT ... FROM *table* ...	None	None
INSERT INTO *table* ...	X	RX
UPDATE *table* ...	X	RX
DELETE FROM *table* ...	X	RX
DDL Operation	None	X

Explicit Locking

User Action	Row-Level lock	Table-Level lock
SELECT FOR UPDATE	X	RS [NOWAIT]
LOCK TABLE IN option	None	Depends on the MODE restrictiveness used

- Override the default lock mechanism:
  - For a consistent view of data when reading across multiple tables
  - When a transaction may change data based on other data that must not change until the whole transaction is complete

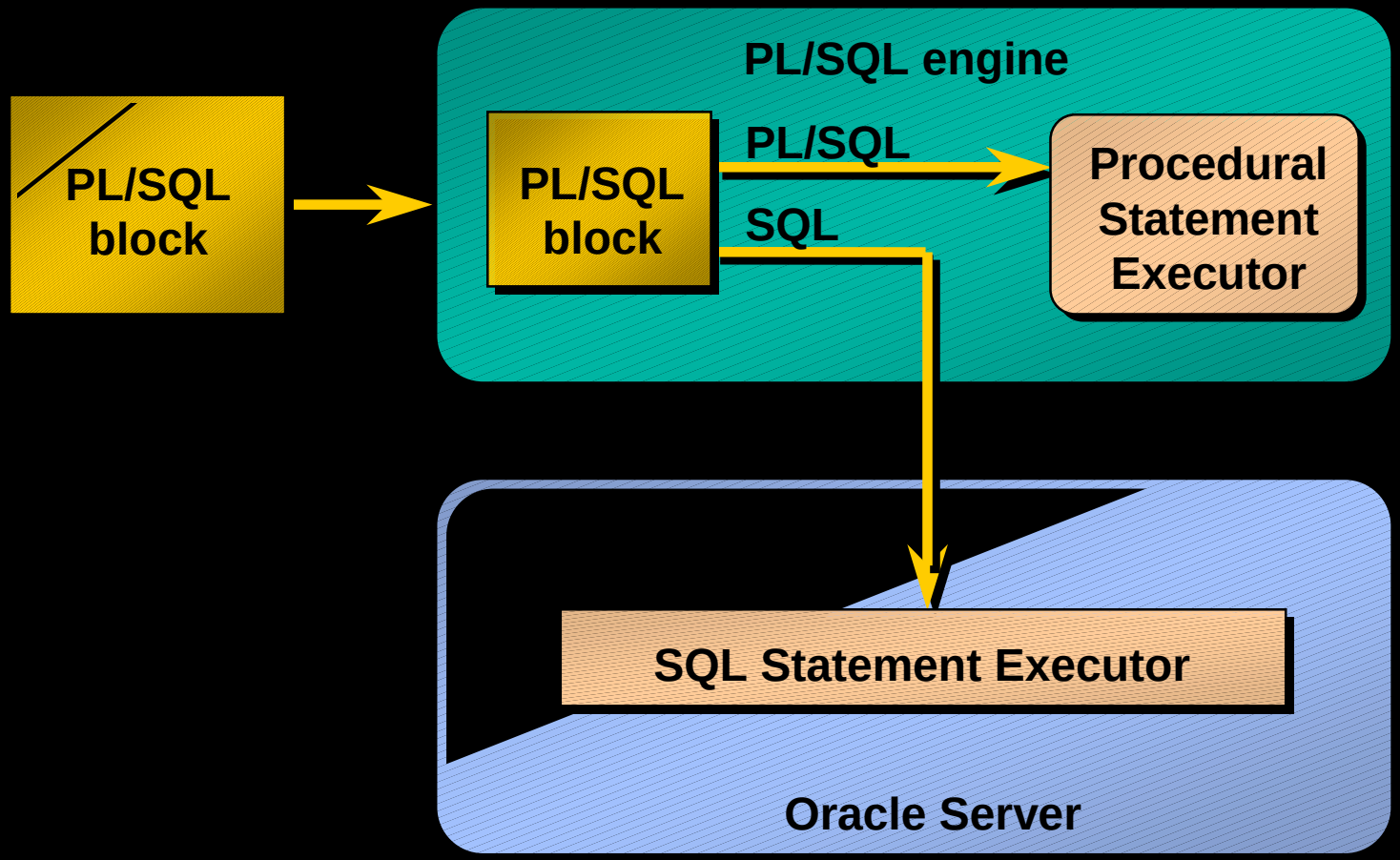
ORACLE
申昌文

Overview of PL/SQL

About PL/SQL

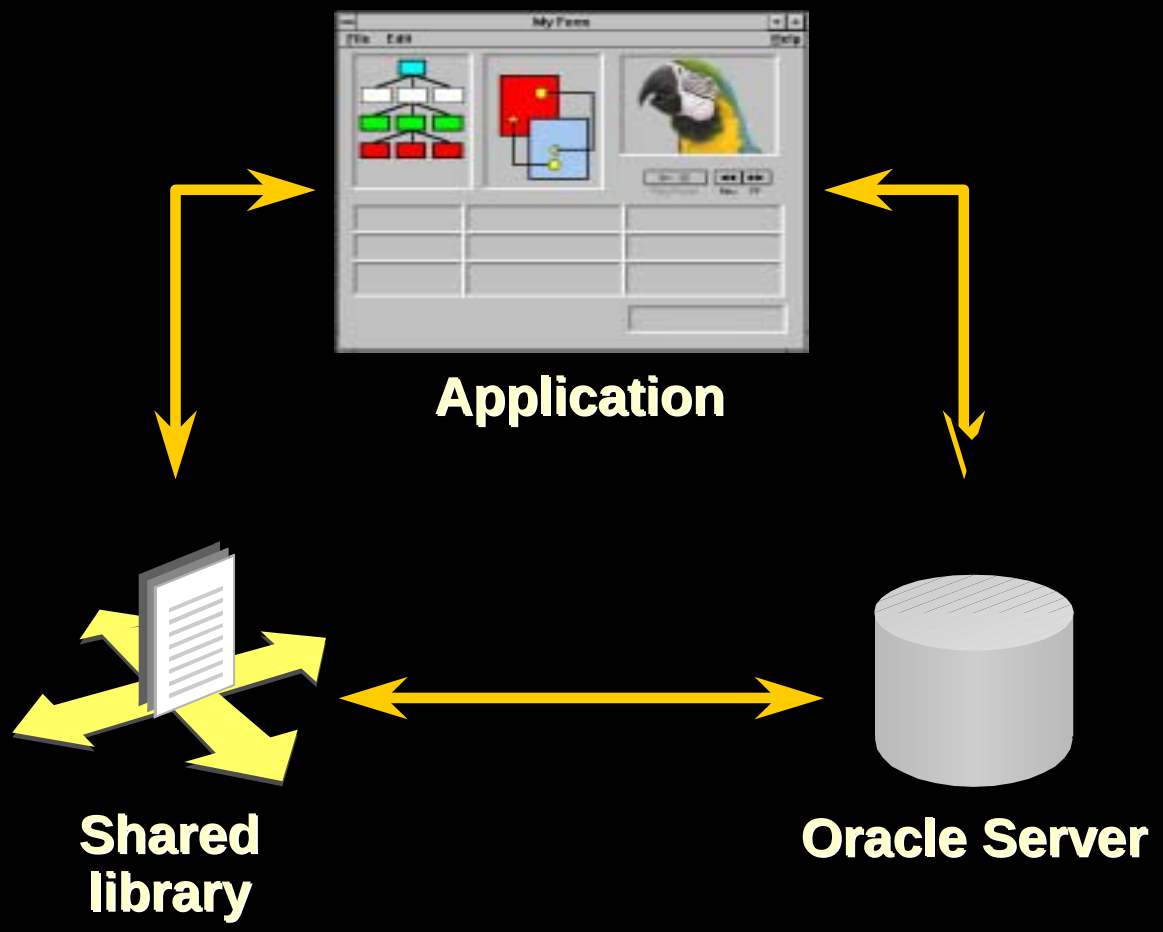
- PL/SQL is an extension to SQL with design features of programming languages.
- Data manipulation and query statements of SQL are included within procedural units of code.

PL/SQL Environment



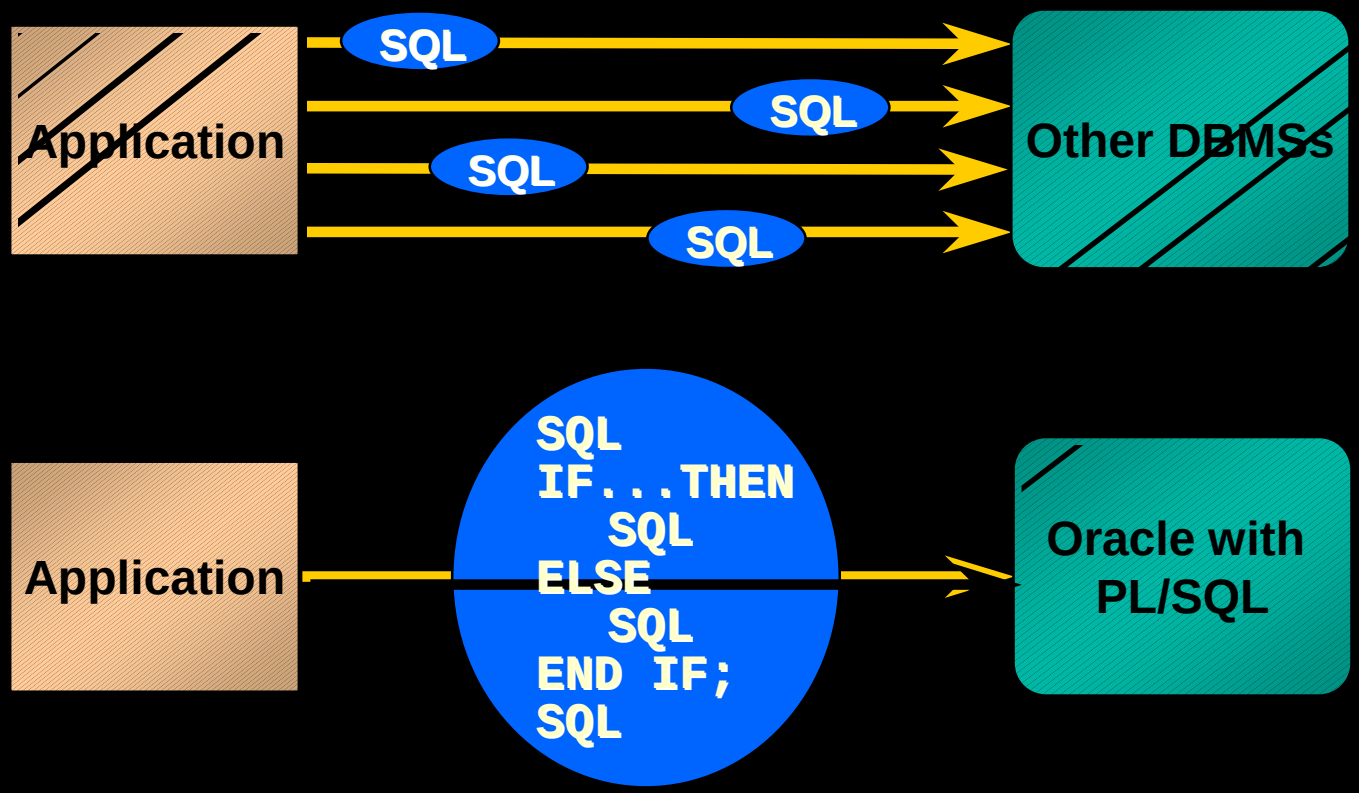
Benefits of PL/SQL

- Integration



Benefits of PL/SQL

- Improve Performance



Benefits of PL/SQL

- Modularize program development

```
DECLARE
  • • •
BEGIN
  • • •
EXCEPTION
  • • •
END;
```


ORACLE
申昌文

Benefits of PL/SQL

- It is portable.
- You can declare identifiers.

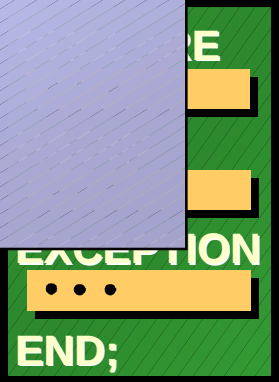
Benefits of PL/SQL

- You can program with procedural language control structures.
- It can handle errors.

Declaring Variables

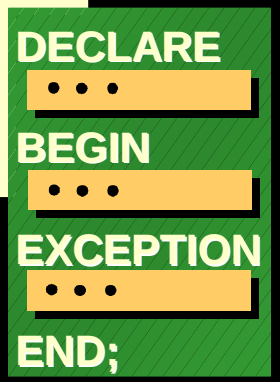
PL/SQL Block Structure

- DECLARE – Optional
  - Variables, cursors, user-defined exceptions
- BEGIN – Mandatory
  - SQL statements
  - PL/SQL statements
- EXCEPTION – Optional
  - Actions to perform when errors occur
- END; – Mandatory



PL/SQL Block Structure

```
DECLARE
  v_variable  VARCHAR2(5);
BEGIN
  SELECT      column_name
    INTO      v_variable
    FROM      table_name;
EXCEPTION
  WHEN exception_name THEN
  ...
END;
```



Block Types

- Anonymous

```
[DECLARE]


BEGIN
  --statements

[EXCEPTION]

END;
```

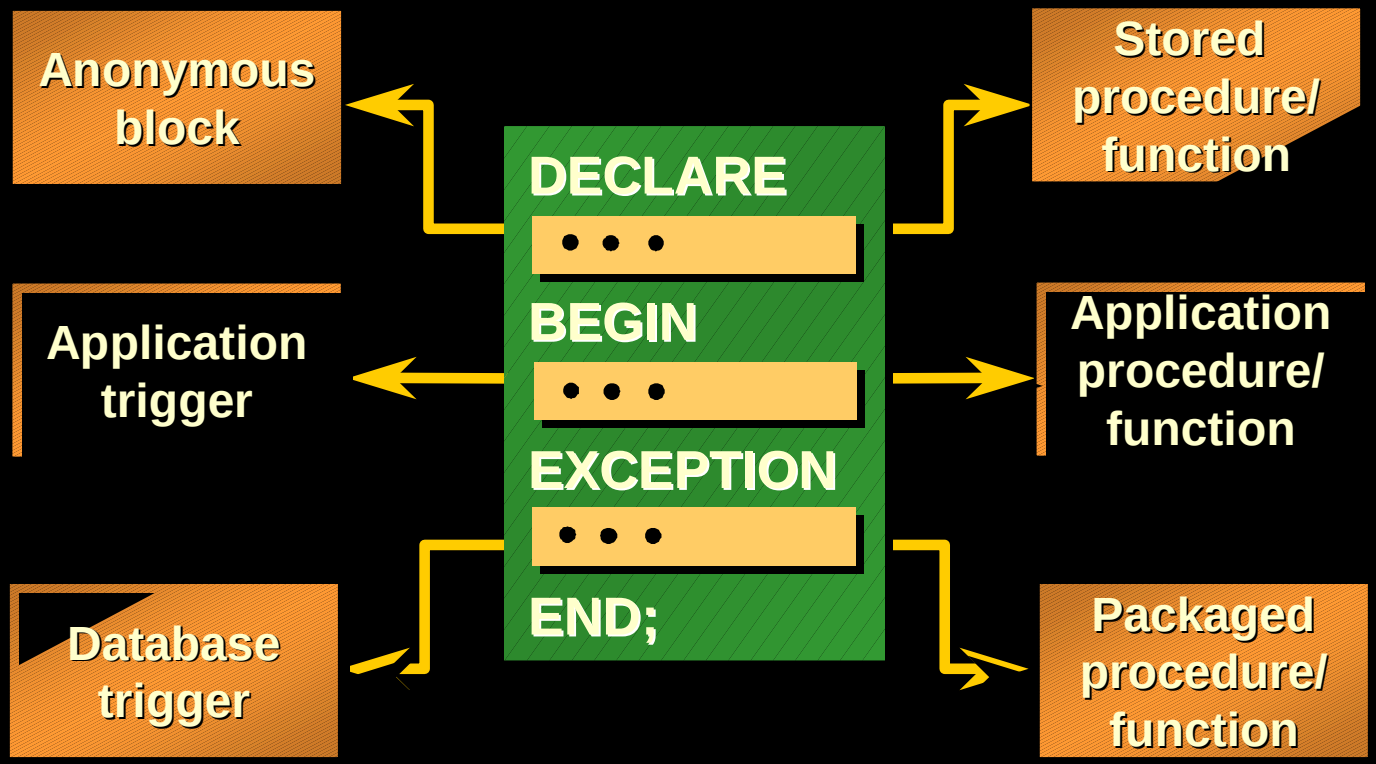
Procedure

```
PROCEDURE name
IS

BEGIN
  --statements

[EXCEPTION]

END;
```

Function

```
FUNCTION name
RETURN datatype
IS
BEGIN
  --statements
  RETURN value;
[EXCEPTION]

END;
```

Program Constructs



Use of Variables

- Use variables for:
  - Temporary storage of data
  - Manipulation of stored values
  - Reusability
  - Ease of maintenance

Handling Variables in PL/SQL

- Declare and initialize variables in the declaration section.
- Assign new values to variables in the executable section.
- Pass values into PL/SQL blocks through parameters.
- View results through output variables.

Types of Variables

- PL/SQL variables:
  - Scalar
  - Composite
  - Reference
  - LOB (large objects)
- Non-PL/SQL variables: Bind and host variables

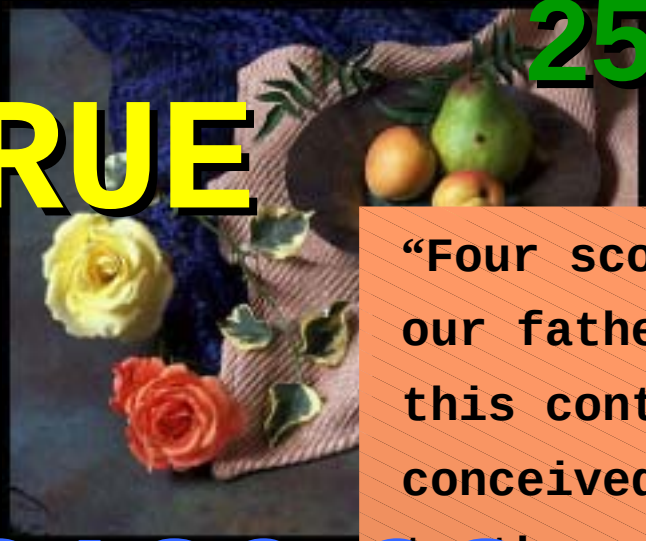
Types of Variables

- PL/SQL variables:
  - Scalar
  - Composite
  - Reference
  - LOB (large objects)
- Non-PL/SQL variables: Bind and host variables

Types of Variables

25-OCT-99

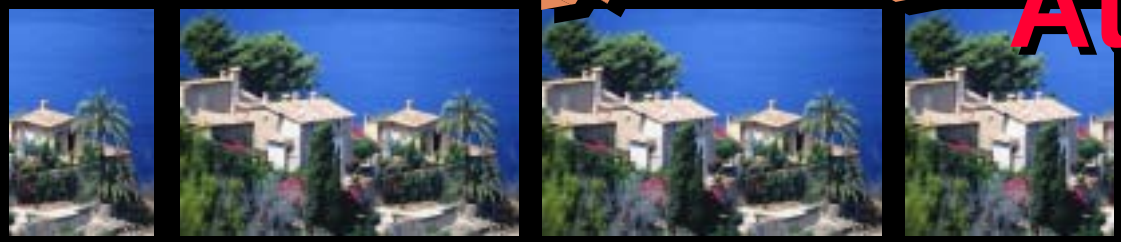
TRUE



"Four score and seven years ago our fathers brought forth upon this continent, a new nation, conceived in LIBERTY, and dedicated to the proposition that all men are created equal."

256120.08

Atlanta



Declaring PL/SQL Variables

Syntax

```
identifier [CONSTANT] datatype [NOT NULL]
     [:= | DEFAULT expr];
```

Examples

```
Declare
  v_hiredate      DATE;
  v_deptno        NUMBER(2) NOT NULL := 10;
  v_location      VARCHAR2(13) := 'Atlanta';
  c_comm          CONSTANT NUMBER := 1400;
```

Declaring PL/SQL Variables

- Guidelines
  - Follow naming conventions.
  - Initialize variables designated as NOT NULL.
  - Initialize identifiers by using the assignment operator (:=) or the DEFAULT reserved word.
  - Declare at most one identifier per line.

Naming Rules

- Two variables can have the same name, provided they are in different blocks.
- The variable name (identifier) should not be the same as the name of table columns used in the block.

```
DECLARE
  empno  NUMBER(4);
BEGIN
  SELECT    empno
  INTO      empno
  FROM      emp
  WHERE     ename = 'SMITH';
END;
```

Adopt a naming convention for PL/SQL identifiers: for example, v_empno

Assigning Values to Variables

Syntax

```
•identifier := expr;
```

Example

- Set a predefined hiredate for new employees.

```
v_hiredate := '31-DEC-98';
```

- Set the employee name to “Maduro.”

```
v_ename := 'Maduro';
```


ORACLE
甲骨文

Variable Initialization and Keywords

- Using:
  - Assignment operator (:=)
  - DEFAULT keyword
  - NOT NULL constraint

Scalar Datatypes

- Hold a single value
- Have no internal components

25-OCT-99

"Four score and seven years
ago our fathers brought
forth upon this continent, a
new nation, conceived in
LIBERTY, and dedicated to
the proposition that all men
are created equal."

TRUE

256120.08

Atlanta

Base Scalar Datatypes

- VARCHAR2 (*maximum_length*)
- NUMBER [(*precision, scale*)]
- DATE
- CHAR [(*maximum_length*)]
- LONG
- LONG RAW
- BOOLEAN
- BINARY_INTEGER
- PLS_INTEGER

Base Scalar Datatypes

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIMEZHONE
- TIMESTAMP WITH LOCAL TIME ZONE
- INTERVAL YEAR TO MONTH
- INVERTAL YEAR TO SECOND

Scalar Variable Declarations

- Example

```
v_job          VARCHAR2(9);
v_count        BINARY_INTEGER := 0;
v_total_sal    NUMBER(9,2) := 0;
v_orderdate    DATE := SYSDATE + 7;
c_tax_rate     CONSTANT NUMBER(3,2) := 8.25;
v_valid        BOOLEAN NOT NULL := TRUE;
```

The %TYPE Attribute

- Declare a variable according to:
  - A database column definition
  - Another previously declared variable
- Prefix %TYPE with:
  - The database table and column
  - The previously declared variable name

Declaring Variables with the %TYPE Attribute

- Example

```
...
  v_ename                emp.ename%TYPE;
  v_balance              NUMBER(7,2);
  v_min_balance          v_balance%TYPE := 10;
...
```

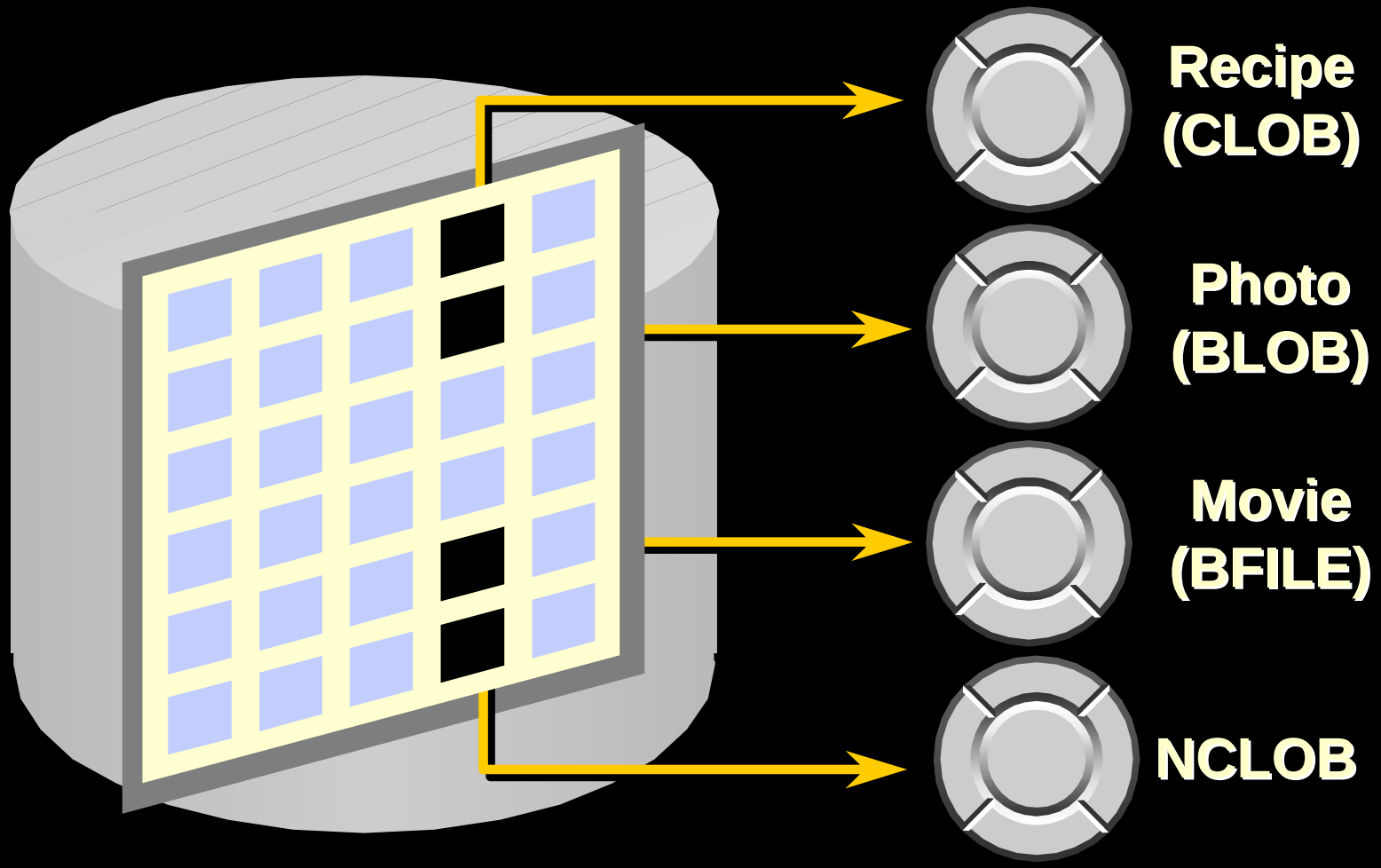
Declaring Boolean Variables

- Only the values TRUE, FALSE, and NULL can be assigned to a Boolean variable.
- The variables are connected by the logical operators AND, OR, and NOT.
- The variables always yield TRUE, FALSE, or NULL.
- Arithmetic, character, and date expressions can be used to return a Boolean value.

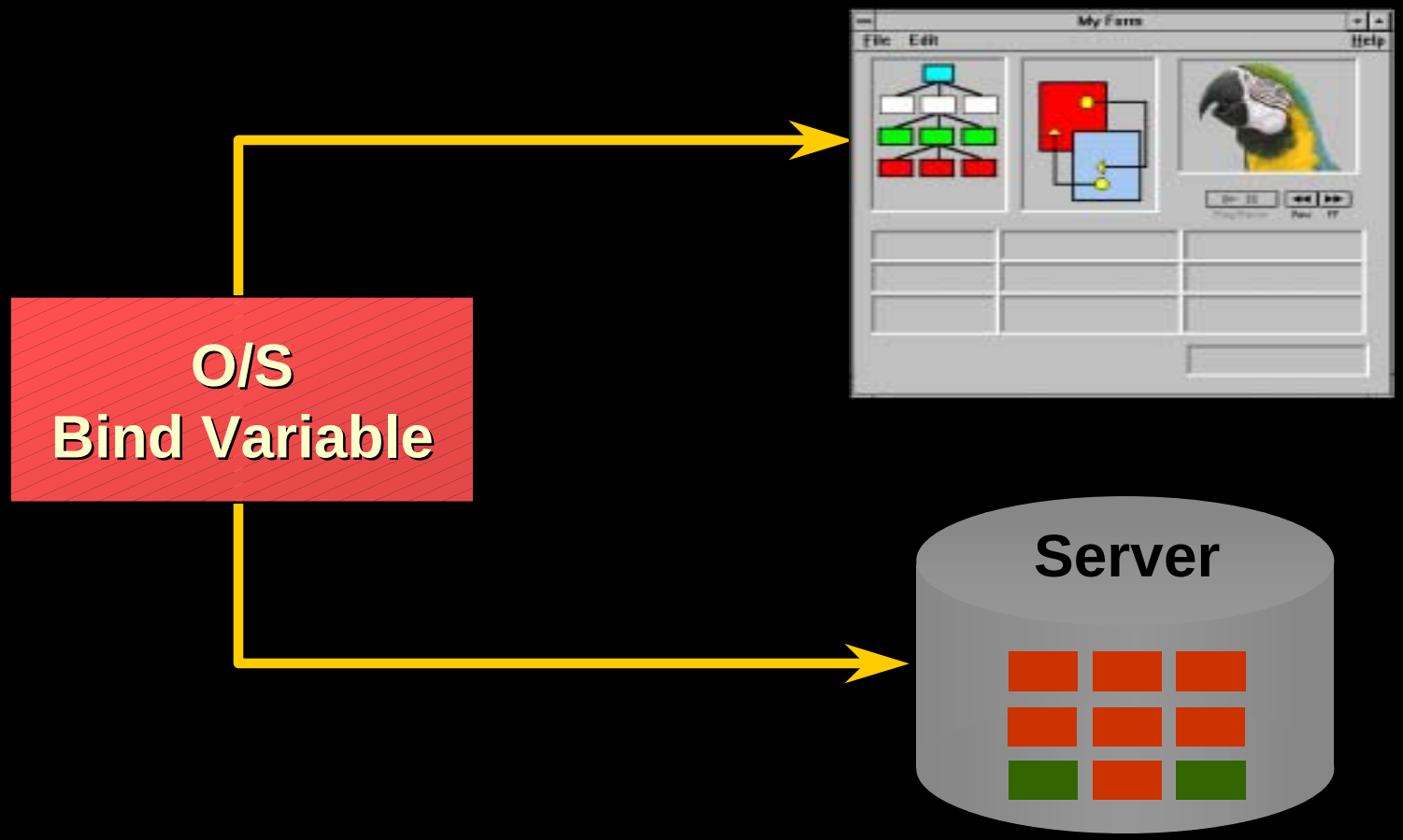
Composite Datatypes

- PL/SQL TABLES
- PL/SQL RECORDS

LOB Datatype Variables



Bind Variables



Referencing Non-PL/SQL Variables

- Store the annual salary into a SQL*Plus host variable.

```
:g_monthly_sal := v_sal / 12;
```

  - Reference non-PL/SQL variables as host variables.
  - Prefix the references with a colon (:).

Using Bind Variables

To reference a bind variable in PL/SQL, you must prefix its name with a colon (:).

Example:

```
VARIABLE      g_salary NUMBER
BEGIN
   SELECT     salary
   INTO        :g_salary
   FROM       employees
   WHERE      employee_id = 178;
END;
/
PRINT g_salary
```

DBMS_OUTPUT.PUT_LINE

- An Oracle-supplied packaged procedure
- An alternative for displaying data from a PL/SQL block
- Must be enabled in SQL*Plus with
  SET SERVEROUTPUT ON

Writing Executable Statements

PL/SQL Block Syntax and Guidelines

- Statements can continue over several lines.
- Lexical units can be separated by:
  - Spaces
  - Delimiters
  - Identifiers
  - Literals
  - Comments

PL/SQL Block Syntax and Guidelines

- Identifiers
  - Can contain up to 30 characters
  - Cannot contain reserved words unless enclosed in double quotation marks
  - Must begin with an alphabetic character
  - Should not have the same name as a database table column name

PL/SQL Block Syntax and Guidelines

- Literals
  - Character and date literals must be enclosed in single quotation marks.

```
v_ename := 'Henderson';
```

Commenting Code

- Prefix single-line comments with two dashes (--).
- Place multi-line comments between the symbols /* and */.

- Example

```
...
  v_sal NUMBER (9,2);
BEGIN
  /* Compute the annual salary based on the
     monthly salary input from the user */
  v_sal := &p_monthly_sal * 12;
END; -- This is the end of the transaction
```

SQL Functions in PL/SQL

- Available:
  - Single-row number
  - Single-row character
  - Datatype conversion
  - Date
- Not available:
  - DECODE
  - Group functions

} **Same as in SQL**

ORACLE
中国文

PL/SQL Functions

- Example
  - Build the mailing list for a company.

```
v_mailing_address := v_name||CHR(10)||
                     v_address||CHR(10)||v_state||
                     CHR(10)||v_zip;
```

  - Convert the employee name to lowercase.

```
v_ename          := LOWER(v_ename);
```

Datatype Conversion

- Convert data to comparable datatypes.
- Mixed datatypes can result in an error and affect performance.
- Conversion functions:
  - TO_CHAR
  - TO_DATE
  - TO_NUMBER

```
DECLARE
   v_date VARCHAR2(15);
BEGIN
   SELECT TO_CHAR(hiredate,
           'MON. DD, YYYY')
   INTO   v_date
   FROM   emp
   WHERE  empno = 7839;
END;
```

Datatype Conversion

This statement produces a compilation error if the variable v_date is declared as datatype DATE.

```
v_date := 'January 13, 1998';
```

To correct the error, use the TO_DATE conversion function.

```
v_date := TO_DATE ('January 13, 1998',
                   'Month DD, YYYY');
```

Nested Blocks and Variable Scope

- Statements can be nested wherever an executable statement is allowed.
- A nested block becomes a statement.
- An exception section can contain nested blocks.
- The scope of an object is the region of the program that can refer to the object.

Nested Blocks and Variable Scope

- An identifier is visible in the regions in which you can reference the unqualified identifier:
  - A block can look up to the enclosing block.
  - A block cannot look down to enclosed blocks.

Nested Blocks and Variable Scope

Example

```
...
  x  BINARY_INTEGER;
BEGIN
  ...
  DECLARE
    y  NUMBER;
  BEGIN
    ...
  END;
  ...
END;
```

Scope of *x*

Scope of *y*

Operators in PL/SQL

- Logical
- Arithmetic
- Concatenation
- Parentheses to control order of operations
- Exponential operator (**)

Same as in SQL

Operators in PL/SQL

- Example
  - Increment the index for a loop.

```
v_count        := v_count + 1;
```

  - Set the value of a Boolean flag.

```
v_equal        := (v_n1 = v_n2);
```

```
v_valid        := (v_empno IS NOT NULL);
```

Using Bind Variables

- To reference a bind variable in PL/SQL, you must prefix its name with a colon (:).
- Example

```
VARIABLE g_salary NUMBER
DECLARE
   v_sal      emp.sal%TYPE;
BEGIN
   SELECT     sal
   INTO       v_sal
   FROM       emp
   WHERE      empno = 7369;
  :g_salary          := v_sal;
END;
/
```

Programming Guidelines

- Make code maintenance easier by:
  - Documenting code with comments
  - Developing a case convention for the code
  - Developing naming conventions for identifiers and other objects
  - Enhancing readability by indenting

Code Naming Conventions

- Avoid ambiguity:
  - The names of local variables and formal parameters take precedence over the names of database tables.
  - The names of columns take precedence over the names of local variables.

Indenting Code

- For clarity, indent each level of code.
- Example

```
BEGIN
  IF x=0 THEN
     y:=1;
  END IF;
END;
```

```
DECLARE
  v_deptno    NUMBER(2);
  v_location     VARCHAR2(13);
BEGIN
  SELECT  deptno,
          loc
  INTO    v_deptno,
          v_location
  FROM    dept
  WHERE   dname = 'SALES';
...
END;
```

Determining Variable Scope

- Class Exercise

```
...
DECLARE
V_SAL           NUMBER(7,2) := 60000;
V_COMM          NUMBER(7,2) := V_SAL * .20;
V_MESSAGE       VARCHAR2(255) := ' eligible for commission';
BEGIN ...
```

```
DECLARE
  V_SAL              NUMBER(7,2) := 50000;
  V_COMM             NUMBER(7,2) := 0;
  V_TOTAL_COMP       NUMBER(7,2) := V_SAL + V_COMM;
BEGIN ...
  V_MESSAGE := 'CLERK not'||V_MESSAGE;
END;
```

```
    V_MESSAGE := 'SALESMAN'||V_MESSAGE;
END;
```

Writing Control Structures

Controlling PL/SQL Flow of Execution

- You can change the logical flow of statements using conditional IF statements and loop control structures.
- Conditional IF statements:
  - IF-THEN-END IF
  - IF-THEN-ELSE-END IF
  - IF-THEN-ELSIF-END IF

IF Statements

Syntax

```
IF condition THEN
  statements;
[ELSIF condition THEN
  statements;]
[ELSE
  statements;]
END IF;
```

Simple IF statement:

Set the manager ID to 22 if the employee name is Osborne.

```
IF v_ename = 'OSBORNE' THEN
  v_mgr := 22;
END IF;
```

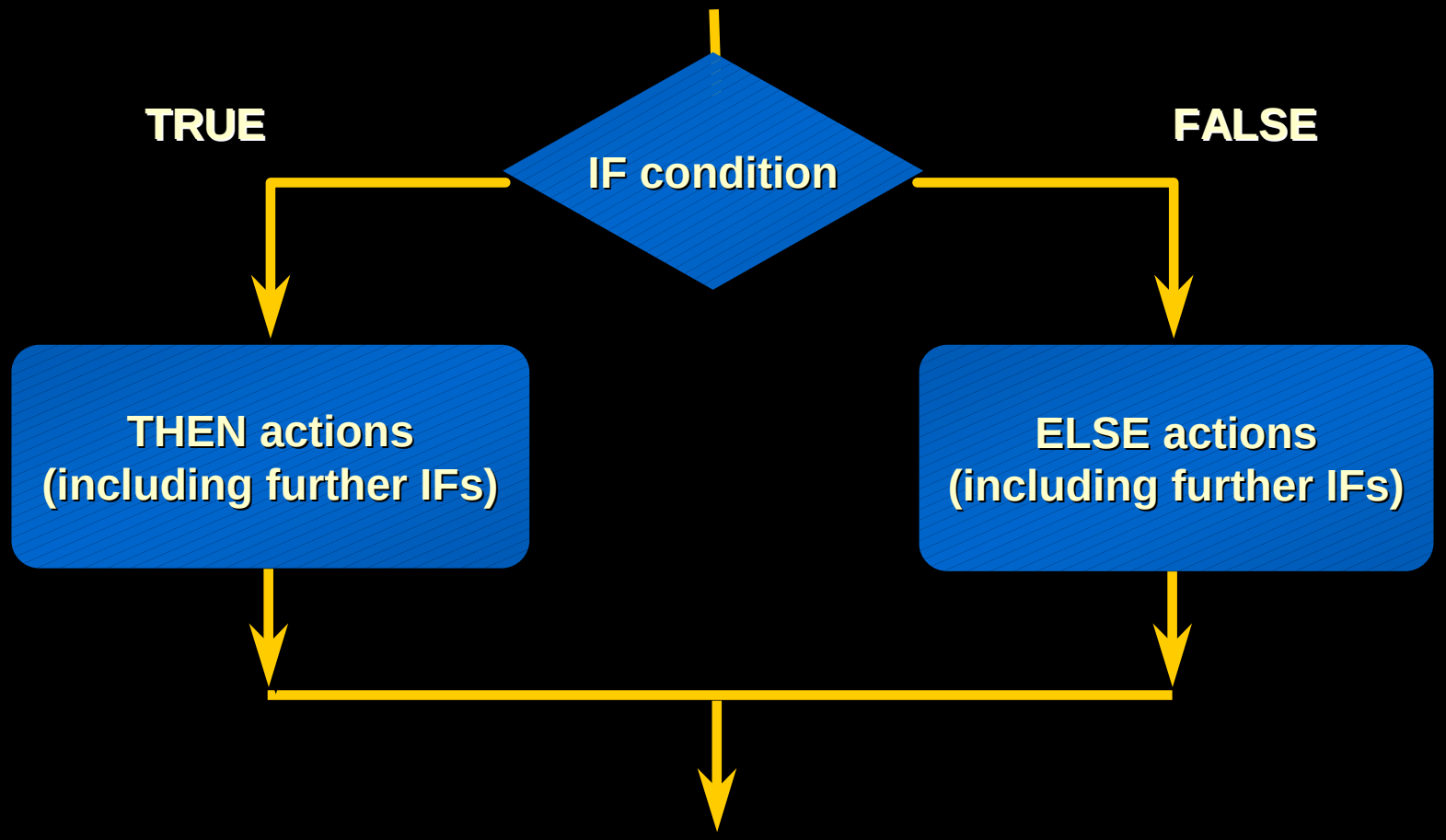

ORACLE
甲骨文

Simple IF Statements

- Set the job title to Salesman, the department number to 35, and the commission to 20% of the current salary if the last name is Miller.
- Example

```
. . .
IF v_ename = 'MILLER' THEN
  v_job := 'SALESMAN';
  v_deptno := 35;
  v_new_comm := sal * 0.20;
END IF;
. . .
```

IF-THEN-ELSE Statement Execution Flow

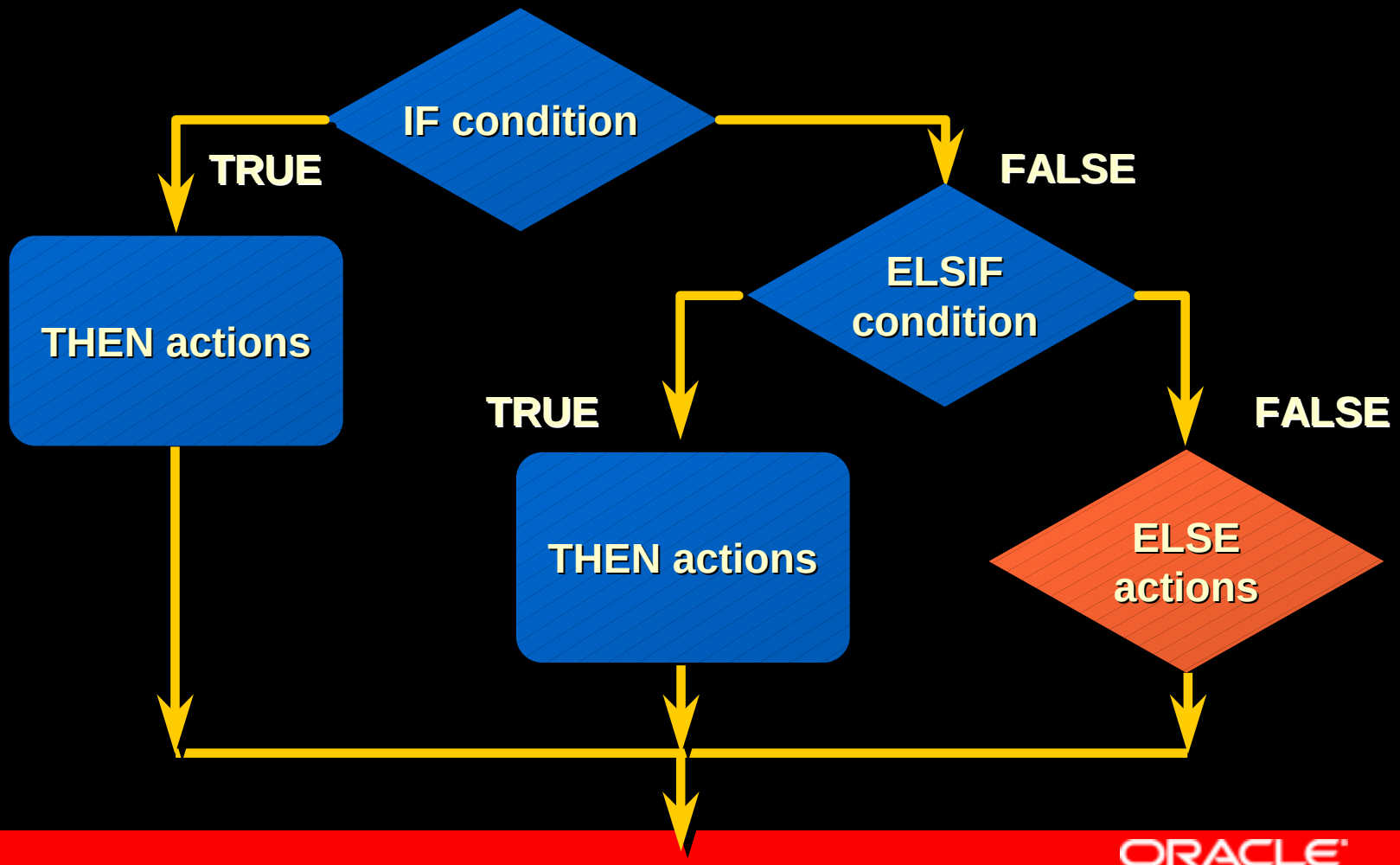


IF-THEN-ELSE Statements

- Set a flag for orders where there are fewer than five days between order date and ship date.
- Example

```
...
IF v_shipdate - v_orderdate < 5 THEN
  v_ship_flag := 'Acceptable';
ELSE
  v_ship_flag := 'Unacceptable';
END IF;
...
```

IF-THEN-ELSIF Statement Execution Flow



IF-THEN-ELSIF Statements

- For a given value, calculate a percentage of that value based on a condition.
- Example

```
. . .
IF v_start > 100 THEN
  v_start := 2 * v_start;
ELSIF v_start >= 50 THEN
  v_start := .5 * v_start;
ELSE
  v_start := .1 * v_start;
END IF;
. . .
```

Building Logical Conditions

- You can handle null values with the IS NULL operator.
- Any arithmetic expression containing a null value evaluates to NULL.
- Concatenated expressions with null values treat null values as an empty string.

Logic Tables

- Build a simple Boolean condition with a comparison operator.

AND	*TRUE*	*FALSE*	*NULL*
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

OR	*TRUE*	*FALSE*	*NULL*
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

NOT	
TRUE	FALSE
FALSE	TRUE
NULL	NULL

Boolean Conditions

- What is the value of V_FLAG in each case?

```
v_flag := v_reorder_flag AND v_available_flag;
```

V_REORDER_FLAG	V_AVAILABLE_FLAG	V_FLAG
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
NULL	TRUE	NULL
NULL	FALSE	FALSE

Iterative Control: LOOP Statements

- Loops repeat a statement or sequence of statements multiple times.
- There are three loop types:
  - Basic loop
  - FOR loop
  - WHILE loop

Basic Loop

- Syntax

```
LOOP                                    -- delimiter
  statement1;                           -- statements
  . . .
  EXIT [WHEN condition];                -- EXIT statement
END LOOP;                               -- delimiter
```

```
where:   condition          is a Boolean variable or
                            expression (TRUE, FALSE,
                            or NULL);
```

Basic Loop

- Example

```
DECLARE
  v_ordid     item.ordid%TYPE := 601;
  v_counter   NUMBER(2) := 1;
BEGIN
  LOOP
    INSERT INTO item(ordid, itemid)
      VALUES(v_ordid, v_counter);
    v_counter := v_counter + 1;
    EXIT WHEN v_counter > 10;
  END LOOP;
END;
```

FOR Loop

```
FOR counter in [REVERSE]
    lower_bound..upper_bound LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

- Use a FOR loop to shortcut the test for the number of iterations.
- Do not declare the index; it is declared implicitly.

FOR Loop

- Guidelines
  - Reference the counter within the loop only; it is undefined outside the loop.
  - Use an expression to reference the existing value of a counter.
  - Do *not* reference the counter as the target of an assignment.

FOR Loop

- Insert the first 10 new line items for order number 601.
- Example

```
DECLARE
  v_ordid     item.ordid%TYPE := 601;
BEGIN
  FOR i IN 1..10 LOOP
    INSERT INTO item(ordid, itemid)
      VALUES(v_ordid, i);
  END LOOP;
END;
```

WHILE Loop

- Syntax

```
WHILE condition LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

Condition is evaluated at the beginning of each iteration.

- Use the WHILE loop to repeat statements while a condition is TRUE.

WHILE Loop

- Example

```
ACCEPT p_new_order PROMPT 'Enter the order number: '
ACCEPT p_items -
  PROMPT 'Enter the number of items in this order: '
DECLARE
v_count     NUMBER(2) := 1;
BEGIN
  WHILE v_count <= &p_items LOOP
    INSERT INTO item (ordid, itemid)
    VALUES (&p_new_order, v_count);
    v_count := v_count + 1;
  END LOOP;
  COMMIT;
END;
/
```

Nested Loops and Labels

- Nest loops to multiple levels.
- Use labels to distinguish between blocks and loops.
- Exit the outer loop with the EXIT statement referencing the label.

Nested Loops and Labels

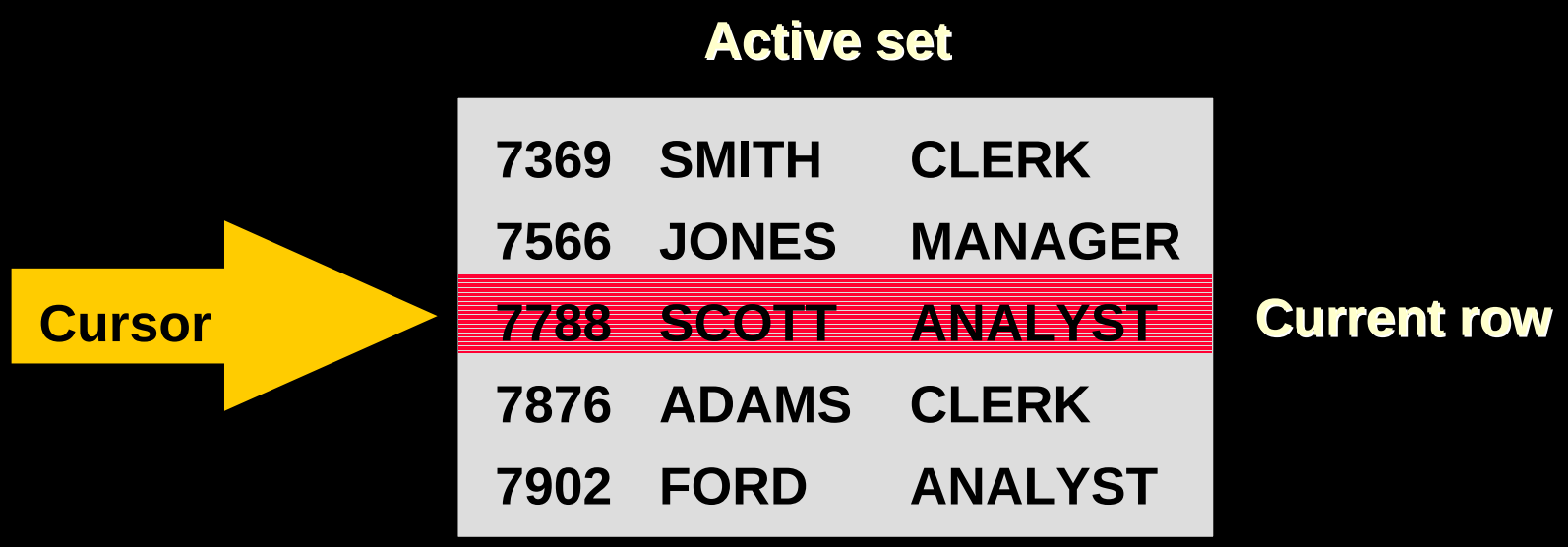
```
...
BEGIN
  <<Outer_loop>>
  LOOP
    v_counter := v_counter+1;
  EXIT WHEN v_counter>10;
    <<Inner_loop>>
    LOOP
      ...
      EXIT Outer_loop WHEN total_done = 'YES';
      -- Leave both loops
      EXIT WHEN inner_done = 'YES';
      -- Leave inner loop only
      ...
    END LOOP Inner_loop;
    ...
  END LOOP Outer_loop;
END;
```

Writing Explicit Cursors

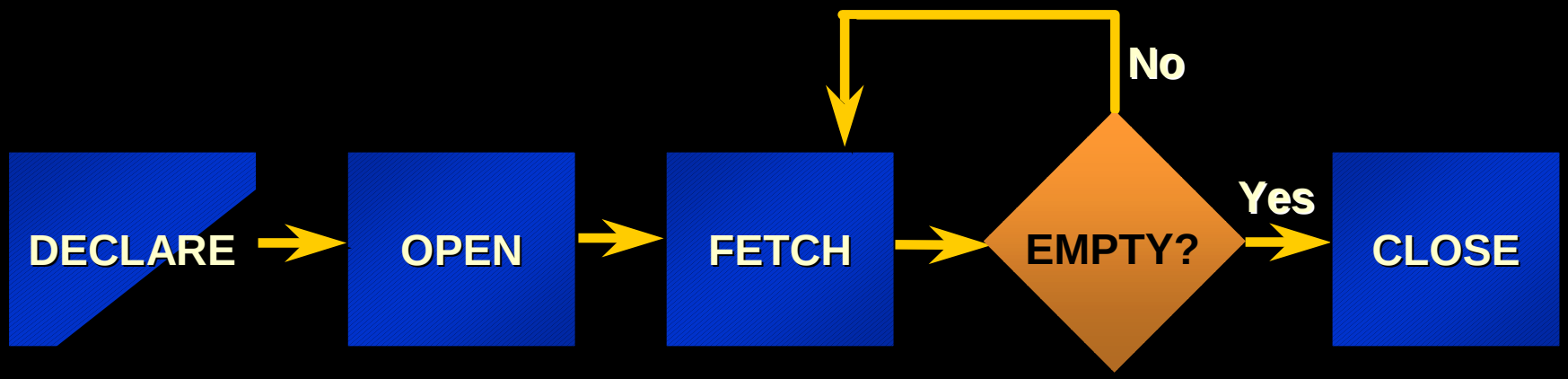
About Cursors

- Every SQL statement executed by the Oracle Server has an individual cursor associated with it:
  - Implicit cursors: Declared for all DML and PL/SQL SELECT statements
  - Explicit cursors: Declared and named by the programmer

Explicit Cursor Functions



Controlling Explicit Cursors



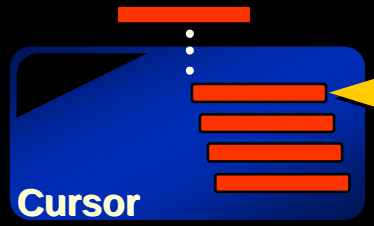
- Create a named SQL area
- Identify the active set
- Load the current row into variables
- Test for existing rows
- Return to FETCH if rows found
- Release the active set

Controlling Explicit Cursors

Open the cursor.

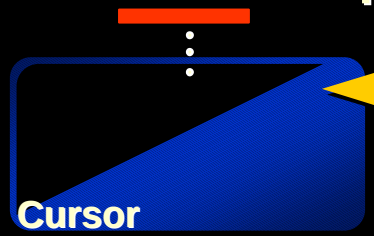
Pointer

Fetch a row from the cursor.



Pointer

Continue until empty.



Pointer

Close the cursor.

Declaring the Cursor

- Syntax

```
CURSOR cursor_name IS
      select_statement;
```

  - Do not include the INTO clause in the cursor declaration.
  - If processing rows in a specific sequence is required, use the ORDER BY clause in the query.

Declaring the Cursor

- Example

```
DECLARE
  CURSOR emp_cursor IS
    SELECT empno, ename
    FROM   emp;

  CURSOR dept_cursor IS
    SELECT *
    FROM   dept
    WHERE  deptno = 10;
BEGIN
  ...
```

Opening the Cursor

- Syntax

```
OPEN  cursor_name;
```

  - Open the cursor to execute the query and identify the active set.
  - If the query returns no rows, no exception is raised.
  - Use cursor attributes to test the outcome after a fetch.

Fetching Data from the Cursor

- Syntax

```
FETCH cursor_name INTO [variable1, variable2, ...]
                       | record_name];
```

  - Retrieve the current row values into output variables.
  - Include the same number of variables.
  - Match each variable to correspond to the columns positionally.
  - Test to see if the cursor contains rows.

Fetching Data from the Cursor

- Example

-
```
FETCH emp_cursor INTO v_empno, v_ename;
```

```
...
OPEN defined_cursor;
LOOP
  FETCH defined_cursor INTO defined_variables
  EXIT WHEN ...;
  ...
    -- Process the retrieved data
  ...
END;
```

Closing the Cursor

- Syntax

```
CLOSE     cursor_name;
```

  - Close the cursor after completing the processing of the rows.
  - Reopen the cursor, if required.
  - Do not attempt to fetch data from a cursor once it has been closed.

Explicit Cursor Attributes

- Obtain status information about a cursor.

Attribute	Type	Description
%ISOPEN	Boolean	Evaluates to TRUE if the cursor is open
%NOTFOUND	Boolean	Evaluates to TRUE if the most recent fetch does not return a row
%FOUND	Boolean	Evaluates to TRUE if the most recent fetch returns a row; complement of %NOTFOUND
%ROWCOUNT	Number	Evaluates to the total number of rows returned so far

ORACLE
申骨文

The %ISOPEN Attribute

- Fetch rows only when the cursor is open.
- Use the %ISOPEN cursor attribute before performing a fetch to test whether the cursor is open.

- Example

```
IF NOT emp_cursor%ISOPEN THEN
    OPEN emp_cursor;
END IF;
LOOP
  FETCH emp_cursor...
```

Controlling Multiple Fetches

- Process several rows from an explicit cursor using a loop.
- Fetch a row with each iteration.
- Use the %NOTFOUND attribute to write a test for an unsuccessful fetch.
- Use explicit cursor attributes to test the success of each fetch.

The %NOTFOUND and %ROWCOUNT Attributes

- Use the %ROWCOUNT cursor attribute to retrieve an exact number of rows.
- Use the %NOTFOUND cursor attribute to determine when to exit the loop.

SQL Cursor Attributes

Delete rows that have the specified employee ID from the EMPLOYEES table. Print the number of rows deleted.

Example:

```
VARIABLE rows_deleted VARCHAR2(30)
DECLARE
  v_employee_id employees.employee_id%TYPE := 176;
BEGIN
  DELETE FROM  employees
  WHERE        employee_id = v_employee_id;
  :rows_deleted := (SQL%ROWCOUNT ||
                        ' row deleted.');
END;
/
PRINT rows_deleted
```

Cursors and Records

- Process the rows of the active set conveniently by fetching values into a PL/SQL RECORD.
- Example

```
DECLARE
  CURSOR emp_cursor IS
    SELECT  empno, ename
    FROM    emp;
  emp_record   emp_cursor%ROWTYPE;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO emp_record;
  ...
```

Cursor FOR Loops

- Syntax

```
FOR record_name IN cursor_name LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

  cursors.
  - Implicit open, fetch, and close occur.
  - The record is implicitly declared.

Cursor FOR Loops

- Retrieve employees one by one until no more are left.
- Example

```
DECLARE
  CURSOR emp_cursor IS
    SELECT ename, deptno
    FROM   emp;
BEGIN
  FOR emp_record IN emp_cursor LOOP
          -- implicit open and implicit fetch occur
    IF emp_record.deptno = 30 THEN
      ...
  END LOOP; -- implicit close occurs
END;
```

Cursor FOR Loops Using Subqueries

- No need to declare the cursor.
- Example

```
BEGIN
  FOR emp_record IN ( SELECT ename, deptno
                      FROM   emp) LOOP
         -- implicit open and implicit fetch occur
    IF emp_record.deptno = 30 THEN
      ...
  END LOOP; -- implicit close occurs
END;
```

Advanced Explicit Cursor Concepts

Cursors with Parameters

- Syntax

```
CURSOR cursor_name
  [(parameter_name datatype, ...)]
IS
  select_statement;
```

  - Pass parameter values to a cursor when the cursor is opened and the query is executed.
  - Open an explicit cursor several times with a different active set each time.

ORACLE
申骨文

Cursors with Parameters

- Pass the department number and job title to the WHERE clause.
- Example

```
DECLARE
  CURSOR emp_cursor
  (v_deptno NUMBER, v_job VARCHAR2) IS
    SELECT    empno, ename
    FROM      emp
    WHERE     deptno = v_deptno
     AND      job = v_job;
BEGIN
  OPEN emp_cursor(10, 'CLERK');
...
```

The FOR UPDATE Clause

- Syntax

```
SELECT ...
FROM          ...
FOR UPDATE [OF column_reference][NOWAIT]
```

  - Explicit locking lets you deny access for the duration of a transaction.
  - Lock the rows *before* the update or delete.

The FOR UPDATE Clause

- Retrieve the employees who work in department 30.
- Example

```
DECLARE
  CURSOR emp_cursor IS
    SELECT empno, ename, sal
    FROM   emp
    WHERE  deptno = 30
    FOR UPDATE NOWAIT;
```

The WHERE CURRENT OF Clause

- Syntax

```
WHERE CURRENT OF cursor
```

  - Use cursors to update or delete the current row.
  - Include the FOR UPDATE clause in the cursor query to lock the rows first.
  - Use the WHERE CURRENT OF clause to reference the current row from an explicit cursor.

The WHERE CURRENT OF Clause

Example

```
DECLARE
  CURSOR sal_cursor IS
    SELECT     sal
    FROM       emp
    WHERE      deptno = 30
    FOR UPDATE NOWAIT;
BEGIN
  FOR emp_record IN sal_cursor LOOP
    UPDATE     emp
    SET        sal = emp_record.sal * 1.10
    WHERE CURRENT OF sal_cursor;
  END LOOP;
  COMMIT;
END;
```

Cursors with Subqueries

Example

```
DECLARE
  CURSOR my_cursor IS
    SELECT t1.deptno, dname, STAFF
    FROM   dept t1, (SELECT deptno,
                       count(*) STAFF
                     FROM   emp
                     GROUP BY deptno) t2
    WHERE  t1.deptno = t2.deptno
    AND    STAFF >= 5;
```

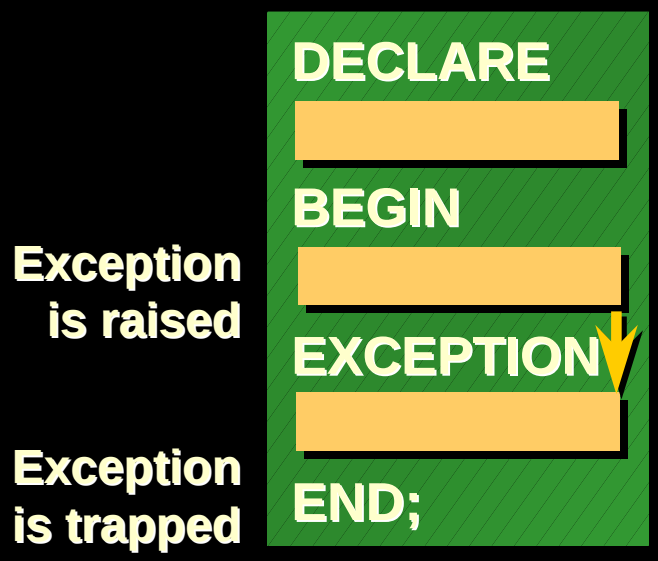
Handling Exceptions

Handling Exceptions with PL/SQL

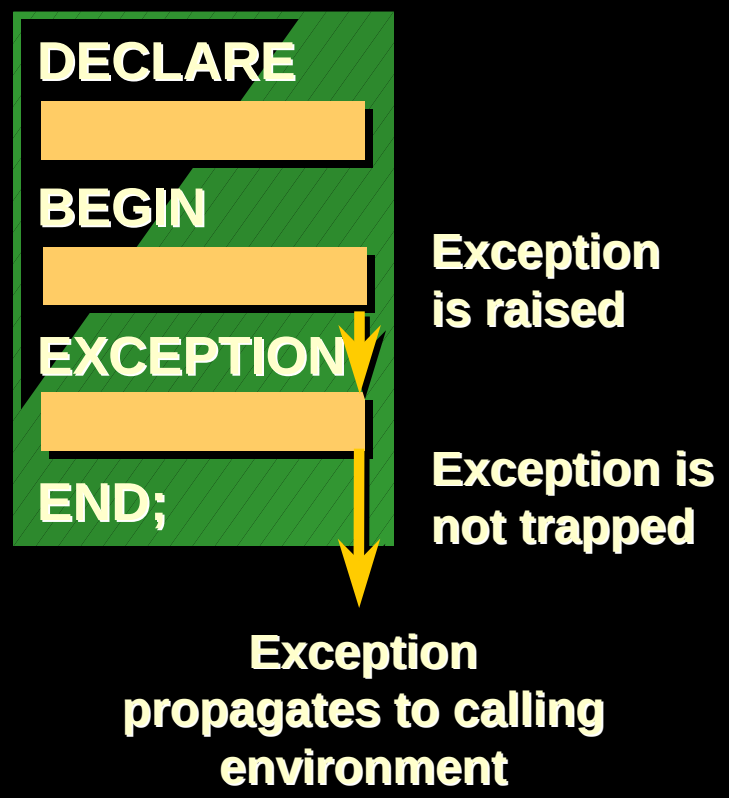
- What is an exception?
  - Identifier in PL/SQL that is raised during execution
- How is it raised?
  - An Oracle error occurs.
  - You raise it explicitly.
- How do you handle it?
  - Trap it with a handler.
  - Propagate it to the calling environment.

Handling Exceptions

- Trap the exception



Propagate the exception



Exception Types

- Predefined Oracle Server
- Non-predefined Oracle Server
- User-defined

} Implicitly raised

Explicitly raised

ORACLE
申骨文

Trapping Exceptions

- Syntax

```
EXCEPTION
  WHEN exception1 [OR exception2 . . .] THEN
    statement1;
    statement2;
    . . .
  [WHEN exception3 [OR exception4 . . .] THEN
    statement1;
    statement2;
    . . .]
  [WHEN OTHERS THEN
    statement1;
    statement2;
    . . .]
```

Trapping Exceptions Guidelines

- WHEN OTHERS is the last clause.
- EXCEPTION keyword starts exception-handling section.
- Several exception handlers are allowed.
- Only one handler is processed before leaving the block.

Trapping Predefined Oracle Server Errors

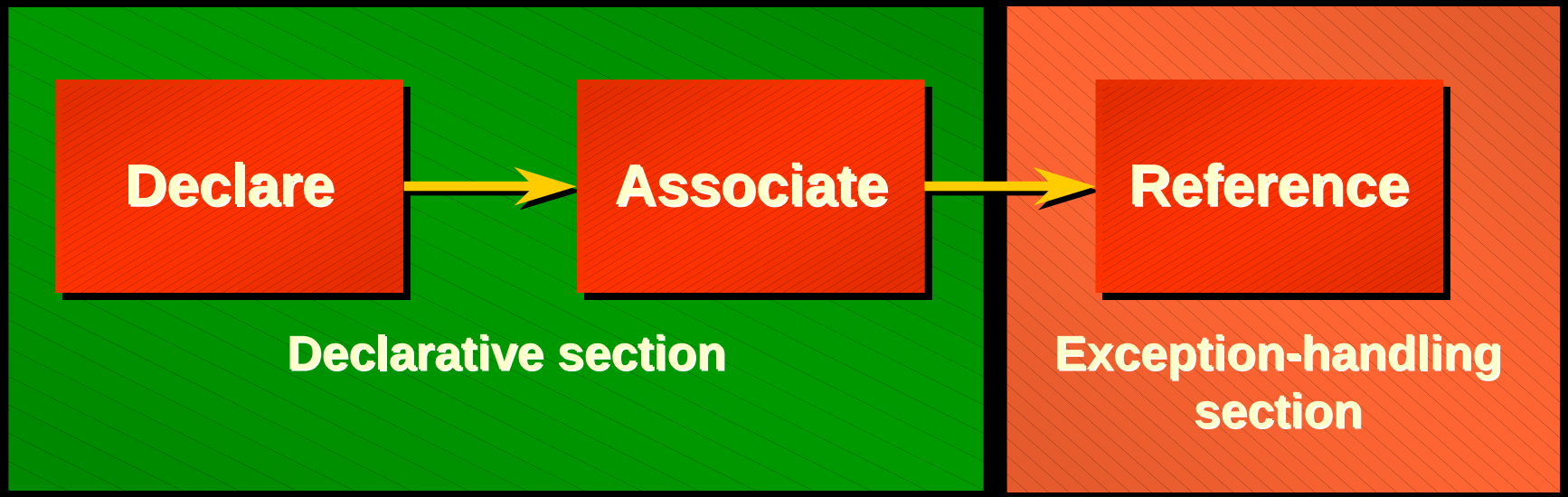
- Reference the standard name in the exception-handling routine.
- Sample predefined exceptions:
  - NO_DATA_FOUND
  - TOO_MANY_ROWS
  - INVALID_CURSOR
  - ZERO_DIVIDE
  - DUP_VAL_ON_INDEX

Predefined Exception

- Syntax

```
BEGIN  SELECT ... COMMIT;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    statement1;
    statement2;
  WHEN TOO_MANY_ROWS THEN
    statement1;
  WHEN OTHERS THEN
    statement1;
    statement2;
    statement3;
END;
```

Trapping Non-Predefined Oracle Server Errors



- Name the exception
- Code the PRAGMA EXCEPTION_INIT
- Handle the raised exception

Non-Predefined Error

- Trap for Oracle Server error number –2292, an integrity constraint violation.

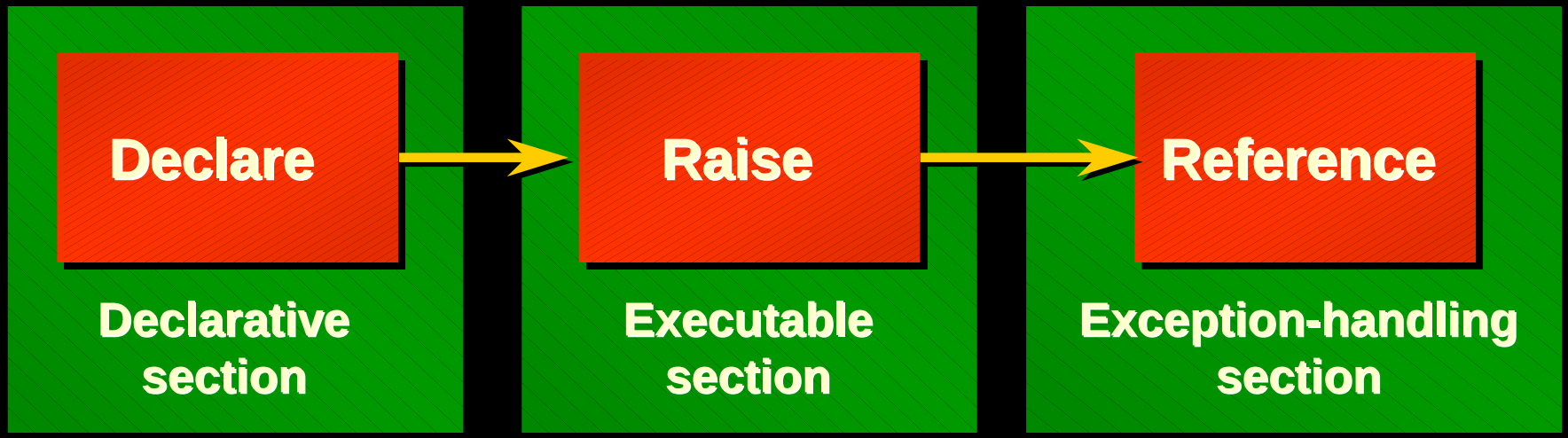
```
DECLARE
  e_emps_remaining    EXCEPTION;
  PRAGMA EXCEPTION_INIT (
             e_emps_remaining, -2292);
  v_deptno     dept.deptno%TYPE := &p_deptno;
BEGIN
  DELETE FROM dept
  WHERE        deptno = v_deptno;
  COMMIT;
EXCEPTION
  WHEN e_emps_remaining THEN
   DBMS_OUTPUT.PUT_LINE ('Cannot remove dept ' ||
   TO_CHAR(v_deptno) || '.  Employees exist. ');
END;
```

1

2

3

Trapping User-Defined Exceptions



- Name the exception
- Explicitly raise the exception by using the RAISE statement
- Handle the raised exception

User-Defined Exception

Example

```
DECLARE
  e_invalid_product  EXCEPTION;
BEGIN
  UPDATE      product
  SET         descrip = '&product_description'
  WHERE       prodid = &product_number;
  IF SQL%NOTFOUND THEN
    RAISE e_invalid_product;
  END IF;
  COMMIT;
EXCEPTION
  WHEN e_invalid_product THEN
    DBMS_OUTPUT.PUT_LINE('Invalid product number.');
END;
```

1

2

3

Functions for Trapping Exceptions

- SQLCODE
  Returns the numeric value for the error code
- SQLERRM
  Returns the message associated with the error number

Functions for Trapping Exceptions

- Example

```
DECLARE
  v_error_code      NUMBER;
  v_error_message   VARCHAR2(255);
BEGIN
...
EXCEPTION
...
  WHEN OTHERS THEN
    ROLLBACK;
    v_error_code := SQLCODE ;
    v_error_message := SQLERRM ;
    INSERT INTO errors VALUES(v_error_code,
                              v_error_message);
END;
```

Calling Environments

SQL*Plus	Displays error number and message to screen
Procedure Builder	Displays error number and message to screen
Oracle Developer Forms	Accesses error number and message in a trigger by means of the ERROR_CODE and ERROR_TEXT packaged functions
Precompiler application	Accesses exception number through the SQLCA data structure
An enclosing PL/SQL block	Traps exception in exception-handling routine of enclosing block

Propagating Exceptions

Subblocks can handle an exception or pass the exception to the enclosing block.

```
DECLARE
  . . .
  e_no_rows       exception;
  e_integrity     exception;
  PRAGMA EXCEPTION_INIT (e_integrity, -2292);
BEGIN
  FOR c_record IN emp_cursor LOOP
    BEGIN
      SELECT ...
      UPDATE ...
      IF SQL%NOTFOUND THEN
        RAISE e_no_rows;
      END IF;
    END;
END LOOP;
EXCEPTION
  WHEN e_integrity THEN ...
  WHEN e_no_rows THEN ...
END;
```

RAISE_APPLICATION_ERROR Procedure

- Syntax

```
raise_application_error (error_number,
                         message[, {TRUE | FALSE}]);
```

  - A procedure that lets you issue user-defined error messages from stored subprograms
  - Called only from an executing stored subprogram

RAISE_APPLICATION_ERROR Procedure

- Used in two different places:
  - Executable section
  - Exception section
- Returns error conditions to the user in a manner consistent with other Oracle Server errors

RAISE_APPLICATION_ERROR

Executable section:

```
BEGIN
...
  DELETE FROM employees
     WHERE  manager_id = v_mgr;
  IF SQL%NOTFOUND THEN
     RAISE_APPLICATION_ERROR(-20202,
       'This is not a valid manager');
  END IF;
   ...
```

Exception section:

```
...
   EXCEPTION
     WHEN NO_DATA_FOUND THEN
       RAISE_APPLICATION_ERROR (-20201,
         'Manager is not a valid employee.');
   END;
```

Procedure and Function

Overview of Subprograms

A subprogram:

- Is a named PL/SQL block that can accept parameters and be invoked from a calling environment
- Is of two types:
  - A procedure that performs an action
  - A function that computes a value
- Is based on standard PL/SQL block structure
- Provides modularity, reusability, extensibility, and maintainability
- Provides easy maintenance, improved data security and integrity, improved performance, and improved code clarity

Block Structure for Anonymous PL/SQL Blocks

`DECLARE` (optional)

Declares PL/SQL objects to be used within this block

`BEGIN` (mandatory)

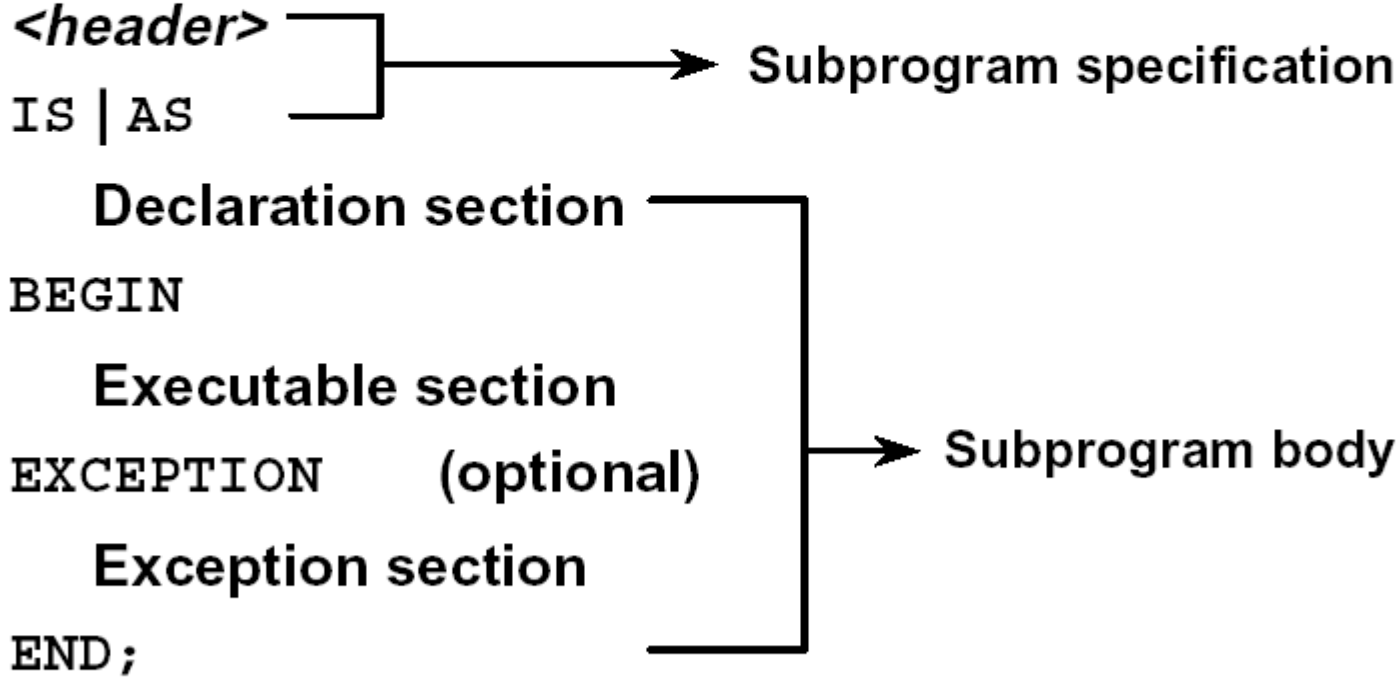
Defines the executable statements

`EXCEPTION` (optional)

Defines the actions that take place if an error or exception arises

`END;` (mandatory)

Block Structure for PL/SQL Subprograms

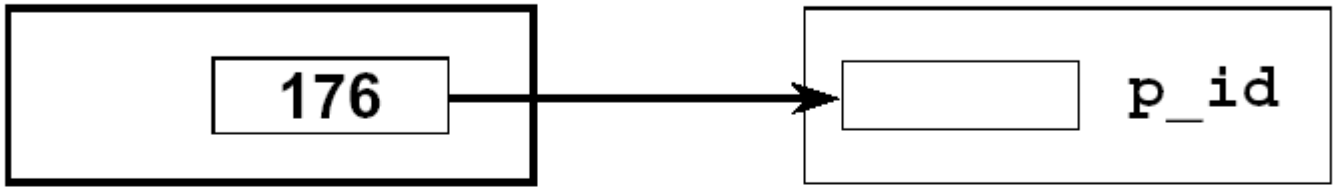


Syntax for Creating Procedures

```
CREATE [OR REPLACE] PROCEDURE procedure_name
 [(parameter1 [mode1] datatype1,
  parameter2 [mode2] datatype2,
  . . .)]
IS|AS
PL/SQL Block;
```

- The `REPLACE` option indicates that if the procedure exists, it will be dropped and replaced with the new version created by the statement.
- PL/SQL block starts with either `BEGIN` or the declaration of local variables and ends with either `END` or `END` *procedure_name*.

IN Parameters: Example




```
CREATE OR REPLACE PROCEDURE raise_salary
  (p_id IN employees.employee_id%TYPE)
IS
BEGIN
  UPDATE employees
  SET    salary = salary * 1.10
  WHERE  employee_id = p_id;
END raise_salary;
/
```

```
Procedure created.
```

OUT Parameters: Example

emp_query.sql

```
CREATE OR REPLACE PROCEDURE query_emp
  (p_id       IN   employees.employee_id%TYPE,
   p_name     OUT  employees.last_name%TYPE,
   p_salary   OUT  employees.salary%TYPE,
   p_comm     OUT  employees.commission_pct%TYPE)
IS
BEGIN
  SELECT   last_name, salary, commission_pct
   INTO     p_name, p_salary, p_comm
   FROM     employees
   WHERE    employee_id = p_id;
END query_emp;
/
```

Viewing OUT Parameters

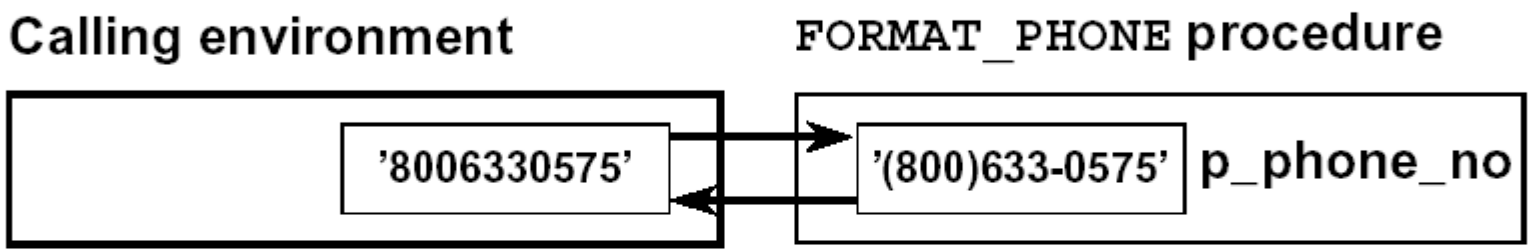
- Load and run the `emp_query.sql` script file to create the `QUERY_EMP` procedure.
- Declare host variables, execute the `QUERY_EMP` procedure, and print the value of the global variable `G_NAME`.

```
VARIABLE g_name      VARCHAR2(25)
VARIABLE g_sal       NUMBER
VARIABLE g_comm      NUMBER

EXECUTE query_emp(171, :g_name, :g_sal, :g_comm)

PRINT g_name
```

PL/SQL procedure successfully completed.

G_NAME
Smith

IN OUT Parameters




```
CREATE OR REPLACE PROCEDURE format_phone
  (p_phone_no IN OUT VARCHAR2)
IS
BEGIN
  p_phone_no := '(' || SUBSTR(p_phone_no,1,3) ||
                ')' || SUBSTR(p_phone_no,4,3) ||
                '-' || SUBSTR(p_phone_no,7);
END format_phone;
/
```

Viewing IN OUT Parameters

```
VARIABLE g_phone_no VARCHAR2(15)
BEGIN
  :g_phone_no := '8006330575';
END;
/
PRINT g_phone_no
EXECUTE format_phone (:g_phone_no)
PRINT g_phone_no
```

PL/SQL procedure successfully completed.

G_PHONE_NO
8006330575

PL/SQL procedure successfully completed.

G_PHONE_NO
(800)633-0575

Invoking a Procedure from an Anonymous PL/SQL Block

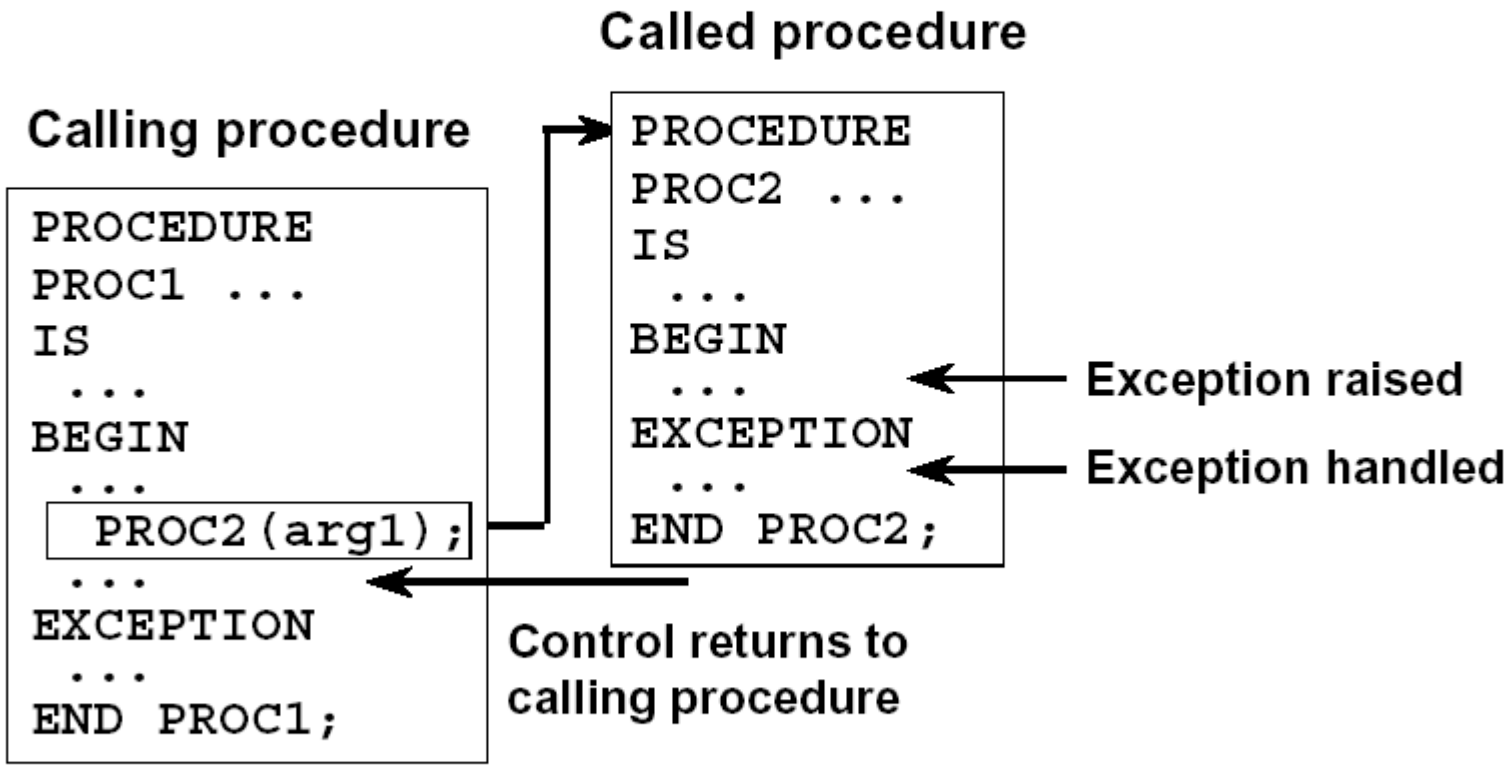
```
DECLARE
  v_id NUMBER := 163;
BEGIN
  raise_salary(v_id);    --invoke procedure
  COMMIT;
...
END;
```

Invoking a Procedure from Another Procedure

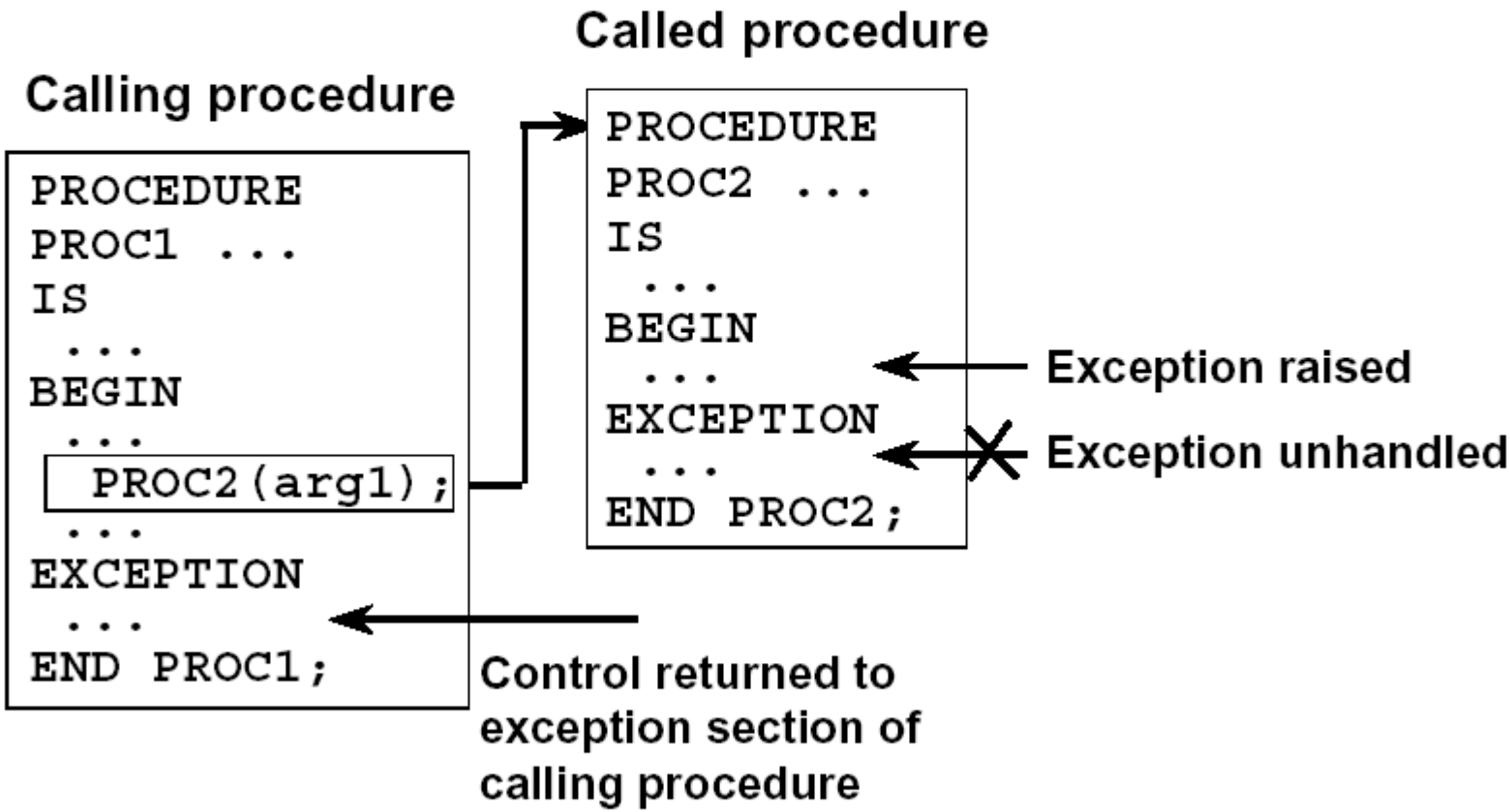
process_emps.sql

```
CREATE OR REPLACE PROCEDURE process_emps
IS
   CURSOR emp_cursor IS
    SELECT employee_id
    FROM   employees;
BEGIN
   FOR emp_rec IN emp_cursor
   LOOP
    raise_salary(emp_rec.employee_id);
   END LOOP;
   COMMIT;
END process_emps;
/
```

Handled Exceptions



Unhandled Exceptions



Removing Procedures

Drop a procedure stored in the database.

Syntax:

```
DROP PROCEDURE procedure_name
```

Example:

```
DROP PROCEDURE raise_salary;
```

```
Procedure dropped.
```

Syntax for Creating Functions

```
CREATE [OR REPLACE] FUNCTION function_name
 [(parameter1 [mode1] datatype1,
  parameter2 [mode2] datatype2,
  . . .)]
RETURN datatype
IS|AS
PL/SQL Block;
```

The PL/SQL block must have at least one `RETURN` statement.

Executing Functions

- Invoke a function as part of a PL/SQL expression.
- Create a variable to hold the returned value.
- Execute the function. The variable will be populated by the value returned through a `RETURN` statement.

Invoking Functions in SQL Expressions: Example

```
CREATE OR REPLACE FUNCTION tax(p_value IN NUMBER)
 RETURN NUMBER IS
BEGIN
   RETURN (p_value * 0.08);
END tax;
/
SELECT employee_id, last_name, salary, tax(salary)
FROM   employees
WHERE  department_id = 100;
```

Function created.

EMPLOYEE_ID	LAST_NAME	SALARY	TAX(SALARY)
108	Greenberg	12000	960
109	Faviet	9000	720
110	Chen	8200	656
111	Sciarra	7700	616
112	Urman	7800	624
113	Popp	6900	552

6 rows selected.

Locations to Call User-Defined Functions

- Select list of a `SELECT` command
- Condition of the `WHERE` and `HAVING` clauses
- `CONNECT BY`, `START WITH`, `ORDER BY`, and `GROUP BY` clauses
- `VALUES` clause of the `INSERT` command
- `SET` clause of the `UPDATE` command

Restrictions on Calling Functions from SQL Expressions

To be callable from SQL expressions, a user-defined function must:

- Be a stored function
- Accept only `IN` parameters
- Accept only valid SQL data types, not PL/SQL specific types, as parameters
- Return data types that are valid SQL data types, not PL/SQL specific types

Restrictions on Calling Functions from SQL Expressions

- Functions called from SQL expressions cannot contain DML statements.
- Functions called from `UPDATE/DELETE` statements on a table T cannot contain DML on the same table T.
- Functions called from a DML statement on a table T cannot query the same table.
- Functions called from SQL statements cannot contain statements that end the transactions.
- Calls to subprograms that break the previous restriction are not allowed in the function.

Removing Functions

Drop a stored function.

Syntax:

```
DROP FUNCTION function_name
```

Example:

```
DROP FUNCTION get_sal;
```

```
Function dropped.
```

- All the privileges granted on a function are revoked when the function is dropped.
- The `CREATE OR REPLACE` syntax is equivalent to dropping a function and recreating it. Privileges granted on the function remain the same when this syntax is used.

Comparing Procedures and Functions

Procedure	Function
Execute as a PL/SQL statement	Invoke as part of an expression
No `RETURN` clause in the header	Must contain a `RETURN` clause in the header
Can return none, one, or many values	Must return a single value
Can contain a `RETURN` statement	Must contain at least one `RETURN` statement

Required Privileges

System privileges

DBA grants →

```
CREATE (ANY) PROCEDURE
ALTER   ANY  PROCEDURE
DROP    ANY  PROCEDURE
EXECUTE ANY  PROCEDURE
```

Object privileges

Owner grants →

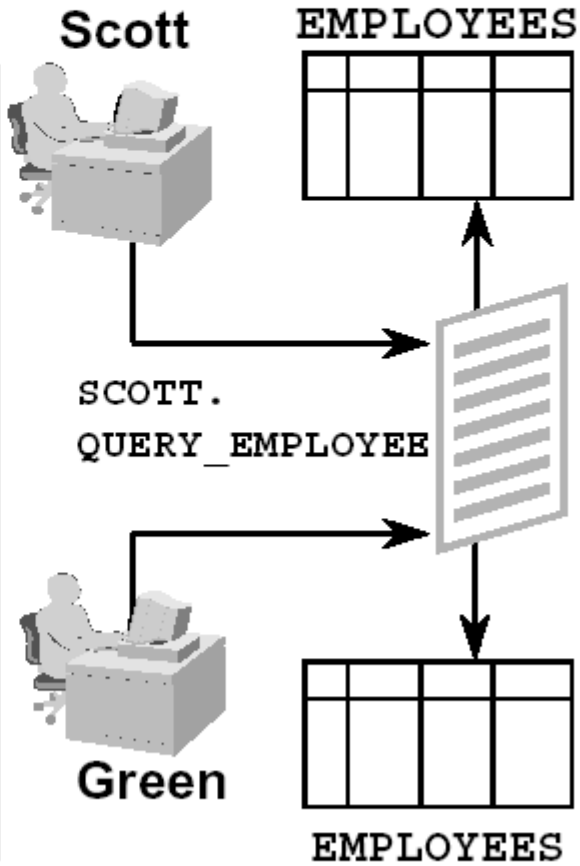
```
EXECUTE
```

To be able to refer and access objects from a different schema in a subprogram, you must be granted access to the referred objects explicitly, not through a role.

Using Invoker's-Rights

The procedure executes with the privileges of the user.

```
CREATE PROCEDURE query_employee
(p_id IN employees.employee_id%TYPE,
 p_name OUT employees.last_name%TYPE,
 p_salary OUT employees.salary%TYPE,
 p_comm OUT
   employees.commission_pct%TYPE)
AUTHID CURRENT_USER
IS
BEGIN
  SELECT last_name, salary,
         commission_pct
   INTO  p_name, p_salary, p_comm
   FROM  employees
   WHERE employee_id=p_id;
END query_employee;
```



USER_OBJECTS

Column	Column Description
OBJECT_NAME	Name of the object
OBJECT_ID	Internal identifier for the object
OBJECT_TYPE	Type of object, for example, TABLE, PROCEDURE, FUNCTION, PACKAGE, PACKAGE BODY, TRIGGER
CREATED	Date when the object was created
LAST_DDL_TIME	Date when the object was last modified
TIMESTAMP	Date and time when the object was last recompiled
STATUS	VALID or INVALID

*Abridged column list

List All Procedures and Functions

```
SELECT object_name, object_type
FROM  user_objects
WHERE object_type in ('PROCEDURE',
'FUNCTION')ORDER BY object_name;
```

OBJECT_NAME	OBJECT_TYPE
ADD_DEPT	PROCEDURE
ADD_JOB	PROCEDURE
ADD_JOB_HISTORY	PROCEDURE
ANNUAL_COMP	FUNCTION
DEL_JOB	PROCEDURE
FORMAT_PHONE	PROCEDURE
LEAVE_EMP	PROCEDURE
LEAVE_EMP2	PROCEDURE
LO[illegible]TION	PROCEDURE

20 rows selected.

USER_SOURCE Data Dictionary View

Column	Column Description
NAME	Name of the object
TYPE	Type of object, for example, PROCEDURE, FUNCTION, PACKAGE, PACKAGE BODY
LINE	Line number of the source code
TEXT	Text of the source code line

List the Code of Procedures and Functions

```
SELECT  text
FROM  user_source
WHERE  name = 'QUERY_EMPLOYEE'
ORDER BY  line;
```

TEXT
PROCEDURE query_employee
(p_id IN employees.employee_id%TYPE, p_name OUT employees.last_name%TYPE,
p_salary OUT employees.salary%TYPE, p_comm OUT employees.commission_pct%TYPE)
AUTHID CURRENT_USER
IS
BEGIN
SELECT last_name, salary, commission_pct
INTO p_name,p_salary,p_comm
FROM employees
WHERE employee_id=p_id;
END query_employee;

11 rows selected.

USER_ERRORS

Column	Column Description
NAME	Name of the object
TYPE	Type of object, for example, PROCEDURE, FUNCTION, PACKAGE, PACKAGE BODY, TRIGGER
SEQUENCE	Sequence number, for ordering
LINE	Line number of the source code at which the error occurs
POSITION	Position in the line at which the error occurs
TEXT	Text of the error message

List Compilation Errors by Using USER_ERRORS

```
SELECT line || '/' || position POS, text
FROM   user_errors
WHERE  name = 'LOG_EXECUTION'
ORDER BY line;
```

POS	TEXT
4/7	PLS-00103: Encountered the symbol "INTO" when expecting one of the following: := . (@ % ;
5/1	PLS-00103: Encountered the symbol "VALUES" when expecting one of the following: . (, % ; limit The symbol "VALUES" was ignored.
6/1	PLS-00103: Encountered the symbol "END"

List Compilation Errors by Using SHOW ERRORS

```
SHOW ERRORS PROCEDURE log_execution
```

Errors for PROCEDURE LOG_EXECUTION:

LINE/COL	ERROR
4/7	PLS-00103: Encountered the symbol "INTO" when expecting one of th e following: := . (@ % ;
5/1	PLS-00103: Encountered the symbol "VALUES" when expecting one of the following: . (, % ; limit The symbol "VALUES" was ignore d.
6/1	PLS-00103: Encountered the symbol "END"

Debugging PL/SQL Program Units

- The `DBMS_OUTPUT` package:
  - Accumulates information into a buffer
  - Allows retrieval of the information from the buffer
- Autonomous procedure calls (for example, writing the output to a log table)
- Software that uses `DBMS_DEBUG`
  - Procedure Builder
  - Third-party debugging software

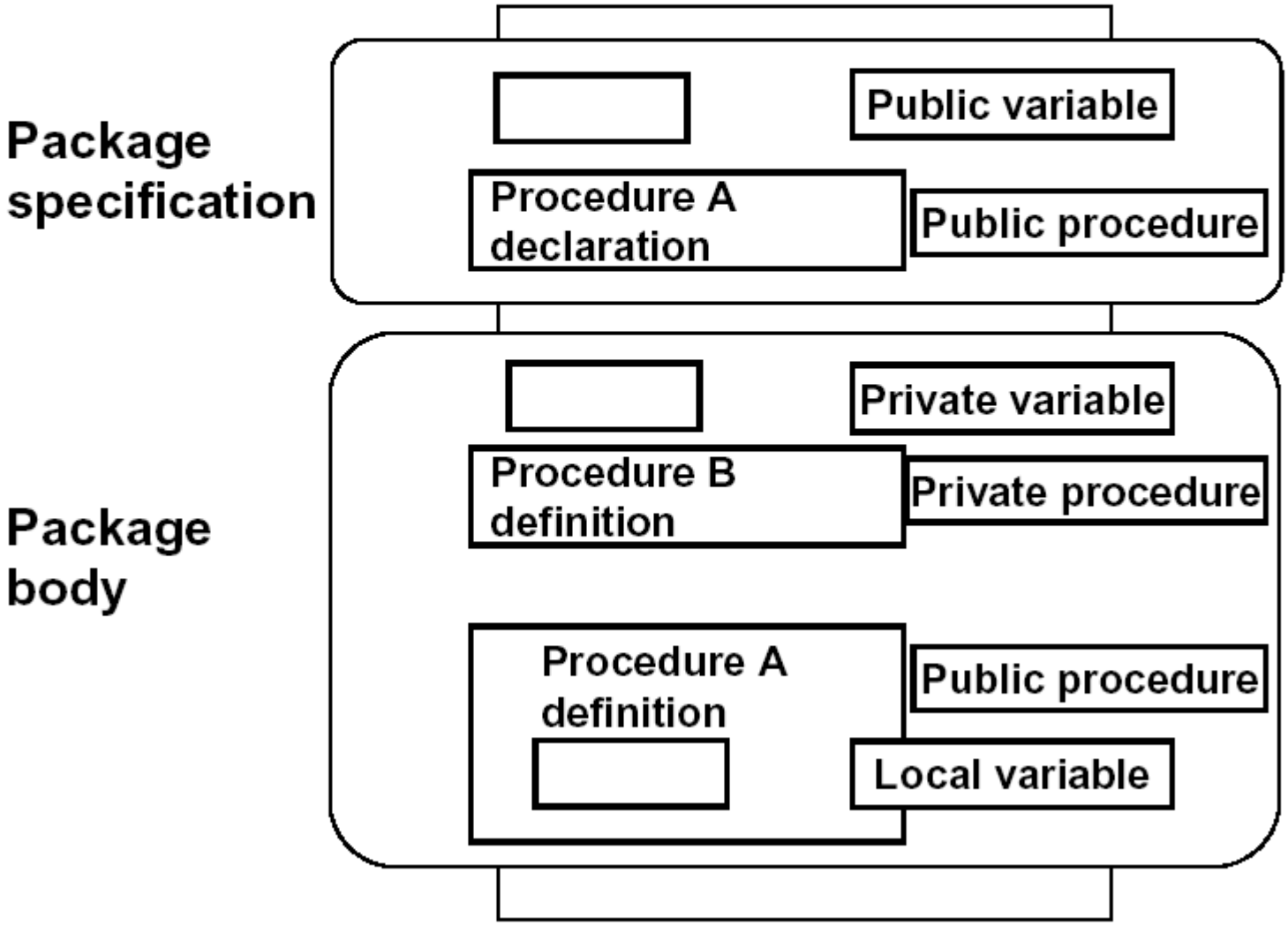
Package

Overview of Packages

Packages:

- Group logically related PL/SQL types, items, and subprograms
- Consist of two parts:
  - Specification
  - Body
- Cannot be invoked, parameterized, or nested
- Allow the Oracle server to read multiple objects into memory at once

Components of a Package



Developing a Package

- Saving the text of the `CREATE PACKAGE` statement in two different SQL files facilitates later modifications to the package.
- A package specification can exist without a package body, but a package body cannot exist without a package specification.

Creating the Package Specification

Syntax:

```
CREATE [OR REPLACE] PACKAGE package_name
IS|AS
    public type and item declarations
    subprogram specifications
END package_name;
```

- The `REPLACE` option drops and recreates the package specification.
- Variables declared in the package specification are initialized to `NULL` by default.
- All the constructs declared in a package specification are visible to users who are granted privileges on the package.

Creating a Package Specification: Example

```
CREATE OR REPLACE PACKAGE comm_package IS
  g_comm NUMBER := 0.10;  --initialized to 0.10
  PROCEDURE reset_comm
  (p_comm   IN  NUMBER);
END comm_package;
/
```

```
Package created.
```

- `G_COMM` is a global variable and is initialized to 0.10.
- `RESET_COMM` is a public procedure that is implemented in the package body.

Creating the Package Body

Syntax:

```
CREATE [OR REPLACE] PACKAGE BODY package_name
IS|AS
    private type and item declarations
    subprogram bodies
END package_name;
```

- The `REPLACE` option drops and recreates the package body.
- Identifiers defined only in the package body are private constructs. These are not visible outside the package body.
- All private constructs must be declared before they are used in the public constructs.

Creating a Package Body: Example

comm_pack.sql

```
CREATE OR REPLACE PACKAGE BODY comm_package
IS
   FUNCTION  validate_comm (p_comm IN NUMBER)
    RETURN BOOLEAN
   IS
     v_max_comm    NUMBER;
   BEGIN
     SELECT    MAX(commission_pct)
      INTO     v_max_comm
      FROM     employees;
     IF   p_comm > v_max_comm THEN RETURN(FALSE);
     ELSE   RETURN(TRUE);
     END IF;
   END validate_comm;
...
```

Creating a Package Body: Example

comm_pack.sql

```
  PROCEDURE  reset_comm (p_comm   IN  NUMBER)
  IS
  BEGIN
   IF  validate_comm(p_comm)
    THEN   g_comm:=p_comm;  --reset global variable
   ELSE
    RAISE_APPLICATION_ERROR(-20210,'Invalid commission');
   END IF;
  END reset_comm;
END comm_package;
/
```

Package body created.

Invoking Package Constructs

Example 1: Invoke a function from a procedure within the same package.

```
CREATE OR REPLACE PACKAGE BODY comm_package IS
   . . .
 PROCEDURE reset_comm
  (p_comm  IN  NUMBER)
 IS
 BEGIN
  IF validate_comm(p_comm)
  THEN g_comm := p_comm;
  ELSE
    RAISE_APPLICATION_ERROR
         (-20210, 'Invalid commission');
  END IF;
 END reset_comm;
END comm_package;
```

ORACLE

Invoking Package Constructs

Example 2: Invoke a package procedure from *i*SQL*Plus.

```
EXECUTE comm_package.reset_comm(0.15)
```

Example 3: Invoke a package procedure in a different schema.

```
EXECUTE scott.comm_package.reset_comm(0.15)
```

Example 4: Invoke a package procedure in a remote database.

```
EXECUTE comm_package.reset_comm@ny(0.15)
```

Declaring a Bodiless Package

```
CREATE OR REPLACE PACKAGE global_consts IS
  mile_2_kilo    CONSTANT  NUMBER  :=  1.6093;
  kilo_2_mile    CONSTANT  NUMBER  :=  0.6214;
  yard_2_meter   CONSTANT  NUMBER  :=  0.9144;
  meter_2_yard   CONSTANT  NUMBER  :=  1.0936;
END global_consts;
/

EXECUTE DBMS_OUTPUT.PUT_LINE('20 miles = '||20*
             global_consts.mile_2_kilo||' km')
```

```
Package created.
20 miles = 32.186 km
PL/SQL procedure successfully completed.
```

Removing Packages

To remove the package specification and the body, use the following syntax:

```
DROP PACKAGE package_name;
```

To remove the package body, use the following syntax :

```
DROP PACKAGE BODY package_name;
```

Guidelines for Developing Packages

- Construct packages for general use.
- Define the package specification before the body.
- The package specification should contain only those constructs that you want to be public.
- Place items in the declaration part of the package body when you must maintain them throughout a session or across transactions.
- Changes to the package specification require recompilation of each referencing subprogram.
- The package specification should contain as few constructs as possible.

Advantages of Packages

- Modularity: Encapsulate related constructs
- Easier application design: Code and compile specification and body separately
- Hiding information :
  - Only the declarations in the package specification are visible and accessible to applications
  - Private constructs in the package body are hidden and inaccessible
  - All coding is hidden in the package body

Advantages of Packages

- Added functionality: Persistency of variables and cursors
- Better performance:
  - The entire package is loaded into memory when the package is first referenced
  - There is only one copy in memory for all users
  - The dependency hierarchy is simplified
- Overloading: Multiple subprograms of the same name

Overloading: Example

over_pack.sql

```
CREATE OR REPLACE PACKAGE over_pack
IS
  PROCEDURE add_dept
   (p_deptno IN departments.department_id%TYPE,
    p_name IN departments.department_name%TYPE
                                  DEFAULT 'unknown',
    p_loc IN departments.location_id%TYPE DEFAULT 0);
  PROCEDURE add_dept
   (p_name IN departments.department_name%TYPE
                                  DEFAULT 'unknown',
    p_loc  IN departments.location_id%TYPE DEFAULT 0);
END over_pack;
/
```

```
Package created.
```

Overloading: Example

- Most built-in functions are overloaded.
- For example, see the `TO_CHAR` function of the `STANDARD` package.

```
FUNCTION TO_CHAR (p1 DATE) RETURN VARCHAR2;
FUNCTION TO_CHAR (p2 NUMBER) RETURN VARCHAR2;
FUNCTION TO_CHAR (p1 DATE, P2 VARCHAR2) RETURN VARCHAR2;
FUNCTION TO_CHAR (p1 NUMBER, P2 VARCHAR2) RETURN VARCHAR2;
```

- If you redeclare a built-in subprogram in a PL/SQL program, your local declaration overrides the global declaration.

Using Forward Declarations

You must declare identifiers before referencing them.

```
CREATE OR REPLACE PACKAGE BODY forward_pack
IS
  PROCEDURE award_bonus(. . .)
  IS
  BEGIN
   calc_rating(. . .);        --illegal reference

  END;

  PROCEDURE calc_rating(. . .)
  IS
  BEGIN
    ...
  END;

END forward_pack;
/
```

Using Forward Declarations

```
CREATE OR REPLACE PACKAGE BODY forward_pack
IS

 PROCEDURE calc_rating(. . .);    -- forward declaration

 PROCEDURE award_bonus(. . .)
 IS                               -- subprograms defined
 BEGIN                            -- in alphabetical order
  calc_rating(. . .);
 . . .
 END;

 PROCEDURE calc_rating(. . .)
 IS
 BEGIN
  . . .
 END;

END forward_pack;
/
```

Invoking a User-Defined Package Function from a SQL Statement

```
SELECT taxes_pack.tax(salary), salary, last_name
FROM   employees;
```

TAXES_PACK.TAX(SALARY)	SALARY	LAST_NAME
1920	24000	King
1360	17000	Kochhar
1360	17000	De Haan
720	9000	Hunold
480	6000	Ernst
384	4800	Austin
384	4800	Pataballa

109 rows selected.

Using Native Dynamic SQL

Dynamic SQL:

- Is a SQL statement that contains variables that may change during run-time
- Is a SQL statement with placeholders and is stored as a character string
- Enables general-purpose code to be written
- Enables data-definition and data-control or session-control statements to be written and executed from PL/SQL
- Is written using either `DBMS_SQL` or native dynamic SQL

Using the DBMS_SQL Package

The DBMS_SQL package is used to write dynamic SQL in stored procedures and to parse DDL statements. Some of the procedures and functions of the package include:

- OPEN_CURSOR
- PARSE
- BIND_VARIABLE
- EXECUTE
- FETCH_ROWS
- CLOSE_CURSOR

Using DBMS_SQL

```
CREATE OR REPLACE PROCEDURE delete_all_rows
   (p_tab_name IN VARCHAR2, p_rows_del OUT NUMBER)
 IS
   cursor_name   INTEGER;
 BEGIN
   cursor_name := DBMS_SQL.OPEN_CURSOR;
   DBMS_SQL.PARSE(cursor_name, 'DELETE FROM '||p_tab_name,
                  DBMS_SQL.NATIVE );
   p_rows_del := DBMS_SQL.EXECUTE (cursor_name);
   DBMS_SQL.CLOSE_CURSOR(cursor_name);
END;
/
```

Use dynamic SQL to delete rows

```
VARIABLE deleted NUMBER
EXECUTE  delete_all_rows('employees', :deleted)
PRINT deleted
```

PL/SQL procedure successfully completed.

DELETED
109

Dynamic SQL Using EXECUTE IMMEDIATE

```
CREATE PROCEDURE del_rows
  (p_table_name  IN  VARCHAR2,
   p_rows_deld   OUT NUMBER)
IS
BEGIN
  EXECUTE IMMEDIATE 'delete from '||p_table_name;
  p_rows_deld := SQL%ROWCOUNT;
END;
/
```

PL/SQL procedure successfully completed.

```
VARIABLE deleted NUMBER
EXECUTE del_rows('test_employees',:deleted)
PRINT deleted
```

DELETED
109

Using the DBMS_DDL Package

The DBMS_DDL Package:

- Provides access to some SQL DDL statements from stored procedures
- Includes some procedures:
  - ALTER_COMPILE (object_type, owner, object_name)

```
DBMS_DDL.ALTER_COMPILE('PROCEDURE','A_USER','QUERY_EMP')
```

  - ANALYZE_OBJECT (object_type, owner, name, method)

```
DBMS_DDL.ANALYZE_OBJECT('TABLE','A_USER','JOBS','COMPUTE')
```

Note: This package runs with the privileges of calling user, rather than the package owner SYS.

Using DBMS_JOB for Scheduling

DBMS_JOB Enables the scheduling and execution of PL/SQL programs:

- Submitting jobs
- Executing jobs
- Changing execution parameters of jobs
- Removing jobs
- Suspending Jobs

Using the DBMS_OUTPUT Package

The DBMS_OUTPUT Package enables you to output messages from PL/SQL blocks.

Available procedures include:

- PUT
- NEW_LINE
- PUT_LINE
- GET_LINE
- GET_LINES
- ENABLE/DISABLE

Interacting with Operating System Files

- `UTL_FILE` Oracle-supplied package:
  - Provides text file I/O capabilities
  - Is available with version 7.3 and later
- The `DBMS_LOB` Oracle-supplied package:
  - Provides read-only operations on external `BFILES`
  - Is available with version 8 and later
  - Enables read and write operations on internal `LOB`s

UTL_HTTP Package

The UTL_HTTP Package:

- Enables HTTP callouts from PL/SQL and SQL to access data on the Internet
- Contains the functions REQUEST and REQUEST_PIECES which take the URL of a site as a parameter, contact that site, and return the data obtained from that site
- Requires a proxy parameter to be specified in the above functions, if the client is behind a firewall
- Raises INIT_FAILED or REQUEST_FAILED exceptions if HTTP call fails
- Reports an HTML error message if specified URL is not accessible

Using the UTL_HTTP Package

```
SELECT UTL_HTTP.REQUEST('http://www.oracle.com',
                        'edu-proxy.us.oracle.com')
FROM DUAL;
```

```
UTL_HTTP.REQUEST('HTTP://WWW.ORACLE.COM','EDU-PROXY.US.ORACLE.COM')
--------------------------------------------------------------------
<head>
<title>Oracle Corporation</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-
8859-1">
<meta name="description" content="Oracle Corporation provides the
software that powers the Internet. For more information about
Oracle, please cal
l 650/506-7000.">
<meta name="keywords" content="Oracle, Oracle Corporation, Oracle
Corp,
Oracle8i, Oracle 9i, 8i, 9i">
</head>
...
```

Using the `UTL_TCP` Package

The `UTL_TCP` Package:

- Enables PL/SQL applications to communicate with external TCP/IP-based servers using TCP/IP
- Contains functions to open and close connections, to read or write binary or text data to or from a service on an open connection
- Requires remote host and port as well as local host and port as arguments to its functions
- Raises exceptions if the buffer size is too small, when no more data is available to read from a connection, when a generic network error occurs, or when bad arguments are passed to a function call

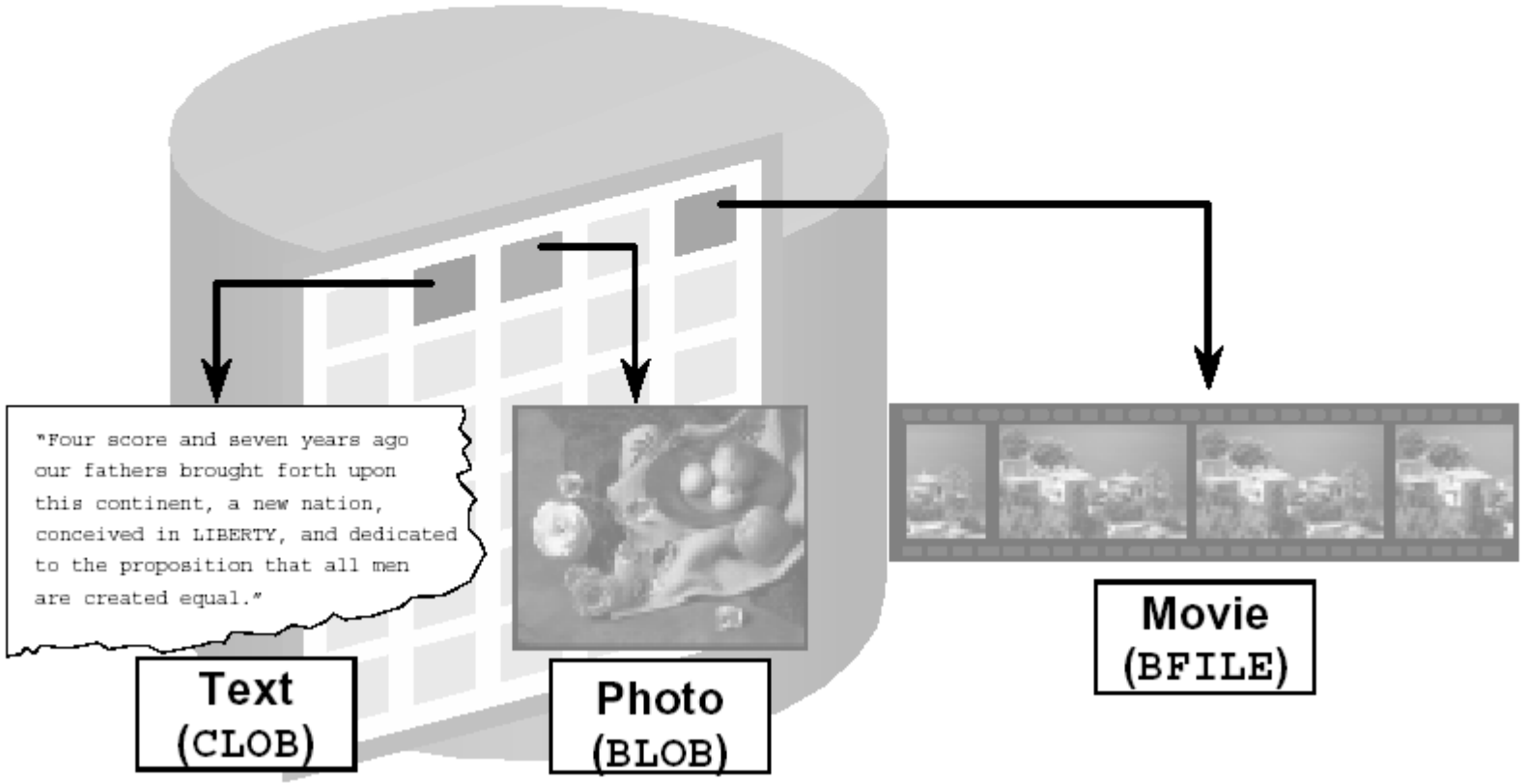
Oracle-Supplied Packages

Other Oracle-supplied packages include:

- DBMS_ALERT
- DBMS_APPLICATION_INFO
- DBMS_DESCRIBE
- DBMS_LOCK
- DBMS_SESSION
- DBMS_SHARED_POOL
- DBMS_TRANSACTION
- DBMS_UTILITY

What Is a LOB?

LOBs are used to store large unstructured data such as text, graphic images, films, and sound waveforms.

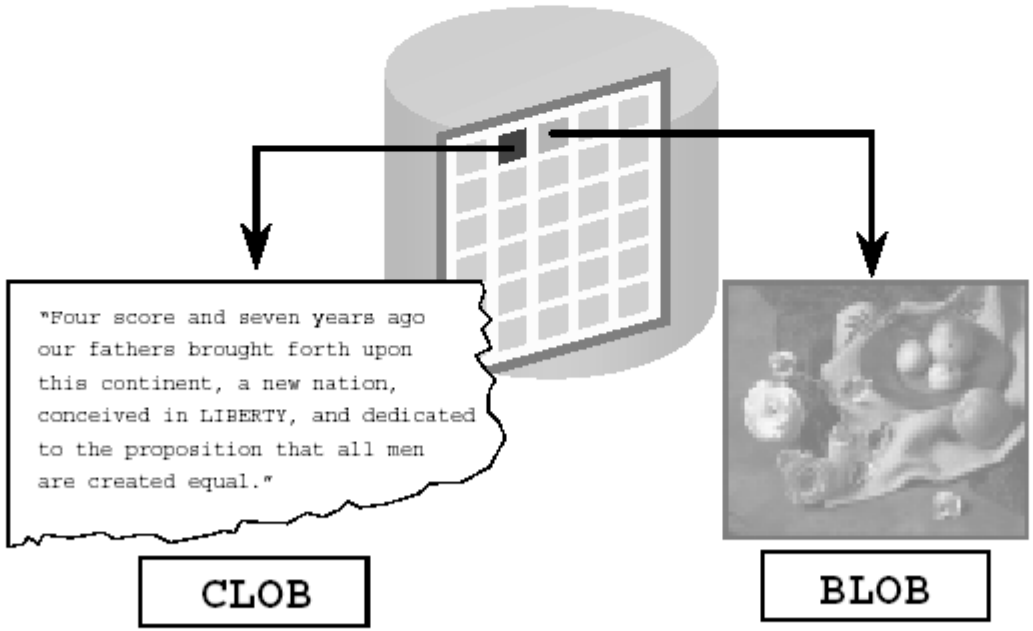


Contrasting LONG and LOB Data Types

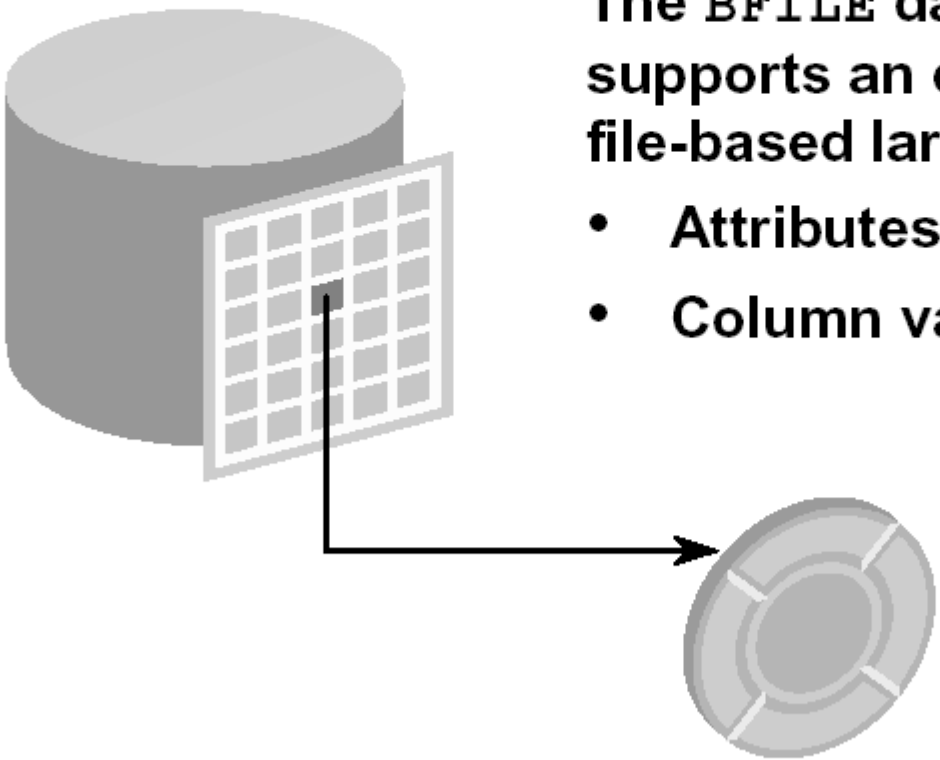
LONG and LONG RAW	LOB
Single LONG column per table	Multiple LOB columns per table
Up to 2 GB	Up to 4 GB
SELECT returns data	SELECT returns locator
Data stored in-line	Data stored in-line or out-of-line
Sequential access to data	Random access to data

Internal LOBs

The LOB value is stored in the database.



What Are BFILEs?



The BFILE data type supports an external or file-based large object as:

- Attributes in an object type
- Column values in a table

Managing BFILEs

- Create an OS directory and supply files.
- Create an Oracle table with a column that holds the BFILE data type.
- Create a DIRECTORY object.
- Grant privileges to read the DIRECTORY object to users.
- Insert rows into the table by using the BFILENAME function.
- Declare and initialize a LOB locator in a program.
- Read the BFILE.

The DBMS_LOB Package

- Working with LOB often requires the use of the Oracle-supplied package DBMS_LOB.
- DBMS_LOB provides routines to access and manipulate internal and external LOBs.
- Oracle9*i* enables retrieving LOB data directly using SQL, without using any special LOB API.
- In PL/SQL you can define a VARCHAR2 for a CLOB and a RAW for BLOB.

Trigger

Types of Triggers

A trigger:

- Is a PL/SQL block or a PL/SQL procedure associated with a table, view, schema, or the database
- Executes implicitly whenever a particular event takes place
- Can be either:
  - Application trigger: Fires whenever an event occurs with a particular application
  - Database trigger: Fires whenever a data event (such as DML) or system event (such as logon or shutdown) occurs on a schema or database

Creating DML Triggers

A triggering statement contains:

- Trigger timing
  - For table: `BEFORE`, `AFTER`
  - For view: `INSTEAD OF`
- Triggering event: `INSERT`, `UPDATE`, or `DELETE`
- Table name: On table, view
- Trigger type: Row or statement
- `WHEN` clause: Restricting condition
- Trigger body: PL/SQL block

DML Trigger Components

Trigger type: Should the trigger body execute for each row the statement affects or only once?

- Statement: The trigger body executes once for the triggering event. This is the default. A statement trigger fires once, even if no rows are affected at all.
- Row: The trigger body executes once for each row affected by the triggering event. A row trigger is not executed if the triggering event affects no rows.

Creating DML Statement Triggers

Example:

```
CREATE OR REPLACE TRIGGER secure_emp
 BEFORE INSERT ON employees
 BEGIN
  IF (TO_CHAR(SYSDATE,'DY') IN ('SAT','SUN')) OR
     (TO_CHAR(SYSDATE,'HH24:MI')
              NOT BETWEEN '08:00' AND '18:00')
   THEN RAISE_APPLICATION_ERROR (-20500,'You may
             insert into EMPLOYEES table only
                    during business hours.');
  END IF;
END;
/
```

Trigger created.

Testing SECURE_EMP

```
INSERT INTO employees (employee_id, last_name,
            first_name, email, hire_date,
            job_id, salary, department_id)
VALUES (300, 'Smith', 'Rob', 'RSMITH', SYSDATE,
        'IT_PROG', 4500, 60);
```

```
INSERT INTO employees (employee_id, last_name, first_name, email,
       *
ERROR at line 1:
ORA-20500: You may only insert into EMPLOYEES during
business hours.
ORA-06512: at "NEWPL.SECURE_EMP", line 4
ORA-04088: error during execution of trigger 'NEWPL.SECURE_EMP'
```

Using Conditional Predicates

```
CREATE OR REPLACE TRIGGER secure_emp
BEFORE INSERT OR UPDATE OR DELETE ON employees
BEGIN
 IF (TO_CHAR (SYSDATE,'DY') IN ('SAT','SUN')) OR
    (TO_CHAR (SYSDATE, 'HH24') NOT BETWEEN '08' AND '18')
 THEN
   IF  DELETING THEN
     RAISE_APPLICATION_ERROR (-20502,'You may delete  from
            EMPLOYEES table only during business hours.');
   ELSIF  INSERTING THEN
     RAISE_APPLICATION_ERROR (-20500,'You may insert into
            EMPLOYEES table only during business hours.');
   ELSIF   UPDATING ('SALARY') THEN
     RAISE_APPLICATION_ERROR (-20503,'You may update
                  SALARY only during business hours.');
   ELSE
     RAISE_APPLICATION_ERROR (-20504,'You may update
            EMPLOYEES table only during normal hours.');
   END IF;
  END IF;
END;
```

Creating DML Row Triggers

```
CREATE OR REPLACE TRIGGER restrict_salary
  BEFORE  INSERT OR UPDATE OF salary ON employees
    FOR EACH ROW
    BEGIN
      IF  NOT (:NEW.job_id IN ('AD_PRES', 'AD_VP'))
          AND :NEW.salary > 15000
      THEN
        RAISE_APPLICATION_ERROR (-20202,'Employee
                                cannot earn this amount');
     END IF;
END;
/
```

```
Trigger created.
```

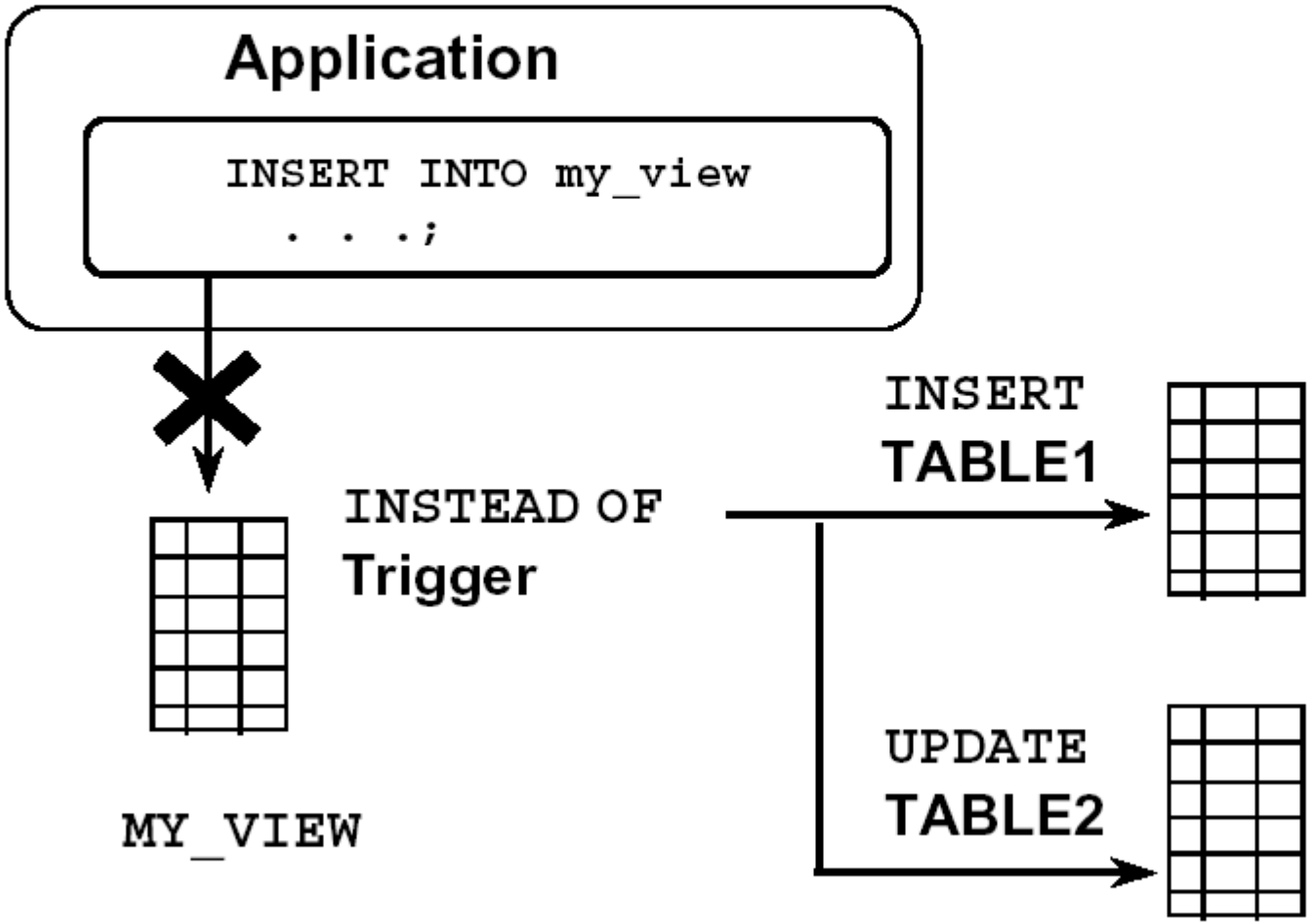
Using OLD and NEW Qualifiers

```
CREATE OR REPLACE TRIGGER audit_emp_values
 AFTER DELETE OR INSERT OR UPDATE ON employees
 FOR EACH ROW
BEGIN
  INSERT INTO audit_emp_table (user_name, timestamp,
     id, old_last_name, new_last_name, old_title,
     new_title, old_salary, new_salary)
  VALUES (USER, SYSDATE, :OLD.employee_id,
        :OLD.last_name, :NEW.last_name, :OLD.job_id,
        :NEW.job_id, :OLD.salary, :NEW.salary );
END;
/
```

Restricting a Row Trigger

```
CREATE OR REPLACE TRIGGER derive_commission_pct
  BEFORE INSERT OR UPDATE OF salary ON employees
  FOR EACH ROW
  WHEN (NEW.job_id = 'SA_REP')
BEGIN
  IF  INSERTING
     THEN :NEW.commission_pct := 0;
  ELSIF :OLD.commission_pct IS NULL
     THEN :NEW.commission_pct := 0;
  ELSE
    :NEW.commission_pct := :OLD.commission_pct + 0.05;
  END IF;
END;
/
```

INSTEAD OF Triggers



Managing Triggers

Disable or reenable a database trigger:

```
ALTER TRIGGER trigger_name  DISABLE | ENABLE
```

Disable or reenable all triggers for a table:

```
ALTER TABLE table_name   DISABLE | ENABLE  ALL TRIGGERS
```

Recompile a trigger for a table:

```
ALTER TRIGGER trigger_name COMPILE
```

DROP TRIGGER Syntax

To remove a trigger from the database, use the DROP TRIGGER syntax:

```
DROP TRIGGER trigger_name;
```

Example:

```
DROP TRIGGER secure_emp;
```

```
Trigger dropped.
```

Note: All triggers on a table are dropped when the table is dropped.

Creating Database Triggers

- Triggering user event:
  - CREATE, ALTER, or DROP
  - Logging on or off
- Triggering database or system event:
  - Shutting down or starting up the database
  - A specific error (or any error) being raised

LOGON and LOGOFF Trigger Example

```
CREATE OR REPLACE TRIGGER logon_trig
AFTER LOGON  ON  SCHEMA
BEGIN
 INSERT INTO log_trig_table(user_id, log_date, action)
 VALUES (USER, SYSDATE, 'Logging on');
END;
/
```

```
CREATE OR REPLACE TRIGGER logoff_trig
BEFORE LOGOFF  ON  SCHEMA
BEGIN
 INSERT INTO log_trig_table(user_id, log_date, action)
 VALUES (USER, SYSDATE, 'Logging off');
END;
/
```

CALL Statement

```
CREATE [OR REPLACE] TRIGGER trigger_name
  timing
    event1 [OR event2 OR event3]
      ON table_name
  [REFERENCING OLD AS old | NEW AS new]
[FOR EACH ROW]
  [WHEN condition]
CALL procedure_name;
```

```
CREATE OR REPLACE TRIGGER log_employee
BEFORE INSERT ON EMPLOYEES
  CALL log_execution
/
```

Implementating Triggers

You can use trigger for:

- Security
- Auditing
- Data integrity
- Referential integrity
- Table replication
- Computing derived data automatically
- Event logging

Viewing Trigger Information

You can view the following trigger information:

- `USER_OBJECTS` data dictionary view: Object information
- `USER_TRIGGERS` data dictionary view: The text of the trigger
- `USER_ERRORS` data dictionary view: PL/SQL syntax errors (compilation errors) of the trigger

Managing Dependencies

Understanding Dependencies

Dependent Objects	Referenced Objects
Table	Function
View	Package Specification
Database Trigger	Procedure
Procedure	Sequence
Function	Synonym
Package Body	Table
Package Specification	View
User-Defined Object and Collection Types	User-Defined Object and Collection Types

Recompiling a PL/SQL Program Unit

Recompilation:

- **Is handled automatically through implicit run-time recompilation.**
- **Is handled through explicit recompilation with the `ALTER` statement.**

```
ALTER PROCEDURE [SCHEMA.]procedure_name COMPILE;
```

```
ALTER FUNCTION  [SCHEMA.]function_name  COMPILE;
```

```
ALTER PACKAGE [SCHEMA.]package_name COMPILE [PACKAGE];
ALTER PACKAGE [SCHEMA.]package_name COMPILE BODY;
```

```
ALTER TRIGGER trigger_name [COMPILE[DEBUG]];
```

Unsuccessful Recompilation

Recompiling dependent procedures and functions is unsuccessful when:

- The referenced object is dropped or renamed
- The data type of the referenced column is changed
- The referenced column is dropped
- A referenced view is replaced by a view with different columns
- The parameter list of a referenced procedure is modified

Recompilation of Procedures

Minimize dependency failures by:

- Declaring records by using the `%ROWTYPE` attribute
- Declaring variables with the `%TYPE` attribute
- Querying with the `SELECT *` notation
- Including a column list with `INSERT` statements

QUESTIONS & ANSWERS