Oracle Database 11*g*：SQL 基础 II

学生指南第 1 册

D49994CN20
版本 2.0
2010 年 5 月
D66916

ORACLE

作者

Chaitanya Koratamaddi
Brian Pottle
Tulika Srivastava

技术撰稿人和审稿人

Claire Bennett
Ken Cooper
Yanti Chang
Laszlo Czinkoczki
Burt Demchick
Gerlinde Frenzen
Joel Goodman
Laura Garza
Richard Green
Nancy Greenberg
Akira Kinutani
Wendy Lo
Isabelle Marchand
Timothy Mcglue
Alan Paulson
Srinivas Putrevu
Bryan Roberts
Clinton Shaffer
Abhishek Singh
Jenny Tsai Smith
James Spiller
Lori Tritz
Lex van der Werff
Marcie Young

编辑

Amitha Narayan
Daniel Milne

制图员

Satish Bettegowda

出版商

Veena Narasimhan

目录

I 简介

课程目标 I-2
课程安排 I-3
课程目标 I-4
前导课程 I-5
课程安排 I-6
本课程中使用的表 I-8
本课程中使用的附录 I-9
开发环境 I-10
课程安排 I-11
复习对数据进行限制 I-12
复习对数据进行排序 I-13
复习 SQL 函数 I-14
复习单行函数 I-15
复习各类组函数 I-16
复习使用子查询 I-17
复习处理数据 I-19
课程安排 I-20
Oracle Database 11*g* 的 SQL 文档 I-21
其它资源 I-22
小结 I-23
练习 I：概览 I-24

1 控制用户访问

课程目标 1-2
课程安排 1-3
控制用户访问 1-4
权限 1-5
系统权限 1-6
创建用户 1-7
用户系统权限 1-8

授予系统权限 1-10
课程安排 1-11
角色是什么 1-12
创建角色和为角色授予权限 1-13
更改口令 1-14
课程安排 1-15
对象权限 1-16
授予对象权限 1-18
传递权限 1-19
确认授予的权限 1-20
课程安排 1-21
撤消对象权限 1-22
小测验 1-24
小结 1-25
练习 1：概览 1-26

2 管理方案对象
课程目标 2-2
课程安排 2-3
ALTER TABLE 语句 2-4
添加列 2-6
修改列 2-7
删除列 2-8
SET UNUSED 选项 2-9
课程安排 2-11
添加约束条件语法 2-12
添加约束条件 2-13
ON DELETE 子句 2-14
延迟约束条件 2-15
INITIALLY DEFERRED 与 INITIALLY IMMEDIATE 之间的区别 2-16
删除约束条件 2-19
禁用约束条件 2-20
启用约束条件 2-21
级联约束条件 2-23
重命名表列和约束条件 2-25
课程安排 2-26

索引概览 2-27
CREATE INDEX 与 CREATE TABLE 语句配合使用 2-28
基于函数的索引 2-30
删除索引 2-31
DROP TABLE … PURGE 2-32
课程安排 2-33
FLASHBACK TABLE 语句 2-34
使用 FLASHBACK TABLE 语句 2-36
课程安排 2-37
临时表 2-38
创建临时表 2-39
课程安排 2-40
外部表 2-41
创建外部表目录 2-42
创建外部表 2-44
使用 ORACLE_LOADER 创建外部表 2-46
查询外部表 2-48
使用 ORACLE_DATAPUMP 创建外部表：示例 2-49
小测验 2-50
小结 2-52
练习 2：概览 2-53

3 使用数据字典视图管理对象
课程目标 3-2
课程安排 3-3
数据字典 3-4
数据字典结构 3-5
如何使用字典视图 3-7
USER_OBJECTS 和 ALL_OBJECTS 视图 3-8
USER_OBJECTS 视图 3-9
课程安排 3-10
表信息 3-11
列信息 3-12
约束条件信息 3-14
USER_CONSTRAINTS：示例 3-15
在 USER_CONS_COLUMNS 中进行查询 3-17

课程安排 3-18
视图信息 3-19
序列信息 3-20
确认序列 3-21
索引信息 3-22
USER_INDEXES：示例 3-23
在 USER_IND_COLUMNS 中进行查询 3-24
同义词信息 3-25
课程安排 3-26
在表中添加注释 3-27
小测验 3-28
小结 3-29
练习 3：概览 3-30

4 处理大型数据集
课程目标 4-2
课程安排 4-3
使用子查询处理数据 4-4
通过将子查询用作源来检索数据 4-5
通过将子查询用作目标来执行插入 4-7
在 DML 语句中使用 WITH CHECK OPTION 关键字 4-9
课程安排 4-11
显式默认值功能概览 4-12
使用显式默认值 4-13
从其它表中复制行 4-14
课程安排 4-15
多表 INSERT 语句概览 4-16
多表 INSERT 语句的类型 4-18
多表 INSERT 语句 4-19
无条件 INSERT ALL 4-21
条件 INSERT ALL：示例 4-23
条件 INSERT ALL 4-24
条件 INSERT FIRST：示例 4-26
条件 INSERT FIRST 4-27
转换 INSERT (Pivoting INSERT) 4-29

课程安排 4-32
MERGE 语句 4-33
MERGE 语句的语法 4-34
合并行：示例 4-35
课程安排 4-38
跟踪数据更改 4-39
闪回版本查询示例 4-40
VERSIONS BETWEEN 子句 4-42
小测验 4-43
小结 4-44
练习 4：概览 4-45

5 管理不同时区中的数据
课程目标 5-2
课程安排 5-3
时区 5-4
TIME_ZONE 会话参数 5-5
CURRENT_DATE、CURRENT_TIMESTAMP 和 LOCALTIMESTAMP 5-6
对会话时区的日期和时间进行比较 5-7
DBTIMEZONE 和 SESSIONTIMEZONE 5-9
TIMESTAMP 数据类型 5-10
TIMESTAMP 字段 5-11
DATE 和 TIMESTAMP 之间的区别 5-12
TIMESTAMP 数据类型的比较 5-13
课程安排 5-14
INTERVAL 数据类型 5-15
INTERVAL 字段 5-17
INTERVAL YEAR TO MONTH：示例 5-18
INTERVAL DAY TO SECOND 数据类型：示例 5-20
课程安排 5-21
EXTRACT 5-22
TZ_OFFSET 5-23
FROM_TZ 5-25
TO_TIMESTAMP 5-26
TO_YMINTERVAL 5-27
TO_DSINTERVAL 5-28

夏令时 5-29
小测验 5-31
小结 5-32
练习 5：概览 5-33

6 使用子查询检索数据
课程目标 6-2
课程安排 6-3
多列子查询 6-4
列比较 6-5
成对比较子查询 6-6
不成对比较子查询 6-8
课程安排 6-10
标量子查询表达式 6-11
标量子查询：示例 6-12
课程安排 6-14
相关子查询 6-15
使用相关子查询 6-17
课程安排 6-19
使用 EXISTS 运算符 6-20
查找没有任何雇员的所有部门 6-22
相关 UPDATE 6-23
使用相关 UPDATE 6-24
相关 DELETE 6-26
使用相关 DELETE 6-27
课程安排 6-28
WITH 子句 6-29
WITH 子句：示例 6-30
递归 WITH 子句 6-32
递归 WITH 子句：示例 6-33
小测验 6-34
小结 6-35
练习 6：概览 6-37

7 正则表达式支持功能

课程目标 7-2
课程安排 7-3
正则表达式是什么 7-4
使用正则表达式的好处 7-5
在 SQL 和 PL/SQL 中使用正则表达式函数和条件 7-6
课程安排 7-7
元字符是什么 7-8
在正则表达式中使用元字符 7-9
课程安排 7-11
正则表达式函数和条件：语法 7-12
使用 REGEXP_LIKE 条件执行基本搜索 7-14
使用 REGEXP_REPLACE 函数替换模式 7-15
使用 REGEXP_INSTR 函数查找模式 7-16
使用 REGEXP_SUBSTR 函数提取子字符串 7-17
课程安排 7-18
子表达式 7-19
结合使用正则表达式支持与子表达式 7-20
为什么要访问第 *n* 个子表达式 7-21
REGEXP_SUBSTR：示例 7-22
课程安排 7-23
使用 REGEXP_COUNT 函数 7-24
正则表达式和检查约束条件：示例 7-25
小测验 7-26
小结 7-27
练习 7：概览 7-28

附录 A：练习和解答

附录 B：表说明

附录 C：使用 SQL Developer

课程目标 C-2
什么是 Oracle SQL Developer C-3
SQL Developer 说明 C-4
SQL Developer 1.5 界面 C-5
创建数据库连接 C-7
浏览数据库对象 C-10
显示表结构 C-11
浏览文件 C-12
创建方案对象 C-13
创建新表：示例 C-14
使用 SQL 工作表 C-15
执行 SQL 语句 C-18
保存 SQL 脚本 C-19
执行已保存的脚本文件：方法 1 C-20
执行已保存的脚本文件：方法 2 C-21
设置 SQL 代码的格式 C-22
使用片段 C-23
使用片段：示例 C-24
调试过程和函数 C-25
数据库报表 C-26
创建用户定义报表 C-28
搜索引擎和外部工具 C-29
设置首选项 C-30
重置 SQL Developer 布局 C-31
小结 C-32

附录 D：使用 SQL*Plus

课程目标 D-2
SQL 和 SQL*Plus 交互 D-3
SQL 语句和 SQL*Plus 命令 D-5
SQL*Plus 概览 D-6
登录到 SQL*Plus D-7
显示表结构 D-8
SQL*Plus 编辑命令 D-10
使用 LIST、n 和 APPEND D-12
使用 CHANGE 命令 D-13
SQL*Plus 文件命令 D-14
使用 SAVE 和 START 命令 D-15
SERVEROUTPUT 命令 D-16
使用 SQL*Plus 的 SPOOL 命令 D-17
使用 AUTOTRACE 命令 D-18
小结 D-19

附录 E：使用 JDeveloper

课程目标 E-2
Oracle JDeveloper E-3
数据库导航器 E-4
创建连接 E-5
浏览数据库对象 E-6
执行 SQL 语句 E-7
创建程序单元 E-8
编译 E-9
运行程序单元 E-10
删除程序单元 E-11
结构窗口 E-12
编辑器窗口 E-13
应用程序导航器 E-14
部署 Java 存储过程 E-15
将 Java 发布到 PL/SQL E-16
如何了解有关 JDeveloper 11*g* 的更多信息 E-17
小结 E-18

附录 F：通过将相关数据分组来生成报表

课程目标 F-2
复习组函数 F-3
复习 GROUP BY 子句 F-4
复习 HAVING 子句 F-5
GROUP BY 与 ROLLUP 和 CUBE 运算符一起使用 F-6
ROLLUP 运算符 F-7
ROLLUP 运算符：示例 F-8
CUBE 运算符 F-9
CUBE 运算符：示例 F-10
GROUPING 函数 F-11
GROUPING 函数：示例 F-12
GROUPING SETS F-13
GROUPING SETS：示例 F-15
组合列 F-17
组合列：示例 F-19
级联分组 F-21
级联分组：示例 F-22
小结 F-23

附录 G：分层检索

课程目标 G-2
EMPLOYEES 表中的示例数据 G-3
自然树结构 G-4
分层查询 G-5
遍历树 G-6
遍历树：自下而上 G-8
遍历树：自上而下 G-9
使用 LEVEL 伪列确定行的等级 G-10
使用 LEVEL 和 LPAD 设置分层报表的格式 G-11
修剪分支 G-13
小结 G-14

附录 H：编写高级脚本

课程目标 H-2
使用 SQL 生成 SQL H-3
创建基本脚本 H-4
控制环境 H-5
完整脚本 H-6
将表内容转储到文件 H-7
生成动态谓词 H-9
小结 H-11

附录 I：Oracle DB 体系结构组件

课程目标 I-2
Oracle DB 体系结构：概览 I-3
Oracle DB Server 结构 I-4
连接到数据库 I-5
与 Oracle DB 交互 I-6
Oracle 内存体系结构 I-8
进程体系结构 I-10
数据库写进程 I-12
日志写进程 I-13
检查点进程 I-14
系统监视器进程 I-15
进程监视器进程 I-16
Oracle DB 存储体系结构 I-17
逻辑和物理数据库结构 I-19
处理 SQL 语句 I-21
处理查询 I-22
共享池 I-23
数据库缓冲区高速缓存 I-25
程序全局区 (PGA) I-26
处理 DML 语句 I-27
重做日志缓冲区 I-29
回退段 I-30
COMMIT 处理 I-31
Oracle DB 体系结构小结 I-33

附加练习和解答

1

简介

课程目标

学完本课后，应能完成下列工作：

- 讨论本课程的目标
- 描述本课程中使用的数据库方案和表
- 确定本课程中可使用的环境
- 复习 SQL 的某些基本概念

课程安排

- 课程目标和课程安排
- 本课程中使用的数据库方案和附录及提供的开发环境
- 复习 SQL 的某些基本概念
- Oracle Database 11*g* 的文档和附加资源

课程目标

学完本课程后，应能完成以下工作：

- 控制对数据库特定对象的访问
- 添加具有不同访问权限级别的新用户
- 管理方案对象
- 使用数据字典视图管理对象
- 使用子查询处理 Oracle DB 中的大型数据集
- 管理不同时区中的数据
- 编写多列子查询
- 使用标量子查询和相关子查询
- 在 SQL 中使用正则表达式支持功能

ORACLE

前导课程

《Oracle Database 11*g*：SQL 基础 I》是本课程的前导课程。

前导课程

本课程的前导课程是《Oracle Database 11*g*：SQL 基础 I》。

本课程将介绍 Oracle Database 11*g* 数据库技术。在本课程中，将介绍有关关系数据库的基本概念及功能强大的 SQL 编程语言。还将介绍一些基本 SQL 技能，利用这些技能可以针对一个和多个表编写查询，处理表数据，创建数据库对象以及对元数据进行查询。

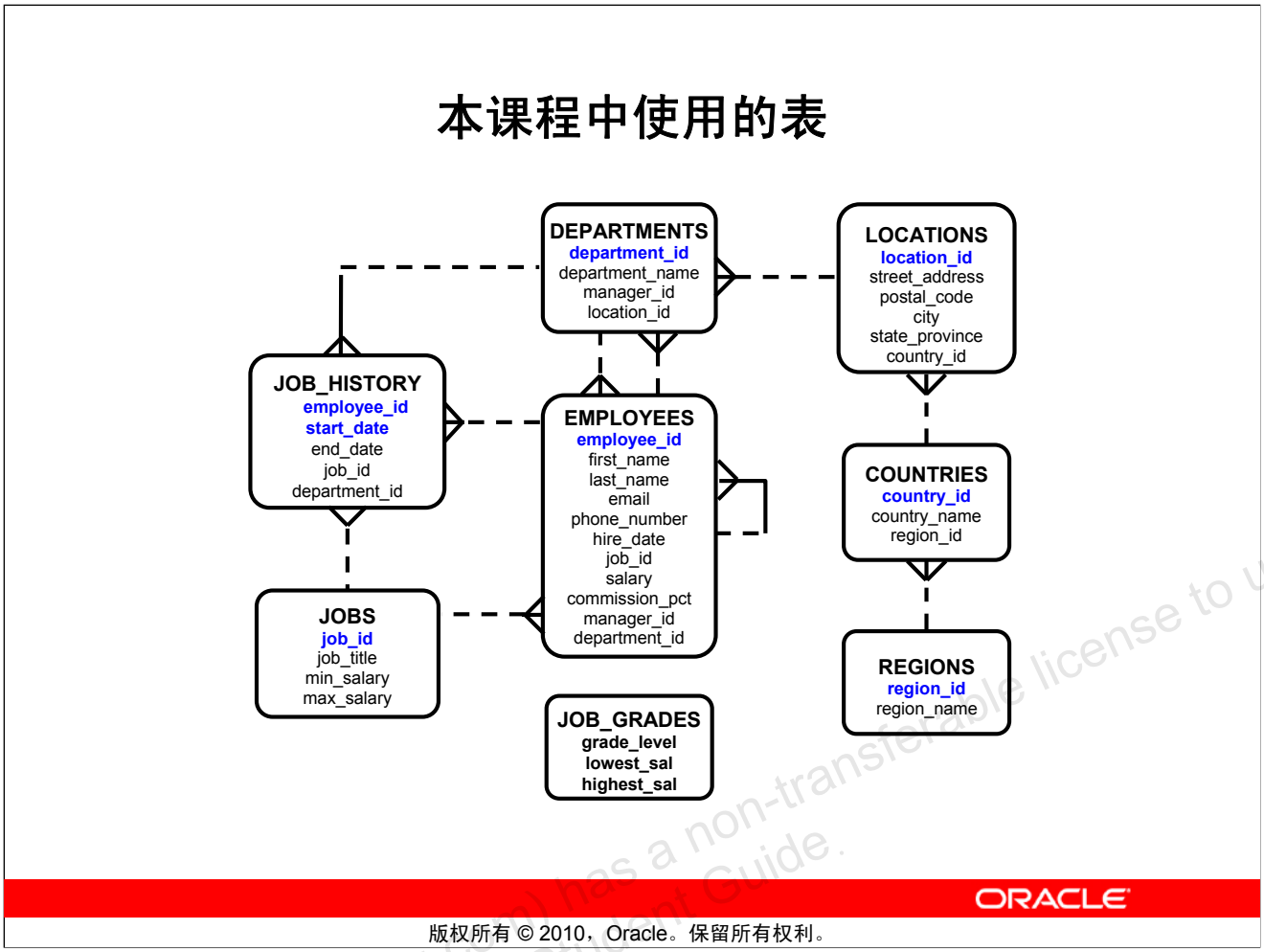
课程安排

- 第一天：
  - 简介
  - 控制用户访问
  - 管理方案对象
  - 使用数据字典视图管理对象
- 第二天：
  - 处理大型数据集
  - 管理不同时区中的数据
  - 使用子查询检索数据
  - 正则表达式支持功能

ORACLE

课程安排

- 课程目标和课程安排
- 本课程中使用的数据库方案和附录及提供的开发环境
- 复习 SQL 的某些基本概念
- Oracle Database 11*g* 的文档和附加资源



表说明

本课程使用的数据来自以下表：

表说明

- EMPLOYEES 表包含关于所有雇员的信息，如他们的姓氏和名字、职务 ID、薪金、雇佣日期、部门 ID 及经理 ID。此表是 DEPARTMENTS 表的子表。
- DEPARTMENTS 表包含诸如部门 ID、部门名称、经理 ID 和位置 ID 之类的信息。此表是 EMPLOYEES 表的主键表。
- LOCATIONS 表包含部门位置信息。其中包含位置 ID、街道地址、城市、州或省、邮政编码及国家/地区 ID 等信息它是 DEPARTMENTS 表的主键表，还是 COUNTRIES 表的子表。
- COUNTRIES 表包含国家/地区名称、国家/地区 ID 和区域 ID。它是 REGIONS 表的子表。此表是 LOCATIONS 表的主键表。
- REGIONS 表包含各个国家/地区的区域 ID 和区域名称。它是 COUNTRIES 表的主键表。
- JOB_GRADES 表确定每个职务级别的薪金范围。薪金范围不重叠。
- JOB_HISTORY 表存储雇员的职务历史记录。
- JOBS 表包含职位和薪金范围。

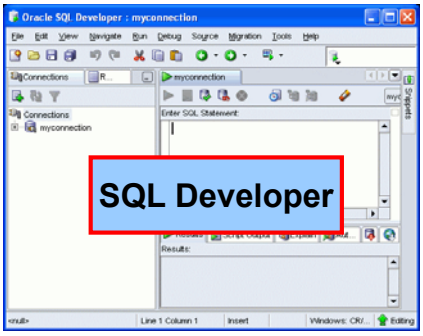
本课程中使用的附录

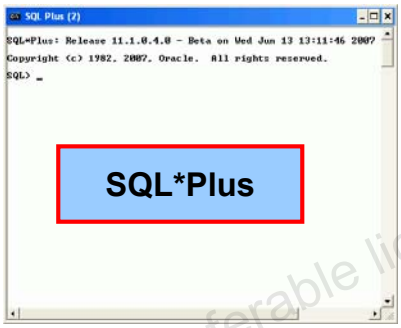
- 附录 A：练习和解答
- 附录 B：表说明
- 附录 C：使用 SQL Developer
- 附录 D：使用 SQL*Plus
- 附录 E：使用 JDeveloper
- 附录 F：通过将相关数据分组来生成报表
- 附录 G：分层检索
- 附录 H：编写高级脚本
- 附录 I：Oracle DB 体系结构组件

开发环境

针对本课程有两个开发环境：

- 主工具是 Oracle SQL Developer。
- 也可以使用 SQL*Plus 命令行接口。





开发环境

SQL Developer

本课程将使用 Oracle SQL Developer 工具来运行在幻灯片示例和练习中讨论的 SQL 语句。

- SQL Developer 版本 1.5.4 随附在 Oracle Database 11*g* 发行版 2 中，是本课的默认工具。
- 此外，课堂计算机上也提供有 SQL Developer 版本 1.5.4，可安装以供使用。截至本课程发布之日，SQL Developer 的最新发行版为版本 1.5.3。

SQL*Plus

SQL*Plus 环境还可用于运行本课程涉及到的所有 SQL 命令。

附注

- 请参阅附录 C“使用 SQL Developer”了解有关使用 SQL Developer 的信息，包括安装版本 1.5.4 的简明指导。
- 请参阅附录 D“使用 SQL*Plus”了解有关使用 SQL*Plus 的信息。

课程安排

- 课程目标和课程安排
- 本课程中使用的数据库方案和附录及提供的开发环境
- 复习 SQL 的某些基本概念
- Oracle Database 11*g* 的文档和附加资源

课程安排

接下来的几张幻灯片将简要介绍在《Oracle Database 11*g*：SQL 基础 I》课程中已介绍的某些基本概念。

复习对数据进行限制

- 使用 WHERE 子句可限制返回的行。
- 使用比较条件可将一个表达式与另一个值或另一个表达式进行比较。

运算符	含义
BETWEEN ...AND...	两值之间（包含这两个值）
IN(set)	与任一列表值相匹配
LIKE	与字符模式相匹配

- 使用逻辑条件可将两个组合条件的结果组合起来并根据这些条件生成一个结果。

ORACLE

复习对数据进行限制

使用 WHERE 子句可限制由查询返回的行。WHERE 子句包含一个必须满足的条件，该子句直接跟在 FROM 子句之后。

WHERE 子句可对列值、文字值、算术表达式或函数中的值进行比较。它包含三个元素：

- 列名
- 比较条件
- 列名、常数或值列表

在 WHERE 子句按以下格式使用比较条件：

```
... WHERE expr operator value
```

除幻灯片中提到的比较条件外，还可以使用其它比较条件，如 =、<、>、<>、<= 和 >= 等。

在 SQL 中可以使用以下三种逻辑运算符：

- AND
- OR
- NOT

复习对数据进行排序

- 使用 ORDER BY 子句可对检索行进行排序：
  - ASC：升序（默认值）
  - DESC：降序
- ORDER BY 子句位于 SELECT 语句的最后：

```
SELECT   last_name, job_id, department_id, hire_date
FROM     employees
ORDER BY hire_date ;
```

	LAST_NAME	JOB_ID	DEPARTMENT_ID	HIRE_DATE
1	King	AD_PRES	90	17-JUN-87
2	Whalen	AD_ASST	10	17-SEP-87
3	Kochhar	AD_VP	90	21-SEP-89
4	Hunold	IT_PROG	60	03-JAN-90
5	Ernst	IT_PROG	60	21-MAY-91
6	De Haan	AD_VP	90	13-JAN-93

...

复习对数据进行排序

查询结果中返回行的顺序是不确定的。可以使用 ORDER BY 子句对这些行进行排序。如果使用 ORDER BY 子句，该子句必须是 SQL 语句中的最后一个子句。可以指定使用表达式、别名或列位置作为排序条件。

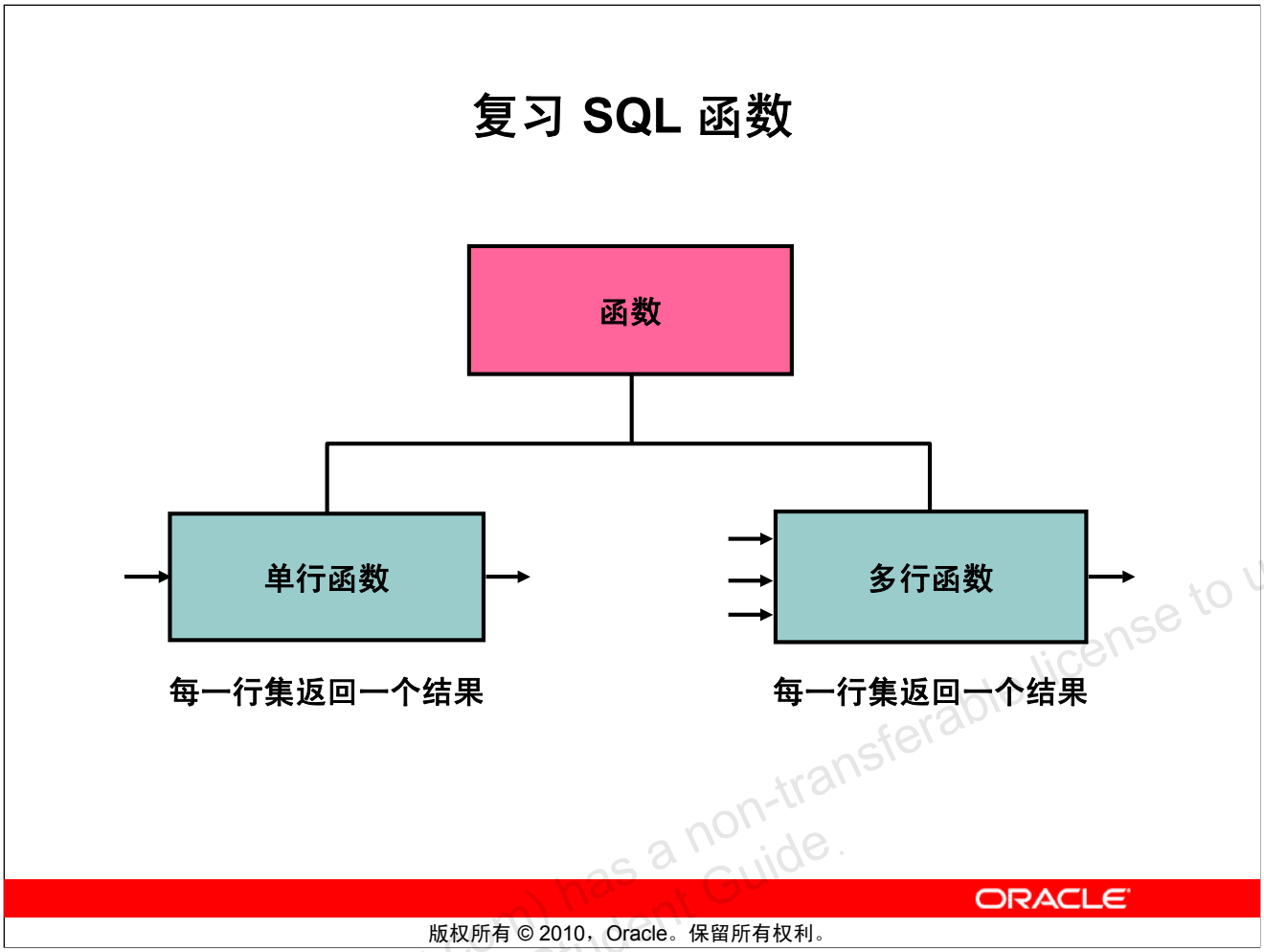
语法

```
SELECT          expr
FROM            table
[WHERE          condition(s)]
[ORDER BY  {column, expr, numeric_position} [ASC|DESC]];
```

在该语法中：

ORDER BY	指定检索行的显示顺序
ASC	按升序对这些行进行排序（这是默认顺序。）
DESC	按降序对这些行进行排序

如果没有使用 ORDER BY 子句，则排序顺序不确定，因此如果同一查询执行两次，Oracle Server 每次提取行的顺序可能不同。使用 ORDER BY 子句可按特定顺序显示行。



复习 SQL 函数

有两种类型的函数：

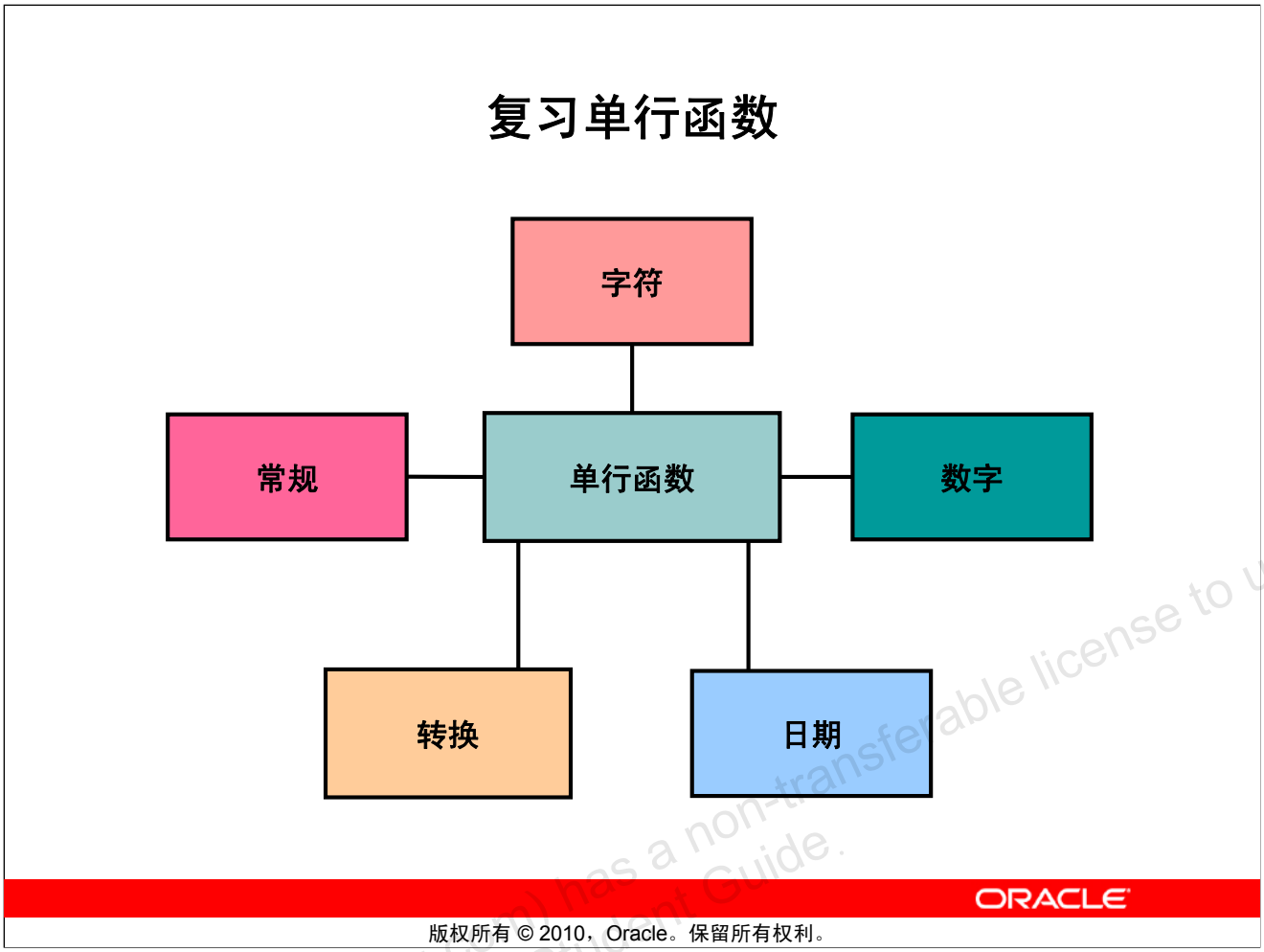
- 单行函数
- 多行函数

单行函数

这些函数仅对单行进行处理，每行返回一个结果。有各种类型的单行函数，如字符函数、数值函数、日期函数、转换函数和常规函数。

多行函数

这些函数可以处理成组的行，每组行返回一个结果。这些函数也称为“组函数”。



复习单行函数

各种类型的单行函数包括：

- **字符函数：**接受字符输入，可以返回字符值和数字值
- **数字函数：**接受数字输入，可以返回数字值
- **日期函数：**对 DATE 数据类型的值进行处理（所有日期函数都会返回一个 DATE 数据类型的值，只有 MONTHS_BETWEEN 函数返回一个数字。）
- **转换函数：**将值从一种数据类型转换为另一种数据类型
- **常规函数：**
  - NVL
  - NVL2
  - NULLIF
  - COALESCE
  - CASE
  - DECODE

复习各类组函数

- AVG
- COUNT
- MAX
- MIN
- STDDEV
- SUM
- VARIANCE

复习各类组函数

每个函数都接受一个参数。下表列出了在语法中可使用的选项：

函数	说明
AVG([DISTINCT|ALL]*n*)	*n* 的平均值，忽略空值
COUNT({*|[DISTINCT|ALL]*expr*})	行数，其中 *expr* 的求值为非空值（使用 * 统计所有选定行的行数，包括重复行和有空值的行）
MAX([DISTINCT|ALL]*expr*)	*expr* 的最大值，忽略空值
MIN([DISTINCT|ALL]*expr*)	*expr* 的最小值，忽略空值
STDDEV([DISTINCT|ALL]*n*)	*n* 的标准差，忽略空值
SUM([DISTINCT|ALL]*n*)	*n* 的总计值，忽略空值
VARIANCE([DISTINCT|ALL]*n*)	*n* 的方差，忽略空值

复习使用子查询

- 子查询是嵌套在另一个 SELECT 语句的子句中的一个 SELECT 语句。
- 语法：

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

- 子查询的类型：

单行子查询	多行子查询
仅返回一行	返回多行
使用单行比较运算符	使用多行比较运算符

ORACLE

复习使用子查询

通过使用子查询，可以用简单的语句构建功能强大的语句。当根据带有未知中间值的搜索标准进行查询时，使用子查询非常有帮助。

可以在许多 SQL 子句中使用子查询，其中包括以下子句：

- WHERE 子句
- HAVING 子句
- FROM 子句

子查询（内部查询）在主查询（外部查询）执行之前执行一次。主查询会使用子查询的结果。

单行子查询使用单行运算符，如 =、>、<、>=、<= 和 <> 等。多行子查询使用多行运算符，如 IN、ANY 和 ALL 等。

复习使用子查询（续）

示例：显示其薪金等于最低薪金的雇员的详细资料。

```
SELECT last_name, salary, job_id
FROM   employees
WHERE  salary = (SELECT MIN(salary)
                 FROM   employees );
```

在该示例中，MIN 组函数向外部查询返回了一个值 (2500)。

注：在本课程中，您将学习如何使用多列子查询。多列子查询通过内部 SELECT 语句返回多个列。

复习处理数据

执行以下操作时会执行数据操纵语言 (DML) 语句：

- 在表中添加新行
- 修改表中的现有行
- 从表中删除现有行

函数	说明
INSERT	在表中添加新行
UPDATE	修改表中的现有行
DELETE	从表中删除现有行
MERGE	根据条件在表中更新、插入或删除一行

ORACLE

复习处理数据

当要在数据库中添加、更新或删除数据时，就需要执行 DML 语句。构成一个逻辑工作单元的一组 DML 语句被称为一个事务处理。使用 INSERT 语句可在表中添加新行。使用以下语法时一次只能插入一行。

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

可以使用 INSERT 语句在一个表中添加一些行，其中的值来自现有表。可以使用子查询代替 VALUES 子句。INSERT 子句中列列表的列数及其数据类型必须与子查询中值的个数及其数据类型相匹配。

如果指定了 WHERE 子句，则 UPDATE 语句会修改特定的行。

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

使用 DELETE 语句可删除现有行。通过在 DELETE 语句中指定 WHERE 子句，可以删除特定行。

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

将在“处理大型数据集”一课中学习关于 MERGE 语句的知识。

课程安排

- 课程目标和课程安排
- 本课程中使用的数据库方案和附录及提供的开发环境
- 复习 SQL 的某些基本概念
- Oracle Database 11*g* 的文档和附加资源

Oracle Database 11*g* 的 SQL 文档

- Oracle Database New Features Guide 11*g* Release 2 (11.2)
- Oracle Database Reference 11*g* Release 2 (11.2)
- Oracle Database SQL Language Reference 11*g* Release 2 (11.2)
- Oracle Database Concepts 11*g* Release 2 (11.2)
- Oracle Database SQL Developer User's Guide Release 1.2

Oracle Database 11*g* 的 SQL 文档

要访问 Oracle Database 11*g* 发行版 2 文档库，请访问 http://www.oracle.com/pls/db112/homepage。

其它资源

有关 Oracle 11*g* 中新增 SQL 的其它信息，请参考以下内容：

- Oracle Database 11*g*：New Features eStudies
- Oracle by Example 系列 (OBE)：Oracle Database 11*g*

小结

在本课中，您应该已经掌握了下列内容：

- 课程目标
- 课程中使用的示例表

练习 I：概览

本练习包含以下主题：

- 运行 SQL Developer 联机教程
- 启动 SQL Developer、创建新数据库连接并浏览表
- 使用 SQL 工作表执行 SQL 语句
- 复习 SQL 的基本概念

ORACLE

练习 I：概览

在本练习中，您将使用 SQL Developer 执行 SQL 语句。

注：所有书面练习都使用 SQL Developer 作为开发环境。虽然建议使用 SQL Developer，但也可以使用本课程中提供的 SQL*Plus 环境。

1

控制用户访问

课程目标

学完本课后，应能完成下列工作：

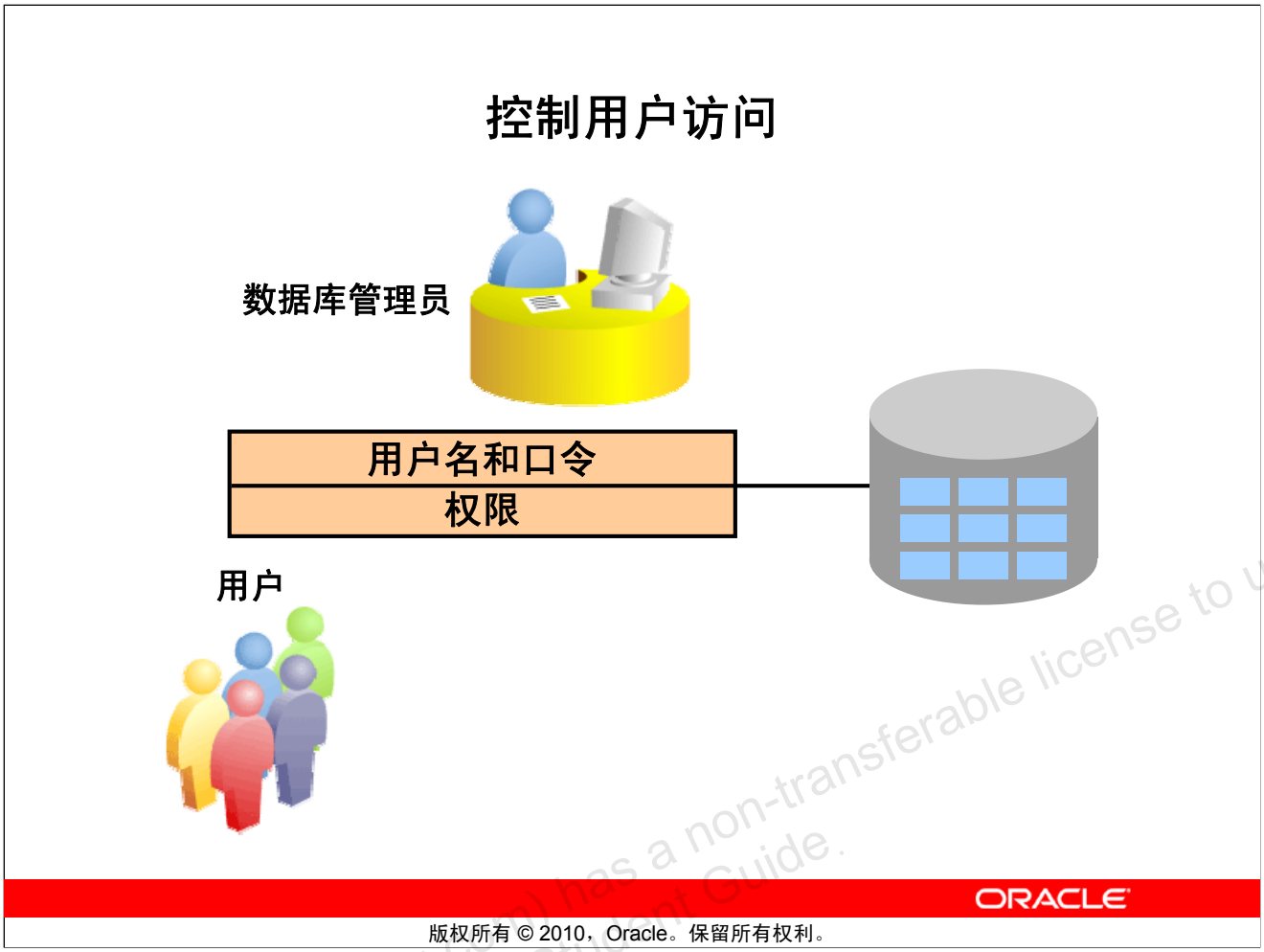
- 区分系统权限与对象权限
- 授予表权限
- 授予角色
- 区别权限与角色

课程目标

在本课中，您将学习如何控制对数据库特定对象的访问，以及如何添加拥有不同级别访问权限的新用户。

课程安排

- 系统权限
- 创建角色
- 对象权限
- 撤消对象权限



控制用户访问

在多用户环境下，您要维护访问及使用数据库的安全性。借助 Oracle Server 数据库的安全性，可以完成以下任务：

- 控制数据库访问。
- 授予对数据库中特定对象的访问权限。
- 使用 Oracle 数据字典确认给定的权限和得到的权限。

数据库安全性可分为以下两类：系统安全性和数据安全性。系统安全性包括在系统级别对数据库进行的访问与使用，如用户名和口令、分配给用户的磁盘空间以及用户可执行的系统操作。数据库安全性包括对数据库对象的访问和使用以及用户可对对象执行的操作。

权限

- 数据库安全性：
  - 系统安全性
  - 数据安全性
- 系统权限：在数据库中执行特定操作
- 对象权限：处理数据库对象的内容
- 方案：诸如表、视图和序列等对象的集合

权限

权限就是执行特定 SQL 语句的权利。数据库管理员 (DBA) 是高级用户，他能创建用户，还能授予用户对数据库及其对象的访问权限。用户需要拥有系统权限才能访问数据库，需要拥有对象权限才能处理数据库对象的内容。还可以为用户授予权限，使其可向其他用户或其它角色授予其它权限。所谓角色就是一组命名的相关权限。

方案

方案是诸如表、视图和序列等对象的集合。方案归数据库用户所有，而且与该用户同名。

系统权限是指执行特定操作的权利，或对特定类型的方案对象执行操作的权利。具有对象权限的用户能对特定的方案对象执行特定的操作。

有关详细信息，请参阅《Oracle Database 2 Day DBA 11*g* Release 2 (11.2)》参考手册。

系统权限

- 可用的系统权限超过 100 个。
- 数据库管理员拥有执行诸如下列任务的高级系统权限：
  - 创建新用户
  - 删除用户
  - 删除表
  - 备份表

系统权限

有 100 多个不同的系统权限可供用户和角色使用。通常情况下，由数据库管理员 (DBA) 提供系统权限。

常见的 DBA 权限

系统权限	授权的操作
CREATE USER	被授予者可以创建其他 Oracel 用户。
DROP USER	被授予者可以删除其他用户。
DROP ANY TABLE	被授予者可以删除任何方案中的表。
BACKUP ANY TABLE	被授予者可以使用导出实用程序备份任何方案中的任何表。
SELECT ANY TABLE	被授予者可以查询任何方案中的表、视图、或实体化视图。
CREATE ANY TABLE	被授予者可以创建任何方案中的表。

创建用户

DBA 可使用 CREATE USER 语句创建用户。

```
CREATE USER user
IDENTIFIED BY   password;
```

```
CREATE USER  demo
IDENTIFIED BY demo;
```


创建用户

DBA 可通过执行 CREATE USER 语句创建用户。此时，用户还没有任何权限。随后 DBA 可为用户授予权限。这些权限决定了用户在数据库级别可执行的操作。

幻灯片中显示创建用户时使用的简单语法。

在该语法中：

user	是要创建用户的名称
Password	指定用户必须使用此口令登录

有关详细信息，请参阅《Oracle Database11g SQL Reference》。

注：启动 Oracle Database 11g 时，口令是区分大小写的。

用户系统权限

- 创建用户之后，DBA 可以为用户授予特定的系统权限。

```
GRANT privilege [, privilege...]
TO user [, user| role, PUBLIC...];
```

- 例如，应用程序开发人员可拥有以下系统权限：
  - CREATE SESSION
  - CREATE TABLE
  - CREATE SEQUENCE
  - CREATE VIEW
  - CREATE PROCEDURE

常见的用户权限

DBA 创建用户之后，就可为用户分配权限。

系统权限	授权的操作
CREATE SESSION	连接到数据库。
CREATE TABLE	在用户方案中创建表。
CREATE SEQUENCE	在用户方案中创建序列。
CREATE VIEW	在用户方案中创建视图。
CREATE PROCEDURE	在用户方案中创建存储过程、函数或程序包。

常见的用户权限（续）

在该语法中：

privilege	是要授予的系统权限
user |role|PUBLIC	是用户名、角色名或 PUBLIC（表示此权限要授予所有用户）

注：在 SESSION_PRIVS 字典视图中可找到当前系统权限。数据字典是在 Oracle Server 中创建并维护的表和视图的集合。这些表和视图包含关于数据库的信息。

授予系统权限

DBA 可为用户授予特定的系统权限。

```
GRANT  create session, create table,
       create sequence, create view
TO     demo;
```

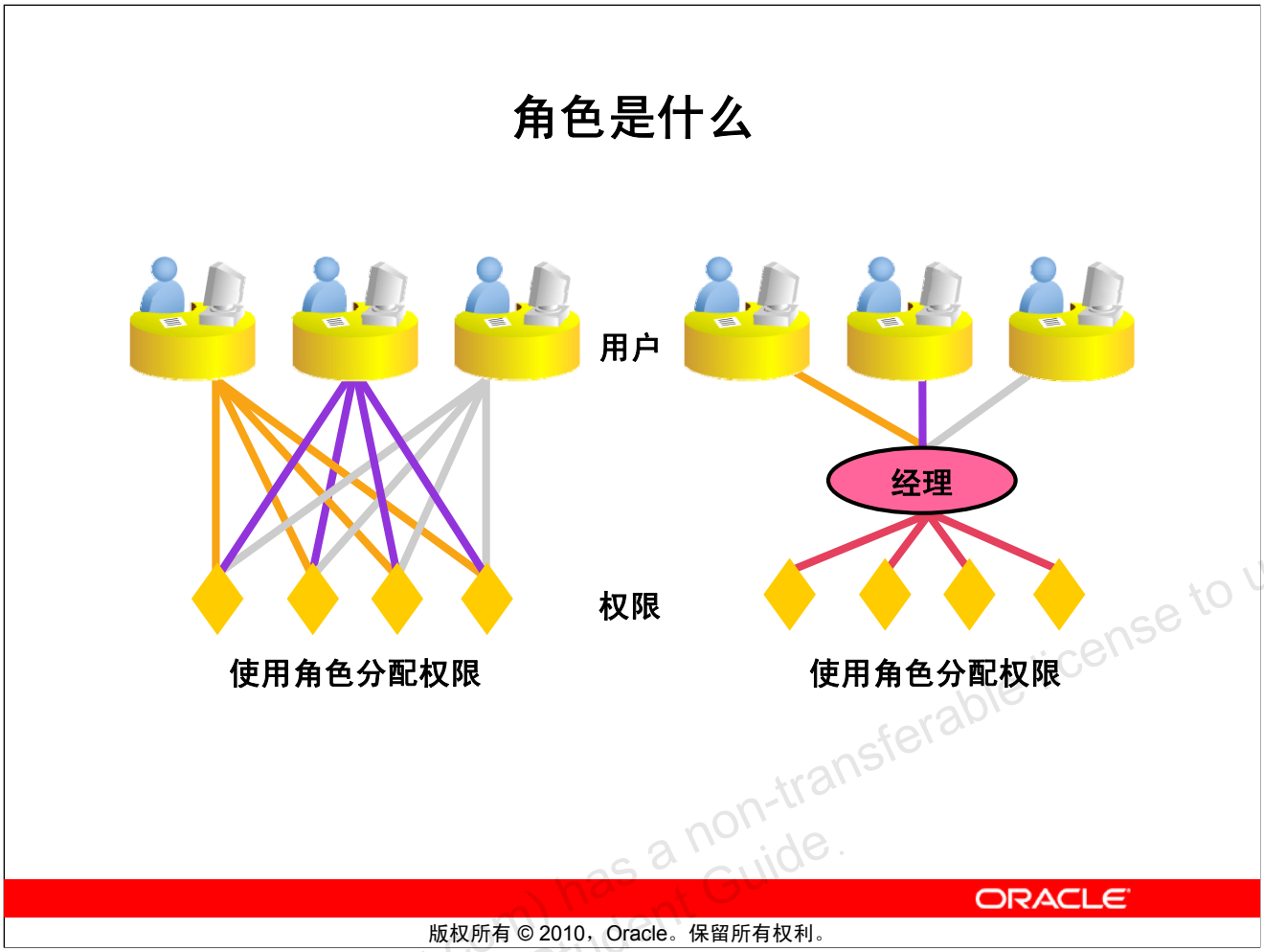

授予系统权限

DBA 使用 GRANT 语句可为用户分配系统权限。将某些权限授予用户之后，用户就可以立即使用这些权限。

在幻灯片示例中，为 demo 用户授予了创建会话、表、序列和视图的权限。

课程安排

- 系统权限
- 创建角色
- 对象权限
- 撤消对象权限



角色是什么

角色是一组命名的可授予用户的相关权限。此方法使撤消和维护权限变得更容易。

一个用户可访问几个角色，同一角色也可分配给几个用户。角色通常是为数据库应用程序创建的。

创建和分配角色

首先，必须由 DBA 创建角色。然后，DBA 可将权限分配给角色，再将角色分配给用户。

语法

```
CREATE   ROLE role;
```

在该语法中：

role　　是要创建角色的名称

创建角色之后，DBA 使用 GRANT 语句既可将角色分配给用户，又可将权限分配给角色。角色不是方案对象，因此任何用户都可以向对象添加权限。

创建角色和为角色授予权限

- 创建角色：

```
CREATE ROLE manager;
```

- 为角色授予权限：

```
GRANT create table, create view
TO manager;
```

- 将角色授予用户：

```
GRANT manager TO alice;
```

ORACLE

创建角色

幻灯片示例中创建了一个经理角色并使经理能够创建表和视图。然后其向用户 Alice 授予经理的角色。现在，Alice 可以创建表和视图了。

如果为某些用户授予多个角色，这些用户就会得到与所有这些角色相关联的所有权限。

更改口令

- DBA 负责创建用户帐户和设置最初的口令。
- 您可以使用 ALTER USER 语句更改口令。

```
ALTER USER demo
IDENTIFIED BY employ;
```


更改口令

DBA 负责为每个用户创建帐户和设置最初的口令。您可以使用 ALTER USER 语句更改口令。

幻灯片示例中显示 demo 用户使用 ALTER USER 语句更改了口令。

语法

```
ALTER USER user IDENTIFIED BY password;
```

在该语法中：

user	是用户名
password	指定新口令

使用此语句不仅可以更改口令，还可以更改很多其它选项。必须拥有 ALTER USER 权限，才能更改任何其它选项。

有关详细信息，请参阅《Oracle Database11*g* SQL Reference》手册。

注：SQL*Plus 提供的 PASSWORD 命令 (PASSW) 可用来在用户登录时更改用户的口令，但在 SQL Developer 中不能使用该命令。

课程安排

- 系统权限
- 创建角色
- 对象权限
- 撤消对象权限

对象权限

对象权限	表	视图	序列
ALTER	✓		✓
DELETE	✓	✓	
INDEX	✓		
INSERT	✓	✓	
REFERENCES	✓		
SELECT	✓	✓	✓
UPDATE	✓	✓	

ORACLE

对象权限

对象权限是指对特定表、视图、序列或过程执行特定操作的权限或权利。每个对象都有一组特定的可授予权限。幻灯片的表中列出了各种对象的权限。请注意，适用于序列的权限只有 SELECT 和 ALTER。通过指定可更新列的子集，可以限制 UPDATE、REFERENCES 和 INSERT 的范围。

要限制 SELECT 权限的范围，可创建一个视图来包含一个列子集，然后只在该视图上授予 SELECT 权限。对同义词授予的权限会转换为对同义词所引用基表的权限。

注：拥有 REFERENCES 权限时，您就可以确保其他用户可创建引用您的表的 FOREIGN KEY 约束条件。

对象权限

- 对象权限因对象而异。
- 所有者拥有对象的全部权限。
- 所有者可将自己对象的特定权限授予其他用户。

```
GRANT       object_priv [(columns)]
ON          object
TO          {user|role|PUBLIC}
[WITH GRANT OPTION];
```


授予对象权限

不同的对象权限适用于不同类型的方案对象。用户会自动拥有用户方案中包含的对象的全部权限。用户可以将自己拥有的任何方案对象的任何对象权限授予其他用户或角色。如果授予语句中包括 WITH GRANT OPTION，则被授予者可再将对象权限授予其他用户；否则，被授予者只能使用对象权限，而不能将对象权限授予其他用户。

在该语法中：

object_priv	是要授予的对象权限
ALL	指定所有对象权限
columns	指定已授予权限的表或视图中的列
ON *object*	是要将其权限授予用户的对象
TO	标识要将权限授予谁
PUBLIC	将对象权限授予所有用户
WITH GRANT OPTION	允许被授予者将对象权限授予其他用户和角色

注：在该语法中，*schema* 与所有者同名。

授予对象权限

- 授予对 EMPLOYEES 表的查询权限：

```
GRANT  select
ON     employees
TO     demo;
```

- 为用户和角色授予更新特定列的权限：

```
GRANT  update (department_name, location_id)
ON     departments
TO     demo, manager;
```


准则

- 要授予一个对象的权限，该对象必须在您自己的方案中，或者使用 WITH GRANT OPTION 给您授予了对象权限。
- 一个对象所有者可将该对象的任何对象权限授予数据库的任何其他用户或角色。
- 对象所有者会自动获得对象的所有对象权限。

幻灯片第一个示例中为 demo 用户授予了查询 EMPLOYEES 表的权限。第二个示例中给 demo 和 manager 角色授予了对 DEPARTMENTS 表中的特定列执行 UPDATE 的权限。

例如，如果方案为 oraxx，现在 demo 用户要使用 SELECT 语句获取 EMPLOYEES 表中的数据，那么该用户必须使用以下语法：

```
SELECT  * FROM oraxx.employees;
```

或者，demo 用户可以创建表的同义词，然后使用同义词发布 SELECT 语句：

```
CREATE SYNONYM emp FOR oraxx.employees;
SELECT * FROM emp;
```

注：通常由 DBA 分配系统权限，由拥有对象的任何用户授予对象权限。

传递权限

- 授予用户权限来传递权限：

```
GRANT  select, insert
ON     departments
TO     demo
WITH   GRANT OPTION;
```

- 允许系统所有用户查询 Alice 的 DEPARTMENTS 表中的数据：

```
GRANT  select
ON     alice.departments
TO     PUBLIC;
```


传递权限

WITH GRANT OPTION 关键字

使用 WITH GRANT OPTION 子句授予的权限可由被授予者传递给其他用户和角色。撤消授予者的权限后，会随之撤销使用 WITH GRANT OPTION 子句授予的对象权限。

幻灯片示例中授予 demo 用户对 DEPARATMENT 表的访问权限，以及查询表和在表中添加行的权限。示例中还允许 demo 用户将这些权限授予其他用户。

PUBLIC 关键字

表的所有者可以使用 PUBLIC 关键字将访问权限授予所有用户。

第二个示例允许系统中的所有用户查询 Alice 的 DEPARTMENTS 表中的数据。

确认授予的权限

数据字典视图	说明
ROLE_SYS_PRIVS	授予角色的系统权限
ROLE_TAB_PRIVS	授予角色的表权限
USER_ROLE_PRIVS	用户可以访问的角色
USER_SYS_PRIVS	授予用户的系统权限
USER_TAB_PRIVS_MADE	在用户对象上授予的对象权限
USER_TAB_PRIVS_RECD	授予用户的对象权限
USER_COL_PRIVS_MADE	在用户对象列上授予的对象权限
USER_COL_PRIVS_RECD	在特定列上授予用户的对象权限

ORACLE

确认授予的权限

当尝试执行未授权的操作时，例如，要在没有 DELETE 权限的表中删除一行时，Oracle Server 不会允许执行此操作。

如果收到了 Oracle Server 错误消息“Table or view does not exist（表或视图不存在）”，则表明可能遇到了以下一种情况：

- 指定的表或视图不存在
- 尝试对没有适当权限的表或视图执行某项操作

数据字典是以表和视图的方式组织的，其中包含关于数据库的信息。您可以通过访问数据字典查看所拥有的权限。幻灯片的表中列出了各种数据字典视图。

在“使用数据字典视图管理对象”一课中将更详细地介绍关于数据字典视图的内容。

注：在 ALL_TAB_PRIVS_MADE 字典视图中描述了由用户授予的所有对象授权或在用户拥有对象上授予的所有对象授权。

课程安排

- 系统权限
- 创建角色
- 对象权限
- 撤消对象权限

撤消对象权限

- 使用 REVOKE 语句可撤消已授予其他用户的权限。
- 还会撤消通过 WITH GRANT OPTION 子句授予其他用户的权限。

```
REVOKE {privilege [, privilege...]|ALL}
ON     object
FROM   {user[, user...]|role|PUBLIC}
[CASCADE CONSTRAINTS];
```


撤消对象权限

使用 REVOKE 语句可取消已授予其他用户的权限。使用 REVOKE 语句时，会撤消指定用户的指定权限；如果某些用户的指定权限是由被撤销用户授予的，还会撤消这些用户的指定权限。

在该语法中：

CASCADE	是一个必需项，可用来取消使用 REFERENCES 权限对 CONSTRAINTS 对象实施的任何引用完整性约束条件

有关详细信息，请参阅《Oracle Database 11*g* SQL Reference》。

注：如果某个用户离开公司后其权限被撤消，则必须重新授予该用户可能已授予其他用户的所有权限。如果只是删除了用户帐户而没有撤消其权限，那么删除操作不会影响由该用户授予其他用户的系统权限。

撤消对象权限

撤消授予 demo 用户在 DEPARTMENTS 表上的 SELECT 和 INSERT 权限。

```
REVOKE  select, insert
ON      departments
FROM    demo;
```


撤消对象权限（续）

幻灯片示例中撤消了授予 demo 用户在 DEPARTMENTS 表上的 SELECT 和 INSERT 权限。

注：如果使用 WITH GRANT OPTION 子句为一个用户授予了权限，则该用户也可以使用 WITH GRANT OPTION 子句授予权限，因此可能存在一长串的被授予者，但是不允许循环授权（即将权限授予最初的授予者）。如果所有者撤消了某个用户的某权限，而该用户已将该权限授予其他用户，则会通过级联方式撤消该权限。

例如，当用户 A 使用 WITH GRANT OPTION 子句将表的 SELECT 权限授予用户 B 时，用户 B 可以使用 WITH GRANT OPTION 子句将 SELECT 权限授予给用户 C，然后，用户 C 可以将 SELECT 权限授予用户 D。当用户 A 撤消了授予用户 B 的权限时，还会撤销授予用户 C 和 D 的权限。

小测验

以下哪句陈述是正确的？

1. 用户创建对象之后，就可以使用 GRANT 语句将任何可用的对象权限传递给其他用户。
2. DBA 可以使用 CREATE ROLE 语句创建角色，以便将一组系统权限或对象权限传递给其他用户。
3. 用户可以更改其自己的口令。
4. 用户可以查看授予给自己的权限以及其对象上已授予的权限。

答案：1、3、 4

小结

在本课中，您应该已经学会：

- 区分系统权限与对象权限
- 授予表权限
- 授予角色
- 区别权限与角色

小结

DBA 通过为用户分配权限，为用户设置了最初的数据库安全设置。

- DBA 创建的用户必须拥有一个口令。DBA 还负责为用户设置最初的系统权限。
- 用户创建对象之后，就可以使用 GRANT 语句将任何可用的对象权限传递给其他用户或所有用户。
- DBA 可以使用 CREATE ROLE 语句创建角色，以便将一组系统权限或对象权限传递给多个用户。角色使授予权限和撤消权限变得更容易。
- 用户可以使用 ALTER USER 语句更改口令。
- 您可以使用 REVOKE 语句删除用户的权限。
- 借助数据字典视图，用户可以查看授予给自己的权限以及其对象上已授予的权限。

练习 1：概览

本练习包含以下主题：

- 为其他用户授予对您的表的权限
- 使用授予给您的权限来修改其他用户的表

ORACLE

练习 1：概览

此练习需要与其他学员合作，才能了解如何控制对数据库对象的访问。

2

管理方案对象

课程目标

学完本课后，应能完成下列工作：

- 添加约束条件
- 创建索引
- 使用 CREATE TABLE 语句创建索引
- 创建基于函数的索引
- 删除列并将列设置为 UNUSED
- 执行 FLASHBACK 操作
- 创建和使用外部表

课程目标

本课包含关于创建索引、约束条件和更改现有对象的信息。您还将了解外部表，以及创建 PRIMARY KEY 约束条件时命名索引的规定。

课程安排

- 使用 ALTER TABLE 语句添加、修改或删除列
- 管理约束条件：
  - 添加和删除约束条件
  - 延迟约束条件
  - 启用和禁用约束条件
- 创建索引：
  - 使用 CREATE TABLE 语句
  - 创建基于函数的索引
  - 删除索引
- 执行闪回操作
- 创建和使用临时表
- 创建和使用外部表

ORACLE

ALTER TABLE 语句

使用 ALTER TABLE 语句可：

- 添加新列
- 修改现有列
- 定义新列的默认值
- 删除列

ALTER TABLE 语句

创建表之后，您可能因为漏掉一列、需要更改列定义或需要删除一些列，而需要更改表结构。可以使用 ALTER TABLE 语句完成此任务。

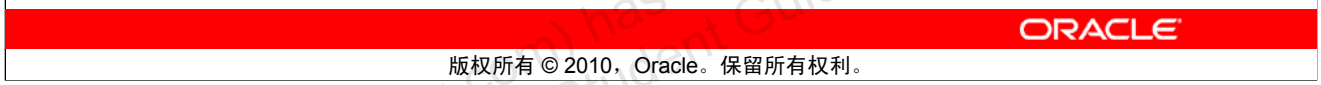
ALTER TABLE 语句

使用 ALTER TABLE 语句可添加、修改或删除列：

```
ALTER TABLE table
ADD         (column datatype [DEFAULT expr]
            [, column datatype]...);
```

```
ALTER TABLE table
MODIFY      (column datatype [DEFAULT expr]
            [, column datatype]...);
```

```
ALTER TABLE table
DROP (column [, column] …);
```



ALTER TABLE 语句（续）

可以使用 ALTER TABLE 语句在表中添加列、修改列和删除列。

在该语法中：

table	是表名称
ADD|MODIFY|DROP	是修改类型
column	是列名称
datatype	是列的数据类型和长度
DEFAULT *expr*	指定列默认值

添加列

- 使用 ADD 子句可添加列：

```
ALTER TABLE dept80
ADD         (job_id VARCHAR2(9));
```

```
ALTER TABLE dept80 succeeded.
```

- 新列会变为最后一列：

	EMPLOYEE_ID	LAST_NAME	ANNSAL	HIRE_DATE	JOB_ID
1	145	Russell	14000	01-OCT-96	(null)
2	146	Partners	13500	05-JAN-97	(null)
3	147	Errazuriz	12000	10-MAR-97	(null)
4	148	Cambrault	11000	15-OCT-99	(null)
5	149	Zlotkey	10500	29-JAN-00	(null)

ORACLE

添加列的准则

- 可以添加列或修改列。
- 不能指定列的显示位置。新列会变为最后一列。

幻灯片示例在 DEPT80 表中添加了名为 JOB_ID 的一列。JOB_ID 列是该表中的最后一列。

注：如果添加列时表中已有若干行，则所有行上的新列中最初都为空或者采用默认值。只有指定了默认值时，才能在其它列含有数据的表中添加强制实施 NOT NULL 的列。没有指定默认值时，可在空表中添加 NOT NULL 列。

修改列

- 可以更改列的数据类型、大小和默认值。

```
ALTER TABLE dept80
MODIFY      (last_name VARCHAR2(30));
```

```
ALTER TABLE dept80 succeeded.
```

- 更改默认值只会影响后来在表中插入的内容。

修改列

可以使用包含 MODIFY 子句的 ALTER TABLE 语句修改列定义。修改列时可以对列的数据类型、大小和默认值进行更改。

准则

- 可以增加数字列的宽度或精度。
- 可以增加字符列的宽度。
- 在下列情况下可以减小列的宽度：
  - 列只包含空值
  - 表中没有行
  - 减小的列宽不小于列中现有值的宽度
- 您可以在列只含空值时更改数据类型。但 CHAR 到 VARCHAR2 的转换是个例外，这种情况下即使列含有数据，也可以更改数据类型。
- 只有列含空值或只有不更改大小时，才能将 CHAR 列转换为 VARCHAR2 数据类型，或将 VARCHAR2 列转换为 CHAR 数据类型。
- 更改列默认值只会影响后来在表中插入的内容。

删除列

使用 DROP COLUMN 子句可从表中删除不再需要的列：

```
ALTER TABLE  dept80
DROP COLUMN  job_id;
```

```
ALTER TABLE dept80 succeeded.
```

	EMPLOYEE_ID	LAST_NAME	ANNSAL	HIRE_DATE
1	145	Russell	14000	01-OCT-96
2	146	Partners	13500	05-JAN-97
3	147	Errazuriz	12000	10-MAR-97
4	148	Cambrault	11000	15-OCT-99
5	149	Zlotkey	10500	29-JAN-00

ORACLE

删除列

可以使用包含 DROP COLUMN 子句的 ALTER TABLE 语句从表中删除列。

准则

- 列可以包含数据，也可以不包含数据。
- 使用 ALTER TABLE DROP COLUMN 语句，一次只能删除一列。
- 表在更改后，必须至少剩余一列。
- 删除一列之后，无法恢复该列。
- 如果列是约束条件或索引关键字的一部分，则不能删除它，除非添加了级联选项。
- 如果列包含大量的值，则删除列会花费一些时间。在此情况下，最好将列设置为“未使用”，然后在系统中用户很少时删除列，这可避免扩大锁定的范围。

注：某些列是不能删除的，如作为分区表分区关键字一部分的那些列，或作为索引表 PRIMARY KEY 一部分的那些列。有关索引表和分区表的详细信息，请参阅《Oracle Database 概念》和《Oracle Database 管理员指南》。

SET UNUSED 选项

- 使用 SET UNUSED 选项可将一列或多列标记为“未使用”。
- 使用 DROP UNUSED COLUMNS 选项可删除已标记为“未使用”的列。

```
ALTER TABLE  <table_name>
SET   UNUSED(<column_name> [ , <column_name>]);
或
ALTER TABLE  <table_name>
SET   UNUSED COLUMN <column_name> [,<column_name>];
```

```
ALTER TABLE <table_name>
DROP  UNUSED COLUMNS;
```

ORACLE

SET UNUSED 选项

使用 SET UNUSED 选项可将一列或多列标记为“未使用”，这样可在对系统资源的需求变低时再删除这些列。指定此子句并没有真正从表的每一行中删除目标列，也就是说，并不会还原这些列所占用的磁盘空间。因此，响应时间比执行 DROP 子句所需的时间要短。“未使用”列被视为已删除列，即使其列数据仍保留在表行中。某一列被标记为“未使用”后，就无法访问该列。执行 SELECT * 查询时将不会检索“未使用”列中的数据。另外，执行 DESCRIBE 语句时不会显示标记为“未使用”的列的名称和类型，因而您可以在表中添加一个名称与“未使用”列同名的新列。在 USER_UNUSED_COL_TABS 字典视图中存储了 SET UNUSED 信息。

注：将列设置为 UNUSED 的准则与删除列的准则类似。

SET UNUSED 选项（续）

DROP UNUSED COLUMNS 选项

DROP UNUSED COLUMNS 会从表中删除当前标记为“未使用”的所有列。如果要回收表中“未使用”列所占用的额外磁盘空间，则可以使用此语句。如果表不包含“未使用”列，则该语句不会返回任何错误。

```
ALTER TABLE  dept80
SET   UNUSED (last_name);

ALTER TABLE succeeded


ALTER TABLE  dept80
DROP  UNUSED COLUMNS;

ALTER TABLE succeeded
```

课程安排

- 使用 ALTER TABLE 语句添加、修改或删除列
- 管理约束条件：
  - 添加和删除约束条件
  - 延迟约束条件
  - 启用和禁用约束条件
- 创建索引：
  - 使用 CREATE TABLE 语句
  - 创建基于函数的索引
  - 删除索引
- 执行闪回操作
- 创建和使用临时表
- 创建和使用外部表

ORACLE

添加约束条件语法

使用 ALTER TABLE 语句可：

- 添加或删除约束条件，但不修改其结构
- 启用或禁用约束条件
- 使用 MODIFY 子句添加 NOT NULL 约束条件

```
ALTER TABLE  <table_name>
ADD [CONSTRAINT <constraint_name>]
type (<column_name>);
```


添加约束条件

可以使用包含 ADD 子句的 ALTER TABLE 语句来为现有表添加约束条件。

在该语法中：

table	是表名称
constraint	是约束条件的名称
type	是约束条件类型
column	是受约束条件影响的列的名称

虽然建议使用约束条件命名语法，但该命名语法是可选项。如果没有为约束条件命名，系统会生成约束条件名称。

准则

- 可以添加、删除、启用或禁用约束条件，但是不能修改其结构。
- 在 ALTER TABLE 语句中使用 MODIFY 子句可在现有列中添加 NOT NULL 约束条件。

注：只有表为空或某列在每行上都有值时，才能将该列定义为 NOT NULL 列。

添加约束条件

在 EMP2 表中添加一个 FOREIGN KEY 约束条件来指明经理必须作为效雇员存在于 EMP2 表中。

```
ALTER TABLE emp2
MODIFY employee_id PRIMARY KEY;
```

```
ALTER TABLE emp2 succeeded.
```

```
ALTER TABLE emp2
ADD CONSTRAINT emp_mgr_fk
  FOREIGN KEY(manager_id)
  REFERENCES emp2(employee_id);
```

```
ALTER TABLE  succeeded.
```

ORACLE

添加约束条件（续）

幻灯片第一个示例中修改了 EMP2 表，目的是为了在 EMPLOYEE_ID 列上添加一个 PRIMARY KEY 约束条件。请注意，由于没有提供约束条件名称，所以 Oracle Server 会自动命名约束条件。幻灯片第二个示例中在 EMP2 表上创建了一个 FOREIGN KEY 约束条件。该约束条件确保经理以作为有效雇员存在于 EMP2 表中。

ON DELETE 子句

- 在删除父关键字后，使用 ON DELETE CASCADE 子句删除子行：

```
ALTER TABLE emp2 ADD CONSTRAINT emp_dt_fk
FOREIGN KEY (Department_id)
REFERENCES departments(department_id) ON DELETE CASCADE;
```

```
ALTER TABLE Emp2 succeeded.
```

- 在删除父关键字后，使用 ON DELETE SET NULL 子句将子行值设置为空值：

```
ALTER TABLE emp2 ADD CONSTRAINT emp_dt_fk
FOREIGN KEY (Department_id)
REFERENCES departments(department_id) ON DELETE SET NULL;
```

```
ALTER TABLE Emp2 succeeded.
```

ORACLE

ON DELETE

如果删除引用的主键值或唯一关键字值，通过使用 ON DELETE 子句，可以确定 Oracle DB 处理引用完整性的方式。

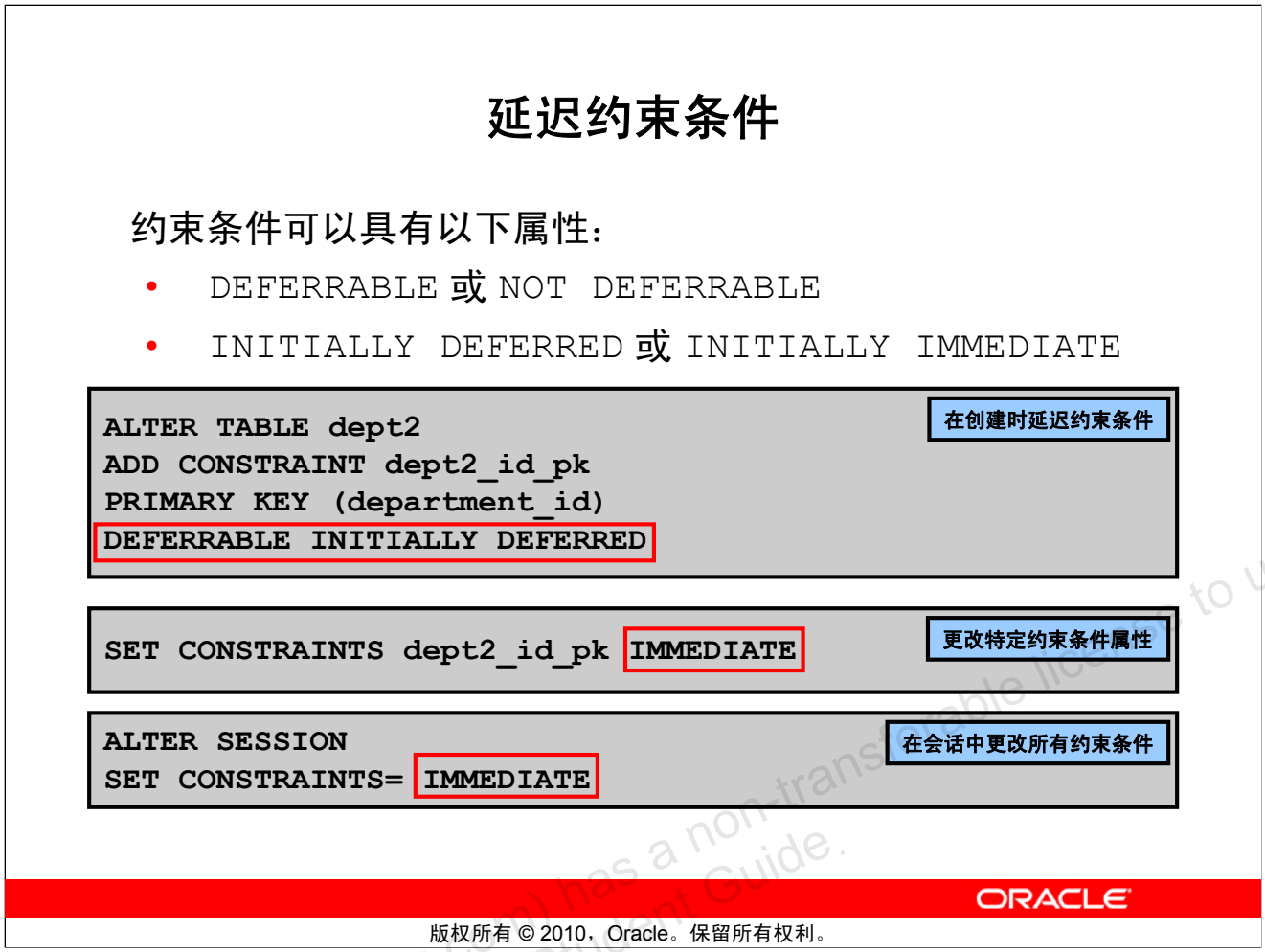
ON DELETE CASCADE

执行 ON DELETE CASCADE 操作时允许删除子表中引用的父关键字数据，但不允许进行更新。删除父关键字数据后，还会删除子表中与被删除的父关键字值相关的所有行。要指定这个引用操作，请在 FOREIGN KEY 约束条件的定义中包括 ON DELETE CASCADE 选项。

ON DELETE SET NULL

删除父关键字中的数据后，ON DELETE SET NULL 操作将导致子表中所有依赖于此已删除父关键字值的行转换为空值。

如果省略此子句，Oracle 将不允许删除父表中在子表内具有相关行的引用键值。



延迟约束条件

可以将检查约束条件的有效性延迟到事务处理结束时再进行。如果系统不检查是否满足了约束条件，则约束条件将一直受到延迟直至提交 COMMIT 语句。如果违反了延迟的约束条件，则数据库将返回一条错误，事务处理不会提交而是被回退。如果约束条件是立即实施（而不是延迟），则会在每条语句结束时检查约束条件。如果违反了这种约束条件，则会立即回退该语句。如果约束条件导致一个操作（如 DELETE CASCADE），则无论约束条件是延迟还是立即实施，该操作都会作为引起该操作的语句的一部分而执行。对于特殊事务处理，可以使用 SET CONSTRAINTS 语句指定是在每条数据操纵语言 (DML) 语句之后还是在提交事务处理时检查可延迟约束条件。要创建可延迟约束条件，必须为该约束条件创建一个非唯一索引。

可以将约束条件定义为可延迟或不可延迟，以及最初延迟或最初立即实施。每个约束条件可以有上述各种属性。

使用方案：公司政策规定，部门编号 40 应更改为 45。更改 DEPARTMENT_ID 列后，对分配到该部门的雇员会有影响。因此，可将 PRIMARY KEY 和 FOREIGN KEY 设置为可延迟且最初延迟。如果您更新了部门和雇员的信息，在提交时就会验证所有行。

INITIALLY DEFERRED 与 INITIALLY IMMEDIATE 之间的区别

INITIALLY DEFERRED	一直等待，直到事务处理结束时再检查约束条件
INITIALLY IMMEDIATE	在语句执行结束时检查约束条件

```
CREATE TABLE emp_new_sal (salary NUMBER
        CONSTRAINT sal_ck
        CHECK (salary > 100)
        DEFERRABLE INITIALLY IMMEDIATE,
        bonus NUMBER
        CONSTRAINT bonus_ck
        CHECK (bonus > 0 )
        DEFERRABLE INITIALLY DEFERRED );
```

```
create table succeeded.
```

ORACLE

INITIALLY DEFERRED 与 INITIALLY IMMEDIATE 之间的区别

定义为可延迟的约束条件可指定为 INITIALLY DEFERRED 或 INITIALLY IMMEDIATE。INITIALLY IMMEDIATE 子句是默认值。

在幻灯片示例中：

- sal_ck 约束条件创建时指定为 DEFERRABLE INITIALLY IMMEDIATE
- bonus_ck 约束条件创建时指定为 DEFERRABLE INITIALLY DEFERRED

创建 emp_new_sal 表（如幻灯片所示）以后，您可以尝试在该表中插入值，然后观察结果。如果 sal_ck 和 bonus_ck 两个约束条件都满足，则插入行时不会发生错误。

示例 1：插入违反 sal_ck 的一行。在 CREATE TABLE 语句中，sal_ck 被指定为最初立即实施的约束条件。这意味着在执行 INSERT 语句后会立即验证该约束条件，因而您会看到一条错误。

```
INSERT INTO emp_new_sal VALUES(90,5);
```

```
SQL Error: ORA-02290: check constraint (ORA21.SAL_CK) violated
02290. 00000 -  "check constraint (%s.%s) violated"
```

INITIALLY DEFERRED 与 INITIALLY IMMEDIATE 之间的区别（续）

示例 2：插入违反 bonus_ck 的一行。在 CREATE TABLE 语句中，bonus_ck 被指定为可延迟而且是最初已延迟的约束条件。因此，不会在执行 COMMIT 之前或将约束条件状态重新设置为立即实施之前验证该约束条件。

```
INSERT INTO emp_new_sal VALUES(110, -1);
```

```
1 rows inserted
```

行插入操作成功。但是，在提交事务处理时会看到一条错误。

```
COMMIT;
```

```
SQL Error: ORA-02091: transaction rolled back
ORA-02290: check constraint (ORA21.BONUS_CK) violated
02091. 00000 -  "transaction rolled back"
```

提交失败的原因是违反了约束条件。因此，此时数据库会回退事务处理。

示例 3：将可延迟的所有约束条件的状态都设置为 DEFERRED。请注意，也可以根据需要将一个约束条件的状态设置为 DEFERRED。

```
SET CONSTRAINTS ALL DEFERRED;
```

```
SET CONSTRAINTS succeeded.
```

现在，如果试图插入违反 sal_ck 约束条件的一行，则会成功执行语句。

```
INSERT INTO emp_new_sal VALUES(90,5);
```

```
1 rows inserted
```

但是，在提交事务处理时会看到一条错误。事务处理失败且被回退。这是因为执行 COMMIT 时会检查两个约束条件。

```
COMMIT;
```

```
SQL Error: ORA-02091: transaction rolled back
ORA-02290: check constraint (ORA21.SAL_CK) violated
02091. 00000 -  "transaction rolled back"
```

示例 4：将在上个示例中状态已设置为 DEFERRED 的两个约束条件的状态都设置为 IMMEDIATE。

```
SET CONSTRAINTS ALL IMMEDIATE;
```

```
SET CONSTRAINTS succeeded.
```

如果试图插入违反 sal_ck 约束条件或 bonus_ck 约束条件的一行，则会看到一条错误。

```
INSERT INTO emp_new_sal VALUES(110, -1);
```

```
SQL Error: ORA-02290: check constraint (ORA21.BONUS_CK) violated
02290. 00000 -  "check constraint (%s.%s) violated"
```

注：如果创建了表但没有指定约束条件是否延迟，则会在每条语句结束时立即检查约束条件。例如，使用 CREATE TABLE 语句创建 newemp_details 表时，如果没有指定 newemp_det_pk 约束条件是否延迟，则会立即检查该约束条件。

```
CREATE TABLE newemp_details(emp_id NUMBER, emp_name
VARCHAR2(20),
CONSTRAINT newemp_det_pk PRIMARY KEY(emp_id));
```

如果试图延迟不可延迟的 newemp_det_pk 约束条件，则会看到以下错误：

```
SET CONSTRAINT newemp_det_pk DEFERRED;
```

```
SQL Error: ORA-02447: cannot defer a constraint that is not deferrable
```

删除约束条件

- 从 EMP2 表中删除经理约束条件：

```
ALTER TABLE emp2
DROP CONSTRAINT emp_mgr_fk;
```

```
ALTER TABLE Emp2 succeeded.
```

- 删除 DEPT2 表上的 PRIMARY KEY 约束条件并删除 EMP2.DEPARTMENT_ID 列上的相关 FOREIGN KEY 约束条件：

```
ALTER TABLE dept2
DROP PRIMARY KEY CASCADE;
```

```
ALTER TABLE dept2 succeeded.
```

ORACLE

删除约束条件

要删除约束条件，可以在 USER_CONSTRAINTS 和 USER_CONS_COLUMNS 数据字典视图中确定约束条件名。然后使用包含 DROP 子句的 ALTER TABLE 语句。如果在 DROP 子句中使用 CASCADE 选项，还会删除所有相关的约束条件。

语法

```
ALTER TABLE   table
 DROP   PRIMARY KEY | UNIQUE (column) |
        CONSTRAINT    constraint   [CASCADE];
```

在该语法中：

table	是表名称
column	是受约束条件影响的列的名称
constraint	是约束条件的名称

如果删除了一个完整性约束条件，Oracle Server 就不再强制执行该约束条件，数据字典中也不再提供该约束条件。

禁用约束条件

- 在 ALTER TABLE 语句中使用 DISABLE 子句可停用完整性约束条件。
- 应用 CASCADE 选项可禁用相关的完整性约束条件。

```
ALTER TABLE emp2
DISABLE CONSTRAINT emp_dt_fk;
```

```
ALTER TABLE Emp2 succeeded.
```


禁用约束条件

可以使用包含 DISABLE 子句的 ALTER TABLE 语句来禁用约束条件，而不必删除或重新创建约束条件。

语法

```
ALTER   TABLE   table
 DISABLE CONSTRAINT constraint [CASCADE];
```

在该语法中：

table	是表名称
constraint	是约束条件的名称

准则

- 在 CREATE TABLE 语句和 ALTER TABLE 语句中都可以使用 DISABLE 子句。
- CASCADE 子句会禁用相关的完整性约束条件。
- 禁用 UNIQUE 或 PRIMARY KEY 约束条件时会导致删除唯一索引。

启用约束条件

- 使用 ENABLE 子句可激活表定义中当前禁用的完整性约束条件。

```
ALTER TABLE       emp2
ENABLE CONSTRAINT emp_dt_fk;
```

```
ALTER TABLE Emp2 succeeded.
```

- 如果启用了 UNIQUE 关键字或 PRIMARY KEY 约束条件，则会自动创建 UNIQUE 索引。

启用约束条件

可以使用包含 ENABLE 子句的 ALTER TABLE 语句来启用约束条件，而不必删除或重新创建约束条件。

语法

```
ALTER   TABLE      table
ENABLE  CONSTRAINT constraint;
```

在该语法中：

table	是表名称
constraint	是约束条件的名称

准则

- 如果启用了一个约束条件，该约束条件会应用于表中的所有数据。表中的所有数据都必须遵守该约束条件。
- 如果启用了 UNIQUE 关键字或 PRIMARY KEY 约束条件，则会自动创建 UNIQUE 或 PRIMARY KEY 索引。如果索引已经存在，这些关键字就会使用索引。
- 在 CREATE TABLE 语句和 ALTER TABLE 语句中都可以使用 ENABLE 子句。

启用约束条件（续）

- 启用使用 CASCADE 选项禁用的 PRIMARY KEY 约束条件时，并不会启用与 PRIMARY KEY 相关的任何 FOREIGN KEY。
- 要启用 UNIQUE 或 PRIMARY KEY 约束条件，必须拥有在表上创建索引所需要的权限。

级联约束条件

- CASCADE CONSTRAINTS 子句应与 DROP COLUMN 子句一起使用。
- CASCADE CONSTRAINTS 子句会删除引用了在被删除列上定义的主键和唯一关键字的所有引用完整性约束条件。
- CASCADE CONSTRAINTS 子句还会删除在被删除列上定义的所有多列约束条件。

级联约束条件

下面的语句说明了 CASCADE CONSTRAINTS 子句的用法。假设按如下方式创建了 TEST1 表：

```
CREATE TABLE test1 (
  col1_pk NUMBER PRIMARY KEY,
  col2_fk NUMBER,
  col1 NUMBER,
  col2 NUMBER,
  CONSTRAINT fk_constraint FOREIGN KEY (col2_fk) REFERENCES
    test1,
  CONSTRAINT ck1 CHECK (col1_pk > 0 and col1 > 0),
  CONSTRAINT ck2 CHECK (col2_fk > 0));
```

执行下列语句时会返回一条错误：

```
ALTER TABLE test1 DROP (col1_pk);  —col1_pk 是父键。
ALTER TABLE test1 DROP (col1);  —col1 被多列约束条件 ck1 引用。
```

级联约束条件

示例：

```
ALTER TABLE emp2
DROP COLUMN employee_id CASCADE CONSTRAINTS;
```

```
ALTER TABLE Emp2 succeeded.
```

```
ALTER TABLE test1
DROP (col1_pk, col2_fk, col1) CASCADE CONSTRAINTS;
```

```
ALTER TABLE test1 succeeded.
```

ORACLE

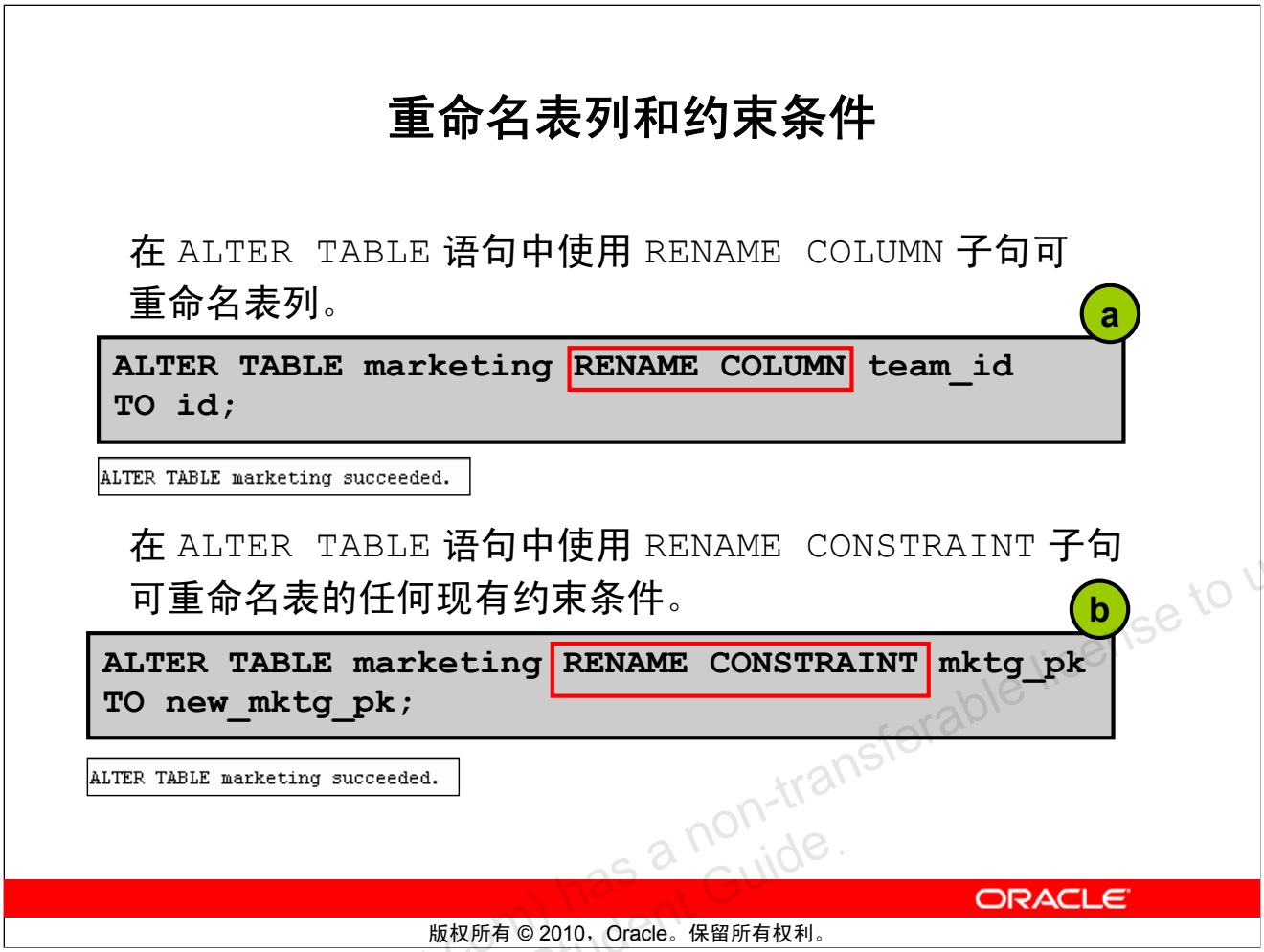
级联约束条件（续）

提交下面的语句时会删除 EMPLOYEE_ID 列、PRIMARY KEY 约束条件和引用 EMP2 表的 PRIMARY KEY 约束条件的所有 FOREIGN KEY 约束条件：

```
ALTER TABLE emp2 DROP COLUMN employee_id CASCADE CONSTRAINTS;
```

另外，如果由被删除列上定义的约束条件引用的所有列都已删除，则不需要 CASCADE CONSTRAINTS。例如，假设其它表中的其它引用约束条件都不引用 COL1_PK 列，则对上一页创建的 TEST1 表提交以下没有 CASCADE CONSTRAINTS 子句的语句是行之有效的：

```
ALTER TABLE test1 DROP (col1_pk, col2_fk, col1);
```



重命名表列和约束条件

重命名表列时，新名称一定不能与表中任何现有列的名称相冲突。不能将任何其它子句与 RENAME COLUMN 子句一起使用。

幻灯片示例创建的 marketing 表中，在 id 列上定义了 PRIMARY KEY mktg_pk。

```
CREATE TABLE marketing (team_id  NUMBER(10),
                        target   VARCHAR2(50),
CONSTRAINT mktg_pk PRIMARY KEY(team_id));
```

```
CREATE TABLE succeeded.
```

示例 **a** 显示 marketing 表中的 id 列已重命名为 mktg_id。示例 **b** 显示 mktg_id 已重命名为 new_mktg_pk。

重命名表的任何现有约束条件时，新名称一定不能与任何现有约束条件名称相冲突。可以使用 RENAME CONSTRAINT 子句重命名系统生成的约束条件名称。

课程安排

- 使用 ALTER TABLE 语句添加、修改或删除列
- 管理约束条件：
  - 添加和删除约束条件
  - 延迟约束条件
  - 启用和禁用约束条件
- 创建索引：
  - 使用 CREATE TABLE 语句
  - 创建基于函数的索引
  - 删除索引
- 执行闪回操作
- 创建和使用临时表
- 创建和使用外部表

ORACLE

索引概览

创建索引的方法：

- 自动创建
  - PRIMARY KEY 创建
  - UNIQUE KEY 创建
- 手动创建
  - CREATE INDEX 语句
  - CREATE TABLE 语句

索引概览

可以创建两种类型的索引。一种类型是唯一索引。如果在表中定义的一列或一组列具有 PRIMARY KEY 或 UNIQUE 关键字约束条件，则 Oracle Server 会自动创建唯一索引。索引的名称与约束条件的名称相同。

另一种类型的索引是用户可以创建的非唯一索引。例如，可以为了提高检索速度，而为在联接中使用的 FOREIGN KEY 列创建一个索引。

通过发出 CREATE INDEX 语句，可以在一列或多列上创建索引。

有关详细信息，请参阅《Oracle Database 11*g* SQL Referenc》。

注：可以手动创建唯一索引，但是建议您创建 UNIQUE 约束条件，这样可隐式创建唯一索引。

CREATE INDEX 与 CREATE TABLE 语句配合使用

```
CREATE TABLE NEW_EMP
(employee_id NUMBER(6)
             PRIMARY KEY USING INDEX
             (CREATE INDEX emp_id_idx ON
             NEW_EMP(employee_id)),
first_name   VARCHAR2(20),
last_name    VARCHAR2(25));
```

```
CREATE TABLE succeeded.
```

```
SELECT INDEX_NAME, TABLE_NAME
FROM   USER_INDEXES
WHERE  TABLE_NAME = 'NEW_EMP';
```

	INDEX_NAME	TABLE_NAME
1	EMP_ID_IDX	NEW_EMP

ORACLE

CREATE INDEX 与 CREATE TABLE 语句配合使用

在幻灯片示例中，CREATE INDEX 子句与 CREATE TABLE 语句一起使用可显式创建 PRIMARY KEY 索引。您可以在创建 PRIMARY KEY 时命名索引，使索引的名称不同于 PRIMARY KEY 约束条件的名称。

您可以通过查询 USER_INDEXES 数据字典视图来查找有关索引的信息。

注：在“使用数据字典视图管理对象”一课中将介绍有关 USER_INDEXES 的详细内容。

以下示例说明在未显式命名索引的情况下数据库的行为：

```
CREATE TABLE EMP_UNNAMED_INDEX
   (employee_id NUMBER(6) PRIMARY KEY ,
    first_name VARCHAR2(20),
    last_name VARCHAR2(25));
```

```
CREATE TABLE succeeded.
```

```
SELECT INDEX_NAME, TABLE_NAME
FROM   USER_INDEXES
WHERE  TABLE_NAME = 'EMP_UNNAMED_INDEX';
```

	INDEX_NAME	TABLE_NAME
1	SYS_C0017294	EMP_UNNAMED_INDEX

CREATE INDEX 与 CREATE TABLE 语句配合使用（续）

注意 Oracle Server 给为 PRIMARY KEY 列创建的索引提供了通用名称。

此外，可以对 PRIMARY KEY 列使用现有索引，例如，在要加载大量数据且要加快操作速度时。您可能要在执行加载时禁用约束条件，然后再启用约束条件，在这种情况下，PRIMARY KEY 上的唯一索引仍会导致在加载期间对数据进行验证。因此，您可以先在被指定为 PRIMARY KEY 的列上创建非唯一索引，然后再创建 PRIMARY KEY 列并指定该列使用现有索引。下面的示例描述了此过程：

步骤 1：创建表：

```
CREATE TABLE NEW_EMP2
 (employee_id NUMBER(6),
 first_name  VARCHAR2(20),
 last_name   VARCHAR2(25)
 );
```

步骤 2：创建索引：

```
CREATE INDEX emp_id_idx2 ON
  new_emp2(employee_id);
```

步骤 3：创建 PRIMARY KEY：

```
ALTER TABLE new_emp2 ADD PRIMARY KEY (employee_id) USING INDEX
 emp_id_idx2;
```

基于函数的索引

- 基于函数的索引是在表达式基础上建立的。
- 索引表达式是用表列、常数、SQL 函数和用户定义函数构建的。

```
CREATE INDEX upper_dept_name_idx
ON dept2(UPPER(department_name));
```

```
CREATE INDEX succeeded.
```

```
SELECT *
FROM   dept2
WHERE  UPPER(department_name) = 'SALES';
```


基于函数的索引

使用 UPPER(*column_name*) 或 LOWER(*column_name*) 关键字定义的基于函数的索引允许执行不区分大小写的搜索。例如，假如有以下索引：

```
CREATE INDEX upper_last_name_idx ON emp2 (UPPER(last_name));
```

这个索引可简化对如下查询的处理：

```
SELECT * FROM emp2 WHERE UPPER(last_name) = 'KING';
```

只有在查询中使用了特殊函数时，Oracle Server 才会使用该索引。例如，下面的语句可能会使用该索引，但是，没有 WHERE 子句时，Oracle Server 可能会执行全表扫描：

```
SELECT   *
FROM     employees
WHERE    UPPER (last_name) IS NOT NULL
ORDER BY UPPER (last_name);
```

注：要使用基于函数的索引，QUERY_REWRITE_ENABLED 初始化参数必须设置为 TRUE。

Oracle Server 将标记为 DESC 的列的索引当作基于函数的索引进行处理。标记为 DESC 的列是按降序排列的。

删除索引

- 使用 DROP INDEX 命令可从数据字典中删除索引：

```
DROP INDEX index;
```

- 从数据字典中删除 UPPER_DEPT_NAME_IDX 索引：

```
DROP INDEX upper_dept_name_idx;
```

```
DROP INDEX upper_dept_name_idx succeeded.
```

- 要删除索引，必须是索引的所有者或拥有 DROP ANY INDEX 权限。

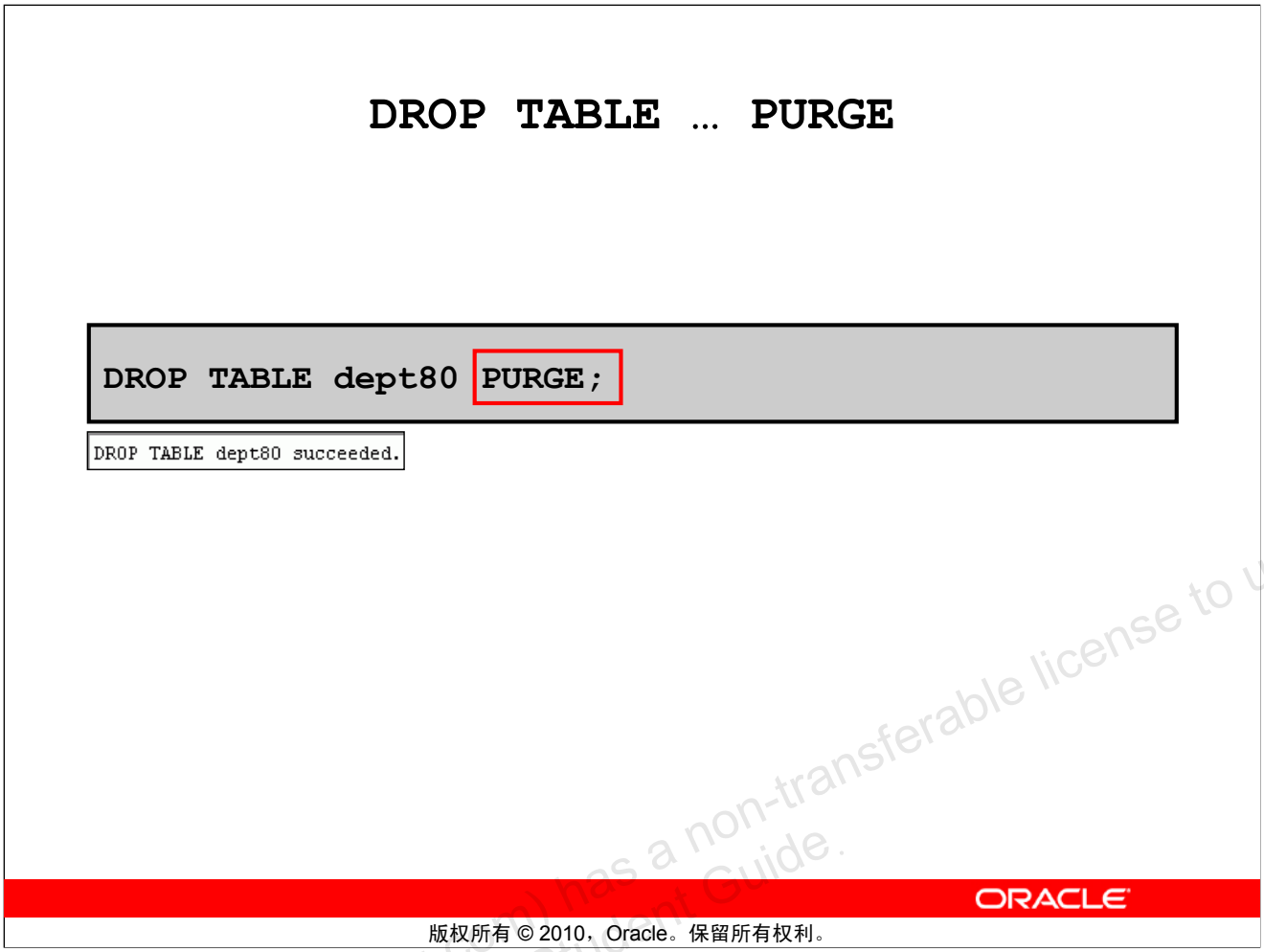
删除索引

您不能修改索引。要更改索引，必须先删除它，然后重新创建。发出 DROP INDEX 语句后，可从数据字典中删除索引定义。要删除索引，必须是索引的所有者或拥有 DROP ANY INDEX 权限。

在该语法中：

index 是索引的名称

注：如果删除了一个表，就会自动删除索引、约束条件和触发器，但会保留视图和序列。



DROP TABLE … PURGE

Oracle DB 提供了删除表的功能。删除一个表后，数据库不会立即释放与该表相关的空间。数据库会重命名该表并将其放入回收站。如果以后发现不应删除该表，则可以使用 FLASHBACK TABLE 语句恢复该表。如果要在发出 DROP TABLE 语句时立即释放与表相关的空间，则应包括 PURGE 子句，如幻灯片语句中所示。

只有要通过执行一个步骤即删除表又释放与其相关的空间时，才需要指定 PURGE。如果指定了 PURGE，数据库就不会将表及其相关对象放入回收站。

使用此子句时等同于先删除表，然后再从回收站中清除表。此子句减少了一个操作步骤。此外，当不希望机密资料出现在回收站时，使用该子句还增强了安全性。

课程安排

- 使用 ALTER TABLE 语句添加、修改或删除列
- 管理约束条件：
  - 添加和删除约束条件
  - 延迟约束条件
  - 启用和禁用约束条件
- 创建索引：
  - 使用 CREATE TABLE 语句
  - 创建基于函数的索引
  - 删除索引
- 执行闪回操作
- 创建和使用临时表
- 创建和使用外部表

ORACLE

FLASHBACK TABLE 语句

- 允许使用一条语句将表恢复到指定的时间点
- 还原表数据以及关联的索引和约束条件
- 允许将表及其内容还原到特定时间点或系统更改号 (SCN)

ORACLE

FLASHBACK TABLE 语句

使用 Oracle 的 Flashback Table 时，使用一条语句就可将表恢复到指定时间点。在数据库联机时，可以还原表数据及关联的索引和约束条件，从而撤消对指定表所做的更改。

Flashback Table 功能类似于一个自助修复工具。例如，如果某个用户无意删除了表中的一些重要行后要恢复被删除行，则可使用 FLASHBACK TABLE 语句将表还原到删除之前的时间并查看表中是否缺少有关行。

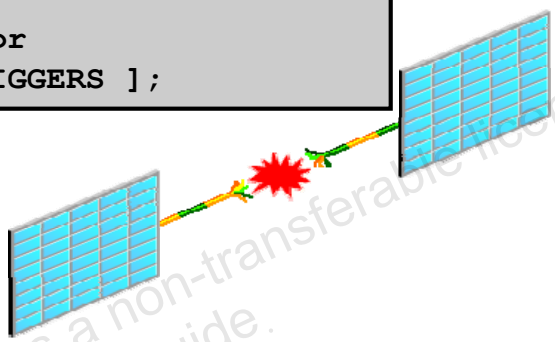
使用 FLASHBACK TABLE 语句时，可以将表及其内容还原到特定时间或 SCN。

注：SCN 是一个与数据库的每次更改关联的整数值。它是数据库中一个唯一的递增数字。每次提交一个事务处理，就会记录一个新 SCN。

FLASHBACK TABLE 语句

- 一个可用来对表意外修改进行修复的工具
  - 将表还原到以前时间点
  - 好处：易于使用、可用性更高和执行速度快
  - 可在本地执行
- 语法：

```
FLASHBACK TABLE[schema.]table[,
[ schema.]table ]...
TO { TIMESTAMP | SCN } expr
[ { ENABLE | DISABLE } TRIGGERS ];
```



ORACLE

FLASHBACK TABLE 语句（续）

自助修复工具

在 Oracle DB 中提供了一个 SQL 数据定义语言 (DDL) 命令 FLASHBACK TABLE，可用于在意外删除或修改了某个表时将该表还原到以前时间点的状态。FLASHBACK TABLE 命令是一个自助修复工具，可用来还原表中的数据及其相关属性，如索引或视图。当数据库联机时，通过只回退对指定表执行的后续更改，便可完成上述任务。与传统的恢复机制相比，此功能带来了一些好处，如易于使用、可用性更高和还原速度更快等。这个功能还减轻了 DBA 的负担，使他们不必查找和还原特定于应用程序的属性。闪回表功能并不能处理因磁盘损坏而导致的物理损坏。

语法

可以调用一个或多个表，甚至是不同方案中各个表上的 FLASHBACK TABLE 操作。通过提供一个有效的时间戳可指定要还原到的时间点。默认情况下，在对所有涉及的表执行闪回操作期间，数据库触发器会处于禁用状态。通过指定 ENABLE TRIGGERS 子句可以改写此默认行为。

注：有关回收站和闪回语义的详细信息，请参阅《Oracle Database Administrator's Guide 11*g* Release 2 (11.2)》。

使用 FLASHBACK TABLE 语句

```
DROP TABLE emp2;
```

```
DROP TABLE emp2 succeeded.
```

```
SELECT original_name, operation, droptime FROM
recyclebin;
```

ORIGINAL_NAME	OPERATION	DROPTIME
EMP2	DROP	2009-05-20:18:00:39

...

```
FLASHBACK TABLE emp2 TO BEFORE DROP;
```

```
FLASHBACK TABLE succeeded.
```

ORACLE

使用 FLASHBACK TABLE 语句

语法和示例

该示例将 EMP2 表还原到执行 DROP 语句之前的状态。

回收站实际上是一个包含关于已删除对象的信息的数据字典表。已删除表及其所有相关对象（如索引、约束条件、嵌套表等）并没有被删除，仍占用一定空间。这些对象会继续占用用户空间，直到明确地从回收站中清除这些对象或因表空间约束条件而必须在数据库中删除这些对象为止。

每一个用户都可被视为回收站的所有者；如果用户没有 SYSDBA 权限，则只能在回收站中访问属于自己的那些对象。使用以下语句时，用户可在回收站中查看自己的对象：

```
SELECT * FROM RECYCLEBIN;
```

删除一个用户后，就不会在回收站中保存属于该用户的任何对象，因此会在回收站中清除所有相关对象。

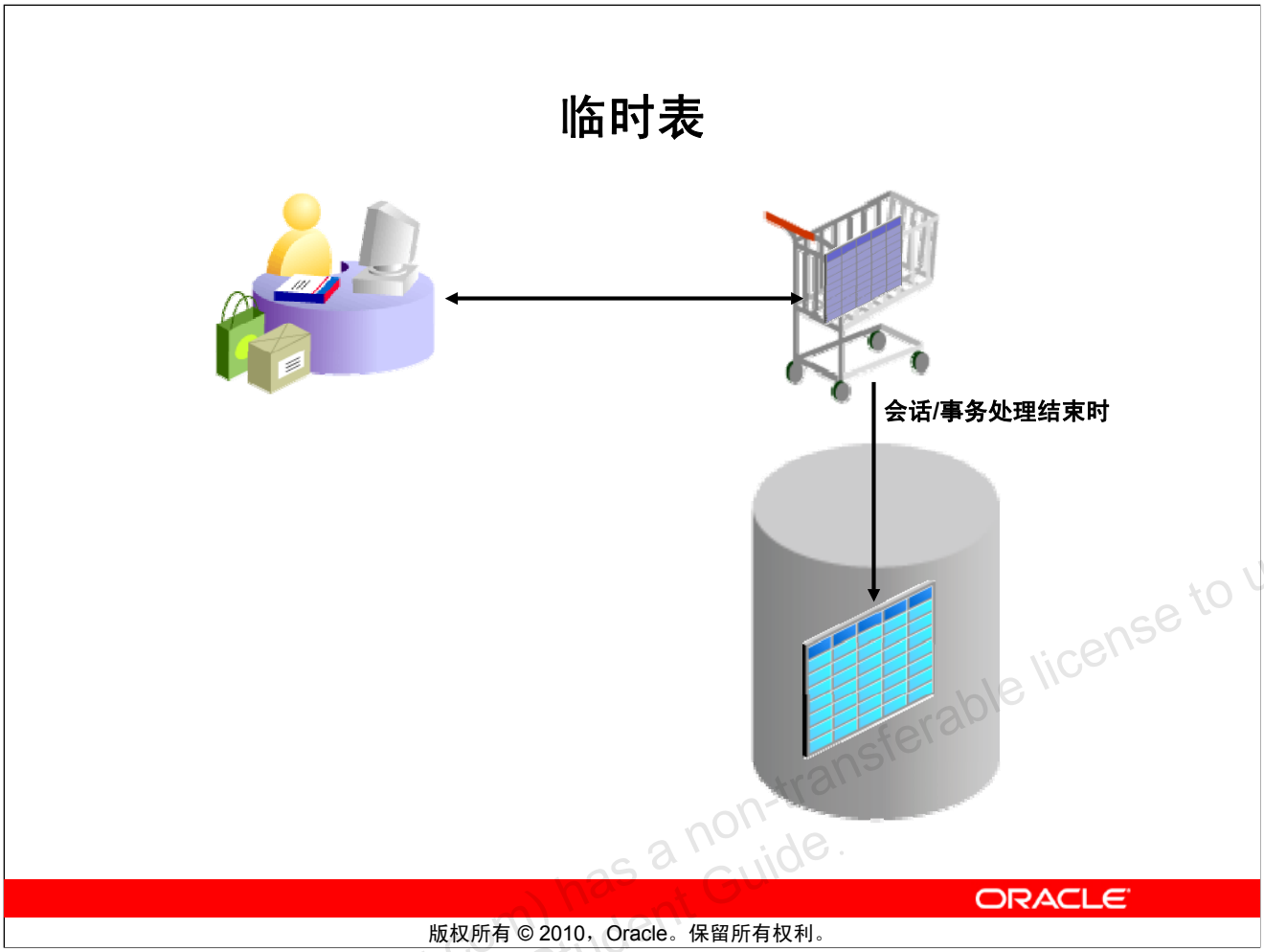
可以使用以下语句清空回收站：

```
PURGE RECYCLEBIN;
```

课程安排

- 使用 ALTER TABLE 语句添加、修改或删除列
- 管理约束条件：
  - 添加和删除约束条件
  - 延迟约束条件
  - 启用和禁用约束条件
- 创建索引：
  - 使用 CREATE TABLE 语句
  - 创建基于函数的索引
  - 删除索引
- 执行闪回操作
- 创建和使用临时表
- 创建和使用外部表

ORACLE

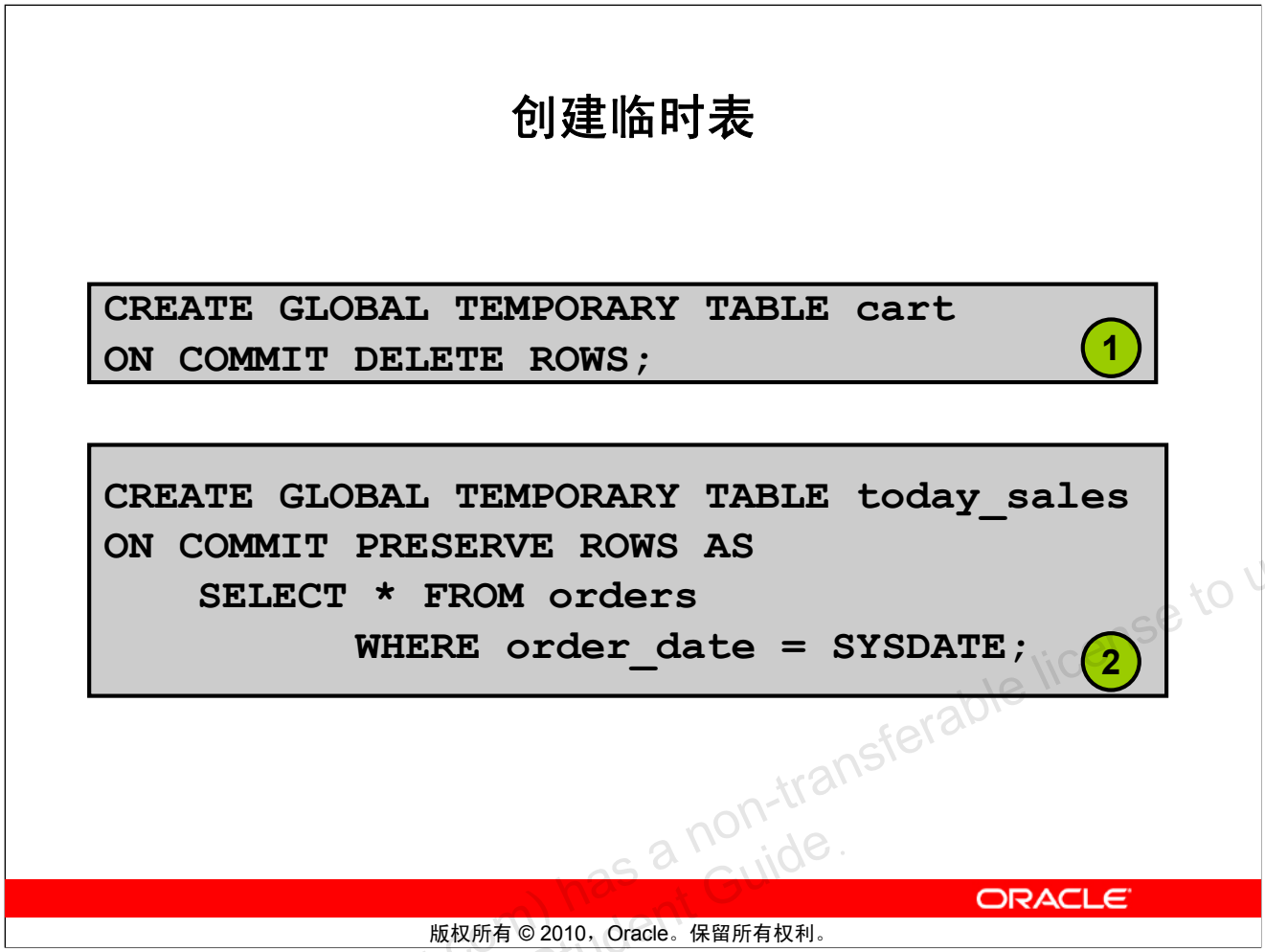


临时表

临时表是一种表，用于存放仅在事务处理或会话持续时间内存在的数据。临时表中的数据专用于会话，意即每个会话只能查看和修改其自己的数据。

对于其中结果集必须进行缓存的应用程序而言，临时表比较有用。例如，联机应用程序中的购物车可以是一个临时表。每一项都由临时表中的行表示。在在线商店中购物时，可以不断地向购物车中添加或从其中删除项目。会话期间，该购物车是专用的。结束购物并进行支付后，应用程序会将选定购物车的行移至永久表中。在会话结束时，临时表中的数据将自动删除。

因为临时表是以静态方式进行定义的，所以可以为其创建索引。在临时表上创建的索引也是临时的。索引中的数据与临时表中的数据拥有相同的会话或事务处理范围。您还可以在临时表上创建视图或触发器。



创建临时表

要创建临时表，可以使用以下命令：

```
CREATE GLOBAL TEMPORARY TABLE tablename
ON COMMIT [PRESERVE | DELETE] ROWS
```

通过将以下设置之一与 ON COMMIT 子句关联，可以确定临时表中的数据是特定于事务处理（默认设置）还是特定于会话。

1. DELETE ROWS：如幻灯片中例 1 所示，DELETE ROWS 设置将创建一个特定于事务处理的临时表。会话将随着事务处理在第一次插入临时表时与该表绑定。该绑定将在事务处理结束时消失。数据库将在每次提交后截断该表（删除所有行）。
2. PRESERVE ROWS：如幻灯片中例 2 所示，PRESERVE ROWS 设置将创建一个特定于会话的临时表。每个销售代表会话都可在该表中存储其自己当天的销售数据。当销售人员在 today_sales 表上执行第一次插入时，其会话将绑定至 today_sales 表。这种绑定会在会话结束时或通过在会话中对表发出 TRUNCATE 命令后消失。数据库将在您终止会话时截断该表。

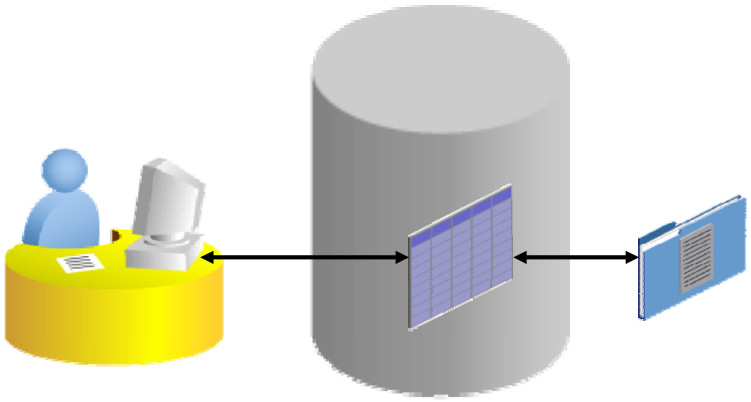
在 Oracle DB 中创建临时表时，将创建一个静态表定义。与永久表类似，临时表在数据字典中进行定义。然而，临时表及其索引在创建时不会自动分配段。反之，临时段将在第一次插入数据时进行分配。直至数据加载至会话中，表才会显示为空。

课程安排

- 使用 ALTER TABLE 语句添加、修改或删除列
- 管理约束条件：
  - 添加和删除约束条件
  - 延迟约束条件
  - 启用和禁用约束条件
- 创建索引：
  - 使用 CREATE TABLE 语句
  - 创建基于函数的索引
  - 删除索引
- 执行闪回操作
- 创建和使用临时表
- 创建和使用外部表

ORACLE

外部表



ORACLE

外部表

外部表是只读表，它的元数据存储在数据库中，它的数据存储在数据库之外。这种外部表定义可视为一个视图，在对外部数据进行任何 SQL 查询时可使用此表，而不需要先在数据库中加载外部数据。查询外部表数据和直接联接外部表数据可以并行进行，而不必先在数据库中加载外部数据。可以使用 SQL、PL/SQL 和 Java 查询外部表中的数据。

外部表与常规表的主要不同之处在于，外部表是只读表。既不能对这种表执行数据操纵语言 (DML) 操作，也不能对这种表创建索引。但是，您可以使用 `CREATE TABLE AS SELECT` 命令创建外部表，从而卸载数据。

Oracle Server 为外部表提供了两种主要访问驱动程序。一种是加载程序访问驱动程序（或 `ORACLE_LOADER`），用于从其格式可由 SQL*Loader 实用程序解释的外部文件中读取数据。注意，SQL*Loader 功能并不一定支持所有外部表。另一种是 `ORACLE_DATAPUMP` 访问驱动程序，该访问驱动程序可使用与平台无关的格式导入和导出数据。`ORACLE_DATAPUMP` 访问驱动程序将要加载的 `SELECT` 语句中的行作为 `CREATE TABLE...ORGANIZATION EXTERNAL...AS SELECT` 语句的一部分而写入外部表。您随后可以使用 `SELECT` 从数据文件中读取有关数据。还可以在其它系统上创建外部表定义，然后使用该数据文件。这样允许在 Oracle DB 之间移动数据。

创建外部表目录

创建一个 DIRECTORY 对象来与外部数据源所在的文件系统上的目录相对应。

```
CREATE OR REPLACE DIRECTORY emp_dir
AS '/…/emp_dir';

GRANT READ ON DIRECTORY emp_dir TO ora_21;
```

ORACLE

创建外部表示例

使用 CREATE DIRECTORY 语句可创建一个目录对象。目录对象为外部数据源所在服务器文件系统上的目录指定了别名。您可以在引用外部数据源时使用目录名，而不是对操作系统路径名进行硬编码，这样可提高文件管理的灵活性。

必须拥有 CREATE ANY DIRECTORY 系统权限才能创建目录。创建目录时，您被自动授予 READ 和 WRITE 对象权限，还可以将 READ 和 WRITE 权限授予其他用户或角色。DBA 也可以将这些权限授予其他用户和角色。

用户需要拥有要访问的外部表中使用的所有目录的 READ 权限，需要拥有正使用的日志文件位置、损坏文件位置和丢弃文件位置的 WRITE 权限。

另外，使用外部表结构卸载数据时，也必须拥有 WRITE 权限。

Oracle 还提供了 ORACLE_DATAPUMP 类型，使用此类型可卸载数据（也就是从数据库表中读取数据，然后在外部表中插入有关数据），然后重新在 Oracle DB 中加载有关数据。这个操作是创建表时可完成的一次性操作。完成创建操作和最初的填充操作后，就不能更新、插入或删除任何行。

创建外部表示例（续）

语法

```
CREATE [OR REPLACE] DIRECTORY AS 'path_name';
```

在该语法中：

OR REPLACE	指定了 OR REPLACE 时，如果目录数据库对象已存在，可重新创建该对象。可以使用此子句更改现有目录的定义，而不必删除、重新创建和重新授予以前对该目录授予的数据库对象权限。以前对重新定义目录拥有权限的用户仍可访问该目录，而不必重新授予权限。
directory	指定了要创建目录对象的名称。目录名称的最大长度为 30 字节。不能使用方案名来限定目录对象。
'path_name'	指定了要访问的操作系统目录的完全路径名称。路径名称区分大小写。

创建外部表

```
CREATE TABLE <table_name>
   ( <col_name> <datatype>, … )
ORGANIZATION EXTERNAL
    (TYPE <access_driver_type>
     DEFAULT DIRECTORY <directory_name>
     ACCESS PARAMETERS
       (… ) )
      LOCATION ('<location_specifier>')
REJECT LIMIT [0 | <number> | UNLIMITED];
```


创建外部表

在 CREATE TABLE 语句中使用 ORGANIZATION EXTERNAL 子句可创建外部表。实际上，创建的不是表，而是在可用来访问外部数据的数据字典中创建元数据。使用 ORGANIZATION 子句可指定表数据行的存储顺序。在 ORGANIZATION 子句中指定 EXTERNAL 后，即指明该表是位于数据库之外的只读表。注意，在数据库之外必须已存在外部文件。

TYPE *<access_driver_type>* 指示外部表的访问驱动程序。访问驱动程序是解释数据库外部数据的应用编程接口 (API)。如果不指定 TYPE，Oracle 会使用默认访问驱动程序 ORACLE_LOADER。另一个选项是 ORACLE_DATAPUMP。

使用 DEFAULT DIRECTORY 子句可指定一个或多个与外部数据源所在文件系统上的目录相对应的 Oracle DB 目录对象。

使用可选的 ACCESS PARAMETERS 子句，可以指定适用于外部表的特定访问驱动程序的参数值。

创建外部表（续）

使用 LOCATION 子句可为每个外部数据源指定一个外部定位器。
<*location_specifier*> 通常是一个文件，但也可以不是。

REJECT LIMIT 子句用于指定在对外部数据进行查询期间出现多少次转换错误后才会返回 Oracle 错误并中止查询。默认值为 0。

使用 ORACLE_DATAPUMP 访问驱动程序的语法如下：

```
CREATE TABLE extract_emps
ORGANIZATION EXTERNAL (TYPE ORACLE_DATAPUMP
                       DEFAULT DIRECTORY …
                       ACCESS PARAMETERS (… )
                       LOCATION (…)
                       PARALLEL 4
                       REJECT LIMIT UNLIMITED
AS
SELECT * FROM …;
```

使用 ORACLE_LOADER 创建外部表

```
CREATE TABLE oldemp (
  fname char(25), lname CHAR(25))
  ORGANIZATION EXTERNAL
  (TYPE ORACLE_LOADER
  DEFAULT DIRECTORY emp_dir
  ACCESS PARAMETERS
  (RECORDS DELIMITED BY NEWLINE
   NOBADFILE
   NOLOGFILE
  FIELDS TERMINATED BY ','
  (fname POSITION ( 1:20) CHAR,
   lname POSITION (22:41) CHAR))
  LOCATION ('emp.dat'))
  PARALLEL 5
  REJECT LIMIT 200;
```

```
CREATE TABLE succeeded.
```


使用 ORACLE_LOADER 访问驱动程序创建外部表的示例

假定一个平面文件中记录的格式如下所示：

```
10,jones,11-Dec-1934
20,smith,12-Jun-1972
```

一条记录占用一行，各个字段都用逗号 (,) 分隔开。此文件的名称为 /emp_dir/emp.dat。

要将此文件转换为外部表的数据源，其元数据仍位于数据库中，必须执行以下步骤：

1. 按如下方式创建目录对象 emp_dir：

```
CREATE DIRECTORY emp_dir AS '/emp_dir' ;
```

2. 运行幻灯片中显示的 CREATE TABLE 命令。

幻灯片示例中显示的是为根据文件创建外部表的表规范：

```
/emp_dir/emp.dat
```

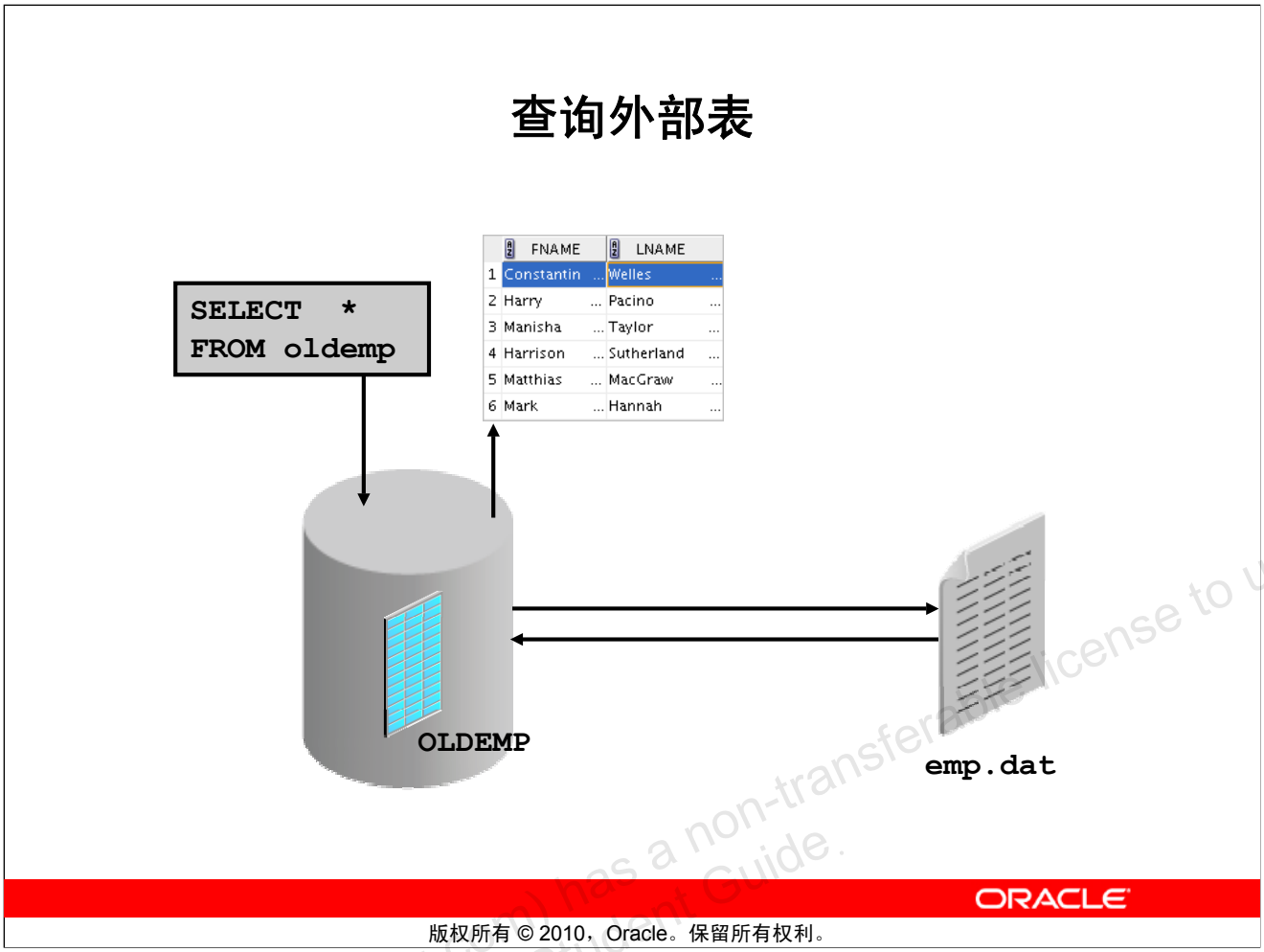
使用 ORACLE_LOADER 访问驱动程序创建外部表的示例（续）

在本示例中，给出 TYPE 规范的目的只是为了说明它的用法。如果没有另行指定，ORACLE_LOADER 是默认访问驱动程序。ACCESS PARAMETERS 选项会提供特定访问驱动程序的参数值，这些值由访问驱动程序解释，而不是由 Oracle Server 解释。

执行 INSERT INTO TABLE 语句时，使用 PARALLEL 子句可使 5 个并行执行服务器同时对外部数据源（文件）进行扫描。例如，如果指定 PARALLEL=5，则可在一个数据源上运行多个并行执行服务器。因为外部表可能非常大，考虑到性能方面的问题，所以建议为查询指定 PARALLEL 子句或指定并行提示。

REJECT LIMIT 子句指定，如果在查询外部数据过程中出现的转换错误超过 200 个，就会中止查询并返回错误。当访问驱动程序尝试转换数据文件中的数据以匹配外部表定义时，就可能出现这些转换错误。

成功执行 CREATE TABLE 命令后，就可以像对待关系表一样显示和查询 OLDEMP 外部表。



查询外部表

在外部表中不会描述数据库中存储的任何数据。也不会描述在外部源中数据是如何存储的。而是描述外部表层怎样向服务器提供数据。访问驱动程序和外部表层会对数据文件中的数据进行必要的转换，使这些数据与外部表定义相匹配，这是访问驱动程序和外部表层的责任。

当数据库服务器访问外部源中的数据时，它会调用相应的访问驱动程序来按数据库服务器期望的格式从外部源中获取数据。

切记数据源中数据的描述与外部表定义是相互独立的，这一点非常重要。源文件包含的字段数可以多于或少于表中的列数。而且，数据源字段的数据类型可以不同于表列的数据类型。访问驱动程序可确保数据源中的数据经过处理后与外部表定义相匹配。

使用 ORACLE_DATAPUMP 创建外部表：示例

```
CREATE TABLE emp_ext
  (employee_id, first_name, last_name)
   ORGANIZATION EXTERNAL
    (
     TYPE ORACLE_DATAPUMP
     DEFAULT DIRECTORY emp_dir
     LOCATION
      ('emp1.exp','emp2.exp')
    )
   PARALLEL
AS
SELECT employee_id, first_name, last_name
FROM   employees;
```

ORACLE

使用 ORACLE_DATAPUMP 创建外部表：示例

使用 ORACLE_DATAPUMP 访问驱动程序时，可以通过外部表执行卸载和重新加载操作。

注：在外部表上下文中，所谓加载数据的过程，是指从外部表读取数据后将其加载在数据库表中的过程。所谓卸载数据的过程，是指从数据库表中读取数据后将其插入在外部表中的过程。

幻灯片示例中显示的是使用 ORACLE_DATAPUMP 访问驱动程序创建外部表的表规范。然后在以下两个文件中填充数据：emp1.exp 和 emp2.exp。

要将从 EMPLOYEES 表读取的数据填充在外部表中，必须执行以下步骤：

1. 按如下方式创建目录对象 emp_dir：
   ```
   CREATE DIRECTORY emp_dir AS '/emp_dir' ;
   ```
2. 运行幻灯片中显示的 CREATE TABLE 命令。

注：emp_dir 目录与上个示例中使用 ORACLE_LOADER 创建的目录相同。

通过执行以下代码可查询外部表：

```
SELECT * FROM emp_ext;
```

小测验

FOREIGN KEY 约束条件强制执行以下操作：删除父关键字数据后，还会删除子表中与被删除的父关键字值相关的所有行。

1. 正确
2. 错误

ORACLE

答案：2

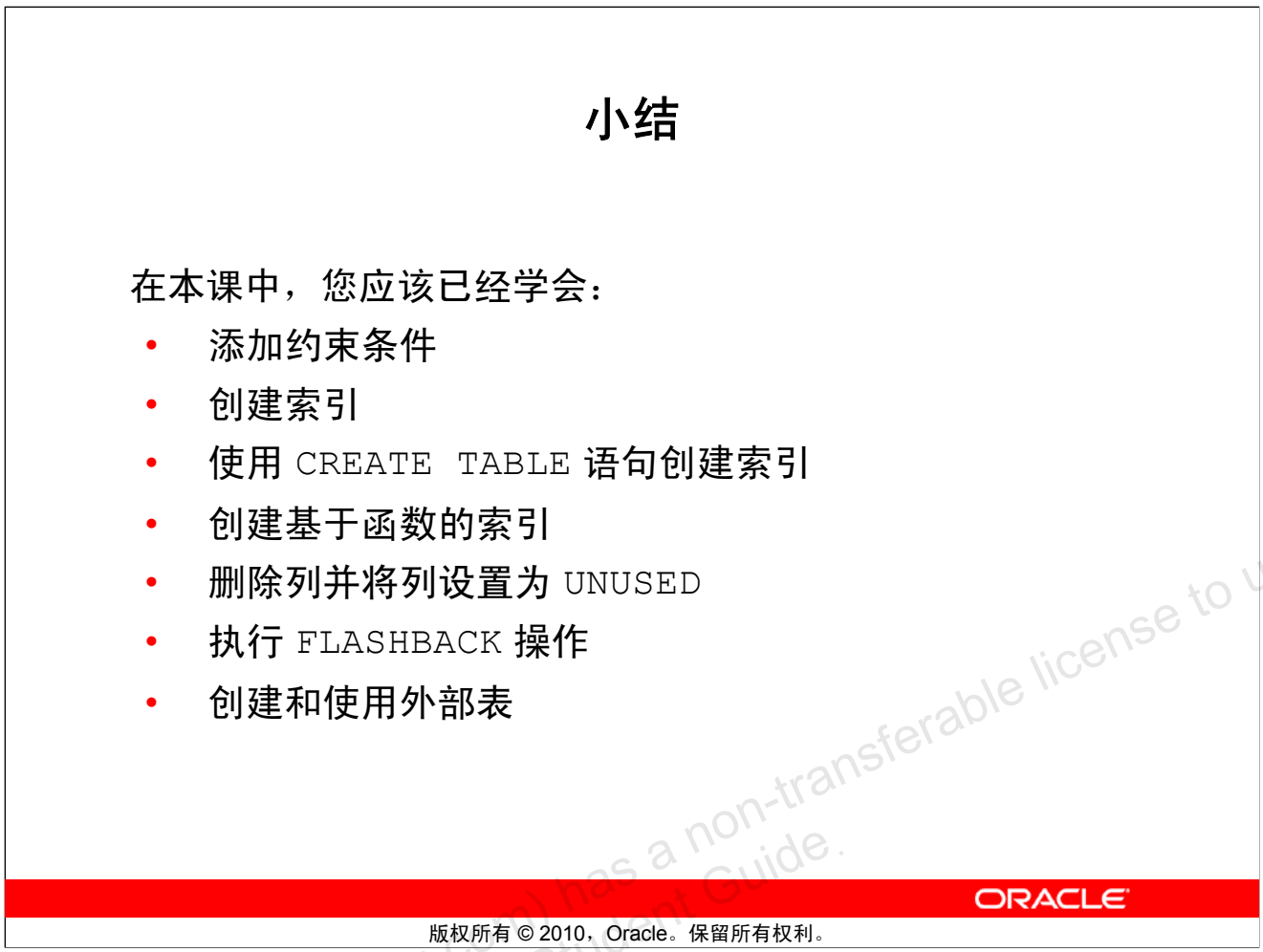
小测验

在所有情况下，当执行 DROP TABLE 命令时，数据库会重命名该表并将其放入回收站，以后可通过使用 FLASHBACK TABLE 语句恢复该表。

1. 正确
2. 错误

ORACLE

答案：2



小结

在本课中，您学习了如何执行以下任务以管理方案对象：

- 通过更改表可添加或修改列或约束条件。
- 使用 CREATE INDEX 语句可创建索引和基于函数的索引。
- 删除未使用的列。
- 使用 FLASHBACK 方法可还原表。
- 使用 CREATE TABLE 语句的 ORGANIZATION EXTERNAL 子句创建外部表。外部表是只读表，它的元数据存储在数据库中，它的数据存储在数据库之外。
- 使用外部表可查询数据，而不必先在数据库中加载数据。
- 使用 CREATE TABLE 语句创建表时，应命名 PRIMARY KEY 列索引。

练习 2：概览

本练习包含以下主题：

- 更改表
- 添加列
- 删除列
- 创建索引
- 创建外部表

ORACLE

练习 2：概览

在本练习中，您将使用 ALTER TABLE 命令修改列和添加约束条件。CREATE INDEX 命令与 CREATE TABLE 命令一起使用的目的是为了在创建表时创建索引。您创建的是外部表。

3

使用数据字典视图管理对象

课程目标

学完本课后，应能完成下列工作：

- 使用数据字典视图检查对象数据
- 在各种数据字典视图中进行查询

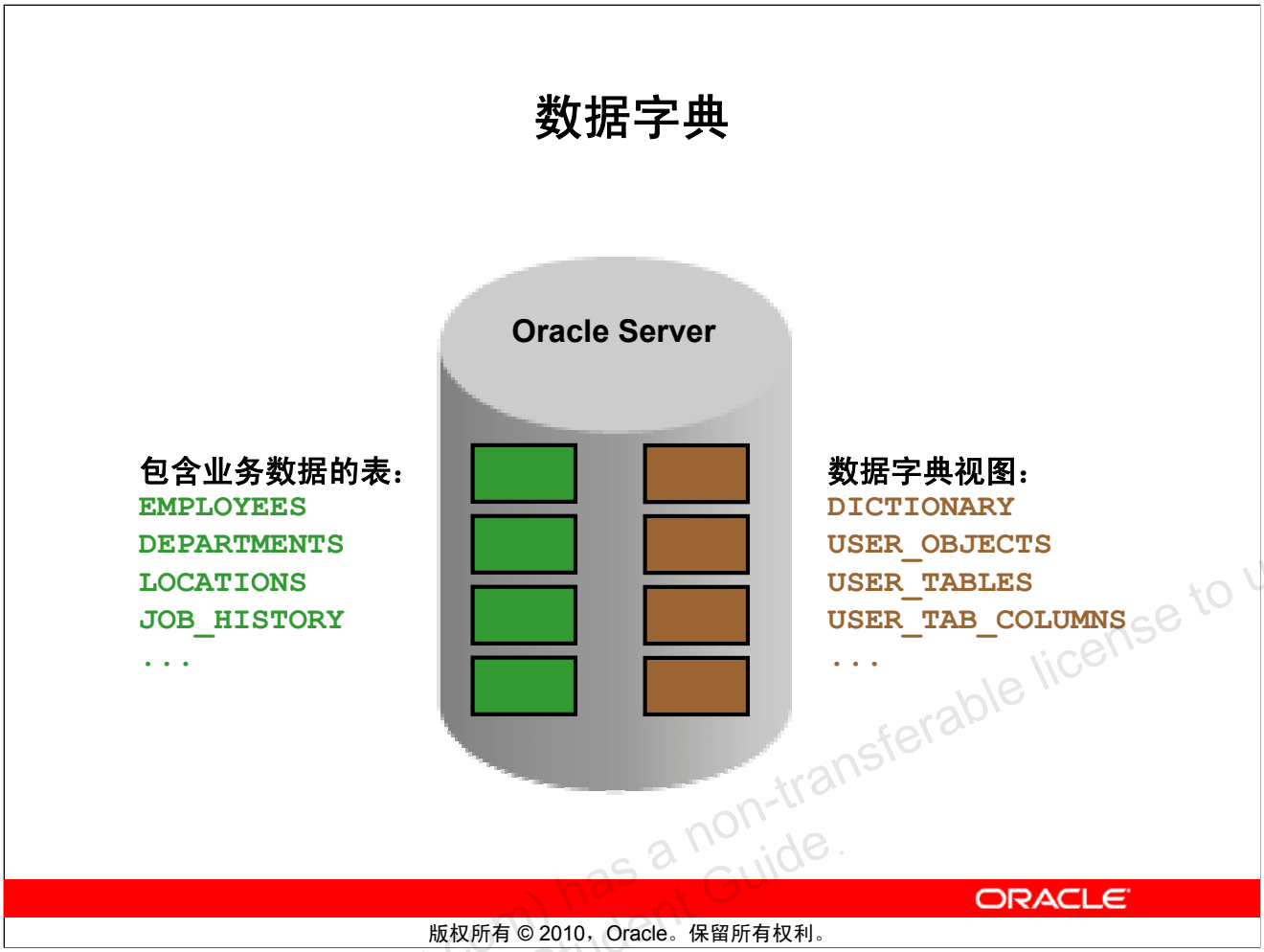
课程目标

本课将向您介绍数据字典视图。您将学习如何使用字典视图来检索元数据和创建关于方案对象的报表。

课程安排

- 数据字典简介
- 在字典视图中查询以下信息：
  - 表信息
  - 列信息
  - 约束条件信息
- 在字典视图中查询以下信息：
  - 视图信息
  - 序列信息
  - 同义词信息
  - 索引信息
- 在表中添加注释以及在字典视图中查询注释信息

ORACLE



数据字典

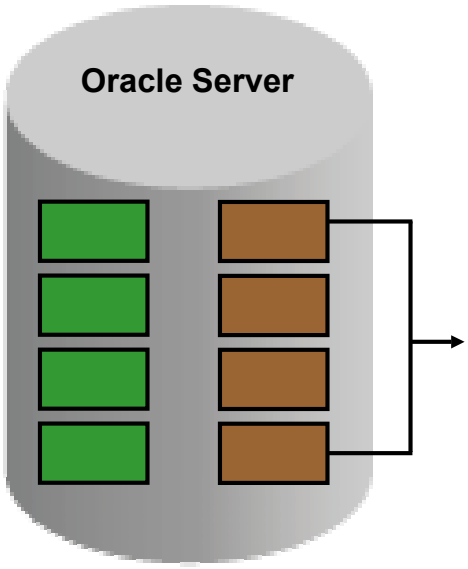
用户表是由用户创建并包含业务数据的一些表，如 EMPLOYEES。另外，Oracle DB 中有一个由表和视图组成的集合，这个集合被称为数据字典。此集合由 Oracle Server 创建和维护，其中包含关于数据库的信息。数据字典是结构化的表和视图，就像其它数据库数据一样。数据字典不仅是每个 Oracle DB 的核心，而且还是所有用户（从最终用户到应用程序设计者和数据库管理员）的一个重要工具。

使用 SQL 语句可访问数据字典。因为数据字典是只读的，所以只能对它的表和视图发出查询命令。

可以通过在基于字典表的字典视图中进行查询来查找如下信息：

- 数据库中所有方案对象（表、视图、索引、同义词、序列、过程、函数、程序包、触发器等等）的定义
- 列的默认值
- 完整性约束条件信息
- Oracle 用户的名称
- 授予每个用户的权限和角色
- 其它常规数据库信息

数据字典结构



由以下几部分组成：
- 基表
- 用户可访问的视图

数据字典结构

基础基表中存储了相关数据库的信息。只有 Oracle Server 可写入和读取这些表。用户很少直接访问这些表。

在若干个视图中汇总并显示在数据字典基表中存储的信息。这些视图使用联接和 WHERE 子句将基表数据解码为有用的信息（如用户名或表名），使这些信息更易于理解。大多数用户可以访问视图，但不能访问基表。

Oracle 用户 SYS 拥有数据字典的所有基表以及用户可访问的视图。Oracle 用户在任何时候都不应更改（UPDATE、DELETE 或 INSERT）在 SYS 方案中包含的任何行或方案对象，因为此类操作会破坏数据完整性。

数据字典结构

视图命名惯例：

视图前缀	用途
USER	用户的视图（您方案中的数据；您所拥有的数据）
ALL	扩展用户的视图（您可以访问的数据）
DBA	数据库管理员的视图（每个人方案中的数据）
V$	性能相关的数据

ORACLE

数据字典结构（续）

数据字典由一些视图集组成。在许多情况下，一个视图集由包含相似信息并用前缀相互区分的三个视图组成。例如，名为 USER_OBJECTS、ALL_OBJECTS 和 DBA_OBJECTS 的三个视图。

这三个视图包含关于数据库对象的相似信息，只是范围有所不同。USER_OBJECTS 包含关于您拥有或创建的对象的信息。ALL_OBJECTS 包含关于您可访问的所有对象的信息。DBA_OBJECTS 包含关于所有用户拥有的所有对象的信息。对于前缀为 ALL 或 DBA 的视图，在名为 OWNER 的视图中通常有一个附加列，用来标识拥有对象的用户。

此外，还有一个前缀为 v$ 的视图集。这些视图本质上是动态的，其中存储了关于性能的信息。动态性能表不是实际存在的表，大多数用户都不能访问此类表。但是，数据库管理员可以使用这些表进行查询和创建视图，然后授予其他用户对这些视图的访问权限。本课程不对这些视图进行详细介绍。

如何使用字典视图

先介绍 DICTIONARY。它包含字典表和字典视图的名称和描述。

```
DESCRIBE DICTIONARY
```

```
Name                           Null     Type
------------------------------ -------- --------------
TABLE_NAME                              VARCHAR2(30)
COMMENTS                                VARCHAR2(4000)

2 rows selected
```

```
SELECT *
FROM   dictionary
WHERE  table_name = 'USER_OBJECTS';
```

	TABLE_NAME	COMMENTS
1	USER_OBJECTS	Objects owned by the user

ORACLE

如何使用字典视图

为了熟悉字典视图，可以使用名为 DICTIONARY 的字典视图。该视图包含您可访问的每个字典视图的名称和简要描述。

可以通过编写查询来搜索关于特定视图名的信息，也可以搜索 COMMENTS 列中的某个词或词组。在上面显示的示例中对 DICTIONARY 视图进行了描述。该视图含有两列。SELECT 语句会检索关于名为 USER_OBJECTS 的字典视图的信息。USER_OBJECTS 视图包含关于您拥有的所有对象的信息。

您可以通过编写查询来搜索 COMMENTS 列中的某个词或词组。例如，下面的查询会返回您可访问的所有视图的名称，这些视图的 COMMENTS 列中包含词 *columns*：

```
SELECT table_name
FROM dictionary
WHERE LOWER(comments) LIKE '%columns%';
```

注：数据字典中的名称均为大写。

USER_OBJECTS 和 ALL_OBJECTS 视图

USER_OBJECTS：

- 在 USER_OBJECTS 中进行查询可查看您拥有的所有对象。
- 使用 USER_OBJECTS，可获得您方案中的所有对象名称及类型的列表以及下列信息：
  - 创建日期
  - 上次修改日期
  - 状态（有效或无效）

ALL_OBJECTS：

- 在 ALL_OBJECTS 中进行查询可查看您拥有访问权限的所有对象。

ORACLE

USER_OBJECTS 和 ALL_OBJECTS 视图

在 USER_OBJECTS 视图中进行查询可查看您方案中所有对象的名称和类型。此视图中有以下几列：

- **OBJECT_NAME**：对象的名称
- **OBJECT_ID**：对象的字典对象编号
- **OBJECT_TYPE**：对象的类型（如 TABLE、VIEW、INDEX、SEQUENCE）
- **CREATED**：创建对象的时间戳
- **LAST_DDL_TIME**：使用数据定义语言 (DDL) 命令上次对对象进行修改时的时间戳
- **STATUS**：对象的状态（VALID、INVALID 或 N/A）
- **GENERATED**：此对象名称是否是系统生成的？(Y | N)

注：以上只列出了部分列。为了查看完整列表，请参阅《Oracle 数据库参考》中的“USER_OBJECTS”。

此外，在 ALL_OBJECTS 视图中进行查询可查看您拥有访问权限的所有对象的列表。

USER_OBJECTS 视图

```
SELECT object_name, object_type, created, status
FROM   user_objects
ORDER BY object_type;
```

	OBJECT_NAME	OBJECT_TYPE	CREATED	STATUS
1	LOC_COUNTRY_IX	INDEX	19-MAY-09	VALID

...

53	EMPLOYEES2	TABLE	22-MAY-09	VALID
54	SECURE_EMPLOYEES	TRIGGER	19-MAY-09	VALID
55	UPDATE_JOB_HISTORY	TRIGGER	19-MAY-09	VALID
56	EMP_DETAILS_VIEW	VIEW	19-MAY-09	VALID

...

ORACLE

USER_OBJECTS 视图

该示例显示此用户所拥有的所有对象的名称、类型、创建日期和状态。

OBJECT_TYPE 列的值可以为 TABLE、VIEW、SEQUENCE、INDEX、PROCEDURE、FUNCTION、PACKAGE 或 TRIGGER。

STATUS 列的值可以为 VALID、INVALID 或 N/A。虽然表始终有效，但视图、过程、函数、程序包和触发器可能无效。

CAT 视图

为了简化查询和输出，可以在 CAT 视图中进行查询。此视图只包含两列：TABLE_NAME 和 TABLE_TYPE。其中提供所有 INDEX、TABLE、CLUSTER、VIEW、SYNONYM、SEQUENCE 或 UNDEFINED 对象的名称。

注：CAT 是 USER_CATALOG 的同义词，USER_CATALOG 是一个视图，其中列出了用户拥有的各种表、视图、同义词和序列。

课程安排

- 数据字典简介
- 在字典视图中查询以下信息：
  - 表信息
  - 列信息
  - 约束条件信息
- 在字典视图中查询以下信息：
  - 视图信息
  - 序列信息
  - 同义词信息
  - 索引信息
- 在表中添加注释以及在字典视图中查询注释信息

ORACLE

表信息

USER_TABLES:

```
DESCRIBE user_tables
```

```
Name                           Null     Type
------------------------------ -------- -------------
TABLE_NAME                     NOT NULL VARCHAR2(30)
TABLESPACE_NAME                         VARCHAR2(30)
CLUSTER_NAME                            VARCHAR2(30)
IOT_NAME                                VARCHAR2(30)
...
```

```
SELECT table_name
FROM   user_tables;
```

	TABLE_NAME
1	REGIONS
2	LOCATIONS
3	DEPARTMENTS
4	JOBS
5	EMPLOYEES
6	JOB_HISTORY

...

ORACLE

表信息

可以使用 USER_TABLES 视图来获得所有表的名称。USER_TABLES 视图包含关于表的信息。除了提供表名外，该视图还包含关于存储空间的详细信息。

TABS 视图是 USER_TABLES 视图的同义词。在该视图中进行查询可查看您所拥有表的列表：

```
SELECT table_name
FROM  tabs;
```

注：为了查看 USER_TABLES 视图中列的完整列表，请参阅《Oracle 数据库参考》中的“USER_TABLES”。

此外，在 ALL_TABLES 视图中进行查询可查看您拥有访问权限的所有表的列表。

列信息

USER_TAB_COLUMNS:

```
DESCRIBE user_tab_columns
```

Name	Null	Type
TABLE_NAME	NOT NULL	VARCHAR2(30)
COLUMN_NAME	NOT NULL	VARCHAR2(30)
DATA_TYPE		VARCHAR2(106)
DATA_TYPE_MOD		VARCHAR2(3)
DATA_TYPE_OWNER		VARCHAR2(30)
DATA_LENGTH	NOT NULL	NUMBER
DATA_PRECISION		NUMBER
DATA_SCALE		NUMBER
NULLABLE		VARCHAR2(1)

...

列信息

在 USER_TAB_COLUMNS 视图中进行查看可查找关于表中各个列的详细信息。虽然 USER_TABLES 视图提供关于表名和存储空间的信息，但在 USER_TAB_COLUMNS 视图中可找到详细的列信息。

此视图包含以下信息：

- 列名
- 列数据类型
- 数据类型的长度
- NUMBER 列的精度和小数位数
- 是否允许空值（即列上是否存在 NOT NULL 约束条件？）
- 默认值

注：为了查看 USER_TAB_COLUMNS 视图中列的完整列表和描述，请参阅《Oracle 数据库参考》中的“USER_TAB_COLUMNS”。

列信息

```
SELECT column_name, data_type, data_length,
       data_precision, data_scale, nullable
FROM   user_tab_columns
WHERE  table_name = 'EMPLOYEES';
```

	COLUMN_NAME	DATA_TYPE	DATA_LENGTH	DATA_PRECISION
1	EMPLOYEE_ID	NUMBER	22	6
2	FIRST_NAME	VARCHAR2	20	(null)
3	LAST_NAME	VARCHAR2	25	(null)
4	EMAIL	VARCHAR2	25	(null)
5	PHONE_NUMBER	VARCHAR2	20	(null)
6	HIRE_DATE	DATE	7	(null)
7	JOB_ID	VARCHAR2	10	(null)
8	SALARY	NUMBER	22	8
9	COMMISSION_PCT	NUMBER	22	2
10	MANAGER_ID	NUMBER	22	6
11	DEPARTMENT_ID	NUMBER	22	4

ORACLE

列信息（续）

通过在 USER_TAB_COLUMNS 表中进行查询，可以查找关于列的详细信息，如列的名称、数据类型、数据类型长度、空值约束条件和默认值。

给出示例中显示了 EMPLOYEES 表的列、数据类型、数据长度和空值约束条件。请注意，此信息与 DESCRIBE 命令产生的输出相似。

要查看关于设置为“未使用”列的信息，请使用 USER_UNUSED_COL_TABS 字典视图。

注：数据字典中对象的名称均为大写。

约束条件信息

- USER_CONSTRAINTS 描述表上的约束条件定义。
- USER_CONS_COLUMNS 描述您所拥有且在约束条件中指定的列。

```
DESCRIBE user_constraints
```

```
Name                           Null     Type
------------------------------ -------- ----------------------
OWNER                          NOT NULL VARCHAR2(30)
CONSTRAINT_NAME                NOT NULL VARCHAR2(30)
CONSTRAINT_TYPE                         VARCHAR2(1)
TABLE_NAME                     NOT NULL VARCHAR2(30)
SEARCH_CONDITION                        LONG()
R_OWNER                                 VARCHAR2(30)
R_CONSTRAINT_NAME                       VARCHAR2(30)
DELETE_RULE                             VARCHAR2(9)
STATUS                                  VARCHAR2(8)
```

...

ORACLE

约束条件信息

可以找到约束条件名称、约束条件类型、约束条件适用的表名称、检查约束条件的条件、外键约束条件信息、外键约束条件删除规则、约束条件状态以及其它类型的约束条件信息。

注：为了查看 USER_CONSTRAINTS 视图中列的完整列表和描述，请参阅《Oracle 数据库参考》中的“USER_CONSTRAINTS”。

USER_CONSTRAINTS：示例

```
SELECT constraint_name, constraint_type,
       search_condition, r_constraint_name,
       delete_rule, status
FROM   user_constraints
WHERE  table_name = 'EMPLOYEES';
```

	CONSTRAINT_NAME	C...	SEARCH_CONDITION	R_CONSTR...	DELET...	STATUS
1	EMP_LAST_NAME_NN	C	"LAST_NAME" IS NOT NULL	(null)	(null)	ENABLED
2	EMP_EMAIL_NN	C	"EMAIL" IS NOT NULL	(null)	(null)	ENABLED
3	EMP_HIRE_DATE_NN	C	"HIRE_DATE" IS NOT NULL	(null)	(null)	ENABLED
4	EMP_JOB_NN	C	"JOB_ID" IS NOT NULL	(null)	(null)	ENABLED
5	EMP_SALARY_MIN	C	salary > 0	(null)	(null)	ENABLED
6	EMP_EMAIL_UK	U	(null)	(null)	(null)	ENABLED
7	EMP_EMP_ID_PK	P	(null)	(null)	(null)	ENABLED
8	EMP_DEPT_FK	R	(null)	DEPT_ID_PK	NO ACTION	ENABLED
9	EMP_JOB_FK	R	(null)	JOB_ID_PK	NO ACTION	ENABLED
10	EMP_MANAGER_FK	R	(null)	EMP_EMP_ID_PK	NO ACTION	ENABLED

USER_CONSTRAINTS：示例

在显示示例中，在 USER_CONSTRAINTS 视图中进行查询的目的是为了查找 EMPLOYEES 表上约束条件的名称、类型、检查条件、外键引用的唯一约束条件的名称、外键删除规则以及状态。

CONSTRAINT_TYPE 可以是：

- C（表上的检查约束条件，或 NOT NULL）
- P（主键）
- U（唯一关键字）
- R（引用完整性）
- V（视图上有检查选项）
- O（视图为只读）

DELETE_RULE 可以是：

- **CASCADE：** 如果删除父记录，还会删除子记录
- **SET NULL：** 如果删除父记录，请将各自的子记录更改为空值
- **NO ACTION：** 只有不存在子记录时才能删除父记录

USER_CONSTRAINTS：示例（续）

STATUS 可以是：

- **ENABLED**：约束条件有效
- **DISABLED**：约束条件无效

在 USER_CONS_COLUMNS 中进行查询

```
DESCRIBE user_cons_columns
```

Name	Null	Type
OWNER	NOT NULL	VARCHAR2(30)
CONSTRAINT_NAME	NOT NULL	VARCHAR2(30)
TABLE_NAME	NOT NULL	VARCHAR2(30)
COLUMN_NAME		VARCHAR2(4000)
POSITION		NUMBER

```
SELECT constraint_name, column_name
FROM   user_cons_columns
WHERE  table_name = 'EMPLOYEES';
```

	CONSTRAINT_NAME	COLUMN_NAME
1	EMP_LAST_NAME_NN	LAST_NAME
2	EMP_EMAIL_NN	EMAIL
3	EMP_HIRE_DATE_NN	HIRE_DATE
4	EMP_JOB_NN	JOB_ID
5	EMP_SALARY_MIN	SALARY
6	EMP_EMAIL_UK	EMAIL

...

ORACLE

在 USER_CONS_COLUMNS 中进行查询

要查找约束条件适用的列的名称，请在 USER_CONS_COLUMNS 字典视图中进行查询。此视图提供约束条件所有者的名称、约束条件名称、有约束条件的表、有约束条件的列的名称以及列或属性在对象定义中的原始位置。

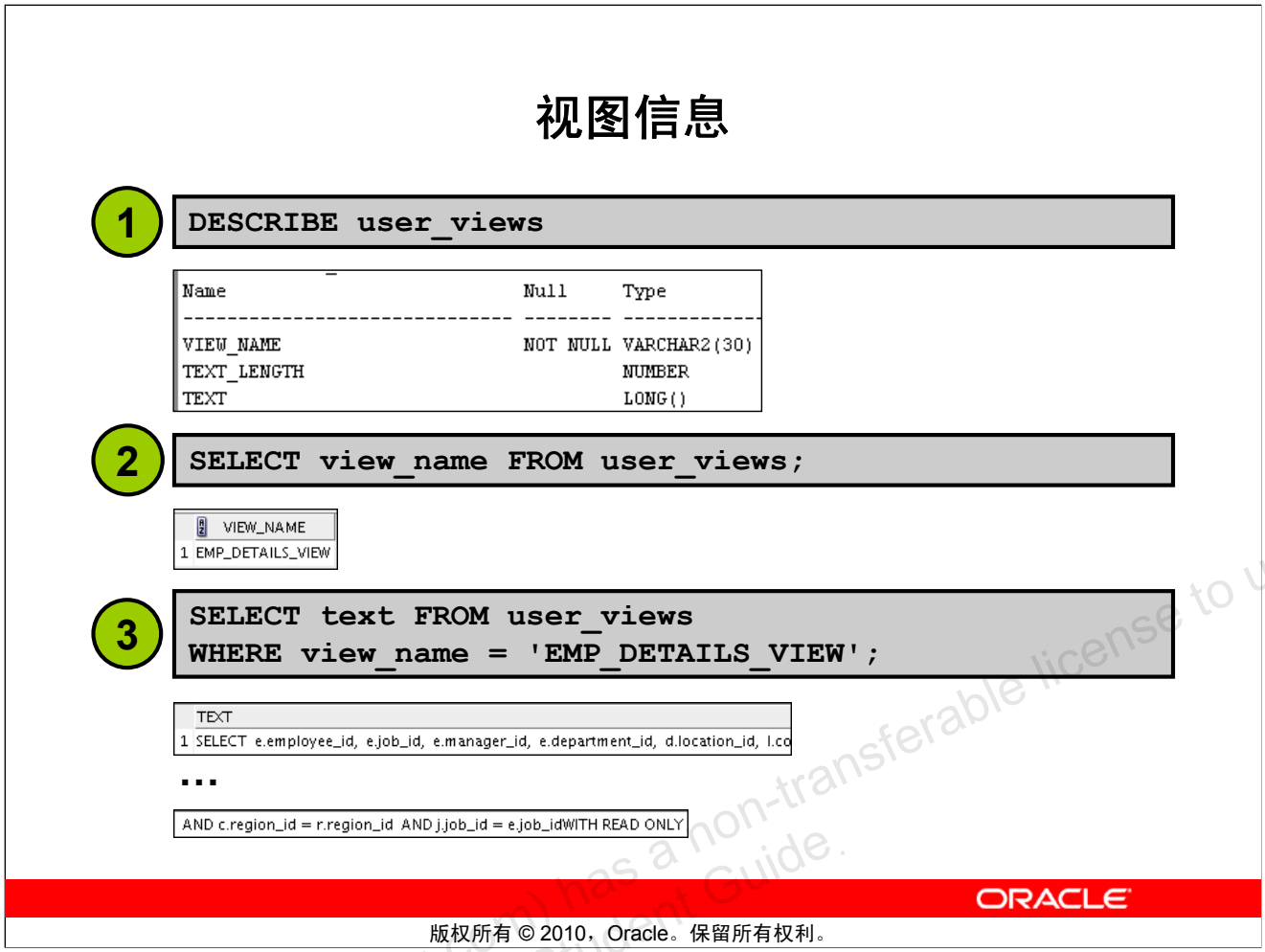
注：一个约束条件可适用于多个列。

此外，可以编写 USER_CONSTRAINTS 和 USER_CONS_COLUMNS 之间的一个联接来创建两个表的自定义输出。

课程安排

- 数据字典简介
- 在字典视图中查询以下信息：
  - 表信息
  - 列信息
  - 约束条件信息
- 在字典视图中查询以下信息：
  - 视图信息
  - 序列信息
  - 同义词信息
  - 索引信息
- 在表中添加注释以及在字典视图中查询注释信息

ORACLE



视图信息

创建视图之后，可以通过在名为 USER_VIEWS 的数据字典视图中进行查询来查看视图名称和视图定义。构成视图的 SELECT 语句的文本会存储在 LONG 列中。LENGTH 列指定了 SELECT 语句中的字符数。默认情况下，在 LONG 列中进行选择时，只会显示列值的前 80 个字符。要在 SQL*Plus 中看到更多的字符（多于 80 个），请使用 SET LONG 命令：

```
SET LONG 1000
```

在幻灯片示例中：

1. 显示了 USER_VIEWS 的各个列。请注意，只列出了其中一部分。
2. 对视图名称进行检索。
3. 显示字典中 EMP_DETAILS_VIEW 的 SELECT 语句。

使用视图访问数据

在使用视图访问数据时，Oracle Server 会执行以下操作：

- 在数据字典表 USER_VIEWS 中检索视图定义。
- 检查视图基表的访问权限。
- 将视图查询转换为对一个或多个基础基表的相应操作。即，从基表中检索数据或对基表进行更新。

序列信息

```
DESCRIBE user_sequences
```

Name	Null	Type
SEQUENCE_NAME	NOT NULL	VARCHAR2(30)
MIN_VALUE		NUMBER
MAX_VALUE		NUMBER
INCREMENT_BY	NOT NULL	NUMBER
CYCLE_FLAG		VARCHAR2(1)
ORDER_FLAG		VARCHAR2(1)
CACHE_SIZE	NOT NULL	NUMBER
LAST_NUMBER	NOT NULL	NUMBER

序列信息

在 USER_SEQUENCES 视图中描述了您拥有的所有序列。在创建序列时，可以指定要存储在 USER_SEQUENCES 视图中的标准。此视图有如下列：

- **SEQUENCE_NAME**：序列的名称。
- **MIN_VALUE**：序列的最小值。
- **MAX_VALUE**：序列的最大值。
- **INCREMENT_BY**：序列递增的增量。
- **CYCLE_FLAG**：在达到限制时序列是否循环使用？
- **ORDER_FLAG**：序列号是否是按顺序生成的？
- **CACHE_SIZE**：要高速缓存的序列号的数量。
- **LAST_NUMBER**：写入磁盘的最后一个序列号。如果序列使用高速缓存，则写入磁盘的编号是序列高速缓存中的最后一个编号。此编号可能大于用过的最后一个序列号。

确认序列

- 验证 USER_SEQUENCES 数据字典表中的序列值。

```
SELECT   sequence_name, min_value, max_value,
         increment_by, last_number
FROM     user_sequences;
```

	SEQUENCE_NAME	MIN_VALUE	MAX_VALUE	INCREMENT_BY	LAST_NUMBER
1	DEPARTMENTS_SEQ	1	9990	10	280
2	EMPLOYEES_SEQ	1	99999999999999...	1	207
3	LOCATIONS_SEQ	1	9900	100	3300

- 如果指定了 NOCACHE，则 LAST_NUMBER 列显示下一个可用的序列号。

确认序列

创建序列之后，就会在数据字典中记录序列。因为序列是一种数据库对象，所以可在 USER_OBJECTS 数据字典表中标识它。

还可以通过在 USER_SEQUENCES 数据字典视图中进行选择来确认序列的设置。

不增加序列值时查看下一个可用序列值

如果序列是使用 NOCACHE 创建的，则在 USER_SEQUENCES 表中进行查询就可以在不增加序列值的情况下查看下一个可用序列值。

索引信息

- USER_INDEXES 提供了关于索引的信息。
- USER_IND_COLUMNS 对组成索引的列以及索引表的列进行了描述。

```
DESCRIBE user_indexes
```

Name	Null	Type
INDEX_NAME	NOT NULL	VARCHAR2(30)
INDEX_TYPE		VARCHAR2(27)
TABLE_OWNER	NOT NULL	VARCHAR2(30)
TABLE_NAME	NOT NULL	VARCHAR2(30)
TABLE_TYPE		VARCHAR2(11)
UNIQUENESS		VARCHAR2(9)

...

索引信息

在 USER_INDEXES 视图中进行查询可找到索引名、创建了索引的表名，还可确认索引是否唯一。

注：为了查看 USER_INDEXES 视图中列的完整列表和描述，请参阅《Oracle Database Reference 11*g* Release 2 (11.1)》中的“USER_INDEXES”。

USER_INDEXES：示例

a

```
SELECT index_name, table_name,uniqueness
FROM   user_indexes
WHERE  table_name = 'EMPLOYEES';
```

	INDEX_NAME	TABLE_NAME	UNIQUENESS
1	EMP_EMAIL_UK	EMPLOYEES	UNIQUE
2	EMP_EMP_ID_PK	EMPLOYEES	UNIQUE
3	EMP_DEPARTMENT_IX	EMPLOYEES	NONUNIQUE
4	EMP_JOB_IX	EMPLOYEES	NONUNIQUE
5	EMP_MANAGER_IX	EMPLOYEES	NONUNIQUE
6	EMP_NAME_IX	EMPLOYEES	NONUNIQUE

b

```
SELECT index_name, table_name
FROM   user_indexes
WHERE  table_name = 'emp_lib';
```

	INDEX_NAME	TABLE_NAME
1	SYS_C0011777	EMP_LIB

ORACLE

USER_INDEXES：示例

在幻灯片示例 **a** 中，在 USER_INDEXES 视图中进行查询的目的是为了查找索引名、创建了索引的表名，以及确认索引是否唯一。

在幻灯片示例 **b** 中，注意 Oracle Server 给为 PRIMARY KEY 列创建的索引提供了一个通用名称。EMP_LIB 表是使用以下代码创建的：

```
CREATE TABLE EMP_LIB
   (book_id NUMBER(6)PRIMARY KEY ,
    title VARCHAR2(25),
    category VARCHAR2(20));
```

```
CREATE TABLE succeeded.
```

在 USER_IND_COLUMNS 中进行查询

```
DESCRIBE user_ind_columns
```

```
Name                           Null     Type
------------------------------ -------- --------------
INDEX_NAME                              VARCHAR2(30)
TABLE_NAME                              VARCHAR2(30)
COLUMN_NAME                             VARCHAR2(4000)
COLUMN_POSITION                         NUMBER
COLUMN_LENGTH                           NUMBER
CHAR_LENGTH                             NUMBER
DESCEND                                 VARCHAR2(4)
```

```
SELECT index_name, column_name,table_name
FROM   user_ind_columns
WHERE  index_name = 'lname_idx';
```

	INDEX_NAME	COLUMN_NAME	TABLE_NAME
1	LNAME_IDX	LAST_NAME	EMP_TEST

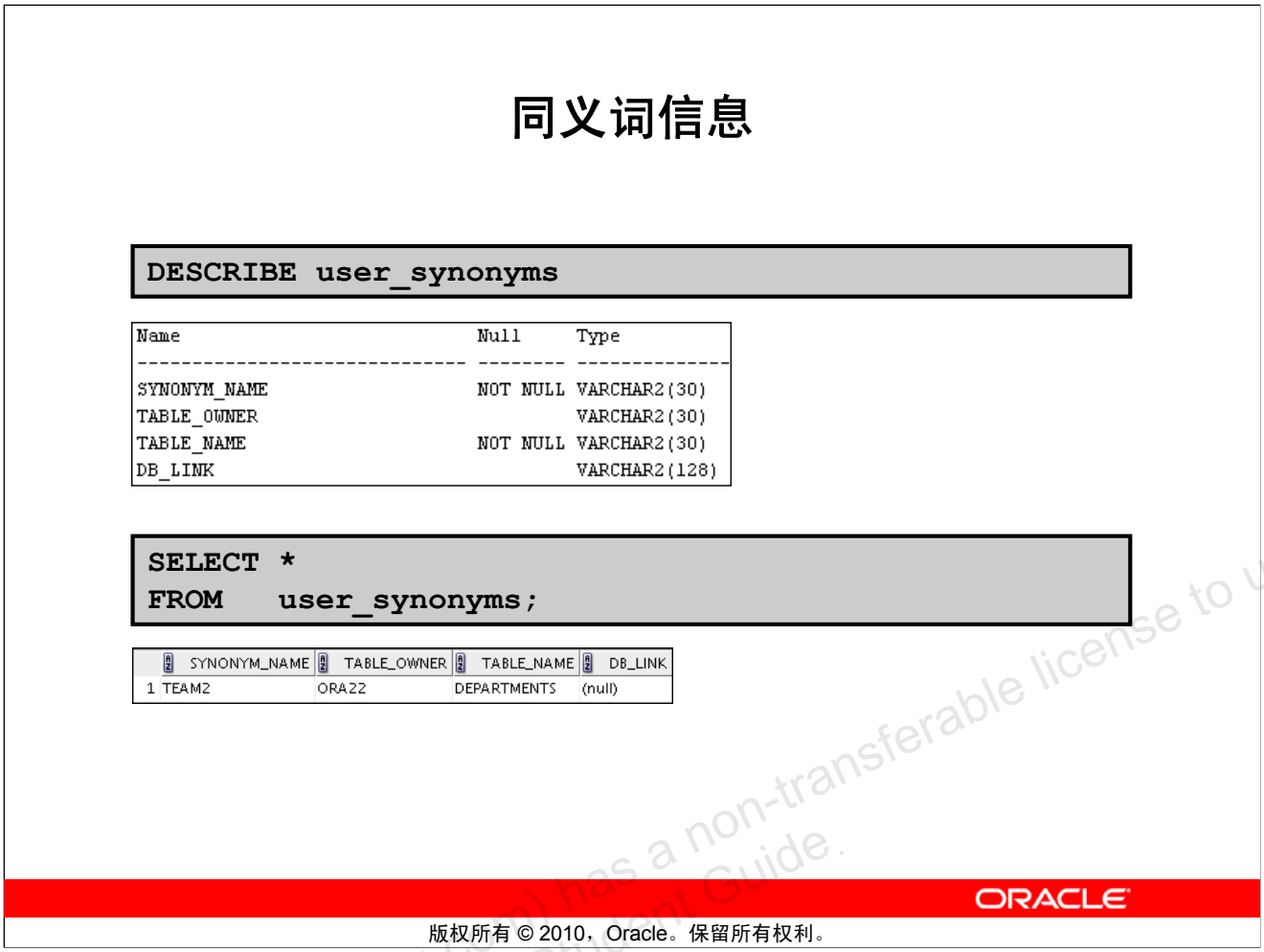
ORACLE

在 USER_IND_COLUMNS 中进行查询

在 USER_IND_COLUMNS 字典视图中提供了各种信息，如索引名、索引表名、索引中列的名称以及索引中列的位置。

在幻灯片示例中，emp_test 表和 LNAME_IDX 索引是使用以下代码创建的：

```
CREATE TABLE emp_test AS SELECT * FROM employees;
CREATE INDEX LNAME_IDX ON emp_test(Last_Name);
```



同义词信息

在 USER_SYNONYMS 字典视图中描述了专用同义词（您所拥有的同义词）。

在这个视图进行查询可以查找您的同义词。在 ALL_SYNONYMS 中进行查询，可找到为您提供的所有同义词的名称以及这些同义词适用的对象。

此视图有如下列：

- **SYNONYM_NAME**：同义词的名称
- **TABLE_OWNER**：同义词引用的对象的所有者
- **TABLE_NAME**：同义词引用的表或视图的名称
- **DB_LINK**：数据库链接引用的名称（如果有）

课程安排

- 数据字典简介
- 在字典视图中查询以下信息：
  - 表信息
  - 列信息
  - 约束条件信息
- 在字典视图中查询以下信息：
  - 视图信息
  - 序列信息
  - 同义词信息
  - 索引信息
- 在表中添加注释以及在字典视图中查询注释信息

ORACLE

在表中添加注释

- 使用 COMMENT 语句可在表或列中添加注释：

```
COMMENT ON TABLE employees
IS 'Employee Information';
```

```
COMMENT ON COLUMN employees.first_name
IS 'First name of the employee';
```

- 可以在以下数据字典视图中查看注释：
  - ALL_COL_COMMENTS
  - USER_COL_COMMENTS
  - ALL_TAB_COMMENTS
  - USER_TAB_COMMENTS

ORACLE

在表中添加注释

使用 COMMENT 语句可为列、表、视图或快照添加最多 4,000 个字节的注释。注释会存储在数据字典中。可以在以下数据字典视图的 COMMENTS 列中查看注释：

- ALL_COL_COMMENTS
- USER_COL_COMMENTS
- ALL_TAB_COMMENTS
- USER_TAB_COMMENTS

语法

```
COMMENT ON {TABLE table | COLUMN table.column}
    IS 'text';
```

在该语法中：

table 是表名称

column 是表中列的名称

text 是注释文本

将注释文本设置为空字符串 ('') 就可以从数据库中删除注释：

```
COMMENT ON TABLE  employees IS '';
```

小测验

基于字典表的字典视图包含以下信息：

1. 数据库中所有方案对象的定义
2. 列的默认值
3. 完整性约束条件信息
4. 授予每个用户的权限和角色
5. 上述所有信息

ORACLE

答案：5

小结

在本课中，您应该已经学会如何在以下字典视图中查找关于对象的信息：

- DICTIONARY
- USER_OBJECTS
- USER_TABLES
- USER_TAB_COLUMNS
- USER_CONSTRAINTS
- USER_CONS_COLUMNS
- USER_VIEWS
- USER_SEQUENCES
- USER_INDEXES
- USER_SYNONYMS

ORACLE

小结

在本课中，您对提供的一些字典视图已经有所了解。可以使用这些字典视图来查找关于表、约束条件、视图、序列和同义词的信息。

练习 3：概览

本练习包含以下主题：

- 在字典视图中查询表和列的信息
- 在字典视图中查询约束条件信息
- 在字典视图中查询视图信息
- 在字典视图中查询序列信息
- 在字典视图中查询同义词信息
- 在字典视图中查询索引信息
- 在表中添加注释以及在字典视图中查询注释信息

ORACLE

练习 3：概览

在本练习中，将通过在字典视图中进行查询来查找关于方案对象的信息。

4

处理大型数据集

课程目标

学完本课后，应能完成下列工作：

- 使用子查询处理数据
- 在 INSERT 和 UPDATE 语句中指定显式默认值
- 描述多表 INSERT 功能
- 使用以下类型的多表 INSERT：
  - 无条件 INSERT
  - 转换 INSERT (Pivoting INSERT)
  - 条件 INSERT ALL
  - 条件 INSERT FIRST
- 合并表行
- 跟踪一段时间内的数据更改

ORACLE

课程目标

在本课中，您要学习如何使用子查询处理 Oracle DB 中的数据。学习如何在 INSERT 和 UPDATE 语句中使用 DEFAULT 关键字来确定默认列值。还要了解多表 INSERT 语句、MERGE 语句以及如何跟踪数据库更改。

课程安排

- 使用子查询处理数据
- 在 INSERT 和 UPDATE 语句中指定显式默认值
- 使用以下类型的多表 INSERT：
  - 无条件 INSERT
  - 转换 INSERT (Pivoting INSERT)
  - 条件 INSERT ALL
  - 条件 INSERT FIRST
- 合并表行
- 跟踪一段时间内的数据更改

使用子查询处理数据

在数据操纵语言 (DML) 语句中使用子查询可：

- 使用内嵌视图检索数据
- 将数据从一个表复制到另一个表中
- 根据一个表中的值更新另一个表中的数据
- 根据一个表中的行删除另一个表中的行

使用子查询处理数据

使用子查询可在一个表中检索数据，检索到的数据可用作 INSERT 语句的输入而插入在另一个表中。采用这种方法时，使用一条 SELECT 语句即可轻易地将大量数据从一个表复制到另一个表中。同样，在 UPDATE 和 DELETE 语句的 WHERE 子句中使用子查询，可以成批地执行更新和删除。此外，还可以在 SELECT 语句的 FROM 子句中使用子查询。这被称为内嵌视图。

注：在《Oracle Database 11*g*：SQL 基础 I》课程中已介绍过如何根据另一个表来更新和删除行。

通过将子查询用作源来检索数据

```
SELECT department_name, city
FROM   departments
NATURAL JOIN (SELECT l.location_id, l.city, l.country_id
              FROM   loc l
              JOIN   countries c
              ON(l.country_id = c.country_id)
              JOIN regions USING(region_id)
              WHERE region_name = 'Europe');
```

	DEPARTMENT_NAME	CITY
1	Human Resources	London
2	Sales	Oxford
3	Public Relations	Munich

通过将子查询用作源来检索数据

可以在 SELECT 语句的 FROM 子句中使用子查询，这与使用视图的方式非常类似。SELECT 语句的 FROM 子句中的子查询也被称为内嵌视图。SELECT 语句的 FROM 子句中的子查询为特定的 SELECT 语句且只为该语句定义了数据源。与数据库视图一样，子查询中的 SELECT 语句可以是简单语句，也可以是复杂语句，这取决于您的需要。

创建数据库视图时，关联的 SELECT 语句会存储在数据字典中。如果您没有创建数据库视图所必需的权限，或者您希望测试 SELECT 语句成为一个视图的适用性，则可使用内嵌视图。

使用内嵌视图时，在一个位置就可以获得支持查询所需要的全部代码。也就是说，可以免去创建另外的数据库视图的复杂过程。幻灯片中的示例显示了如何使用内嵌视图显示欧洲区域内部门的名称和所在城市。FROM 子句中的子查询通过联接三个不同的表可提取位置 ID、城市名称和国家/地区。内部查询的输出结果被当作是外部查询要使用的一个表。内部查询类似于数据库视图查询，但没有指定实际名称。

幻灯片示例中的 loc 表是通过运行以下语句创建的：

```
CREATE TABLE loc AS SELECT * FROM locations;
```

通过将子查询用作源来检索数据（续）

通过执行以下两个步骤，可以显示与幻灯片示例一样的输出结果：

1. 创建数据库视图：

```
CREATE OR REPLACE VIEW european_cities
AS
SELECT l.location_id, l.city, l.country_id
FROM   loc l
JOIN   countries c
ON(l.country_id = c.country_id)
JOIN regions USING(region_id)
WHERE region_name = 'Europe';
```

2. 将 EUROPEAN_CITIES 视图与 DEPARTMENTS 表联接：

```
SELECT department_name, city
FROM   departments
NATURAL JOIN european_cities;
```

注：在《Oracle Database 11*g*：SQL 基础 I》课程中已介绍过如何创建数据库视图。

通过将子查询用作目标来执行插入

```
INSERT INTO (SELECT l.location_id, l.city, l.country_id
             FROM   locations l
             JOIN   countries c
             ON(l.country_id = c.country_id)
             JOIN regions USING(region_id)
             WHERE region_name = 'Europe')
VALUES (3300, 'Cardiff', 'UK');
```

```
1 rows inserted
```


通过将子查询用作目标来执行插入

您可以用一个子查询替换 INSERT 语句的 INTO 子句中的表名。这个子查询中 SELECT 列表的列数必须与 VALUES 子句中列列表的列数相同。为了让 INSERT 语句成功执行，必须遵守基表列上的所有规则。例如，不能输入重复的位置 ID，也不能在强制实施 NOT NULL 的列中遗漏值。

这样使用子查询时，可以不必只为执行 INSERT 而创建视图。

幻灯片中的示例使用一个子查询代替 LOC 来创建一条新欧洲城市记录。

注：还可以使用以下代码在 EUROPEAN_CITIES 视图上执行 INSERT 操作：

```
INSERT INTO european_cities
VALUES (3300,'Cardiff','UK');
```

通过将子查询用作目标来执行插入

请验证结果。

```
SELECT location_id, city, country_id
FROM   loc
```

	LOCATION_ID	CITY	COUNTRY_ID
20	2900	Geneva	CH
21	3000	Bern	CH
22	3100	Utrecht	NL
23	3200	Mexico City	MX
24	3300	Cardiff	UK

ORACLE

通过将子查询用作目标来执行插入（续）

幻灯片中的示例显示通过内嵌视图执行插入而在基表 LOC 中创建一条新记录。

以下示例显示子查询的结果，该结果被用做 INSERT 语句中使用的表。

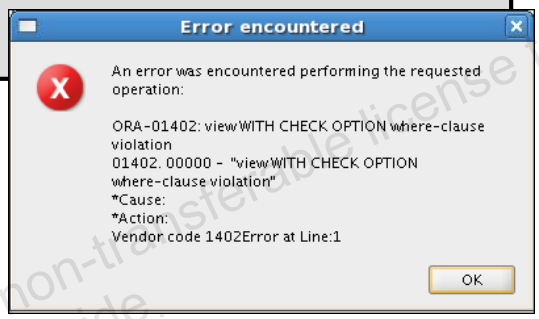
```
SELECT l.location_id, l.city, l.country_id
FROM   loc l
JOIN   countries c
ON(l.country_id = c.country_id)
JOIN regions USING(region_id)
WHERE region_name = 'Europe'
```

	LOCATION_ID	CITY	COUNTRY_ID
6	2700	Munich	DE
7	2900	Geneva	CH
8	3000	Bern	CH
9	3100	Utrecht	NL
10	3300	Cardiff	UK

在 DML 语句中使用 WITH CHECK OPTION 关键字

使用 WITH CHECK OPTION 关键字可禁止对不在子查询中的行进行更改。

```
INSERT INTO ( SELECT location_id, city, country_id
              FROM   loc
              WHERE  country_id IN
              (SELECT country_id
               FROM countries
               NATURAL JOIN regions
               WHERE region_name = 'Europe')
               WITH CHECK OPTION )
VALUES (3600, 'Washington', 'US');
```



ORACLE

在 DML 语句中使用 WITH CHECK OPTION 关键字

指定 WITH CHECK OPTION 关键字时表示，使用子查询代替 INSERT、UPDATE 或 DELETE 语句中的一个表时，允许在该表上进行的所有更改都不会生成在子查询中不包括的行。

幻灯片中的示例显示如何使用内嵌视图与 WITH CHECK OPTION。该 INSERT 语句可避免在 LOC 表中创建对应于非欧洲城市的记录。

以下示例因 VALUES 列表已更改而成功执行。

```
INSERT INTO (SELECT location_id, city, country_id
             FROM   loc
             WHERE  country_id IN
             (SELECT country_id
              FROM countries
              NATURAL JOIN regions
              WHERE region_name = 'Europe')
              WITH CHECK OPTION )
VALUES (3500, 'Berlin', 'DE');
```

在 DML 语句中使用 WITH CHECK OPTION 关键字（续）

通过在使用内嵌视图时使用 WITH CHECK OPTION，为避免更改表提供了一种简单的方法。

为避免创建非欧洲城市记录，还可以使用数据库视图。为此请执行以下步骤：

1. 创建数据库视图：

```
CREATE OR REPLACE VIEW european_cities
AS
SELECT location_id, city, country_id
FROM   locations
WHERE  country_id in
  (SELECT country_id
   FROM countries
   NATURAL JOIN regions
   WHERE region_name = 'Europe')
WITH CHECK OPTION;
```

2. 在插入数据时验证结果：

```
INSERT INTO european_cities
VALUES (3400,'New York','US');
```

第二步会产生与幻灯片中一样的错误。

课程安排

- 使用子查询处理数据
- 在 INSERT 和 UPDATE 语句中指定显式默认值
- 使用以下类型的多表 INSERT：
  - 无条件 INSERT
  - 转换 INSERT (Pivoting INSERT)
  - 条件 INSERT ALL
  - 条件 INSERT FIRST
- 合并表行
- 跟踪一段时间内的数据更改

显式默认值功能概览

- 在需要默认列值之处使用 DEFAULT 关键字作为列值。
- 此功能允许用户控制应用数据默认值的位置和时间。
- 显式默认值可用在 INSERT 和 UPDATE 语句中。

显式默认值

在 INSERT 和 UPDATE 语句中可使用 DEFAULT 关键字标识默认列值。如果不存在默认值，则会使用空值。

使用 DEFAULT 选项减少了工作量，您不必像未引入此功能之前那样，必须要在程序中对默认值进行硬编码，或者要通过查询字典来查找默认值。如果默认值更改了，对默认值进行硬编码时就面临一个问题，因为代码也需要进行相应的更改。有时在应用程序中不能访问字典，因此这是一项极其重要的功能。

使用显式默认值

- DEFAULT 与 INSERT 配合使用：

```
INSERT INTO deptm3
  (department_id, department_name, manager_id)
VALUES (300, 'Engineering', DEFAULT);
```

- DEFAULT 与 UPDATE 配合使用：

```
UPDATE deptm3
SET manager_id = DEFAULT
WHERE department_id = 10;
```

ORACLE

使用显式默认值

指定 DEFAULT 可将先前指定的列值设置为列的默认值。如果尚未指定相应列的默认值，Oracle Server 会将此列设置为空值。

在幻灯片第一个示例中，INSERT 语句在 MANAGER_ID 列中使用了默认值。如果没有定义此列的默认值，则会插入空值。

第二个示例中使用 UPDATE 语句将 MANAGER_ID 列设置为默认值，即部门 10。如果没有定义此列的默认值，该列会更改为空值。

注：创建表时，可以指定列的默认值。《SQL 基础 I》中已介绍过此内容。

从其它表中复制行

- 编写包含子查询的 INSERT 语句。

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

```
33 rows inserted
```

- 请勿使用 VALUES 子句。
- 使 INSERT 子句中的列数与子查询中的列数相匹配。

从其它表中复制行

可以使用 INSERT 语句在一个表中添加一些行，其中的值来自现有表。可以使用子查询代替 VALUES 子句。

语法

```
INSERT INTO table [ column (, column) ] subquery;
```

在该语法中：

table	是表名
column	是表中要填充的列的名称
subquery	是在表中返回行的子查询

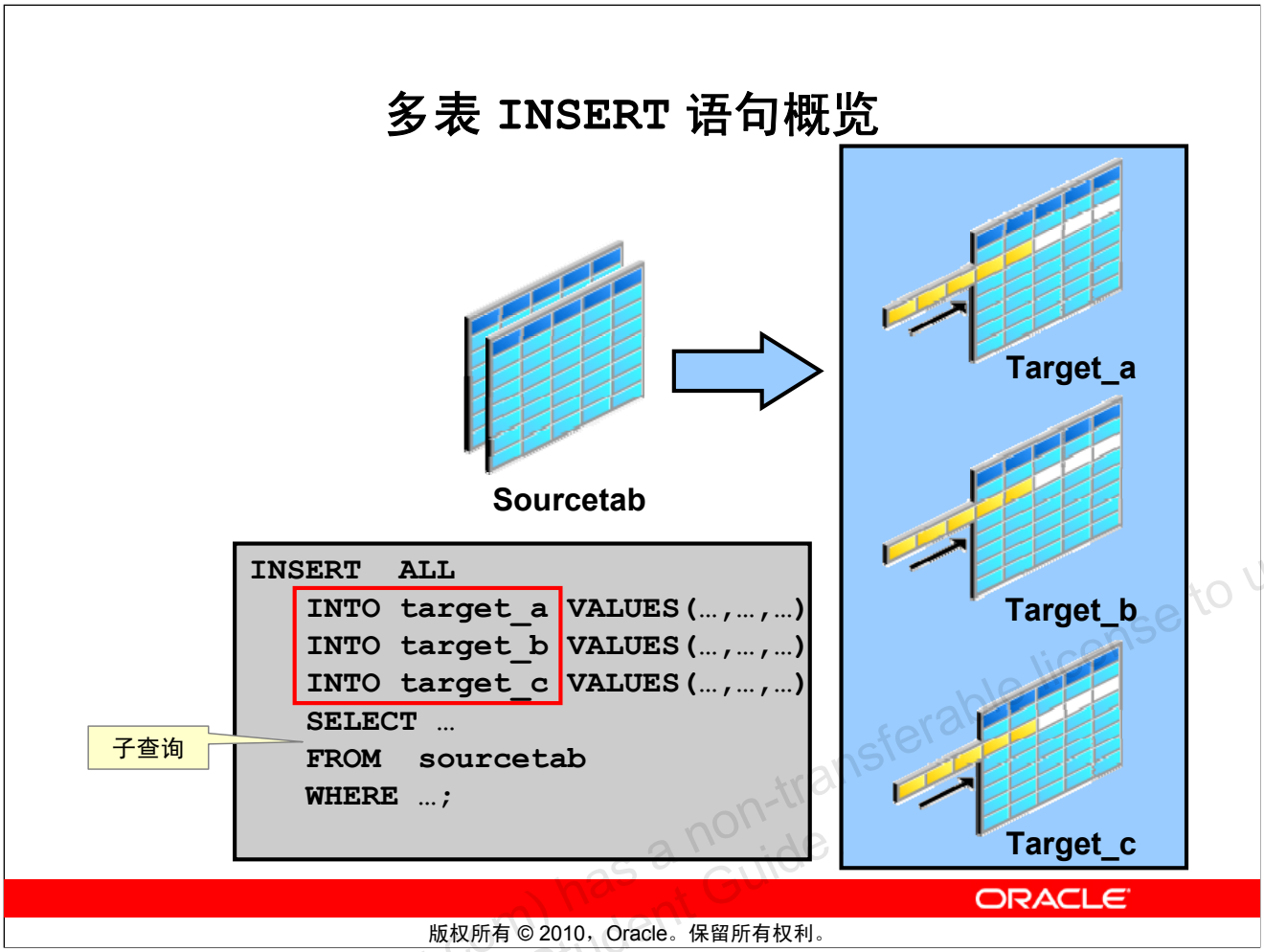
INSERT 子句中列列表的列数及其数据类型必须与子查询中值的个数及其数据类型相匹配。要创建包含表行的副本，请在子查询中使用 SELECT *。

```
INSERT INTO EMPL3
   SELECT *
   FROM   employees;
```

注：在 DML 语句中使用 LOG ERRORS 子句时可允许 DML 操作完成，而不管是否发生错误。Oracle 会将详细的错误消息写入已创建的错误日志表中。有关详细信息，请参阅《Oracle Database 11*g* SQL Reference》。

课程安排

- 使用子查询处理数据
- 在 INSERT 和 UPDATE 语句中指定显式默认值
- 使用以下类型的多表 INSERT：
  - 无条件 INSERT
  - 转换 INSERT (Pivoting INSERT)
  - 条件 INSERT ALL
  - 条件 INSERT FIRST
- 合并表行
- 跟踪一段时间内的数据更改



多表 INSERT 语句概览

在多表 INSERT 语句中，可以将计算求出的某些行插入在一个或多个表中，这些行是根据子查询计算操作返回的行而得到的那些行。

多表 INSERT 语句在使用数据仓库情况下很有用。您需要定期加载数据仓库，使之可以达到帮助进行业务分析的目的。为此，必须从一个或多个操作系统中提取数据，然后在数据仓库中复制数据。通常将从源系统提取数据后在数据仓库中复制数据的过程称为 ETL，ETL 代表提取 (extraction)、转换 (transformation) 和加载 (loading)。

在提取过程中，必须在多个不同的源（如数据库系统和应用程序）中确定所需的数据，然后进行提取。提取后，数据必须实际传输到目标系统或中间系统做进一步的处理。根据所选的传输方法，在此过程中可以执行某些转换。例如，一条 SQL 语句通过网关直接访问远程目标时可在 SELECT 语句中将两列连接起来。

在 Oracle DB 中加载数据后，可以使用 SQL 操作执行数据转换。多表 INSERT 语句是用于实现 SQL 数据转换的一种方式。

多表 INSERT 语句概览

- 使用 INSERT…SELECT 语句可在一条 DML 语句的多个表中插入若干行。
- 在数据仓库系统中使用多表 INSERT 语句可将数据从一个或多个操作源传送到一组目标表中。
- 多表 INSERT 语句带来的性能改进明显地高于：
  - 一条 DML 语句与多条 INSERT…SELECT 语句
  - 一条 DML 语句与一个使用 IF…THEN 语法执行多次插入的过程

多表 INSERT 语句概览（续）

在目标中包含多个表时，多表 INSERT 语句体现了使用 INSERT ... SELECT 语句的好处。如果没有多表 INSERT，则必须处理 *n* 个独立的 INSERT ... SELECT 语句，因此必须对同一源数据处理 *n* 次，因而转换工作量也增加 *n* 倍。

相对于现有的 INSERT ... SELECT 语句，新语句可在使用直接加载机制时并行使用，因此性能更高。

任意输入流中的每一条记录，如非关系数据库表中的一条记录，现在都可以转换为更大的关系数据库表环境中的多条记录。要用其它方法实现此功能，需要编写多条 INSERT 语句。

多表 INSERT 语句的类型

不同类型的多表 INSERT 语句包括：

- 无条件 INSERT
- 条件 INSERT ALL
- 转换 INSERT (Pivoting INSERT)
- 条件 INSERT FIRST

多表 INSERT 语句的类型

使用不同的子句可指出要执行的 INSERT 语句的类型。多表 INSERT 语句的类型包括：

- **无条件 INSERT：**对于子查询返回的每一行，在每一个目标表中插入一行。
- **条件 INSERT ALL：**对于子查询返回的每一行，在满足指定条件的每一个目标表中插入一行。
- **转换 INSERT：**这是无条件 INSERT ALL 的一个特例。
- **条件 INSERT FIRST：**对于子查询返回的每一行，在满足条件的第一个目标表中插入一行。

多表 INSERT 语句

- 多表 INSERT 的语法：

```
INSERT [conditional_insert_clause]
[insert_into_clause values_clause] (subquery)
```

- conditional_insert_clause：

```
[ALL|FIRST]
[WHEN condition THEN] [insert_into_clause values_clause]
[ELSE] [insert_into_clause values_clause]
```


多表 INSERT 语句

幻灯片中显示的是多表 INSERT 语句的通用格式。

无条件 INSERT：ALL into_clause

指定 ALL 后面跟着执行无条件多表 INSERT 的多条 insert_into_clause。Oracle Server 会为子查询返回的每一行而执行一次每一条 insert_into_clause。

条件 INSERT：conditional_insert_clause

指定执行条件多表 INSERT 的 conditional_insert_clause。Oracle Server 根据相应的 WHEN 条件过滤每一条 insert_into_clause，以确定是否执行该 insert_into_clause。一条多表 INSERT 语句最多可包含 127 条 WHEN 子句。

条件 INSERT：ALL

如果指定了 ALL，Oracle Server 会对每一条 WHEN 子句进行计算，而不管其它任何 WHEN 子句的计算结果如何。对于每一条条件计算结果为真的 WHEN 子句，Oracle Server 会执行相应的 INTO 子句列表。

多表 INSERT 语句（续）

条件 INSERT：FIRST

如果指定了 FIRST，Oracle Server 会按照每一条 WHEN 子句在语句中出现的顺序对其进行计算。如果第一条 WHEN 子句的计算结果为真，Oracle Server 会执行相应的 INTO 子句，并跳过指定行的后续 WHEN 子句。

条件 INSERT：ELSE 子句

如果对于指定行没有计算结果为真的 WHEN 子句：

- 如果已指定 ELSE 子句，Oracle Server 会执行与 ELSE 子句相关联的 INTO 子句列表
- 如果未指定 ELSE 子句，Oracle Server 不会对该行执行任何操作

多表 INSERT 语句的限制

- 只能对表执行多表 INSERT 语句，而不能对视图或实体化视图执行多表 INSERT 语句。
- 不能对远程表执行多表 INSERT。
- 执行多表 INSERT 时，不能指定表集合表达式。
- 在多表 INSERT 中，所有 insert_into_clauses 指定的目标列加起来不能超过 999 个。

无条件 INSERT ALL

- 在 EMPLOYEES 表中，选择 EMPLOYEE_ID 大于 200 的雇员的 EMPLOYEE_ID、HIRE_DATE、SALARY 和 MANAGER_ID 值。
- 通过使用多表 INSERT 在 SAL_HISTORY 和 MGR_HISTORY 表中插入这些值。

```
INSERT  ALL
   INTO sal_history VALUES(EMPID,HIREDATE,SAL)
   INTO mgr_history VALUES(EMPID,MGR,SAL)
   SELECT employee_id EMPID, hire_date HIREDATE,
          salary SAL, manager_id MGR
   FROM  employees
   WHERE employee_id > 200;
```

```
12 rows inserted
```

ORACLE

无条件 INSERT ALL

幻灯片示例中在 SAL_HISTORY 和 MGR_HISTORY 表中插入了若干行。

SELECT 语句在 EMPLOYEES 表中检索雇员 ID 大于 200 的那些雇员的雇员 ID、薪金和经理 ID 之类的详细资料。雇员 ID、受雇日期和薪金之类的详细资料会插入在 SAL_HISTORY 表中。雇员 ID、经理 ID 和薪金之类的详细资料会插入在 MGR_HISTORY 表中。

这种 INSERT 语句称为无条件 INSERT，因为对 SELECT 语句检索到的那些行没有再应用任何限制。SELECT 语句检索到的所有行都会插入在这两个表中：SAL_HISTORY 和 MGR_HISTORY。INSERT 语句中的 VALUES 子句指定了 SELECT 语句中必须插入在每一个表中的那些列。SELECT 语句返回的每一行都导致执行两个插入操作：一个在 SAL_HISTORY 表中执行，另一个在 MGR_HISTORY 表执行。

无条件 INSERT ALL（续）

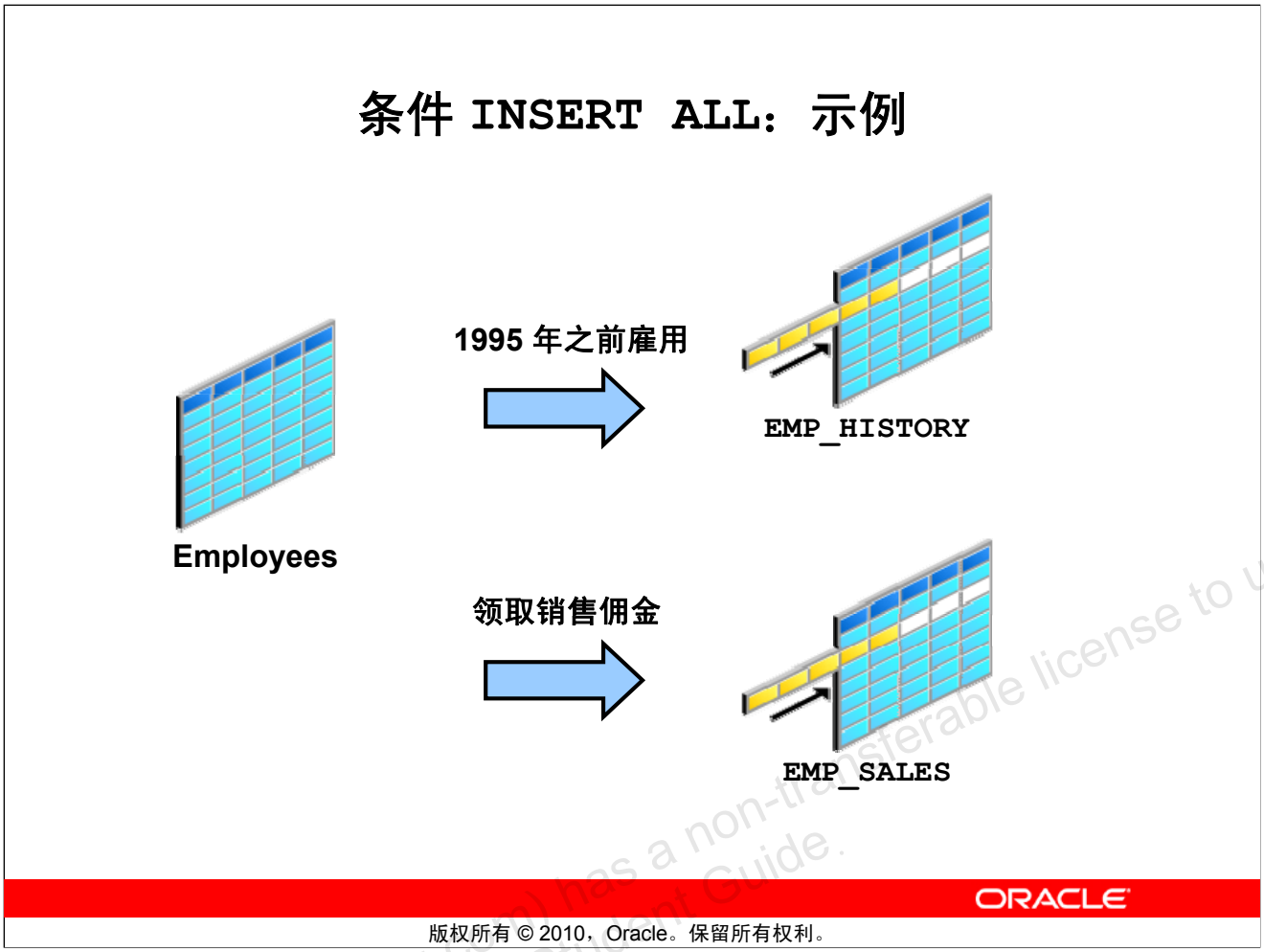
总共选定了 12 行：

```
SELECT COUNT(*) total_in_sal FROM sal_history;
```

	TOTAL_IN_SAL
1	6

```
SELECT COUNT(*) total_in_mgr FROM mgr_history;
```

	TOTAL_IN_MGR
1	6



条件 INSERT ALL：示例

对于 employees 表中的所有雇员，如果雇员的受雇日期在 1995 年之前，则该雇员记录插入在雇员历史记录中。如果雇员领取销售佣金，则该记录信息插入在 EMP_SALES 表中。SQL 语句显示在下一页上。

条件 INSERT ALL

```
INSERT  ALL
 WHEN HIREDATE < '01-JAN-95' THEN
   INTO emp_history VALUES(EMPID,HIREDATE,SAL)
 WHEN COMM IS NOT NULL THEN
   INTO emp_sales VALUES(EMPID,COMM,SAL)
   SELECT employee_id EMPID, hire_date HIREDATE,
          salary SAL, commission_pct COMM
   FROM  employees
```

```
48 rows inserted
```

ORACLE

条件 INSERT ALL

幻灯片中的示例类似于上一张幻灯片中的示例，因为也是在 EMP_HISTORY 表和 EMP_SALES 表中插入若干行。SELECT 语句在 EMPLOYEES 表中检索所有雇员的雇员 ID、受雇日期和佣金百分比等详细资料。雇员 ID、受雇日期和薪金之类的详细资料会插入在 EMP_HISTORY 表中。雇员 ID、佣金百分比和薪金之类的详细资料会插入在 EMP_SALES 表中。

这种 INSERT 语句称为条件 INSERT ALL，因为对 SELECT 语句检索到的那些行又应用了限制。在 SELECT 语句检索到的那些行中，只有受雇日期早于 1995 年的那些行才会插入在 EMP_HISTORY 表中。类似地，只有佣金百分比值不为空的那些行才会插入在 EMP_SALES 表中。

```
SELECT count(*) FROM emp_history;
```

	COUNT(*)
1	13

```
SELECT count(*) FROM emp_sales;
```

	COUNT(*)
1	35

条件 INSERT ALL（续）

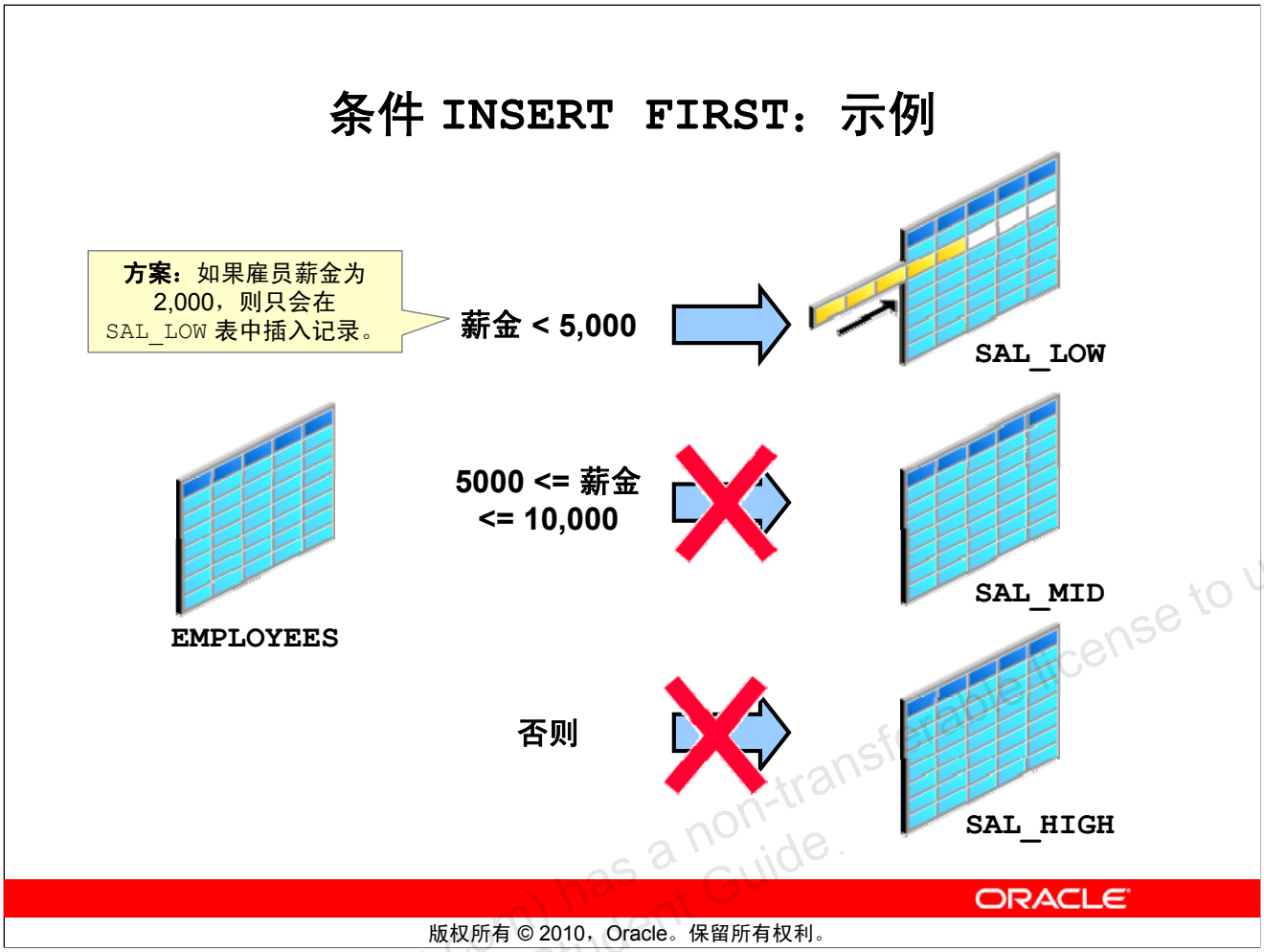
您还可以选择结合使用 ELSE 子句与 INSERT ALL 语句。

示例：

```
INSERT ALL
WHEN job_id IN
(select job_id FROM jobs WHERE job_title LIKE '%Manager%') THEN
INTO managers2(last_name,job_id,SALARY)
VALUES (last_name,job_id,SALARY)
WHEN SALARY>10000 THEN
INTO richpeople(last_name,job_id,SALARY)
VALUES (last_name,job_id,SALARY)
ELSE
INTO poorpeople VALUES (last_name,job_id,SALARY)
SELECT * FROM employees;
```

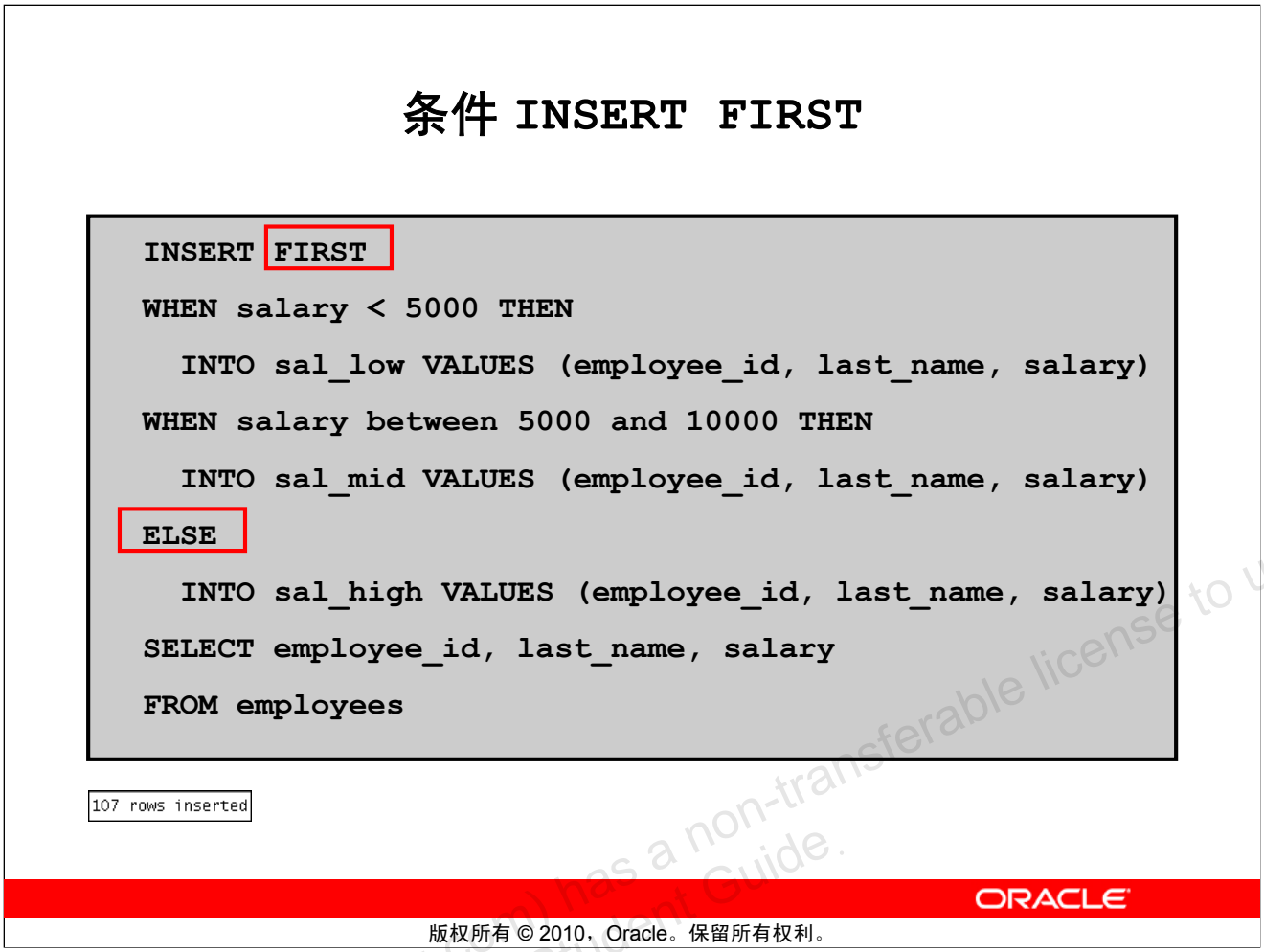
结果：

```
插入了 116 行
```



条件 INSERT FIRST：示例

对于 EMPLOYEES 表中的所有雇员，会在满足条件的第一个目标表中插入相关的雇员信息。在示例中，如果雇员薪金为 2,000，则只会在 SAL_LOW 表中插入记录。SQL 语句显示在下一页上。



条件 INSERT FIRST

SELECT 语句在 EMPLOYEES 表中检索每个雇员的雇员 ID、姓氏和薪金之类的详细资料。每一条雇员记录都会插入在满足条件的第一个目标表中。

这种 INSERT 语句称为条件 INSERT FIRST。WHEN salary < 5000 条件是第一个计算条件。如果第一个 WHEN 子句的计算结果为真，Oracle Server 会执行相应的 INTO 子句，并在 SAL_LOW 表中插入记录。它会跳过此行的后续 WHEN 子句。

如果该行不满足第一个 WHEN 条件 (WHEN salary < 5000)，则会计算下一个条件 (WHEN salary between 5000 and 10000)。如果此条件的计算结果为真，则会在 SAL_MID 表插入记录，并跳过最后一个条件。

如果第一个条件 (WHEN salary < 5000) 和第二个条件 (WHEN salary between 5000 and 10000) 的计算结果都不为真，Oracle Server 就会执行 ELSE 子句中的相应 INTO 子句。

条件 INSERT FIRST（续）

总共插入了 20 行：

```
SELECT count(*) low FROM sal_low;
```

	LOW
1	49

```
SELECT count(*) mid FROM sal_mid;
```

	MID
1	43

```
SELECT count(*) high FROM sal_high;
```

	HIGH
1	15

转换 INSERT (Pivoting INSERT)

将非关系数据库表中的一组销售记录转换为关系格式。

Emp_ID	Week_ID	MON	TUES	WED	THUR	FRI
176	6	2000	3000	4000	5000	6000

Employee_ID	WEEK	SALES
176	6	2000
176	6	3000
176	6	4000
176	6	5000
176	6	6000

ORACLE

转换 INSERT (Pivoting INSERT)

转换 (Pivoting) 是一种操作，在这种操作过程中必须建立一种转换机制，使得来自任何输入流（如非关系数据库表）的每一条记录都必须转换为更大的关系数据库表环境中的多条记录。

假设您从非关系数据库表中收到了一组销售记录：

```
SALES_SOURCE_DATA，其格式如下：
      EMPLOYEE_ID, WEEK_ID, SALES_MON, SALES_TUE, SALES_WED,
SALES_THUR, SALES_FRI
```

您希望采用下列更典型的关系格式在 SALES_INFO 表中存储这些记录：

```
EMPLOYEE_ID、WEEK、SALES
```

要解决此问题，必须建立一种转换机制，使得来自原始非关系数据库表 SALES_SOURCE_DATA 中的每一条记录都可转换为数据仓库 SALES_INFO 表中的 5 条记录。这种操作通常称为转换 (Pivoting)。

此问题的解决方案显示在下一页。

转换 INSERT (Pivoting INSERT)

```
INSERT ALL
  INTO sales_info VALUES (employee_id,week_id,sales_MON)
  INTO sales_info VALUES (employee_id,week_id,sales_TUE)
  INTO sales_info VALUES (employee_id,week_id,sales_WED)
  INTO sales_info VALUES (employee_id,week_id,sales_THUR)
  INTO sales_info VALUES (employee_id,week_id, sales_FRI)
  SELECT EMPLOYEE_ID, week_id, sales_MON, sales_TUE,
         sales_WED, sales_THUR, sales_FRI
  FROM sales_source_data;
```

```
5 rows inserted
```


转换 INSERT (Pivoting INSERT)（续）

在幻灯片示例中，从非关系数据库表 SALES_SOURCE_DATA 中得到了一些销售数据，这些数据提供了某位销售代表在特定周 ID 对应一周中的每个工作日的详细销售信息。

```
DESC SALES_SOURCE_DATA
```

Name	Null	Type
EMPLOYEE_ID		NUMBER(6)
WEEK_ID		NUMBER(2)
SALES_MON		NUMBER(8,2)
SALES_TUE		NUMBER(8,2)
SALES_WED		NUMBER(8,2)
SALES_THUR		NUMBER(8,2)
SALES_FRI		NUMBER(8,2)

转换 INSERT (Pivoting INSERT)（续）

```
SELECT * FROM SALES_SOURCE_DATA;
```

	EMPLOYEE_ID	WEEK_ID	SALES_MON	SALES_TUE	SALES_WED	SALES_THUR	SALES_FRI
1	178	6	1750	2200	1500	1500	3000

```
DESC SALES_INFO
```

Name	Null	Type
EMPLOYEE_ID		NUMBER(6)
WEEK		NUMBER(2)
SALES		NUMBER(8,2)

```
SELECT * FROM sales_info;
```

	EMPLOYEE_ID	WEEK	SALES
1	178	6	1750
2	178	6	2200
3	178	6	1500
4	178	6	1500
5	178	6	3000

在上一个示例中您会注意到，通过使用转换 INSERT (pivoting INSERT)，SALES_SOURCE_DATA 表中的一行已转换为关系表 SALES_INFO 中的 5 条记录。

课程安排

- 使用子查询处理数据
- 在 INSERT 和 UPDATE 语句中指定显式默认值
- 使用以下类型的多表 INSERT：
  - 无条件 INSERT
  - 转换 INSERT (Pivoting INSERT)
  - 条件 INSERT ALL
  - 条件 INSERT FIRST
- 合并表行
- 跟踪一段时间内的数据更改

MERGE 语句

- 提供有条件地在数据库表中更新、删除或插入数据的功能
- 如果行存在，则执行 UPDATE，如果行为新行，则执行 INSERT：
  - 避免单独执行更新
  - 改进了性能且易于使用
  - 在数据仓库应用程序中很有用

MERGE 语句

Oracle Server 支持使用 MERGE 语句执行 INSERT、UPDATE 和 DELETE 操作。使用此语句时，可以有条件地在表中更新、插入或删除行，因此可避免使用多条 DML 语句。在目标表中究竟是执行更新、插入还是删除，取决于 ON 子句中的条件。

您必须在目标表上拥有 INSERT 和 UPDATE 对象权限，在来源表上拥有 SELECT 对象权限。要指定 merge_update_clause 中的 DELETE 子句，还必须在目标表上拥有 DELETE 对象权限。

MERGE 语句具有确定性。您不能在同一 MERGE 语句中多次更新目标表中的同一行。

还有另一种方法可选择，这就是使用 PL/SQL 循环和多条 DML 语句。但是，MERGE 语句更易于使用，也更简单，用一条 SQL 语句就可以表示。

MERGE 语句适用于很多数据仓库应用程序。例如，在数据仓库应用程序中，可能需要处理来自多个源的数据，其中一些数据可能是重复的。使用 MERGE 语句时，就可以有条件地添加或修改行。

MERGE 语句的语法

使用 MERGE 语句时，可以有条件地在表中插入、更新或删除行。

```
MERGE INTO table_name table_alias
  USING (table|view|sub_query) alias
  ON (join condition)
  WHEN MATCHED THEN
    UPDATE SET
    col1 = col1_val,
    col2 = col2_val
 WHEN NOT MATCHED THEN
    INSERT (column_list)
    VALUES (column_values);
```

ORACLE

合并行

通过使用 MERGE 语句可以有条件地更新现有行和插入新行。使用 MERGE 语句时，可以在表中更新行的同时删除过时行。为此，需要在 MERGE 语句的语法中包括一个自带 WHERE 子句的 DELETE 子句。

在该语法中：

INTO 子句	指定要执行更新或插入的目标表
USING 子句	标识要更新或插入的数据来源，可以是一个表、一个视图或一个子查询
ON 子句	MERGE 操作执行更新或插入时所依据的条件
WHEN MATCHED | WHEN NOT MATCHED	指示服务器如何对联接条件的结果做出反应

注：有关详细信息，请参阅《Oracle Database 11*g* SQL Reference》。

合并行：示例

在 COPY_EMPL3 表中插入或更新行使之与 EMPLOYEES 表相匹配。

```
MERGE INTO copy_emp3 c
USING (SELECT * FROM EMPLOYEES ) e
ON (c.employee_id = e.employee_id)
WHEN MATCHED THEN
UPDATE SET
c.first_name = e.first_name,
c.last_name = e.last_name,
...
DELETE WHERE (E.COMMISSION_PCT IS NOT NULL)
WHEN NOT MATCHED THEN
INSERT VALUES(e.employee_id, e.first_name, e.last_name,
e.email, e.phone_number, e.hire_date, e.job_id,
e.salary, e.commission_pct, e.manager_id,
e.department_id);
```

ORACLE

合并行：示例

```
MERGE INTO copy_emp3 c
USING (SELECT * FROM EMPLOYEES ) e
ON (c.employee_id = e.employee_id)
WHEN MATCHED THEN
UPDATE SET
c.first_name = e.first_name,
c.last_name = e.last_name,
c.email = e.email,
c.phone_number = e.phone_number,
c.hire_date = e.hire_date,
c.job_id = e.job_id,
c.salary = e.salary*2,
c.commission_pct = e.commission_pct,
c.manager_id = e.manager_id,
c.department_id = e.department_id
DELETE WHERE (E.COMMISSION_PCT IS NOT NULL)
WHEN NOT MATCHED THEN
INSERT VALUES(e.employee_id, e.first_name, e.last_name,
e.email, e.phone_number, e.hire_date, e.job_id,
e.salary, e.commission_pct, e.manager_id,
e.department_id);
```

合并行：示例（续）

COPY_EMP3 表是使用以下代码创建的：

```
CREATE TABLE COPY_EMP3 AS SELECT * FROM EMPLOYEES
WHERE SALARY<10000;
```

然后，在 COPY_EMP3 表中进行查询。

```
SELECT employee_id, salary, commission_pct FROM COPY_EMP3;
```

	EMPLOYEE_ID	SALARY	COMMISSION_PCT
1	198	5200	(null)
2	199	5200	(null)
3	200	8800	(null)
4	202	12000	(null)
5	203	13000	(null)

...

64	197	6000	(null)
65	162	10500	0.25
66	146	13500	0.3
67	150	10000	0.3

...

此时看到，有一些雇员的薪金少于 10000，有两位雇员有佣金百分比。

幻灯片示例中使 COPY_EMPL3 表中的 EMPLOYEE_ID 与 EMPLOYEES 表中的 EMPLOYEE_ID 相匹配。如果找到了匹配行，则在 COPY_EMPL3 表中更新该行，以匹配 EMPLOYEES 表中的对应行，从而可将雇员薪金加倍。此时会删除 COMMISSION_PCT 列中有值的两个雇员的记录。如果没有发现匹配行，则会在 COPY_EMPL3 表中插入若干行。

合并行：示例

```
TRUNCATE TABLE copy_emp3;
SELECT * FROM copy_emp3;
0 rows selected
```

```
MERGE INTO copy_emp3 c
USING (SELECT * FROM EMPLOYEES ) e
ON (c.employee_id = e.employee_id)
WHEN MATCHED THEN
UPDATE SET
c.first_name = e.first_name,
c.last_name = e.last_name,
...
DELETE WHERE (E.COMMISSION_PCT IS NOT NULL)
WHEN NOT MATCHED THEN
INSERT VALUES(e.employee_id, e.first_name, ...
```

```
SELECT * FROM copy_emp3;
107 rows selected.
```

ORACLE

合并行：示例（续）

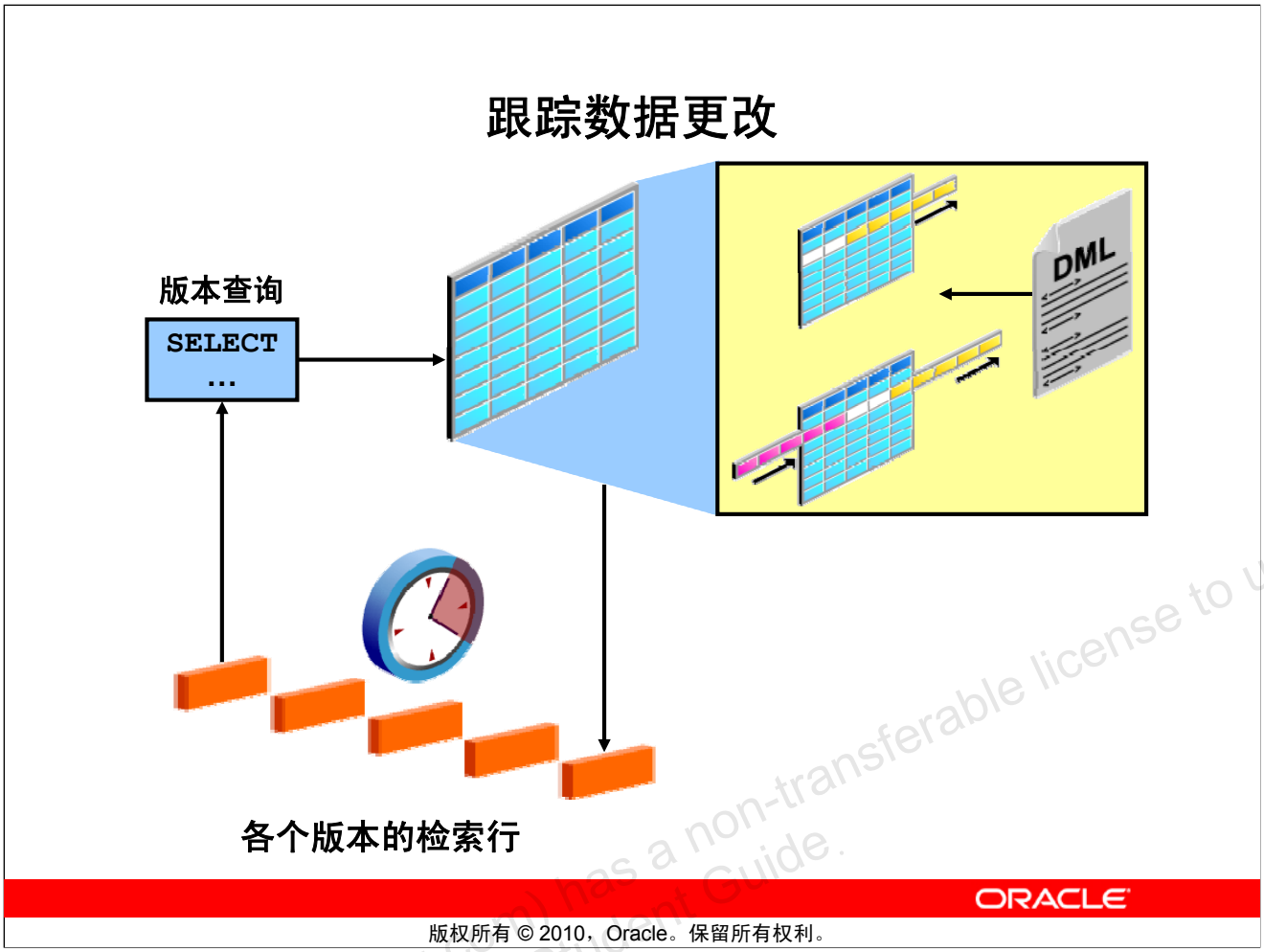
幻灯片中的示例显示 COPY_EMP3 表是空的。对条件 c.employee_id = e.employee_id 进行计算。该条件返回的计算结果为假，即没有匹配行。这个计算结果落入 WHEN NOT MATCHED 子句中，因此 MERGE 命令会在 COPY_EMPL3 表中插入 EMPLOYEES 表中的若干行。这意味着，现在 COPY_EMP3 表中的数据与 EMPLOYEES 表中的数据完全相同。

```
SELECT employee_id, salary, commission_pct from copy_emp3;
```

	EMPLOYEE_ID	SALARY	COMMISSION_PCT
1	144	2500	(null)
2	143	2600	(null)
3	202	6000	(null)
4	141	3500	(null)
5	174	11000	0.3
...			
15	149	10500	0.2
16	206	8300	(null)
17	176	8600	0.2
18	124	5800	(null)
19	205	12000	(null)
20	178	7000	0.15

课程安排

- 使用子查询处理数据
- 在 INSERT 和 UPDATE 语句中指定显式默认值
- 使用以下类型的多表 INSERT：
  - 无条件 INSERT
  - 转换 INSERT (Pivoting INSERT)
  - 条件 INSERT ALL
  - 条件 INSERT FIRST
- 合并表行
- 跟踪一段时间内的数据更改



跟踪数据更改

您可能发现不知何故对表数据进行了不适当地更改。要调查这个问题，可以使用多个闪回查询来查看特定时间点的行数据。但更有效的方法是，可以使用“闪回版本查询”功能查看一段时间内对行进行的所有更改。使用此功能时，可以通过在 SELECT 语句中附加 VERSIONS 子句来指定系统更改号 (SCN) 或时间戳的范围，这个范围是要查看的行值更改对应的范围。此查询还会返回相关的元数据，如负责更改的事务处理。

此外，确定了错误的事务处理之后，可以使用“闪回事务处理查询”功能确定由该事务处理执行的其它更改。然后，可以选择使用“闪回表”功能将表还原到更改前的状态。

使用 VERSIONSONS 子句对表进行查询，可以生成所有版本的所有行，其中包括现有行或在发出此查询后到当前时间前的 undo_retention 秒之间曾经存在的那些行。undo_retention 是一个初始化参数，这是一个经过自动优化的参数。包含 VERSIONS 子句的查询称为版本查询。版本查询的结果表现为好像对行版本应用了 WHERE 子句。版本查询仅返回跨多个事务处理的那些行的版本。

系统更改号 (SCN)：Oracle Server 分配 SCN 的目的是为了标识每一个提交事务处理的重做记录。

闪回版本查询示例

```
SELECT salary FROM employees3
WHERE  employee_id = 107;
```

1

```
UPDATE employees3 SET salary = salary * 1.30
WHERE  employee_id = 107;

COMMIT;
```

2

```
SELECT salary FROM employees3
  VERSIONS BETWEEN SCN MINVALUE AND MAXVALUE
WHERE  employee_id = 107;
```

3

1

	SALARY
1	4200

3

	SALARY
1	5460
2	4200

ORACLE

闪回版本查询示例

在幻灯片示例中，(1) 检索雇员 107 的薪金。(2) 雇员 107 的薪金增长 30% 并提交此更改。(3) 显示不同版本的薪金。

VERSIONS 子句不会更改查询计划。例如，如果在某个表上运行的一个查询使用索引访问方法，则使用 VERSIONS 子句在同一表上执行的同一查询会继续使用索引访问方法。由版本查询返回的行的版本是跨多个事务处理的行的版本。VERSIONS 子句对查询的事务处理行为没有影响。这意味着使用 VERSIONS 子句在表上进行的查询仍继承正在进行事务处理的查询环境。

默认的 VERSIONS 子句可指定为 VERSIONS BETWEEN {SCN|TIMESTAMP} MINVALUE AND MAXVALUE。

VERSIONS 子句是 SQL 扩展，仅适于在查询中使用。可以在子查询中使用 VERSIONS 子句执行 DML 和 DDL 操作。行版本查询会检索选定行的所有已提交版本。不会返回当前活动事务处理进行的更改。版本查询会检索行的所有副本。这实际上意味着在返回版本中包括行的删除版本和后续重新插入版本。

闪回版本查询示例（续）

版本查询的行访问可定义为以下两种类别：

- **基于 ROWID 的行访问：**在基于 ROWID 访问时，会返回指定 ROWID 的所有版本，与行内容无关。这实际上意味着会返回块中由 ROWID 指定的槽中的所有版本。
- **其它所有行访问：**进行其它所有行访问时，会返回行的所有版本。

VERSIONS BETWEEN 子句

```
SELECT versions_starttime "START_DATE",
       versions_endtime   "END_DATE",
       salary
FROM   employees
    VERSIONS BETWEEN SCN MINVALUE
    AND MAXVALUE
WHERE  last_name = 'Lorentz';
```

	START_DATE	END_DATE	SALARY
1	18-JUN-09 05.07.10.000000000 PM	(null)	5460
2	(null)	18-JUN-09 05.07.10.000000000 PM	4200

VERSIONS BETWEEN 子句

可以使用 VERSIONS BETWEEN 子句检索行的所有版本，这些行是现有行或者是从发出查询到之后某个时间点之间存在的那些行。

如果还原保留时间小于 BETWEEN 子句中的下限时间或下限 SCN，则在查询中最多只检索从发出查询到还原保留时间之间存在的那些版本。BETWEEN 子句中的时间间隔可指定为 SCN 间隔或系统时钟间隔。此时间间隔在到达下限和上限时结束。

在该示例中，检索了 Lorentz 的薪金变动。第一个版本的 END_DATE 的值为 NULL，这表示该版本在查询时是现有版本。最后一个版本的 START_DATE 的值为 NULL，这表示此版本的创建时间在还原保留时间之前。

小测验

使用 INSERT 或 UPDATE 命令时，DEFAULT 关键字减少了工作量，你不必在程序中对默认值进行硬编码，或者通过查询字典来查找默认值。

1. 正确
2. 错误

答案：1

小结

在本课中，您应该已经学会：

- 使用 DML 语句和控制事务处理
- 描述多表 INSERT 功能
- 使用以下类型的多表 INSERT：
  - 无条件 INSERT
  - 转换 INSERT (Pivoting INSERT)
  - 条件 INSERT ALL
  - 条件 INSERT FIRST
- 合并表行
- 使用子查询处理数据
- 跟踪一段时间内的数据更改

ORACLE

小结

在本课中，您应该已经学会如何使用子查询处理 Oracle DB 中的数据。还应该掌握了多表 INSERT 语句、MERGE 语句以及如何跟踪数据库更改。

练习 4：概览

本练习包含以下主题：

- 执行多表 INSERT
- 执行 MERGE 操作
- 跟踪行版本

5

管理不同时区中的数据

课程目标

学完本课后，应能完成下列工作：

- 使用与 DATE 类似的数据类型来存储零点几秒和跟踪时区
- 使用数据类型来存储两个日期时间值之间的差异
- 使用以下日期时间函数：
  - CURRENT_DATE
  - CURRENT_TIMESTAMP
  - LOCALTIMESTAMP
  - DBTIMEZONE
  - SESSIONTIMEZONE
  - EXTRACT
  - TZ_OFFSET
  - FROM_TZ
  - TO_TIMESTAMP
  - TO_YMINTERVAL
  - TO_DSINTERVAL

ORACLE

课程目标

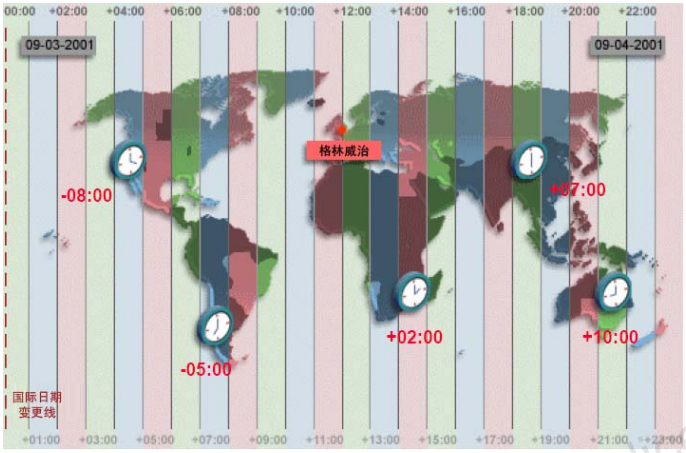
在本课中，您将学习如何使用与 DATE 类似的数据类型来存储零点几秒和跟踪时区。本课将介绍 Oracle DB 中提供的某些日期时间函数。

课程安排

- CURRENT_DATE、CURRENT_TIMESTAMP 和 LOCALTIMESTAMP
- INTERVAL 数据类型
- 使用以下函数：
  - EXTRACT
  - TZ_OFFSET
  - FROM_TZ
  - TO_TIMESTAMP
  - TO_YMINTERVAL
  - TO_DSINTERVAL

ORACLE

时区



上图中提供了格林威治时间为 12:00 时各个时区的时间。

ORACLE

时区

一天的小时数是通过地球自转测量出的。一天中任何特定时刻的时间取决于您所在的位置。英国格林尼治的时间为中午时，国际日期变更线沿线地区的时间为午夜。地球被划分为 24 个时区，一个时区代表一天中的一个小时。英国格林尼治本初子午线沿线的时间被称为格林尼治标准时间 (GMT)。GMT 现在还称为国际协调时间 (UTC)。UTC 是世界上所有其它时区参照的时间标准。它全年保持不变，不受夏时制或夏令时的影响。子午线是一条假想的从北极到南极的线。其经度为零，它是度量所有其它经线的基准。所有时间都是相对于 UTC 度量得到的，所有地点都有一个纬度（距赤道以北或以南的距离）和一个经度（距格林尼治本初子午线以东或以西的距离）。

TIME_ZONE 会话参数

TIME_ZONE 可设置为：

- 绝对偏移量
- 数据库时区
- 操作系统本地时区
- 命名区域

```
ALTER SESSION SET TIME_ZONE = '-05:00';
ALTER SESSION SET TIME_ZONE = dbtimezone;
ALTER SESSION SET TIME_ZONE = local;
ALTER SESSION SET TIME_ZONE = 'America/New_York';
```

ORACLE

TIME_ZONE 会话参数

Oracle DB 支持以日期和时间数据的形式存储时区，还支持以零点几秒的形式存储时区。ALTER SESSION 命令可用来更改用户会话中的时区值。时区值可设置为绝对偏移量、命名时区、数据库时区或本地时区。

CURRENT_DATE、CURRENT_TIMESTAMP 和 LOCALTIMESTAMP

- CURRENT_DATE：
  - 返回用户会话的当前日期
  - 数据类型为 DATE
- CURRENT_TIMESTAMP：
  - 返回用户会话的当前日期和时间
  - 数据类型为 TIMESTAMP WITH TIME ZONE
- LOCALTIMESTAMP：
  - 返回用户会话的当前日期和时间
  - 数据类型为 TIMESTAMP

ORACLE

CURRENT_DATE、CURRENT_TIMESTAMP 和 LOCALTIMESTAMP

CURRENT_DATE 和 CURRENT_TIMESTAMP 函数分别返回当前日期和当前时间戳。CURRENT_DATE 的数据类型为 DATE。CURRENT_TIMESTAMP 的数据类型为 TIMESTAMP WITH TIME ZONE。返回的各个值显示执行函数的 SQL 会话的时区偏移量。时区偏移量是本地时间与 UTC 之间的差值（以小时和分钟表示）。TIMESTAMP WITH TIME ZONE 数据类型的格式为：

```
TIMESTAMP [ (fractional_seconds_precision) ] WITH TIME ZONE
```

其中 fractional_seconds_precision 是一个可选参数，可用来指定 SECOND 日期时间字段中小数部分的位数，这个参数可以是 0 到 9 之间的一个数字。默认值为 6。

LOCALTIMESTAMP 函数返回会话时区的当前日期和时间。LOCALTIMESTAMP 与 CURRENT_TIMESTAMP 之间的区别在于，LOCALTIMESTAMP 返回 TIMESTAMP 值，而 CURRENT_TIMESTAMP 返回 TIMESTAMP WITH TIME ZONE 值。

这些函数与国家语言支持 (NLS) 相关，也就是说，结果将采用当前 NLS 日历和日期时间格式。

注：SYSDATE 函数会返回 DATE 数据类型的当前日期和时间。在《Oracle Database 11*g*：SQL 基础 I》课程中，您学习了如何使用 SYSDATE 函数。

对会话时区的日期和时间进行比较

将 TIME_ZONE 参数设置为 -5:00，然后针对每一日期和时间执行 SELECT 语句来比较差异。

```
ALTER SESSION
SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH24:MI:SS';
ALTER SESSION SET TIME_ZONE = '-5:00';

SELECT SESSIONTIMEZONE, CURRENT_DATE FROM DUAL;          1


SELECT SESSIONTIMEZONE, CURRENT_TIMESTAMP FROM DUAL;     2


SELECT SESSIONTIMEZONE, LOCALTIMESTAMP FROM DUAL;        3
```

ORACLE

对会话时区的日期和时间进行比较

使用 ALTER SESSION 命令将会话的日期格式设置为 'DD-MON-YYYY HH24:MI:SS'，即“日期 (1-31)-缩写月份名-4 位数字年份小时 (0-23)：分钟 (0-59)：秒 (0-59)”。

幻灯片示例中显示通过更改会话将 TIME_ZONE 参数设置为 -5:00，然后针对 CURRENT_DATE、CURRENT_TIMESTAMP 和 LOCALTIMESTAMP 执行 SELECT 语句来观察格式的差异。

注：TIME_ZONE 参数指定当前 SQL 会话的默认本地时区偏移量。TIME_ZONE 只是会话参数，不是初始化参数。TIME_ZONE 参数可设置为如下：

```
TIME_ZONE = '[+ | -] hh:mm'
```

格式掩码 ([+ | -] hh:mm) 表示在 UTC 之前或之后的小时数或分钟数。

对会话时区的日期和时间进行比较

查询的结果：

```
ALTER SESSION  succeeded.
```

	SESSIONTIMEZONE	CURRENT_DATE
1	-05:00	23-JUN-2009 01:34:52

1

	SESSIONTIMEZONE	CURRENT_TIMESTAMP
1	-05:00	23-JUN-09 01.35.26.239882000 AM -05:00

2

	SESSIONTIMEZONE	LOCALTIMESTAMP
1	-05:00	23-JUN-09 01.36.21.811798000 AM

3

ORACLE

对会话时区的日期和时间进行比较（续）

此示例下，CURRENT_DATE 函数返回会话时区中的当前日期，CURRENT_TIMESTAMP 函数返回会话时区中数据类型值为 TIMESTAMP WITH TIME ZONE 的当前日期和时间，而 LOCALTIMESTAMP 函数返回会话时区中的当前日期和时间。

DBTIMEZONE 和 SESSIONTIMEZONE

- 显示数据库时区值：

```
SELECT DBTIMEZONE FROM DUAL;
```

	DBTIMEZONE
1	+00:00

- 显示会话时区值：

```
SELECT SESSIONTIMEZONE FROM DUAL;
```

	SESSIONTIMEZONE
1	-05:00

ORACLE

DBTIMEZONE 和 SESSIONTIMEZONE

DBA 通过在 CREATE DATABASE 语句中使用 SET TIME_ZONE 子句来设置数据库的默认时区。如果省略该子句，则默认的数据库时区是操作系统时区。使用 ALTER SESSION 语句不能更改会话的数据库时区。

DBTIMEZONE 函数会返回数据库时区值。返回类型可以是时区偏移量（一种有如下格式的字符类型：'[+|-]TZH:TZM'），也可以是时区区域名，这取决于用户在最近执行的 CREATE DATABASE 或 ALTER DATABASE 语句中如何指定数据库时区值。幻灯片示例中显示数据库时区设置为“-05:00”，TIME_ZONE 参数的格式为：

```
TIME_ZONE = '[+ | -] hh:mm'
```

SESSIONTIMEZONE 函数会返回当前会话的时区值。返回类型可以是时区偏移量（格式为 '[+|-]TZH:TZM' 的字符类型），也可以是时区区域名，这取决于用户在最近执行的 ALTER SESSION 语句中如何指定会话时区值。幻灯片示例中显示会话时区的偏移量为 UTC -8 小时。请注意，数据库时区不同于当前会话的时区。

TIMESTAMP 数据类型

数据类型	字段
TIMESTAMP	精确到零点几秒的年、月、日、小时、分钟、秒
TIMESTAMP WITH TIME ZONE	与 TIMESTAMP 数据类型相同；还包括以下类型： TIMEZONE_HOUR 和 TIMEZONE_MINUTE 或 TIMEZONE_REGION
TIMESTAMP WITH LOCAL TIME ZONE	与 TIMESTAMP 数据类型相同；其值中还包含时区偏移量

TIMESTAMP 数据类型

TIMESTAMP 数据类型是 DATE 数据类型的扩展。

```
TIMESTAMP (fractional_seconds_ precision)
```

此数据类型包含日期的年、月和日值，还包含时间的小时、分钟和秒值，其中零点几秒的有效精度是指 SECOND 日期时间字段中小数部分的位数。可接受的有效 fractional_seconds_precision 值是 0 到 9 之间的一个数字。默认值为 6。

```
TIMESTAMP (fractional_seconds_precision) WITH TIME ZONE
```

此数据类型包含 TIMESTAMP 的所有值，还包含时区偏移量值。

```
TIMESTAMP (fractional_seconds_precision) WITH LOCAL TIME ZONE
```

此数据类型包含 TIMESTAMP 的所有值，但下列情况除外：

- 数据存储在数据库中时已按照数据库时区的标准进行转换。
- 检索数据时，用户会看到会话时区中的数据。

TIMESTAMP 字段

日期时间字段	有效值
YEAR	-4712 到 9999（不包括 0 年）
MONTH	01 到 12
DAY	01 到 31
HOUR	00 到 23
MINUTE	00 到 59
SECOND	00 到 59.9(N)，其中 9(N) 为精度
TIMEZONE_HOUR	-12 到 14
TIMEZONE_MINUTE	00 到 59

TIMESTAMP 字段

每个日期时间数据类型都由上述字段中的若干个字段组成。只能对包含相同字段的日期时间进行相互比较和相互指定。

DATE 和 TIMESTAMP 之间的区别

A

```
-- when hire_date is
of type DATE

SELECT hire_date
FROM employees;
```

	HIRE_DATE
1	21-JUN-99
2	13-JAN-00
3	17-SEP-87
4	17-FEB-96
5	17-AUG-97
6	07-JUN-94
7	07-JUN-94
8	07-JUN-94

B

```
ALTER TABLE employees
MODIFY hire_date TIMESTAMP;

SELECT hire_date
FROM employees;
```

	HIRE_DATE
1	21-JUN-99 12.00.00.000000000 AM
2	13-JAN-00 12.00.00.000000000 AM
3	17-SEP-87 12.00.00.000000000 AM
4	17-FEB-96 12.00.00.000000000 AM
5	17-AUG-97 12.00.00.000000000 AM
6	07-JUN-94 12.00.00.000000000 AM
7	07-JUN-94 12.00.00.000000000 AM
8	07-JUN-94 12.00.00.000000000 AM

...

ORACLE

TIMESTAMP 数据类型：示例

在幻灯片中，示例 A 显示 EMPLOYEES 表 hire_date 列中的数据，该列的数据类型为 DATE。在示例 B 中，对表进行了更改，hire_date 列的数据类型已更改为 TIMESTAMP。在输出中可以看到两者的差异。如果该列包含数据，则可以将 DATE 转换为 TIMESTAMP，但不能将 DATE 或 TIMESTAMP 转换为 TIMESTAMP WITH TIME ZONE，除非该列为空。

可以为时间戳指定零点几秒精度。如果没有指定任何值，如本示例所示，则默认精度为 6。

例如，以下语句将零点几秒精度设置为 7：

```
ALTER TABLE employees
MODIFY hire_date TIMESTAMP(7);
```

注：在默认情况下，Oracle 日期数据类型的格式如本示例中所示。但是，日期数据类型还包含其他信息，如小时、分钟、秒、AM 和 PM。要获取这种格式的日期，可对日期值应用格式掩码或函数。

TIMESTAMP 数据类型的比较

```
CREATE TABLE web_orders
(order_date TIMESTAMP WITH TIME ZONE,
 delivery_time TIMESTAMP WITH LOCAL TIME ZONE);
```

```
INSERT INTO web_orders values
(current_date, current_timestamp + 2);
```

```
SELECT * FROM web_orders;
```

	ORDER_DATE	DELIVERY_TIME
1	23-JUN-09 01.56.39.000000000 AM -05:00	25-JUN-09 01.56.39.000000000 AM

ORACLE

TIMESTAMP 数据类型的比较

在幻灯片示例中，创建的新表 web_orders 中包含数据类型为 TIMESTAMP WITH TIME ZONE 和 TIMESTAMP WITH LOCAL TIME ZONE 的两列。每当提交 web_order 时就会填充此表。此时会根据 CURRENT_DATE 值插入用户提交订单时的时间戳和时区。每次提交订单时，都通过插入 CURRENT_TIMESTAMP + 2 这样一个值来填充本地时间戳和时区。当一家通过 Web 运营的公司要确定运送时间时，就可以根据提交订单客户所在的时区来估计交货时间。

课程安排

- CURRENT_DATE、CURRENT_TIMESTAMP 和 LOCALTIMESTAMP
- INTERVAL 数据类型
- 使用以下函数：
  - EXTRACT
  - TZ_OFFSET
  - FROM_TZ
  - TO_TIMESTAMP
  - TO_YMINTERVAL
  - TO_DSINTERVAL

INTERVAL 数据类型

- INTERVAL 数据类型用来存储两个日期时间值之间的差值。
- 间隔分为两种类型：
  - 年-月
  - 日-时间
- 间隔的精度：
  - 是构成间隔的字段的实际子集
  - 用间隔限定符指定

数据类型	字段
INTERVAL YEAR TO MONTH	年、月
INTERVAL DAY TO SECOND	天、小时、分钟、精确到零点几秒的秒

ORACLE

INTERVAL 数据类型

INTERVAL 数据类型用来存储两个日期时间值之间的差值。间隔分为两种类型：年-月间隔和日-时间间隔。年-月间隔由 YEAR 和 MONTH 两个字段的一个连续子集组成，而日-时间间隔由包含 DAY、HOUR、MINUTE 和 SECOND 这些字段的一个连续子集组成。构成间隔的字段的实际子集被称为间隔的精度，可用间隔限定符来指定。因为一年中的天数与日历相关，所以年-月间隔与 NLS 相关，而日-时间间隔与 NLS 无关。

间隔限定符还可以指定前导字段的精度，这个精度是前导字段或单个字段中的位数，如果尾部字段为 SECOND，则间隔限定符还可以指定零点几秒精度，即 SECOND 值小数部分的位数。如果没有指定，则前导字段精度的默认值为 2 位数，而零点几秒精度的默认值为 6 位数。

INTERVAL 数据类型（续）

INTERVAL YEAR (year_precision) TO MONTH

此数据类型用来存储以年和月表示的时间段，其中 year_precision 是 YEAR 日期时间字段中的位数。可接受值为 0 到 9 之间的一个数字。默认值为 6。

INTERVAL DAY (day_precision) TO SECOND (fractional_seconds_precision)

此数据类型用来存储以天数、小时数、分钟数和秒数表示的时间段，其中 day_precision 是 DAY 日期时间字段中的最大位数（可接受值为 0 到 9 之间的一个数字，默认值为 2），而 fractional_seconds_precision 是 SECOND 字段小数部分的位数。可接受值为 0 到 9 之间的一个数字。默认值为 6。

INTERVAL 字段

INTERVAL 字段	有效的间隔值
YEAR	任意正整数或负整数
MONTH	00 到 11
DAY	任意正整数或负整数
HOUR	00 到 23
MINUTE	00 到 59
SECOND	00 到 59.9(N)，其中 9(N) 为精度

INTERVAL 字段

INTERVAL YEAR TO MONTH 可以包含 YEAR 和 MONTH 字段。

INTERVAL DAY TO SECOND 可以包含 DAY、HOUR、MINUTE 和 SECOND 字段。

用一个间隔限定符可定义构成任一类间隔中某一项的字段的实际子集，此子集称为该项的精度。

年-月间隔只能与年-月间隔相互比较和相互指定，日-时间间隔只能与日-时间间隔相互比较和相互指定。

INTERVAL YEAR TO MONTH：示例

```
CREATE TABLE warranty
(prod_id number,  warranty_time INTERVAL YEAR(3) TO
MONTH);
INSERT INTO warranty VALUES (123, INTERVAL '8' MONTH);
INSERT INTO warranty VALUES (155, INTERVAL '200'
YEAR(3));
INSERT INTO warranty VALUES (678, '200-11');
SELECT * FROM warranty;
```

	PROD_ID	WARRANTY_TIME
1	123	0-8
2	155	200-0
3	678	200-11

ORACLE

INTERVAL YEAR TO MONTH 数据类型

INTERVAL YEAR TO MONTH 使用 YEAR 和 MONTH 日期时间字段存储时间段。INTERVAL YEAR TO MONTH 可指定为如下格式：

```
INTERVAL YEAR [(year_precision)] TO MONTH
```

其中 year_precision 为 YEAR 日期时间字段中的位数。year_precision 的默认值为 2。

限制：前导字段必须比尾部字段更大。例如，INTERVAL '0-1' MONTH TO YEAR 是无效的。

示例

- INTERVAL '123-2' YEAR(3) TO MONTH
  表明间隔为 123 年 2 个月
- INTERVAL '123' YEAR(3)
  表明间隔为 123 年 0 个月

INTERVAL YEAR TO MONTH 数据类型（续）

- `INTERVAL '300' MONTH(3)`

  表明间隔为 300 个月

- `INTERVAL '123' YEAR`

  因为默认精度为 2，而 123 有三位数，所以会返回一条错误

Oracle DB 支持两种间隔数据类型：`INTERVAL YEAR TO MONTH` 和 `INTERVAL DAY TO SECOND`；列类型、PL/SQL 参数、变量和返回类型必须采用其中一种类型。但是，若使用了文字间隔，系统还会识别美国国家标准协会 (ANSI) 的其它间隔类型，如 `INTERVAL '2' YEAR` 或 `INTERVAL '10' HOUR`。在这些情况下，每种间隔都会被转换为一种支持的类型。

在幻灯片示例中，创建了 `WARRANTY` 表，该表包含的 `warranty_time` 列采用的数据类型为 `INTERVAL YEAR(3) TO MONTH`。在该表中插入了不同的值来指出各种产品的年数和月数。在该表中检索这些行时，您会看到年值和月值用符号 (-) 已分隔开。

INTERVAL DAY TO SECOND 数据类型：示例

```
CREATE TABLE lab
( exp_id number, test_time INTERVAL DAY(2) TO SECOND);

INSERT INTO lab VALUES (100012, '90 00:00:00');
INSERT INTO lab VALUES (56098,
  INTERVAL '6 03:30:16' DAY  TO SECOND);
```

```
SELECT * FROM lab;
```

	EXP_ID	TEST_TIME
1	100012	90 0:0:0.0
2	56098	6 3:30:16.0

INTERVAL DAY TO SECOND 数据类型：示例

在幻灯片示例中，创建的 lab 表中 test_time 列的数据类型为 INTERVAL DAY TO SECOND。然后在该表中插入了值“90 00:00:00”来表示 90 天 0 小时 0 分钟 0 秒，还插入了 INTERVAL '6 03:30:16' DAY TO SECOND 来表示 6 天 3 小时 30 分钟 16 秒。SELECT 语句显示这些数据在数据库中是如何显示的。

课程安排

- CURRENT_DATE、CURRENT_TIMESTAMP 和 LOCALTIMESTAMP
- INTERVAL 数据类型
- 使用以下函数：
  - EXTRACT
  - TZ_OFFSET
  - FROM_TZ
  - TO_TIMESTAMP
  - TO_YMINTERVAL
  - TO_DSINTERVAL

ORACLE

EXTRACT

- 显示 SYSDATE 中的 YEAR 组成部分。

```
SELECT EXTRACT (YEAR FROM SYSDATE) FROM DUAL;
```

	EXTRACT(YEARFROMSYSDATE)
1	2009

- 显示 MANAGER_ID 为 100 的那些雇员的 HIRE_DATE 中的 MONTH 组成部分。

```
SELECT last_name, hire_date,
       EXTRACT (MONTH FROM HIRE_DATE)
FROM employees
WHERE manager_id = 100;
```

	LAST_NAME	HIRE_DATE	EXTRACT(MONTHFROMHIRE_DATE)
1	Hartstein	17-FEB-1996 00:00:00	2
2	Kochhar	21-SEP-1989 00:00:00	9
3	De Haan	13-JAN-1993 00:00:00	1
4	Raphaely	07-DEC-1994 00:00:00	12
5	Weiss	18-JUL-1996 00:00:00	7

ORACLE

EXTRACT

EXTRACT 表达式用来从日期时间或间隔值表达式中提取并返回指定日期时间字段的值。使用 EXTRACT 函数可提取以下语法中涉及的任何组成部分。EXTRACT 函数的语法为：

```
SELECT  EXTRACT ([YEAR] [MONTH][DAY] [HOUR] [MINUTE][SECOND]
                 [TIMEZONE_HOUR] [TIMEZONE_MINUTE]
                 [TIMEZONE_REGION] [TIMEZONE_ABBR]
FROM [datetime_value_expression] [interval_value_expression]);
```

提取 TIMEZONE_REGION 或 TIMEZONE_ABBR（缩写）时，返回值是一个包含相应时区名称或缩写的字符串。提取任何其它值时，返回值是一个用公历表示的日期。从包含时区值的日期时间中进行提取时，返回值用 UTC 表示。

在幻灯片第一个示例中，EXTRACT 函数用来从 SYSDATE 中提取 YEAR。在幻灯片第二个示例中，EXTRACT 函数用来从 EMPLOYEES 表的 HIRE_DATE 列中提取其经理 EMPLOYEE_ID 为 100 的那些雇员的 MONTH。

TZ_OFFSET

显示 'US/Eastern'、'Canada/Yukon' 和 'Europe/London' 时区的时区偏移量：

```
SELECT TZ_OFFSET('US/Eastern'),
       TZ_OFFSET('Canada/Yukon'),
       TZ_OFFSET('Europe/London')
 FROM DUAL;
```

	TZ_OFFSET('US/EASTERN')	TZ_OFFSET('CANADA/YUKON')	TZ_OFFSET('EUROPE/LONDON')
1	-04:00	-07:00	+01:00

TZ_OFFSET

TZ_OFFSET 函数用来返回对应于输入值的时区偏移量。返回值取决于执行语句的日期。例如，如果 TZ_OFFSET 函数返回的值是 -08:00，则此值表示执行命令的时区比 UCT 晚 8 个小时。可以输入一个有效的时区名、与 UTC 的时区偏移量（此时只返回它本身）或者关键字 SESSIONTIMEZONE 或 DBTIMEZONE。TZ_OFFSET 函数的语法为：

```
TZ_OFFSET ( ['time_zone_name'] '[+ | -] hh:mm']
          [ SESSIONTIMEZONE] [DBTIMEZONE]
```

Fold Motor Company 的总部位于美国的密歇根州，该地区属于美国/东部时区。公司总裁 Fold 先生要与加拿大的执行副总裁和欧洲的执行副总裁（他们分别位于加拿大/育空时区和欧洲/伦敦时区）一起主持一个电话会议。Fold 先生要了解这两个地区的时间，以确保公司高级经理可以参加会议。他的秘书 Scott 先生协助他发出示例中显示的查询后得到了以下结果：

- 美国/东部时区比 UTC 晚 4 个小时。
- 加拿大/育空时区比 UTC 晚 7 个小时。
- 欧洲/伦敦时区比 UTC 早 1 个小时。

TZ_OFFSET（续）

为了获取有效时区名称值列表，可以在 V$TIMEZONE_NAMES 动态性能视图中进行查询。

```
SELECT * FROM V$TIMEZONE_NAMES;
```

	TZNAME	TZABBREV
1	Africa/Abidjan	LMT
2	Africa/Abidjan	GMT
3	Africa/Accra	LMT
4	Africa/Accra	GMT
5	Africa/Accra	GHST

...

FROM_TZ

将 TIMESTAMP 值 '2000-03-28 08:00:00' 显示为 'Australia/North' 时区区域的 TIMESTAMP WITH TIME ZONE 值。

```
SELECT FROM_TZ(TIMESTAMP
       '2000-07-12 08:00:00', 'Australia/North')
FROM DUAL;
```

FROM_TZ(TIMESTAMP'2000-07-1208:00:00','AUSTRALIA/NORTH')
1 12-JUL-00 08.00.00.000000000 AM AUSTRALIA/NORTH

ORACLE

FROM_TZ

FROM_TZ 函数用来将 TIMESTAMP 值转换为 TIMESTAMP WITH TIME ZONE 值。

FROM_TZ 函数的语法为：

FROM_TZ(TIMESTAMP timestamp_value, time_zone_value)

其中 time_zone_value 是一个格式为 'TZH:TZM' 的字符串，或者是一个字符表达式，可用来返回一个以 TZR（时区区域）表示的字符串且可带有 TZD 格式。TZD 是缩写的时区字符串，它用来提供夏令时信息。TZR 表示日期时间输入字符串中的时区区域。例如，'Australia/North'、代表美国/太平洋标准时间的 'PST'、代表美国/太平洋夏令时的 'PDT' 等。

幻灯片示例中将 TIMESTAMP 值转换为了 TIMESTAMP WITH TIME ZONE。

注：要查看 TZR 和 TZD 格式元素的有效值列表，请在 V$TIMEZONE_NAMES 动态性能视图中进行查询。

TO_TIMESTAMP

将字符串 '2007-03-06 11:00:00' 显示为 TIMESTAMP 值：

```
SELECT TO_TIMESTAMP ('2007-03-06 11:00:00',
                     'YYYY-MM-DD HH:MI:SS')
FROM DUAL;
```

TO_TIMESTAMP('2007-03-0611:00:00','YYYY-MM-DDHH:MI:SS')
06-MAR-07 11.00.00.000000000

ORACLE

TO_TIMESTAMP

TO_TIMESTAMP 函数用来将 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型的字符串转换为 TIMESTAMP 数据类型的值。TO_TIMESTAMP 函数的语法为：

```
TO_TIMESTAMP (char,[fmt],['nlsparam'])
```

fmt 是一个可选参数，用于指定 char 的格式。如果省略了 fmt，则字符串的默认格式必须为 TIMESTAMP 数据类型。nlsparams 也是一个可选参数，用于指定返回的月份名称、日期名称以及缩写采用的语言。此参数的格式为：

```
'NLS_DATE_LANGUAGE = language'
```

如果省略了 nlsparams，此函数会使用会话的默认日期语言。

幻灯片示例中将字符串转换为了 TIMESTAMP 值。

注：使用 TO_TIMESTAMP_TZ 函数可将 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型的字符串转换为 TIMESTAMP WITH TIME ZONE 数据类型的值。有关此函数的详细信息，请参阅《Oracle Database SQL Reference 11*g* Release 1 (11.1)》。

TO_YMINTERVAL

显示在 DEPARTMENT_ID 为 20 的部门中工作的雇员的受雇日期之后一年零两个月那天对应的日期。

```
SELECT hire_date,
       hire_date + TO_YMINTERVAL('01-02') AS
       HIRE_DATE_YMININTERVAL
FROM   employees
WHERE department_id = 20;
```

	HIRE_DATE	HIRE_DATE_YMININTERVAL
1	17-FEB-1996 00:00:00	17-APR-1997 00:00:00
2	17-AUG-1997 00:00:00	17-OCT-1998 00:00:00

TO_YMINTERVAL

TO_YMINTERVAL 函数用来将 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型的字符串转换为 INTERVAL YEAR TO MONTH 数据类型。INTERVAL YEAR TO MONTH 数据类型使用 YEAR 和 MONTH 日期时间字段存储时间段。INTERVAL YEAR TO MONTH 的格式如下：

```
INTERVAL YEAR [(year_precision)] TO MONTH
```

其中 year_precision 为 YEAR 日期时间字段中的位数。year_precision 的默认值为 2。

TO_YMINTERVAL 函数的语法为：

```
TO_YMINTERVAL (char)
```

其中 char 是要转换的字符串。

幻灯片示例中计算了在 EMPLOYEES 表的部门 20 中工作的雇员的受雇日期之后一年零两个月那天对应的日期。

TO_DSINTERVAL

显示所有雇员的受雇日期之后 100 天零 10 小时所对应的日期。

```
SELECT last_name,
 TO_CHAR(hire_date, 'mm-dd-yy:hh:mi:ss') hire_date,
  TO_CHAR(hire_date +
   TO_DSINTERVAL('100 10:00:00'),
     'mm-dd-yy:hh:mi:ss') hiredate2
FROM employees;
```

	LAST_NAME	HIRE_DATE	HIREDATE2
1	OConnell	06-21-99:12:00:00	09-29-99:10:00:00
2	Grant	01-13-00:12:00:00	04-22-00:10:00:00
3	Whalen	09-17-87:12:00:00	12-26-87:10:00:00
4	Hartstein	02-17-96:12:00:00	05-27-96:10:00:00
5	Fay	08-17-97:12:00:00	11-25-97:10:00:00
6	Mavris	06-07-94:12:00:00	09-15-94:10:00:00
7	Baer	06-07-94:12:00:00	09-15-94:10:00:00
8	Higgins	06-07-94:12:00:00	09-15-94:10:00:00

...

ORACLE

TO_DSINTERVAL

TO_DSINTERVAL 用来将 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型的字符串转换为 INTERVAL DAY TO SECOND 数据类型。

在幻灯片示例中，显示受雇日期之后 100 天零 10 个小时所对应的日期。

夏令时

- 四月的第一个星期日
  - 时间从 01:59:59 AM 跳到 03:00:00 AM。
  - 02:00:00 AM 到 02:59:59 AM 之间的值变为无效。
- 十月的最后一个星期日
  - 时间从 02:00:00 AM 跳到 01:00:01 AM。
  - 01:00:01 AM 到 02:00:00 AM 之间的值因出现两次而不确定。

ORACLE

夏令时 (DST)

大多数西方国家都在夏季将时钟拨快一小时。这个期间就称为夏令时期间。在美国、墨西哥和加拿大的大部分地区，夏令时从四月的第一个星期日开始，一直持续到十月的最后一个星期日。欧盟各国也实行夏令时，但它们将其称为夏时制。欧洲夏时制的开始时间比北美地区早一个星期，但结束时间相同。

Oracle DB 会自动确定任何指定时区区域是否在实行夏令时并返回相应的本地时间值。在任何情况下，Oracle DB 都能根据日期时间值来确定指定区域是否在实行夏令时，只要不是处于分界情况下。分界情况出现在夏令时开始或结束这段时间内。例如，在美国/东部区域，夏令时开始时，时间从 01:59:59 AM 跳至 03:00:00 AM。02:00:00 AM 到 02:59:59 AM 之间的一个小时不存在了。夏令时结束时，时间从 02:00:00 AM 又回到 01:00:01 AM，01:00:01 AM 与 02:00:00 AM 之间的一小时出现两次。

夏令时 (DST)（续）

ERROR_ON_OVERLAP_TIME

ERROR_ON_OVERLAP_TIME 是一个会话参数，用于通知系统在遇到处于重叠期间的日期时间且没有指定时区缩写来区分该期间时发出错误。

例如，夏令时在 10 月 31 日 02:00:01 AM 结束。重叠期间为：

- 10/31/2004 01:00:01 AM 至 10/31/2004 02:00:00 AM (EDT)
- 10/31/2004 01:00:01 AM 至 10/31/2004 02:00:00 AM (EST)

如果输入的一个日期时间字符串处于其中任何一个期间，则需要在输入的字符串中指定时区缩写（例如 EDT 或 EST），使系统能够确定该期间。如果没有指定这个时区缩写，系统会执行以下操作：

当 ERROR_ON_OVERLAP_TIME 参数为 FALSE 时，系统假定输入时间为标准时间（例如 EST）。否则，会引发一个错误。

小测验

TIME_ZONE 会话参数可以设置为：

1. 相对偏移量
2. 数据库时区
3. 操作系统本地时区
4. 命名区域

答案：2、3、4

小结

在本课中，您应该已经学会如何使用下列函数：

- CURRENT_DATE
- CURRENT_TIMESTAMP
- LOCALTIMESTAMP
- DBTIMEZONE
- SESSIONTIMEZONE
- EXTRACT
- TZ_OFFSET
- FROM_TZ
- TO_TIMESTAMP
- TO_YMINTERVAL
- TO_DSINTERVAL

ORACLE

小结

本课介绍了在 Oracle DB 中提供的某些日期时间函数。

练习 5：概览

在此练习中，您将练习使用日期时间函数。

ORACLE

练习 5：概览

在本练习中，您将显示时区偏移量、CURRENT_DATE、CURRENT_TIMESTAMP 和 LOCALTIMESTAMP。还要设置时区和使用 EXTRACT 函数。

6

使用子查询检索数据

课程目标

学完本课后，应能完成下列工作：

- 编写多列子查询
- 在 SQL 中使用标量子查询
- 使用相关子查询解决问题
- 使用相关子查询更新和删除行
- 使用 EXISTS 和 NOT EXISTS 运算符
- 使用 WITH 子句

ORACLE

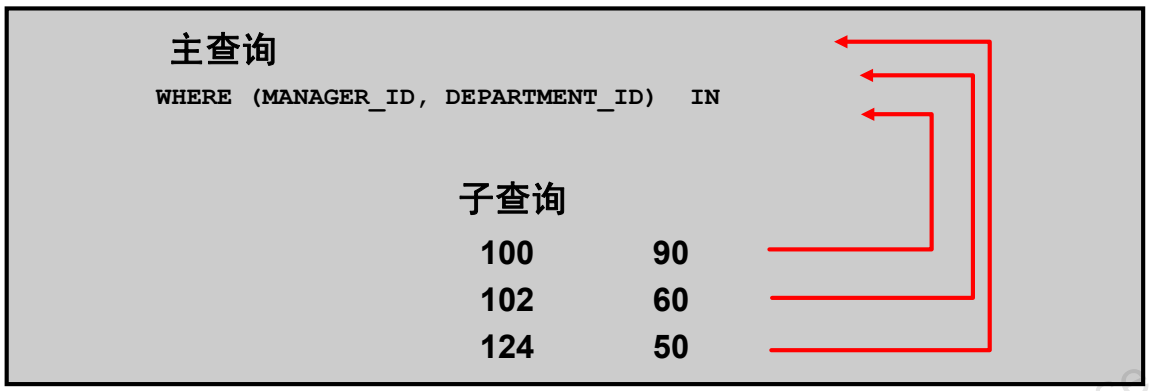
课程目标

在本课中，您将学习如何在 SELECT 语句的 FROM 子句中编写多列子查询。还将学习如何使用标量子查询、相关子查询和 WITH 子句解决问题。

课程安排

- 编写多列子查询
- 在 SQL 中使用标量子查询
- 使用相关子查询解决问题
- 使用 EXISTS 和 NOT EXISTS 运算符
- 使用 WITH 子句

多列子查询



主查询的每一行都会与多行和多列子查询中的各个值进行比较。

多列子查询

到目前为止，您已编写过单行子查询和多行子查询，其中内部 SELECT 语句只返回一列，此列用来计算父 SELECT 语句中表达式的值。如果要比较两列或多列，则必须使用逻辑运算符编写复合 WHERE 子句。使用多列子查询时，可将重复的 WHERE 条件合并在一条 WHERE 子句中。

语法

```
SELECT     column, column, ...
FROM  table
WHERE (column, column, ...) IN
              (SELECT column, column, ...
                                  FROM    table
                                  WHERE   condition);
```

幻灯片图中显示主查询中的 MANAGER_ID 值和 DEPARTMENT_ID 值要与子查询检索到的 MANAGER_ID 值和 DEPARTMENT_ID 值进行比较。因为要比较的列数大于 1，所以该示例是一个多列子查询。

注：运行下面几张幻灯片上的那些示例之前，需要创建 empl_demo 表，还要使用 lab_06_insert_empdata.sql 文件在该表中填充数据。

列比较

涉及子查询的多列比较可以是：

- 不成对比较
- 成对比较

成对比较与不成对比较

涉及子查询的多列比较可以是不成对比较，也可以是成对比较。例如，如果要显示与‘Daniel’在同一部门工作且在同一位经理领导之下的雇员的详细资料，可以使用以下语句得到正确的结果：

```
SELECT first_name, last_name, manager_id, department_id
FROM empl_demo
WHERE manager_id IN (SELECT manager_id
                     FROM empl_demo
                     WHERE first_name = 'Daniel')
AND department_id IN (SELECT department_id
                      FROM empl_demo
                      WHERE first_name = 'Daniel');
```

在 EMPL_DEMO 表中，只有一个“Daniel”，即 Daniel Faviet，他在雇员 108 的领导之下且在部门 100 中工作。但是，如果子查询返回多行，结果就有可能不正确。例如，如果运行同一查询，但用“John”替代“Daniel”，就会得到不正确的结果。这是因为 department_id 与 manager_id 的组合很重要。要使此查询得到正确的结果，需要进行成对比较。

成对比较子查询

显示与名字为“John”的雇员在同一经理领导下且在同一部门中工作的雇员的详细资料。

```
SELECT  employee_id, manager_id, department_id
FROM    empl_demo
WHERE   (manager_id, department_id) IN
                    (SELECT manager_id, department_id
                     FROM empl_demo
                     WHERE first_name = 'John')
AND first_name <> 'John';
```

ORACLE

成对比较子查询

幻灯片示例中 EMPL_DEMO 表每一行上的 MANAGER_ID 列值和 DEPARTMENT_ID 列值的组合要与 FIRST_NAME 为“John”的雇员的 MANAGER_ID 列值和 DEPARTMENT_ID 列值进行比较。首先，在子查询中会检索 FIRST_NAME 为“John”的雇员的 MANAGER_ID 值和 DEPARTMENT_ID 值。此子查询会返回以下结果：

	MANAGER_ID	DEPARTMENT_ID
1	108	100
2	123	50
3	100	80

成对比较子查询（续）

这些值要与 EMPL_DEMO 表每一行上的 MANAGER_ID 列和 DEPARTMENT_ID 列进行比较。如果组合相匹配，则显示该行。在输出中，将不会显示 FIRST_NAME 为“John”的雇员的记录。幻灯片中查询的输出如下所示：

	EMPLOYEE_ID	MANAGER_ID	DEPARTMENT_ID
1	113	108	100
2	112	108	100
3	111	108	100
4	109	108	100
5	195	123	50
6	194	123	50
7	193	123	50
8	192	123	50
9	140	123	50
10	138	123	50
11	137	123	50
12	149	100	80
13	148	100	80
14	147	100	80
15	146	100	80

不成对比较子查询

显示与名字为“John”的雇员在同一经理领导下且在同一部门中工作的雇员的详细资料。

```
SELECT  employee_id, manager_id, department_id
FROM    empl_demo
WHERE   manager_id IN
                (SELECT manager_id
                 FROM empl_demo
                 WHERE first_name = 'John')
AND department_id IN
                (SELECT department_id
                 FROM empl_demo
                 WHERE first_name = 'John')
AND first_name <> 'John';
```


不成对比较子查询

该示例显示对列进行不成对比较。首先，在第一个子查询中检索 FIRST_NAME 为“John”的雇员的 MANAGER_ID 值。然后同样在第二个子查询中检索 FIRST_NAME 为“John”的雇员的 DEPARTMENT_ID 值。检索到的 MANAGER_ID 列值和 DEPARTMENT_ID 列值要与 EMPL_DEMO 表每一行上的 MANAGER_ID 列和 DEPARTMENT_ID 列进行比较。如果 EMPL_DEMO 表某一行上的 MANAGER_ID 列与内部子查询检索到的任何 MANAGER_ID 值相匹配，且 EMPL_DEMO 表中该行上的 DEPARTMENT_ID 列与第二个子查询检索到的任何 DEPARTMENT_ID 值相匹配，则显示该记录。

不成对比较子查询（续）

上一张幻灯片中查询的输出如下所示：

	EMPLOYEE_ID	MANAGER_ID	DEPARTMENT_ID
1	109	108	100
2	111	108	100
3	112	108	100
4	113	108	100
5	120	100	50
6	121	100	50
7	122	100	50
8	123	100	50
9	124	100	50
10	137	123	50
11	138	123	50
12	140	123	50
13	192	123	50
14	193	123	50
15	194	123	50
16	195	123	50
17	146	100	80
18	147	100	80
19	148	100	80
20	149	100	80

此查询比成对比较要多检索几行（虽然名为“John”的雇员没有这样的组合，但还是会检索 `manager_id=100` 和 `department_id=50` 或 80 组合对应的某些行）。

课程安排

- 编写多列子查询
- 在 SQL 中使用标量子查询
- 使用相关子查询解决问题
- 使用 EXISTS 和 NOT EXISTS 运算符
- 使用 WITH 子句

标量子查询表达式

- 标量子查询表达式是从一行中只返回一个列值的子查询。
- 标量子查询可用于：
  - DECODE 和 CASE 的条件和表达式部分
  - SELECT 语句中除 GROUP BY 之外的所有子句
  - UPDATE 语句中的 SET 子句和 WHERE 子句

SQL 中的标量子查询

从一行中只返回一个列值的子查询又称为标量子查询。使用复合 WHERE 子句和逻辑运算符编写的多列子查询可用来对两列或多列进行比较，这种子查询不能称为标量子查询。

标量子查询表达式的值就是子查询选择列表项的值。如果子查询返回 0 行，则标量子查询表达式的值为 NULL。如果子查询返回多行，则 Oracle Server 返回一条错误。Oracle Server 始终支持在 SELECT 语句中使用标量子查询。标量子查询可用于：

- DECODE 和 CASE 的条件和表达式部分
- SELECT 语句中除 GROUP BY 之外的所有子句
- UPDATE 语句中的 SET 子句和 WHERE 子句

但是，标量子查询在以下场合不是有效表达式：

- 作为列的默认值和集群的散列表达式时
- 用于数据操纵语言 (DML) 语句中的 RETURNING 子句时
- 作为基于函数的索引的基准时
- 用于 GROUP BY 子句、CHECK 约束条件、WHEN 条件中时
- 用于 CONNECT BY 子句中时
- 用于与查询无关联的语句（如 CREATE PROFILE）中时

标量子查询：示例

- CASE 表达式中的标量子查询：

```
SELECT employee_id, last_name,
       (CASE
        WHEN department_id =
                (SELECT department_id
                 FROM departments
                     WHERE location_id = 1800)
         THEN 'Canada' ELSE 'USA' END) location
FROM   employees;
```

20

- ORDER BY 子句中的标量子查询：

```
SELECT   employee_id, last_name
FROM     employees e
ORDER BY (SELECT department_name
          FROM departments d
          WHERE e.department_id = d.department_id);
```

ORACLE

标量子查询：示例

幻灯片第一个示例中显示可以在 CASE 表达式中使用标量子查询。内部查询返回值 20，该值是位置 ID 为 1800 的部门的部门 ID。外部查询中的 CASE 表达式会使用内部查询的结果来显示雇员 ID、姓氏以及值“Canada”或“USA”，这取决于外部查询检索记录的部门 ID 是否为 20。

幻灯片第一个示例的结果如下所示：

...

	EMPLOYEE_ID	LAST_NAME	LOCATION
1	198	OConnell	USA
2	199	Grant	USA
3	200	Whalen	USA
4	201	Hartstein	Canada
5	202	Fay	Canada
6	203	Mavris	USA

标量子查询：示例（续）

幻灯片第二个示例中显示可以在 ORDER BY 子句中使用标量子查询。在该示例中，将 EMPLOYEES 表中的 DEPARTMENT_ID 与 DEPARTMENTS 表中的 DEPARTMENT_ID 相匹配时会根据 DEPARTMENT_NAME 对输出进行排序。此比较通过在 ORDER BY 子句中使用标量子查询来完成。第二个示例的结果如下所示：

	EMPLOYEE_ID	LAST_NAME
1	205	Higgins
2	206	Gietz
3	200	Whalen
4	100	King
5	101	Kochhar
6	102	De Haan
7	112	Urman
8	108	Greenberg
9	109	Faviet

...

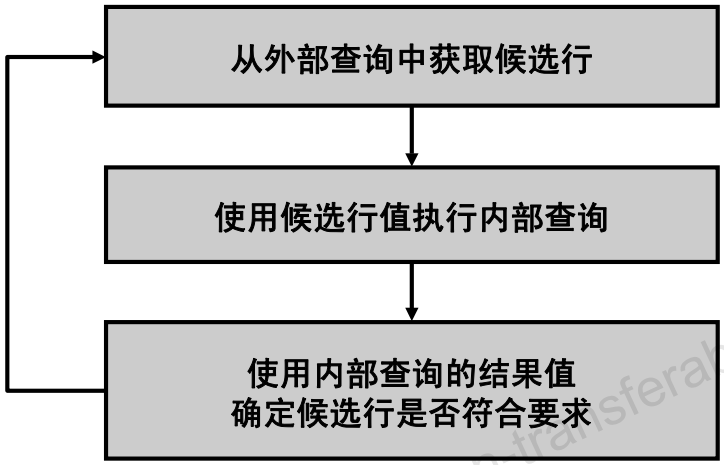
第二个示例使用了相关子查询。在相关子查询中，子查询会引用父语句中所引用表中的一列。本课稍后对相关子查询进行介绍。

课程安排

- 编写多列子查询
- 在 SQL 中使用标量子查询
- 使用相关子查询解决问题
- 使用 EXISTS 和 NOT EXISTS 运算符
- 使用 WITH 子句

相关子查询

在进行逐行处理时会使用相关子查询。每一个子查询都会为外部查询的每一行而执行一次。



ORACLE

相关子查询

当子查询引用父语句所引用表中的一列时，Oracle Server 就会执行相关子查询。相关子查询会为父语句处理的每一行而计算一次。父语句可以是 SELECT、UPDATE 或 DELETE 语句。

嵌套子查询与相关子查询

使用普通的嵌套子查询时，内部 SELECT 查询会首先运行且只执行一次，就会返回供主查询使用的值。但是，相关子查询会为外部查询所考虑的每一候选行而执行一次。也就是说，内部查询是由外部查询驱动的。

嵌套子查询的执行过程

- 内部查询首先执行并查找一个值。
- 外部查询使用内部查询的结果值执行一次。

相关子查询的执行过程

- 获得一个候选行（由外部查询提取）。
- 使用候选行的值执行内部查询。
- 使用内部查询的结果值确定候选行是否符合要求。
- 重复执行操作，直到没有候选行为止。

相关子查询

子查询引用父查询所引用表中的一列。

```
SELECT column1, column2, ...
FROM   table1   Outer_table
WHERE  column1 operator
                    (SELECT   column1, column2
                     FROM     table2
                     WHERE    expr1 =
                              Outer_table.expr2);
```


相关子查询（续）

相关子查询提供了一种方法，可用来在读取表中的每一行后将读出值与相关数据进行比较。如果子查询必须为主查询所考虑的每一候选行返回不同的结果或结果集，则需要使用相关子查询。也就是说，使用相关子查询可以解决一个由多个部分组成的问题，这个问题中各个部分的答案取决于父语句处理的每一行上的值。

当子查询引用父查询所引用表中的一列时，Oracle Server 就会执行相关子查询。

注：可在相关子查询中使用 ANY 和 ALL 运算符。

使用相关子查询

查找其薪金高于所在部门的平均薪金的所有雇员。

```
SELECT last_name, salary, department_id
FROM   employees outer_table
WHERE  salary >
              (SELECT AVG(salary)
               FROM   employees inner_table
               WHERE inner_table.department_id =
               outer_table.department_id);
```

每次在外部查询中处理一行时，都会计算内部查询的值。

使用相关子查询

幻灯片示例中确定了其薪金高于所在部门的平均薪金的雇员。在此例中，相关子查询专门计算每个部门的平均薪金。

因为外部查询和内部查询都在 FROM 子句中使用了 EMPLOYEES 表，为清楚起见，为外部 SELECT 语句中的 EMPLOYEES 指定了别名。使用别名可使整条 SELECT 语句更具可读性。如果不使用别名，查询将无法正常进行，因为内部语句无法区分内部表列和外部表列。

使用相关子查询

显示至少调换过两次职务的那些雇员的详细资料。

```
SELECT e.employee_id, last_name,e.job_id
FROM   employees e
WHERE  2 <= (SELECT COUNT(*)
             FROM   job_history
             WHERE  employee_id = e.employee_id);
```

	EMPLOYEE_ID	LAST_NAME	JOB_ID
1	200	Whalen	AD_ASST
2	101	Kochhar	AD_VP
3	176	Taylor	SA_REP

使用相关子查询（续）

幻灯片示例中显示至少调换过两次职务的那些雇员的详细资料。Oracle Server 按以下方式计算相关子查询的值：

1. 从外部查询指定的表中选择一行。该行是当前候选行。
2. 存储子查询所引用列中来自此候选行上的值。（在幻灯片示例中，子查询所引用的列是 E.EMPLOYEE_ID。）
3. 执行子查询，其条件中引用了外部查询候选行上的值。（在幻灯片示例中，根据步骤 2 获取的 E.EMPLOYEE_ID 列的值来计算组函数 COUNT(*) 的值。）
4. 根据步骤 3 中执行子查询的结果，计算外部查询中 WHERE 子句的值。这确定了是否选择候选行作为输出。（在该示例中，雇员调换职务的次数，即由子查询计算出来的值，要与外部查询 WHERE 子句中的 2 进行比较。如果满足条件，则显示该雇员记录。）
5. 对表中下一个候选行重复该过程，直到表中所有行都处理完为止。

由于在子查询中使用了来自外部查询的元素，所以就建立了相关性。在本示例中，子查询中表的 EMPLOYEE_ID 与外部查询中表的 EMPLOYEE_ID 进行了比较。

课程安排

- 编写多列子查询
- 在 SQL 中使用标量子查询
- 使用相关子查询解决问题
- 使用 EXISTS 和 NOT EXISTS 运算符
- 使用 WITH 子句

使用 EXISTS 运算符

- EXISTS 运算符用来测试在子查询的结果集中是否存在有关行。
- 如果找到了子查询行值：
  - 则不在内部查询中继续进行搜索
  - 该条件会标记为 TRUE
- 如果未找到子查询行值：
  - 该条件会标记为 FALSE
  - 在内部查询中继续进行搜索

EXISTS 运算符

使用嵌套 SELECT 语句时，所有逻辑运算符都有效。此外，还可以使用 EXISTS 运算符。此运算符经常与相关子查询配合使用，目的是为了测试外部查询检索到的值是否存在于内部查询检索到的值结果集中。如果子查询至少返回一行，则该运算符返回 TRUE。如果该值不存在，则返回 FALSE。相应地，NOT EXISTS 运算符用来测试外部查询检索到的值是否不在内部查询检索到的值结果集中。

使用 EXISTS 运算符

```
SELECT employee_id, last_name, job_id, department_id
FROM   employees outer
WHERE  EXISTS ( SELECT 'X'
                FROM   employees
                WHERE  manager_id =
                       outer.employee_id);
```

	EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
1	201	Hartstein	MK_MAN	20
2	205	Higgins	AC_MGR	110
3	100	King	AD_PRES	90
4	101	Kochhar	AD_VP	90
5	102	De Haan	AD_VP	90
6	103	Hunold	IT_PROG	60
7	108	Greenberg	FI_MGR	100
8	114	Raphaely	PU_MAN	30

ORACLE

使用 EXISTS 运算符

当经理和雇员编号至少有一个满足以下条件的匹配项时，EXISTS 运算符就能保证不在内部查询中继续进行搜索：

```
WHERE manager_id = outer.employee_id.
```

请注意，内部 SELECT 查询不需要返回特定值，因此可以选择一个常数。

查找没有任何雇员的所有部门

```
SELECT department_id, department_name
FROM departments d
WHERE NOT EXISTS (SELECT 'X'
                  FROM   employees
                  WHERE  department_id = d.department_id);
```

	DEPARTMENT_ID	DEPARTMENT_NAME
1	120	Treasury
2	130	Corporate Tax
3	140	Control And Credit
4	150	Shareholder Services
5	160	Benefits
6	170	Manufacturing
7	180	Construction

...

All Rows Fetched: 16

使用 NOT EXISTS 运算符

另一种解决方案

NOT IN 构造可用来替代 NOT EXISTS 运算符，如下示例所示：

```
SELECT department_id, department_name
FROM   departments
WHERE  department_id NOT IN (SELECT department_id
                             FROM   employees);
```

All Rows Fetched: 0

但是，如果集合中任一成员的值为 NULL，则 NOT IN 的计算结果为 FALSE。因此，即使 department 表中存在满足 WHERE 条件的行，也不会在查询中返回任何行。

相关 UPDATE

使用相关子查询可根据一个表中的行来更新另一个表中的行。

```
UPDATE table1 alias1
SET    column = (SELECT expression
                 FROM   table2 alias2
                 WHERE  alias1.column =
                        alias2.column);
```


相关 UPDATE

在 UPDATE 语句中，使用相关子查询可根据一个表中的行来更新另一个表中的行。

使用相关 UPDATE

- 通过添加一个用于存储部门名称的列来改变 EMPL6 表。
- 使用相关 UPDATE 来填充该表。

```
ALTER TABLE empl6
ADD(department_name VARCHAR2(25));
```

```
UPDATE empl6 e
SET    department_name =
               (SELECT department_name
               FROM   departments d
               WHERE  e.department_id = d.department_id);
```


相关 UPDATE（续）

幻灯片示例中通过添加一个用于存储部门名称的列改变了 EMPL6 表，然后使用相关 UPDATE 来填充该表。

下面是相关 UPDATE 的另一个示例。

问题说明

REWARDS 表中包含一个雇员列表，这些雇员的业绩比预期的好。使用相关子查询，根据 REWARDS 表中的行来更新 EMPL6 表中的行。

```
UPDATE empl6
SET    salary = (SELECT empl6.salary + rewards.pay_raise
                 FROM   rewards
                 WHERE  employee_id  =
                        empl6.employee_id
                 AND  payraise_date =
                      (SELECT MAX(payraise_date)
                      FROM   rewards
                      WHERE  employee_id = empl6.employee_id))
WHERE  empl6.employee_id
IN     (SELECT employee_id FROM rewards);
```

相关 UPDATE（续）

此示例使用了 REWARDS 表。REWARDS 表包含以下各列：EMPLOYEE_ID、PAY_RAISE 和 PAYRAISE_DATE。每次雇员加薪时，就会在 REWARDS 表中插入一条包含雇员 ID、加薪金额和加薪日期等详细资料的记录。REWARDS 表可以包含一个雇员的多条记录。PAYRAISE_DATE 列用于标识雇员最近的加薪。

在该示例中，EMPL6 表中的 SALARY 列更新后会反映雇员最近的加薪。通过将雇员的当前薪金与 REWARDS 表中的相应加薪相加，就可以实现此更新。

相关 DELETE

使用相关子查询可根据一个表中的行删除另一个表中的行。

```
DELETE FROM table1 alias1
WHERE  column operator
               (SELECT expression
                FROM   table2 alias2
                WHERE  alias1.column = alias2.column);
```


相关 DELETE

在 DELETE 语句中，可以使用相关子查询来仅删除同时还存在于另一个表中的那些行。如果决定在 JOB_HISTORY 表中只维护 4 条最新职务历史记录，那么当雇员第 5 次调换职务时，可以通过在 JOB_HISTORY 表中查找雇员的 MIN(START_DATE) 来删除最早的 JOB_HISTORY 行。下面的代码说明如何使用相关 DELETE 执行上述操作：

```
DELETE FROM emp_history JH
WHERE employee_id =
      (SELECT employee_id
       FROM employees E
       WHERE JH.employee_id = E.employee_id
       AND START_DATE =
             (SELECT MIN(start_date)
              FROM job_history JH
              WHERE JH.employee_id = E.employee_id)
              AND 5 >  (SELECT COUNT(*)
                        FROM job_history JH
                        WHERE JH.employee_id = E.employee_id
                        GROUP BY EMPLOYEE_ID
                        HAVING COUNT(*) >= 4));
```

使用相关 DELETE

使用相关子查询可仅删除 EMPL6 表中同时还存在于 EMP_HISTORY 表中的那些行。

```
DELETE FROM empl6 E
WHERE employee_id =
            (SELECT employee_id
             FROM   emp_history
             WHERE  employee_id = E.employee_id);
```


相关 DELETE（续）

示例

此示例中使用了两个表。其中包括：

- EMPL6 表，该表提供所有当前雇员的详细资料
- EMP_HISTORY 表，该表提供以前雇员的详细资料

EMP_HISTORY 包含有关以前雇员的数据，因此如果同一雇员的记录既存在于 EMPL6 表中，又存在于 EMP_HISTORY 表中，则该记录就是错误的。使用幻灯片中显示的相关子查询，可以删除此类错误记录。

课程安排

- 编写多列子查询
- 在 SQL 中使用标量子查询
- 使用相关子查询解决问题
- 使用 EXISTS 和 NOT EXISTS 运算符
- 使用 WITH 子句

ORACLE

WITH 子句

- 如果某个查询块在一个复杂查询内多次出现，则使用 WITH 子句时可以在 SELECT 语句中使用同一查询块。
- WITH 子句会检索查询块的结果，并将结果存储在用户临时表空间中。
- WITH 子句可以提高性能。

WITH 子句

使用 WITH 子句时，可以在查询中使用某个查询块之前定义该查询块。如果某个查询块在一个复杂查询内多次出现，则使用 WITH 子句（以前称为 subquery_factoring_clause）时可以在 SELECT 语句中重复使用同一查询块。当一个查询多次引用同一查询块且存在联接和聚集时，该子句非常有用。

如果计算某个查询块花费的时间很长且该查询块在一个复杂查询中多次出现，那么使用 WITH 子句时可以重复使用同一查询。使用 WITH 子句时，Oracle Server 会检索查询块的结果，并将该结果存储在用户临时表空间中。这样可以提高性能。

WITH 子句的好处

- 使查询简单易读
- 只计算子句一次，即使子句在查询中多次出现
- 在大多数情况下，可以提高大型查询的性能

WITH 子句：示例

使用 WITH 子句编写一个查询来显示薪金总额高于各个部门平均薪金的部门的部门名称和薪金总额。

WITH 子句：示例

幻灯片中的问题需要用到下列中间计算：

1. 计算每个部门的薪金总额，并存储使用 WITH 子句的结果。
2. 计算各部门的平均薪金，并存储使用 WITH 子句的结果。
3. 将第一步计算出的薪金总额与第二步计算出的平均薪金进行比较。如果特定部门的薪金总额高于各部门的平均薪金，则显示该部门的部门名称和薪金总额。

此问题的解决方案显示在下一页。

WITH 子句：示例

```
WITH
dept_costs  AS (
   SELECT d.department_name, SUM(e.salary) AS dept_total
   FROM   employees e JOIN departments d
   ON     e.department_id = d.department_id
   GROUP BY d.department_name),
avg_cost    AS (
   SELECT SUM(dept_total)/COUNT(*) AS dept_avg
   FROM   dept_costs)
SELECT *
FROM   dept_costs
WHERE  dept_total >
        (SELECT dept_avg
         FROM avg_cost)
ORDER BY department_name;
```

ORACLE

WITH 子句：示例（续）

幻灯片中的 SQL 代码是一个示例，在这个示例中，通过使用 WITH 子句可提高性能和编写更简单的 SQL 语句。该查询创建了查询名称 DEPT_COSTS 和 AVG_COST，然后在主查询主体中使用它们。WITH 子句在内部解析为内嵌视图或临时表。优化程序会根据临时存储 WITH 子句的结果花费的时间或带来的好处来选择适当的解决方案。

幻灯片中 SQL 代码生成的输出如下所示：

	DEPARTMENT_NAME	DEPT_TOTAL
1	Sales	304500
2	Shipping	156400

WITH 子句使用注意事项

- 只能与 SELECT 语句一起使用。
- 查询名称对于在其后定义的所有 WITH 元素查询块（包括其子查询块）和主查询块本身（包括其子查询块）都是可见的。
- 如果查询名称与现有表名相同，则分析程序会从内向外进行搜索，且查询块名称的优先级高于表名。
- WITH 子句中可以有多个查询。各个查询之间可用逗号分隔开。

递归 WITH 子句

递归 WITH 子句

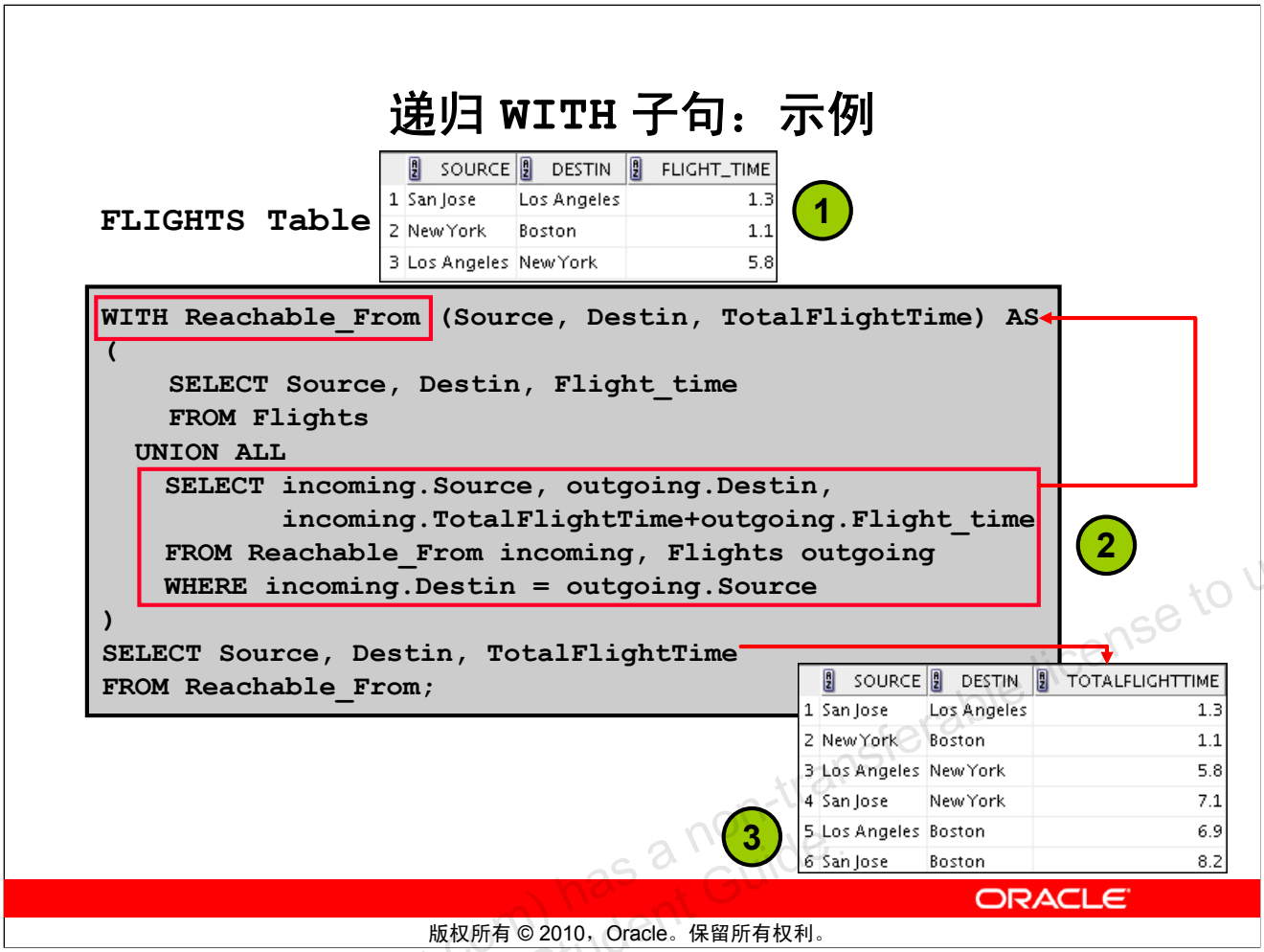
- 实现递归查询的公式化
- 创建用递归 WITH 元素名命名的查询
- 包含两类查询块成员：锚定点和递归
- 符合 ANSI 标准

递归 WITH 子句

在 Oracle Database 11g 发行版 2 中，WITH 子句得到了扩展，可实现递归查询的公式化。

递归 WITH 使用递归 WITH 元素名定义递归查询。递归 WITH 元素定义必须至少包含两个查询块：锚定点成员和递归成员。锚定点成员可以有多个，但递归成员只能有一个。

递归 WITH 子句，Oracle Database 11g 发行版 2 部分地符合美国国家标准协会 (ANSI) 的标准。递归 WITH 可用于查询分层数据，如组织图表。



递归 WITH 子句：示例

幻灯片中的示例 1 显示了 FLIGHTS 表中的记录，该表介绍了两个城市之间的航班情况。

使用示例 2 中的查询，可以查询 FLIGHTS 表以显示任何出发地和目的地之间的总飞行时间。该查询中的 WITH 子句（名为 Reachable_From）的 UNION ALL 查询具有两个分支。第一个分支是锚定点分支，可选定 Flights 表中的所有行。第二个分支是递归分支。它将 Reachable_From 的内容和 Flights 表相结合，以查找可以到达的其它城市，并将这些信息添加到 Reachable_From 的内容中。此操作将在递归分支无法再找到更多的行时结束。

示例 3 显示了从 WITH 子句元素 Reachable_From 选取所有信息的查询结果。

有关详细信息，请参阅：

- Oracle Database SQL Language Reference 11*g* Release 2.0 (11.1)
- Oracle Database Data Warehousing Guide 11*g* Release 2.0

小测验

使用相关子查询，内部 SELECT 语句将驱动外部 SELECT 语句。

1. 正确
2. 错误

答案：2

小结

在本课中，您应该已经学会：

- 多列子查询返回多列
- 多列比较可以是成对比较，也可以是不成对比较
- 多列子查询还可用在 SELECT 语句的 FROM 子句中

小结

可以使用多列子查询将多个 WHERE 条件组合在一条 WHERE 子句。多列子查询中的列比较可以是成对比较，也可以是不成对比较。

可以使用子查询定义一个主查询会对其执行操作的表。

标量子查询可用于：

- DECODE 和 CASE 的条件和表达式部分
- SELECT 语句中除 GROUP BY 之外的所有子句
- UPDATE 语句中的 SET 子句和 WHERE 子句

小结

- 当子查询必须为每一候选行返回不同的结果时，使用相关子查询可以提供帮助
- EXIST 运算符是一个布尔运算符，用于测试某个值是否存在
- 相关子查询可以与 SELECT、UPDATE 和 DELETE 语句配合使用
- 如果某一查询块出现多次，则在使用 WITH 子句时可在 SELECT 语句中重复使用同一查询块

小结（续）

当子查询引用父语句所引用表中的一列时，Oracle Server 就会执行相关子查询。相关子查询会为父语句处理的每一行而计算一次。父语句可以是 SELECT、UPDATE 或 DELETE 语句。当重复计算某个查询块花费的时间很长且该查询块在一个复杂查询中多次出现时，使用 WITH 子句时可以重复使用同一查询。

练习 6：概览

本练习包含以下主题：

- 创建多列子查询
- 编写相关子查询
- 使用 EXISTS 运算符
- 使用标量子查询
- 使用 WITH 子句

练习 6：概览

在此练习中，您需要编写多列子查询、相关子查询和标量子查询。还要通过编写 WITH 子句解决问题。

7

正则表达式支持功能

课程目标

学完本课后，应能完成下列工作：

- 列出使用正则表达式的好处
- 使用正则表达式搜索、匹配和替换字符串

课程目标

在本课中，您将学习使用正则表达式支持功能。在 SQL 和 PL/SQL 中都提供了正则表达式支持功能。

课程安排

- 正则表达式简介
- 在正则表达式中使用元字符
- 使用正则表达式函数：
  - REGEXP_LIKE
  - REGEXP_REPLACE
  - REGEXP_INSTR
  - REGEXP_SUBSTR
- 访问子表达式
- 使用 REGEXP_COUNT 函数
- 正则表达式和检查约束条件

ORACLE

正则表达式是什么

- 使用正则表达式可利用标准语法惯例搜索（和处理）简单和复杂模式的字符串数据。
- 使用一组 SQL 函数和条件可在 SQL 和 PL/SQL 中搜索和处理字符串。
- 指定正则表达式时请使用：
  - 元字符，这些是指定搜索算法的运算符
  - 文字，这些是要搜索的字符

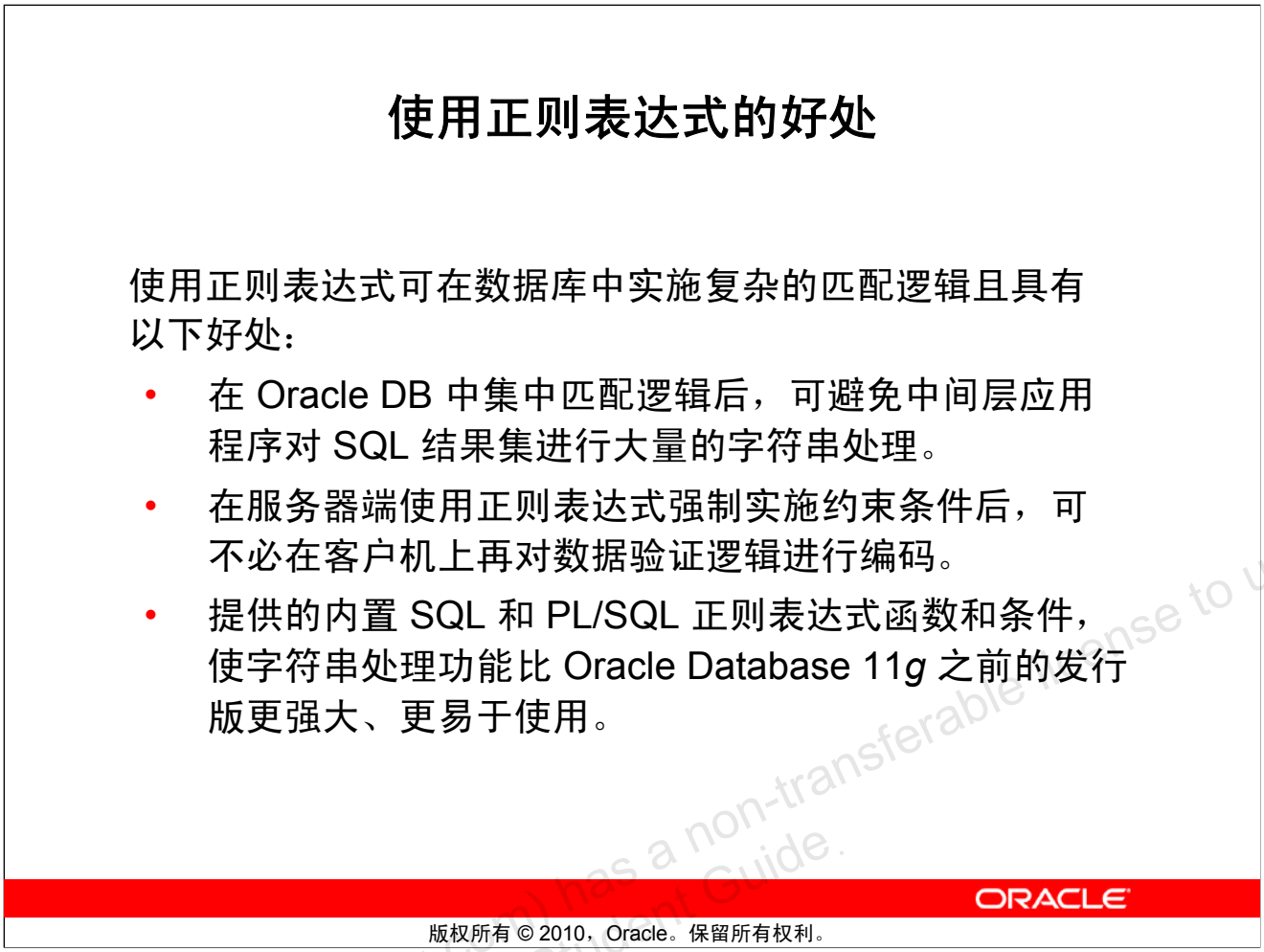
ORACLE

正则表达式是什么

在 Oracle DB 中支持使用正则表达式。这个实施遵守电气电子工程协会 (IEEE) 制定的 UNIX 可移植操作系统 (POSIX) 标准，符合 ASCII 数据的语义和语法。Oracle 的多语言功能扩展了匹配 POSIX 标准以外的运算符的能力。正则表达式提供了一种方法，使用这种方法可描述简单和复杂模式的搜索处理操作。

字符串搜索处理操作在基于 Web 运行的应用程序所执行的各种操作中占很大的百分比。所以正则表达式的应用范围很广泛，无论是简单任务（如在特定文本中查找“San Francisco”），还是复杂任务（如从文本中提取 URL），甚至更复杂的任务（如查找第二个字符是元音的所有单词），都可以使用正则表达式。

在本机 SQL 中使用正则表达式时，可以对 Oracle DB 中存储的任何数据执行功能强大的搜索处理操作。使用这个功能可以轻松地解决可能需要通过复杂编程才能解决的某些问题。



使用正则表达式的好处

正则表达式是一种采用编程语言（如 PERL 和 Java）处理文本的强大功能组件。例如，PERL 脚本可以处理目录中的每一个 HTML 文件，将其内容以单个字符串的形式读取到标量变量中，然后使用正则表达式在该字符串中搜索 URL。许多开发人员都愿意使用 PERL 编程，其中一个原因就是它具有强大的模式匹配功能。由于在 Oracle 中支持使用正则表达式，因此开发人员可在数据库中实施复杂的匹配逻辑。这种技术之所以很有用，原因如下：

- 在 Oracle DB 中集中匹配逻辑后，可避免中间层应用程序对 SQL 结果集进行大量的字符串处理。SQL 正则表达式函数使处理逻辑与数据的关系更紧密，从而提供了更有效的解决方案。
- 在 Oracle Database 10*g* 之前，开发人员常常要在客户机上对数据验证逻辑进行编码，因而需要在多个客户机上为相同的验证逻辑重复进行编码。在服务器端使用正则表达式强制实施约束条件后，就解决了这个问题。
- 提供的内置 SQL 和 PL/SQL 正则表达式函数和条件，使字符串处理功能比 Oracle Database 10*g* 之前的发行版更强大、更易于使用。

在 SQL 和 PL/SQL 中使用正则表达式函数和条件

函数名或条件名	说明
REGEXP_LIKE	与 LIKE 运算符相似，但执行正则表达式匹配，而不执行简单模式匹配（条件）
REGEXP_REPLACE	搜索到某个正则表达式模式后用替换字符串替换该模式
REGEXP_INSTR	在字符串中搜索到某个正则表达式模式后返回找到匹配项的位置
REGEXP_SUBSTR	在指定字符串中搜索到某个正则表达式模式后提取匹配的子字符串
REGEXP_COUNT	返回在输入字符串中找到某个模式匹配项的次数

ORACLE

在 SQL 和 PL/SQL 中使用正则表达式函数和条件

在 Oracle DB 中提供了一组 SQL 函数，这样您可以使用正则表达式来搜索和处理字符串。可以对文本文字、绑定变量或存放字符数据（如 CHAR、NCHAR、CLOB、NCLOB、NVARCHAR2 和 VARCHAR2 等，但不包括 LONG）的任何列使用这些函数。正则表达式必须放在单引号内。这可确保整个表达式由 SQL 函数进行解释，还可提高代码的可读性。

- REGEXP_LIKE：这是一个条件，用于搜索字符列中的一个模式。在查询的 WHERE 子句中使用此条件，可返回与指定的正则表达式相匹配的行。
- REGEXP_REPLACE：这是一个函数，用于在搜索到字符列中的一个模式后将每次出现的该模式替换为指定的模式。
- REGEXP_INSTR：这是一个函数，用于在字符串中搜索指定出现的正则表达式模式。应指定要找到哪个出现及搜索的开始位置。此函数返回一个整数，指示在字符串中找到匹配项的位置。
- REGEXP_SUBSTR：这是一个函数，用于返回与指定的正则表达式模式相匹配的实际子字符串。
- REGEXP_COUNT：这是一个函数，用于返回在输入字符串中找到某个模式匹配项的次数。

课程安排

- 正则表达式简介
- 在正则表达式中使用元字符
- 使用正则表达式函数：
  - REGEXP_LIKE
  - REGEXP_REPLACE
  - REGEXP_INSTR
  - REGEXP_SUBSTR
- 访问子表达式
- 使用 REGEXP_COUNT 函数

ORACLE

元字符是什么

- 元字符是具有特殊含义的特殊字符，如通配符、重复字符、非匹配字符或某个范围的字符。
- 可以在模式匹配中使用若干个预定义的元字符符号。
- 例如，使用 `^(f|ht)tps?:$` 正则表达式可从字符串的开始处搜索以下内容：
  - 文字 `f` 或 `ht`
  - 文字 `t`
  - 文字 `p`，后可跟文字 `s`
  - 字符串末尾的冒号文字“:”

元字符是什么

幻灯片中的正则表达式匹配 http:、https:、ftp: 和 ftps: 字符串。

注：有关正则表达式元字符的完整列表，请参阅《Oracle Database Advanced Application Developer's Guide 11*g* Release 2》。

在正则表达式中使用元字符

语法	说明
.	匹配支持字符集中的任一字符，但 NULL 除外
+	匹配一个或多个出现
?	匹配零个或一个出现
*	匹配其中出现零个或多个前面子表达式的字符串
{m}	匹配其中只出现 *m* 个前面表达式的字符串
{m, }	匹配其中至少出现 *m* 个前面子表达式的字符串
{m,n}	匹配其中至少出现 *m* 个但不超过 *n* 个前面子表达式的字符串
[...]	匹配大括号内列表中的任一个字符
|	匹配其中一个候选项
(...)	将放在括号中的表达式视为一个单元。子表达式可以是一个文字字符串或是包括运算符的复杂表达式

ORACLE

在正则表达式函数中使用元字符

任一字符“**.**”：**a.b** 匹配字符串 **abb**、**acb** 和 **adb**，但不匹配 **acc**。

一个或多个“**+**”：**a+** 匹配字符串 **a**、**aa** 和 **aaa**，但不匹配 **bbb**。

零个或一个“**?**”：**ab?c** 匹配字符串 **abc** 和 **ac**，但不匹配 **abbc**。

零个或多个“*****”：**ab*c** 匹配字符串 **ac**、**abc** 和 **abbc**，但不匹配 **abb**。

准确计数“**{m}**”：**a{3}** 匹配字符串 **aaa**，但不匹配 **aa**。

最少计数“**{m,}**”：**a{3,}** 匹配字符串 **aaa** 和 **aaaa**，但不匹配 **aa**。

计数范围“**{m,n}**”：**a{3,5}** 匹配字符串 **aaa**、**aaaa** 和 **aaaaa**，但不匹配 **aa**。

匹配字符列表“**[...]**”：**[abc]** 匹配字符串 **all**、**bill** 和 **cold** 中的第一个字符，但不匹配字符串 **doll** 中的任何字符。

或“**|**”：**a|b** 匹配字符 **a** 或字符 **b**。

子表达式“**(...)**”：**(abc)?def** 匹配后跟 **def** 的可选字符串 **abc**。该表达式匹配 **abcdefghi** 和 **def**，但不匹配 **ghi**。子表达式可以是一个文字字符串或是包括运算符的复杂表达式。

在正则表达式中使用元字符

语法	说明
^	匹配字符串的开头
$	匹配字符串的结尾
\	将表达式中的后续元字符作为文字对待
\n	匹配第 *n* (1-9) 个前面的子表达式（括号中的任何内容）。括号用于记住一个表达式；向后引用时将引用该表达式
\d	一个数字字符
[:class:]	匹配属于指定 POSIX 字符类的任何字符
[^:class:]	匹配不在大括号内列表中的任何单个字符

在正则表达式函数中使用元字符（续）

行开头/行结尾定位点“^”和“$”：** **^def** 匹配字符串 **defghi** 中的 **def**，但不匹配 **abcdef** 中的 **def**。**def$ 匹配字符串 **abcdef** 中的 **def**，但不匹配字符串 **defghi** 中的 **def**。

转义符“\”： **\+** 将搜索字符 **+**。它匹配字符串 **abc+def** 中的加号，但不匹配 **Abcdef**。

向后引用“\n”： **(abc|def)xy\1** 匹配字符串 **abcxyabc** 和 **defxydef**，但不匹配 **abcxydef** 或 **abcxy**。使用向后引用可以搜索一个重复的字符串，而不必事先知道实际字符串是什么。例如，表达式 **^(.*)\1$** 匹配由两个相邻的相同字符串组成的行。

数字字符“\d”： 表达式 ^\[\d{3}\] \d{3}-\d{4}$ 匹配 [650] 555-1212，但不匹配 650-555-1212。

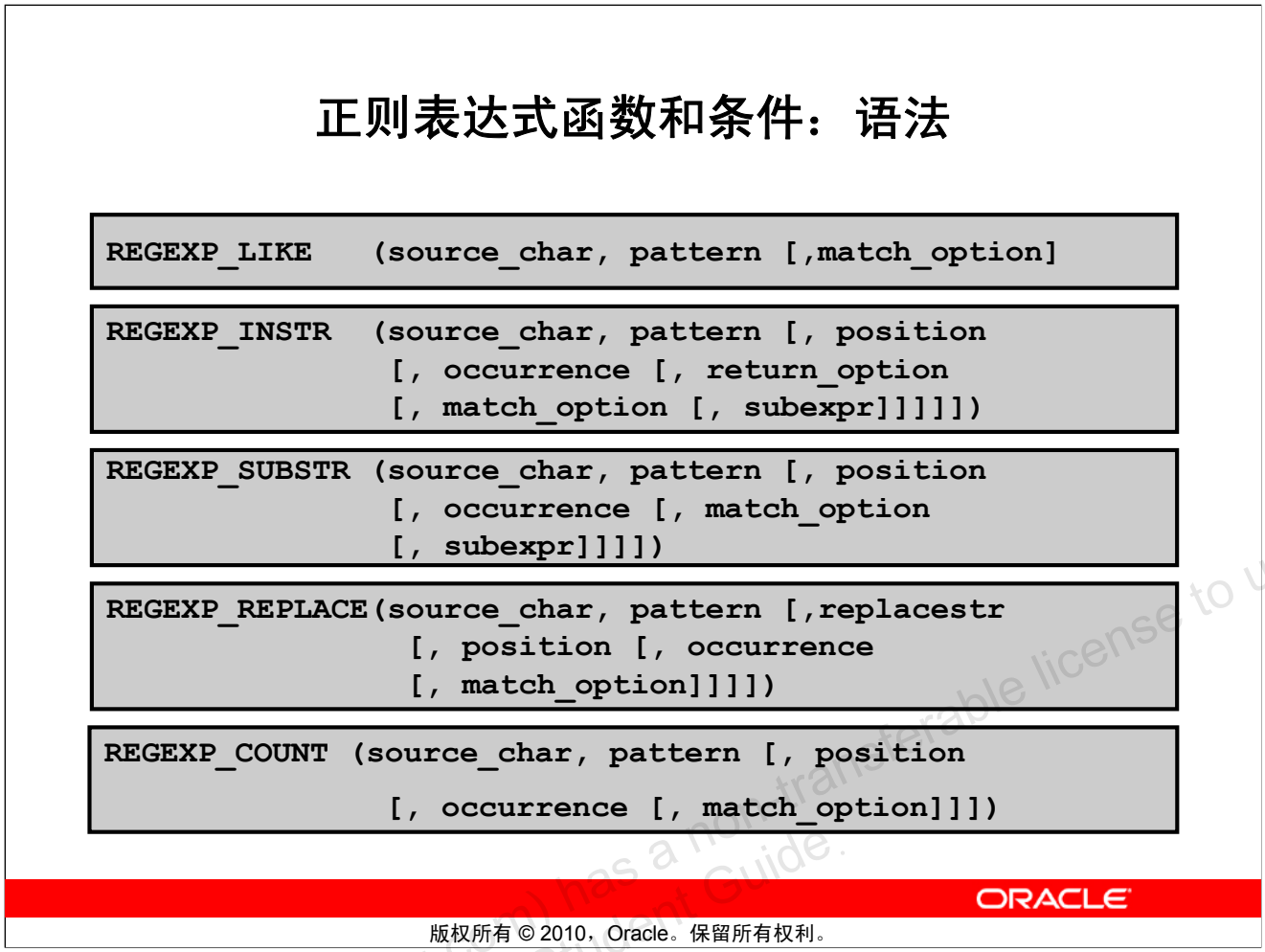
字符类“[:class:]”： **[[:upper:]]+** 将搜索一个或多个连续大写字符。这会匹配字符串 **abcDEFghi** 中的 **DEF**，但不匹配字符串 **abcdefghi**。

非匹配字符列表（或类）“[^...]”： **[^abc]** 匹配字符串 **abcdef** 中的字符 **d**，但不匹配字符 **a**、**b** 或 **c**。

课程安排

- 正则表达式简介
- 在正则表达式中使用元字符
- 使用正则表达式函数：
  - REGEXP_LIKE
  - REGEXP_REPLACE
  - REGEXP_INSTR
  - REGEXP_SUBSTR
- 访问子表达式
- 使用 REGEXP_COUNT 函数

ORACLE



正则表达式函数和条件：语法

正则表达式函数和条件的语法如下：

- source_char：用作搜索值的一个字符表达式。
- pattern：一个正则表达式，一个文本文字。
- occurrence：一个正整数，指示 Oracle Server 应搜索 source_char 中出现的哪个模式。默认值为 1。
- position：一个正整数，指示 Oracle Server 从 source_char 中的哪个字符开始搜索。默认值为 1。
- return_option：
  - 0：返回出现字符串中第一个字符的位置（默认值）
  - 1：返回出现字符串之后那个字符的位置
- Replacestr：字符串替换模式。

正则表达式函数和条件：语法（续）

- match_parameter：
  - “c”：使用区分大小写匹配（默认设置）
  - “i”：使用不区分大小写匹配
  - “n”：允许匹配任一字符运算符
  - “m”：将源字符串视为多行
- subexpr：括号中的模式片段。本课稍后将介绍有关子表达式的详细内容。

使用 REGEXP_LIKE 条件执行基本搜索

```
REGEXP_LIKE(source_char, pattern [, match_parameter ])
```

```
SELECT first_name, last_name
FROM employees
WHERE REGEXP_LIKE (first_name, '^Ste(v|ph)en$');
```

	FIRST_NAME	LAST_NAME
1	Steven	King
2	Steven	Markle
3	Stephen	Stiles

使用 REGEXP_LIKE 条件执行基本搜索

REGEXP_LIKE 类似于 LIKE 条件，但 REGEXP_LIKE 执行正则表达式匹配，而 LIKE 执行简单模式匹配。此条件使用由输入字符集定义的字符计算字符串的值。

REGEXP_LIKE 示例

在本查询中，将显示 EMPLOYEES 表中名字包含 Steven 或 Stephen 的所有雇员。在使用 '^Ste(v|ph)en$ ' 的表达式中：

- ^ 表示表达式的开头
- $ 表示表达式的结尾
- | 表示两者之一

使用 REGEXP_REPLACE 函数替换模式

```
REGEXP_REPLACE(source_char, pattern [,replacestr
[, position [, occurrence [, match_option]]]])
```

```
SELECT REGEXP_REPLACE(phone_number, '\.','-') AS phone
FROM employees;
```

原始数据

	LAST_NAME	PHONE
1	OConnell	650.507.9833
2	Grant	650.507.9844
3	Whalen	515.123.4444
4	Hartstein	515.123.5555

部分结果

	LAST_NAME	PHONE
1	OConnell	650-507-9833
2	Grant	650-507-9844
3	Whalen	515-123-4444
4	Hartstein	515-123-5555

ORACLE

使用 REGEXP_REPLACE 函数替换模式

使用 REGEXP_REPLACE 函数可以重新设定电话号码格式，将句点 (.) 分隔符替换为短划线 (-) 分隔符。以下是对正则表达式示例中使用的每个元素的说明：

- phone_number 是来源列。
- '\.' 是搜索模式。
  - 使用单引号 (' ') 搜索文字字符句点 (.)。
  - 使用反斜杠 (\) 搜索正常情况下作为元字符处理的字符。
- '-' 是替换字符串。

使用 REGEXP_INSTR 函数查找模式

```
REGEXP_INSTR  (source_char, pattern [, position [,
   occurrence [, return_option [, match_option]]]])
```

```
SELECT street_address,
REGEXP_INSTR(street_address,'[[:alpha:]]') AS
   First_Alpha_Position
FROM  locations;
```

	STREET_ADDRESS	FIRST_ALPHA_POSITION
1	1297 Via Cola di Rie	6
2	93091 Calle della Testa	7
3	2017 Shinjuku-ku	6
4	9450 Kamiya-cho	6

使用 REGEXP_INSTR 函数查找模式

在本示例中，REGEXP_INSTR 函数用来通过搜索街道地址来查找第一个字母字符的位置，而不考虑此字母字符是大写还是小写。请注意，[:<*class*>:] 表示一个字符类，可匹配该类中的任一字符，而 [:alpha:] 可匹配任一字母字符。部分结果如上所示。

在查询使用的表达式 '[[:alpha:]]' 中：

- [表示表达式开始
- [:alpha:] 表示字母字符类
-] 表示表达式结束

注：使用 POSIX 字符类运算符时，可以在属于指定 POSIX 字符类的一个字符列表中搜索一个表达式。使用这种运算符可以搜索特定格式的字符，如大写字符，也可以搜索特殊字符，如数字和标点符号。系统支持整个 POSIX 字符类集。使用语法 [:class:]，其中 class 是要搜索的 POSIX 字符类的名称。下面的正则表达式用来搜索一个或多个连续大写字符：[[:upper:]]+。

使用 REGEXP_SUBSTR 函数提取子字符串

```
REGEXP_SUBSTR (source_char, pattern [, position
               [, occurrence [, match_option]]])
```

```
SELECT REGEXP_SUBSTR(street_address , ' [^ ]+ ') AS Road
FROM locations;
```

	ROAD
1	Via
2	Calle
3	(null)
4	(null)
5	Jabberwocky

使用 REGEXP_SUBSTR 函数提取子字符串

在本示例中，要从 LOCATIONS 表中提取道路名称。为此，使用 REGEXP_SUBSTR 函数返回 STREET_ADDRESS 列中第一个空格后的内容。在查询使用的表达式 ' [^]+ ' 中：

- [表示表达式开始
- ^ 表示“非”
- 表示空格
-] 表示表达式结束
- + 表示一个或多个
- 表示空格

课程安排

- 正则表达式简介
- 在正则表达式中使用元字符
- 使用正则表达式函数：
  - REGEXP_LIKE
  - REGEXP_REPLACE
  - REGEXP_INSTR
  - REGEXP_SUBSTR
- 访问子表达式
- 使用 REGEXP_COUNT 函数

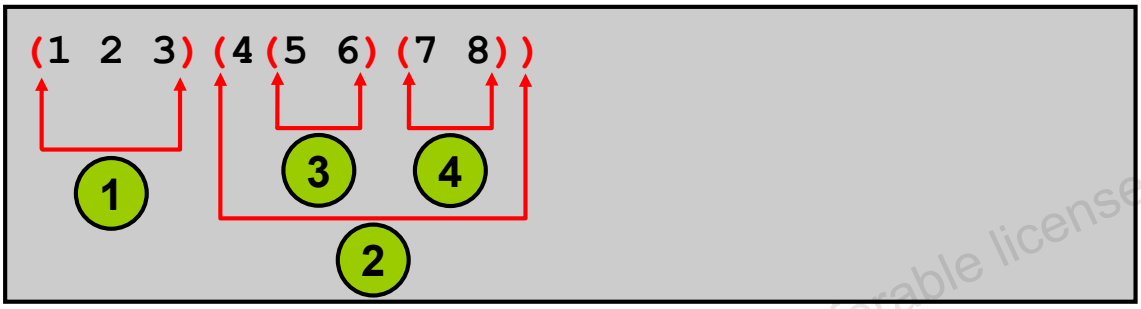
ORACLE

子表达式

请检查以下表达式：

```
(1  2  3)(4(5  6)(7  8))
```

子表达式有：



子表达式

Oracle Database 11*g* 提供了可用于访问子表达式的正则表达式支持参数。在幻灯片示例中，显示一个数字字符串。括号确定了该数字字符串中的子表达式。按从左往右，从外括号到内括号的顺序读取，该数字字符串中的子表达式有：

1. 123
2. 45678
3. 56
4. 78

可以使用 REGEXP_INSTR 和 REGEXP_SUBSTR 函数搜索上述子表达式中的任何一个。

结合使用正则表达式支持与子表达式

```
SELECT
  REGEXP_INSTR
① ('0123456789',       -- source char or search value
② '(123)(4(56)(78))', -- regular expression patterns
③ 1,                   -- position to start searching
④ 1,                   -- occurrence
⑤ 0,                   -- return option
⑥ 'i',                 -- match option (case insensitive)
⑦ 1)              -- sub-expression on which to search
   "Position"
FROM dual;
```

Position
2

ORACLE

在使用正则表达式支持功能时使用子表达式

REGEXP_INSTR 和 REGEXP_SUBSTR 有一个可选的 SUBEXPR 参数；使用该参数时可将计算的正则表达式中的特定子字符串作为目标。

在幻灯片示例中，可以搜索子表达式列表中的第一个子表达式模式。显示示例中标识出了 REGEXP_INSTR 函数的若干个参数。

1. 确定要搜索的字符串。
2. 确定子表达式。第一个子表达式为 123，第二个子表达式为 45678，第三个为 56，第四个为 78。
3. 第三个参数确定从哪个位置开始搜索。
4. 第四个参数确定出现的要查找的模式。1 表示查找第一个出现的模式。
5. 第五个参数是返回选项。这是出现第一个字符的位置。（如果指定 1，则返回出现字符之后那个字符的位置。）
6. 第六个参数确定搜索是否区分大小写。
7. 最后一个参数是 Oracle Database 11g 中新增的参数。此参数指定要查找哪个子表达式。在显示示例中，要搜索的是第一个子表达式，即 123。

为什么要访问第 *n* 个子表达式

- 一个更实际的用途：DNA 测序
- 可能需要查找某个特定子模式来确定老鼠 DNA 中免疫能力所需要的某种蛋白质

```
SELECT
    REGEXP_INSTR('ccacctttccctccactcctcacgttctcacctgtaaagcgtccctc
    cctcatccccatgccccttaccctgcagggtagagtaggctagaaaccagagagctccaagc
    tccatctgtggagaggtgccatccttgggctgcagagagaggagaatttgccccaaagctgcc
    tgcagagcttcaccacccttagtctcacaaagccttgagttcatagcatttcttgagttttca
    ccctgcccagcaggacactgcagcacccaaagggcttcccaggagtagggttgccctcaagag
    gctcttgggtctgatggccacatcctggaattgttttcaagttgatggtcacagccctgaggc
    atgtaggggcgtggggatgcgctctgctctgctctcctctcctgaacccctgaaccctctggc
    taccccagagcacttagagccag',
                    '(gtc(tcac)(aaag))',
                     1, 1, 0, 'i',
                     1) "Position"
FROM dual;
```

	Position
1	195

ORACLE

为什么要访问第 *n* 个子表达式

在生命科学中，可能需要从 DNA 序列中提取子表达式匹配项的偏移量做进一步处理。例如，可能需要查找一个特定蛋白质序列，例如，前面为 gtc、后跟 tcac 接着是 aaag 的 DNA 序列的开始偏移量。要达到此目标，可以使用 REGEXP_INSTR 函数；该函数可返回找到匹配项的位置。

在幻灯片示例中，返回了第一个子表达式 (gtc) 的位置。gtc 在 DNA 字符串的第 195 位置开始出现。

如果通过修改幻灯片示例来搜索第二个子表达式 (tcac)，则查询结果的输出如下。tcac 在 DNA 字符串的第 198 位置开始出现。

	Position
1	198

如果通过修改幻灯片示例来搜索第三个子表达式 (aaag)，则查询结果的输出如下。aaag 在 DNA 字符串的第 202 位置开始出现。

	Position
1	202

REGEXP_SUBSTR：示例

```
SELECT
  REGEXP_SUBSTR
①('acgctgcactgca', -- source char or search value
②  'acg(.*)gca',    -- regular expression pattern
③  1,               -- position to start searching
④  1,               -- occurrence
⑤  'i',            -- match option (case insensitive)
⑥  1)              -- sub-expression
 "Value"
FROM dual;
```

Value
1 ctgcact

ORACLE

REGEXP_SUBSTR：示例

在幻灯片显示示例中：

1. `acgctgcactgca` 为要搜索的源字符串。
2. `acg(.*)gca` 为要搜索的模式。查找 `acg` 且后跟 `gca`，在 `acg` 和 `gca` 之间可能有一些字符。
3. 在源字符串的第一个字符处开始搜索。
4. 搜索第一个出现的模式。
5. 对源字符串使用不区分大小写的匹配。
6. 使用一个非负整数值确定作为目标的第 *n* 个子表达式。这是子表达式参数。在此示例中，`1` 表示第一个子表达式。可以使用 0-9 之间的一个值，0 表示没有指定目标子表达式。此参数的默认值为 0。

课程安排

- 正则表达式简介
- 在正则表达式中使用元字符
- 使用正则表达式函数：
  - REGEXP_LIKE
  - REGEXP_REPLACE
  - REGEXP_INSTR
  - REGEXP_SUBSTR
- 访问子表达式
- 使用 REGEXP_COUNT 函数

使用 REGEXP_COUNT 函数

```
REGEXP_COUNT (source_char, pattern [, position
                [, occurrence [, match_option]]])
```

```
SELECT REGEXP_COUNT(
    'ccacctttccctccactcctcacgttctcacctgtaaagcgtccctccctcatccccatgccccttaccctgcag
    ggtagagtaggctagaaaccagagagctccaagctccatctgtggagaggtgccatccttgggctgcagagagaggag
    aatttgccccaaagctgcctgcagagcttcaccacccttagtctcacaaagccttgagttcatagcatttcttgagtt
    ttcaccctgcccagcaggacactgcagcacccaaagggcttcccaggagtagggttgccctcaagaggctcttgggtc
    tgatggccacatcctggaattgttttcaagttgatggtcacagccctgaggcatgtaggggcgtggggatgcgctctg
    ctctgctctcctctcctgaacccctgaacccctctggctaccccagagcacttagagccag',
    'gtc') AS Count
FROM dual;
```

	COUNT
1	4

ORACLE

使用 REGEXP_COUNT 函数

REGEXP_COUNT 函数使用由输入字符集定义的字符来计算字符串的值。该函数返回一个整数，指出模式出现的次数。如果未找到任何匹配项，则返回 0。

在幻灯片示例中，使用 REGEXP_COUNT 函数确定了一个 DNA 子字符串出现的次数。

以下示例显示模式 123 在字符串 123123123123 中出现的次数为 3。该搜索从字符串的第二个位置开始。

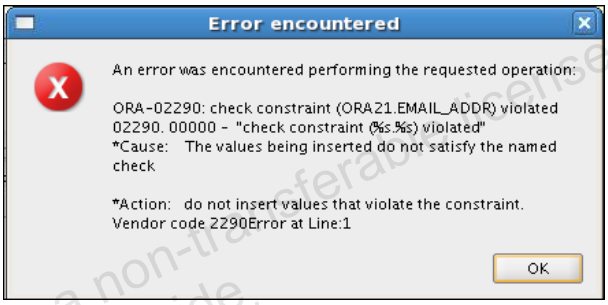
```
SELECT REGEXP_COUNT
   ('123123123123', -- source char or search value
   '123',           -- regular expression pattern
    2,              -- position where the search should start
   'i')             -- match option (case insensitive)
   As Count
FROM dual;
```

	COUNT
1	3

正则表达式和检查约束条件：示例

```
ALTER TABLE emp8
  ADD CONSTRAINT email_addr
  CHECK(REGEXP_LIKE(email,'@')) NOVALIDATE;
```

```
INSERT INTO emp8 VALUES
  (500,'Christian','Patel','ChrisP2creme.com',
   1234567890,'12-Jan-2004','HR_REP',2000,null,102,40);
```



ORACLE

正则表达式和检查约束条件：示例

正则表达式还可以用在 CHECK 约束条件中。在本示例中，在 EMPLOYEES 表的 EMAIL 列上添加了一个 CHECK 约束条件。用于确保只接受包含“@”符号的字符串。然后对约束条件进行测试。因为电子邮件地址不包含所需符号，所以违反了 CHECK 约束条件。NOVALIDATE 子句用于确保不对现有数据进行检查。

在幻灯片示例中，emp8 表是使用以下代码创建的：

```
CREATE TABLE emp8 AS SELECT * FROM employees;
```

注：幻灯片示例是在 SQL Developer 中使用“Execute Statement（执行语句）”选项执行的。如果使用“Run Script（运行脚本）”选项，则输出格式有所不同。

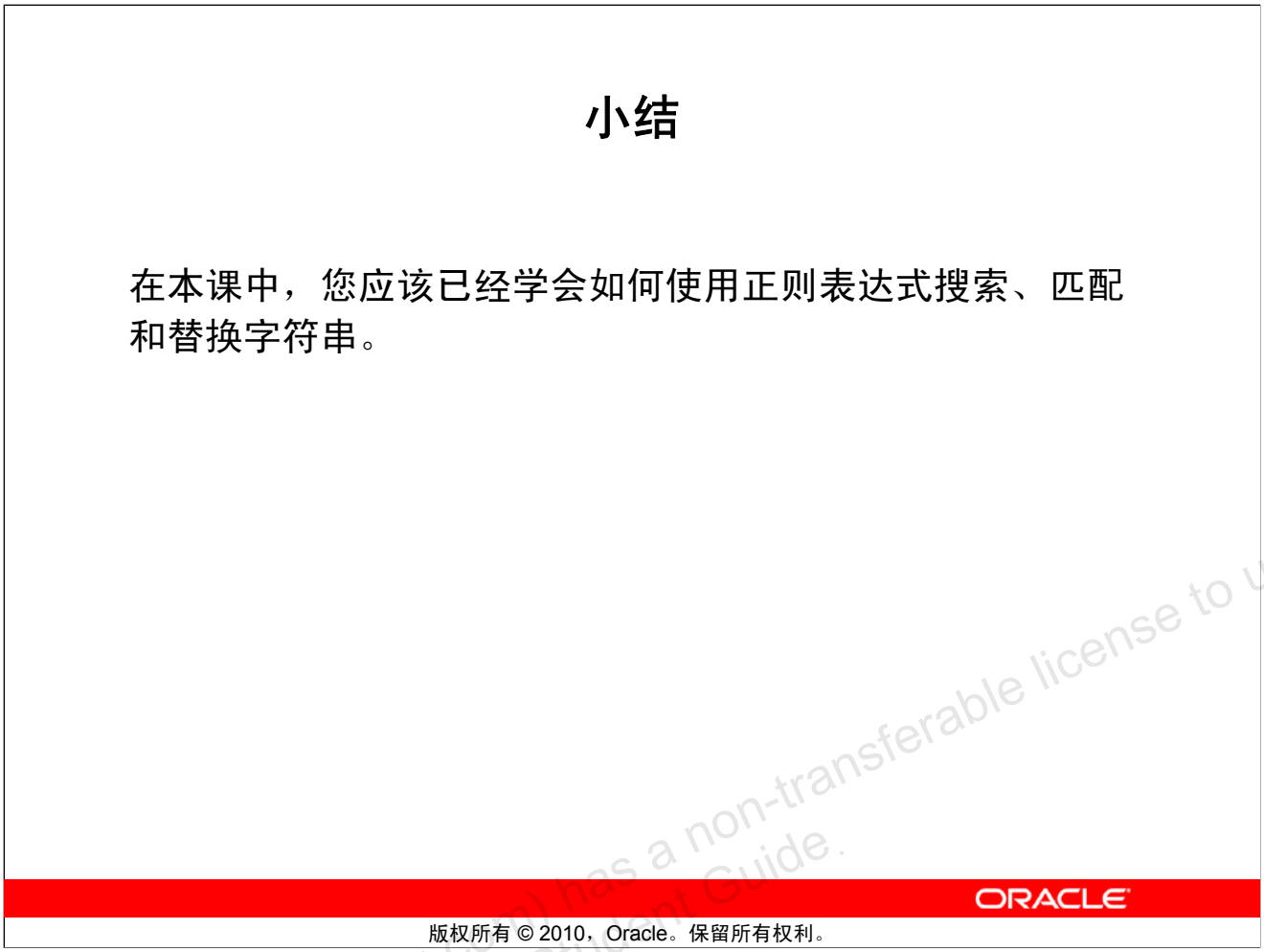
小测验

通过在 SQL 和 PL/SQL 中使用正则表达式，您可以：

1. 避免中间层应用程序对 SQL 结果集进行大量的字符串处理
2. 避免在客户机上出现数据验证逻辑
3. 在服务器上强制实行约束条件

ORACLE

答案：1、2、3



小结

在本课中，您已经学习使用正则表达式支持功能。在 SQL 和 PL/SQL 中都提供了正则表达式支持功能。

练习 7：概览

此练习包括使用正则表达式函数执行以下操作：

- 搜索、替换和处理数据
- 创建一个新的 CONTACTS 表，并在 p_number 列上添加一个 CHECK 约束条件，以确保在数据库中输入特定标准格式的电话号码
- 测试在 p_number 列中添加各种格式的电话号码

练习 7：概览

在本练习中，您将使用正则表达式函数搜索、替换和处理数据。还将创建一个新的 CONTACTS 表，并在 p_number 列上添加一个 CHECK 约束条件来确保在数据库中输入特定标准格式的电话号码。