Teradata SQL基础教程

第一章　关系数据库基础

1.1 关系数据库模型

关系数据库理论最早是由Codd博士提出的，一个关系的数学描述其实就是一个二维表，这些二维表按照业务运行的规律组合起来，就是关系数据库模型。这种模型可以简洁地表达出企业或机构的业务运作规律，抓住事物本质，因此非常实用。

每个二维表被称为一个实体(Entity)，它可以是人、地点或者某种事物等。表中的每个列被称为属性(Attribute)或者字段(Field)，表中的每一行代表了该实体的一个特定实例，称为记录(Record)。表1-1、1-2和1-3分别给出了一个雇员表、部门表和工作表的实例。

表1-1 雇员表(Employee Table)

EMPLOYEE NUMBER	MANAGER EMPLOYEE NUMBER	DEPARTMENT NUMBER	JOB CODE	LAST NAME	FIRST NAME	HIRE DATE	BIRTH DATE	SALARY AMOUNT
PK	FK	FK	FK					
1018	1017	501	512101	Ratzlaff	Larry	1978-07-15	1954-05-31	54000.00
1022	1003	401	412102	Machado	Albert	1979-03-01	1957-07-14	32300.00
1014	1011	402	422101	Crane	Robert	1978-01-15	1960-07-04	24500.00
1003	801	401	411100	Trader	James	1976-07-31	1947-06-19	37850.00
1007	1005	403	432101	Villegas	Arnando	1977-01-02	1937-01-31	49700.00
1010	1003	401	412101	Rogers	Frank	1977-03-01	1935-04-23	46000.00

表1-2 部门表(Department Table)

department_number	department_name	budget_amount	manager_employee_number
PK			FK
402	software support	308000.00	1011
401	customer support	982300.00	1003
201	technical operations	293800.00	1025
100	president	400000.00	801
501	marketing sales	308000.00	1017
403	education	932000.00	1005

表1-3 工作表(Job Table)

job_code	description	hourly_billing_rate	hourly_cost_rate
PK			
421100	Manager - Software Support	0.00	0.00
512101	Sales Rep	0.00	0.00
511100	Manager - Marketing Sales	0.00	0.00
312101	Software Engineer	0.00	0.00
411100	Manager - Customer Support	0.00	0.00
431100	Manager - Education	0.00	0.00
413201	Dispatcher	0.00	0.00
432101	Instructor	0.00	0.00
422101	Software Analyst	0.00	0.00
321100	Manager - Product Planning	0.00	0.00

在一个关系数据库模型中，表和表之间是有关联的，这种关联常用所谓的E-R图(Entity-Relationship Diagram)来表示。上面三个表的E-R图可以表示为：

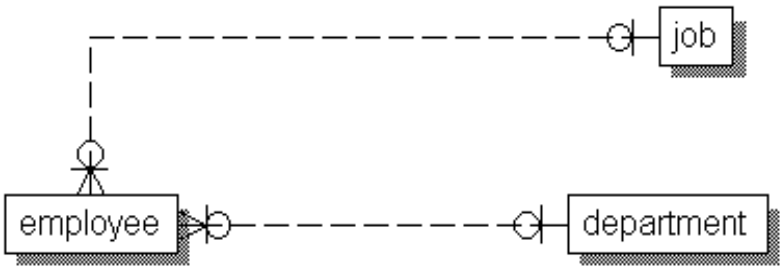


图1-1 实体关系图(E-R图)

实体与实体之间的关系有以下三种：

- **一对多**

指一个实体中有且只有一条记录与另一个实体中的多条记录对应。如图1-1中，一个工作岗位可以有多个雇员，一个部门也可以有多个雇员。我们一般把“一”方称作父表，而把“多”方称作子表。图中的符号是一种比较常见的表示一对多关系的方法，另外一种也经常使用的表示方法如图1-2所示：

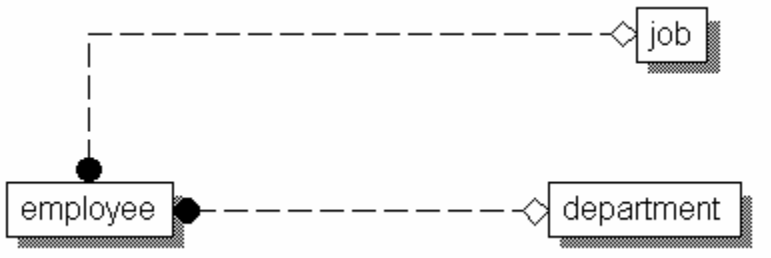


图1-2 实体关系图的另一种表示方法(E-R图)

- **一对一**

指一个实体中的记录与另一个实体的记录处于一对一的关系。

- **多对多**

指一个实体中的记录与另一个实体的记录是多对多的关系。比如雇员与地址之间的关系就可能是一种多对多关系，因为一个雇员可能有多个地址，而一个地址也可能对应多个雇员。对于这种关系，在进行数据库逻辑模型设计时，一般要进行分解处理，即分解成两个一对多的关系。

1.2 键的定义

在一个实体中，存在一些能唯一标识该实体中各个记录的属性(或属性组合)，这些属性被称作键(Key)。真正被挑选出来唯一区分各记录的属性称为主键(Primary Key，缩写为PK)，其它未被选中的键称为候选键(Alternate Key，缩写为AK)。例如在雇员表中，一般可以把雇员编号选作主键。

一个实体中的哪些属性可以是主键，这与具体的系统和业务规则有关。举例来说，如果一个公司内没有重名的雇员，则将其姓名当作主键也未尝不可。一般来讲，主键属性必须满足以下三个条件：

- 必须能唯一区分各数据记录，即不能有重复值
- 不可以是空值
- 其值很少发生变化

前面两个条件很容易理解，对第三个条件可以举个例子来说明。如在雇员表中，使用雇员编号作为主键。假设某个雇员进公司时编号为1000，几年后他升为主管，将其编号改为2000。这样就可能产生这样的问题：当我们寻找编号1000的雇员信息时，只能发现他以前的记录，而找不到他的当前情况；同样，寻找编号2000的雇员时，只有现在的信息而没有以前的记录。因此，最好的办法是不要改动雇员编号，或者选用其他候选键作为主键。

在前面的E-R图中，只表示了实体的名字而没有包含实体的属性。如果把实体属性包含进来，则一般用图1-3的形式：

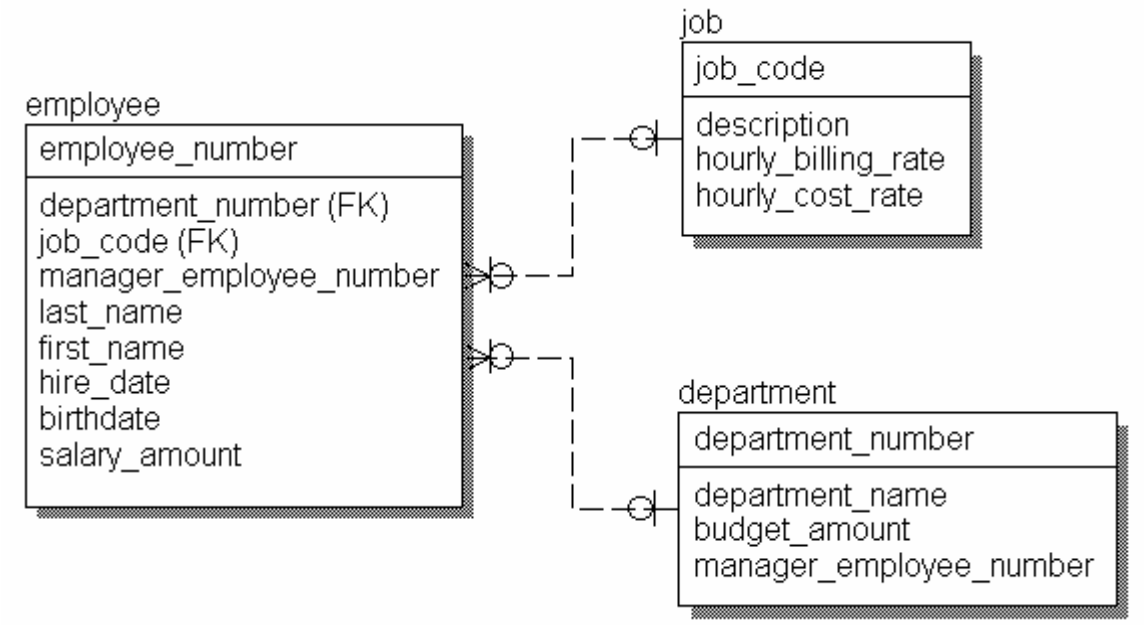


图1-3 带属性定义的实体关系图(E-R图)

我们看到，在这种E-R图中，一般把实体的各属性分别放在上下两个框中，上面是主键，下面则是其它属性。

由于实体与实体之间存在的关系，引出了另外一个概念：外键(Foreign Key，缩写为FK)。所谓外键，是指一个实体(子表)的一个属性，而这个属性正好是另一个实体(父表)的主键。如图1-3中的雇员表，部门编号(Department_Number)与工作编号(Job_Code)分别是部门表和工作表的主键，因此被称为外键。很显然，外键只有透过实体之间的关系才能体现出来。利用主键与外键，可以保证各个实体之间的参照完整性。

1.3 范式理论

为了描述一个企业或机构的业务运行方式，我们要建立相应的E-R图。在选择实体和实体的属性、定义它们之间的关系时，是不是可以随意而为呢？显然不是。一个设计不好的关系模型可能会产生很多的问题，如过多的冗余、数据不一致、插入与删除异常等。

为了能正确地描述和处理这些问题，在关系数据库理论的发展过程当中，逐渐形成并完善了所谓的范式理论，它是数据库逻辑模型设计的基本理论。首先来看几个基本概念。

第一范式(The First Normal Form)

在一个关系中，如果每个属性的值唯一而不具有多义性，则称它符合第一范式。

举例来说，在一个描述雇员的关系中，如果用一个属性来记录某个雇员加入或者离开公司的日期，显然就不符合这个条件。

第二范式(The Second Normal Form)

如果一个关系符合第一范式，并且每个非主属性完全依赖于整个主键，而非主键的一部分，则称它符合第二范式。

当多个属性组成复合主键时，要特别注意这个条件。

第三范式(The Third Normal Form)

如果一个关系符合第二范式，并且每个非主属性不能依赖于其它关系中的属性(因为这样的话，这种属性应该归到其它关系中去)，则称它符合第三范式。

除了上面介绍的三个范式外，还有更高阶的第四、第五范式。

一个关系模型可以从第一范式到第五范式进行无损分解，这个过程也称为规范化(Normalize)。在数据库的逻辑模型设计中一般采用第三范式就已经足够了。

我们可以看到，第三范式的定义基本上是围绕主键与非主属性之间的关系而作出的。

1.4 逻辑模型与物理模型

逻辑模型主要是依据业务规则建立的，为了清晰地反映企业操作模式，我们建立的逻辑模型一般应满足第三范式。而把一个逻辑模型在一个平台上实施时，就变成了物理模型。由于技术上的一些限制，在一些系统中常常不得不对逻辑模型作不规范化处理(De-Normalize)。

举例来说，在数据仓库中经常要用到多表的连接操作，而表的连接是比较耗系统资源的。为了避免或减少表的连接操作，一些数据库系统常常不得不把两个或多个表合并起来，这实际上就是一种不规范化处理。

图1-3表示的是一个逻辑模型，它定义了主键、外键以及其它属性，至于属性的数据类型等则反映不出来。图1-4是对应的物理模型，每个属性的数据类型都有详细的定义。

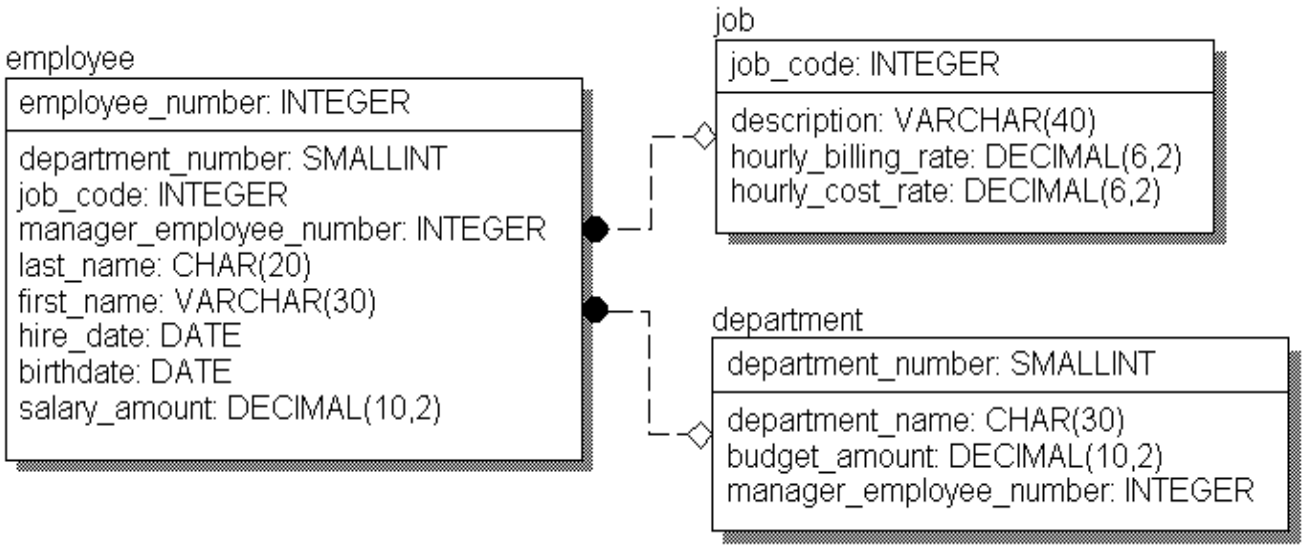


图1-4 物理模型

1.5 索引

索引是物理模型中的一个概念，利用索引，可以直接存取表中的某一条记录而不需要搜索整个表。因此，索引提供了一条更快速访问数据记录的途径。

当在数据库中针对某个表创建一个索引时，系统将根据此索引建立一个相应的子表。相对原来的表(主表)而言，子表要小得多。它将存储索引的值以及一个与此索引对应的数据记录在主表中的存储位置，这好比一个指向数据记录物理位置的指针，如图1-5所示。因此，当搜索一条数据记录时，如果有相应的索引，则只要在索引子表中找到与此索引值匹配的记录，就找到了数据项在主表中的物理存储位置，这样不需要搜索整个主表就能迅速定位数据记录了。显然，建立索引需要占用额外的存储空间；另外，索引子表是由系统自动维护的，当主表的数据记录发生变化时，系统要自动更新索引子表的相应记录，从而占用系统资源。这就是使用索引的代价。

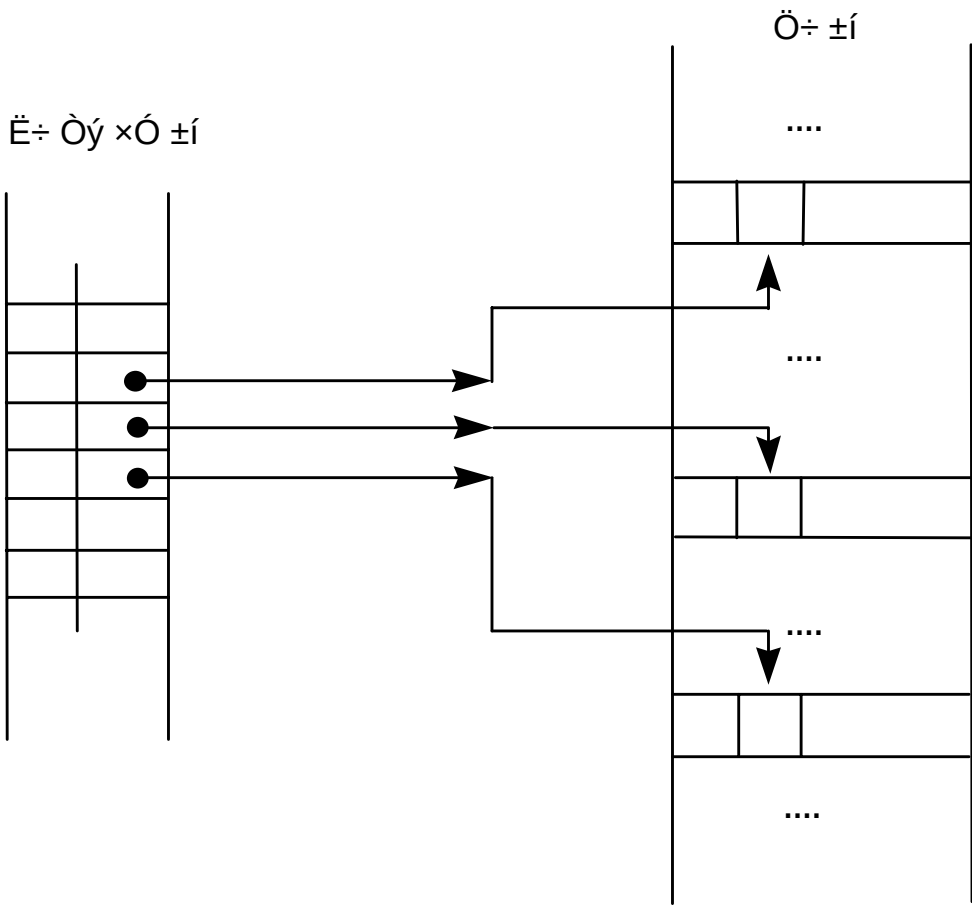


图1-5 索引子表与主表

在Teradata数据库管理系统中，它使用一个所谓的主索引进行数据的分配，而主索引是没有子表的，不占用额外的存储空间，也不需要专门的维护。因此，Teradata中的索引有主次之分，只有次索引才有相应的子表。有关Teradata中主次索引的用法和原理在第四和第五章中会有详细的讨论，这里只介绍一下它们的种类和定义。

我们已经谈到，主键与外键是数据库逻辑模型中的定义，而索引则存在于物理模型中。Teradata中又把索引分为主索引和次索引，主索引不一定就对应主键。选择主索引的基本原则是：尽量选择那些访问频率高的属性作为主索引。举例来

说，在雇员表中，如果经常根据姓名来查找数据，则应选择姓名而不是雇员编号作为主索引。

主索引与主键的区别可以总结为表1-4。

表1-4 主键与主索引的区别

主键	**主索引**
逻辑模型中的一个概念	物理模型中用于数据分配与存取的一种物理机制
没有属性数目的限制	最多由16个属性组成
在逻辑模型中定义	在创建表时定义
取值必须唯一	可以唯一也可以不唯一
用来区分数据记录	用来进行数据分配
其值不发生改变	其值可以发生变化
不可以取空值	可以取空值

从表中看到，主索引的取值可以是唯一的，也可以是不唯一的。唯一的主索引简称为UPI(Unique Primary Index)，不唯一的主索引简称为NUPI(Non-Unique Primary Index)。同样，次索引也有唯一与不唯一之分，前者简称为USI(Unique Secondary Index)，后者简称为NUSI(Non-Unique Secondary Index)。

1.6 关系数据库操作语言

对关系数据库进行操作的标准语言是所谓的结构化查询语言SQL(Structure Query Language)，和其它的程序设计语言不一样的是，SQL是一种非过程语言。

SQL采用一种类似自然英语的结构，因此非常容易理解和学习。目前SQL已经有了ANSI标准，各个数据库厂商除了支持ANSI标准的SQL语法外，一般都作了不同程度的扩展，以加强其功能。如果在系统开发时过多地使用非ANSI标准的扩展功能，将会增加系统在不同平台下移植时的困难程度。

根据各种SQL语句的功能不同，可以把它们分成以下几类：

1.6.1 数据定义语言

数据定义语言(Data Definition Language)简称为DDL，可用来在系统中创建或者修改各种对象的结构，也可用来删除系统中已有的对象。其命令集主要包括：

- CREATE - 定义一个新的数据库、用户、数据库对象或索引
- DROP - 删除一个已存在的数据库、用户、数据库对象或索引
- ALTER - 修改一个表的结构和保护定义，启动或禁止触发器

这些命令所操作的对象可以是数据库(Database)、用户(User)、表(Table)、视图(View)、宏(Macro)、触发器(Trigger)和存储过程(Stored Procedure)。

1.6.2 数据操作语言

数据操作语言(Data Manipulation Language)简称为DML，当使用DDL定义好数据库中的对象后，可以使用DML来存取对象中的信息，因此，DML是SQL中使用最频繁的语言。DML的命令集主要包括：

- SELECT - 执行关系查询
- INSERT - 在表中增加新行
- UPDATE - 修改表中已存在的行的值
- DELETE - 删除表中已存在的行

1.6.3 数据控制语言

数据控制语言(Data Control Language)简称为DCL，用来控制用户存取数据库的权限或者方式。它的命令集主要包括：

- GRANT - 赋给用户权限
- REVOKE - 去除用户的权限
- GIVE - 转让数据库所有权

1.6.4 其它

上述三种语言是SQL中的标准语言，许多数据库厂商都在此基础上作了扩充，以补充或加强数据库管理系统的功能。在Teradata中，主要增加了以下命令：

命令	功能
HELP	帮助用户了解数据库中各种对象的结构
SHOW	帮助用户了解某种对象的定义，即返回其DDL语句
EXPLAIN	返回一个SQL语句经优化处理后的执行步骤，注意并未真正执行
FALLBACK	对数据加以保护的一种方式，是冗余的备份
RENAME	对表重命名
NULLIFZERO	对数据作累计处理时，忽略零值
ZEROIFNULL	对数据作累计处理时，将空值作零处理
WITH...BY	对详细数据记录作分类统计(Sub-Total)时有用
MODIFY USER /DATABASE	对用户/数据库对象作动态修改而无需数据库重组

1.7 数据字典/目录

数据字典/目录(Data Dictionary/Directory)简称为DD/D，其中存储了数据库管理系统中所有的系统对象(表、视图与宏)，从中可以得到有关用户、数据库、系统资源的使用情况、数据分布情况、安全规则、访问权限等各种信息。

在Teradata中，这些信息均存储在一个称为DBC的用户下面。

第二章 TeradataSQL基础

结构化查询语言SQL是一种接近英文自然表达的查询语言，和其它程序设计语言不一样的是，它是非过程的。SQL的语法非常容易掌握，功能却很强大，目前已

成为所有关系数据库的标准查询语言。Teradata支持标准的ANSISQL，并在此基础

2.1 Teradata中支持的数据类型

在Teradata环境中，数据是以8比特ASCII码形式存放的。当客户端采用EBCDIC码时，Teradata能自动作数据的转换，因此对客户端没有任何影响。它所支持的数据类型如表2-1所示：

表2-1 Teradata所支持的数据类型

类型	名字	字节长度	说明
数值型	DATE	4	YYMMDD，特殊的INTEGER类型
	DECIMAL(n,m)	1,2,4,8	n: 1~18, m: 0~n，可缩写成DEC。 如256.78可表示为DEC(5,2)
	NUMERIC(n,m)	1,2,4,8	同上
	BYTEINT	1	128～＋127
	SMALLINT	2	32768～＋32767
	INTEGER	4	可缩写成INT
	REAL	8	$2 * 10^{-307} \sim 2 * 10^{+308}$
	FLOAT	8	同上
	DOUBLE PRECISION	8	同上
字符型	CHAR(n)	n	n: 1~64000，也可写成CHARACTER
	VARCHAR(n)	n+2	n: 1~64000，变长，最前面两字节表示字符串长度。也可写成CHAR VARYING(n)
	LONG VARCHAR	32002	相当于VARCHAR(64000)
二进制	BYTE(n)	n	n: 1~64000，无符号二进制整数
	VARBYTE(n)	n+2	n: 1~64000，变长，最前面两字节表示长度
其它	GRAPHIC(n)	n	n: 1~32000，这三种数据类型可支持双字节日

			本字符和汉字
	VARGRAPHIC	n+2	n: 1~32000，变长，最前面两字节表示长度
	LONG VARGRAPHIC		相当于VARGRAPHIC(32000)

这里需要说明的是，对于GRAPHIC型数据，主要是用来支持双字节日本字符和汉字的。利用这种特性，可以使数据库中各对象的名字都采用双字节字符，如将某个表的各列取名为汉字。如果只是单纯为了存储和处理汉字信息、二进制数据文件等，采用BYTE或CHAR类型都可以。因此，一般来说在安装Teradata数据库时都使用所谓的国际通用(International)型选项。在这种情况下，Teradata可以存储各种汉字或图形信息，但数据库各对象的名字不能使用汉字，而应采用标准的英文字母。

2.2 Teradata中SQL命令分类

和标准ANSI-SQL一样，Teradata的SQL命令也分成以下几类：

1. 数据定义语言Data Definition Language (DDL)

SQL语句	功能
CREATE	定义新的表、视图、宏、索引、触发器和存储过程
DROP	删除表、视图、宏、索引、触发器和存储过程
ALTER	表结构与保护机制的调整

2. 数据操作语言Data Manipulation Language (DML)

SQL语句	功能
SELECT	执行关系查询操作(选择、投影、连接、合并、交集等)
INSERT	向表中插入一条新记录
UPDATE	修改表中记录的值
DELETE	删除表中指定的记录

3.数据控制语言Data Control Language (DCL)

SQL语句	功能
GRANT	给用户授予某种权限
REVOKE	删除用户某种权限
GIVE	转移用户所有权关系

基本上说来，SQL是一种基于集合进行操作的语言，它是非过程化的。举例来说，要从雇员表中选取所有在401号部门工作的员工，列出他们的姓名、加入公司日期及薪资情况，可以使用下面的SQL语句：

```
SELECT  employee_number
        ,hire_date
        ,last_name
        ,first_name
FROM    employee
WHERE  department_number = 401
;
```

注意最后以分号结束，这表示一个SQL交易的结束。

2.3 Teradata中会话层的建立

当用户从前端访问Teradata时，首先要建立连接，即所谓的会话层(Session)，这可以通过两种方式来建立。一种是标准的ODBC连接，另一种是调用层接口CLI (Call Level Interface)。对于ODBC，这是目前数据库行业所遵循的一个事实上的标准，绝大多数关系数据库系统和前端工具都支持它。而CLI是NCR公司自行开发的一个数据库接口，和其它数据库的专用接口(如Oracle的SQL*Net)一样，CLI是Teradata专用的一个编程接口，不能用于其它数据库。

CLI的速度比ODBC快许多，Teradata中很多工具都是基于CLI开发的，如最常用的查询工具BTEQ(详细说明参见附录一)、数据加载工具FastLoad和MultiLoad、数据备份与恢复工具ASF2、数据输出工具FastExport等。

Teradata支持标准的ANSI SQL语法，同时也在此基础上作了适当的扩充和加强。当通过CLI与系统建立对话时，有一些参数会影响Teradata处理交易的特征和行为。下面简单地作一个描述。

利用BTEQ (假设是UNIX系统)登录到Teradata后，系统将提示下面的信息：

Teradata BTEQ 04.01.00.00 for UNIX5. Enter your logon or BTEQ command:

此时可以通过下面的命令来观察有哪些会话层参数：

.show control;

系统将返回类似下面的信息：

Default Maximum Byte Count = 4096

Default Multiple Maximum Byte Count = 2048

Current Response Byte Count = 4096

Maximum number of sessions = 20

Maximum number of the request size = 32000

.

.

. .

[SET] ECHOREQ = ON

[SET] ERRORLEVEL = ON

[SET] FOLDLINE = OFF ALL

[SET] FOOTING = NULL

.

.

[SET] RETCANCEL = OFF

[SET] RETLIMIT = No Limit

[SET] RETRY = ON

[SET] RTITLE = NULL

[SET] SECURITY = NONE

[SET] SEPARATOR = two blanks

[SET] SESSION CHARSET = ASCII

[SET] SESSION SQLFLAG = ENTRY

[SET] SESSION TRANSACTION = ANSI

[SET] SESSIONS = 1

[SET] SIDETITLES = OFF for the normal report.

And, it is ON for results of WITH clause number: 1 2 3 4 5 6 7 8 9.

[SET] SKIPDOUBLE = OFF ALL

[SET] SKIPLINE = OFF ALL

[SET] SUPPRESS = OFF ALL

[SET] TDP = l5442

[SET] TITLEDASHES = ON for the normal report.
And, it is ON for results of WITH clause number: 1 2 3 4 5 6 7 8 9.
[SET] UNDERLINE = OFF ALL
[SET] WIDTH = 75

其中前面标有SET的都是可以通过SET命令进行设置的参数。与Teradata会话有关的命令是以SET SESSION开头的几个，即SET SESSION CHARSET、SET SESSION TRANSACTION、SET SESSION SQLFLAG和SET SESSIONS。这里主要讨论中间的两种。

Teradata处理SQL交易请求时有两种方式，即所谓的BTET和ANSI，前者是Teradata缺省的处理方式，后者则是针对ANSI的。在这两种模式下，Teradata对个别交易的处理会表现出不同的结果。对于这种情况，本书后面的各章节中将结合具体的例子加以说明。

改变交易处理模式的命令如下：

.SET SESSION TRANSACTION [ANSI|BTET]

前面已经介绍过，Teradata对SQL语法在ANSI的基础上作了一部分的扩展。根据ANSI组织的规定，在处理SQL交易请求时，如果该交易不符合标准的ANSI SQL语法规定，数据库系统必须提供一种机制来显示或者加以说明。在Teradata中，这是通过一个标志参数SQLFLAG来实现的。

.SET SESSION SQLFLAG [NONE|ENTRY|INTERMEDIATE]

系统的缺省设置是NONE，这样，当一个SQL交易请求不符合ANSI SQL时，系统不作任何提示。如果设置成ENTRY，当一个SQL交易请求不符合ANSI SQL的ENTRY级时，Teradata将给出警告信息。同理，如果设置成INTERMEDIATE，则当SQL语法不符合ANSI SQL的INTERMEDIATE级时，系统也将给出警告信息。

举例来说，如果SQLFLAG设置成ENTRY，执行下面的命令后，

```
select date;
```

系统将给出下面的响应：

```
select date;
$
*** SQL Warning 5836 Token is not an entry level ANSI Identifier or Keyword.
select date;
$
*** SQL Warning 5821 Built-in values DATE and TIME are not ANSI.
select date;
$
*** SQL Warning 5804 A FROM clause is required in ANSI Query Specification.
Date
--------
99/03/26
```

这是由于DATE不是ENTRY级ANSI SQL标准的函数和关键词才产生的警告信息，另外，在ENTRY级ANSI SQL中也要求有FROM子句。

在Teradata中，对许多关键词都支持缩写。如SELECT可以缩写为SEL，这也不符合ENTRY级ANSI SQL的语法规定。如果将上面的命令改成：

```
sel date;
```

则系统返回下面的信息：

```
sel date;
$
*** SQL Warning 5836 Token is not an entry level ANSI Identifier or Keyword.
sel date;
$
*** SQL Warning 5818 Synonyms for Operators or Keywords are not ANSI.
sel date;
$
*** SQL Warning 5821 Built-in values DATE and TIME are not ANSI.
sel date;
$
*** SQL Warning 5804 A FROM clause is required in ANSI Query Specification.
Date
--------
99/03/26
```

其中第二条警告信息表示SEL不是ENTRY级ANSI SQL所支持的操作符。

需要注意的是，这两个SET命令都必须在用户登录(指建立连接)前执行才有效果，事实上，所有SET SESSION的命令都是如此，其它SET命令则应在登录系统后再执行。以BTEQ来说，当系统出现提示符后，即可执行SET命令来设置这两个会话层参数。

第三章　数据库试验环境

为了更好地掌握Teradata SQL，本书尽可能多地通过实例来说明SQL命令及其操作。另外，每章后面还附有一定的练习题，以便加深和巩固所学知识。所有这些都是基于一个试验环境来进行的。

这是一个模仿客户服务部门的数据库，其逻辑模型如图3-1所示：

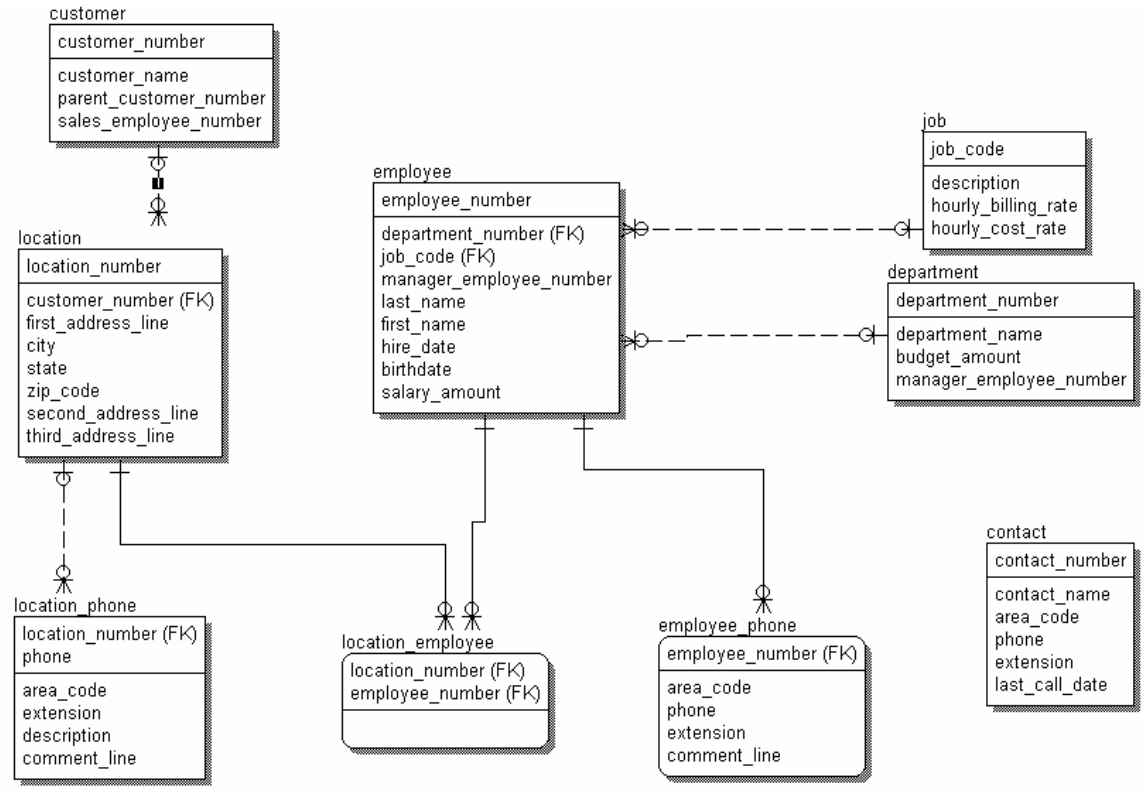


图3-1 数据库试验环境逻辑模型

从图中来看，可以用9个表来模拟该客户服务部门的运行。这9个表分别是：

Customer:	客户信息表
Location:	位置信息表
Employee:	雇员信息表
Job:	工作信息表
Department:	部门表
Employee_Phone:	员工电话表
Location_Employee:	员工与位置的对照表
Location_Phone:	位置与电话的对照表
Contact:	联系表

系统的物理结构如图3-2所示。图中PK表示Primary Key，即主键，标有PK的字段为主键字段。FK表示Foreign Key，即外键，标有FK的字段为外键字段。

SQL Lab Customer Service Database

CUSTOMER

CUSTOMER NUMBER	CUSTOMER NAME	PARENT CUSTOMER NUMBER	SALES EMPLOYEE NUMBER
PK		**FK**	**FK**
8	Colby Co.	?	1018

DEPARTMENT

DEPARTMENT NUMBER	DEPARTMENT NAME	BUDGET AMOUNT	MANAGER EMPLOYEE NUMBER
PK			**FK**
403	education	93200000	1005

CONTACT

CONTACT NUMBER	CONTACT NAME	AREA CODE	PHONE	EXTENSION	CALL DATE
PK					
8005	Hughes, Jack	212	5432162	710	87/08/05

EMPLOYEE

EMPLOYEE NUMBER	MANAGER EMPLOYEE NUMBER	DEPARTMENT NUMBER	JOB CODE	LAST NAME	FIRST NAME	HIRE DATE	BIRTH DATE	SALARY AMOUNT
PK	**FK**	**FK**	**FK**					
1005	801	403	431100	Ryan	Loretta	761015	550910	3120000

EMPLOYEE PHONE

EMPLOYEE NUMBER	AREA CODE	PHONE	EXTENSION	COMMENT LINE
PK				
FK				
1007	213	2274764	?	?

JOB

JOB CODE	DESCRIPTION	HOURLY BILLING RATE	HOURLY COST RATE
PK			
432101	Instructor	?	?

LOCATION

LOCATION NUMBER	CUSTOMER NUMBER	FIRST ADDRESS LINE	CITY	STATE	ZIP CODE	SECOND ADDRESS LINE	THIRD ADDRESS LINE
PK	**FK**						
14000012	12	510 Benton Av	Chicago	Illinois	606483930	?	?

LOCATION EMPLOYEE

LOCATION NUMBER	EMPLOYEE NUMBER
PK	
FK	**FK**
9000005	1010

LOCATION PHONE

LOCATION NUMBER	AREA CODE	PHONE	EXTENSION	DESCRIPTION	COMMENT LINE
PK					
FK					
9000005	202	3239119	?	Switchboard	

图3-2 数据库试验环境物理模型

试验数据库的层次结构如图3-3所示：

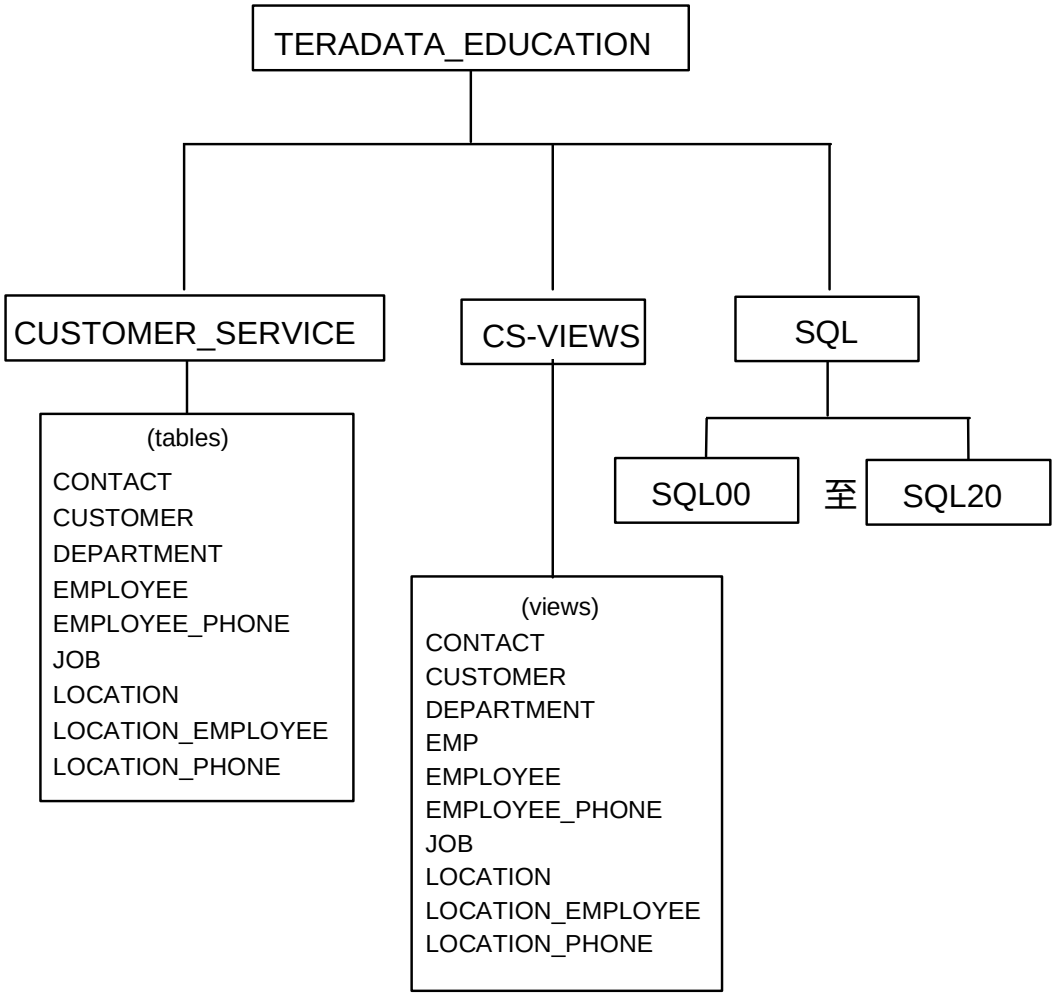


图3-3 数据库试验环境层次结构

从图中来看，整个试验数据库的名称是TERADATA_EDUCATION，其下分成以下三个数据库：

CUSTOMER_SERVICE数据库：包含所有的物理表

CS_VIEWS数据库：包含所有的视图

SQL用户：下面进一步分成SQL00至SQL20共21个数据库用户，供学员登录系统进行测试使用。每个用户登录的密码与其用户名相同，如用SQL01登录，其密码也为SQL01。

一般的SQL交易均针对CS_VMDB中的视图来进行，这样可以避免随意改变测试数据库中的内容。

第四章　帮助系统

Teradata中提供了丰富的联机帮助功能，工作时如果能灵活利用这一联机帮助系统，可以节省很多翻阅参考文档的时间。

Teradata的帮助系统主要由三条命令组成，一条是HELP，一条是SHOW，另一条是EXPLAIN。HELP命令可以提供有关数据库中各种目标的信息，SHOW命令则用来显示这些目标的结构，包括创建这些目标的DDL语句。EXPLAIN命令以英文文字的方式显示了系统处理一个SQL交易请求的执行过程。

我们将详细地讨论这三条命令。

4.1 HELP命令

HELP命令可以用来显示数据库中各个目标的相关信息、当前对话连接的特性，并能提供SQL语法的联机帮助。

4.1.1 对数据库对象的帮助信息

数据库中的对象包括表、视图、宏、触发器和存储过程，HELP命令可以提供有关数据库和用户以及这些对象的信息。HELP命令可以汇总如表4-1所示：

表4-1 针对数据库对象的HELP命令

HELP命令	参数
HELP DATABASE	databasename;
HELP USER	username;
HELP TABLE	tablename ;
HELP VIEW	viewname;
HELP MACRO	macroname;
HELP COLUMN	table or viewname.*;
HELP COLUMN	table or viewname.colname . . ., colname;
HELP INDEX	tablename;
HELP STATISTICS	tablename;
HELP CONSTRAINT	table or viewname.constraintname;
HELP JOIN INDEX	join_indexname;
HELP TRIGGER	triggername;
HELP PROCEDURE	procedurename;
HELP PROCEDURE	procedurename ATTRIBUTES;

下表逐一举例说明。

HELP DATABASE/USER的使用

HELP DATABASE可以显示一个指定数据库所包含的所有对象，如表、视图、宏等。如要显示数据库customer_service中的对象，可以使用下面的命令：

HELP DATABASE customer_service;

系统返回信息如下：

Table/View/Macro name	Kind	Comment
contact	T	?
customer	T	?
department	T	?
employee	T	?
employee_phone	T	?
job	T	?
location	T	?
location_employee	T	?
location_phone	T	?

这里返回了三列信息。第一列是表、视图或宏的名字，第二列则是类型，T表示表(Table)，V表示视图(View)，M则表示宏(Macro)。第三列表示注释，?表示没有注释。

同理，如果要显示某个用户中所包含对象的信息，则可以使用HELP USER命令。如下面的命令将显示DBC用户下所包含的所有对象：

```
HELP USER DBC;
```

HELP TABLE/VIEW/MACRO的使用

根据HELP DATABASE/USER返回的信息，如果想进一步了解其中某一个具体的表、视图或者宏，则可以使用HELP TABLE/ VIEW/MACRO命令。如显示雇员表employee的信息，可以使用下面的命令：

```
HELP     TABLE          customer_service.employee;
```

系统返回信息如下：

Column Name	Type	Comment
employee_number	I	?
manager_employee_number	I	?
department_number	I	?
job_code	I	?
last_name	CF	?
first_name	CV	?
hire_date	DA	?
birthdate	DA	?
salary_amount	D	?

在TYPE栏中，可能的输出说明如下：

类型	说明
I	INTEGER
I1	BYTEINT
I2	SMALLINT
DA	DATE
D	DECIMAL
CV	CHARACTER VARIABLE (VARCHAR)
CF	CHARACTER FIXED (CHAR)

了解视图EMPLOYEE_PHONE的情况可用下面的命令：

```
HELP VIEW cs_views.employee_phone;
```

系统返回信息如下：

Column Name	Type	Comment
employee_number	?	?
area_code	?	?
phone	?	?
extension	?	?
comment_line	?	?

如欲了解EMPLOYEE表中各列的信息，可用下面的命令：

HELP COLUMN customer_service.employee.* ;

系统返回信息如下，这里包含了各列的显示格式：

Column Name	Type	Nullable	Format
employee_number	I	Y	-(10)9
manager_employee_number	I	Y	-(10)9
department_number	I	Y	-(10)9
job_code	I	Y	-(10)9
last_name	CF	N	X(20)
first_name	CV	N	X(30)
hire_date	DA	N	YY/MM/DD
birthdate	DA	N	YY/MM/DD
salary_amount	D	N	---------.99

其他HELP命令将在后面章节中介绍。

4.1.2 对数据库连接(或会话)的帮助信息

当客户端与Teradata建立好连接后，可以使用下面的命令来了解当前连接的一些信息。

HELP SESSION;

系统以下面的格式来返回信息：

User Name	Account Name	Logon Date
SQL01	$M_P0623	99/03/17

这种格式是在BTEQ下产生的。有关BTEQ的使用方法可参见附录。由于显示宽度的限制，还有很多列的信息没有显示出来。这时，可以使用下面的命令来调整输出方式：

.SET FOLDLINE ON
.SET SIDETITLES ON

此时再执行HELP SESSION的命令，将得到类似下面的输出结果：

User Name	SQL01
Account Name	$M_P0623
Logon Date	99/03/17
Logon Time	14:24:43
Current DataBase	CS_VIEWS
Collation	ASCII
Character Set	ASCII
Transaction Semantics	Teradata

在BTEQ方式下递交SQL交易请求时，如果产生的输出太长，可以结合使用SET FOLDLINE ON和SET SIDETITLES ON这两个命令来调整输出格式。例如前面的HELP COLUMN命令，产生的输出也是很长，如果将FOLDLINE和SIDETITLES设置成ON，则产下面的命令：

```
HELP COLUMN CUSTOMER_SERVICE.EMPLOYEE.EMPLOYEE_NUMBER
```

将产生输出：

```
                 Column Name  employee_number
                        Type  I
                    Nullable  Y
                      Format  -(10)9
                  Max Length  4
          Decimal Total Digits  ?
       Decimal Fractional Digits  ?
                   Range Low  ?
                  Range High  ?
                   UpperCase  N
                 Table/View?  T
                    Indexed?  Y
                     Unique?  Y
                    Primary?  P
                       Title  ?
           Column Constraint  ?
```

4.1.3 对SQL语法的联机帮助信息

如果忘记了SQL命令，可以使用下面的命令来得到所有SQL命令的列表信息：

```
HELP 'SQL';
```

如果要了解某条SQL命令，的使用方法，则可以使用下面的命令：

```
HELP 'SQL sqlcommand';
```

如：

```
HELP 'SQL SELECT';
HELP 'SQL INSERT';
HELP 'SQL UPDATE';
```

4.2 SHOW命令

上面谈到的HELP命令提供了对数据库中各种对象的帮助信息，也可以提供SQL语法的联机帮助。如果要显示数据库中各种对象的结构，换言之，要知道这些数据库对象是用什么样的DDL命令创建的，则要使用SHOW命令。表4-2包括SHOW命令。

表4-2 针对数据库对象的HELP命令

SHOW命令	参数

SHOW TABLE	Tablename ;
SHOW VIEW	Viewname;
SHOW MACRO	Macroname;
SHOW INDEX	Tablename;
SHOW JOIN INDEX	join_indexname;
SHOW TRIGGER	Triggername;
SHOW PROCEDURE	Procedurename;

BTEQ还有一个SHOW命令，SHOW CONTROL，这是一个附加的SQL SHOW命令。它能够格式化显示当前BTEQ会话的信息。

SHOW TABLE/VIEW/MACRO的使用

如果要显示EMPLOYEE表的结构，可使用下面的命令：

SHOW TABLE customer_service.employee;

系统返回信息如下：

```
CREATE SET TABLE customer_service.employee ,FALLBACK ,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL
    (
     employee_number INTEGER,
     manager_employee_number INTEGER,
     department_number INTEGER,
     job_code INTEGER,
```

```
    last_name CHAR(20) NOT CASESPECIFIC NOT NULL,
    first_name VARCHAR(30) NOT CASESPECIFIC NOT NULL,
    hire_date DATE NOT NULL,
    birthdate DATE NOT NULL,
    salary_amount DECIMAL(10,2) NOT NULL)
UNIQUE PRIMARY INDEX ( employee_number );
```

由此可知，SHOW TABLE的命令返回了指定表的DDL语句。

如果要显示CS_VIEWS下面一个视图EMPLOYEE_PHONE的结构，可以使用下面的命令：

```
SHOW VIEW cs_views.employee_phone;
```

系统返回信息如下：

```
CREATE VIEW employee_phone
   (employee_number
   ,area_code
   ,phone
   ,extension
   ,comment_line)
  AS
  SELECT
    employee_number
   ,area_code
   ,phone
   ,extension
   ,comment_line
  FROM CUSTOMER_SERVICE.employee_phone;
```

同理，可以使用SHOW MACRO来显示某个宏的创建命令。

4.3 EXPLAIN命令

最后要介绍的帮助命令是EXPLAIN。利用EXPLAIN命令，可以了解Teradata执行一个SQL交易请求的详细过程和计划，这对于更进一步地理解Teradata的查询处理机制有很大的帮助。另一方面，对于复杂SQL交易的调试来说，这也是不可缺少的一个工具。

利用EXPLAIN解释一个SQL交易的方法很简单，就是在原来SQL语句的前面加上EXPLAIN即可，其它完全不变。系统返回的信息包括：

- 提供完整的由分解器对 SQL 语句进行分解和优化后的 AMP 执行步。
- 这种执行计划是基于当前的数据分布情况而作出的，因此当数据分布发生变化时，同样 SQL 语句产生的执行步可能不相同。
- EXPLAIN 还会产生执行每个 SQL 步骤大致所需要的时间，但需要注意的是，这个时间由于是根据早期版本的 CPU 处理时间来计算，因此往往和实际情况相差很多，仅能作参考而已。

下面让我们来举个例子进行说明。假设一个如下的SQL语句：

```
SELECT * FROM department;
```

该语句非常简单，从DEPARTMENT表中将所有的记录选择并显示出来。如果要看一下它的执行过程，而不需要实际的结果，可以在前面加上EXPLAIN，如下：

```
EXPLAIN          SELECT * FROM department;
```

系统返回信息如下：

Explanation

1) First, we lock CUSTOMER_SERVICE.department for read.

2) Next, we do an all-AMPs RETRIEVE step from CUSTOMER _SERVICE.department by way of an all-rows scan with no residual conditions into Spool 1, which is built locally on the AMPs. The size of Spool 1 is estimated to be 4 rows. The estimated time for this step is 0.08 seconds.

3) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.

-> The contents of Spool 1 are sent back to the user as the result of statement 1. The total estimated time 0.08 seconds.

从系统返回的信息可以看到，该SQL请求是一个牵涉到所有AMP的操作(All-AMP Operation)。对于一个复杂的SQL交易请求，如果执行出错(如空间溢出等)或返回结果与所预料的相差很远，则有可能是SQL语句本身存在问题。这时可以利用EXPLAIN机制来了解一下Teradata的处理过程，从而发现和解决问题。

练习

Lab 4_1

利用HELP DATABASE命令找出CS_VMDBS数据库中所有的表、视图和宏，在该数据库中有哪些类型的数据库对象？同样再检查一下CUSTOMER_SERVICE数据库，看有哪些类型的数据库对象？

Lab 4_2

利用HELP命令找出DEPARTMENT表中所有的字段名。

Lab 4_3

在Lab 4_1中，应该能找到CS_VMDB数据库中有一个视图EMP，怎样知道它是如何建立的？

Lab 4_4

利用EXPLAIN命令检查一下SQL语句SELECT * FROM EMPLOYEE的执行过程。

Lab 4_5

如何观察EMPLOYEE_PHONE表的定义？

Lab 4_6

在CUSTOMER表中存在哪些类型的索引？

Lab 4-7

用DATABASE命令将当前数据库改为CUSTOMER_SERVICE，再执行下面的SQL语句：

```
SEL * FROM emp;
```

它还能正常运行吗？如果不行，应该如何调整。

第五章　逻辑与条件表达式

用户利用SELECT语句进行数据查询时，无论是描述想要的查询结果，还是规定查询的约束条件，均要使用相应的逻辑和条件表达式来实现。本章将详细介绍Teradata支持的各种逻辑和条件表达式。

5.1 逻辑表达式运算符

逻辑表达式由运算符和操作数两部分组成，其结果是一个布尔值(True/False)。这种表达式可以用在WHERE子句的条件表达式中。

标准的逻辑表达式运算符种类如表5-1所示：

表5-1 逻辑表达式运算符分类

运算符种类	符号	含义
比较运算符	=	等于
	<>	不等于
	>	大于
	<	小于
	>=	大于或等于
	<=	小于或等于
	[NOT] BETWEEN <a> AND <b>	介于a和b之间或不介于a和b之间
[NOT] IN	[NOT] IN	属于或不属于某个集合
IS [NOT] NULL	IS [NOT] NULL	一个数值是空值或不是空值
[NOT] EXISTS	[NOT] EXISTS	一个查询至少返回一行或不返回任何行

LIKE	LIKE	与某个数值匹配

Teradata还提供了下面一组扩展符号，可以与标准的比较运算符相互替换。

Teradata扩展	EQ	NE	GT	LT	GE	LE
ANSI标准	=	<>	>	<	>=	<=

5.2 逻辑表达式

下面分别讨论各种逻辑表达式的操作。

5.2.1 [NOT] BETWEEN...AND

BETWEEN <a> AND <b>表示某列数值(数字型或字符型)介于a和b之间，且包括a和b。

举例来说，在雇员表中把所有工作代码以43开头的员工的姓名及其经理的员工代码找出来。可以使用下面的SQL语句，其中第二个查询就使用了BETWEEN ... AND操作符。

```
SELECT  first_name
        ,last_name
        ,manager_employee_number
FROM    employee
WHERE   job_code >= 430000 AND
        job_code <= 439999;
```

```
SELECT  first_name
        ,last_name
        ,manager_employee_number
FROM    employee
WHERE  job_code BETWEEN 430000 AND 439999;
```

需要注意的是，当操作数是字符时，ANSI标准是区分大小写的，而Teradata中缺省不区分大小写，但可以通过CASESPECIFIC/NOT CASESPECIFIC来明确定义是否区分。如：

```
SELECT      ...
WHERE       last_name (CASESPECIFIC);

SELECT      ...
WHERE       last_name (NOT CASESPECIFIC);
```

查询姓名以字母R开头的员工，可以使用下面的SQL语句：

```
SELECT last_name
FROM    employee
WHERE  last_name BETWEEN 'r' AND 's';
```

由于Teradata的缺省方式不区分大小写，因此，Ryan、Roger等都将作为结果返回。如果显示区分大小写，上面的表达式中必须使用大写字母R和S。

5.2.2 集合操作符[NOT] IN

[NOT] IN表示某列数值属于或不属于某个集合。对于显示列举的数据集合，IN和OR可以相互替代使用。例如:在员工表中查找部门代码在401和403之间的员工姓名和相应的部门代码，可以使用下面两个SQL语句，它们的作用是等同的。显然，当要列举的数据值很多时，用OR就不如用IN方便和简洁。

```
SELECT  first_name
        ,last_name
        ,department_number
FROM    employee
WHERE  department_number = 401
   OR      department_number = 403;

SELECT  first_name
        ,last_name
        ,department_number
FROM    employee
WHERE  department_number IN (401, 403);
```

反过来，如果查找部门代码不属于401和403的员工姓名和部门代码，则可以使用下面的SQL语句，它们也是等价的。

```
SELECT  first_name
        ,last_name
        ,department_number
FROM    employee
WHERE  NOT (department_number = 401
   OR      department_number = 403);
```

```
SELECT  first_name
        ,last_name
        ,department_number
FROM    employee
WHERE  department_number NOT IN (401, 403);
```

5.2.3 匹配符LIKE

LIKE用来进行字符串数据的模式匹配。用作匹配的字符串中可以包含下面的通配符。

- %：表示除了 NULL 外的零个或多个字符组成的字符串。
- _：表示任何单个字符位置。

从下面的例子可以清楚地了解这两个通配符的使用方法：

表达式	含义
LIKE 'JO%'	‘JO’在开始位置的任意字符串
LIKE '%JO%'	‘JO'在任何位置的字符串
LIKE '%HN'	'HN'在结束位置的任意字符串
LIKE '%H_'	'H'在倒数第二个位置时的任意字符串
LIKE'慱_HN’	‘HN’分别在第三和第四位置的任意字符串

在使用LIKE进行字符串匹配时，要特别注意字符的大小写。ANSI标准中是区分大小写的，如果不要区分大小写，可以使用UPPER函数将其转换成大写字母来进行匹配。Teradata缺省不区分大小写，如果要区分，可以使用其扩展参数CASESPECIFIC。

例如：查询员工姓中有’Ra’字符的员工，可以根据是否区分大小写而分别使用下面的SQL语句：

不区分大小写

1.在Teradata缺省模式下

```
SELECT  first_name
        ,last_name
FROM    employee
WHERE  last_name LIKE '%Ra%';
```

2.在ANSI缺省模式下

```
SELECT  first_name
        ,last_name
FROM    employee
WHERE  UPPER (last_name) LIKE UPPER ('%Ra%');
```

区分大小写

1.在Teradata缺省模式下

```
SELECT  first_name
        ,last_name
FROM    employee
WHERE  last_name (CASESPECIFIC) LIKE '%Ra%';
```

2.在ANSI缺省模式下

```
SELECT  first_name
        ,last_name
FROM    employee
WHERE  last_name LIKE '%Ra%';
```

LIKE中限定词的使用

利用一些限定词可以扩充LIKE在字符串匹配方面的功能。可以使用的限定词包含：

限定词	含义
ANY	与一个或多个数值匹配
SOME (ANY的同义词)	同上
ALL	与列举的所有数值匹配

例如：查询员工姓中任意位置有字母’E’和’S’的员工，可以使用下面的SQL语句。

```
SELECT  first_name
        ,last_name
FROM    employee
WHERE  last_name LIKE ALL ('%E%', '%S%');
```

如果改变一下上面的问题，要求查找员工姓中任意位置有字母’E’或者’S’的员工，则应使用ANY或SOME。

```
SELECT  first_name
        ,last_name
FROM    employee
WHERE  last_name LIKE ANY ('%E%', '%S%');
```

通配符作为一般字符使用

前面谈到，在LIKE结构的字符串中，'%'和'_'可以作为通配符使用，但是如果需要匹配这些字符本身(比如查找95%)，即把它们作为一般字符时使用，该如何区分呢？

我们可以通过定义ESCAPE字符来达到这个目的，紧跟在ESCAPE字符后的’%’和’_’作为一般字符看待。

模式	含义
<ESCAPE char>_	_ (下划线)作为一个字符
<ESCAPE char>%	% (百分号)作为一个字符
<ESCAPE char><ESCAPE char>	ESCAPE字符本身

例：

LIKE "%A%%AAA__" ESCAPE "A"

在这个表达式中，将字母A定义为ESCAPE字符，其中：

- 第一个%为通配符;
- 第一个 A 和其后的%　?合表示字符%;
- 第三个%为通配符;
- 第二个 A 和其后的 A 联合表示字符 A;
- 第四个 A 和其后的’_’联合表示字符_;
- 最后一个’_’为通配符。

例如，下面的SQL语句可用来查找Teradata RDBMS数据字典中对象名称中第二个字符为’_’的所有对象。

```
SELECT tvmname
FROM    dbc.tvm
```

```
WHERE  tvmname LIKE "_Z_%" ESCAPE "Z'
;
```

5.3 NULL的使用

NULL是SQL的一个关键字，在数据库的操作中有很重要的作用，下面是一些关于NULL的说明：

- NULL 显示没有数据的字段
- NULL 表示不存在或未发现的值
- NULL 既不是数字类型也不是字符类型
- 具有 NULL 值的字段可以被压缩，不占任何空间

NULL也可以参与运算，其运算规则为：

- NULL 在算术运算中产生的结果为 NULL(空)
- NULL 在比较运算中产生的结果为 False
- UNKNOWN DATA, MISSING DATA 和 NULL 是同样的含义
- 当进行升序排列时,NULL 在数字列排列在负数前，在字符列排列在空格前。

下面给出了一些实例：

功能	例子			结果
+	10	+	NULL	NULL
-	10	-	NULL	NULL
*	10	*	NULL	NULL

/	10	/	NULL	NULL
>	10	>	NULL	UNKNOWN/FALSE
<	10	<	NULL	UNKNOWN/FALSE
>=	10	>=	NULL	UNKNOWN/FALSE
<=	10	<=	NULL	UNKNOWN/FALSE
=	10	=	NULL	UNKNOWN/FALSE
<>	10	<>	NULL	UNKNOWN/FALSE
>	NULL	>	NULL	UNKNOWN/FALSE
<	NULL	<	NULL	UNKNOWN/FALSE
>=	NULL	>=	NULL	UNKNOWN/FALSE
<=	NULL	<=	NULL	UNKNOWN/FALSE
=	NULL	=	NULL	UNKNOWN/FALSE
<>	NULL	<>	NULL	UNKNOWN/FALSE

NULL也可以用在SQL的SELECT中，如找出没有或未知分机号码的员工，使用下面的SQL语句：

```
SELECT employee_number
FROM   employee_phone
WHERE  extension IS NULL;
```

反过来，找出有分机号码的员工，则可以使用：

```
SELECT employee_number
FROM   employee_phone
WHERE  extension IS NOT NULL;
```

5.4 条件表达式

条件表达式是可以通过逻辑运算符来构造的，系统中的逻辑运算符为：

运算符	含义
AND	所有条件都必须成立
OR	其中任何一个条件成立即可
NOT	否定,即对条件求反

使用一个或多个逻辑运算符可以表现一些复杂的关系。

5.4.1 AND

AND连接两个或两个以上条件表达式。如果所有条件同时为真，则表达式结果为真；否则表达式结果为假。

例：查找收入小于35000并且部门号为403的员工姓名，可以使用下面的SQL语句：

```
SELECT  first_name
        ,last_name
        ,employee_number
FROM    employee
WHERE  salary_amount  <       35000.00
    AND department_number    = 403;
```

5.4.2 OR

OR连接两个或两个以上条件表达式。如果任何一个条件为真，则表达式结果为真；如果所有条件均为假，则表达式结果为假。

例：查找收入小于35000或部门号为403的员工姓名。

```
SELECT  first_name
        ,last_name
        ,employee_number
FROM    employee
WHERE salary_amount  <      35000.00
   OR     department_number   =       403;
```

5.4.3 多个AND . . . OR

例：查找部门号是403或401，并且工作号为412101或432101的员工姓名、部门号和工作号。

```
SELECT  last_name
        ,department_number
        ,job_code
FROM    employee
WHERE (       department_number = 401
        OR    department_number = 403)
    AND (     job_code = 412101
        OR    job_code = 432101);
```

5.4.4 优先级和括号

括号也是逻辑运算符，可用来改变逻辑运算的优先级。如果条件表达式中不使用括号，逻辑运算符的缺省优先级顺序从高到低为：NOT、AND、OR。

为了避免出错，应尽量使用括号来显示地指定运算秩序。

5.4.5 NOT

NOT既可以否定操作符，也可以否定条件表达式。

例；查找部门号不是301的员工姓名及其代码。

否定操作符:

```
SELECT  first_name
        ,last_name
        ,employee_number
FROM    employee
WHERE  department_number NOT = 301;
```

否定条件：

```
SELECT  first_name
        ,last_name
        ,employee_number
FROM    employee
```

WHERE NOT (department_number = 301);

练习

Lab 5_1 查询收入在$40001和$50000之间所有员工的号码、姓、收入和工作代码。解决这个问题有许多方法，请先使用比较运算符，并按姓分类；其次，使用BETWEEN运算符并按收入分类。

Lab 5_2 查找在501、301、201三个部门中收入超过$30000的员工的号码、部门号、收入，要求按部门号和收入分类。

Lab 5_3

从Location_Phone表中查找那些描述栏(description)为经理(Manager)的各项记录，显示位置代码、区域代码、电话、分机号和描述，并按位置代码排序。

Lab 5_4

查找满足下面条件的所有客户：

1、其上级客户编号(parent_customer_number)未知

2、其对应的销售编号(sales_employee_number)未知

列出除上级客户编号外的所有字段，并按客户编号排序。

Lab 5_5

查找在415区域并且没有专门负责的任何经理的位置(Location)，列出除comment和extension以外的所有信息，并按电话号码排序。

Lab 5_6

查询工作代码是400000系列(属于管理人员)的员工以及所有的分析人员，显示其工作代码和描述，并按描述排序。然后用Teradata的EXPLAIN来观察一下该查询的执行过程。

Lab 5_7

列出没有电话分机的所有位置(Location)，并且其地区代码为415和619，显示除描述和注释字段外的所有信息，并按地区代码和电话号码排序。

第六章　数据转换和计算

本章主要介绍Teradata的数据类型以及数据类型之间的转换和数据计算。数据类型是Teradata存储和操纵的数据的标准形式，反映和规定了系统中常量和变量的固有特性，这些特性使得对不同数据类型的数据将采取不同的处理方式。

6.1 数据类型

前面已经列举了Teradata中支持的各种数据类型，下面将对这些数据类型作更为详细的介绍。

6.1.1 字符型数据

在ANSI标准中关于字符型数据定义了两类：CHAR和VARCHAR。其中CHAR表示固定长度的字符串，VARCHAR表示可变长度的字符串。

Teradata除了上述两类基本字符型数据外，还扩展了LONG VARCHAR类型，它等同于VARCHAR(64000)，是最长的字符串。

表6-1 Teradata字符数据类型

字符型	意义	例

数据类型		
CHAR (size)	固定长度的字符串 最大长度：64000字节	last_name CHAR(20) Ryan_____
VARCHAR (size) CHAR VARYING (size) CHARACTER VARYING (size)	可变长度字符串 最大长度：64000字节	first_name VARCHAR (30) Loretta
LONG VARCHAR	等同于VARCHAR(64000)	

6.1.2 二进制数据

二进制数据类型是Teradata的扩展，ANSI标准没有此类型。Teradata支持两类二进制数据BYTE和VARBYTE。其中BYTE表示固定长度的二进制串，VARCHAR表示可变长度的二进制串。

表6-2 Teradata二进制数据类型

类型	意义
BYTE (size)	固定长度的二进制字串 默认值：(1) 最大值：64000字节
VARBYTE (size)	可变长的二进制字串 默认值：(1) 最大值：64000字节

6.1.3 数字型数据

在ANSI标准中关于数字型数据定义了四类：SMALLINT、INTEGER、FLOAT、DECIMAL。Teradata除以上四类外，还有两类扩展的数字型数据，即BYTEINT和DATE。

表6-3 ANSI标准的数字型数据：

数据类型	描述	范例
SMALLINT	整数 范围：-32,768 - 32,767	area_code +00213
INTEGER	整数 范围：-2,147,483,648 - 2,147,483,647	phone +0006495252
DECIMAL (size, dec)	小数 最大：18位数字	salary_amount DEC (10,2) +00035000.00
NUMERIC (precision, dec)	DECIMAL的同义字	salary_amount NUMERIC (_,_) +00035000.00
FLOAT	表示浮点数	salaryfactor 4.35400000000000E-001
FLOAT [(precision)]	同FLOAT	
REAL	同FLOAT	
DOUBLE PRECISION	双精度浮点数	

Teradata扩展的数字型数据如表6-4所示：

表6-4 Teradata扩展的数字型数据

数据类型	描述	实例
BYTEINT	有符号整数 范围：-128至127	item_code BYTEINT +100
DATE	特殊整数，格式为YYMMDD 或YYYMMDD表示日期	hire_date DATE +960921

注意：

1. BYTEINT表示不包括零的有符号整数，占一个字节存储空间。

2. DATE用来表示日期，但内部以整数形式存储，其公式为：

(year - 1900) * 10000 + (month * 100) + day

Teradata定义的DATE和ANSI定义的DATE在内部表示上是不兼容的。ANSI的日期格式是YYYY-MM-DD，在Teradata中也推荐使用这种格式来进行表达，这样可以避免由于使用两位数字表示年而带来的2000年问题。

6.1.4 图形数据

Teradata扩展了图形数据类型，如：GRAPHIC、VARGRAPHIC、LONG VARGRAPHIC。图形数据类型主要是用来支持双字节字符的，如日文。当把系统设置成支持双字节字符时，数据库中表、视图等对象的名字、字段等都可以使用双字节字符。如果系统设置成不支持双字节字符，则数据库对象名、字段等只能使用

英文字母类的单字节字符，此时，前端工具仍然可以采用汉化界面，而且利用前面提到的CHAR和BYTE类型数据类型同样可以存储图形数据。

表6-5 Teradata扩展的图形数据(双字节系统)

数据类型	意义
GRAPHIC [(n)]	固定长度的图形字符串 默认长度：1
VARGRAPHIC (n)	可变长的图形字符串
LONG VARGRAPHIC	可变长的图形字符串

6.2 算术运算符

Teradata提供以下ANSI标准规定的算术运算符:

* 乘

/ 除

+ 加

- 减

+ 正号

- 负号

除此以外，Teradata还扩展了下面两个算术运算符：

** (求幂)

MOD(取模)

表示方法：<n><arg>,

例：

4**3 = 4 * 4 * 4 = 64.

MOD是取模运算符，表示除运算的余数。

例：

60 MOD 7 = 4

即：60除以7等于8，余数是4。

下面将Teradata的算术运算符总结如表6-6所示：

表6-6 Teradata算术运算符

运算符	意义
( )	括号内算术运算优先级高
*	乘
/	除
+	加(或正号)
-	减(或负号)
**	幂
MOD	模(余数)

6.3 Teradata算术函数

在ANSI标准中没有算术函数，Teradata作了扩充，它支持的算术函数如表6-7所示。

表6-7 Teradata支持的算术函数

函数	意义
ABS (arg)	求绝对值
EXP (arg)	增加幂
LOG (arg)	10的对数
LN (arg)	自然对数
SQRT (arg)	平方根

注：arg代表任意常数或变量。

例：

SQRT(16) = 4

6.4 运用算术运算符计算

例：查询部门代码为401的员工的姓名和月平均工资，并按姓名的字母顺序排序。

```
SELECT  last_name
                ,first_name
```

```
                ,salary_amount/ 12
    FROM employee
    WHERE       department_number = 401
    ORDER BY    last_name;
```

6.5 系统变量

在Teradata中，有几个系统变量DATE、TIME、USER和DATABASE，分别表示当前的系统日期、系统时间、当前登录的用户和当前缺省的数据库，它们的使用可参考下面的例子。

```
SELECT DATE;
Date
--------
99/04/15

SELECT TIME;
Time
--------
09:15:37

SELECT USER;
User
-----
SQL01

SELECT DATABASE;
Database
----------------
```

CUSTOMER_SERVICE

6.6 字符常量、数字常量和计算模式

字符文字常用于更加清晰的标识查询结果的意义。同样，在ANSI方式下区分大小写，而在Teradat缺省模式下不区分大小写。

例：查找员工表中的员工姓名，并在查询结果前加注'Employee'标识，可使用下面的SQL语句。

```
SELECT  'Employee'
        ,last_name
        ,first_name
FROM    employee;
```

结果

Employee	last_name	first_name
Employee	Stein	John
Employee	Kanieski	Carol
Employee	Ryan	Loretta

数字型常量最多可以包含15个数字，数字前面的零是无意义的。

例：在查询的员工姓名前加注12345数字型常量标识。

```
SELECT  12345
```

```
        ,last_name
        ,first_name
FROM    employee;
```

结果

12345	last_name	first_name
12345	Stein	John
12345	Kanieski	Carol
12345	Ryan	Loretta

计算模式是指在SQL的SELECT语句中直接进行数学计算，如下面各例所示：

```
SELECT 2*2593;
```

结果

(2 * 2593)
5186

例：识别location表中末尾是四个零的邮政编码。

```
SELECT  customer_number
        ,zip_code
FROM    location
WHERE  zip_code MOD 10000 = 0;
```

6.7 对日期的处理

6.7.1 日期计算

在Teradata数据库中将DATE型数据作为整数看待，但不容许无效的日期。计算公式如下：

((YEAR - 1900) * 10000) + (MONTH * 100) + DAY

例：1997年3月31日的表达方式

YEAR = (1997 - 1900) * 10000 = 97 * 10000 = 970000
MONTH = (3 * 100) = 300
DAY = 31
DATE = 970331

2002年3月31日的表达方式：

YEAR = (2002 - 1900) * 10000 = 102 * 10000 = 1020000
MONTH = (3 * 100) = 300
DAY = 31
DATE = 1020331

对日期的计算举例：

查询	语法
从现在起30天：	(DATE + 30)

从现在起365天：	(DATE + 365)
某个人的年龄：	(DATE - birthdate) / 365
工作十年或以上的员工：	(DATE - hire_date) / 365 > = 10

6.7.2 与日期有关的数据函数

Teradata中与日期有关的数据函数分别介绍如下：

1. EXTRACT

ANSI标准中EXTRACT函数允许选取日期和时间中任意字段或任意间隔的值，Teradata中EXTRACT函数支持日期数据中选取年、月、日，从时间数据中选取小时、分钟和秒。

EXTRACT使用举例如下：

日期函数	结果
SELECT DATE;	96/11/07
SELECT EXTRACT (YEAR FROM DATE);	1996
SELECT EXTRACT (MONTH FROM DATE + 30);	12
SELECT EXTRACT (DAY FROM DATE + 2);	09
SELECT TIME;	14:52:32
SELECT EXTRACT (HOUR FROM TIME);	14
SELECT EXTRACT (MINUTE FROM TIME);	52
SELECT EXTRACT (SECOND FROM TIME + 30);	Invalid time

2. ADD_MONTHS

ADD_MONTHS表示从某日期增加或减少指定月份的日期。它考虑了大小月问题，所以计算日期是准确的。ADD_MONTHS使用举例如下：

日期函数	结果
SELECT DATE;	96/08/07
ADD_MONTHS的使用	
SELECT ADD_MONTHS (DATE, 2);	1996-10-07
SELECT ADD_MONTHS (DATE, 12*14);	2010-08-07
SELECT ADD_MONTHS (DATE, -11);	1995-09-07
SELECT ADD_MONTHS (‘’1996-07-31’’, 2);	1996-09-30
SELECT ADD_MONTHS (‘’1995-12-31’’, 2);	1996-02-29
SELECT ADD_MONTHS (‘’1995-12-31’’, 14);	1997-02-28

注意ADD_MONTHS考虑了月份的大小，而如果在日期上直接加数字来计算月份，需要仔细考虑。下面的例子可以说明这个问题。

```
SELECT  DATE, ADD_MONTHS (DATE,2)
        ,DATE + 60;
```

Date	ADD_MONTHS(Date, 2)	(Date+60)
96/11/15	1997-01-15	97/01/14

我们看到，在当前日期后使用ADD_MONTHS来推算2个月后是哪一天，结果很准确。而(DATE + 60)只能表示60天后的日期，用这种方式来推算几个月后的日期很容易产生不正确的结果。

3. DATE在SELECT语句中的使用

例：查询在公司工作至少十年以上的员工姓名。

```
SELECT  last_name
        ,first_name
  FROM employee
  WHERE     (DATE-hire_date) / 365>= 10
  ORDER BY  last_name;
```

6.7.3 利用CAST作数据转换

ANSI标准中利用CAST函数将一种数据类型转换成另一种数据类型。

例：

```
SELECT CAST (salary_amount AS INTEGER)
FROM   employee;
```

Value	Result
50500.75	50500

```
SELECT CAST (salary_amount AS DEC (6,0))
FROM   employee;
```

Value	Result
50500.75	50501.

```
SELECT CAST (last_name AS CHAR (5))
FROM   employee
WHERE  department_number = 401;
```

last_name
Johns
Trade

Teradata也可以使用CAST函数来完成上面的操作，另外，它也作了扩充。举例来说，为了完成上面相同的操作，也可以使用下面的表达方式：

```
SELECT salary_amount (INTEGER);
   SELECT salary_amount (DEC(6,0));
   SELECT last_name (CHAR(5));
```

Teradata对CAST函数本身也作了扩展，比如为了将显示结果以大写表示，可以使用下面的SQL语句

```
SELECT CAST (last_name AS CHAR (5) UPPERCASE)
FROM    employee
WHERE  department_number = 401;
```

last_name
JOHNS
TRADE

练习

Lab 6_1

利用SQL语句显示当前的系统日期、系统时间和用户，显示从今天开始365天后的日期，查询60天后是几月？

Lab 6_2

创建一张报表，利用文字'Math Function'作为第一列，利用数字1作为第二列，计算2*3+4*5的值作为第三列。观察结果后，在3+4处加括号再提交，注意不同的结果。

Lab 6_3

401部门的经理希望知道部门全体员工加薪10%后的薪水是多少？要求显示401部门员工姓的前10个字符、当前薪水、加薪后薪水，并按薪水降序排列。然后用EXPLAIN观察一下该查询的执行过程。

Lab 6_4

显示15个月后是哪年？

Lab 6_5

如果对于5_3题作些改动，比较每人加薪10%和加薪$500的薪水值的差别。要求显示员工姓、加薪10%后薪水、加500元后薪水值以及两种加薪方式的差别。

Lab 6_6

政府需要作特殊的人口统计研究。创建年龄在40岁以下的员工报表，显示员工姓的前10个字符、受雇佣日期、年龄，并按年龄大小排序。

第七章　简单的宏

宏(Macro)的基本特征是：

- 可以包含一条或多条 SQL 语句
- 可以包含多个 BTEQ 语句
- 可以包含注解
- 存储在数据字典中

宏(Macro)是Teradata扩展的性能，ANSI标准不支持宏。

Terdata中与宏有关的命令如表7-1所示：

表7-1 Teradata中用于宏的命令

CREATE MACRO macroname AS (. . .);	定义宏
EXECUTE macroname;	执行宏语句
SHOW MACRO macroname;	显示宏定义
REPLACE MACRO macroname AS (. . .);	改变宏定义
DROP MACRO macroname;	从字典中删除宏定义
EXPLAIN EXEC macroname;	显示宏执行的解释

7.1 宏的定义

宏是用CREATE MACRO命令来定义的，如下例所示：

```
CREATE MACRO      birthday_list   AS
        (SELECT       last_name
                      ,first_name
                      ,birthdate
         FROM         employee
         WHERE        department_number = 201
         ORDER BY     birthdate; );
```

7.2 宏的执行

宏的执行很简单，使用EXEC命令就可以。例如，为了执行上面定义的宏，可以使用下面的语句：

```
EXEC     birthday_list;
```

7.3 宏的删除

使用DROP命令可以删除宏，如下所示：

```
DROP MACRO birthday_list;
```

7.4 宏的显示和改变

使用SHOW命令可以显示一个宏的定义，如下所示：

```
SHOW MACRO birthday_list;
```

使用REPLACE MACRO命令可以改变宏的定义，如：

```
REPLACE MACRO      birthday_list    AS
   (SELECT         last_name
                   ,first_name
                   ,birthdate
   FROM employee
   WHERE           department_number = 201
   ORDER BY        birthdate, last_name; ) ;
```

练习

在下面的练习中，使用练习题名作为宏名，例如，习题7_1的宏名可以为Lab 7_1。

Lab 7_1

写一个简单的宏来创建客户地理位置报表。利用位置表location中的信息，显示location number, customer number, city和state (city和state只显示14个字符)，按city和state分类。注意：必须在自己的数据库空间来创建宏，否则可能会有权限方面的限制。

Lab 7_2

修改Lab 7_1的宏，不显示location number，在报表最后一列增加显示zip code，并在分类中增加Zip code。且只选择位于Illinois州的客户。为宏重新命名。

Lab 7_3

利用HELP命令显示你所有的宏。

Lab 7_4

修改Lab 7_1的宏，要求：只选择Chicago市的信息，并将其第一地址作为报表最后一列。按客户编号作降序排列。

Lab 7_5

删除在Lab 7_1建立的宏。

第八章　子查询

在构造一些复杂查询时，经常都要使用到子查询。本章将主要讨论子查询。

8.1 基本子查询

假设需要查询所有部门经理的姓名，一种方法是先手工查询部门表中所有经理的员工代码，记住这些数据，再通过下述查询获得经理的姓名。

```
SELECT last_name
FROM    employee
WHERE  employee_number IN (1017,1005,1003);
```

这里假设各部门经理的代码分别是1017、1005、1003。如果有几千条记录，这种方式肯定不行。所以引出了子查询的概念。子查询是利用另一条SQL语句中的SELECT语句来获得中间结果，如下所示：

```
SELECT last_name
FROM    employee
WHERE  employee_number IN
   (SELECT manager_employee_number
    FROM          department );
```

再看一个例子，查出预算超过$900,000的部门经理。

```
SELECT  last_name
```

```
        ,first_name
FROM    employee
WHERE   employee_number IN
   (SELECT      manager_employee_number
    FROM        department
    WHERE       budget_amount>900000 );
```

如果进一步问：查找收入低于$35000并且预算超过$900,000的部门经理。

```
   SELECT       last_name
                ,first_name
   FROM employee
   WHERE        salary_amount<35000
       AND      employee_number IN
                (SELECT     manager_employee_number
                 FROM       department
                 WHERE      budget_amount > 900000 ) ;
```

8.2 复杂子查询

在子查询中可以使用一些限制符，如下所示：

= ANY	等 于	IN
NOT = ALL	等 于	NOT IN
= SOME	等 于	IN

例：查询工作在Support部门的员工信息。

```
SELECT last_name
       ,first_name
       ,department_number
FROM employee
WHERE department_number = ANY
       (SELECT      department_number
        FROM        department
        WHERE       department_name LIKE '%Support%');
```

例：查询被分配管理同一部门的部门级经理。

```
SELECT employee_number
       ,last_name
       ,first_name
FROM employee
WHERE       (department_number, employee_number) IN
       (SELECT      department_number
                    ,manager_employee_number
        FROM        department   );
```

8.3 EXISTS在子查询中的使用

EXISTS可以使用在子查询中，用来表示查询至少返回一行。如果前面加上否定词NOT，则表示查询时无记录存在。EXISTS可以代替IN，而NOT EXISTS可以代替NOT IN。

例：查询哪个部门没有员工？

```
SELECT department_number
FROM     department
WHERE department_number NOT IN
   (SELECT  department_number FROM employee);


SELECT 'YES'
WHERE EXISTS
   (SELECT  department_number FROM department
    WHERE   department_number NOT IN
            (SELECT  department_number
             FROM employee));
```

第二个SQL返回的结果如下：

```
'YES'
YES
```

例：查询在600部门是否有员工？

```
SELECT TRUE
WHERE EXISTS
            (SELECT   *   FROM employee
             WHERE    department_number = 600);
```

*** Query completed. No rows found.

8.4 关于子查询的一些基本规则

下面总结一下使用子查询时的一些基本规则。

- 子查询必须用括号括起来
- 子查询可以是 IN 或 NOT IN 子句的操作目标
- 也可以是 EXISTS 或 NOT EXISTS 子句的操作目标
- 支持限定词 ALL, ANY, SOME
- 支持 LIKE 或 NOT LIKE
- 子查询中可以指定匹配多个字段
- 子查询结果均为唯一值，即自动去除重复记录，相当于自动加上 DISTINCT 关键词
- ORDER BY 不能用于子查询内
- 子查询中最多可以指定 16 个表或视图

练习

Lab 8_1

利用EXISTS识别是否有员工的工作号码是无效的，如果有则返回TRUE。

Lab 8_2

查询员工表中工作代码无效的员工姓名和工作代码。要求利用一个子查询来进行参照完整性的检查。

注意：解决这个问题可以使用限定词，也可不使用。

Lab 8_3

查询员工表中部门号无效或为空的员工信息，显示员工代码、姓和部门代码。

Lab 8_4

查询在California州的客户信息，要求显示客户编号、客户姓名和相应的销售雇员代码。按销售雇员代码排序。

Lab 8_5

查询所有工作代码是211100的部门经理的信息，显示雇员代码、姓、部门代码和工作代码。

Lab 8_6

在上个练习中我们找到了工作代码为211100的部门经理的姓名，这里要求找出所有为这些经理工作的员工，显示员工代码、姓、部门代码和工作代码。

第九章　属性和函数

在Teradata中使用SQL的SELECT时，如果不作特殊的规定，输出将采用缺省的格式。Teradata提供了丰富的命令和函数来进行输出的格式化，本章主要讨论与此有关的问题。

9.1 表达式属性

列和表达式的属性可以定义标题和格式，表达式属性主要包括AS (NAMED)、TITLE和FORMAT，表9-1总结了它们的主要功能。

表9-1字段格式化关键词汇总

关键词	描述
AS (NAMED)	将字段名改成指定的名字输出
TITLE	定义一个与缺省列名不同的标题用于显示或打印结果
FORMAT	用于显示或打印指定格式的列名或表达式

其中NAMED、TITLE和FORMAT是Teradata在ANSI外的扩充。这些关键词的使用方法可以通过下面的例子很容易地掌握。

例：

SELECT last_name

```
        ,first_name
        ,salary_amount / 12 AS monthly_salary
FROM    employee
WHERE   department_number = 401
ORDER BY    monthly_salary;
```

返回结果如下：

last_name	first_name	monthly_salary
Johnson	Darlene	3025.00
Trader	James	3154.17

使用TITLE关键字时注意下面的几点基本规则：

- 格式化列标题最多可以堆成三行
- 使用符号//表示标题的分行
- 标题中可以包含空格

例：

```
SELECT  last_name
        ,first_name
        ,salary_amount / 12 (TITLE 'MONTHLY // SALARY')
FROM    employee
WHERE   department_number = 401
ORDER BY    3;
```

返回结果如下：

		MONTHLY
last_name	first_name	SALARY
Johnson	Darlene	3025.00
Trader	James	3154.17

该例子中格式化部分的另一种写法是：

```
SELECT ...
   ,CAST (salary_amount/12 AS TITLE "MONTHLY//SALARY")
   FROM ...
```

9.2 CHARACTERS函数

CHARACTERS函数也是Teradata的扩展，用于计算VARCHAR型数据字段的实际字符串长度。CHARACTERS函数可以简写成CHARACTER、CHARS或者CHAR。

例：查询所有姓中包含5个以上字符的员工。

```
SELECT first_name
FROM    employee
WHERE  CHARACTERS (first_name) > 5;
```

9.3 TRIM函数

ANSI标准的TRIM函数用于去除字符数据中前头或后端的空格或者二进制数据(BYTE与VARBYTE)中前头或后端的零。它的基本语法如表9-2所示。

表9-2 ANSI标准TRIM函数的使用方法

语法	意义
TRIM (<expression>)	去除字符数据中前后端的空格或者二进制数据中前后头的零
TRIM (BOTH FROM <expression>)	同上
TRIM (TRAILING FROM <expression>)	去除后端的空格或二进制零
TRIM (LEADING FROM <expression>)	去除前端的空格或二进制零

Teradata中TRIM函数的作用与ANSI标准略有不同，差别在于：在Teradata缺省模式下，TRIM (<expression>)只能去除后端的空格或二进制零。

例：查询姓是由四个字符组成的员工有哪些？

```
SELECT  first_name
        ,last_name (TITLE 'last')
FROM    employee
WHERE  CHAR (TRIM (TRAILING FROM last_name)) = 4 ;
```

first_name	last
Loretta	Ryan

James Daly

9.4 FORMAT短语

FORMAT用于数据在输出时的格式化处理，但它并不影响数据的内部存储格式。它主要有如下面例子所示的两种表示方式。

例：

```
SELECT salary_amount (FORMAT "$$$,$$9.99");
```

例：

```
SELECT CAST (salary_amount AS FORMAT "$$$,$$9.99");
```

注意，在第二个例子中CAST的使用方法是Teradata的扩展，因为这里CAST处理的是一个数据的属性，而在ANSI标准中，CAST只能用于处理一种数据类型。

我们再看几个例子来进一步了解FORMAT短语的使用方法。

例：

```
SELECT salary_amount
                    (FORMAT '$$$,$$9.99')
FROM    employee
WHERE  employee_number = 1004;
```

salary_amount
$36,300.00

对该例子的另一种写法是：

```
SELECT CAST (salary_amount  AS FORMAT "$$$,$$9.99") FROM ...
```

例：管理人员打算将1004号员工的工资增加$1000元，他想了解工资提高的百分率。

```
SELECT (1000/salary_amount) * 100
                    (FORMAT 'ZZ9%')
                    (TITLE 'Increase Percentage')
FROM    employee
WHERE  employee_number = 1004;
```

Increase Percentage
3%

对该例子的另一种写法是：

```
SELECT CAST (1000/salary_amount) * 100  AS FORMAT 'ZZ9%'
 TITLE 'Increase Percentage") FROM ...
```

FORMAT短语中可以使用的格式化字符主要为：

- $ 美元标识符
- 9 数字位
- Z 将数字中的前缀零去除
- , 在指定位置插入逗号
- . 指定小数点位置
- - 在指定位置插入连字号
- / 在指定位置插入斜线
- % 在指定位置插入百分号
- X 字符数据，每个 X 代表一个字符
- G 图形数据.一个 G 代表一个逻辑字符(双字节)
- B 在指定位置插入空格

根据上面的定义，看看下面各格式化输出的结果是什么。

```
FORMAT '999999'     Data: 08777     Result ?
FORMAT 'ZZZZZ9'     Data: 08777     Result ?
FORMAT '999-9999'   Data: 6495252   Result ?
FORMAT 'X(3)'       Data: 'Smith'   Result ?
FORMAT '$$9.99'     Data: 85.65     Result ?
FORMAT '999.99'     Data: 85.65     Result ?
FORMAT    (3)'      Data: 85.65     Result ?
```

9.5 对日期的格式化处理

在Teradata中，日期数据的缺省输出格式是：YY/MM/DD，这和ANSI标准是一样的。而ANSI标准建议的日期显示格式是：YYYY-MM-DD。

其它一些常用的日期显示格式列举如下，其中的B表示空格。

YYYY/MM/DD'
YYYY-MM-DD'
YYYY.DDD'
DBMMMMBYYYY'
MMBDD,BYYYY'
YYYYBMMMMBDD'
YY/MM/DD'
D-MM-YY'
YBDDD
MM'

注意：由于2000年问题，ANSI推荐使用日期格式为YYYY-MM-DD，或者其它采用四位年的格式。

下面是一些对日期进行格式化的例子。

句法	结果
FORMAT 'YYYY/MM/DD'	1996/03/27
FORMAT 'DDbMMMbYYYY'	27 Mar 1996
FORMAT 'mmmBdd,Byyyy'	Mar 27, 1996
FORMAT 'DD.MM.YYYY'	27.03.1996
FORMAT 'MM/DD/YY'	03/27/96
FORMAT 'MMM.DD.YY'	Mar.27.96
FORMAT 'yy -- mm -- dd'	96 -- 03 -- 27
FORMAT 'DDDYY'	08696

下面再看一下在SELECT语句中对日期进行格式化的例子。

```
SELECT  last_name
        ,first_name
        ,hire_date (FORMAT 'mmmBdd,Byyyy')
FROM    employee
ORDER BY    last_name;
```

last_name	first_name	hire_date
Johnson	Darlene	Oct 15, 1976
Kanieski	Carol	Feb 01, 1977
Ryan	Loretta	Oct 15, 1976

也可以从日期中抽取某一部分的内容来进行显示。

例：选取日期某部分

1.利用FORMAT:

```
SELECT  last_name
        ,first_name
        ,birthdate      (FORMAT 'mmdd')
                        AS birthday
        ,birthdate      AS whole_date
FROM    employee
WHERE  department_number = 401
ORDER BY    3;
```

last_name	first_name	birthday	whole_date
Rogers	Frank	0423	35/04/23

Brown	Alan	0809	44/08/09
Johnson	Darlene	0423	46/04/23

2.利用取模函数

```
SELECT  last_name
        ,first_name
        ,birthdate MOD 10000 (FORMAT "9999")
                AS birthday
        ,birthdate      AS whole_date
FROM    employee
WHERE  department_number = 401
ORDER BY      3;
```

last_name	first_name	birthday	whole_date
Johnson	Darlene	0423	46/04/23
Rogers	Frank	0423	35/04/23
Brown	Alan	0809	44/08/09

9.6 对字符数据的截取

利用FORMAT短语，可以将字符字段或表达式进行截取处理，这种处理只影响显示格式，而不会影响数据的内部存储格式。

例：查询所有姓Brown的员工并显示其姓的第一个大写字母。

1.利用FORMAT

```
SELECT  last_name
        ,first_name
        ,first_name  (FORMAT 'X')
FROM    employee
WHERE  last_name = 'Brown'
ORDER BY 3;
```

last_name	first_name	first_name
Brown	Alan	A
Brown	Allen	A

2.使用数据类型的转换

```
SELECT  last_name
        ,first_name
        ,CAST (first_name AS CHAR(1))
FROM    employee
WHERE  last_name = 'Brown'
ORDER BY      3;
```

last_name	first_name	first_name
Brown	Allen	A
Brown	Alan	A

9.7 属性函数

Teradata中还扩展了以下各种属性函数，它们用来返回指定变量或操作数据的有关描述信息。这些函数如表9-3所示。

表9-3 Teradata的属性函数

属性函数	返回信息说明
TYPE	数据类型
TITLE	标题短语
FORMAT	格式短语
NAMED	NAMED子句
CHARACTERS	字符个数

下面是一些相应的例子：

查询			结果
SELECT DISTINCT	TYPE (job_code)	FROM job;	INTEGER
SELECT DISTINCT	TITLE (job_code)	FROM job;	job_code
SELECT DISTINCT	FORMAT (job_code)	FROM job;	(10)9
SELECT DISTINCT	NAMED (job_code)	FROM job;	job_code
SELECT DISTINCT	CHARACTERS (job_code)	FROM job;	11

练习

Lab 9_1 401部门的经理需要一个报表，包括该部门每个员工的姓、雇用日期和双周薪水(一年26个双周)。要求：按薪水降序排列，双周薪水的标题为'Salary

Amount'，雇用日期的格式采用'MMMBDD, YYYY'，薪水前加上美元符号，每三位一个逗号，小数点后两位。注意，雇员表中的薪水是年薪。

Lab 9_2

查找员工的编号和电话号码，显示的标题分别采用'Employee'和'Phone'，在电话号码的第三位和第四位之间加上连字号，并按电话号话排序。

Lab 9_3

打印301部门员工的号码、姓名和年龄，姓、名、年龄分别用'Last Name'、'First Name'、 "Age"作为标题，姓显示9个字符，并按年龄升序排列。

Lab 9_4

列表显示员工姓(10个字符)、部门号和雇用日期。分别考虑以下两种情况：

1.找出所有雇用日期的月份与当前月份相同的员工(不考虑年)，以进行周年庆祝活动。

2.找出四个月前参加周年纪念的所有员工。

Lab 9_5

调整Lab 9_4的报表，使得按雇佣月份降序排列，并在报表中增加一列显示雇佣月份，标题为'month'，格式为'MMM'。

Lab 9_6

找出所有在501部门工作的员工，显示其姓名(姓和名均显示8个字符)，并增加另外三个字段：雇佣日期的月份(采用'month'作为标题)、雇佣月份(没有标题，输出格式为'MMM')、雇佣年份(采用'year'作为标题)。

第十章　内连接

对于关系数据库系统来说，范式理论是它的基础。多个表按照一定的规定进行组合，可以非常清晰地描述一个企业的业务运行方式。如何灵活地从多个表中查找所需要的信息，很重要的一个手段就是表的连接操作。

对任何关系数据库系统而言，表的连接操作都是非常耗系统资源的。我们常看到一些对某些商用RDBMS颇有经验的人指出，在把系统的逻辑模型转换到物理模型，进行真正的物理实施时，要作不规范化处理(De-Normalise)。一个常用的手段就是将多个表合并成一个表，尽量减少表的连接。之所有这样做的主要原因在于，这些商用RDBMS在处理复杂的多表连接操作时速度太慢，因此将表进行预先的合并，减少表的连接，以提高系统的查询处理速度。这样处理的最大问题是数据冗余量太大，而且将表合并后，会丢失一些实体之间关联的重要信息。

对于Teradata来说，它是针对大型数据的分析和处理而设计的，具有非常好的并行处理能力，因此在执行表的连接这类复杂操作时能表现出非常好的性能。我们可以通过一些实际的练习来体会这一点。

表的连接操作分成两种，即内连接(Inner Join)和外连接(Outer Join)。本章只讨论内连接操作，外连接留待后面专门介绍。

10.1 内连接基本介绍

对于内连接来说，答案集是由来自两个或更多表的字段组成，并且各个表参与连接的字段必须匹配。很多情况下都要用到表的连接，如针对试验数据库提出下面的问题：列出所有员工的编号、姓和所在部门的名字。这个问题就要使用到表的

内连接操作。图10-1表示了表EMPLOYEE和表DEPARTMENT按照部门编号DEPARTMENT_NUMBER进行连接的例子，答案集分别由来自两个表的EMPLOYEE_NUMBER、LAST_NAME和DEPARTMENT_NAME组成。这实际上就是我们刚刚提出的问题。

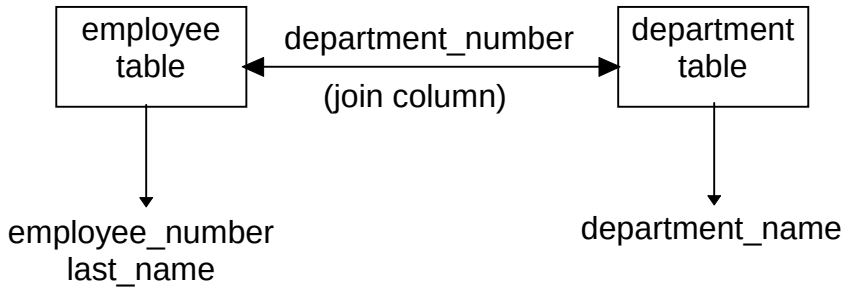


图10-1 EMPLOYEE表与DEPARTMENT表连接示意图

相应的SQL语句是：

```
SELECT  employee.employee_number
   ,employee.last_name
   ,department.department_name
FROM     employee
            ,department
WHERE  employee.department_number = department.department_number;
```

事实上，如果按照标准的内连接操作语法，上面的SQL语句应该是下面的形式：

```
SELECT  employee . employee_number
            ,employee . last_name
            ,department . department_name
FROM     employee
```

```
        INNER JOIN  department
ON      employee . department_number = department . department_number;
```

由此可见，连接条件应该定义在ON子句中。

10.2 别名的定义和使用

在进行连接操作时，经常要使用别名。例如上节中的SQL语句可以写成下面的形式：

```
SELECT  e.employee_number
        ,e.last_name
        ,d.department_name
FROM    employee      e
        ,department   d
WHERE  e.department_number = d.department_number;
```

这里EMPLOYEE的别名是E，而DEPARTMENT的别名是D。由此可见，所谓别名，是表或者视图的一个临时名字，它必须在FROM子句中加以定义。对于名字很长的表或者视图，使用别名是很方便的。另外，当一个表与它自己进行连接，即所谓的自连接(Self Join)时，一定要使用别名。一旦定义好别名，在此SQL语句中就必须使用它。

如果使用别名，并用ON子句来定义连接条件，那上面的SQL语句可写成下面的形式：

```
SELECT  e.employee_number
```

```
        ,e.last_name
        ,d.department_name
FROM    employee e
        INNER JOIN  department d
ON      e.department_number=d.department_number
;
```

10.3 交叉连接(Cross Join)

在上面讨论的内连接操作中，有一个条件特别要注意：那就是要有连接条件，通过连接条件，把参与连接的各表的相应字段进行匹配，从而得到合适的结果。

如果在一个连接中没有连接条件，那这个连接就变成了交叉连接(Cross Join)，也称为乘积连接(Product Join)。如下面的SQL语句就是一个交叉连接：

```
SELECT  employee.employee_number
        ,employee.department_number
FROM    employee
        ,department
WHERE  employee.employee_number = 1008;
```

这里虽然定义了WHERE条件，但显然两个表EMPLOYEE和DEPARTMENT之间没有任何进行匹配的操作，即没有连接条件。这就是一个交叉连接。如果写成下面的形式，就更清楚了。

```
SELECT  e.employee_number
        ,d.department_number
```

```
FROM    employee e
        CROSS JOIN  department d
WHERE   e.employee_number = 1008;
```

在编写带有连接操作的SQL语句时，要特别注意，因为不小心的话很容易产生交叉乘积。如下面的SQL语句，由于使用别名不当(定义了别名而未完全使用)，就产生了交叉乘积。

```
SELECT  employee.employee_number
        ,d.department_name
FROM    employee e
        INNER JOIN  department d
ON      e.department_number = d.department_number;
```

而下面的SQL语句则是由于使用的连接条件不正确而产生的交叉乘积。

```
SELECT  e.employee_number
        ,d.department_name
FROM    employee e
        INNER JOIN  department d
ON      3 = 3;
```

在这个SQL语句中，连接条件" 3=3 "和两个表没有任何关系，显然是不正确的。

完全没有任何限制的交叉连接称为笛卡尔乘积(Cartesian Product)，如下面的SQL语句就是一个笛卡尔乘积。

```
SELECT  employee.employee_number
```

```
        ,employee.department_number
FROM    employee
        CROSS JOIN  department;
```

笛卡尔乘积耗费系统很多的资源，特别是缓冲空间，因为它返回的结果非常庞大。假设一个表有1000条记录，另一个表有2000条记录，这是两个很小的表，但它们进行笛卡尔乘积后，返回的结果是200万行数据！

一般情况下，这种交叉连接都没有什么实际的意义，没有必要做深入的研究。

10.4 多个表的内连接

前面讨论的都是两个表的内连接操作，当把更多的表进行关联时，就是多表连接了。在典型的数据仓库应用中，经常要考虑各个表之间的关系，从各个表之间来挖取信息，因此常常使用多表连接。一般来讲，参与连接的表越多，所消耗的系统资源也越多，这都要求数据库系统具有非常强的并行处理能力。

多表连接时必须定义足够的连接条件。一般来讲，N个表的内连接操作需要定义N-1个连接条件。如果缺少一个连接条件，就会产生交叉乘积。下面是一个多表连接的例子。

```
SELECT  e.last_name
        ,d.department_name
        ,j.description
FROM    employee e
        INNER JOIN  department d
```

```
ON      e.department_number = d.department_number
        INNER JOIN  job j
ON      e.job_code = j.job code;
```

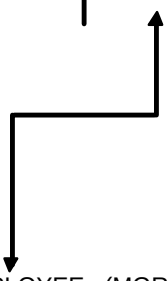
10.5 自连接(Self Join)

在有些情况下，要把一个表和它自身进行连接，这就是自连接(Self Join)。举例来说，要把所有姓Brown的雇员及其经理找出来，可以使用下面的SQL语句。

```
SELECT  emp.first_name      (TITLE 'Emp//First Name')
        ,emp.last_name      (TITLE 'Emp//Last Name')
        ,mgr.first_name     (TITLE 'Mgr//First Name')
        ,mgr.last_name      (TITLE 'Mgr//Last Name')
FROM    employee    emp
        ,department mgr
WHERE emp.manager_employee_number = mgr.employee_number
        AND  emp.last_name = 'Brown';
```

EMPLOYEE (EMP)

EMPLOYEE NUMBER	MANAGER EMPLOYEE NUMBER	DEPARTMENT NUMBER	JOB CODE	LAST NAME	FIRST NAME	HIRE DATE	BIRTH DATE	SALARY AMOUNT
PK	**FK**	**FK**	**FK**					
1006	1019	301	312101	Stein	John	761015	531015	2945000
1024	1005	403	432101	Brown	Allen	790501	540116	4370000
1005	0801	403	431100	Ryan	Loretta	761015	550910	3120000
1002	1003	401	413201	Brown	Alan	760731	440809	4310000
1007	1005	403	432101	Villegas	Arnando	770102	370131	4970000
1003	0801	401	411100	Trader	James	760731	470619	3785000



EMPLOYEE (MGR)

EMPLOYEE NUMBER	MANAGER EMPLOYEE NUMBER	DEPARTMENT NUMBER	JOB CODE	LAST NAME	FIRST NAME	HIRE DATE	BIRTH DATE	SALARY AMOUNT
PK	**FK**	**FK**	**FK**					
1006	1019	301	312101	Stein	John	761015	531015	2945000
1024	1005	403	432101	Brown	Allen	790501	540116	4370000
1005	0801	403	431100	Ryan	Loretta	761015	550910	3120000
1002	1003	401	413201	Brown	Alan	760731	440809	4310000
1007	1005	403	432101	Villegas	Arnando	770102	370131	4970000
1003	0801	401	411100	Trader	James	760731	470619	3785000

产生的输出如下：

Emp	Emp	Mgr	Mgr
First Name	Last Name	First Name	Last Name
Allen	Brown	Loretta	Ryan
Alan	Brown	James	Trader

我们也可以用ON子句来定义连接条件，如：

```
SELECT  emp.first_name        (TITLE 'Emp//First Name')
   ,emp.last_name         (TITLE 'Emp//Last Name')
   ,mgr.first_name        (TITLE 'Mgr//First Name')
   ,mgr.last_name         (TITLE 'Mgr//Last Name')
FROM    employee emp
           INNER JOIN employee mgr
ON         emp.manager_employee_number  =  mgr.employee_number
WHERE  emp.last_name  =  'Brown'
;
```

试想一下，如果把最后的WHERE子句改成：

```
AND emp.last_name  =  'Brown'
```

结果会有什么不同吗？

10.6 子查询(Subquery)与表的连接

在有些情况下，表的连接操作也可以通过子查询来实现。例如，下面采用表连接的SQL语句：

```
SELECT  employee.first_name
           ,employee.last_name
           ,employee.department_number
FROM    employee
           INNER JOIN  department
ON         employee.department_number  = department.department_number
WHERE  department.department_name LIKE '%Research%';
```

和下面采用子查询的SQL语句实现的是同样的功能。

```
SELECT  first_name
        ,last_name
        ,department_number
FROM    employee
WHERE  department_number IN
        (SELECT      department_number
         FROM        department
        WHERE        department_name LIKE '%Research%');
```

那么什么情况下使用子查询，什么情况下使用内连接呢？我们通过图10-2来加以说明。

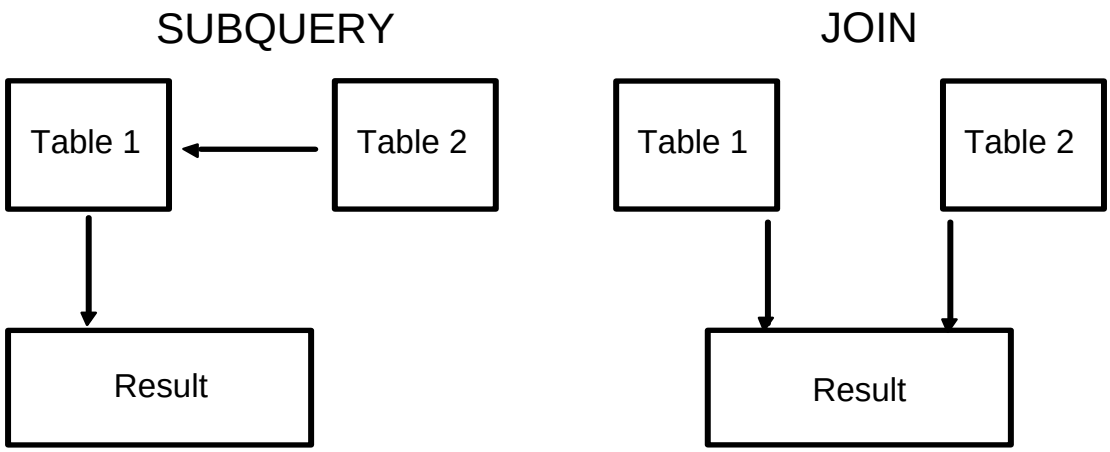


图10-2 子查询与表连接的比较

图10-2左边表示的是子查询，而右边表示的是表的内连接操作。我们可以看到，在子查询中，返回的结果集必须来自一个表(图中的表1)，而另一个表(图中的表2)只是对前一个表的数据作一些限制，看哪些是满足条件的，哪些不满足。如果返回的结果来自两个或更多的表，则必须使用连接操作。至于哪些数据可以返回，必须看连接条件的定义，只有匹配连接条件的数据记录才会返回。

严格说来，连接条件应该在ON子句中加以定义，其它限制条件则在WHERE子句中加以定义。表的连接操作是在缓冲区中完成的，WHERE子句则进一步对连接后的结果进行限制，把那些符合WHERE条件子句的数据记录返回给前端。

练习

Lab 10_1

选择除公司总裁(其部门编号为100)以外的所有雇员的编号、姓、工资、工作描述，将输出进行格式化，使姓不超过9个字符长、工作描述不超过20个字符长，并按工作描述和员工编号进行排序。

Lab 10_2

用表的内连接来完成练习Lab 8-4，用EXPLAIN来观察一下用子查询和用内连接的执行过程，比较一下估计执行的时间，看哪一种方式更有效？

Lab 10_3

查找所有在1978年1月1日以前加入公司的雇员，显示其雇员编号、部门代码以及其部门经理的姓名，将结果按部门编号排序。

Lab 10_4

用EXPLAIN观察一下雇员表和工作表的交叉连接，要求显示工作代码、工作描述和员工姓。

Lab 10_5

在地区代码为213的区域，查找工作代码为412101的所有雇员，显示其姓、部门名、工作描述和电话。要求在连接操作中使用别名，结果按姓排序。

第十一章　数据定义

在Teradata中，数据库是由数据库、用户、表、视图、宏和索引等对象构成的。可以使用SQL语句来创建这些对象、修改已存在对象的定义或者删除对象，完成这些工作的SQL语句称为DDL (数据定义语言)，如表11-1所示。

表11-1 DDL汇总

Databases数据库	
CREATE DATABASE	创建数据库
MODIFY DATABASE	修改数据库定义
DROP DATABASE	删除数据库
Users 用户	
CREATE USER	创建用户
MODIFY USER	修改用户
DROP USER	删除用户
Tables 表	
CREATE TABLE	创建表
ALTER TABLE	修改表定义
DROP TABLE	删除表
Views 视图	
CREATE VIEW	创建视图
REPLACE VIEW	修改视图定义
DROP VIEW	删除视图
Macros宏	

CREATE MACRO	创建宏
REPLACE MACRO	修改宏定义
DROP MACRO	删除宏
Indexes 索引	
CREATE INDEX	创建索引
DROP INDEX	删除索引
Triggers 触发器	
CREATE TRIGGER	创建触发器
DROP TRIGGER	删除触发器
Stored Procedures 存储过程	
CREATE PROCEDURE	创建存储过程
DROP PROCEDURE	删除存储过程

下面将分别介绍这些DDL的使用方法。有关触发器和存储过程的内容在后面章节专门介绍。

11.1 创建表

CREATE TABLE语句创建新表，定义新表的列、索引和其他属性。新表创建后，表结构定义存放在Teradata的数据字典中。CREATE TABLE语句的语法如下：

```
CREATE <SET/MULTISET> TABLE  <Table Name>
   <Create Table Options>
   <Column Definitions>
   <Table-level Constraints>
   <Index Definitions>;
```

其中：

Create Table Options 表选项	定义表的物理属性 Fallback Journaling Freespace Datablocksize
Column Definitions字段定义	定义表的各个字段
Table-level Constraints 表级约束	定义约束 Primary key Unique CHECK条件 Foreign key
Index Definitions索引定义	定义表索引

一个最简单的例子：

CREATE TABLE table1 (field1 integer);

这里创建了一个名为table1的新表，该表只有一个字段field1，其类型是整数(integer)。

如果一个表已不需要，可使用删除表（DROP TABLE）语句将其删除。例如，删除表employee可使用下面的命令：

DROP TABLE employee;

注意，DROP TABLE将删除表中的所有数据，同时从数据字典中删除表的定义。如上例中表　mployee将从数据库中彻底删除，再不可使用。如果只是要删除表中的所有数据，可以使用下面的命令：

DELETE ALL employee;

实际创建表时常常需要定义一些其它选项，下面较为详细地介绍这些可选项。

11.1.1 创建表的可选项(Create Table Options)

Teradata DDL允许在创建表时指定表的物理属性，包括：

- 是否允许重复记录
  - SET　不允许记录重复

    CREATE SET TABLE table1 ...
  - MULTISET 允许记录重复

    CREATE MULTISET TABLE table1 ...
- 数据保护

数据保护要结合FALLBACK和JOURNAL (流水或日志)。

FALLBACK是Teradata的一种数据保护机制，数据表的每一条记录都同时存放两份，而且位于不同的AMP所控制的存储单元中；当数据发生问题或者AMP失败时，可以利用存放在其他AMP上的数据保证对数据表的访问。

- FALLBACK　　使用FALLBACK保护机制
- NO FALLBACK 不使用FALLBACK保护机制

日志有BEFORE和AFTER两种，分别保存了一条记录变化前后的状态。当系统出错时，可以利用日志进行恢复。

存储空间选项

DATABLOCKSIZE用来指定数据块大小，最小的数据块为6144字节，最大的数据块是32256字节。

FREESPACE用来定义在每个磁盘柱面上保留的空间(0-75%)。

例：

```
CREATE MULTISET TABLE table_1
        , FALLBACK, NO JOURNAL
        , FREESPACE = 10 PERCENT
        , DATABLOCKSIZE = 16384 BYTES
   (field1 INTEGER);
```

11.1.2 字段定义

Teradata的表可定义多达256个字段，每个字段的定义包括如下五项：

- 字段名，在同一数据库中必须唯一。
- 字段数据类型，参见 1.1 和 5.1 节关于数据类型的说明。

例：

```
CREATE TABLE emp_data
```

```
(employee_number      INTEGER
,department_number    SMALLINT
,job_code         INTEGER,
,last_name            CHAR(20)
,first_name           VARCHAR(20)
.
.
.
,birthdate        DATE
,salary_amount        DECIMAL(10,2)
.
.
.
```

- 字段数据类型属性。可定义如下属性：

DEFAULT	当字段无数据时用默认值来替代NULL
WITH DEFAULT	用字段的系统默认值替换NULL
FORMAT	缺省的显示格式
TITLE	缺省的列标题
NOT NULL	不允许空值
CASESPECIFIC	字母大小写敏感
UPPERCASE	字母大小写不敏感，内部用大写字母存储

例：

```
CREATE TABLE emp_data
(employee_number      INTEGER      NOT NULL
,last_name            CHAR(20)     NOT NULL WITH DEFAULT
```

```
,street_address        VARCHAR (30)      TITLE "Address'
,city                  CHAR (15)    DEFAULT "Boise'
,state                 CHAR (2)     WITH DEFAULT
,birthdate             DATE         FORMAT"mm/dd/yyyy'
,salary_amount         DEC(10,2)
,sex                   CHAR (1)     UPPERCASE
);
```

- 数据存储属性。包括下面各项：

COMPRESS	压缩值为NULL的字段存储空间为0
COMPRESS NULL	同上
COMPRESS <constant>	压缩值为NULL和指定值的字段存储空间为0

例：

```
CREATE TABLE emp_data
(employee_number       INTEGER
,department_number     INTEGER COMPRESS
.
.

CREATE TABLE bank_account_data
(customer_id       INTEGER
,account_type      CHAR(10) COMPRESS "SAVINGS'
.
.
```

- 字段约束定义。Teradata 支持字段级约束，即限制字段的值满足某些条件，如某个字段取值是否唯一、是否为主键或外键等。对字段的约束总结如下：

CONSTRAINT name	约束名称--可选
PRIMARY KEY	非空,无重复值
UNIQUE	无重复值
CHECK <布尔条件>	指定合法值的范围
REFERENCES	与其他字段的相关性(外键)

例：

```
CREATE TABLE employee_badge
(emp_id INTEGER        NOT NULL
   CONSTRAINT primary_1 PRIMARY KEY
,id_badge_number INTEGER
   CONSTRAINT unique_1 UNIQUE
,salary INTEGER
   CONSTRAINT check_1 CHECK (salary>0)
,job_code INTEGER
   CONSTRAINT ref_1 REFERENCES job (job_code)
);
```

如在上面的例子中，最后一项定义了在EMPLOYEE_BADGE表中的JOB_CODE必须和JOB表中的JOB_CODE对应，即前一个表中该字段的值必须在第二个表中有对应的项。这实际上是一种所谓的参照完整性。另外要注意的是，具有主键(Primary Key)约束的字段一定要定义为非空(NOT NULL)。

11.1.3 表级约束定义

上面讨论了对字段的定义、格式化以及对字段的各种约束。事实上，在Teradata中，在表级也可以定义一些约束。这些约束可以归纳为：

表级约束	意义
唯一性定义 [CONSTRAINT name]	约束名
[UNIQUE]	所指定的多个字段的组合值在表中不能重复
[PRIMARY KEY]	这些列将用作主索引或次索引
参照定义 [CONSTRAINT name]	约束名，配合外键的定义
FOREIGN KEY (<col_list>)	<col_list>所列举的字段为外键，它对应于另一个表(父表)中相同的字段
REFERENCES <tablename> (<primary key list>)	定义父表或引用表中的主键所包含的字段
Check定义 [CONSTRAINT name]	约束名
CHECK <布尔条件>	对表中指定字段的值进行约束

表级约束与字段级约束的主要区别是：在表级约束中可以指定当前表的多个字段或其组合，而字段级约束只能引用当前字段。

我们来看一个表级约束的例子。

```
CREATE TABLE employee_badge
(emp_id          INTEGER NOT NULL
,id_badge_num    INTEGER NOT NULL
,salary          INTEGER
,job_code        INTEGER
,CONSTRAINT primary_1      PRIMARY KEY (emp_id)
,CONSTRAINT unique_1       UNIQUE (id_badge_num)
,CONSTRAINT check_1        CHECK (salary > 0 AND
                   job_code BETWEEN 100000 AND 499999)
,CONSTRAINT ref_1          FOREIGN KEY (job_code)
                        REFERENCES job (job_code));
```

比较字段级约束和表级约束的例子，可以看到：

字段级约束必须写在每个字段定义的后面，而表级约束是在字段定义结束后再进行的。在表级约束中，一个约束可以同时定义多个字段。

如果用SHOW TABLE的命令来观察上面创建的表和它的相应约束，可以看到下面的结果。

```
SHOW TABLE employee_badge;

 *** Text of DDL statement returned.
 *** Total elapsed time was 1 second.

CREATE SET TABLE MJ1.employee_badge ,NO FALLBACK ,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL
    (
     emp_id INTEGER NOT NULL,
```

```
    id_badge_num INTEGER NOT NULL,
    salary INTEGER,
    job_code INTEGER,
    CONSTRAINT check_1 CHECK ( (salary >  0 ) AND  ((job_code >=  100000)
  AND  (job_code <=  499999 )) ),
  CONSTRAINT ref_1 FOREIGN KEY ( job_code ) REFERENCES MJ1.JOB
  ( JOB_CODE ))
UNIQUE PRIMARY INDEX primary_1 ( emp_id )
UNIQUE INDEX unique_1 ( id_badge_num );
```

我们注意到，在原来的DDL中有关主键和UNIQUE的约束在Teradata内部表示为主索引和次索引。

当定义好约束后，可以使用HELP 'Contraint_Name'的方法来观察约束的有关信息。如：

```
help constraint emp_data.emp_key;

             Name  EMP_KEY
             Type  PRIMARY KEY
          Unique?  Y
         Index Id  1
 Column Names  EMPLOYEE_NUMBER
```

有关HELP命令的使用方法已经在前面详细讨论过了。

11.1.4 索引定义

在" Teradata基础教程"一书中，我们谈到其数据分配机制是基于主索引的HASHING算法，因此，Teradata的每个表都必须有主索引。除主索引是必须的外，也可以建立次索引来提高表的存取性能和实现约束。相关的术语如下：

PK	Primary Key 主键
PI	Primary Index 主索引
UPI	Unique Primary Index唯一性主索引
NUPI	Non Unique Primary Index 非唯一性主索引
USI	Unique Secondary Index 唯一性次索引

关于主键PK和主索引PI的作用及其区别已经做了较详细的介绍，这里就不再重复。我们着重来研究一下它的具体定义和使用。

索引可以在CREATE TABLE时就加以定义，同时还可以定义主键。如果创建表时不定义主索引，Teradata就按照下面的规则缺省来建立主索引，因为没有主索引的话，Teradata就无法进行数据的分配。

- 没有在 CREATE TABLE 时指定 PI

IF 定义了PK，THEN PK = UPI

ELSE IF 存在定义为UNIQUE的字段，

THEN 第一个NIQUE的字段为UPI

ELSE 表中定义的第一个字段作为NUPI

- CREATE TABLE 时指定了 PI

IF 定义了PK，THEN PK作为USI

AND为每一个定义为UNIQUE的字段建立一个USI

下面是一个比较复杂的创建表的例子，注意学习。创建该表后用SHOW TABLE观察一下内部的表达方式。

```
CREATE MULTISET TABLE emp_data
,FALLBACK
,NO BEFORE JOURNAL
,NO AFTER JOURNAL
,FREESPACE = 30
,DATABLOCKSIZE = 10000 BYTES (
employee_number INTEGER NOT NULL
,department_number SMALLINT
   CONSTRAINT dep_check
   CHECK (department_number BETWEEN 100 AND 999)
   REFERENCES Department (department_number)
,job_code          INTEGER      COMPRESS
,last_name         CHAR(20)     NOT NULL
,first_name        VARCHAR (20)
,street_address    VARCHAR (30) TITLE 'Address'
,city              CHAR (15)    DEFAULT 'Boise'
                   COMPRESS     Boise'
,state             CHAR (2)     WITH DEFAULT
,birthdate         DATE FORMAT 'mm/dd/yyyy'
,salary_amount     DECIMAL (10,2)
```

```
,sex                CHAR (1)        UPPERCASE
,CONSTRAINT emp_key
            PRIMARY KEY (employee_number)
) INDEX (department_number);
```

11.2 删除表

可以使用DROP TABLE语句删除表，该语句将删除表中的所有数据和在数据字典中的表结构定义。

例：

删除前面例子中创建的雇员数据表。

```
DROP TABLE emp_data;
```

删除了表emp_data中的所有数据，并删除了emp_data在数据字典中的定义。如果希望在使用这个表，必须重新创建。

例：

```
DELETE FROM emp_data;
```

或

```
DELETE emp_data;
```

删除了表emp_data中的所有数据。表定义仍然存在，可以增加数据。

11.3 修改表

当一个表已经创建后，可以使用ALTER TABLE语句来修改其定义。表定义的一些属性是不可修改的(如PI)，如果要改变这些属性，常用方法是建立一个满足新属性的新表，然后使用Insert-Select语句把数据从原来的表转移到新表，然后再修改新表的名称。

ALTER TABLE完整的语法可参见Teradata SQL手册，这里通过几个例子来加以说明。

例：增加或删除字段

```
ALTER TABLE emp_data
ADD educ_level CHAR(1), ADD insure_type SMALLINT;

ALTER TABLE emp_data
DROP educ_level, DROP insure_type;
```

例：修改已有字段的属性

```
ALTER TABLE emp_data
   ADD birthdate FORMAT 'mmmBdd,Byyyy";
```

例：对没有FALLBACK的表建立FALLBACK保护

```
ALTER TABLE emp_data, FALLBACK;
```

例：同时修改表的多个属性

```
ALTER TABLE emp_data
   , NO FALLBACK
     DROP insure_type
   , ADD educ_level CHAR(1);
```

例：修改约束定义

增加约束

```
ALTER TABLE emp_data
   ADD CONSTRAINT
   CHECK (sex = 'F' OR sex = 'M');
```

修改约束:

```
ALTER TABLE emp_data
   MODIFY CONSTRAINT sal_range
   CHECK ( salary_amount>0 AND salary_amount     < 1000000);
```

注意：表中已有数据如果不符合新的约束条件，约束的增加或修改不能成功。

删除约束：

```
ALTER TABLE emp_data
   DROP CONSTRAINT sal_range;
```

11.4 次索引

前面已经讨论过索引，并且说明，创建表时就应定义主索引，同时也可以定义次索引。事实上，次索引也可以使用单独的CREATE INDEX语句来定义。换言之，主索引只能在CREATE TABLE时定义，而次索引既可以在创建表时定义，也可以使用CREATE INDEX来定义。

例：为雇员表创建下面两个次索引。

为雇员名字建立命名的唯一次索引USI

```
CREATE UNIQUE INDEX fullname (last_name, first_name)
ON emp_data;
```

为工作代码建立非唯一性次索引NUSI，不命名NUSI

```
CREATE INDEX (job_code)  ON emp_data;
```

从这个例子可以看到，次索引可以命名，如第一个USI的名字为FULLNAME；也可以不命名，如第二个NUSI就没有取名。

定义好索引或次索引后，可以利用HELP INDEX <表名>来显示指定表的所有索引定义，如果索引是未命名的，索引名称显示为NULL。

例：显示表emp_data上的所有索引。

```
HELP INDEX emp_data;
```

返回结果如下：

Unique?	Y
Primary//or//Secondary?	P
Column Names	employee_number
Index Id	1
Approximate Count	0
Index Name	emp_key

Unique?	N
Primary//or//Secondary?	S
Column Names	department_number
Index Id	4
Approximate Count	0
Index Name	?

Unique?	N
Primary//or//Secondary?	S
Column Names	last_name,first_name
Index Id	8
Approximate Count	0
Index Name	fullname

Unique?	N
Primary//or//Secondary?	S
Column Names	job_code
Index Id	12
Approximate Count	0
Index Name	?

当次索引创建后，也可以利用DROP INDEX来删除它们。注意，只有次索引可以被删除，主索引是不能被删除的。

当删除命名索引时，可以只指定索引名称，也可以指定索引定义。而删除未命名索引时，必须指定索引定义。

例：删除雇员表的所有次索引

删除命名索引

DROP INDEX　FullName ON emp_data;

删除未命名索引

DROP INDEX (job_code) ON emp_data;

练习

Lab 11_1

在自己的数据库空间上创建一个雇员表EMP_NEW，使之与测试数据库中雇员表的定义完全相同。

Lab 11_2

在自己的数据库空间上创建一个部门表DEPT_NEW，使之具有测试数据库中部门表相同的字段，针对部门号定义一个主键而不是主索引，主键名为

Primary_1，删除基于部门名的索引。使用SHOW TABLE来观察一下主键约束是如何内部实现的。然后：

a)修改表dept_new，为部门名增加一个唯一性约束unique_ nm。用SHOW TABLE观察一下该约束是如何实现的。

b)修改表dept_new，为部门代码增加一个CHECK约束，以保证部门代码的值大于99，将此约束命名为hundred_plus。用SHOW TABLE观察一下该约束是如何实现的。

c)修改表dept_new，针对经理的雇员代码(manager_employee_number)增加一个参照约束mgr_ref，使之参照表emp_new的雇员代码。用SHOW TABLE观察该约束的实现方法。

Lab 11_3

用HELP CONSTRAINT命令来观察在Lab 11_2中创建的所有命令约束。

Lab 11_4

修改在Lab 11_1中创建的表emp_new，使得：

1)删除FALLBACK保护

2)增加一个数字(numeric)字段salary_level，其取值范围为3到20。

3)为salary_level建立一个唯一次索引

Lab 11_5

使用HELP INDEX来显示上面刚创建的索引。

Lab 11_6

使用ALTER TABLE来修改表emp_new，使其恢复到Lab 11_4之前的状态。

第十二章　数据操作

数据操作完成对数据库中单个表(或视图)记录的添加、修改和删除，所使用的SQL命令为SELECT、INSERT、UPDATE和DELETE。这些被称为DML (数据操作语言Data Manipulation Language)。由于SELECT在前几章已经讨论过了，本章主要介绍其它三个DML命令的基本内容和使用方法。

12.1 INSERT

INSERT语句用于向表中添加一行或多行记录。插入一行记录的命令格式为：

```
INSERT INTO <表名> (列名1，列名2,...,列名n)
    VALUES (列值表达式1，列值表达式2, ...,列值表达式n);
```

例：在雇员表中添加一新雇员信息：

```
INSERT INTO employee (last_name, first_name, hire_date, birthdate,
salary_amount, employee_number)
VALUES(    arcia',    aria',861027,541110,76500.00,1291);
```

如果添加整条记录，即给每个字段都有相应的值，则表名后的字段名可以省略。如上面的例子可以改写成：

```
INSERT INTO employee
VALUES (1210,NULL,401,41201,    mith',    ames',890303,460421,41000);
```

Teradat对INSERT作了扩充，增加了一个称为INSERT-SELECT的功能。它以子查询的的方式将一个表的数据抽取并插入到另一个表中。举例来说，假设表emp_copy与表emp的结构相同，下面的语句可以把表emp的所有行添加到表emp_copy中，即复制表emp。

```
INSERT  INTO emp_copy
 SELECT * FROM emp;
```

INSERT-SELECT也可以将不同结构表的记录添加到目标表中。例如，我们创建一张雇员生日表：

```
CREATE TABLE birthdays
  (empno INTEGER NOT NULL
, lname CHAR(20) NOT NULL
, fname VARCHAR(30)
, birth DATE)
UNIQUE PRIMARY INDEX(empno);
```

然后，我们从雇员表中提取生日信息添加到生日表中。

```
 INSERT INTO birthdays
      SELECT employee_number ,last_name, first_name, birthdate
      FROM employee;
```

12.2 UPDATE

UPDATE语句用来更新表内满足条件的数据记录，基本语法为：

```
UPDATE  <表名>
      SET  <列名1>=<列值表达式1>
           ,<列名2> = <列值表达式2>
          , ...
           ,<列名n>=<列值表达式n>
      WHERE <条件子句>;
```

如果UPDATE语句中没有WHERE子句，则更新表中的所有记录。

下面的例句是将编号为1010的雇员的部门编号修改为403，工作编号修改为432101，经理的雇员编号修改为1005。

```
UPDATE employee
SET       department_number    = 403
   ,job_code = 432101
   ,manager_employee_number = 1005
WHERE  employee_number = 1010;
```

Teradata SQL允许在WHERE子句中使用子查询和联接，因此同样的工作可以通过子查询或联接来实现。考虑如下的情况，我们要给支援部门的人员加薪10%，如果使用子查询的方式，可以写成：

```
UPDATEemployee
SET       salary_amount = salary_amount * 1.10
WHERE  department_number    IN
```

```
        (SELECT     department_number
        FROM        department
        WHERE       department_name LIKE '%Support%');
```

如果使用联接的方式，可以写成：

```
UPDATEemployee
SET     salary_amount = salary_amount * 1.10
WHERE   employee.department_number =
        department.department_number
AND     department_name LIKE '%Support%';
```

12.3 DELETE

DELETE删除表中满足条件的记录，基本语法为：

```
DELETE FROM<表名>
WHERE  <条件子句>;
```

如果DELETE语句中没有WHERE语句，则删除表中的所有行。

例：删除雇员表中编号为301的员工：

```
DELETE FROM employee WHERE department_number = 301;
```

删除雇员表的所有数据，可使用：

```
DELETE FROM  employee;
```

与UPDATE语句相似，DELETE语句也支持子查询和联接操作。假设在雇员表中，删除所在部门名称为'None'的所有雇员信息，使用子查询的语句为：

```
DELETE FROM employee
   WHERE        department_number    IN
            (SELECT       department_number
             FROM         department
            WHERE         department_name = 'None');
```

使用连接的语句为：

```
DELETE FROM employee
   WHERE employee.department_number=
              department.dempartment_number
          AND department.department_name = 'None';
```

12.4 交易完整性

我们在这章介绍了对数据库记录的更新、插入和删除。试想一下，如果某个数据库的更新操作在进行到一半时系统产生问题，如突然停电等，交易的完整性是否能得到保证？

在Teradata中，系统将保证一个交易的完整。怎样才算是一个交易呢，在Teradata中，根据其所处方式的不同在处理时也有所不同。在Teradata缺省模式下，

以分号结束的每个SQL语句都是一个完整的交易，也可以使用BT (Begin Transaction)和ET (End Transaction)来显示地定义一个交易。下面看一个例子：

例：

缺省方式

```
.LOGON
INSERT row1; (txn #1)
INSERT row2; (txn #2)
.LOGOFF
```

用BT和ET显示定义交易

```
.LOGON
BT;
INSERT row1; (txn #1)
INSERT row2;
COMMIT WORK;
ET;
.LOGOFF
```

第一部分中有两个SQL语句，用分号结束，表示两个交易，任何一个失败不会影响另一个的执行。而第二部分用BT和ET显示地规定：在BT和ET之间的所有SQL是一个交易，只有最后的COMMIT WORK执行成功后，才会真正地更新数据库。执行过程中任何一个SQL语句失败，都会使整个交易失败，系统将自动进行恢复(Rollback)处理。

在ANSI方式下，必须进行显示地提交才能完成一个交易。换言之，执行多个数据记录插入动作后，如果不显示提交就退出，则这些插入动作都将Rollback。

```
.LOGON
```

INSERT row1; (txn #1)

INSERT row2;

.LOGOFF

没有显示提交就退出，两个INSERT将Rollback

.LOGON

INSERT row1; (txn #1)

INSERT row2;

COMMIT WORK;

.LOGOFF

显示提交，两个INSERT作为一个交易，要么完全成功，要么两个都失败。

练习

Lab 12_1

利用INSERT-SELECT将雇员表中的所有记录复制到在Lab 11_1中创建的表emp_new中。

Lab 12_2

编写SQL代码来完成下面的工作：

1)从Lab 12_1创建的表中选择所有属于301号部门的雇员，显示其雇员代码、姓、工作代码和薪水

2)将301号部门所有员工的薪水调高15%。

3)注意验证调薪前后的差别

Lab 12_3

编写代码，恢复Lab 12_2前的状态，即不调薪。

Lab 12_4

利用INSERT-SELECT和视图Department将数据复制到表dept_new中，这样的操作能成功吗？为什么？

Lab 12_5

在Lab 12_4的练习中增加一个WHERE子句，将那些引起操作失败的记录去掉。

Lab 12_6

尝试增加下面的记录到dept_new表中，哪些能成功，哪些会失败，为什么？

Dept #	dept name	budget	manager
99	'new dept'	900000	1021
400	'education'	900000	1021
400	'new dept'	900000	1099
400	'new dept'	900000	1021

第十三章　参数宏

虽然宏不是ANSI标准支持的，但大部分RDBMS都支持宏。在Teradata中，在ANSI和BTET缺省模式下都可以创建和执行宏，只不过在ANSI模式下会给出警告信息。第七章中我们介绍了宏的初步知识，Teradata支持参数宏，通过参数宏可以实现更加灵活的功能。

13.1 简单参数宏

所谓参数宏，是指在宏中包含可以替代值的变量。下面是一个简单的参数宏定义：

```
CREATE MACRO dept_list(dept INTEGER)
AS
( SELECT        last_name
  FROM          employee
    WHERE       department_number = :detpt );
```

该宏的功能是在雇员表中选取某个部门全部雇员的姓，宏dept_list定义了一个参数dept，类型是整数。作为部门代码参数。

运行宏dept_list的语句为：

```
EXEC dept_list(301);
```

其结果是返回部门编号为301的所有雇员的姓。

如同这个简单的例子，参数在宏中的引用是通过冒号(:) +参数名而实现的。

13.2 多参数宏

参数宏可以包含多个参数，每个参数可以定义各自的类型和属性。我们通过参数宏new_dept来介绍多参数宏的创建和运行。

宏new_dept的功能是向部门表添加一行数据，每个字段的值通过参数传递；然后，显示添加的部门信息。具体的宏定义如下：

```
CREATE MACRO       new_dept
          (dept    INTEGER
          ,budgetDEC(10,2)     DEFAULT     0
          ,name   CHAR(30)
          ,mgr     INTEGER)
AS
(   INSERT INTO department
                  (department_number
                  ,department_name
                  ,budget_amount
                  ,manager_employee_number)
          VALUES( :dept
                    ,:name
                    ,:budget
                    ,:mgr);
```

```
SELECT  department_number    (TITLE 'number')
        ,department_name     (TITLE 'name')
        ,budget_amouunt      (TITLE 'budget')
        ,manager_employee_number (TITLE 'manager')
   FROM department
   WHERE       department_number = :dept;
);
```

运行宏new_dept时有两种传递参数值的格式：一种是按照参数定义顺序来传递参数值，另一种是利用参数名称来传递参数值。

按照参数定义顺序传递参数值时必须遵守以下规则：

- 参数值的顺序必须与宏中参数表的定义顺序严格相同
- 参数值的个数与定义严格相同
- 允许参数值为空（NULL）
- 使用逗号（,）或 NULL 作为没有指定值的参数值

例：

```
EXEC new_dept (505,610000.00, 'Marketing Research', 1007);
```

其结果为：

Number	Name	Budget	Manager
505	Marketing Research	610000.00	1007

例：

```
EXEC new_dept (102 , , 'Payroll', NULL);
```

其结果为：

Number	Name	Budget	Manager
102	Payroll	.00	?

当利用参数名称来传递参数值时可以不考虑参数的定义顺序和参数个数。下面来看一个例子：

```
EXEC new_dept ( name = 'accounting'
                ,budget= 425000.00
                ,dept = 106 );
```

其结果为：

Number	Name	Budget	Manager
106	accounting	425000.00	?

13.3 利用宏实现参照完整性

在宏中同样可以利用约束条件来保证参照完整性，关于约束在第十一章中已经介绍了，这里用一个例子来说明。

```
CREATE MACRO new_employee
 ( number INTEGER
   ,MGR  INTEGER
   ,dept    INTEGER
   ,job     INTEGER
   ,lastname        CHAR (20)
   ,firstname VARCHAR (30)
```

```
   ,hired   DATE
   ,birth   DATE
   ,salary DECIMAL (10, 2))
 AS
(ROLLBACK WORK `Invalid Hire'
 WHERE (:hired - :birth) / 365 < 21;
 ROLLBACK WORK `Invalid Department'
 WHERE :dept NOT IN
            (SELECT department_number
            FROM department
            WHERE department_number = :dept);
 ROLLBACK WORK `Invalid Job Code'
 WHERE :job NOT IN
            (SELECT job_code
            FROM   job
            WHERE  job_code = :job);
 INSERT INTO employee
   ( employee_number
     ,manager_employee_number
     ,department_number
     ,job_code
     ,last_name
     ,first_name
     ,hire_date
     ,birthdate
    ,salary_amount )
 VALUES
   ( :number
   , :mgr
   , :dept
   , :job
```

```
   , :lastname
   , :firstname
   , :hired
   , :birth
   , :salary );
);
```

利用这个宏来录入新雇员时，必须满足如下条件：

- 受雇时应年满 21 岁
- 应有一个合法的部门编号
- 应有一个合法的工作代码

实际上，在创建表时也可以定义参照完整性(约束)，如：

```
CREATE TABLE employee
   (employee_number     INTEGER
   , ......
   ,salary_amount          DECIMAL (10,2)
   ,CHECK (hire_date - birthdate) /365 < 21)
   ,FOREIGN KEY (department_number)
            REFERENCES department (department_number)
   ,FOREIGN KEY (job_code)
           REFERENCES job (job_code)
   );
```

练习

Lab 13_1

创建一个参数宏，向表dept_new中插入数据记录，并显示结果以便验证。使用该参数宏插入下面三条记录。

Department_number	department_name	department_number	manager_emp_no
601	shipping	800,000	?
701	credit	1,200,000	1018
905	president staff	5,000,000	801

Lab 13_2

重新执行13_1的宏，在EXECUTE语句前加上EXPLAIN来观察宏的执行过程。

Lab 13_3

写一个参数宏，首先显示然后更新雇员表，使某个指定部门(用参数指定)的所有雇员有一个固定数字(用参数指定)的加薪。显示时只显示雇员代码、姓和两个表示薪水的列：一列标题为Current Salary，另一列标题为New Salary。薪水栏要加上美元标识符，对姓只显示10个字符，并按姓排序。执行该宏，使得401部门的所有员工加薪$1000。然后再执行一次，使得该部门所有员工降薪$1000。

Lab 13_4

写一个参数宏来创建一个报表，显示指定薪水范围(用参数指定)内所有雇员的姓、部门名称和薪水。执行该宏，找出薪水在$50000和$100000之间的所有员工。然后再找出薪水在$30000以下的员工。

第十四章　分组与聚合

本章介绍Teradata SQL中数据分组聚合函数的功能和使用方法。

14.1 聚合函数

聚合操作用来完成对一组指定数据进行聚合计算，完成聚合计算的函数主要有：

- MIN　求最小值，计算中忽略空值。
- MAX　求最大值，计算中忽略空值。
- SUM　求合计，计算中忽略空值。
- COUNT 返回个数，计算中包括空值。
- AVG　求平均值，计算中忽略空值。

例：显示所有雇员的人数、薪水总计、薪水平均值、最高薪水和最低薪水。

```
SELECT
    COUNT ( salary_amount )    (TITLE `COUNT')
   ,SUM  ( salary_amount )     (TITLE `SUM//SALARY')
   ,AVG  ( salary_amount )     (TITLE `AVG//SALARY')
   ,MAX  ( salary_amount )     (TITLE `MAX//SALARY')
   ,MIN   ( salary_amount )    (TITLE `MIN//SALARY')
FROM    employee;
```

结果：

COUNT	SUM SALARY	AVG SALARY	MAX SALARY	MIN SALARY
6	213750.00	35625.00	49700.00	29250.00

14.2 Group By

利用GROUP BY和聚合函数可以实现分组累计。举例来说，如果要求显示各个部门的薪水合计，可以使用下面的语句。

```
SELECT  department_number
        ,SUM (salary_amount)
FROM    employee
GROUP BY    department_number;
```

结果：

department_number	Sum(salary_amount)
401	74150.00
403	80900.00
301	58700.00

需要注意的是，在SELECT子句中不作分组累计的所有字段必须出现在GROUP BY子句中，否则会返回如下出错信息：

ERROR: 3504 Selected non-aggregate values must be part of the associated group.

以上例来说，Department_Number字段未作累计，因此它必须出现在GROUP BY子句中。这条基本规则必须牢记。

14.3 WHERE子句和GROUP BY子句

WHERE子句和GROUP BY子句同时使用时，GROUP BY只对符合WHERE限制的数据记录进行分组聚合计算。换言之，在作真正的聚合计算之前，WHERE子句将不符合条件的数据记录剔除了。

例：部门401和403的合计薪水是多少？

```
SELECT  department_number
        ,SUM (salary_amount)
FROM    employee
WHERE  department_number IN (401, 403)
GROUP BY       department_number
;
```

结果：

department_number	Sum (salary_amount)
403	80900.00
401	74150.00

14.4 GROUP BY和ORDER BY

在GROUP BY后加上ORDER BY，可以使得分组统计按照指定的秩序来显示。例如，按部门编号顺序显示部门人数、合计薪水、部门最高薪水、部门最底薪水和部门平均薪水，可以使用下面的SQL语句：

```
SELECT  department_number          (TITLE 'DEPT')
        ,COUNT (*)                 (TITLE '#_EMPS')
        ,SUM (salary_amount)       (TITLE 'TOTAL')
                                    (FORMAT 'zz,zzz,zz9.99')
        ,MAX (salary_amount)       (TITLE 'HIGHEST')
                                   (FORMAT 'zz,zzz,zz9.99')
        ,MIN (salary_amount)       (TITLE 'LOWEST')
                                   (FORMAT 'zz,zzz,zz9.99')
        ,AVG (salary_amount)       (TITLE 'AVERAGE')
                                   (FORMAT 'zz,zzz,zz9.99')
FROM            employee
GROUP BY        department_number
ORDER BY        department_number
;
```

结果如下：

DEPT	#_EMPS	TOTAL	HIGHEST	LOWEST	AVERAGE
301	3	116,400.00	57,700.00	29,250.00	38,800.00
401	7	245,575.00	46,000.00	24,500.00	35,082.14
403	6	233,000.00	49,700.00	31,000.00	38,833.33

上面的SQL语句也可以写成：

```
SELECT department_number          AS DEPT
   ,COUNT (*)                     AS #_EMPS
   ,CAST ( SUM (salary_amount)    AS FORMAT 'zz,zzz,zz9.99')
                                  AS TOTAL
   ,CAST ( MAX (salary_amount)    AS FORMAT 'zz,zzz,zz9.99')
                                  AS HIGHEST
   ,CAST ( MIN (salary_amount)    AS FORMAT 'zz,zzz,zz9.99')
                                  AS LOWEST
   ,CAST ( AVG (salary_amount)    AS FORMAT 'zz,zzz,zz9.99')
                                  AS _AVERAGE)
FROM          employee
GROUP BY      department_number
ORDER BY      department_number;
```

由于AVERAGE本身是一个关键词，所以在上面的例子中在它前面加上下划线以便区分。

当对多个字段进行分组统计时，GROUP BY只能产生一个级别的汇总。例如：对部门401和403按照工作代码分组统计薪水。

```
SELECT  department_number
        ,job_code
        ,SUM (salary_amount)
FROM    employee
WHERE  department_number IN (401, 403)
GROUP BY      department_number, job_code
ORDER BY      1, 2;
```

结果：

department_number	job_code	SUM (salary_amount)
401	411100	37850.00
401	412101	107825.00
401	412102	56800.00
401	413201	43100.00
403	431100	31200.00
403	432101	201800.00

从这个例子可以看到，当GROUP BY中有多个字段时，它只能产生一个级别的汇总，而且是按照最后一个字段(这里是job_code)来进行汇总。

14.5 GROUP BY和HAVING条件限定

HAVING条件子句是和GROUP一起使用的，用来对分组统计的结果进行限定，只返回满足其条件的分组统计结果。

举例来说，按部门编号顺序显示部门人数、合计薪水、部门最高薪水、部门最低薪水和部门平均薪水，条件是只显示部门平均薪水小于36000的部门。

```
SELECT  department_number          (TITLE 'DEPT')
        ,COUNT (*)                 (TITLE '#_EMPS')
        ,SUM (salary_amount)       (TITLE 'TOTAL')
                                   (FORMAT 'zz,zzz,zz9.99')
        ,MAX (salary_amount)       (TITLE 'HIGHEST')
                                   (FORMAT 'zz,zzz,zz9.99')
        ,MIN (salary_amount)       (TITLE 'LOWEST')
                                   (FORMAT 'zz,zzz,zz9.99)
        ,AVG (salary_amount)       (TITLE 'AVERAGE')
```

```
                                        (FORMAT 'zz,zzz,zz9.99')
FROM            employee
GROUP BY        department_number
HAVING AVG (salary_amount) < 36000;
```

结果：

DEPT	#_EMPS	TOTAL	HIGHEST	LOWEST	AVERAGE
401	7	245,575.00	46,000.00	24,500.00	35,082.14

14.6 GROUP BY小结

在进行分组聚合操作时，要特别注意以下各点：

- WHERE：用来限定参与分组聚合运算的表的数据记录，只有满足条件的数据记录才会被选中参与分组聚合。
- GROUP BY：将符合 WHERE 条件子句的记录进行分组
- HAVING：用来限定可以返回的分组聚合的结果
- ORDER BY：用来指定结果的输出顺序

练习

Lab 14_1

创建一个下面格式的报表，每个部门员工的薪水将向上调整10%，显示每个部门的最低和最高薪水，按部门编号排序，部门名限定为9个字符，薪水前的零去掉并加逗号。

Dept Nbr	Dept Name	Minimum Salary	Maximum Salary	Current Salary	Adjusted Salary
.	.	.	.	.	.

Lab 14_2

创建一个报表，显示所有的部门编号以及每个部门内的工作代码，并统计有工作的员工人数和每个工种的平均薪水。按照部门和工作代码来排序。

第十五章　总计与小计

本章主要学习Select语句中的总计(totals)和小计(subtotals)功能。通过本章的学习，我们将能够：

- 利用 WITH BY 语句创建子计(subtotals)
- 利用 WITH 语句建立最后的总计(totals)
- 利用 DISTINCT 选项剔除重复结果

15.1 利用WITH BY进行数据小计

WITH BY的主要特点包括：

- 它为明细数据表创建分类小计。
- 跟 GROUP BY 不同的是，WITH BY 没有剔除明细记录，而是在明细记录后面按照分类增加小计行。
- 可以允许多于一个字段进行小计，即小计当中可以嵌套小计。
- 输出结果将根据 BY 后面的所有字段自动进行排序。
- 它是 Teradata 的一个扩展特性。

请看下面的一些例子。为了说明方便，在下面的诸例子中均利用了雇员表。

1.利用WITH BY产生基本报表。

例如在Employee表中，欲显示所有雇员姓和工资并要求按部门进行小计。其语句如下：

```
SELECT  last_name              AS NAME
        ,salary_amount         AS SALARY
        ,department_number     AS DEPT
FROM    employee
WITH    SUM (salary)  BY DEPT;
```

返回的报表如下所示：

```
NAME    SALARY      DEPT
Stein   29450.00    301
Kanieski 29250.00   301
   --------------
   Sum(SALARY)58700.00

Johnson  36300.00   401
Trader   37850.00   401
   --------------
   Sum(SALARY)74150.00

Villegas 49700.00   403
Ryan     31200.00   403
   --------------
   Sum(SALARY) 80900.00
```

2. WITH BY多聚合计算

如欲显示雇员名和工资，并要求按照部门进行小计和计算平均值。其语法如下：

```
SELECT  last_name              AS NAME
        ,salary_amount         AS SALARY
        ,department_number     AS DEPT
FROM    employee
WITH     SUM (salary)  (TITLE `Dept Total')
        ,AVG (salary)  (TITLE `Dept Avg  ') BY DEPT;
```

返回的报表格式如下：

```
NAME    SALARY      DEPT
Stein   29450.00     301
Kanieski 29250.00    301
   --------------
   Dept Total 58700.00
   Dept Avg  29350.00

Johnson  36300.00    401
Trader   37850.00    401
   --------------
   Dept Total 74150.00
   Dept Avg  37075.00

Villegas 49700.00    403
Ryan     31200.00    403
   --------------
   Dept Total 80900.00
   Dept Avg  40450.00
```

这里同时显示了部门小计和平均薪水。

3. WITH BY多层小计

利用WITH BY进行多层小计时，越在后面的字段表明它的小计和排序层次越高。看下面的例子：

列出部门401和501所有雇员的编号和工资，要求按照部门进行工资小计，在同一部门中还要求按照工作代码进行小计。其语句如下：

```
SELECT  department_number   AS Dept
        ,job_code           AS Job
        ,employee_number    AS Emp
        ,salary_amount      AS Sal
FROM    employee
WHERE   department_number IN (401, 501)
WITH    SUM(salary_amount) (TITLE 'Job Total') BY Job
WITH    SUM(salary_amount) (TITLE 'DEPT TOTAL') BY Dept;
```

返回结果如下：

```
Dept      Job       Emp       Sal
401     411100      1003    37850.00
                    ----------
              Job Total     37850.00

401     412101      1004    36300.00
```

```
401      412101        1001      25525.00
401      412101        1010      46000.00
------------
                Job Total     107825.00

401      412102        1013      24500.00
401      412102        1022      32300.00
------------
                Job Total      56800.00

401      413201        1002      43100.00
------------
                Job Total      43100.00
------------
               DEPT TOTAL     245575.00
```

15.2 利用WITH语句产生最后的总计

如果在WITH语句中不带BY则只产生总计。如下面的例子：

列出部门301所有的雇员编号和其工资数，并对该部门的工资作统计，语句如下：

```
SELECT  employee_number
             ,salary_amount
FROM     employee
WHERE  department_number = 301
WITH     SUM(salary_amount) (TITLE 'GRAND TOTAL')
ORDER BY       employee_number;
```

返回结果如下：

employee_number	salary_amount
1006	29450.00
1008	29250.00
1019	57700.00

GRAND TOTAL	116400.00

15.3 DISTINCT修饰语

DISTINCT主要用来剔除重复的数据记录。利用DISTINCT修饰语可以用来作一些特定的计算。

下列语句没有使用DISTINCT修饰语：

```
SELECT  employee_number
        ,department_number
        ,manager_employee_number (NAMED manager)
FROM    employee
WHERE  employee_number BETWEEN   1003   AND   1008
WITH    COUNT (manager)    (TITLE 'TOTAL MANAGERS');
```

其结果为：

employee_number	department_number	manager
1006	301	1019
1005	403	801

1003	401	801
1007	403	1005
1008	301	1019
1004	401	1003
TOTAL MANAGERS	6	

如果使用DISTINCT修饰语，如下：

```
SELECT  employee_number
        ,department_number
        ,manager_employee_number  AS manager
FROM    employee
WHERE  employee_number BETWEEN   1003   AND   1008
WITH    COUNT (DISTINCT manager) (TITLE 'TOTAL MANAGERS');
```

则结果为：

employee_number	department_number	manager
1006	301	1019
1006	301	1019
1005	403	801
1003	401	801
1007	403	1005
1008	301	1019
1004	401	1003
TOTAL MANAGERS	4	

从这里可以看到，加上DISTINCT后，重复的数据记录被剔除了。

15.4 进一步的例子

1. WITH BY, WITH和ORDER BY的联合使用

假设要显示所有雇员的姓及其工资，按部门进行小计，得出最后总计，并要求按姓排序，则语句为：

```
SELECT  last_name              AS NAME
        ,salary_amount         AS SALARY
        ,department_number     AS DEPT
FROM    employee
WITH    SUM (SALARY) BY DEPT
WITH    SUM (SALARY)  (TITLE 'GRAND TOTAL')
ORDER BY      NAME;
```

结果如下：

```
NAME    SALARY      DEPT
Kanieski  29250.00      301
Stein     29450.00      301
   -------------
Sum  (SALARY)   58700.00

Johnson   36300.00      401
Trader    37850.00      401
   -------------
Sum  (SALARY)   74150.00
```

```
Ryan        31200.00        403
Villegas    49700.00        403
   -------------
Sum  (SALARY)   80900.00
   -------------
   GRAND TOTAL    213750.00
```

2. WITH和GROUP BY的联合使用

利用WITH和GROUP BY可以建立最后的总计。看下面的例子：

列出每个部门的工资总计和平均值，最后列出整个公司的工资总额和平均值，则语句如下：

```
SELECT  department_number   (TITLE 'dept_no')
            ,SUM (salary_amount)
            ,AVG (salary_amount)
FROM    employee
GROUP BY        department_number
WITH      SUM (salary_amount)         (TITLE 'GRAND TOTAL')
            ,AVG (salary_amount)         (TITLE '')
ORDER BY        department_number;
```

结果如下：

```
dept_no   SUM (salary_amount)AVG (salary_amount)
301         58700.00                29350.00
401         74150.00                37075.00
403         80900.00                40450.00
 ----------------------   ----------------------
```

GRAND TOTAL 213750.00 35635.00

15.5 WITH BY和WITH总结

WITH BY WITH都是Teradata扩展的特性，ANSI SQL标准不支持。

WITH BY的基本格式为：

格式：WITH 汇总列表 BY 分类列表ASC/DESC

特点如下：

- 为详细数据列表建立总计和小计行
- 汇总列表可以多于一个字段
- 分类列表可以多于一个字段
- 分类列表的列已经隐含了 ORDER BY 功能
- WITH. . . BY 后的相应字段决定了主要的排序键
- 一个 SQL 语句可以有多个 WITH BY 子句
- 第一个 WITH BY 排序层次最低
- ORDER BY 用来指定其它附加的子排序，指在某个字段内的进一步排序，如部门内再按薪资高低来进一步排序。

WITH的基本格式为：

格式：WITH 汇总列表

WITH的特点如下：

- 提供最终或总体的总计
- 汇总列表可以多于一个字段

练习

Lab 15_1

这个练习与Lab 8_2类似。利用表location来找出位于California州的所有客户，列出客户姓名(从客户表中来)、编号和销售代表编号。按销售代表编号排序，并在最后累计客户个数，报表格式如下：

customer_name customer_number sales_employee_number

. . .

. . .

. . .

Total Cust X

Lab 15_2

为部门401和403建立一个符合下面格式的报表，但不包含411100、413201和431100三个工种。显示雇员的姓名(限制为10个字符)、部门编号、工作代码和薪水，针对每个部门及部门内各工种进行薪水统计，并在最后进行总计。对每个累结，再加上相应的平均数。在部门和工种内再按姓进行排序。

第十六章　集合操作

集合操作主要包括：合并操作(UNION)、相交操作(INTERSECT)和排外操作(EXCEPT)

Teradata的集合操作与标准ANSI集合操作的不同之出在于返回结果的重复记录处理上。在ANSI标准中集合操作将重复记录自动剔除，而Teradata增加了ALL关键词，ALL关键词允许保留重复记录。

16.1 集合操作的定义

- 相交操作

返回在每一个SELECT语句结果中都存在的记录。可由下图16-1形象表示：

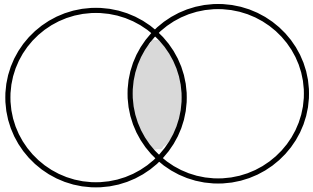


图16-1 INTERSECT操作

INTERSECT用来连接两个SELECT查询，这两个查询所返回的数据集必须具有相同的字段数和数据类型。经相交操作后，只有在两个查询中完全相同的数据行才会返回。举例来说，如果第一个查询返回的记录为A、B、C，第二个查询返回的记录为B、C、D，则相交后的结果为B和C。

- 合并操作

合并所有SELECT语句的结果，重复记录在结果集中只显示一次。如下图16-2所示。

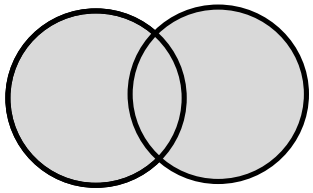


图16-2 UNION操作

UNION连接两个或更多的查询，虽然每个查询的字段名可以不相同，但必须具有相同的字段数和数据类型。

- 排它操作：

返回第一个SELECT语句结果中除第二个SELECT语句结果中以外的所有记录。如图16-3所示。

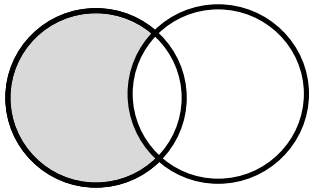


图16-3 EXCEPT操作

同样，两个SELECT查询所返回的字段名可以不同，但数目和数据类型必须一致。在有些数据库系统中，排它操作也称为相减操作(MINUS)。

16.2 合并操作

利用UNION也可以实现外连接(参看第十九章 外连接)。

这里总结一下使用UNION的基本规则：

1. 所有的SELECT语句：

- 必须要有同样多的表达式数目
- 相关表达式的域必须兼容

2. 第一个SELECT语句：

- 决定输出的格式(FORMAT)
- 决定输出的标题(TITLE)

3. 最后一个SELECT语句：

- 包含整个结果集的 ORDER BY 选项(如果有的话)
- ORDER BY 后面的列最好用数字顺序表示

例：谁是经理1019并且谁为他工作？

```
SELECT  first_name
        ,last_name
        ,'employee' (TITLE 'employee//type')
FROM    employee
```

```
WHERE  manager_employee_number = 1019
UNION
SELECT  first_name
        ,last_name
        ,' manager '
FROM    employee
WHERE  employee_number = 1019
ORDER BY      2;
```

返回结果如下：

		employee
first_name	last_name	type
Carol	Kanieski	employee
Ron	Kubic	manager
John	Stein	employee

16.3 相交操作

我们通过例子来说明。列出所有拥有下属雇员的部门经理(有些部门经理可能没有下属)。

```
SELECT manager_employee_number
FROM    employee
INTERSECT
SELECT manager_employee_number
FROM    department
ORDER BY      1;
```

结果为：

manager_employee_number
801
1003
1005
1011
1017
1019
1025

16.4 排它操作

例子：列出所有没有下属雇员的部门经理。

```
SELECT manager_employee_number
FROM    department
EXCEPT
SELECT manager_employee_number
FROM    employee
ORDER BY       1;
```

结果为：

manager_employee_number
1016
1099

16.5 关于集合操作的补充规则

我们将有关集合操作的一些补充规则列举如下：

- 在子查询中不能使用集合操作
- 在定义视图时不能使用集合操作
- 不能包含 WITH 或 WITH BY 子句
- 集合操作的优先级为：INTERSECT 第一，其后分别为 UNION 和 EXCEPT，从左到右。可以使用括号改变优先级。
- 每一个 SELECT 语句必须有一个 FROM <表名>的子句
- 每个单独的 SELECT 语句中可以使用 GROUP BY
- Group By 不能用于或影响整个返回结果集
- 重复记录将会抛弃，除非使用 ALL 选项

练习

Lab 16_1

利用UNION操作产生区号为415的所有电话号码列表，而不管这些电话号码是属于一个地方的或是属于某个雇员的。对产生的每一行电话号码记录需指定它的属别(是地方还是雇员)，给出这一列的标题为　wner。结果按电话号码排序，电话号码以电话号码连接符(连线号)进行连接。

Lab 16_2

对部门级经理找出其有效的员工代码。这是一个在部门表与雇员表之间进行的参照完整性检查。先用子查询来完成，然后再用INTERSECT完成。用EXPLAIN观察一下两种方法的执行过程，看哪一种更有效些，为什么？

Lab 16_3

列出雇员所有的工作代码(不考虑有效性)，然后减去有效的工作代码(指在工作表中登记过)，得到无效工作代码表(如果一个雇员的工作代码未在工作表中出现，则为无效)，只显示工作代码即可。要求标题为INVALID JOB CODES，并使用EXCEPT来实现。

第十七章　视图

视图是数据库中非常重要的一个概念，本章主要讨论与视图有关的内容。完成本章学习后，将能够：

- 创建视图
- 利用视图提供安全的数据存取
- 删除和修改视图
- 列出使用视图的几个原因

如图17-1所示，视图是一张虚拟表。它可以是表中数据记录的子集，也可以是表中某几个字段形成的子集。视图可以参照一张或多张基表。

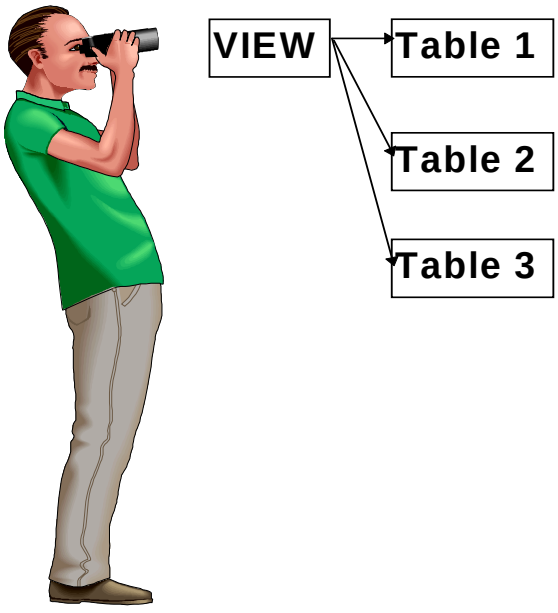


图17-1 视图基本概念

视图并不另外单独存储和复制数据，它只是一个结构。视图定义存储在数据字典中，而不是在用户自己的空间里。

17.1 创建和使用视图

利用CREATE VIEW命令可以创建视图。需要注意的是，视图是基于一些表来实现的，如果这些表的定义发生了变化，将不会反映到视图的定义中。此时将根据需要进行手工的同步更新或维护。

视图定义的例子：从雇员表中，为部门403创建一个视图以利于专门读取和更新记录。要求仅限制于雇员号、姓和工资。语句为：

```
CREATE VIEW emp_403
   AS      SELECT employee_number
                  ,last_name
                  ,salary_amount
           FROM employee
           WHERE department_number = 403;
```

读取此视图的方法和读取普通表无异，如：

```
SELECT *
FROM    emp_403;
```

也可以利用视图来更新记录，如：

```
UPDATE emp_403
```

```
SET       salary_amount = 100000
WHERE  last_name = 'Villegas';
```

17.2 视图中的连接

在视图中可以包含一个或多个表的列，这时需要使用到多表的连接。举例来说，从雇员表和call_employee表中建立一个视图，作为分派电话使用：

```
CREATE VIEW employee_call
   AS      SELECT employee.employee_number
                  ,last_name
                  ,first_name
                  ,call_number
                  ,call_status_code
                  ,assigned_date
                  ,assigned_time
                  ,finished_date
                  ,finished_time
           FROM employee
                  INNER JOIN  call_employee
           ON      call_employee.employee_number
                  = employee.employee_number;
```

如果查询谁是雇员1002，他回的电话有哪些？可以使用下面的方式来从视图得到回答：

```
SELECT  last_name
           ,first_name
           ,call_number
```

```
FROM    employee_call
WHERE  employee_number = 1002;
```

17.3 利用视图重命名列

利用视图可以对列进行重新改名。参考下面的例子：

```
CREATE VIEW shortcut (emp, dept, last, first, sal)
   AS      SELECT employee_number
                  ,department_number
                  ,last_name
                  ,first_name
                  ,salary_amount
           FROM  employee
           WHERE department_number = 201;
```

从视图中选取记录与从普通表无异，这里可以使用已经更名的视图字段，如：

```
SELECT  last
           ,sal
FROM    shortcut
ORDER BY last DESC;
```

利用视图修改记录：

```
UPDATE shortcut
SET       sal = 100000.00
```

```
WHERE  emp = 1019;
```

17.4 改变视图定义

利用REPLACE语句可修改视图定义。如，改变上例中shortcut视图，使之仅包含部门301。我们采用下面的步骤：

1)从数据字典中显示当前的视图定义

```
.EXPORT REPORT FILE = shortcut.ddl
SHOW VIEW shortcut ;
```

2)视图定义被输出到shortcut.ddl文件，内容为：

```
CREATE VIEW shortcut (emp, dept, last, first, sal)
   AS      SELECT employee_number
                  ,department_number
                  ,last_name
                  ,first_name
                  ,salary_amount
           FROM employee
           WHERE department_number = 201;
```

3)利用REPLACE语句改变视图定义：

```
   REPLACE VIEW shortcut (emp, dept, last, first, sal)
```

```
AS      SELECT employee_number
                ,department_number
                ,last_name
                ,first_name
                ,salary_amount
        FROM employee
        WHERE department_number = 301;
```

另外，利用视图还可以对列标题重新命名和对列进行格式化。如下面创建了一个基于部门201的视图，它具有较好的格式化输出。

```
CREATE VIEW   Report AS
SELECT  employee_number     AS Emp  (FORMAT '9999')
        ,department_number  AS dept (FORMAT '999')
        ,last_name          AS Last (TITLE 'Name')
        ,first_name         AS First (TITLE '')
        ,salary_amount / 12 AS Monthly_Salary (FORMAT '$$$,$$9.99')
FROM    employee
WHERE  department_number = 201;
```

视图输出结果：

```
SELECT *
FROM    report
ORDER BY monthly_salary;
```

Emp	Dept	Name	Monthly_Salary
1025	201	Short Michael	$2,891.67
1021	201	Morrissey James	$3,229.17

17.5 聚合视图(Aggregate View)

聚合视图是指视图的列通过聚合计算而得到的视图。

例：建立一个视图依部门汇总工资信息。

```
CREATE VIEW deptsals
AS      SELECT  department_number    AS department
                ,SUM  (salary_amount)   AS salary_total
                ,AVG  (salary_amount)   AS salary_average
                ,MAX  (salary_amount)   AS salary_max
                ,MIN   (salary_amount)   AS salary_min
        FROM  employee
        GROUP BY    department_number;
```

从这个视图的定义可以看出，聚合或派生出来的列必须要给一个名字。

从视图中显示每个部门的平均工资，语句为：

```
SELECT  department
        ,salary_average
FROM    deptsals;
```

department	salary_average
100	100000.00
201	36725.00
403	38833.33
...	...

401 35082.14

... ...

17.6 使用HAVING的聚合视图

利用HAVING子句，修改上述deptsals视图，使之仅包含平均工资小于$36,000的部门，则语句为：

```
REPLACE VIEW deptsals
  AS    SELECT department_number          AS department
              ,SUM  (salary_amount) AS salary_total
              ,AVG  (salary_amount) AS salary_average
              ,MAX  (salary_amount) AS salary_max
              ,MIN   (salary_amount) AS salary_min
        FROM employee
        GROUP BY    department_number
        HAVING AVG(salary_amount) < 36000 ;
```

如欲显示平均工资小于$36,000的部门，可利用视图：

```
SELECT  department
        ,salary_average
FROM    deptsals;
```

department	salary_average
401	35082.14

我们再来看两个例子，下面的视图显示了自1996年以来按客户分类的销售情况。

```
CREATE VIEW cust_prod_sales  (custno, item, sales)
   AS      SELECT customer_number
                  ,item_number
                  ,SUM (item_sales_amount)
           FROM sales_hist_1996
           GROUP BY    1,2;
```

如果要查找哪些客户购买567号商品超过$10000，可以将上面的视图与客户表再进行连接，如下：

```
SELECT  customer_name
           ,sales  (format 'ZZ,ZZ9')
FROM    cust_prod_sales a
           INNER JOIN  customer b
ON         a.customer_number = b.custno
WHERE  a.sales > 10000
           AND    a.item  = 567;
```

17.7 视图的限制和总结

视图的限制：

- 不能基于视图来建立索引，因为视图只是一个定义，本身没有任何数据
- 视图中不能包含ORDER BY 子句
- 派生和聚合的列必须要有一个AS子句指定列名
- 视图不能被UPDATE，如果它包含：

- 数据来自多个表(JOIN VIEW)
- 两次同样的列
- 派生的列
- 包含DISTINCT 子句
- 包含GROUP BY 子句

视图的作用和特点：

- 提供了一个额外的安全、授权层次
- 帮助控制读和修改权利
- 如果基表增加了列，该列对视图无影响。
- 如果列从基表中删除，视图也不受影响，除非删除跟该列相关的视图。
- 简单化了用户存取

根据视图的特点，对于生产系统，我们建议：

- 对每一个基表建立至少一个视图
- 根据需要建立视图查询

练习

Lab 17_1

查询在California州的客户信息，要求显示客户编号、客户姓名和相应的销售雇员代码。按销售雇员代码排序。

使用连接修改查询。

现在，建立一个名为sales_territory的视图。不要把它当作宏的一部分建立。此视图的数据可修改吗？

Lab 17_2

从sales_territory视图中选取所有的列。

Lab 17_3

建立一个名为dept_sal的聚合视图，它可存取雇员表并包含两列：部门号和该部门的工资数。从该视图中选择所有的行。

Lab 17_4

利用视图dept_sal产生一张报表，显示工资总数超过$100,000的部门及其工资数。

Lab 17_5

把数据库customer_service中的所有表建立相应的视图，自己来定义标题和进行输出的格式化，从定义的视图中选择所有的字段和记录，观察输出时的标题和格式。

第十八章　字符串函数

字符串函数不是ANSI标准中支持的，但大部分关系数据库系统都提供了这类函数来进行字符串的操作。本章主要介绍以下字符串函数：

- 利用 SUBSTRING 析取字符串.
- 利用串联符号"||"合并字符串。
- 利用 INDEX 定位字符串的开始位置.

18.1 SUBSTRING函数

SUBSTRING函数用来从字符串中析取一个子字符串，其格式为：

SUBSTRING (<字符串表达式> FROM <开始位置> [FOR <长度>])

如：

SELECT SUBSTRING('catalog' FROM 5 FOR 3);

结果为log。

下面通过一些例子说明了SUBSTRING的使用方法：

SUBSTRING	结果

SUBSTRING('catalog' FROM 5 FOR 4)	log
SUBSTRING('catalog' FROM 0 FOR 3)	ca
SUBSTRING('catalog' FROM -1 FOR 3)	c
SUBSTRING('catalog' FROM 8 FOR 3)	长度为0的字符串
SUBSTRING('catalog' FROM 1 FOR 0)	长度为0的字符串
SUBSTRING('catalog' FROM 5 FOR -2)	error
SUBSTRING('catalog' FROM 0)	catalog
SUBSTRING('catalog' FROM 10)	长度为0的字符串
SUBSTRING('catalog' FROM -1)	catalog
SUBSTRING('catalog' FROM 3)	talog

1、在列表中利用SUBSTRING

例：显示部门403所有雇员姓的第一个字符和姓，按姓排序。

```
SELECT  SUBSTRING (first_name FROM 1 FOR 1) (TITLE 'FI')
            ,last_name
FROM    employee
WHERE  department_number = 403
ORDER BY       last_name;
```

结果为：

FI	last_name
L	Ryan
A	Villegas

2.整型数据类型中使用SUBSTRING

在Teradata数据类型中有三类整型数据类型，它们分别为BYTEINT、SMALLINT和INTEGER。有关这些数据类型的定义在第四章中已经详细介绍过了。

当SUBSTRING函数作用于整型数据类型时，SUBSTRING函数先将整型函数转化为字符串。对于不同的整型数据类型，它们的长度是不一样的，归纳为：

- BYTEINT　加上符号后 4 个字符长
- SMALLINT 加上符号后 6 个字符长
- INTEGER　加上符号后 11 个字符长

当整型数转换为字符类型时，数字向右对齐，不足位补零，最前面为符号位。

例如：下面no. of dependents字段定义为BYTEINT类型；area code字段定义为SMALLINT类型；phone字段定义为INTEGER类型；则它们转换的格式为：

```
+ 1 2 7              (  no. of dependents  ) BYTEINT
+ 0 0 2 1 3          (  area code  )        SMALLINT
+ 0 0 0 5 5 5 4 1 3 4  (  phone  )          INTEGER
```

如欲选取area code中的三位区位码，利用函数：

```
SELECT DISTINCT
        SUBSTRING (area_code FROM 4 FOR 3) AS AREA_CODE
FROM    location_phone
ORDER BY 1;
```

3.在WHERE子句中使用SUBSTRING函数

例：在Location表中找出邮政编码(zip_code)最后四位为零的所有客户。

```
SELECT  customer_number
            ,zip_code    (FORMAT '9(9)')
FROM    location
WHERE  SUBSTRING (zip_code FROM 8 FOR 4) =  '0000';
```

18.2 字符串合并

字符串合并的符号是"||"，它把两个字符串串联成一个字符串。其基本格式为：

```
<字符串1> || <字符串2>
```

注意，BYTE数据类型仅能跟BYTE数据类型进行串联。

例如：’bc’ || ’yz’的结果为’bcyz’

例：利用字符串串联操作，显示部门403所有雇员的姓名，将first name和last name一起显示：

```
SELECT first_name || '   ' || last_name ( TITLE 'EMPLOYEE')
FROM    employee
WHERE  department_number = 403 ;
```

字符串串联与SUBSTRING和TRIM的联合使用举例：

```
SELECT SUBSTRING (first_name FROM 1 FOR 1) || '.  '|| last_name
```

(TITLE 'EMPLOYEE')

```
FROM   employee
WHERE  department_number = 403;
```

结果为：

EMPLOYEE

L. Ryan

A. Villegas

利用TRIM和串联操作删除Last name后面的空格：

```
SELECT TRIM (last_name) || ',  '|| first_name ( TITLE 'EMPLOYEE')
FROM     employee
WHERE  department_number = 403;
```

结果为：

EMPLOYEE

Ryan, Loretta

Villegas, Arnando

18.3 INDEX (字符串定位函数)

INDEX用来在一个字符串中定位一个子串的开始位置。如下面的例子：

```
SELECT INDEX('abc', 'b');                返回结果2
```

```
SELECT INDEX('abc', 'ab');          返回结果1
SELECT INDEX('abc', 'd');           返回结果0
```

又如：在下列所示的Department表中，列出所有部门名包含"SUPPORT"的部门，并指出"SUPPORT"位于该部门名的起始位置。

```
SELECT  department_name
        ,INDEX(department_name,'SUPPORT')
FROM    department
WHERE  INDEX  (department_name,'SUPPORT') > 0
ORDER BY       department_number;
```

返回结果如下：

department_name	Index(department_name, ' SUPPORT')
customer support	10
software support	10

同时使用SUBSTRING和INDEX的例子：

利用CONTACT表显示表中每个人的first name和last name，使用SUBSTRING和INDEX函数的方法为：

```
SELECT
   SUBSTRING (contact_name, FROM INDEX (contact_name,', ') +2)
   || '  ' || SUBSTRING (contact_name    FROM 1
                                  FOR INDEX (contact_name, ',') -1)
   (TITLE 'Contact Names')
FROM    contact;
```

返回结果如下：

Contact Names
Alison Torres
Ginny Smith
Nancy Dibble

练习

Lab 18_1

列出在609区域的所有雇员，包括Last_name和First_name、区号和电话号码。对First_name限制为10个字符。利用表的别名作连接，显示的格式要求如下：

last_name	first_name	Phone Number
Kanieski	Carol	(213) 378-8092

提示：利用SUBSTRING和字符串合并函数解决此问题。

Lab 18-2

在工作表的描述字段中找出所有包含'Analyst'的工作来，显示工作描述和Analyst字符在其中的位置。

Lab 18-3

修改Lab 18_1，将电话分机与电话号码合并起来，如果有些分机为NULL值，则利用COALSCE进行转换，以例整个电话号码栏显示为NULL。按下面的格式输出：

Phone Number Ext
(xxx) xxx-xxxx xxx

Lab 18_4

从CONTACT表中列出所有的CONTACTS，将姓(last name)和名(first name)析取出来各成为单独一列，包含电话号码，只显示408区域的信息，按姓排序。报表格式如下：

CONTACT	TYPE	NULLS	SIZE
contact_number	L	Y	-(10)9
contact_name	CV	N	X(30)
area_code	12	n	-(10)5
phone	I	N	-(10)9
extension	L	Y	-(10)9
last_call_date	DA	N	YY/MM/DD

第十九章　外连接

在数据库中，由于各种原因有可能产生数据的不一致或不完整。如在公司的雇员表中，有可能出现某个雇员没有有效的部门号这种情况。如果提出这样一个问题：产生一个报表，列出所有雇员的部门编号、部门名字和雇员的姓。一个自然的想法是使用下面的SQL语句：

```
SELECT  E.Department_Number
        ,Department_Name
        ,Last_Name
FROM    Employee E
        INNER JOIN  Department D
ON      E.Department_Number =
        D.Department_Number
;
```

仔细考虑一下，如果某个雇员没有有效的部门号(比如由于录入员的打字错误，其部门号在部门表中没有对应的值)或部门号为空值(没有录入)，在上面查询所产生的列表中就不会有他的信息，因为他的部门号不能在部门表中匹配到有效的值。

解决这类问题就必须使用外连接(Outer Join)。本章将详细讨论外连接的种类、使用方法和要注意的地方。

19.1 外连接基础

外连接主要有三种，即所谓的左外连接(Left Outer Join)、右外连接(Right Outer Join)和全外连接(Full Outer Join)。我们用图19-1来说明会更形象些。

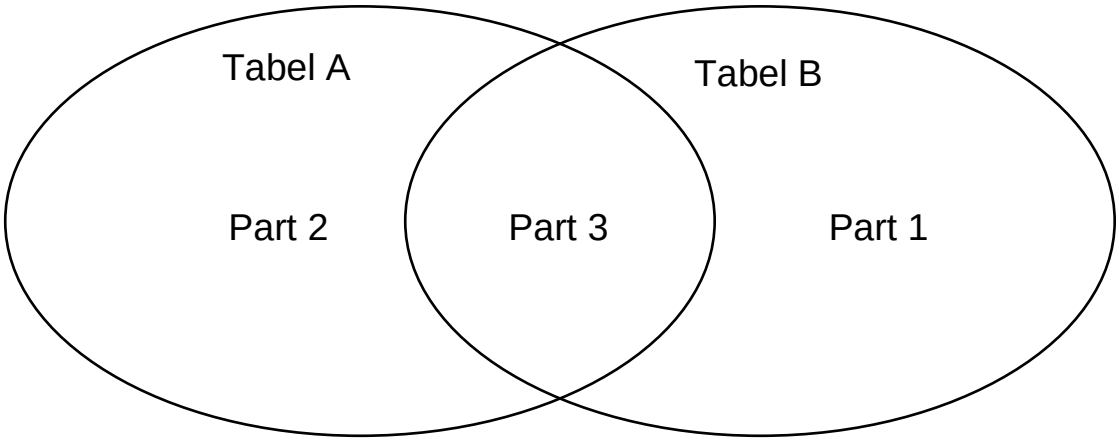


图19-1 内连接与外连接比较

图中表示表A和表B进行连接，显然，如果没有进一步的WHERE条件子句的限制，那传统的内连接操作所选择的就是图中表A和表B的交集所表示的数据，即Part 3。如果要返回Part 2加上Part 3的数据集合，就应该使用表A对表B的左外连接。这时，在返回的数据中，对应表B的字段的值一定是NULL值。同理，如果要返回Part 3加上Part 1的结果，就要使用表A对表B的右外连接；在返回的数据中，对应表A的字段的值也是NULL值。

从图中可以看出，表A对表B的左外连接与表B对表A的右外连接是完全相同的。换言之，交换进行左外连接两个表的位置，就变成了右外连接。所以我们在本章中不专门讨论右外连接。

这里要区分二个概念：即无效的值和空值。以图中的表A (假设是雇员表)和表B (假设是部门表)为例，如果在雇员表中某个雇员的部门号为100，而100这个值在部门表中不存在，它就是一个无效的值(不一致)；如果雇员表中某个雇员的部门号

未输入，它就是一个空值(即NULL)。将雇员表与部门表按部门编号进行左外连接后，这两部分(即无效的部门号或者空值)对应的记录都将返回。在下面的讨论中应该注意这一点。

外连接的完整语法可以描述为：

```
SELECT cname [, cname ,
FROM     tname [aname]
{
LEFT [OUTER]
RIGHT [OUTER]
FULL [OUTER]
}
JOIN      tname [aname]
ON                condition ;
```

其中：

cnmae表示字段名或表达式

tname表示表名

aname表示表的别名

condition表示连接的条件

OUTER可以省略

现在回到本章开始时提出的问题，即找出所有雇员的部门编号、部门名字和雇员的姓。考虑到有些雇员没有有效的部门编号或部门编号为空，可以使用下面的SQL语句：

```
SELECT  E.Department_Number
```

```
        ,Department_Name
        ,Last_Name
FROM    Employee E
        LEFT OUTER JOIN  Department D
ON      E.Department_Number = D.Department_Number;
```

department_number	department_name	last_name
402	software support	Crane
111	?	James
501	marketing sales	Runyon
301	research and development	Stein
?	?	Green
100	executive	Trainer
301	research and development	Kanieski

从上面的结果可以看出，有些雇员没有有效的部门号，有些则没有输入有效的部门名字。如果不使用外连接，是得不到上面的结果的。

19.2 多个表的外连接操作

根据前面的讨论，外连接操作中表的先后顺序很重要。因此，它既不能交换，也不能结合。当两个以上表进行外连接时，必须仔细定义连接的秩序，因此ON子句的正确位置就显得很重要。考虑雇员、部门和工作三个表，如果要知道所有的雇员信息，而不考虑该雇员是否有有效的部门编号或工作代码，可以使用下面这种形式的SQL语句：

```
SELECT  last_name              AS Employee
        ,department_name       AS Dept
        ,description           AS Job
FROM    Department     D
        RIGHT OUTER JOIN Employee E
ON      D.Department_Number = E.Department_Number
        LEFT OUTER JOIN   Job J
ON      E.Job_Code = J.Job_Code;
```

当多个表进行外连接时，处理顺序是按照ON子句的先后位置来进行的，放在前面的ON子句先处理。每处理一个ON子句，即完成一次连接操作，此时实际上是形成了一个临时表，此临时表再与下一个表按照后续ON子句的定义来进行进一步的连接操作。

在这个例子中，首先是部门表(Department)与雇员表(Employee)作一个右外连接，找出具有有效部门编号、部门编号为空以及无效部门编号的所有雇员，形成一个临时表。该临时表包含雇员姓、部门名字(注意SELECT出来的是部门名字，而参与连接的是部门编号！因此，右外连接是针对部门编号进行而非部门名字来进行)和工作代码三个字段。接下来此临时表再与工作表作一个左外连接，将那些没有有效工作代码或工作代码为空值的雇员也包含进来，并显示其工作描述。

如果将上面的SQL语句用括号来标识出连接操作的顺序，则可以写成下面的形式：

```
SELECT  last_name              AS Employee
        ,department_name       AS Dept
        ,description           AS Job
FROM   (Department     D
        RIGHT OUTER JOIN Employee E
ON      D.Department_Number = E.Department_Number  )
```

```
            LEFT OUTER JOIN   Job J
ON          E.Job_Code = J.Job_Code;
```

在多个表参与外连接的情况下，也可以将表的连接操作关系全部定义在FROM子句中，然后将ON条件按照连接的顺序写出来。如将前面的SQL语句写成：

```
SELECT  last_name               AS Employee
            ,department_name    AS Dept
            ,description        AS Job
FROM    Department     D
            RIGHT OUTER JOIN Employee E
            LEFT OUTER JOIN   Job J
ON          D.Department_Number = E.Department_Number
ON          E.Job_Code = J.Job_Code;
```

这样写更清楚些。考虑一下，加上括号后将它写成这样的形式是否可以？

```
SELECT  last_name               AS Employee
            ,department_name    AS Dept
            ,description        AS Job
FROM    Department D
            RIGHT OUTER JOIN
            (Employee E
                        LEFT OUTER JOIN   Job J
ON          D.Department_Number = E.Department_Number )
ON          E.Job_Code = J.Job_Code;
```

如果不行，为什么？

提示，加上括号后正确的写法应该是：

```
SELECT  last_name              AS Employee
        ,department_name       AS Dept
        ,description           AS Job
FROM    Job J
        LEFT OUTER JOIN
        (Department    D
        RIGHT OUTER JOIN Employee E
ON      D.Department_Number = E.Department_Number )
ON      E.Job_Code = J.Job_Code;
```

想想看，为什么？

提示：在进行多表外连接操作时应该记住两点：

1、ON子句的顺序规定了处理连接的顺序

2、每个连接操作形成一个临时表，此临时表再与后续的表时行进一步的连接操作。

练习

Lab 19_1

修改Lab 10_1，把那些具有无效或空的工作代码的员工找出来，要求显示工作代码、工作描述和员工的姓。

Lab 19_2

列出所有具有有效部门编号与工作代码的员工的姓、工作描述和部门名，按姓进行排序，将部门名格式化成13个字符长。

Lab 19_3

修改Lab19_2，将那些具有有效部门编号但没有有效工作代码的员工包含进来。

Lab 19_4

修改Lab19_2，包含所有具有有效部门编号和有效工作代码的员工，同时将那些尚未分配的工作显示出来(即那些还没有人的工作岗位)

第二十章　相关子查询与导出表

本章学习相关子查询(correlated subquery)和导出表(derived table)。

20.1 相关子查询（Correlated Subqueries）

子查询是指在SQL语句限定子句(where)中嵌套的查询语句。子查询分为两种类型：独立子查询和相关子查询。独立子查询是单独执行，其结果参与外层的查询，整个查询中子查询只执行一次，执行完后再执行外层查询；相关子查询是指子查询(内层查询)中引用了外层查询所引用表的字段，因此外层查询处理每一条记录时都必须执行一次子查询，因为子查询中引用的字段的值发生了变化。

例：找出每个部门中薪水最高的人员名字和薪水。

```
SELECT  last_name
        ,salary_amount
FROM    employee ee
WHERE  salary_amount =
        (SELECT MAX (salary_amount)
         FROM          employee em
         WHERE ee.department_number = em.department_number);
```

该子查询语句的执行过程如下：

1.读取雇员表的一条记录

2.执行子查询，得到该雇员所在部门的最高薪水数目

3.比较该雇员薪水和最高薪水

4.如果相等，输出该行信息

5.转到1

再举一个例子：查找薪水高于部门平均薪水的所有员工的名字和薪水。

```
SELECT  last_name
        ,salary_amount (Format "$$$,$99.99")
FROM    employee ee
WHERE  salary_amount >
        (SELECT AVG (salary_amount)
         FROM employee em
         WHERE ee.department_number = em.department_number);
```

利用相关子查询可以完成较为复杂的功能，下面是在IN和EXISTS表达式中使用相关子查询的例子：

哪些代码的工作没有雇员？

```
SELECT job_code
FROM    job
WHERE  job_code  NOT  IN
                (SELECT       job_code
                 FROM         employee) ;
```

如果使用NOT EXISTS表达式，可以写成：

```
SELECT job_code
FROM    job
WHERE  NOT EXISTS (SELECT       *
                   FROM employee ee
                   WHERE ee.job_code = job.job_code);
```

20.2 相关子查询和连接

显示在所属部门薪水最高的经理的雇员编号、部门编号和薪水。

```
SELECT  d.manager_employee_number (TITLE 'mgr_emp_# ')
        ,d.department_number
        ,e.salary_amount
FROM    department d
        INNER JOIN  employee e
ON      e.employee_number = d.manager_employee_number
WHERE  e.salary_amount =
        (SELECT MAX (salary_amount)
         FROM employee em
         WHERE d.department_number = em.department_number);
```

结果：

mgr_emp_#	department_number	salary_amount
1019	301	57700.00
1017	501	66000.00

1011	402	52500.00
1016	302	56500.00
801	100	100000.00

这是一个比较复杂的查询，试着用EXPLAIN来观察一下它的执行过程。

20.3 使用临时表

相关子查询的执行次数依赖于所引用的外部查询的记录行数，实际上我们可以利用临时表来减少子查询的执行次数而提高查询效率。

例：显示所有高于所属部门平均薪水的雇员信息。对此问题，前面已经用相关子查询解决过，它的子查询执行次数为雇员表中的雇员数量。如果雇员数量很多，则对数据库引擎而言是很大的负载。考虑到部门的平均薪水在部门内是固定的，我们可以利用临时表来存储所有部门的平均薪水，从而减少部门薪水的计算次数。具体步骤如下：

1.首先创建一个临时表：

```
CREATE TABLE my_temp (avgsal DECIMAL (10,2));
```

2.在临时表中加入部门平均工资：

```
INSERT INTO my_temp
   SELECT    VG (salary_amount) FROM employee;
```

3.执行查询：

```
SELECT  e.last_name
        ,e.salary_amount        (FORMAT '$,$$$,$99.99')
        ,t.avgsal               (FORMAT "$,$$$,$99.99')
FROM    employee e
        ,my_temp t
WHERE  e.salary_amount>t.avgsal
ORDER BY 2 DESC;
```

4.删除临时表：

```
DROP TABLE   my_temp;
```

查询结果：

last_name	salary_amount	avgsal
Trainer	$100,000.00	$42,386.54
Runyon	$66,000.00	$42,386.54
.	.	.
.	.	.

20.4 导出表

临时表需要用单独的SQL语句来创建和维护，我们也可以使用导出表代替临时表。事实上，导出表就是一个命名的临时表，它在整个SQL语句中自动创建和删除。在定义导出表的整个SQL语句中都可以通过导出表名来引用它。

导出表一般由FROM子句导出表，语法为：

```
FROM (子查询) [AS] <导出表名称> [(列表)]
```

再来看上一节讨论的问题，即找出所有薪水高于公司平均薪水的员工。在上一节我们使用临时表，需要四个步骤。如果使用导出表的方法，可以写成下面的方式：

```
SELECT  last_name
        ,salary_amount          (FORMAT '$,$$$,$99.00')
        ,avgsal           (FORMAT '$,$$$,$99.00')
 FROM   (SELECT AVG (salary_amount)
         FROM employee)     my_temp (avgsal)
        ,employee
WHERE  salary_amount > avgsal
ORDER BY 2 DESC;
```

这里了名为my_temp的导出表，它只有一个字段即平均薪水。这个导出表在整个SQL语句中都可以引用，如在SELECT部分引用了导出表的字段，在WHERE条件子句中也引用了导出表的字段。

研究这个SQL查询，可以得出这样的结论：

- 导出表实际上还是一种临时表，如果把临时表的操作集成在一条 SQL 语句中，就是导出表。
- 导出表在整个 SQL 语句范围内有效。
- 完成指定的 SQL 操作后导出表将自动被删除。

20.5 在导出表中使用分组(GROUP)

在Where子句中不能直接使用聚集函数，如WHERE AVG (salary_amount) > salary_amount。我们可以通过导出表实现Where子句中对聚集函数的引用。

例如，显示所有薪水高于部门平均薪水的雇员信息。

```
SELECT last_name          AS last
   ,salary_amount         (FORMAT '$,$$$,$99.99')  AS sal
   ,department_number AS dep
   ,avgsal       (FORMAT '$,$$$,$99.99')
FROM    (SELECT        AVG (salary_amount)
                  ,department_number
            FROM          employee
            GROUP BY   department_number)
            my_temp (avgsal, deptno)
           ,employee      ee
WHERE  sal       > avgsal
AND       dep      = deptno
ORDER BY 2 DESC;
```

在WHERE子句不能直接使用聚合函数AVG，因此我们把它放在导出表中，由导出表字段AVGSAL来获得联均薪水，再在WHERE条件子句中进行比较。

前边使用相关子查询实现过相同功能，但相关子查询不能显示部门平均薪水而且需多次执行子查询。而使用临时表需要多个单独的步骤。因此在解决这类问题时，使用导出表的效率更高。

练习

Lab 20_1

使用相关子查询来产生一个报表，显示所有薪水高于部门平均薪水的雇员姓、薪水以及部门编号，在部门编号内按薪水排序。

Lab 20_2

使用导出表产生上面相同的报表，并增加一列显示部门平均薪资水平。

Lab 20_3

分别使用NOT IN和NOT EXISTS列出没有雇员的部门的代码清单。

Lab 20_4

修改Lab20_1以便增加一列显示部门名称，要求部门名称限制为12个字符，标题为department，仍在部门编号内按薪水排序。

第二十一章 CASE 表达式

完成本章学习后，可以掌握：

- 利用 CASE 表达式返回可选的值
- 在 CASE 表达式中使用特定的变量
- NULLIF 的使用方法
- COALESCE (接合)

CASE表达式被用来根据搜索的条件返回可选的值。CASE表达式有两种格式：

- Valued (基于值)
- Searched (基于搜索)

这两种格式CASE表达式将在下面进行详细讨论。

21.1 基于值(Valued)的CASE语句

CASE语句允许对返回的数据记录进行条件处理。基于值的CASE表达式的基本格式为：

```
CASE value-expr        WHEN expr1 THEN result1
          WHEN expr2 THEN result2
                        :
          ELSE resultn END
```

它的特点是：

- 对处理的每一条数据记录都将返回一个单独的结果
- 每一行的值决定于每一个 WHEN 子句
- 对于某行的处理首先找到第一个匹配的值并返回结果，如果没有匹配的值，则返回 ELSE 子句后面的值。

例：计算部门401的工资数占整个公司工资总数的比例。

```
SELECT SUM(
   CASE department_number
         WHEN 401 THEN salary_amount
         ELSE 0
   END) / SUM(salary_amount)
FROM employee;
```

结果如下：

```
(Sum(<CASE  expression>)/Sum(salary_amount))
2.22834717118098E-001
```

如需计算部门401和501的工资总额，可以使用下面的SQL语句：

```
SELECT CAST (SUM(
         CASE department_number
               WHEN 401 THEN salary_amount
               WHEN 501 THEN salary_amount
```

```
                ELSE 0
            END) AS NUMERIC(9,2))
                AS total_sals_401_501
FROM EMPLOYEE;
```

返回结果如下：

```
total_sals_401_501
445700.00
```

我们看到，在基于值的CASE语句中，必须在CASE后面指定一个表达式，对此表达式列出几种可能的值来进行匹配。如果没有表达式来计算值，可以使用下面基于搜索的CASE结构。

21.2 基于搜索(Searched)的CASE语句

在基于搜索的CASE语句中，没有必要指定一个表达式来进行计算，将计算的值作为匹配的基础。取而代之的是，可以指定多个、任意的搜索条件，对每一个搜索条件返回相应的值。对于不匹配的条件处理，可以明确地指定一个ELSE子句，或利用缺省的ELSE NULL子句。

基于搜索的CASE语句格式为：

```
CASE    WHEN  condition1 THEN value-expr1
        WHEN  condition2 THEN value-expr2
```

:

ELSE value-expr END

例：

```
SELECT last_name,
CASE
WHEN salary_amount < 30000
   THEN 'Under $30K'
WHEN salary_amount < 40000
   THEN 'Under $40K'
WHEN salary_amount < 50000
   THEN 'Under $50K'
ELSE
    'Over  $50K'
END
FROM employee
ORDER BY salary_amount;
```

返回结果如下：

```
last_name        <CASE expression>
----------       ----------------------------
Phillips         Under $30K
Crane            Under $30K
Hoover           Under $30K
Rabbit           Under $30K
```

再研究一个例子：计算部门401和501占工资总数的比例，允许部门501的员工的工资都增加10%，则语句为：

```
SELECT SUM(
   CASE
   WHEN department_number = 401 THEN salary_amount
   WHEN department_number = 501 THEN salary_amount * 1.1
   ELSE 0
   END) / SUM(CASE WHEN department_number = 501
                        THEN salary_amount * 1.1
                        ELSE salary_amount
                        END) (FORMAT 'Z.99')
   AS sal_ratio
FROM employee;
```

返回结果如下：

```
sal_ratio
.42
```

21.3 NULLIF表达式

NULLIF实际上用来作为CASE语句在某种情况下的缩写，其格式为：

```
NULLIF ( <expression1> , <expression2> )
```

规则是：

- 如果表达式 1 等于表达式 2，则返回 NULL
- 如果表达式 1 不等于表达式 2，则返回表达式 1 的值。

NULLIF等价于：

```
CASE
   WHEN <expression1> = <expression2> THEN NULL
   ELSE <expression1>
END
```

例：

```
SELECT  call_number
            ,labor_hours    (TITLE 'ACTUAL HOURS')
            ,NULLIF (labor_hours, 0)
                        (TITLE 'NULLIF ZERO HOURS')
FROM    call_employee
ORDER BY labor_hours;
```

除法中使用NULLIF

在除法表达式中，如果被除数为零，则系统返回divide by zero错误。如果被除数为NULL，则返回结果也为NULL，但没有错误产生。因此，在除法表达式中，如果被除数有可能为零值，经常使用NULLIF来避免除零错。

例：找出工作描述包含"analyst"项目的每小时收费与每小时成本的比率。

如果没有NULLIF时:

```
SELECT  description
        ,hourly_billing_rate / hourly_cost_rate
                (TITLE 'Billing to Cost Ratio')
FROM    job
WHERE  description like "%analyst%';
```

产生错误：

Error Message: Division by zero in an expression involving job. hourly_cost_rate.

使用NULLIF时:

```
SELECT  description
        ,hourly_billing_rate /NULLIF(hourly_cost_rate, 0)
                (TITLE 'Billing to Cost Ratio')
FROM    job
WHERE  description like '%analyst%';
```

结果为：

description	Billing to Cost Ratio
Software Analyst	1.29
System Support Analyst	?
System Analyst	1.14

21.4 COALESCE(接合)表达式

COALESCE实际上也是CASE语句在某种特殊情况下的宿写。COALESCE将返回第一个非NULL表达式的值。其格式为：

```
COALESCE ( <expression1> , <expression2> [, <expressionX> ] )
```

等价于：

```
CASE
   WHEN <expression1> IS NOT NULL THEN  <expression1>
   WHEN <expression2> IS NOT NULL THEN <expression2 >
  ...
WHEN <expressionX> IS NOT NULL THEN <expressionX>
   ELSE NULL
END
```

例：从phone_table表中，列出姓名和电话号码，如果办公室电话存在则列出办公室电话，否则列出家里电话。

```
SELECT  name
        ,COALESCE (office_phone, home_phone)
FROM    phone_table;
```

例：转换可能的NULL值为零：

```
SELECT  course_name
        ,COALESCE (num_students,0)
```

```
                    (TITLE '# Students')
FROM    class_schedule;
```

结果如下：

course_name	# Students
Teradata SQL	17
Physical DB Design	0

例：转换可能的NULL值，用NULL VALUE来表示。

```
SELECT  course_name
        ,COALESCE (num_students,    NULL VALUE')
                    (TITLE '# Students')
FROM    class_schedule;
```

返回结果如下：

course_name	# Students
Teradata SQL	17
Physical DB Design	NULL VALUE

例：COALESCE和NULLIF在聚合计算中的使用

假设有下面的表：

CALL_EMPLOYEE

CALL NUMBER	EMPLOYEE NUMBER	CALL STATUS CODE	ASSIGNED DATE	ASSIGNED TIME	FINISHED DATE	FINISHED TIME	LABOR HOURS
PK							
FK	**FK**	**FK**					
5	1004	5	861216	1025			
4	1010	1	861215	1250			
1	1004	1	861215	0905	861216	1625	8.5
6	1004	2	861216	1110			
3	1001	16	861215	1215			.0
2	1001	2	861215	0930	861216	1375	4.0

使用下面的SQL语句：

```
SELECT
    AVG  (labor_hours)  (TITLE 'DEFAULT//AVG')
   ,AVG  (NULLIF ( labor_hours, 0)) (TITLE 'NIZ//AVG')
   ,AVG  (COALESCE ( labor_hours, 0)) (TITLE 'ZIN//AVG')
                                      (FORMAT 'zz,zzz.z')
   ,COUNT ( labor-hours)         (TITLE 'DEFAULT//COUNT') (FORMAT   'zz9')
   ,COUNT (NULLIF ( labor_hours, 0)) (TITLE   'NIZ//COUNT')
                                     (FORMAT 'zz9')
   ,COUNT (COALESCE ( labor_hours, 0)) (TITLE 'ZIN//COUNT')
                                      (FORMAT 'zz9')
FROM    call_employee;
```

输出结果如下：

DEFAULT	NIZ	ZIN	DEFAULT	NIZ	ZIN
AVG	AVG	AVG	COUNT	COUNT	COUNT

4.2 6.2 2.1 3 2 6

练习

Lab 21_1

计算部门401和403占整个公司预算的比例，要求输出格式如下：

```
401/ 403
Bgt Ratio
  xx%
```

Lab 21_2

在部门403的预算增加5%以后，计算部门401和403占公司总预算的比例， 要求输出格式如下：

```
Dept_401_403_Bgt_Ratio
 .xx
```

Lab 21_3

从部门表中建立一个预算报表。列出总计、平均、最少和最大的预算数目。标题分别为"Total", Avg", "Min"和 "Max"。

分别用下面两种方法来完成这个问题：

a.将NULL值当零看待

b.将NULL值不包括在汇总计算内

比较两种方法的结果。

Lab 21_4

财务部门作一个报表，统计每个部门的薪水总和，并用下面三种方法计算部门平均薪水：

-不作任何修改，即NULL值和零值都保持不变

-将NULL值转换为零

-将零转换为NULL

要求在报表中列出参与平均薪水计算的行数。按部门排序，报表格式如下：

Dept Tot # Tot Sal Avg Sal ZIN # ZIN Avg NIZ # NIZ Avg

(ZIN表示ZERO IF NULL，NIZ表示NULL IF ZERO)

第二十二章　系统日历

完成本章学习后，将能够：

- 基于系统日历创建视图
- 相对于任何日期查询历史数据
- 基于任何形式的日历获取信息

22.1 系统日历

SQL有非常有限的日期运算功能，对日期运算比较困难。因此，有必要开发更复杂的基于时间的计算工具，这就是开发系统日历的原因。系统日历与用户自己定义的日历相比，最重要的一点是性能提高。

Teradata的系统日历涵盖200年的范围，没有性能问题。因为日历表仅仅按照当前执行的查询物化所需要的实际行数据。

"我想知道....."

- 本季度与去年同季相比怎样?
- 星期日与星期六我们销售了多少鞋?
- 本月的哪一周我们销售的比萨饼最多?
- 本周与去年同周相比怎样?

如果没有系统日历，这些问题都比较难回答。

好处

系统日历的一些好处包括：

- 通过与日历连接，扩展了 DATE 数据类型的特性
- 很容易与其他表连接，就象星型模式(star schema)中的维度表一样
- 高性能 - 日历能够物化在内存中 - 很少的 I/O
- 已经对 Statistics 进行了 Collected，优于用户自定义的日历

标准用法

系统日历与传统表不一样，不需要用户维护。你只需要连接它执行复杂日期运算。Statistics也自动基于物化的表创建。

22.2 日历表的布局

系统日历包含从1900-01-01到2100-12-31的每天的数据，每天在表中都有一行数据。

下面是系统日历可以访问的列：

calendar_date DATE UNIQUE (标准Teradata日期)

day_of_week BYTEINT, (1-7,这里 1 = Sunday)

day_of_month BYTEINT, (1-31)

day_of_year SMALLINT, (1-366)

day_of_calendar INTEGER, (从01/01/1900开始的天)

weekday_of_month BYTEINT, (本月中该星期出现的次数)

week_of_month BYTEINT, (本月中第几周，起始是0)

week_of_year BYTEINT, (0-53) (本年中第几周，起始是0)

week_of_calendar INTEGER, (0-n) (日历中的第几周，起始是0)

month_of_quarter BYTEINT, (1-3)

month_of_year BYTEINT, (1-12)

month_of_calendar INTEGER, (1-n) (从1900年1月)

quarter_of_year BYTEINT, (1-4)

quarter_of_calendar INTEGER, (从1900年1月)

year_of_calendar SMALLINT, (从1900)

系统日历视图

系统日历有一组高性能的视图，包含日历中每一天的详细信息。

日历很容易与其他表连接，获得任何类型的时间点或时间段的信息。基础表包含从1900-01-01到2100-12-31的每一天的数据，每天一行数据，只有一个日期类型的字段。

系统日历实际上有4级嵌套的视图，每级视图都增加了一些智能的日期功能。如果使用SHOW VIEW命令，你可以看见底层的视图，执行另一个SHOW VIEW 命令，可以看见另一个底层的视图。一直下去，最终能够看见底层的基础表。视图执行一些复杂的计算。

系统日历有下列特性：

- 基础表是 Sys_calendar.Caldates
- 它只有一列"cdate"，数据类型是 DATE
- 日历中的每个日期都有一行数据
- 唯一主索引(UPI)是"cdate"
- 每个视图都增加了一些智能的日期功能

22.3 日历中的一行

问题

查看系统日历中的一行数据。假设今天是1998年9月21日。

Sept 1998

S	M	T	W	T	F	S
20	21	22	23	24	25	26

解答

```
SELECT *
FROM Sys_calendar.Calendar
WHERE calendar_date = current_date;
```

结果

```
calendar_date: 98/09/21
day_of_week: 2
day_of_month: 21
day_of_year: 264
day_of_calendar: 36058
weekday_of_month: 3
```

week_of_month: 3

week_of_year: 38

week_of_calendar: 5151

month_of_quarter: 3

month_of_year: 9

month_of_calendar: 1185

quarter_of_year: 3

quarter_of_calendar: 395

year_of_calendar: 1998

注：SELECT CURRENT_DATE是ANSI标准，等同于SELECT DATE。

22.4 使用日历

下面是使用日历的例子：

```
CREATE SET TABLE daily_sales ,NO FALLBACK
                ,NO BEFORE JOURNAL
                ,NO AFTER JOURNAL
                        (itemid INTEGER
                        ,salesdate DATE FORMAT 'YY/MM/DD'
                        ,sales DECIMAL(9,2))
        PRIMARY INDEX ( itemid );
```

问题

显示1998年1季度item 10的销售总额。

解答

```
SELECT ,ds.itemid
      ,SUM(ds.sales)
FROM sys_calendar.calendar sc
    ,daily_sales ds
WHERE sc.calendar_date = ds.salesdate
AND  sc.quarter_of_year = 1
AND sc.year_of_calendar = 1998AND ds.itemid = 10
GROUP BY 1;
```

结果

itemid	Sum(sales)
10	4050.00

22.5 使用今天的视图

系统日历可以用于报告日期：

- 相对于今天
- 相对于从 1900-01-01 到 2100-12-31 中的任何日期

今天的视图

建立一个今天的视图‘Today’是一个普遍采用的技术，可以相对于今天的日期查询日历。通过定义包含一行今天日期的视图，可以写查询获得相对于当前天、周、月、季的信息。'Today'视图创建如下：

```
CREATE VIEW today AS (
        SELECT * FROM sys_calendar.calendar
        WHERE calendar.calendar_date=current_date);
```

该视图显示今天的日历中的所有信息。

下面是使用日历的例子：

```
CREATE SET TABLE daily_sales ,NO FALLBACK
                ,NO BEFORE JOURNAL
                ,NO AFTER JOURNAL
                        (itemid INTEGER
                        ,salesdate DATE FORMAT 'YY/MM/DD'
                        ,sales DECIMAL(9,2))
        PRIMARY INDEX ( itemid );
```

问题

获得item 10在当前月的销售额。

解答

```
SELECT  SUM(ds.sales)
FROM sys_calendar.calendar sc
    ,daily_sales ds
```

```
    ,today  td
WHERE sc.calendar_date = ds.salesdate
AND  sc.month_of_calendar = td.month_of_calendar
AND ds.itemid = 10;
```

结果

```
Sum(sales)
2550.00
```

22.6 查询相对于今天的信息

连接视图"Today"，查询相对于今天的信息。下面是一个典型的例子。

问题

比较item 10在今年和去年的这个月和上个月的销售额。

解答

```
SELECT sc.year_of_calendar AS "year"
  ,sc.month_of_year AS "month"
  ,ds.itemid
  ,sum(ds.sales)
FROM sys_calendar.calendar sc
    ,daily_sales ds
```

```
    ,today  td
WHERE sc.calendar_date = ds.salesdate
AND  ((sc.month_of_calendar BETWEEN td.month_of_calendar - 1
        AND td.month_of_calendar)
OR     (sc.month_of_calendar BETWEEN td.month_of_calendar - 13
        AND td.month_of_calendar - 12))
AND ds.itemid = 10
GROUP BY 1,2,3
ORDER BY 1,2;
```

结果

year	month	itemid	Sum(sales)
1997	8	10	1950.00
1997	9	10	2100.00
1998	8	10	2200.00
1998	9	10	2550.00

22.7 分组结果

现在我们做前面同样的查询，但是按周分隔信息。我们在SELECT语句中增加一列"Week of Month" 。

解答

```
SELECT sc.year_of_calendar AS "year"
  ,sc.month_of_year AS "month"
  ,sc.week_of_month AS "Week of//Month"
```

```
  ,ds.itemid
  ,SUM(ds.sales)
FROM sys_calendar.calendar sc
   ,daily_sales ds
   ,today td
WHERE sc.calendar_date = ds.salesdate
AND  ((sc.month_of_calendar BETWEEN td.month_of_calendar - 1
                AND td.month_of_calendar)
OR     (sc.month_of_calendar BETWEEN td.month_of_calendar - 13
                AND td.month_of_calendar - 12))
AND ds.itemid = 10
GROUP BY 1,2,3,4
ORDER BY 1,2,3;
```

结果

year	month	week_of_month	itemid	Sum(sales)
1997	8	0	10	350.00
1997	8	1	10	600.00
1997	8	2	10	550.00
1997	8	3	10	150.00
1997	8	4	10	200.00
1997	8	5	10	100.00
1997	9	0	10	750.00
1997	9	2	10	1000.00
1997	9	3	10	350.00
1998	8	0	10	150.00
1998	8	1	10	1050.00
1998	8	2	10	550.00
1998	8	3	10	150.00
1998	8	4	10	200.00

1998	8	5	10	100.00
1998	9	0	10	400.00
1998	9	1	10	800.00
1998	9	2	10	550.00
1998	9	3	10	450.00
1998	9	4	10	350.00

22.8 比较相关周

系统日历使用数字0到6表示每月的周。0，如果有的话，表示月的第一个不完整的周；1表示月的第一个完整的周。

问题

显示item 10在上月第一个完整周和去年对应周的销售额。

解答

```
SELECT sc.year_of_calendar AS "year"
  ,sc.month_of_year AS "month"
  ,sc.week_of_month AS "Week of//Month"
  ,ds.itemid
  ,SUM(ds.sales)
FROM sys_calendar.calendar sc
    ,daily_sales ds
    ,today td
WHERE sc.calendar_date = ds.salesdate
```

```
AND       sc.week_of_month = 1
AND   ((sc.month_of_calendar =  td.month_of_calendar - 1)
OR      (sc.month_of_calendar = td.month_of_calendar - 13))
AND ds.itemid = 10
GROUP BY 1,2,3,4
ORDER BY 1,2;
```

结果

year	month	week_of_month	itemid	Sum(sales)
1997	8	1	10	600.00
1998	8	1	10	1050.00

22.9 按星期聚合

按照星期几来汇总数据是很常见的需求。系统日历具有这个能力，使用数字1至7来表示星期，星期日是第一天。

问题

显示item 10在本月和上月所有星期日的销售额，并且日销售额应大于$150。

解答

```
SELECT  sc.calendar_date AS "date"
  ,sc.week_of_month AS "Week of//Month"
```

```
  ,'Sunday'
  ,ds.itemid
  ,SUM(ds.sales)
FROM sys_calendar.calendar sc
   ,daily_sales ds
   ,today td
WHERE sc.calendar_date = ds.salesdate
AND      sc.day_of_week = 1
AND   ((sc.month_of_calendar =  td.month_of_calendar)
OR        (sc.month_of_calendar = td.month_of_calendar - 1))
AND ds.itemid = 10
HAVING SUM(ds.sales) > 150
GROUP BY 1,2,3,4
ORDER BY 1,2;
```

结果

date	Week_of_Month	'Sunday'	itemid	Sum(sales)
98/08/02	1	Sunday	10	200.00
98/09/06	1	Sunday	10	350.00
98/09/20	3	Sunday	10	450.00
98/09/27	4	Sunday	10	350.00

比较周

要比较周，需要使用"week_of_calendar"列。例子聚合了今年和去年本周和上周的信息，当前周可以通过今天视图"today"获得。

问题

显示item 10在本周和上周及去年同期的销售额。

解答

```
SELECT sc.year_of_calendar as "year"
   ,sc.month_of_year as "month"
  ,sc.week_of_month as "week"
  ,ds.itemid
  ,sum(ds.sales)
FROM sys_calendar.calendar sc
    ,daily_sales ds
    ,today td
WHERE sc.calendar_date = ds.salesdate
AND   ((sc.week_of_calendar BETWEEN td.week_of_calendar - 1
                  AND td.week_of_calendar)
OR     (sc.week_of_calendar BETWEEN td.week_of_calendar - 53
                  AND td.week_of_calendar - 52))
AND ds.itemid = 10
GROUP BY 1,2,3,4
ORDER BY 1,2,3;
```

结果

year	month	week	itemid	Sum(sales)
1997	9	2	10	1000.00
1997	9	3	10	350.00
1998	9	2	10	550.00
1998	9	3	10	450.00

使用星期表

如果需要显示星期几的名称，那么应该使用一个星期表。先创建表，并增加数据。以后的查询就可以连接这个表，产生所需的结果。

例：

创建星期表day_of_week，并增加数据。

```
CREATE TABLE day_of_week
(numeric_day INTEGER
,char_day CHAR(9)
)
UNIQUE PRIMARY INDEX (numeric_day);

INSERT INTO day_of_week VALUES (1, 'Sunday');
INSERT INTO day_of_week VALUES (2, 'Monday');
INSERT INTO day_of_week VALUES (3, 'Tuesday');
INSERT INTO day_of_week VALUES (4, 'Wednesday');
INSERT INTO day_of_week VALUES (5, 'Thursday');
INSERT INTO day_of_week VALUES (6, 'Friday');
INSERT INTO day_of_week VALUES (7, 'Saturday');
```

现在就可以使用星期表了。

问题

显示item 10这周每天的平均销售额。

解答

```
SELECT dw.char_day "Day of//  Week"
 ,AVG(ds.sales) avgsal
FROM daily_sales ds, sys_calendar.calendar sc , day_of_week dw
WHERE sc.calendar_date =  ds.salesdate
AND sc.day_of_week = dw.numeric_day
GROUP BY 1;
```

结果

Day of Week	avgsal
Thursday	250.00
Friday	272.22
Wednesday	350.00
Monday	275.00
Sunday	295.00
Tuesday	285.71
Saturday	325.00

练习

注：在练习之前，先创建一个视图"Today"，但不使用今天的日期，使用一个固定日期1998-09-21。确保视图名前有你自己的用户名。

```
CREATE VIEW sqlxxxx.today AS
```

```
( SELECT *
FROM sys_calendar.calendar
WHERE calendar.calendar_date=980921);
```

Lab 22_1

比较item 10在今年第3季度和去年同期的结果。

使用三中方法：

a.) 使用绝对的年和季度值。

b.) 使用相对年来得到上一年。

c.) 使用相对季度来得到上一年。

Lab 22_2

创建一个星期表"day-of-week"，使用英文名称表示星期，如Sunday = 1。

从日历中找出你出生在星期几。

Lab 22_3

按星期显示，Item 10 哪天的平均销售额> $300

Lab 22_4

显示item 10每个星期几的销售总额，显示英文星期名称，按数字星期几排序。

第二十三章　OLAP函数

完成本章学习后，将能够：

- 使用标准 SQL 进行数据挖掘
- 在标准 SQL 中使用 OLAP 函数
- 执行统计方面的采样(samplings)、排队(rankings)和分位数(quantiles)
- 了解 OLAP 统计函数

23.1 OLAP函数简介

OLAP即联机分析处理(On-Line Analytical Process)。Teradata数据库本身提供了一些OLAP函数，包括：

RANK - 排队(Rankings)

QUANTILE - 分位数(Quantiles)

CSUM - 累计(Cumulation)

MAVG - 移动平均(Moving Averages)

MSUM - 移动合计(Moving Sums)

MDIFF - 移动差分(Moving Differences)

MLINREG - 移动线性回归(Moving Linear Regression)

OLAP函数与聚合函数有类似的地方：

- 对数据进行分组操作 (类似于 GROUP BY 子句)
- 能够使用 QUALIFY 子句过滤组 (类似于 HAVING 子句)

OLAP函数又与 聚合函数不同，因为：

- 返回满足条件的每行的数据值，而不是组的值
- 不能在子查询内使用

OLAP函数可以对下面的数据库对象或动作使用：

- Tables (Perm, Temp, Derived)
- Views
- INSERT/SELECT

23.2 累计函数

累计函数(CSUM) 计算一列的连续的累计的值。语法为：

```
CSUM(colname, sort list)
```

表'daily_sales'在许多查询中都将使用，其定义如下。

```
CREATE SET TABLE daily_sales ,NO FALLBACK
                ,NO BEFORE JOURNAL
                ,NO AFTER JOURNAL
```

```
    (itemid INTEGER
    ,salesdate DATE FORMAT 'YY/MM/DD'
    ,sales DECIMAL(9,2))
PRIMARY INDEX ( itemid );
```

问题

创建item 10从1998年1月和2月的连续的日汇总报表。

解答

```
SELECT salesdate, sales, csum(sales, salesdate)
FROM daily_sales
WHERE salesdate BETWEEN 980101 AND 980301
AND itemid = 10;
```

结果

salesdate	sales	Csum
98/01/01	150.00	150.00
98/01/02	200.00	350.00
98/01/03	250.00	600.00
98/01/05	350.00	950.00
98/01/10	550.00	1500.00
98/01/21	150.00	1650.00
98/01/25	200.00	1850.00
98/01/31	100.00	1950.00
98/02/01	150.00	2100.00
98/02/03	250.00	2350.00

98/02/06 350.00 2700.00
98/02/17 550.00 3250.00
98/02/20 450.00 3700.00
98/02/27 350.00 4050.00

在上面的报表中，每行都代表item 10一天的数据。注意，不是每天都销售了item 10。最右边的列代表其在两个月内的累计销售额。

如果想每月重新累计，该怎么办？

累计汇总可以使用GROUP BY子句在特殊的点复位，即重新开始累计。注意，OLAP函数和标准聚合函数(SUM, COUNT,AVG, MIN, MAX) 是不能在同一查询中兼容的。因此，对这类查询使用GROUP BY，将会起分隔的作用。

问题

创建item 10从1998年1月和2月的连续的日汇总报表，并且每月重新开始累计。

解答

```
SELECT salesdate, sales, csum(sales, salesdate)
FROM daily_sales ds, sys_calendar.calendar sc
WHERE ds.salesdate = sc.calendar_date
AND sc.year_of_calendar = 1998
AND sc.month_of_year in (1,2)
```

```
AND ds.itemid = 10
GROUP BY sc.month_of_year;
```

结果

salesdate	sales	Csum	
98/01/01	150.00	150.00	
98/01/02	200.00	350.00	
98/01/03	250.00	600.00	
98/01/05	350.00	950.00	
98/01/10	550.00	1500.00	
98/01/21	150.00	1650.00	
98/01/25	200.00	1850.00	
98/01/31	100.00	1950.00	
98/02/01	150.00	150.00	重新累计
98/02/03	250.00	400.00	
98/02/06	350.00	750.00	
98/02/17	550.00	1300.00	
98/02/20	450.00	1750.00	
98/02/27	350.00	2100.00	

要回答上面的问题，你需要得到年和月。连接到系统日历，可以获得月。GROUP BY子句累计重新开始，告诉系统"当月改变时，累计值清零"。

23.3 移动平均函数

移动平均函数(MAVG) 基于预定的行数(查询宽度)计算一列的移动平均值。如果行数小于这个宽度，则基于前面已有的行计算平均值。使用移动平均的语法是：

```
MAVG(colname, n, sortlist)
```

colname = 计算移动平均值的列

n = 行数(< 4096)，计算时将使用，包括当前行('n' 也称为平均宽度)

sortlist = 确定行顺序的列

问题

显示item 10基于7天宽度的移动平均值。

解答

```
SELECT salesdate,itemid, sales, MAVG(sales, 7, salesdate)
FROM daily_sales;
```

结果

salesdate	itemid	sales	MAvg
98/01/01	10	150.00	150.00

98/01/02 10 200.00 175.00 前2行的平均值
98/01/03 10 250.00 200.00
98/01/05 10 350.00 237.50
98/01/10 10 550.00 300.00
98/01/21 10 150.00 275.00
98/01/25 10 200.00 264.29 前7行的平均值
98/01/31 10 100.00 257.14
98/02/01 10 150.00 250.00
98/02/03 10 250.00 250.00
98/02/06 10 350.00 250.00
98/02/17 10 550.00 250.00
98/02/20 10 450.00 292.86

计算时，使用当前行和前面n-1行。如果行数小于n-1，则使用前面所有行。缺省，按照sortlist中的列升序排列。

23.4 移动汇总函数

移动汇总函数(MSUM) 基于预定的查询宽度计算一列的移动汇总值。宽度决定有多少行合计到汇总值中。如果前面的行数小于n，则仅使用前面所有行。移动汇总函数的语法是：

```
MSUM(colname, n, sortlist)
```

注意：'n'是一个整数，表示有多少行参加汇总求和。

问题

显示item 10基于3天宽度的移动汇总值。

解答

```
SELECT salesdate, itemid, sales, msum(sales, 3, salesdate)
FROM daily_sales;
```

结果

salesdate	itemid	sales	MSum	
98/01/01	10	150.00	150.00	
98/01/02	10	200.00	350.00	前面2行的总和
98/01/03	10	250.00	600.00	
98/01/05	10	350.00	800.00	
98/01/10	10	550.00	1150.00	
98/01/21	10	150.00	1050.00	
98/01/25	10	200.00	900.00	前面3行的总和
98/01/31	10	100.00	450.00	
98/02/01	10	150.00	450.00	
98/02/03	10	250.00	500.00	
98/02/06	10	350.00	750.00	
98/02/17	10	550.00	1150.00	
98/02/20	10	450.00	1350.00	
98/02/27	10	350.00	1350.00	

移动汇总基于特定的时间段进行聚合汇总。上面的例子表达了这样的含义：

- 销售 3 后，我们共销售了多少？
- 销售 6 后，我们共销售了多少？
- 等等

移动汇总(MSum)与移动平均(MAvg)有下列相同的规则：

- 使用当前行和前 n-1 行
- 如果行数小于 n-1，使用前面所有的行
- 缺省按照 sortlist 中的列升序排列

23.5 移动差分函数

移动差分函数(MDIFF) 基于预定的查询宽度计算一列的移动差分值。宽度决定有多少行参与计算。如果前面的行数小于n，则产生一个空值(null)代表差值。移动差分函数的语法是：

```
MDIFF(colname, n, sortlist)
```

宽度n<=4096

问题

显示item 10基于3天宽度的移动差分值。

解答

```
SELECT salesdate, itemid, sales, mdiff(sales, 3, salesdate)
FROM daily_sales;
```

结果

salesdate	itemid	sales	MDiff	
98/01/01	10	150.00	?	
98/01/02	10	200.00	?	
98/01/03	10	250.00	?	
98/01/05	10	350.00	200.00	2行的差值
98/01/10	10	550.00	350.00	
98/01/21	10	150.00	-100.00	
98/01/25	10	200.00	-150.00	
98/01/31	10	100.00	-450.00	
98/02/01	10	150.00	.00	
98/02/03	10	250.00	50.00	2行的差值
98/02/06	10	350.00	250.00	
98/02/17	10	550.00	400.00	
98/02/20	10	450.00	200.00	
98/02/27	10	350.00	.00	

Mdiff列的值表示某日销售额与三天前日销售额的差。

MDIFF的用法与MAvg 和MSum 有一些不同：

- 使用当前行和前面的第 n 行

- 如果前面没有第 n 行，返回空值(null)
- 缺省按照 sortlist 中的列升序排列

23.6 排队函数

23.6.1 准备数据表

在数据库Customer_Service中增加一个销售表，后面的OLAP查询将使用这个表。

表定义

```
CREATE TABLE salestbl
    (storeid        INTEGER,
     prodid         CHAR(1),
     sales  DECIMAL(9,2));
```

表中数据为：

storeid	prodid	sales
1001	A	100000.00
1001	C	60000.00
1001	D	35000.00
1001	F	150000.00
1002	A	40000.00
1002	C	35000.00
1002	D	25000.00
1003	A	30000.00

1003 B 65000.00
1003 C 20000.00
1003 D 50000.00

23.6.2 简单排队

排队函数对一列进行排队，可以按照升序或者降序排队。缺省，输出结果按照降序排队，对应的排队名次是升序。换句话说，如果一个销售代表在某季度的销售额最高，其排名为1，这是一个最小的值。

排队函数(RANK)的语法是：

```
RANK(colname)
```

这里，colname表示排队的列名，其结果降序排列。

问题

显示商店1001的产品销售额排队。

解答

```
SELECT storeid, prodid, sales, RANK(sales)
FROM salestbl
WHERE storeid = 1001;
```

结果

storeid	prodid	sales	Rank
1001	F	150000.00	1
1001	A	100000.00	2
1001	C	60000.00	3
1001	D	35000.00	4

如上所示，列Rank的最大值代表最低的销售额。

使用排队函数的规则包括：

- WHERE 子句限定参与排队的记录。
- 应用排队函数时，缺省最大的数名次最低。
- 缺省顺序是按排队列的降序。

23.6.3 带限定的排队

QUALIFY子句限制排队输出的最终结果。QUALIFY子句与HAVING子句类似，使输出限制在一定范围内。

问题

按商店得到销售前3名的产品。

解答

```
SELECT storeid, prodid, sales, rank(sales)
FROM salestbl
GROUP BY storeid
QUALIFY rank(sales) <= 3;
```

结果

storeid	prodid	sales	Rank
1001	A	100000.00	1
1001	C	60000.00	2
1001	D	35000.00	3
1002	A	40000.00	1
1002	C	35000.00	2
1002	D	25000.00	3
1003	B	65000.00	1
1003	D	50000.00	2
1003	A	30000.00	3

上面的例子中，GROUP BY子句不是做聚合，它实际上是改变查询的范围，也引起排序。

使用排队函数的规则包括：

- 对某列的每行都应用了排队
- GROUP BY 子句控制范围，如商店内的销售额排队(注意- 查询中并没有聚合)

- QUALIFY 子句限制显示，如仅显示销售额的前 3 名
- 缺省的顺序是按照排队的列的降序
- 因为 GROUP BY 子句，排序将限制在组内，如商店内。

23.6.4 排队中的变化

GROUP BY子句可以和RANK函数一起使用，改变排队的范围。没有GROUP BY子句，缺省的范围是排队的列。前面的例子中，范围是销售额，排队是基于销售额进行的。

排队的列是缺省的排序列，GROUP BY子句增加了一级排序。

缺省排序 - 按排队列降序

次排序 - 按GROUP BY中的列升序

我们看一些使用排队函数的例子。第一个例子，通过使用GROUP BY子句，增加了一级范围，现在的范围是商店内产品的销售情况。因为3个元素构成了排队的组，不同的商店和产品只有一行记录，所以不会产生排队。

问题

得到前3名的销售额 - 任何商店、任何产品。

解答

```
SELECT storeid
    ,prodid
    ,sales
    ,rank(sales)
FROM salestbl
GROUP BY storeid, prodid
QUALIFY RANK(sales) <= 3
;
```

结果

storeid	prodid	sales	Rank
1001	A	100000.00	1
1001	C	60000.00	1
1001	D	35000.00	1
1001	F	150000.00	1
1002	A	40000.00	1
1002	C	35000.00	1
1002	D	25000.00	1
1003	A	30000.00	1
1003	B	65000.00	1
1003	C	20000.00	1
1003	D	50000.00	1

排名都是1的原因是使用了GROUP BY子句，范围变成了商店内的产品的销售额。因为每种产品在一个商店中只有一种，所以排名都是1。

现在去掉GROUP BY子句，范围变成了缺省的销售额。

解答

```
SELECT storeid
   ,prodid
   ,sales
   ,rank(sales)
FROM salestbl
QUALIFY RANK(sales) <= 3
;
```

结果

storeid	prodid	sales	Rank
1001	F	150000.00	1
1001	A	100000.00	2
1003	B	65000.00	3

23.6.5 带聚合的排队

由于上面的原因，OLAP函数与聚合函数在同一查询内不兼容，否则将引起二意性。替代的办法，是使用导出表(derived)或临时表来解决这类问题。临时表包含聚合信息，再对临时表进行OLAP查询。

问题

获得销售额在前3名的产品，跨所有商店。

解答

```
SELECT t.prodid, t.sumsales,  RANK(t.sumsales)
FROM (SELECT a.prodid, sum(a.sales) FROM salestbl  a
   GROUP BY 1) AS t(prodid, sumsales)
QUALIFY RANK(sumsales) <= 3;
```

注意：'t' 是一个从salestbl产生的导出表，包含两列：prodid and sumsales。

结果

prodid	Sumsales	Rank
A	170000.00	1
C	115000.00	2
D	110000.00	3

缺省顺序是按照列sumsales的降序。这里主查询中没有GROUP BY子句，范围是Sumsales。

问题

获得销售额在最后3名的产品，跨所有商店。

解答

```
SELECT t.prodid, t.sumsales,  RANK(t.sumsales)
FROM (SELECT a.prodid, sum(a.sales)
   FROM salestbl  a
   GROUP BY 1) AS t(prodid, sumsales)
QUALIFY RANK(sumsales ASC) <= 3;
```

结果

prodid	sumsales	Rank
B	65000.00	5
D	110000.00	4
C	115000.00	3

注意：名次可能大于3。

这个查询与前面的唯一区别是QUALIFY子句中的sumsales为升序。为什么会得到大于3的结果呢？

答案是按sumsales的升序(sumsales ASC) 排列后，会给我们最底下的3行数据。QUALIFY子句的意思是‘按销售额升序排列，得到前3行数据’。

23.6.6 排队和排序

有时得到低的排名也很有用，例如，显示销售最差的10个产品，薪水最低的5个员工，预算最少的3个部门。要得到低的排名，可以把排队的顺序从降序改为升序。

因此，将RANK(col)改成RANK(col ASC)。然而，RANK(col ASC)将会引起升序排队，最低的值会在最前面。

注意：QUALIFY子句不会查看绝对的数值。QUALIFY子句<= 3的含义是得到列表中的前3行，不管列表的顺序如何。得到的名次可能大于3。

现在看看ORDER BY 子句怎样影响查询结果。

问题

得到销售最差的3种产品，跨所有商店。

解答

```
SELECT  t.prodid, t.sumsales,  rank(t.sumsales)
FROM (SELECT a.prodid, sum(a.sales) FROM salestbl  a
   GROUP BY 1) as t(prodid, sumsales)
QUALIFY RANK(sumsales asc) <= 3;
```

结果

prodid	sumsales	Rank

B	65000.00	5
D	110000.00	4
C	115000.00	3

现在做同样的查询，这次包括ORDER BY子句。

问题

得到销售最差的3种产品，跨所有商店，并按照产品号(prodid)排序。

解答

```
SELECT t.prodid, t.sumsales,  rank(t.sumsales)
FROM (SELECT a.prodid, sum(a.sales) from salestbl  a
 GROUP BY 1) as t(prodid, sumsales)
QUALIFY RANK(sumsales asc) <= 3
ORDER BY 1;
```

结果

Prodid	sumsales	Rank
B	65000.00	5
C	115000.00	3
D	110000.00	4

看上面的结果，同样的3行数据，但顺序不一样。ORDER BY子句中的列成为主排序列。

问题

如果查询变成<=2，结果会怎样?

行B和D将返回，这是销售额最低的2种产品。

排队时，首先查看RANK列的顺序，缺省是降序，产生降序的排队。因此，最大值排名第一，放在列表的顶部。QUALIFY子句决定输出列表顶部多少行记录。

Alternatively, 如果RANK 函数表示按升序排队，最大值仍然排名第一，但是放在列表的最底端，最小值放在列表的顶部(名次的值最大)。当使用QUALIFY子句时，仍然从列表顶部读数据， 最小值会先得到。

23.7 分位数函数

23.7.1 使用分位数

分位数用于将一组记录分成大致相等的部分。最常见的分位数是百分位数(基于100)，也有4分位数 (基于4)、3分位数 (基于3) 和10分位数 (基于10)。注意，缺省地，分位数的列和值都按升序输出。

可以使用ORDER BY子句重新排序。如前所述，聚合函数不能和OLAP函数混合使用，如果要使用聚合函数，可以使用导出表或临时表。分位数函数的语法是：

```
QUANTILE (quantile_constant,sortlist)
```

quantile_constant = 定义分位数大小的常量。

sortlist = 用于分割和排序的列。

问题

显示产品销售额的百分位。

解答

```
SELECT t.prodid, t.sumsales, QUANTILE (100,sumsales)
FROM (SELECT a.prodid, sum(a.sales) from salestbl a
GROUP BY 1) as t(prodid, sumsales);
```

注： 宽度= 100，分位数的范围是0-99 。

结果

prodid	sumsales	Quantile
B	65000.00	0

D	110000.00	20
C	115000.00	40
F	150000.00	60
A	170000.00	80

上述查询使用导出表(derived table)进行聚合，使用缺省的分位数升序排序。可以使用ORDER BY子句重新排序。总共有5个产品，所以等分为20% 。

问题

显示销售额的百分位数为60+的产品。

解答

```
SELECT t.prodid, t.sumsales, QUANTILE(100,sumsales)
FROM (SELECT a.prodid, sum(a.sales) from salestbl a
GROUP BY 1) as t(prodid, sumsales)
QUALIFY QUANTILE(100,sumsales) >= 60;
```

结果

prodid	sumsales	Quantile
F	150000.00	60
A	170000.00	80

23.7.2 分位数的变化

分位数可以看作是排队的相反的函数。我们可以做一些变化。

观察下面的例子：

QUANTILE(3, sumsales DESC)

DESC说明符是QUANTILE 函数缺省的，好象与前面矛盾，实际上并不矛盾。观察QUANTILE (3, sumsales DESC)的执行情况，输出结果仍然按sumsales 升序排序。限定词DESC 指示列‘sumsales’ 按降序分位值，与输出的顺序(如ORDER BY)无关。换句话说，最小的 ‘sumsales’ 值(65,000) 对应于最小的分位值 (0) ，最大的 ‘sumsales’ 值(170,000) 对应于最大的分位值(2)。

增加DESC 是不必要的，数据的输出顺序与上一节的例子是一样的。

解答

```
SELECT t.prodid, t.sumsales, QUANTILE (3,sumsales desc)
FROM (SELECT a.prodid, sum(a.sales) from salestbl a
 GROUP BY 1) as t(prodid, sumsales);
```

结果

prodid	sumsales	Quantile
B	65000.00	0
D	110000.00	0
C	115000.00	1

F 150000.00 1

A 170000.00 2

较低的分位数得到额外的行。

缺省按照降序产生分位数。

与分位数一样，列值也按照降序排列。

注意下面的例子：

QUANTILE(3, sumsales ASC)

使用非缺省的选项ASC，列 ‘sumsales’ 将是升序，分位数值是降序。换句话说，最大的 ‘sumsales’ 值(170,000) 对应最小的分位数值 (0)，反之则相反。

可以使用ORDER BY重新排序，但是对两列之间的相对顺序没有影响。

解答

```
SELECT t.prodid, t.sumsales, QUANTILE (3,sumsales asc)
FROM (select a.prodid, sum(a.sales) from salestbl a
 GROUP BY 1) as t(prodid, sumsales);
```

结果

prodid	sumsales	Quantile
A	170000.00	0
F	150000.00	0
C	115000.00	1
D	110000.00	1
B	65000.00	2

在分位函数中，升序的分位意味着最小的值得到最大的分位值。

23.7.3 分位与聚合

下面第一个查询返回公司内薪水在前25%的雇员。第二个查询得到薪水前25%的雇员的薪水的总和，解决办法是通过使用导出表，混合聚合函数和OLAP函数。

问题

显示公司前25%的所有员工的薪水。

解答

```
SELECT salary_amount, QUANTILE (100, salary_amount)
FROM employee
QUALIFY QUANTILE (100, salary_amount) >=75;
```

结果

salary_amount	Quantile
53625.00	76
54000.00	80
56500.00	84
57700.00	88
66000.00	92
100000.00	96

问题

计算公司前25%的薪水的总和。

解答1

```
SELECT sum(salary_amount)
FROM employee QUALIFY QUANTILE (100, salary_amount) >= 75;
```

结果

```
****Error - Can't mix stat functions with aggregates***
```

解答2

```
SELECT sum(sals)
FROM (SELECT salary_amount from employee
QUALIFY QUANTILE(100, salary_amount) >= 75)
```

temp(sals);

结果

Sum(sals)
387825.00

23.7.4 分位与排序

下面的例子中，分位数函数表示如下：

QUANTILE (100, salary_amount, employee_number)

这里 ‘employee_number’ 是用于处理有相同薪水员工的情形。记住，‘salary_amount’ (缺省是DESC) 告诉我们salary_amount将与分位数值一样降序排序。使用employee_number，进一步保证在相同薪水的情况下，employee_number将与分位数值一样降序排序。

问题

显示薪水最低的25%的所有员工。

解答1

SELECT employee_number, salary_amount,

```
QUANTILE (100, salary_amount , employee_number)
FROM employee
QUALIFY QUANTILE (100, salary_amount) < 25;
```

结果

employee_number	salary_amount	Quantile
1014	24500.00	03
1013	24500.00	00
1001	25525.00	07
1023	26500.00	11
1008	29250.00	15
1006	29450.00	19
1009	31000.00	23

在第二个例子中，分位数函数表示如下：

```
QUANTILE (100, salary_amount, employee_number ASC)
```

在这个例子中，employee_number (在相同薪水的情况下)是升序，分位数值是降序。与前面的例子相比，两个雇员号颠倒了。记住，你可以使用ORDER BY 子句重新排序。

解答2

```
SELECT employee_number, salary_amount,
QUANTILE (100, salary_amount , employee_number ASC)
```

```
FROM employee
QUALIFY QUANTILE (100, salary_amount) < 25;
```

结果

employee_number	salary_amount	Quantile
1013	24500.00	03
1014	24500.00	00
1001	25525.00	07
1023	26500.00	11
1008	29250.00	15
1006	29450.00	19
1009	31000.00	23

23.8 移动线性回归函数

23.8.1 使用线性回归预测

移动线性回归函数MLINREG基于一个序列数据对得到一个预测值。序列对包含一个独立的变量和一个依赖的变量。MLINREG函数基于前面的n对数预测依赖变量的值。n叫做宽度。

MLINREG 的语法是：

```
MLINREG (y, n, x)
```

函数基于前面n-1行计算y值。y是依赖变量。x是独立变量，也是排序值。x和y都必须是数字列，不能是日期。宽度n必须>=3 且<=4096。

例1

给出序列对x 和y的集合，使用线性回归预测y值，宽度是3。

解答

```
SELECT x, y, MLINREG(y, 3, x)
FROM linreg;
```

结果

x	y	MLinReg
1	2	?
2	4	?
3	6	6
4	7	8
5	8	8
6	4	9
7	6	0
8	5	8
9	10	4
10	15	15

例2

给出序列对x 和y的集合，使用线性回归预测y值，宽度是6。

解答

```
SELECT x, y, MLINREG(y, 6, x)
FROM linreg;
```

结果

x	y	MLinReg
1	2	?
2	4	?
3	6	6
4	7	8
5	8	9
6	4	10
7	6	6
8	5	5
9	10	4
10	15	8

头两行总是空值(null)，结果的线性依赖于两个变量的线性。缺省的顺序是按列 (x)的升序。可以使用ORDER BY重新排序。

线性回归是一个数学算法，其公式为：

$$B = \frac{sum(x*y) - sum(x)*sum(y)/n}{sum(x*x) - sum(x)*sum(x)/n}$$

$$A = sum(y)/n - B*sum(x)/n$$

MLINREG 的结果 = A + B*x

23.8.2 按日期预测

移动线性回归函数的一个限制是不能使用日期作为独立变量。因为计算回归需要使用加法、减法和乘法，对DATE 数据类型运算有问题。

一种解决办法是与系统日历做连接，用day_of_year 列代替日期，如果产生基于月的结果，也可以使用month_of_year列。

注意GROUP BY子句会引起函数复位，GROUP BY子句也是主要的排序键，独立变量是次要的排序键。

在移动线性回归函数中使用日期，可以使用CAST把他们转换成CHAR类型，从而作为非日期类型使用。

问题

使用线性回归算法预测items 10 和11在1998年头两周的日销售额，宽度为5。

解答

```
SELECT  itemid, CAST(salesdate as char(10)) as chardate, sales
,mlinreg(sales, 5, chardate)
FROM jan_sales
WHERE salesdate between 980101 and 980114
AND itemid in (10,11)
GROUP BY 1
;
```

结果

itemid	chardate	sales	MLinReg	
10	98/01/01	150.00	?	
10	98/01/02	200.00	?	
10	98/01/03	250.00	250.00	
10	98/01/04	350.00	300.00	
10	98/01/05	550.00	400.00	
10	98/01/06	150.00	625.00	
10	98/01/07	200.00	300.00	
10	98/01/08	100.00	100.00	
10	98/01/09	150.00	-75.00	
10	98/01/10	250.00	125.00	
10	98/01/11	350.00	225.00	
10	98/01/12	550.00	425.00	
10	98/01/13	450.00	650.00	
10	98/01/14	350.00	600.00	
11	98/01/01	350.00	?	复位
11	98/01/02	100.00	?	
11	98/01/03	450.00	-150.00	
11	98/01/04	550.00	400.00	

11	98/01/05	250.00 600.00
11	98/01/06	350.00 475.00
11	98/01/07	200.00 250.00
11	98/01/08	150.00 100.00
11	98/01/09	250.00 125.00
11	98/01/10	450.00 150.00
11	98/01/11	550.00 475.00
11	98/01/12	250.00 700.00
11	98/01/13	350.00 400.00
11	98/01/14	350.00 250.00

上面的内容有些需要注意：

- GROUP BY 子句引起线性回归算法复位
- GROUP BY 子句优先排序
- DATE 数据类型需要转换成 CHAR 类型

23.9 采样函数

23.9.1 简单采样

采样函数SAMPLE用于从表或视图中产生一些样本数据。有两种形式：

- 基于实际的行数
- 基于表的百分比

SAMPLE n - 这里n是一个整数。如果表中记录数>=n，将产生n行记录；如果表中记录数<n，样本是表中的所有记录。

SAMPLE n - 这里n 是一个小数，并且0.00<n<1.00。将作为百分比抽取表中数据，并且采用四舍五入。如对雇员表采样25%，雇员表有26行记录，则采样26 * .25 = 6.50 = 6 行记录。

例1

```
SELECT department_number
FROM employee
SAMPLE 10;
```

结果

****Query completed. 10 rows found.

department_number
401
401
403
401
301
401
403
402
401
401

注意：这里没有使用DISTINCT子句，仅仅任意抽取10行记录。

例2

```
SELECT department_number
FROM employee
SAMPLE  .25;
```

结果

****Query completed. 6 rows found.

department_number
403
401
402
301
501
403

23.9.2 采样不同的值

第一个例子中，采样13个雇员记录，并计数他们代表多少不同的部门，但是查询没有得到这个结果。所有雇员都被考虑进来，部门数总共为8。SAMPLE 13指示输出13行结果，但COUNT函数只返回1行结果，出现警告信息。

要得到上面要求的结果，必须使用临时表或导出表，先采样的13行数据放入表中，再对样本数据进行计数。第二个例子就是采用这种方法。重复执行查询，结果可能不一样，因为采样出的数据可能不同。

问题

采样13 个不同的部门号。

解答1

```
SELECT COUNT(DISTINCT department_number)
FROM employee sample 13;
```

结果

```
*** Warning: 7473 Requested sample is larger than table rows. All rows returned

Count(Distinct(department_number))
8
```

得到了一个警告。希望返回13行记录，但是使用了DISTINCT操作符，同样的行只返回一次，没有13行。

解答2

使用导出表(derived table)进行采样。

```
SELECT COUNT(DISTINCT dept)
FROM
(SELECT department_number from employee sample 13)
temp(dept);
```

结果 - 第一次执行

```
Count(Distinct(dept))
6
```

结果 - 第二次执行

```
Count(Distinct(dept))
5
```

两次运行结果不一样，原因是导出表是采样产生的13行记录，DISTINCT是基于导出表而不是整个雇员表。

23.9.3 使用SAMPLEID

在一个查询中可以产生多个采样集合。为了标识特别的集合，使用SAMPLEID 来与每个集合联系。SAMPLEID可以用于SELECT子句和ORDER BY子句， 也可以作为新表中的一列。SAMPLEID是一个关键字，是采样集合的标识。

在第一个例子中，即使采样百分比加起来是1.00，但也只返回部门表9行记录中的8行。计算如下：

9 *.25 = 2.25 = 2
9 *.25 = 2.25 = 2
9 *.50 = 4.50 = 4

8

在第二个例子中，前两个采样值足够大，可以返回记录；但第三个值(.02) 不能。

在第三个例子中，因为记录不会重复采样，在第三个采样完成之前已经没有记录了。因此，结果会产生警告。

例1

```
SELECT department_number
   ,sampleid
FROM department
SAMPLE .25, .25, .50
ORDER BY sampleid ;
```

结果

****Query completed. 9 rows found.***

department_number	SampleId
301	1
403	1
302	2
401	2
100	3
402	3
201	3
600	3

例2

```
SELECT department_number
   ,sampleid
FROM department
SAMPLE .27, .25, .02
ORDER BY sampleid;
```

结果

department_number	SampleId
302	1
401	1

302	1
401	1
301	2
403	2
201	2

例3

```
SELECT department_number
   ,sampleid
FROM sample 3, 5, 8
ORDER BY sampleid;
```

结果

*** Warning: 7473 Requested sample is larger than table rows. All rows returned.***

department_number	SampleId
302	1
301	1
403	1
100	2
402	2
401	2
201	2
600	2
501	3

23.10 OLAP统计函数

Teradata V2R4中增加了一些OLAP统计函数。

一元统计函数包括：

STDDEV_SAMP, STTDEV_POP, VAR_SAMP, VAR_POP, SKEW, KURTOSIS

二元统计函数包括：

CORR, COVAR_POP, REGR_SLOPE, REGR_INTERCEPT

OLAP统计函数的特点包括：

- 能够使用 GROUP BY 产生分组。
- 可以与聚合函数混合使用。
- 不能与其他 OLAP 函数混合使用。

23.10.1 标准偏差函数

样本标准偏差

```
STDDEV_SAMP ({DISTINCT} value_expression)
```

- Value_expression 是计算样本标准偏差的列表达式。
- 返回 value_expression 的样本标准偏差。
- 从给的样本中计算差量。

STDDEV_SAMP(x)的计算公式如下：

STDDEV_SAMP = SQRT((COUNT(x) * SUM(x**2) - (SUM(x))**2/(COUNT(x) * (COUNT(x) - 1)))

全体标准偏差

STDDEV_POP ({DISTINCT} value_expression)

- Value_expression 是计算全体标准偏差的列表达式。
- 返回 value_expression 的全体标准偏差。
- 从全体数据中计算差量。

STDDEV_POP(x)的计算公式如下：

STDDEV_POP = SQRT((COUNT(x) * SUM(x**2) - (SUM(x))**2/(COUNT(x) * (COUNT(x)**2 - 1)))

23.10.2 变异函数

样本变异函数

VARSAMP ({DISTINCT} value_expression)

- Value_expression 是计算样本变异的列表达式。
- 返回 value_expression 的样本变异。
- 从给的样本或样本标准偏差的平方中计算差量。

VAR_SAMP(x)的计算公式如下：

VAR_SAMP = (COUNT(x) * SUM(x**2) - (SUM(x))**2/(COUNT(x) * (COUNT(x) - 1))

全体变异函数

```
VAR_POP ({DISTINCT} value_expression)
```

- Value_expression 是计算全体变异的列表达式。
- 返回 value_expression 的全体变异。
- 从全体数据中计算差量。

VAR_POP(x)的计算公式如下：

VAR_POP = (COUNT(x) * SUM(x**2) - (SUM(x))**2/(COUNT(x) * (COUNT(x)**2))

23.10.3 分布函数

分布歪斜函数

```
SKEW ({DISTINCT} value_expression)
```

- Value_expression 是计算分布歪斜的列表达式。
- 返回 value_expression 的分布歪斜。
- 测量数据分布的不对称，与正常分布(歪斜值为 0)进行比较。
- 正歪斜值指示朝正方向的不对称。
- 负歪斜值指示朝负方向的不对称。

SKEW(x)的计算公式如下：

SKEW = (COUNT(x) /((COUNT(x) - 1) * (COUNT(x) 2)) * SUM(((x - AVG(x))/ STDDEV_SAMP(x))**3

分布峰态函数

KURTOSIS ({DISTINCT} value_expression)

- Value_expression 是计算分布峰态的列表达式。
- 返回 value_expression 的分布峰态。
- 测量数据分布的高峰或平缓，与正常分布(峰态值为 0)进行比较。
- 正峰态值指示朝正方向的峰值。
- 负峰态值指示朝负方向的峰值。

KURTOSIS(x)的计算公式如下：

KURTOSIS = (COUNT(x) * (COUNT(x) + 1) * (COUNT(x) - 2) * (COUNT(x) - 3))) SUM((x - AVG(x)) / STDDEV_SAMP(x))*4 - (3* (COUNT(x) - 1)**2 / ((COUNT(x) - 2) * (COUNT(x) - 3))

23.10.4 相关性和协方差函数

协方差函数

COVAR_POP (value_expression, value_expression)

- Value_expression 是计算协方差的一对列表达式。
- 返回数据对的协方差。
- 是数据对的平均背离乘积。

COVAR_POP(x,y)的计算公式如下：

COVAR_POP = SUM((x - AVG(x)) * (y - AVG(y)))/COUNT(x)

相关性函数

CORR (value_expression, value_expression)

- Value_expression 是相关的一对列表达式。
- 返回数据对的皮尔森积(Pearson product)。
- 是测量变量之间的非因果的线性联系(non-causal linear association)。

CORR(x,y)的计算公式如下：

CORR = COVAR_POP(x,y) / (STDDEV_SAMP (x) * STDDEV_SAMP(y))

23.10.5 线性回归函数

回归倾斜函数

REG_SLOPE (value_expression1, value_expression2)

- Value_expression1 是独立变量。
- Value_expression2 是依赖变量。
- 基于独立变量和依赖变量返回线性回归的倾斜(slope)值。
- 测试独立变量在依赖变量上的变化率。

REG_SLOPE(x,y)的计算公式如下：

REG_SLOPE = (COUNT(x) * SUM(x) * SUM(Y)) / (COUNT(x) * (SUM(x**2)) - SUM(x)**2)

回归截取函数

```
REG_INTERCEPT (value_expression1, value_expression2)
```

- Value_expression1 是独立变量。
- Value_expression2 是依赖变量。
- 基于独立变量和依赖变量返回线性回归的截取(intercept)值。
- 是回归线与纵坐标的截取点。

REG_INTERCEPT(x,y)计算公式如下：

REG_INTERCEPT= AVG(y) - (REGR_SLOPE(x,y) * AVG(x))

练习

Lab 23_1

显示item 10在1998年头两月的销售报表，包括销售日期、销售量、累计、3天的移动汇总、3天的移动差分和3天的移动平均值。

Lab 23_2

基于budget_amounts排队得到前3个部门。显示部门号和预算。

Lab 23_3

基于总salary_amounts排队得到前3个部门。显示部门号和总销售额。

Lab 23_4

产生一个薪水在前20%的雇员列表。显示雇员号、薪水和分位数。

Lab 23_5

修改查询Lab 23_5，显示最后的20%，按分位数降序排序。

Lab 23_6

基于1998年1月头13天的销售情况，预测item 10在1月14日的销售额。显示前13天的日期和销售额及14号的预测值。使用另外一个查询，计算item 10在这13天的平均销售额。比较两个结果。

Lab 23_7

创建一个表：

```
CREATE TABLE sqlxxxx.empsamp
    ( empno INTEGER,
    deptno INTEGER,
    job INTEGER,
    sampid BYTEINT)
```

UNIQUE PRIMARY INDEX (empno);

往表中装入从雇员表采样的数据。有3个采样集合，sampleid表示为1、2和3，每个采样都包含雇员表的33% 的记录。选择新表中的所有列，按sampleid排序。

Lab 23_8

删除表empsamp中的数据，重新装入采样数据。仍然是3个采样，每个从雇员表中采样15行记录。选择新表中的所有列，按sampleid排序。第三个采样会发生什么情况？

Lab 23_9

产生一个唯一的(distinct)部门列表，部门包含薪水占公司前10%的雇员。使用一个查询完成。

第二十四章　触发器

完成本章学习后，将能够：

- CREATE、 ALTER 和 DROP 触发器
- 实现嵌套的触发器和审核的触发器
- 区别触发语句和被触发语句

24.1 触发器基础

24.1.1 什么是触发器

触发器(trigger)是数据库中的一个对象，就象宏和视图是数据库对象一样。触发器使用CREATE TRIGGER 语句创建，定义一些事件，这些事件由其他事件所引起。

一个触发器包含一条或多条与一个表相关联的SQL语句，当触发器被激活时，执行这些SQL语句。

触发器有一些限制，定义了触发器的表不能使用Fastload、Multiload 和可更新的游标(Cursor)。适合于其他数据库对象的数据定义语言，也适合于触发器。

总之，触发器是：

- 与一个表相关联的一条或多条 SQL 语句。
- 是事件驱动的过程。
- 与表、视图、宏一样，也是数据库的一个对象。

下列语句是与触发器相关的：

- CREATE TRIGGER
- DROP TRIGGER
- SHOW TRIGGER
- ALTER TRIGGER
- RENAME TRIGGER
- REPLACE TRIGGER
- HELP TRIGGER

触发器不能与下列工具混合使用：

- FastLoad 实用程序
- MultiLoad 实用程序
- 可更新的游标(存储过程或预处理程序)
- 连接索引

如果要使用FastLoad、MultiLoad实用程序，或创建含可更新的游标的存储过程，必须先使用ALTER TRIGGER命令，禁止(disable)定义在此表上的触发器。

定义了触发器的 表不允许使用连接索引(Join index)。

可以使用下列命令删除所有的触发器：

- DELETE DATABASE
- DELETE USER

创建或删除触发器需要有相应的权限：

- GRANT CREATE 或 DROP Trigger
- REVOKE CREATE 或 DROP Trigger

权限存储在数据字典中。

24.1.2 触发与被触发语句

当条件满足、触发事件发生时，我们说触发器被激活。一个触发器激活时，会引起其他事件发生，即被触发的事件。一个被触发的事件包含一个或多个被触发的语句。

触发语句是引起触发器活动的一条SQL语句，是发动语句。

触发语句可能是：

- INSERT
- UPDATE
- DELETE
- INSERT SELECT

被触发语句是激活的触发器执行的结果语句。

被触发语句可能是：

- INSERT
- UPDATE
- DELETE
- INSERT SELECT
- ABORT/ROLLBACK
- EXEC (macro)

宏只可以包含上面提到的数据操作语句。

被触发语句不能是：

- BEGIN TRANSACTION
- CHECKPOINT
- COMMIT
- END TRANSACTION

- SELECT

可以不使用Begin Transaction/End Transaction (BTET)，在被触发语句中做事务处理。后面我们将告诉你怎样做。

24.1.3 定义触发器

触发器包含许多部件，要创建触发器，他们的语法都要正确。下面是定义触发器的部件。

触发器名称 - 触发器对象的名字，在一个数据库内必须唯一。

触发器动作时间 - 触发器激活的时间，在触发事件之前(BEFORE)、之后(AFTER)，或代替(INSTEAD OF)触发事件。

触发器事件 - 引起触发器激活的数据操作语句。

触发动作 - 同触发器事件。

主题表 - 发生触发器事件的表。

参照子句 - 定义表中旧值和新值怎样命名。

触发器类型 - 触发器分行(Row)类型或语句(Statement)类型。

被触发的语句 - 触发器激活后，执行的语句。

被触发的动作 - 由被触发语句组成，可以包含WHEN子句。

外部参照 - 从被触发动作中参照主题表。

24.1.4 触发器选项

触发器类型

有两类触发器类型：

- 行(ROW)
- 语句(STATEMENT)

行(ROW)触发器

- 触发语句影响的每行都激活一次。
- 参照主题表的 OLD 和 NEW 行。
- 允许在被触发语句中包含简单的 insert、rollback、或宏。

语句(STATEMENT)触发器

- 每个语句激活一次。
- 参照主题表 OLD_TABLE 和 NEW_TABLE 。

触发器动作时间

- BEFORE - 被触发语句在触发语句之前执行。
- AFTER - 被触发语句在触发语句之后执行。
- INSTEAD OF - 被触发语句代替触发语句执行。

触发器可以定义为，当表中的一列或几列更新时被激活。

多列更新

看一些在多列更新时使用触发器的例子。

例1

...BEFORE UPDATE OF (empno, deptno) ON employee...

在上面的例子中，每列都必须更新，才能激活触发器。

例2

...AFTER UPDATE ON employee...

employee表的任意一列被更新，都激活触发器。

语法

我们看一个CREATE TRIGGER语法的例子。

```
CREATE TRIGGER MgrUpdate
  BEFORE UPDATE OF (manager_employee_number) ON employee
  REFERENCING OLD AS oldtable
            NEW AS newtable
FOR EACH STATEMENT
(UPDATE employee SET manager_employee_number
             =newtable.manager_employee_number
WHERE manager_employee_number
```

=oldtable.manager_employee_number;);

24.2 有条件的行触发器

触发器可以有条件地激活，在触发器定义中使用WHEN子句可以实现。WHEN子句指定激活触发器必须满足的条件。WHEN子句有一个限制，就是只能参照主题表的行。

如果一个更新影响1000行的表中的5行记录，那么只有这5行能够使用WHEN子句中的条件。WHEN子句返回的结果可能是True、False或Null ，Null值当作False。False值阻止触发器被激活，也就是执行触发语句，但不执行被触发语句。

如果同一个表上定义了多个触发器，并且有同样的下列属性，可以使用ORDER子句定义触发器的顺序。

- 同样的触发器动作之间 - (如，BEFORE, AFTER, INSTEAD OF)
- 同样的触发器事件 - (如，UPDATE, DELETE, INSERT)
- 同样的触发器类型 - (如，ROW 或 STATEMENT)

如果触发器满足上面的条件，并且有同样的ORDER BY值，将按照随机的顺序激活。ORDER BY的取值范围是1 到2,147,483,647，缺省是32,767。

现在看一个使用WHEN子句的例子。

问题

创建一个行触发器，当任何雇员的薪水增幅超过10%时，触发器向薪水日志表中插入一条记录。(Update操作时，每行记录改变都激活触发器)

解答

```
CREATE TRIGGER raiseTrig
AFTER UPDATE OF (salary_amount) ON employee ORDER 1
    REFERENCING OLD AS oldrow
    NEW AS newrow
    FOR EACH ROW
WHEN((newrow.salary_amount-oldrow.salary_amount)/
                oldrow.salary_amount>.10)
(INSERT INTO salary_log values
(newrow.last_name, oldrow.salary_amount
,newrow.salary_amount,date););
```

WHEN 子句

WHEN子句测试激活触发器的条件。只有更新语句影响的记录能够被WHEN子句参照。WHEN子句：

- 对触发器而言，就象 WHERE 子句。
- 返回 True 或 False - Null 作为 False。
- 只能参照触发语句影响的主题表的记录行。

ORDER 子句

- 决定被触发事件的顺序。
- 取值范围是 1 到 2,147,483,647，缺省值是 32,767。
- 相同值时，触发器随机激活。
- 修改 ORDER 子句，必须使用 REPLACE TRIGGER。

ORDER子句是相对于下列条件：

- 同一表上定义了多个触发器；
- 触发器有相同的动作时间(Before/After)、触发器事件(Upd/Ins/Del)、触发器类型(Row/Stmt)。

24.3 有条件的语句触发器

除了WHEN子句外，在触发的语句中可以使用WHERE子句。

问题

创建一个语句触发器，当任何雇员的薪水增幅超过10%时，触发器向薪水日志表中插入一条记录。(每次Update操作时，只激活一次触发器)

解答

```
CREATE  TRIGGER raiseTrig
AFTER UPDATE OF (salary_amount) ON employee ORDER 1
   REFERENCING OLD_TABLE AS oldtable
   NEW_TABLE AS newtable
   FOR EACH STATEMENT
(INSERT INTO salary_log
 SELECT n.last_name
   , o.salary_amount
   , n.salary_amount
FROM newtable n
  ,oldtable o
```

```
WHERE n.last_name=o.last_name
AND (n.salary_amount - o.salary_amount) /
    o.salary_amount > .10;);

UPDATE employee SET salary_amount = salary_amount * 1.15
WHERE employee_number = 1008;

SELECT * FROM salary_log;

 employee_name oldsal          newsal
 ----------------- -----------  -----------
 Kanieski          32175.00     37001.25
```

WHERE子句

- 能够用于限定被触发动作操作的记录。
- 只参照触发语句涉及的主题表的记录行。

24.4 层叠的触发器

层叠的触发器是一组触发器，被触发的语句引起其他触发器被激活。

下面的例子中，触发器Trig1触发一个Insert操作，激活了触发器Trig2，Trig2向表Tab3中插入一行记录。

触发器不能修改主题表(如果触发的语句又激活触发器，这可能引起无限的循环)。触发器层叠的级别没有限制。

例1

```
CREATE TABLE tab1 (a INT, b INT, c INT);
CREATE TABLE tab2 (d INT, e INT, f INT);
CREATE TABLE tab3 (g INT, h INT, i INT);
```

例2

```
CREATE TRIGGER trig1 AFTER INSERT ON tab1
REFERENCING NEW_TABLE AS newtable
FOR EACH STATEMENT
(INSERT INTO tab2 SELECT a + 10, b + 10, c FROM newtable;);
```

例3

```
CREATE TRIGGER trig2 AFTER INSERT ON tab2
REFERENCING NEW_TABLE AS newtable
FOR EACH STATEMENT
(INSERT INTO tab3 SELECT  d + 100, e + 100, f FROM newtable;);
```

例4

```
INSERT INTO tab1 VALUES (1,2,3);
SELECT * FROM tab1;
```

a	b	c
1	2	3

```
SELECT * FROM tab2;
```

d	e	f

11	12	3

SELECT * FROM tab3;

g	h	i
111	112	3

层叠的触发器语句：

- 不能修改主题表。
- 层叠的层次不限。

24.5 语句触发器和When子句

语句触发器在每个语句执行时仅激活一次。语句触发器使用别名定义旧表和新表，而不是表中的个别行记录。语句触发器象正常的SQL语句一样，基于行的集合操作。

下面的例子中，如果满足下列条件，将激活触发器。

- 更新 employee 表中的 salary_amount 列。
- 每天时间在 8:00 a.m. 和 5:00 p.m.之间。
- 被更新的记录数大于 6。

注意，此处使用聚合函数(COUNT)是正确的，因为它是在子查询中使用。并且，COUNT函数只记数触发器可见的旧表的记录行。这时，只有被更新的雇员记录是可见的，记数反映的是被更新的记录数，而不是所有雇员数。

因为触发器动作时间是INSTEAD OF，实际的更新操作根本不会发生(即使当WHEN子句为true时)，更新时，会将一行记录插入到表StageTbl中。

要求的更新操作影响表中的7行记录，引起触发器被激活，因此插入7行记录到表StageTbl中。

最后，在触发语句中可以使用WHERE子句加入条件，能够限制触发语句改变的行。

这是StageTbl表的定义：

```
CREATE TABLE StageTbl
(sdate  date
,stime  int
,dmltype char(10)
,sempno int
,ssalamt dec(10,2));
```

问题

创建一个语句触发器，如果发生任何更新超过6行记录的操作，则记录日志。

解答

```
CREATE TRIGGER StageUpd
    INSTEAD OF UPDATE OF (salary_amount) ON employee
REFERENCING OLD_TABLE AS oldtable
        NEW_TABLE AS newtable
    FOR EACH STATEMENT
WHEN (TIME BETWEEN 80000 AND 170000
```

```
AND 6<(SELECT COUNT(employee_number) FROM oldtable)
    )
(INSERT INTO StageTbl
SELECT DATE, TIME, 'Update',
employee_number, salary_amount FROM newtable;);
```

问题

对薪水最低的25%的雇员，增加10%的薪水。

解答

```
UPDATE employee SET salary_amount=salary_amount*1.1
WHERE salary_amount<31100;
```

结果

```
 ***0 rows updated, 7 rows inserted.
```

没有行被修改，但是记录了日志。

查看表StageTbl中的内容，可以看见更新操作怎样进行的。

```
SELECT * from StageTbl;
```

结果

sdate	stime	dmltype	sempno	ssalamt
98/10/20	111338	Update	1009	34100.00
98/10/20	111338	Update	1013	26950.00
98/10/20	111338	Update	1014	26950.00
98/10/20	111338	Update	1006	32395.00
98/10/20	111338	Update	1001	28077.50
98/10/20	111338	Update	1023	29150.00
98/10/20	111338	Update	1008	32175.00

24.6 参照规则

对于行触发器和语句触发器，REFERENCING的规则和语法有细微的差别。

行(Row)触发器和语句(Statement)触发器都有必须遵守的参照规则。行触发器使用别名参照触发动作影响的记录行，包括旧行和新行。语句触发器使用别名参照触发语句影响的表，包括旧表和新表。

大多数情况，旧参照和新参照都会在触发器中使用。不允许混合行参照和表参照。INSERTS不需要旧参照，DELETES不需要新参照。

WHEN子句控制触发器的激活条件。WHEN子句和被触发语句可以通过referencing子句参照主题表。这相当于一个‘外部参照(outer reference)’。不论什么情况，主题表不能被触发器修改。

行触发器和语句触发器可以参照主题表以外的其他表。

对两类触发器而言，都只有触发动作影响的主题表中的记录行是可参照的。

行触发器	语句触发器
REFERENCING	REFERENCING
OLD AS oldrow	OLD_TABLE AS oldtable
NEW AS newrow	NEW_TABLE AS newtable
FOR EACH ROW	FOR EACH STATEMENT

规则

- OLD 和 NEW 总是参照行，称为“相关(correlations)”。
- 大多数情况，每个触发器只指定一个。
- OLD_TABLE 和 NEW_TABLE 参照表，称为“别名(aliases)”。
- 每个触发器只允许使用 OLD 或 OLD_TABLE 中的一个。
- 每个触发器只允许使用 NEW 或 NEW_TABLE 中的一个。
- INSERTS 不能使用 OLD_TABLE 或 OLD。
- DELETES 不能使用 NEW_TABLE 或 NEW。

例

```
CREATE TRIGGER abc AFTER UPDATE ON Tbla
REFERENCING    OLD_TABLE AS oldtable
          NEW_TABLE as newtable
   FOR EACH STATEMENT
(UPDATE Tblb SET Tblb.c2 = newtable.c2
WHERE Tblb.c1 = oldtable.c1;);
```

附加规则

- 被触发语句不能修改主题表。

- 只有触发语句影响的行是可以访问的(不是整个表)。

24.7 启用触发器

触发器可以在不需要时被禁止，在需要时重新启用。使用ALTER TRIGGER语句完成这项任务。当需要对定义了触发器的表执行Fastload 或Multiload脚本时，这非常有用。在执行这类操作之前，必须禁止触发器。HELP TRIGGER命令显示触发器是启用状态还是禁止状态，也显示触发器的一些定义信息。

禁止/启用触发器的语法是：

```
ALTER TRIGGER [triggername] [ENABLED or DISABLED];
```

例

ALTER TRIGGER StageUpd DISABLED;

可以执行HELP TRIGGER StageUpd; 得到触发器StageUpd的信息，包括其状态。命令执行如下：

```
.sidetitles on
.foldline on
HELP TRIGGER StageUpd;
```

结果

Name	StageUpd
ActionTime	I
Event	U
Kind	S
Decimal Order Value	32,767
Enabled	N
Comment	?

可以给触发器增加注释。

例

COMMENT ON TRIGGER StageUpd 'After Hours Updates';

启用触发器

ALTER TRIGGER StageUpd ENABLED;
HELP TRIGGER StageUpd;

结果

Name	StageUpd
ActionTime	I
Event	U
Kind	S
Decimal Order Value	32,767
Enabled	Y
Comment	After Hours Updates

24.8 触发器与交易

24.8.1 触发器和交易

触发器关联的所有语句都作为单一的交易，包括触发语句和被触发语句。在内部，Teradata把所有语句作为一个多语句的宏。不论是ANSI模式或Teradata模式，任何个别的语句失败，都会引起整个交易回滚。

可以使用BEGIN ATOMIC和END语句将被触发语句括起来，但这不是必要的。

注意，下面被触发语句执行参照完整性检查。

两个被触发语句 - 一个交易

```
CREATE TRIGGER trig1 AFTER INSERT ON tab1
REFERENCING NEW_TABLE AS newtable
FOR EACH STATEMENT
(INSERT INTO Tab2 SELECT a+10, b+10, c FROM newtable;
 UPDATE Tab3 SET h = newtable.b
   WHERE g = newtable.a;);
```

修改代码 -- 具有同样的效果：

```
CREATE TRIGGER trig1 AFTER INSERT ON tab1
REFERENCING NEW_TABLE AS newtable
FOR EACH STATEMENT
BEGIN ATOMIC
(INSERT INTO Tab2 SELECT a+10, b+10, c FROM newtable;
 UPDATE Tab3 SET h = newtable.b
```

```
    WHERE g = newtable.a;)
END;
```

当触发器被激活时，宏变成：

```
INSERT INTO Tab1
INSERT INTO Tab2
UPDATE OF Tab 3
```

任何语句失败都会引起整个交易回滚。

24.8.2 多个触发器与交易

当一个触发器被激活时，可能引起一条或多条被触发的语句执行，这又可能引起其他触发器被激活。

在下面的例子中，两个触发器有同样的触发动作，一个是在之前动作，一个是在之后动作。当形成交易/宏时，语句的顺序如下：

1. 之前动作的被触发语句
2. 触发语句
3. 之后动作的被触发语句

如果两个触发器有同样的触发动作时间(BEFORE, AFTER, INSTEAD OF)，使用ORDER子句控制执行顺序。没有ORDER子句，则使用随机顺序。

注意：下列被触发语句进行参照完整性检查。

如果触发语句激活了多个触发器，所有触发器中的所有语句都作为一个交易。

```
CREATE TRIGGER Trigger_2
BEFORE UPDATE OF (col_x) ON Tbl_x
... FOR EACH STATEMENT
(DELETE  FROM Tbl_y WHERE .....;);

CREATE TRIGGER Trigger_3
AFTER UPDATE OF (col_x) ON Tbl_x
... FOR EACH STATEMENT
(INSERT  FROM Tbl_y SELECT ... WHERE .....;);
```

当触发语句被激活时，宏变成：

- DELETE FROM Tbl_y
- UPDATE Tbl_x SET col_x = ...
- INSERT INTO Tbl_y

任何语句失败，引起整个交易回滚。

24.8.3 触发器、交易与ORDER

当两个触发器有相同的触发器动作时间、相同的触发动作、相同的触发器类型时，ORDER子句变得有用了。没有ORDER子句，使用随机顺序。

下面的例子中，两个触发器都是行触发器，都是之后触发，更新相同的列。ORDER子句定义了激活的顺序。

注意：下列被触发语句进行参照完整性检查。

应用ORDER子句的前提是两个触发器有相同的：

- 动作时间 (AFTER)
- 触发器事件 (UPDATE)
- 触发器类型 (STATEMENT)

```
CREATE TRIGGER Trigger_4
AFTER UPDATE OF (col_1) ON Tbl_1  ORDER 1
...  FOR EACH STATEMENT
(DELETE FROM Tbl_2 WHERE.....;);

CREATE TRIGGER Trigger_5
AFTER UPDATE OF (col_1) ON Tbl_1  ORDER 2
...  FOR EACH STATEMENT
(INSERT INTO SELECT ... WHERE.....;);
```

当触发语句被激活时，宏变成：

- UPDATE Tbl_1 SET col_1 = ...
- DELETE FROM Tbl_2
- INSERT INTO Tbl_2

任何语句失败，引起整个交易回滚。

24.8.4 触发器和参照完整性

下列在一个触发动作中的被触发语句进行了参照完整性检查，因此，要确保被触发动作的成功完成，赋予正确的触发器动作时间是非常关键的。

下面的例子中，如果通过更新语句给雇员赋予了一个工作代码，但job表中当前没有该代码，则向job表中插入此新的工作代码。

如果job表中没有相应的工作代码， 由于参照完整性约束，对employee表的更新将会失败。在这种情况下，将触发器动作时间设置为之前，这很重要，这样才能在参照完整性检查之前插入必要的记录。

给Job表增加参照完整性约束：

```
ALTER TABLE employee ADD CONSTRAINT job_1
FOREIGN KEY (job_code) REFERENCES job(job_code);
```

问题

定义一个触发器，如果赋予雇员的工作代码在Job表中不存在，则在Job表中增加该工作代码。

```
CREATE TRIGGER trigger_6
BEFORE UPDATE OF (job_code) ON employee
REFERENCING NEW_TABLE AS newtable
FOR EACH STATEMENT
(INSERT INTO job SELECT job_code, 'New job',Null,Null
        FROM newtable
 WHERE job_code NOT IN (SELECT job_code from job);
);
```

当触发语句被激活时，宏变成：

- INSERT INTO job -- (仅当 job_code 不存在时)
- UPDATE employee SET job_code = ...

任何语句失败，引起整个交易回滚。如果Trigger_6是一个AFTER触发器，由于参照完整性检查，交易失败。

24.9 触发器使用指导

要创建触发器，需要下列权限：

- CREATE TRIGGER 的权限。
- 对主题表有触发动作的权限。
- 对 WHEN 子句中参照的任何表有权限。
- 对参照的表有被触发语句的权限。

与宏不同，触发器没有EXEC权限。如果你有权限激活触发语句，也应该有权限执行被触发语句。

要删除触发器，使用DROP TRIGGER命令。定义了触发器的表不能被删除，必须先删除该表上的所有触发器，才能删除表。

Create/Drop触发器需要的权限：

- 对主题表或数据库有 CREATE/DROP TRIGGER 权限。
- 对 WHEN 子句或被触发语句参照的表有 SELECT 权限。
- 对被触发语句的目标表有 INS、UPD 或 DEL 权限。

执行触发器需要的权限：

- 需要有执行触发语句的各种权限。
- 没有象宏一样的 EXEC 权限。

触发器限制：

- CREATE TRIGGER 的文本限制在 12,500 字节。
- 触发器层叠的层次不限 -- 直到系统资源耗尽。
- 被触发语句不能引用主题表(避免出现无限循环)。

WHEN子句限制：

- WHEN 子句中，只能在子查询中使用聚合函数。
- WHEN 子句中允许使用行级的 SELECT。

DDL限制：

- 在删除参照的表之前，必须先删除触发器。

练习

Lab 24_1

创建一个叫做'NullForKey'的触发器，当从employee表删除一个雇员时，触发器触发一个对department表的更新动作。在删除雇员之前，触发器检查该雇员是否是部门经理，如果是，则将department表中的部门经理雇员号修改为空值(null)。

(要创建这个触发器，应该在你自己的数据库中建立自己的employee表和department表。)

Lab 24_2

测试触发器'NullForKey'。从employee表中删除雇员801，检查department表，看更新是否正确。(提示：雇员801管理部门100)

Lab 24_3

创建一个叫做'AvgSalTrig'的触发器，如果一个雇员的新的薪水超过了部门平均预算的10%，则触发器往日志表中插入一条记录。触发器在修改发生后激活，并且要满足10%的条件。在创建触发器之前，需要先创建表'exceed_10_pcent'，该表包含雇员号、今天的日期、原来的薪水、新的薪水、和置为空(null)的预算百分比，触发器将往该表插入记录。

```
CREATE TABLE exceed_10_pcent
(empnum      INT
,sal_date      DATE
,oldsal         DEC(10,2)
,newsal        DEC(10,2)
,percent_of_budget DEC(4,1)
     );
```

Lab 24_4

测试触发器'AvgSalTrig'。修改雇员1015的记录，薪水增加3%。检查是否往'exceed_10_pcent'表增加了记录。

Lab 24_5

修改Lab 24_3的触发器。除了往日志表增加记录外，还执行更新语句，计算雇员新的薪水占部门预算的百分比。被触发动作包含两条SQL语句。

Lab 24_5

测试新的触发器。删除exceed_10_pcent表中的记录，给雇员1015的薪水增加3% 。检查日志表，显示新的薪水、原来的薪水、薪水占部门预算的百分比。

第二十五章　临时表

完成本章学习后，将能够：

- 创建两种临时表，并装入数据。
- 在不同需求下，决定使用哪种临时表。
- 使用临时表提高应用性能。

25.1 临时表简介

25.1.1 为什么使用临时表

Teradata中有3类临时表：

- 全局临时表(Global Temporay Table)
- 可变临时表(Volatile Temporay Table)
- 导出表(Derived Table)

本章我们讨论全局临时表和可变临时表。

为什么使用临时表？

临时表是一种辅助工具，能够提高SQL操作的性能。特别是针对下列情况的SQL操作：

- 不能使用规范化的表；

- 要求多条 SQL 语句完成。

临时表对于非规范化非常有用，如：

- 汇总表
- 重复分组

临时表对于频繁产生的中间结果或作为后续工作基础的中间结果也非常有用。

怎样创建“永久”临时表

看下面的例子，创建一个永久表，临时保存数据。如果没有临时表，这种方法非常有用。后面，我们将使用全局临时表和可变临时表取代这种方法。

```
CREATE TABLE Daily_Net_Trans
        (Account_Number       INTEGER
        ,Total_Trans_Amount  DECIMAL(14,2))
        UNIQUE PRIMARY INDEX (Account_Number);
```

往表中装入数据

```
INSERT INTO Daily_Net_Trans
      SELECT Account_Number
           ,SUM(Trans_Amount)
      FROM   TRANS
      GROUP BY Account_Number;
```

25.1.2 访问临时表

可以使用SELECT语句访问表

例

```
SELECT *
FROM   Daily_Net_Trans
WHERE  Account_Number IN
     (20035223, 20024048, 20036699, 20045853)
```

结果

Account_Number	Total_Trans_Amount
20035223	-3280.00
20024048	-470.00
20036699	-870.00
20045853	-6910.00

使用临时表的一些好处：

- 简化 SQL 语句。
- 系统不是非得进行聚合操作。
- 系统可以使用主索引访问，响应时间快。

使用永久表作为临时使用的缺点：

- 采用不同的步骤来创建表和给表装入数据。
- 表需要永久空间(Perm space)。
- 表不需要时，要手动删除 - 不能自动进行。
- 创建表和删除表时，要访问数据字典。

25.1.3 选择临时表

本节，我们讨论3类临时表和选择他们的条件。

导出表(Derived Tables)

导出表在Teradata V2R2中已经实现，其特点包括：

- 对查询是本地的 - 存在于整个查询期间，查询结束后，表被丢掉。
- 并入 SQL 查询的语法。
- 查询完成后，Spool 缓冲区的记录被丢掉。
- 不使用数据字典 - 减少系统负载。

可变临时表(Volatile Temporary Tables)

可变临时表在Teradata V2R3中实现，与导出表相比，它有许多优点。其特点包括：

- 对会话(session)是本地的 - 存在于整个会话期间，而不是单个查询。
- 使用 CREATE VOLATILE TABLE 语法创建。
- 会话(session)结束时，自动丢掉。
- 不使用数据字典。

全局临时表(Global Temporary Tables)

全局临时表在Teradata V2R3中实现，与可变临时表的主要区别是，全局临时表在数据字典中有定义，可以被多个用户共享。每个用户会话能够物化自己本地的表的实例。其特点包括：

- 对会话(session)是本地的，但是每个用户会话可以有自己的实例。
- 使用 CREATE GLOBAL TEMPORARY TABLE 语法。
- 会话(session)结束时，物化的表的实例被丢掉。
- 在数据字典中创建并保持表的定义。

25.2 导出表

前面我们已经讨论了导出表，本节再简单回顾一下。

我们回顾前面使用RANK函数的例子。RANK函数与聚合函数不能同时使用，所以我们可以先聚合，结果放入导出表中，在进行排队(RANK)。

问题

获得销售额在前3名的产品，跨所有商店。

解答

```
SELECT t.prodid, t.sumsales, RANK(t.sum sales)
FROM (select prodid, sum(sales) from salestbl
      GROUP BY 1) as t(prodid, sumsales)
QUALIFY RANK(sumsales)<=3;
```

结果

prodid	Sumsales	Rank
A	170000.00	1
C	115000.00	2
D	110000.00	3

上面查询的一些说明：

- 导出表的名字是't'。
- 导出表中的列名是'prodid' 和'sumsales'。
- 使用 SELECT 语句在 spool 缓冲区中创建导出表。
- SELECT 语句放在 FROM 子句之后，并用括号括起来。

在下列情况，最好选择导出表：

- 只有一个查询要求使用临时表，其他查询都不要求。
- 查询结果只使用一次。

25.3 可变临时表

25.3.1 可变临时表

可变临时表与导出表有一些类似的地方：

- 在 spool 缓冲区中物化。
- 不使用数据字典和交易锁。

- 在 cache 中保留表的定义。
- 用于优化性能。

可变临时表也有一些与导出表不同的地方：

- Are local to the session, not the query.
- 在一个会话中，能够被多个查询使用。
- 可以随时被手动删除，会话结束时自动删除。
- 使用 CREATE VOLATILE TABLE 语句创建。

例

```
CREATE VOLATILE TABLE vt_deptsal, LOG
(deptno smallint
,avgsal   dec(9,2)
,maxsal   dec(9,2)
,minsal   dec(9,2)
,sumsal   dec(9,2)
,empcnt   smallint)
ON COMMIT PRESERVE ROWS;
```

上面的例子中，使用ON COMMIT PRESERVE ROWS，允许会话中的其他查询使用这个可变临时表。缺省是ON COMMIT DELETE ROWS，意味着查询提交后，数据被删除。

LOG指示维护交易日志，NO LOG的性能更好。缺省是LOG。

系统重启动后，可变临时表被丢掉了。

25.3.2 可变临时表的约束

使用可变临时表的限制包括：

- 一个会话中，最多有 64 个可变临时表。
- 每个可变临时表必须有唯一的名称。
- 可变临时表必须被会话的用户名限定。

例

CREATE VOLATILE TABLE username.table1 (显式)

CREATE VOLATILE TABLE table1 (隐式)

CREATE VOLATILE TABLE databasename.table1

(如果数据库名不是用户名，则出错)

多个会话

- 不同会话可以使用同样的可变临时表名称 (because it is local to the session).
- 但是可变临时表名不能与此用户已有的对象重名，包括
  - 永久表
  - 临时表
  - 视图
  - 宏

FALLBACK

创建可变临时表时，可以选择FALLBACK，但是它起不到保护的作用，因为数据库重启时，可变临时表会丢掉。

不允许的选项

创建可变临时表时，不允许使用的CREATE TABLE选项包括：

- 永久日志(Permanent Journaling)
- 参照完整性(Referential Integrity)
- 检查约束(Check)
- 列压缩
- 列缺省值
- 列标题
- 命名的索引

25.3.3 使用可变临时表

记住前面我们使用过的可变临时表vt_deptsal。

```
CREATE VOLATILE TABLE vt_deptsal, LOG
(deptno smallint
,avgsal   dec(9,2)
,maxsal   dec(9,2)
,minsal   dec(9,2)
,sumsal   dec(9,2)
,empcnt   smallint)
ON COMMIT PRESERVE ROWS;
```

给表装入数据，然后进行查询。

```
INSERT into vt_deptsal
SELECT dept ,avg(sal) ,max(sal) ,min(sal) ,sum(sal) ,count(emp)
FROM emp
GROUP BY 1;
```

问题

显示各个部门薪水最低的雇员。

解答

```
SELECT emp, last, dept, sal
FROM emp INNER JOIN vt_deptsal
ON dept=deptno
WHERE sal=minsal
ORDER BY 3;
```

结果

emp	last	dept	sal
801	Trainer	100	100000.00
1025	Short	201	34700.00
1008	Kanieski	301	29250.00
1016	Rogers	302	56500.00
1013	Phillips	100	24500.00
1014	Crane	402	24500.00
1009	Lombardo	403	31000.00

1023 Rabbit 501 26500.00

当使用可变临时表时，已经进行了聚合操作。临时表非常小，因此节省了查询时间。在会话结束时，表被自动丢掉，不需要用户或数据库管理员维护。

问题

显示高于各部门平均薪水的雇员。

解答

```
SELECT emp, dept, sal, avgsal
FROM emp INNER JOIN vt_deptsal
ON dept=deptno
WHERE sal>avgsal
ORDER BY 2;
```

结果

emp	dept	sal	avgsal
1021	201	38750.00	36725.00
1019	301	57700.00	38800.00
1010	401	46000.00	35082.14
1004	401	36300.00	35082.14
1002	401	43100.00	35082.14
1003	401	37850.00	35082.14
1011	402	52500.00	38500.00
1007	403	49700.00	38833.33
1020	403	39500.00	38833.33

1024	403	43700.00	38833.33
1015	501	53625.00	50031.25
1017	501	66000.00	50031.25
1018	501	54000.00	50031.25

25.3.4 得到帮助

可以使用HELP VOLATILE TABLE命令获得存在于会话中的所有可变临时表的信息。

例

```
.set foldline on
.set sidetitles on

help volatile table;
```

结果

```
Session ID 3685
Table Name vt_deptsal
Table Id 10C001000000
Protection N
Creator Name PET
Commit Option P
Transaction Log Y
```

也可以使用SHOW TABLE命令查看可变临时表。

show table vt_deptsal;

结果

```
CREATE SET VOLATILE TABLE PED1.vt_deptsal ,NO FALLBACK,
  LOG
  (
  deptno SMALLINT,
  avgsal DECIMAL(9,2)
  maxsal DECIMAL(9,2)
  minsal DECIMAL(9,2)
  sumsal DECIMAL(9,2)
  empcnt SMALLINT)
PRIMARY INDEX (deptno)
ON COMMIT PRESERVE ROWS;
```

注意：HELP DATABASE命令不会显示可变临时表，因为数据字典没有记录可变临时表。

25.3.5 可变临时表的限制

下列命令不能用于可变临时表：

- COLLECT/DROP/HELP STATISTICS
- CREATE/DROP INDEX
- ALTER TABLE
- GRANT/REVOKE privileges

- DELETE DATABASE/USER (不能删除可变临时表)

可变临时表不能：

- 使用存取日志
- 改名
- 使用 Multiload 或 Fastload 实用程序装载

可变临时表不能被视图和宏引用。

例

CREATE MACRO vt1 AS (SELECT * FROM vt_deptsal;);

Session A	Session B
EXEC vt1	EXEC vt1

每个会话都有自己的vt_deptsal的物化实例，所以每个会话都可能返回不同的结果。

可变临时表可以在会话结束前删除。

例

DROP TABLE vt_deptsal;

25.3.6 可变临时表的测验

下面的练习中，用户A有两个并行的会话。假设按照下面的顺序，哪些会成功，哪些会失败，为什么？

(注： CVT = 创建可变临时表， CPT = 创建永久表)

User A

Session 1	**Session 2**	**Success/Failure**
CVT T1	CVT T1	_____
CPT T2		_____
	CVT T2	_____
CVT T3		_____
	CPT T3	_____
INSERT T3		_____
	CVT T3	_____
	INSERT T3	_____
DROP T3		_____
SEL*FROM T3		_____

答案

User A

Session 1	**Session 2**	**Success/Failure/Reason**
CTV T1	CTV T1	Success-each gets local instance
CTP T2		Success-adds perm T2 to Sess1
	CTV T2	Failure-T2 already exists as perm
CTV T3		Success-add VT T3 to Sess1___
	CTP T3	Success-add perm T3 to Sess2__

INSERT T3		Success-inserts into VT T3
	CTV T3	Failure-T3 exists as permtable
	INSERT T3	Success-inserts into perm T3
DROP T3		Success-Drops VT T3
SEL*FROM T3		Success-Selects from perm T3

25.4 全局临时表

25.4.1 全局临时表

全局临时表使用CREATE GLOBAL TEMPORARY命令创建，在数据字典中保存有基础定义。全局临时表被下列第一条SQL语句所物化：

- CREATE INDEX .... ON TEMPORARY .....
- DROP INDEX .... ON TEMPORARY .....
- COLLECT STATISTICS
- DROP STATISTICS
- INSERT
- INSERT SELECT

全局临时表与可变临时表有不同的地方：

- 基础定义是永久的，保存在数据字典中。
- 要物化表，要有相应 SQL 的权限。
- 空间要占用用户的“ 临时空间(temporary space) ”。
- 每个会话最多可以物化 32 全局临时表。
- 系统重启动后，还存在。

全局临时表与可变临时表有相似的地方：

- 对会话而言，每个实例是本地的。

- 会话结束后，物化的表被自动删除。(但基础定义仍然存储在数据字典中)
- 都有 LOG 和 ON COMMIT PRESERVE/DELETE 选项。
- 物化表中内容与其他会话不共享。
- 在会话开始时，表被清空。

25.4.2 创建全局临时表

我们看一个创建全局临时表的例子。

例

```
CREATE GLOBAL TEMPORARY TABLE gt_deptsal
(deptno SMALLINT
,avgsal   DEC(9,2)
,maxsal   DEC(9,2)
,minsal   DEC(9,2)
,sumsal   DEC(9,2)
,empcnt   SMALLINT);
```

缺省是ON COMMIT DELETE ROWS选项。如果希望ON COMMIT PRESERVE ROWS，必须在CREATE TABLE语句中说明。执行后，表的基础定义存储在数据字典中。

ALTER TABLE可以改变表。

例

```
ALTER TABLE gt_deptsal, ON COMMIT PRESERVE ROWS;
```

使用表

```
INSERT INTO gt_deptsal
SELECT dept ,AVG(sal) ,MAX(sal) ,MIN(sal) ,SUM(sal) ,COUNT(emp)
FROM emp
GROUP BY 1;
```

现在表被物化了，并且在DBC.TempTables中插入了一行记录。

```
DELETE FROM gt_deptsal;
```

物化的表仍然存在，除非被删除或会话结束。

25.4.3 空间分配

```
CREATE USER john AS
        PERM=5000000,
        PASSWORD=lucky,
        SPOOL=5000000
        TEMPORARY=2000000;
```

临时空间(Temporary space)先于spool空间被分配，空间都来自于现存的空闲空间。如果没有说明，缺省设置为其直接拥有者的最大值。

```
CREATE SET TABLE dbc.temptables, FALLBACK,
```

```
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL
  (
  HostNo SMALLINT FORMAT'--,-9' NOT NULL,
  SessionNo INTEGER FORMAT '-,--,--,--9' NOT NULL,
  TableId BYTE (6) NOT NULL,
  BaseDbld BYTE (4) NOT NULL,
  BaseTableId BYTE(6) NOT NULL,
  AccountDbId BYTE(4) NOT NULL,
  StatisticsCnt SMALLINT FORMAT '--,--9'NOT NULL)
PRIMARY INDEX(HostNo ,SessionNo);
```

使用了新的数据字典表DBC.Temptables，对于每个全局临时表的物化实例都有一行记录。

25.4.4 得到帮助

可以使用标准的HELP和SHOW命令得到全局临时表的信息。

例

```
HELP DATABASE ped;
```

结果

Table/View/Macro	Kind	Comment
employee	T	?

empsamp	T	?
exceed_10_pcent	T	?
gt_deptsal	T	?

例

HELP TABLE gt_deptsal;

结果

Column Name	Type	Comment
deptno	12	?
avgsal	D	?
maxsal	D	?
minsal	D	?
sumsal	D	?
empcnt	12	?

例

SHOW TABLE gt_deptsal;

结果

```
CREATE SET GLOBAL TEMPORARY TABLE PED.gt_deptsal
,NO FALLBACK,   LOG
  (
```

```
  deptno SMALLINT,
  avgsal DECIMAL(9,2),
  maxsal DECIMAL(9,2),
  minsal DECIMAL(9,2),
  sumsal DECIMAL(9,2),
  empcnt SMALLINT)
PRIMARY INDEX (deptno)
ON COMMIT PRESERVE ROWS;
```

25.4.5 使用全局临时表

我们看一些使用全局临时表的例子。

问题

显示各个部门薪水最高的雇员。

解答

```
SELECT emp, last, dept, sal
FROM emp INNER JOIN gt_deptsal
ON dept=deptno
WHERE sal=maxsal
ORDER BY 3;
```

结果

emp	last	dept	sal

801	Trainer	100	100000.00
1021	Morrissey	201	38750.00
1019	Kubic	301	57700.00
1016	Rogers	302	56500.00
1010	Rogers	401	46000.00
1011	Daly	402	52500.00
1007	Villegas	403	49700.00
1017	Runyon	501	66000.00

问题

显示那些雇员，他们所在的部门拥有的雇员少于3人。

解答

```
SELECT emp, last, dept, empcnt
FROM emp INNER JOIN gt_deptsal
ON dept=deptno
WHERE empcnt < 3 ;
```

解答

emp	last	dept	empcnt
801	Trainer	100	1
1014	Crane	402	2
1025	Short	201	2
1011	Daly	402	2
1021	Morrissey	201	2
1016	Rogers	302	1

25.4.6 全局临时表和数据定义语言

对不同人而言，临时表(Temporary Table)有不同的含义。在Teradata的术语中，临时表指全局临时表(Global Temporary table)，而不是指可变临时表。可变临时表(Volatile Temporary table)可以简称为可变表。

看一些使用全局临时表的数据定义的例子。

删除表

DROP TEMPORARY TABLE gt_deptsal;

只删除表在本地的实例。

DROP TABLE gt_deptsal;

删除表的基础定义和本地实例。如果系统中存在表的其他实例，将失败。

DROP TABLE gt_deptsal ALL;

删除表的基础定义和所有实例。如果有任何实例正处于一个活动的交易中，将失败。

在命令中的关键字TEMPORARY表示只有表的实例。如果没有该关键字，则应用到表的基础定义和实例。

注：TEMPORARY和ALL是互斥的。

删除数据库/用户

DELETE DATABASE db1;

删除数据库中的所有对象，包括全局临时表。如果全局临时表的任何实例被物化了，将失败。

DELETE DATABASE db1 ALL;

删除数据库中的所有对象，包括物化了的全局临时表。

显示表定义

SHOW TABLE gt_deptsal;

显示基础表的定义。

SHOW TEMPORARY TABLE gt_deptsal;

显示临时表的物化实例的定义。

修改表

ALTER TABLE gt_deptsal, ON COMMIT PRESERVE ROWS;

也可用于修改LOG选项。改变表的基础定义。如果存在表的物化的实例，将失败。

视图和宏

CREATE MACRO gt1 AS (SELECT * FROM gt_deptsal;);

视图和宏可以引用全局临时表。如果宏中包含正确的物化SQL命令，则宏的执行能够物化表。往视图中插入数据，也能够物化表。

权限的授予与收回

GRANT INSERT, SELECT ON gt_deptsal TO user1;

只应用到基础定义上。要物化表，也需要适当的权限。 对于基础表或数据库，也需要适当的权限。一旦表被物化后，就不进行权限检查了。

获得索引的帮助

HELP TEMPORARY INDEX gt_deptsal;

显示定义在物化表上的索引。如果没有TEMPORARY，将显示定义在基础表上的索引。

获得统计(Statistics)的帮助

HELP TEMPORARY STATISTICS gt_deptsal;

显示定义在物化表上的统计。如果没有TEMPORARY，将显示定义在基础表上的统计。

25.4.7 次索引

创建次索引

在全局临时表上可以创建次索引。即可以建在基础表上，也可以建在表的实例上。

例

```
CREATE INDEX (empcnt)ON gt_deptsal;
```

创建了基础表的索引。如果表的实例已经存在，将失败。以后物化的实例都将有索引。不能使用命名的索引。

执行SHOW TABLE命令，查看全局临时表gt_deptsal的定义。

例

```
SHOW TABLE gt_deptsal;
```

结果

```
CREATE SET GLOBAL TEMPORARY TABLE PED.gt_deptsal
,NO FALLBACK,LOG
```

```
 (
  deptno SMALLINT,    avgsal DECIMAL(9,2),maxsal DECIMAL(9,2),
  minsal DECIMAL(9,2), sumal DECIMAL(9,2), empcnt SMALLINT)
PRIMARY INDEX (deptno )
INDEX ( empcnt)
ON COMMIT PRESERVE ROWS;
```

创建索引

```
CREATE INDEX (empcnt) ON TEMPORARY gt_deptsal;
```

创建了物化表的索引。如果基础表已经定义了这个索引，将失败。

显示表的定义

```
SHOW TEMPORARY TABLE gt_deptsal;
```

结果

```
CREATE SET GLOBAL TEMPORARY TABLE PED.gt_deptsal
,NO FALLBACK ,LOG
 (
  deptno SMALLINT,   avgsal DECIMAL(9,2),maxsal DECIMAL(9,2),
  minsal DECIMAL(9,2), sumsal DECIMAL(9,2), emptcnt SMALLINT)
PRIMARY INDEX (deptno )
INDEX ( empcnt)
ON COMMIT PRESERVE ROWS;
```

删除次索引

全局临时表的次索引能够删除。基础表和表的实例上的索引都可以删除。

例

DROP INDEX (empcnt) ON TEMPORARY gt_deptsal;

删除物化表上的索引。

例

HELP TEMPORARY INDEX gt_deptsal;

结果

Unique?	Primary or Secondary?	Column Names
N	P	deptno

例

HELP INDEX gt_deptsal;

结果

Unique?	Primary or Secondary?	Column Names
N	P	deptno
N	S	empcnt

注：基础表保持索引的定义。

例

DROP INDEX (empcnt) ON gt_deptsal;

删除基础表的索引。如果表的实例存在，将失败。

25.4.8 STATISTICS

COLLECT STATISTICS

使用下面的表做一些Collecting Statistics的例子。

```
CREATE GLOBAL TEMPORARY TABLE gt_deptsal
(deptno smallint
,avgsal dec(9,2)
,maxsal dec(9,2)
,minsal dec(9,2)
,sumsal dec(9,2)
,empcnt smallint);
```

COLLECT STATISTICS的规则

- 可以针对基础表或表的物化实例。
- 针对基础表 - 定义将收集哪些列。
- 针对实例 - 执行收集，并存储统计。

例

COLLECT STATISTICS on gt_deptsal index (deptno);

定义收集的索引。

COLLECT STATISTICS on gt_deptsal column avgsal;

定义收集的列。

HELP STATISTICS gt_deptsal;

结果

Date	Time	Unique Values	Column Names
98/09/11	13:38:37	0	deptno
98/09/11	13:39:01	0	avgsal

往表中装入数据

```
INSERT INTO gt_deptsal
SELECT dept ,avg(sal) ,max(sal) ,min(sal) ,sum(sal) ,count(emp)
FROM emp
GROUP BY 1;
```

COLLECT STATISTICS on gt_deptsal;

结果

失败。因为表物化后，不允许执行这种操作。没有收集基础表的统计。

COLLECT STATISTICS on temporary gt_deptsal;

收集了临时表实例的统计。

HELP temporary statistics gt_deptsal;

结果

Date	Time	Unique Values	Column Names
98/09/11	13:46:08	8	deptno
98/09/11	13:46:08	8	avgsal

DROP STATISTICS

DROP STATISTICS的规则：

- 可以针对于基础表或表的物化实例。
- 针对基础表 - 从基础定义中删除这些列。
- 针对实例 - 删除表实例的统计。

例

DROP STATISTICS on gt_deptsal;

失败。当表物化以后，不允许执行这种操作。

DROP STATISTICS on temporary gt_deptsal column avgsal;

删除了表实例基于列avgsal的统计。

HELP TEMPORARY STATISTICS gt_deptsal;

结果

Date	Time	Unique Values	Column Names
98/09/11	13:46:08	8	deptno

例

DROP STATISTICS on temporary gt_deptsal;

删除表实例的所有统计。

HELP TEMPORARY STATISTICS gt_deptsal;

没有统计返回。该实例的所有统计已经删除了。

HELP STATISTICS gt_deptsal;

结果

Date	Time	Unique Values	Column Names
98/09/11	13:38:37	0	deptno
98/09/11	13:39:01	0	avgsal

例

COLLECT STATISTICS on temporary gt_deptsal;

失败。该实例没有定义统计。

练习

Lab 25_1

在你自己的空间内创建全局临时表gt_deptsal，提交后保留记录行，不使用LOG选项。表说明如下：

deptno	smallint (as Unique Primary Index)
avgsal	dec(9,2)
maxsal	dec(9,2)
minsal	dec(9,2)
sumsal	dec(9,2)
empcnt	smallint)

Lab 25_2

给表装入数据。选择Customer_Service数据库中employee表的数据，并进行适当的聚合操作。

Lab 25_3

在你自己的空间创建可变临时表vt_emp_job_dept，提交后保留记录行，不使用LOG选项。表说明如下：

empno	int	(as Unique Primary Index)
deptname	char(15)	
job_desc	char(15)	

Lab 25_4

给表装入数据。选择Customer_Service数据库中employee表和job表的数据，确保每个雇员都在表中。

Lab 25_5

显示每个部门内薪水最高的雇员，显示部门名称和工作描述(不显示雇员号和薪水)。使用临时表和可变表解决问题，结果按部门名称排序。

Lab 25_6

显示公司内薪水最低的6个雇员，显示部门名称和工作描述(不显示雇员号和薪水)。结果按部门名称排序。

Lab 25_7

显示公司内占薪水75%的雇员，显示部门名称和工作描述(不显示雇员号和薪水或百分比)。缺省排序。

Lab 25_8

执行Lab 25_7相同的查询，显示薪水和百分比。

Lab 25_9

显示占平均薪水的50% 且雇员数少于6个的部门，显示部门名称、平均薪水、和雇员数。

第二十六章　索引的特殊作用

完成本章学习后，将能够：

- 了解按值排序的索引的作用与实现；
- 了解连接索引的作用与实现；
- 创建连接索引；
- 确定什么时候应该使用聚合连接索引；
- 确定优化器什么时候使用聚合的连接索引。

26.1 按值排序的非唯一次索引

26.1.1 回顾非唯一的次索引

非唯一次索引(NUSI)是Teradata的一种索引，非主索引，索引的列值允许不唯一。典型地，在WHERE子句中使用索引的列，将提高查询性能。创建非唯一次索引，可以使用CREATE TABLE语法与表一起创建，也可以使用CREATE INDEX语法在建表后创建。如果索引不再需要了，可以使用DROP INDEX删除索引。

方法1 – CREATE TABLE 语法

创建employee表，工作代码上建立非唯一次索引。

```
CREATE SET TABLE employee ,FALLBACK ,
(
```

```
employee_number INTEGER,
manager_employee_number INTEGER,
department_number INTEGER,
job_code INTEGER,
last_name CHAR(20) NOT NULL,
first_name VARCHAR(30) NOT NULL,
hire_date DATE FORMAT 'YY/MM/DD' NOT NULL,
birthdate DATE FORMAT 'YY/MM/DD' NOT NULL,
salary_amount DECIMAL(10,2) NOT NULL)
UNIQUE PRIMARY INDEX ( employee_number )
INDEX (job_code);
```

方法2 – CREATE INDEX

employee表已经存在，创建工作代码的非唯一次索引。

```
CREATE INDEX (job_code) ON employee;
```

例 – DROP INDEX语法

删除employee表中工作代码的非唯一次索引。

```
DROP INDEX (job_code) ON employee;
```

创建了非唯一次索引，每个AMP上都建立了一个子表。子表中存储了一些记录，包含每个索引值和基础表记录的记录号(row-id)，子表中记录按照索引值的哈稀值排序存储。这样，按照索引值来查找记录非常方便，但是进行范围搜索，索引

就没有用了。例如，使用上面的索引，查询工作代码为122100的雇员，索引起作用；查询工作代码在122000和123000之间的雇员，索引不起作用。

26.1.2 创建按值排序的非唯一次索引

按值排序的非唯一次索引(Value Ordered NUSI)的索引子表按数据值存储记录，而不是哈稀值。在按照范围查询时，这种索引非常有用。

方法1 – CREATE TABLE 语法

创建employee表，工作代码上建立按值排序的非唯一次索引。

```
CREATE SET TABLE employee ,FALLBACK ,
(
employee_number INTEGER,
manager_employee_number INTEGER, department_number INTEGER,
job_code INTEGER,
last_name CHAR(20) NOT NULL,
first_name VARCHAR(30) NOT NULL,
hire_date DATE FORMAT 'YY/MM/DD' NOT NULL,
birthdate DATE FORMAT 'YY/MM/DD' NOT NULL,
salary_amount DECIMAL(10,2) NOT NULL)
UNIQUE PRIMARY INDEX ( employee_number )
INDEX (job_code) ORDER BY VALUES (job_code);
```

方法2 – CREATE INDEX语法

employee表已经存在，工作代码上创建按值排序的非唯一次索引。

CREATE INDEX (job_code) ORDER BY VALUES (job_code) ON employee;

优化器现在能够对工作代码的索引做范围搜索。

次索引(NUSI)在Teradata数据库系统中是自动维护的，当基础表的记录改变时，次索引子表中对应的记录也改变。可以创建索引，删除索引，收集索引的统计。

26.1.3 按值排序的非唯一次索引的限制

按值排序的非唯一次索引的列必须是：

- 单一的列
- 属于索引定义中的列
- 数字列 – 不允许非数字列
- 长度不能大于 4 个字节 – INT, SMALLINT, BYTEINT, DATE, DEC 是有效的。

注：虽然允许DECIMAL数据类型，但长度不能超过4个字节，不能有小数。

索引覆盖面

如果一个查询只引用了索引中的列，我们称这个索引“ 覆盖 ”了查询。这时，优化器会选择访问索引子表，不会访问基础表。

查询中引用列的地方包括：

- SELECT 列表
- WHERE 子句
- 聚合函数
- GROUP BY
- 表达式

如果索引列出现在WHERE子句的条件中，但索引没有覆盖查询，则优化器将考虑成本，选择一个优化的方案。

优化器选择索引，对用户是透明的，不需要用户干预。但用户可以使用EXPLAIN来查看，是否使用了索引。

26.2 连接索引

连接索引是一种能够提高某些类型查询的性能的索引技术，可以包含一个或多个表中的列。连接索引被创建后，由优化器决定是否使用，用户不能直接访问。

连接索引的目的，是从索引子表提供数据，避免访问基础表。连接索引有3类：

- 多表连接索引 – 预先连接多个表
- 单表连接索引 – 按照外部键的哈稀值分布单个表的记录
- 聚合连接索引 – 聚合一个或多个表中的列，形成汇总表

聚合索引是Teradata V2R4的特征。

26.2.1 多表连接索引

多表连接索引用于预先连接两个或多个表。考虑下面的表：

```
CREATE TABLE customer
( cust_id INTEGER NOT NULL,
cust_name CHAR(15),
cust_addr CHAR(25) )UNIQUE PRIMARY INDEX ( cust_id );

CREATE TABLE orders
( order_id INTEGER NOT NULL,
order_date DATE FORMAT 'yyyy-mm-dd',
cust_id INTEGER,
order_status CHAR(1)) UNIQUE PRIMARY INDEX ( order_id );
```

表中有49 有效的客户有订单，有1个有效的客户没有订单，有1个订单的客户无效。

查询1 – 没有连接索引

有多少订单分配给了客户？

```
SELECT COUNT(order_id) FROM orders
    WHERE cust-id IS NOT NULL;
```

结果

```
Count(order_id)
---------------
```

50

连接索引没有用，order表覆盖了查询。

查询2 – 没有连接索引

有多少订单分配给了有效的客户？

```
SELECT COUNT(o.order_id) FROM customer c INNER JOIN orders o
    ON c.cust_id = o.cust_id;
```

结果

```
Count(order_id)
---------------
49
```

使用连接索引会有帮助，查询使用了两个表。

创建连接索引

创建一个连接索引，索引将提高它覆盖的连接的性能。

```
CREATE JOIN INDEX cust_ord_ix
AS SELECT (c.cust_id, cust_name),(order_id, order_status, order_date)
FROM customer c INNER JOIN orders o
ON c.cust_id = o.cust_id
```

PRIMARY INDEX (cust_id);

连接索引包括两部分：固定部分(第一个括号内) 和可重复部分 (第二个括号内)。这是非规范化的数据，逻辑上可以看作：

CUST_ID	CUST_NAME	ORDER_ID	ORDER_STATUS	ORDER_DATE
1001	ABC Corp	501	C	990120
		502	C	990220
		503	C	990320
		504	C	990420
		505	C	990520
		506	C	990620
1002	BCD Corp	507	C	990122
		508	C	990222
		509	C	990322

现在，我们再执行同样的查询。

查询2 – 使用连接索引

有多少订单分配给了有效的客户？

```
SELECT COUNT(o.order_id) FROM customer c INNER JOIN orders o
    ON c.cust_id = o.cust_id;
```

结果

```
Count(order_id)
---------------
49
```

连接索引覆盖了查询，因此发生作用，不访问基础表就获得了查询结果。通过EXPLAIN获得两种情况的查询成本。

没有连接索引 – .39 秒

使用连接索引 – .17 秒

由于使用连接索引，查询时间减少了50%。

上面的连接索引也可以覆盖其他查询。

查询3 – 使用连接索引

在1999年1月，有多少有效的客户有订单？

```
SELECT COUNT(c.cust_id) FROM customer c INNER JOIN orders o
ON c.cust_id = o.cust_id
WHERE o.order_date BETWEEN 990101 AND 990131;
```

结果

```
Count(order_id)
---------------
49
```

因为order_date也在连接索引中，所以索引也覆盖了这个查询。通过EXPLAIN获得两种情况的查询成本。

没有连接索引 – .40 秒

使用连接索引 – .17 秒

由于使用连接索引，查询时间减少了50%以上。

26.2.2 给连接索引赋予主索引

连接索引有主索引，用于跨AMP进行哈稀分布。通过定义，连接索引可以有非唯一的主索引(NUPI)，不能有唯一主索引(UPI)。

连接索引的主索引可以说明，缺省将第一列作为主索引。在后面的单表连接索引中，我们将会看见给连接索引选择一个主索引的作用。

```
CREATE JOIN INDEX cust_ord_ix AS
SELECT (c.cust_id, cust_name),(order_id, order_status, order_date)
FROM customer c INNER JOIN orders o
ON c.cust_id = o.cust_id
PRIMARY INDEX (cust_id);
```

按值排序的连接索引

我们可以说明连接索引记录在每个AMP上的顺序。通常，记录在每个AMP上按照主索引的哈稀值的顺序。这种顺序查询范围时有局限，可以使用ORDER BY子句改变其顺序。

```
CREATE JOIN INDEX cust_ord_ix AS
SELECT (c.cust_id, cust_name),(order_id, order_status, order_date)
FROM customer c INNER JOIN orders o
```

```
ON c.cust_id = o.cust_id
ORDER BY order_date
PRIMARY INDEX (cust_id);
```

查询3 – 使用按照order_date顺序的连接索引

在1999年1月，有多少有效的客户有订单？

```
SELECT COUNT(c.cust_id) FROM customer c INNER JOIN orders o
ON c.cust_id = o.cust_id
WHERE o.order_date BETWEEN 990101 AND 990131;
```

结果

```
Count(order_id)
---------------
49
```

因为查询访问订单日期的范围，连接索引已经按照订单日期的顺序排列，优化器选择连接索引时，效率会更高。

ORDER BY的列的规则与按值排序的非唯一次索引一样，ORDER BY的列必须是：

- 单一的列
- 属于索引定义固定部分中的列
- 数字列 – 不允许非数字列
- 长度不能大于 4 个字节 – INT, SMALLINT, BYTEINT, DATE, DEC 是有效的。

注：虽然允许DECIMAL数据类型，但长度不能超过4个字节，不能有小数。

26.2.3 给连接索引增加次索引

为了提高性能，可以在连接索引中可以创建非唯一的次索引(NUSI)。

如果索引的记录已经按照其他列排序了，如cust_id，而连接索引只能使用一列排序，因此，我们需要使用其他技术来处理对订单日期范围的查询。

要解决这种问题，可以给连接索引增加一个非唯一的次索引，并按订单日期排序。给连接索引创建非唯一次索引，可以使用CREATE JOIN INDEX语法，也可以使用CREATE INDEX语法在连接索引创建后增加。

方法1 – CREATE JOIN INDEX语法

```
CREATE JOIN INDEX cust_ord_ix AS
SELECT (c.cust_id, cust_name),(order_id, order_status, order_date)
FROM customer c INNER JOIN orders o
ON c.cust_id = o.cust_id
ORDER BY c.cust_id
/* This ORDER BY controls how the rows of the Join Index will be sorted on the AMPs */
PRIMARY INDEX (cust_id)
INDEX (order_date) ORDER BY (order_date)
/* This ORDER BY controls how the rows of the NUSI will be sorted on the AMPs */ ;
```

方法2 – CREATE INDEX 语法

CREATE INDEX (order_date) ORDER BY VALUES (order_date) ON cust_ord_ix;

注：关键字VALUES是可选的。

26.2.4 单表连接索引

创建单表连接索引是为了按照非主索引的列重新哈稀或重新分布记录。重新分布的索引表可以是基础表列的子集。这会显著减少连接处理时表重新分布的成本。

在构造两表的连接计划时，优化器首先要保证连接的记录都分布在同一个AMP上。如果两表按照主索引连接，连接的记录已经分布在同一个AMP上，不需要重新分布数据。如果有一个表不使用主索引连接，就会重新分布数据。

单表连接索引按照连接的索引值预先分布记录，避免在连接时重新分布。

考虑两个表，employee表和department表。

```
CREATE SET TABLE employee ,FALLBACK ,
( employee_number INTEGER,
manager_employee_number INTEGER,
department_number INTEGER,
job_code INTEGER,
last_name CHAR(20) NOT NULL,
first_name VARCHAR(30) NOT NULL,
```

```
hire_date DATE FORMAT 'YY/MM/DD' NOT NULL,
birthdate DATE FORMAT 'YY/MM/DD' NOT NULL,
salary_amount DECIMAL(10,2) NOT NULL)
UNIQUE PRIMARY INDEX ( employee_number );

CREATE TABLE department, FALLBACK
(department_number SMALLINT
,department_name CHAR(30) NOT NULL
,budget_amount DECIMAL(10,2)
,manager_employee_number INTEGER )
UNIQUE PRIMARY INDEX (department_number);
```

查询4 – 选择所有的雇员号、部门号和部门名。

```
SELECT e.employee_number
,d.department_number
,d.department_name
FROM employee e INNER JOIN department d
ON e.department_number = d.department_number;
```

基于department_number列连接两个表，引起employee表重新分布数据。Department表的主索引是department_number，数据是按照department_number分布的；但employee表是按照其主索引employee_number分布的，需要重新分布。

一个加速连接处理的技术就是在employee表上创建单表连接索引。

```
CREATE JOIN INDEX emp_deptno
AS SELECT employee_number, department_number
FROM employee
PRIMARY INDEX (department_number)
```

;

再执行查询4，优化器将考虑是否使用连接索引，能够避免employee表重新分布。使用EXPLAIN，我们能够看见确实使用了连接索引。

查询4 – 选择所有的雇员号、部门号和部门名。

```
EXPLAIN SELECT e.employee_number
,d.department_number
,d.department_name
FROM employee e INNER JOIN department d
ON e.department_number = d.department_number;
```

结果(部分)

```
We do an all-AMPs JOIN step from PED.d by way of a RowHash match scan with
no residual conditions, which is joined to PED.emp_deptno. PED.d and
PED.emp_deptno are joined using a merge join, with a join condition of
("PED.emp_deptno.department_number = PED.d.department_number"). The result
goes into Spool 1, which is built locally on the AMPs.' with low confidence to be
24 rows. The estimated time for this step is 0.18 seconds.
```

26.3 聚合索引

26.3.1 为什么使用聚合索引

汇总表

查询计数、合计或平均值，需要执行聚合。如果表非常大，聚合的开销就很大，查询受影响。传统上，当这种查询运行频繁时，用户创建汇总表提高性能。但是汇总表有一些不利的因素。

汇总表的限制：

- 要求创建独立的表；
- 要求往表中装入数据；
- 数据改变后，要求刷新数据；
- 要求直接访问汇总表，而不是基础表；
- 当汇总表的数据不能刷新时，出现多种结果。

聚合索引

聚合索引就是用于解决上述问题的。 聚合索引类似于前面的连接索引，不同之处在于使用了合计、计数和数据抽取。内部创建了一个非规范化的汇总表，用户不能直接访问，由优化器决定是否使用。

聚合索引不需要用户维护，当基础表的数据更新后，聚合索引自动刷新。所以，当基础表改变时，会有附加的处理工作。

26.3.2 聚合索引的特点

聚合索引与其他连接索引类似的地方：

- 自动维护，无需用户干预；
- 用户不能直接访问；
- 靠优化器选择；
- 定义索引后，不能使用 Multiload 和 Fastload 装载表。

聚合索引与其他连接索引不同的地方：

- 使用 SUM 和 COUNT 函数；
- 允许使用 EXTRACT YEAR 和 EXTRACT MONTH 处理日期。

要创建连接索引，需要下列两种权限中的一种：

- 对数据库或用户有 CREATE TABLE 权限；
- 对基础表有 INDEX 权限。

另外，对基础表还必须有DROP TABLE权限。

26.3.3 没有聚合索引

下面的表在后续例子中将会使用：

```
CREATE SET TABLE PED.daily_sales ,NO FALLBACK ,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL
  (itemid INTEGER
  ,salesdate DATE FORMAT 'YY/MM/DD'
  ,sales DECIMAL(9,2))
```

```
PRIMARY INDEX ( itemid );
```

例

显示产品10的月销售额。

```
SELECT EXTRACT(YEAR FROM salesdate)AS Yr
  , EXTRACT(MONTH FROM salesdate)AS Mon
  , SUM(sales)
FROM daily_sales
WHERE itemid = 10 AND Yr IN ('1997', '1998')
GROUP BY 1,2
ORDER BY 1,2;
```

结果

```
Yr Mon Sum(sales)
--- ---- ---------
1997 1 2150.00
1997 2 1950.00
1997 8 1950.00
1997 9 2100.00
1998 1 1950.00
1998 2 2100.00
1998 8 2200.00
1998 9 2550.00
```

使用EXPLAIN解释上面的查询，显示通过主索引访问daily_sales表。(注：因为没有计算聚合的成本，没有产生查询的最终成本)

```
EXPLAIN SELECT EXTRACT(YEAR FROM salesdate)AS Yr
 , EXTRACT(MONTH FROM salesdate)AS Mon
 , SUM(sales)
FROM daily_sales
WHERE itemid = 10 AND Yr IN ('1997', '1998')
GROUP BY 1,2
ORDER BY 1,2;
```

结果

Explanation

---

1. First, we do a SUM step to aggregate **from PED1.daily_sales** by way of the primary index "PED1.daily_sales.itemid = 10" with a residual condition of ("((EXTRACT(YEAR FROM (PED1.daily_sales.salesdate)))= 1997) OR ((EXTRACT(YEAR FROM (PED1.daily_sales.salesdate)))= 1998)"), and the grouping identifier in field 1. Aggregate Intermediate Results are computed locally, then placed in Spool 2. The size of Spool 2 is estimated with high confidence to be 1 to 1 rows.

2. Next, we do a single-AMP RETRIEVE step from Spool 2 (Last Use) by way of the primary index "PED1.daily_sales.itemid = 10" into Spool 1, which is built locally on that AMP. Then we do a SORT to order Spool 1 by the sort key in spool field1. The size of Spool 1 is estimated with high confidence to be 1 row. The estimated time for this step is 0.17 seconds .

3. Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request. -> The contents of Spool 1 are sent back to the user as the result of statement 1.

26.3.4 使用聚合索引

创建一个连接索引，进行预先聚合。

```
CREATE JOIN INDEX monthly_sales AS
SELECT itemid AS Item
  ,EXTRACT(YEAR FROM salesdate) AS Yr
  ,EXTRACT(MONTH FROM salesdate) AS Mon
  ,SUM(sales) AS SumSales
FROM daily_sales
GROUP BY 1,2,3;
```

执行前面同样的查询，返回同样的结果，但是会发挥聚合索引的作用。

```
SELECT EXTRACT(YEAR FROM salesdate)AS Yr
, EXTRACT(MONTH FROM salesdate)AS Mon
, SUM(sales)
FROM daily_sales
WHERE itemid = 10 AND Yr IN ('1997','1998')
GROUP BY 1,2
ORDER BY 1,2;
```

结果

Yr Mon Sum(sales)

--- ---- ---------

1997 1 2150.00

1997 2 1950.00

1997 8 1950.00

1997 9 2100.00

1998 1 1950.00

1998 2 2100.00

1998 8 2200.00

1998 9 2550.00

使用EXPLAIN解释上面的查询，显示使用了聚合索引。

```
EXPLAIN SELECT EXTRACT(YEAR FROM salesdate)AS Yr
  , EXTRACT(MONTH FROM salesdate)AS Mon
  , SUM(sales)
FROM daily_sales
WHERE itemid = 10 AND Yr IN ('1997', '1998')
GROUP BY 1,2
ORDER BY 1,2;
```

结果

Explanation

---

1. First, we do a SUM step to aggregate **from join index table PED1.monthly_sales** by way of the primary index "PED1.monthly_sales.Item = 10", and the grouping identifier in field 1. Aggregate Intermediate Results are computed locally, then placed in Spool 2. The size of Spool 2 is estimated with low confidence to be 4 to 4 rows.

2. Next, we do a single-AMP RETRIEVE step from Spool 2 (Last Use) by way of the primary index "PED1.monthly_sales.Item = 10" into Spool 1, which is built locally on that AMP. Then we do a SORT to order Spool 1 by the sort key in spool field1. The size of Spool 1 is estimated with low confidence to be 4 rows. The estimated time for this step is 0.17 seconds.

3. Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.

-> The contents of Spool 1 are sent back to the user as the result of statement 1.

因为聚合已经提前计算好，可以从索引中获得，步骤1的成本降低了。步骤2的成本没有变(0.17)。

26.3.5 显示聚合索引

可以使用SHOW JOIN INDEX语句查看聚合索引的定义。

```
SHOW JOIN INDEX monthly_sales;

CREATE JOIN INDEX PED1.monthly_sales ,NO FALLBACK AS
SELECT COUNT(*)(FLOAT, NAMED CountStar )
  ,PED1.daily_sales.itemid (NAMED Item )
  ,EXTRACT(YEAR FROM (PED1.daily_sales.salesdate ))(NAMED Yr )
  ,EXTRACT(MONTH FROM (PED1.daily_sales.salesdate ))(NAMED Mon )
  ,SUM(PED1.daily_sales.sales )(NAMED SumSales )
FROM PED1.daily_sales
GROUP BY PED1.daily_sales.itemid (NAMED Item )
```

```
 ,EXTRACT(YEAR FROM (PED1.daily_sales.salesdate ))(NAMED Yr )
 ,EXTRACT(MONTH FROM (PED1.daily_sales.salesdate ))(NAMED Mon )
PRIMARY INDEX ( Item );
```

上面显示的定义，发生了一些变化：

- 所有列名都加上了数据库一级的限定词。
- 自动增加了计数(COUNT(*) NAMED CountStar)，支持计数聚合操作。
- 如果索引中有 COUNT 和 SUM，那么索引可以直接使用 AVERAGE 。

26.3.6 覆盖查询

创建了连接索引，由优化器来选择是否使用。

按下列格式访问daily_sales表的查询，都将使用monthly_sales索引。

- 年销售额合计
- 月销售额合计
- 年内按月销售额合计
- 销售额总计

如果优化器使用索引产生结果，我们称索引覆盖了查询。

例1

显示产品10的总销售额。

解答

```
SELECT itemid
  ,SUM(sales)
FROM daily_sales
WHERE itemid = 10
GROUP BY 1;
```

结果

```
itemid Sum(sales)
------- ---------
10 16950.00
```

```
 EXPLAIN SELECT itemid
  ,SUM(sales)
FROM daily_sales
WHERE itemid = 10
GROUP BY 1;
```

Explanation (部分)

---

1. First, we do a SUM step to aggregate from join index table PED1.monthly_sales by way of the primary index "PED1.monthly_sales.Item = 10" with no residual conditions, and the grouping identifier in field 1. Aggregate Intermediate Results are computed locally, then placed in Spool 2. The size of Spool 2 is estimated with high confidence to be 1 to 1 rows.

注：例子中，索引monthly_sales覆盖了查询。

例2

显示1998年产品10的销售额。

解答

```
SELECT itemid
  ,SUM(sales)
FROM daily_sales
WHERE itemid = 10 AND EXTRACT (YEAR FROM salesdate) = '1998'
GROUP BY 1;
```

结果

```
itemid Sum(sales)
------- ---------
10 8800.00
```

```
EXPLAIN SELECT itemid
  ,SUM(sales)
FROM daily_sales
WHERE itemid = 10 AND EXTRACT (YEAR FROM salesdate) = '1998'
GROUP BY 1;
```

Explanation (部分)

---

1. First, we do a SUM step to aggregate from join index table PED1.monthly_sales by way of the primary index "PED1.monthly_sales.Item = 10", and the grouping identifier in field 1. Aggregate Intermediate Results are computed locally, then placed in Spool 2. The size of Spool 2 is estimated with high confidence to be 1 to 1 rows.

练习

Lab 26_1

(1) 查看City表和State表的定义。构造一个查询，通过内连接两个表，返回下列信息，结果按state population和city population排序。

City Name	City Population	State Name	State Population
...	...	...	...

(2) 在自己的数据库中创建连接索引sqlxxxx.citystateidx。索引的固定部分包含state name和state population，变动部分包含city name和city population。

(3) 执行查询，查看是否返回同样的结果。使用EXPLAIN，查看是否使用了连接索引。

Lab 26_2

(1) 删除连接索引sqlxxxx.citystateidx。

(2) 修改查询，只显示人口在1百万到3百万之间的states。执行查询。

(3) 重新创建连接索引，索引记录按照state population列顺序存储。

(4) 重新运行查询，查看是否返回同样的结果。使用EXPLAIN，查看是否使用了连接索引。

Lab 26_3

(1) 在连接索引中增加一个非唯一次索引，索引基于city population列，按city population列的值排序。

(2) 重新运行查询，查看是否返回同样的结果。使用EXPLAIN，查看是否使用了非唯一的次索引。

Lab 26_4

(1) 在你自己的数据库中，创建一个聚合索引dept_sals，按部门汇总雇员的薪水。

(2) 显示部门号和部门的薪水汇总，结果按部门号排序。

(3) 使用Explain解释(2)中的查询，是否使用了聚合索引？

(4) 修改(2)中的查询，增加每个部门的平均薪水，叫做Avg_Sal。

(5) 使用Explain解释查询，是否使用了聚合索引？

第二十七章　从已有表创建新表

完成本章学习后，将能够：

- 基于已经存在的表定义，创建空的新表。
- 基于已经存在的表和数据，创建包含数据的新表。
- 从多个表创建新表。

27.1 使用已有的定义创建空表

使用CREATE TABLE AS语法，可以基于已经存在的表创建新表。如果只创建表定义，不复制数据，使用WITH NO DATA选项。

要执行这个操作，用户应该拥有CREATE TABLE权限和对原表的SELECT权限。

例

创建一个新表dept1，该表与department表有同样的定义。

解答

CREATE TABLE dept1 AS department WITH NO DATA;

使用SHOW TABLE命令查看表的定义。

```
SHOW TABLE dept1;


CREATE SET TABLE PED.dept1 ,NO FALLBACK ,
 NO BEFORE JOURNAL,
 NO AFTER JOURNAL
 (
 department_number SMALLINT,
 department_name CHAR(30) CHARACTER SET LATIN NOT CASESPECIFIC
NOT NULL,
 budget_amount DECIMAL(10,2),
 manager_employee_number INTEGER)
UNIQUE PRIMARY INDEX ( department_number )
;
```

27.2 可复制的属性

大部分标准的列属性都可以复制：

- 列名
- 数据类型
- 缺省值
- NOT NULL 约束
- CHECK 约束
- UNIQUE 约束
- PRIMARY KEY 约束

大部分表级的属性可以复制：

- Fallback 选项(仅永久表)
- Journal 选项(仅永久表)
- 所有索引(除了连接索引)

不能复制的属性：

- 参照约束
- 触发器
- 统计(Statistics)

注：属性可以被CREATE TABLE AS语句覆盖。

例

复制表，增加fallback属性，增加次索引。

```
CREATE TABLE dept1, FALLBACK AS department WITH NO DATA
UNIQUE INDEX (department_name);

SHOW TABLE dept1;

CREATE SET TABLE PED1.dept1 ,FALLBACK ,
 NO BEFORE JOURNAL,
 NO AFTER JOURNAL
 (
 department_number SMALLINT,
```

```
    department_name CHAR(30) CHARACTER SET LATIN NOT CASESPECIFIC
NOT NULL,
    budget_amount DECIMAL(10,2),
    manager_employee_number INTEGER)
    PRIMARY INDEX ( department_number )
   UNIQUE INDEX ( department_name );
```

27.3 使用子查询创建表

可以使用子查询创建表，子查询能够限制目标表的列和行。

考虑employee表：

```
SHOW TABLE employee;

CREATE SET TABLE Customer_Service.employee ,FALLBACK ,
    NO BEFORE JOURNAL,
    NO AFTER JOURNAL
    (
    employee_number INTEGER,
    manager_employee_number INTEGER,
    department_number INTEGER,
    job_code INTEGER,
    last_name CHAR(20) CHARACTER SET LATIN NOT CASESPECIFIC NOT
NULL,
    first_name VARCHAR(30) CHARACTER SET LATIN NOT CASESPECIFIC
NOT NULL,
    hire_date DATE FORMAT 'YY/MM/DD' NOT NULL,
```

```
 birthdate DATE FORMAT 'YY/MM/DD' NOT NULL,
 salary_amount DECIMAL(10,2) NOT NULL)
UNIQUE PRIMARY INDEX ( employee_number );
```

例

使用子查询创建表，并选择所需的列。

```
CREATE TABLE emp1 AS
 (SELECT employee_number
 ,department_number
 ,salary_amount
FROM employee) WITH NO DATA;

SHOW TABLE emp1;

CREATE SET TABLE Customer_Service.emp1 ,NO FALLBACK ,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL
  (
  employee_number INTEGER,
  department_number INTEGER,
  salary_amount DECIMAL(10,2) NOT NULL)
PRIMARY INDEX ( employee_number );
```

使用子查询创建表，有一些限制：

- 不允许使用 ORDER BY 子句。
- 所有列或表达式都必须有名字，缺省的或赋予的。

子查询允许使用下列内容：

- 连接(Join)表达式(包括外连接)
- OLAP 函数
- 嵌套子查询

27.4 改列名

可以使用AS子句改列名，也可以使用Teradata的扩展NAMED。

例

下面的例子修改了列名。

```
CREATE TABLE emp1 AS
  (SELECT employee_number AS emp
  ,department_number AS dept
  ,salary_amount AS sal
FROM employee) WITH NO DATA;

SHOW TABLE emp1;

CREATE SET TABLE Customer_Service.emp1 ,NO FALLBACK ,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL
  (
  emp INTEGER,
  ,dept INTEGER
  ,sal DECIMAL(10,2))
PRIMARY INDEX ( emp );
```

完成同样任务的另外一种方法：

```
CREATE TABLE emp1(emp, dept, sal) AS
  (SELECT employee_number
  ,department_number
  ,salary_amount
FROM employee) WITH NO DATA;
```

27.5 改变列属性

目标表的列属性可以修改。

例

下面的例子改变了两列的名字和一列的数据类型。

```
CREATE TABLE dept1 AS
(SELECT department_number AS dept
  ,budget_amount (INTEGER) AS budget
FROM department) WITH NO DATA;

SHOW TABLE dept1;

CREATE SET TABLE Customer_Service.dept1 ,NO FALLBACK ,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL
  (
  dept SMALLINT,
  budget INTEGER)
```

```
PRIMARY INDEX ( dept );
```

例

下面的例子改变了列的名字、属性和数据类型。

```
CREATE TABLE dept1 (dept DEFAULT 0 UNIQUE NOT NULL,
                    budget CHECK (budget > 0) ) AS
(SELECT department_number (INTEGER)
  ,budget_amount (INTEGER)
FROM department) WITH NO DATA;
```

注：数据类型的修改必须在SELECT语句中，不能在参数列表中。

```
SHOW TABLE dept1;

CREATE SET TABLE Customer_Service.dept1 ,NO FALLBACK ,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL
  (
  dept INTEGER NOT NULL DEFAULT 0 ,
  budget INTEGER CHECK ( budget > 0 ))
UNIQUE PRIMARY INDEX ( dept );
```

27.6 使用已有的表创建有数据的表

使用CREATE TABLE AS语法，可以基于一个或多个已经存在的表创建新表。WITH NO DATA选项用于从原表复制数据。

例

创建一个新表dept1，该表与department表定义相同，且包含同样的数据。

```
CREATE TABLE dept1 AS department WITH DATA;
```

验证dept1表中的内容：

```
SELECT department_number AS dept_num
  ,department_name AS dept_name
  ,budget_amount AS budget
  ,manager_employee_number AS mgr
FROM dept1
ORDER BY 1;
```

dept_num	dept_name	budget	mgr
100	president	400000.00	801
201	technical operations	293800.00	1025
301	research and development	465600.00	1019
302	product planning	226000.00	1016
401	customer support	982300.00	1003
402	software support	308000.00	1011
403	education	932000.00	1005
501	marketing sales	308000.00	1017

600 None ? 1099

27.7 子查询中使用连接

子查询中可以使用连接，从多个表中复制数据。

例

创建一个表，显示部门号、部门名、部门经理名称。为了显示部门经理名称，需要连接employee表。

```
CREATE TABLE dept3 AS
(SELECT d.department_number
  , d.department_name
  , e.last_name AS mgr_name
FROM department d INNER JOIN employee e
ON e.employee_number = d.manager_employee_number ) WITH DATA;

SHOW TABLE dept3;

CREATE SET TABLE Customer_Service.dept3 ,NO FALLBACK ,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL
  (
  department_number SMALLINT,
  department_name CHAR(30) CHARACTER SET LATIN NOT CASESPECIFIC
NOT NULL,
  mgr_name CHAR(20) CHARACTER SET LATIN NOT CASESPECIFIC)
```

PRIMARY INDEX (department_number);

SELECT * FROM dept3 ORDER BY 1;

deptartment_number	deptartment_name	Mgr_name
100	president	Trainer
201	technical operations	Short
301	research and development	Kubic
302	product planning	Rogers
401	customer support	Trader
402	software support	Daly
403	education	Ryan
501	marketing sales	Runyan

27.8 使用计算和表达式

目标表的列可以是计算或表达式。

例

创建一个表并装入数据，包括employee_number、last_name、hire_date、birthdate和雇佣年限hire_age。

```
CREATE TABLE emp2(emp,last,hire,birth,hire_age) AS
(SELECT employee_number
  ,last_name
  ,hire_date
  ,birthdate
  ,(hire_date - birthdate) YEAR
```

FROM employee) WITH DATA;

SELECT * FROM emp2 WHERE emp < 1015 ORDER BY 1;

emp	last	hire	birth	hire_age
801	Trainer	73/03/01	45/08/11	27
1001	Hoover	76/06/18	50/01/14	26
1002	Brown	76/07/31	44/08/09	31
1003	Trader	76/07/31	47/06/19	29
1004	Johnson	76/10/15	46/04/23	30
1005	Ryan	76/10/15	55/09/10	21
1006	Stein	76/10/15	53/10/15	23
1007	Villegas	77/01/02	37/01/31	39
1008	Kanieski	77/02/01	58/05/17	18
1009	Lombardo	77/02/01	45/11/15	31
1010	Rogers	77/03/01	35/04/23	41
1011	Daly	77/03/15	49/12/11	27
1012	Hopkins	77/03/15	42/02/18	35
1013	Phillips	77/04/01	63/08/10	13
1014	Crane	78/01/15	60/07/04	17

计算列的数据类型由表达式结果确定。上例中，计算列的数据类型为DECIMAL(15,2)。

SHOW TABLE emp2;

```
CREATE SET TABLE Customer_Service.emp2 ,NO FALLBACK ,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL
```

```
 (
  emp INTEGER,
  last CHAR(20) CHARACTER SET LATIN NOT CASESPECIFIC NOT NULL,
  hire DATE FORMAT 'YY/MM/DD' NOT NULL,
  birth DATE FORMAT 'YY/MM/DD' NOT NULL,
  hire_age DECIMAL(15,2))
PRIMARY INDEX ( emp );
```

27.9 覆盖数据类型

创建时，可以覆盖数据类型。

```
CREATE TABLE emp3(emp,last,hire,birth,hire_age) AS
  (SELECT employee_number
  ,last_name
  ,hire_date
  ,birthdate
  ,(hire_date - birthdate)/365.25 (INTEGER)
FROM employee) WITH DATA;

SELECT * FROM emp3 WHERE emp < 1015 ORDER BY 1;
```

emp	last	hire	birth	hire_age
801	Trainer	73/03/01	45/08/11	27
1001	Hoover	76/06/18	50/01/14	26
1002	Brown	76/07/31	44/08/09	31
1003	Trader	76/07/31	47/06/19	29
1004	Johnson	76/10/15	46/04/23	30
1005	Ryan	76/10/15	55/09/10	21

1006	Stein	76/10/15	53/10/15	23
1007	Villegas	77/01/02	37/01/31	39
1008	Kanieski	77/02/01	58/05/17	18
1009	Lombardo	77/02/01	45/11/15	31
1010	Rogers	77/03/01	35/04/23	41
1011	Daly	77/03/15	49/12/11	27
1012	Hopkins	77/03/15	42/02/18	35
1013	Phillips	77/04/01	63/08/10	13
1014	Crane	78/01/15	60/07/04	17

27.10 设置缺省的标题

在子查询中使用AS子句，也可以设置标题，将会覆盖表原有的标题。Teradata扩展TITLE子句可以起到同样的作用。

例

创建表并装入数据，显示部门401中的每个雇员的月薪水。

```
CREATE TABLE monthly_sal_401 AS
(SELECT employee_number AS emp
  ,salary_amount/12 AS "Monthly Salary"
FROM employee
WHERE department_number = 401) WITH DATA;

SHOW TABLE monthly_sal_401;

CREATE SET TABLE Customer_Service.monthly_sal_401 ,NO FALLBACK ,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL
```

```
 (
  emp INTEGER,
  "Monthly Salary" DECIMAL(10,2))
PRIMARY INDEX ( emp );
```

注：使用这种方法，标题变成了列名。

```
SELECT * FROM monthly_sal_401;
```

emp	Monthly Salary
1003	3154.17
1004	3025.00
1010	3833.33
1001	2127.08
1002	3591.67
1013	2041.67
1022	2691.67

27.11 增加UNIQUE和PRIMARY KEY约束

```
SHOW TABLE department;

CREATE SET TABLE Customer_Service.department ,NO FALLBACK ,
 NO BEFORE JOURNAL,
 NO AFTER JOURNAL
 (
```

```
 department_number SMALLINT,
 department_name CHAR(30) CHARACTER SET LATIN NOT CASESPECIFIC
NOT NULL,
 budget_amount DECIMAL(10,2),
 manager_employee_number INTEGER)
UNIQUE PRIMARY INDEX ( department_number );
```

增加UNIQUE 或PRIMARY KEY约束时，有必要标明NOT NULL，不管原来的列是否标明了NOT NULL。

```
CREATE TABLE dept1 (deptno UNIQUE NOT NULL
 ,deptname PRIMARY KEY NOT NULL
 ,budget
 ,manager)
AS department WITH NO DATA;

SHOW TABLE dept1;

CREATE SET TABLE Customer_Service.dept1 ,NO FALLBACK ,
 NO BEFORE JOURNAL,
 NO AFTER JOURNAL
 (
 deptno SMALLINT NOT NULL,
 deptname CHAR(30) CHARACTER SET LATIN NOT CASESPECIFIC NOT
NULL,
 budget DECIMAL(10,2),
 manager INTEGER)
UNIQUE PRIMARY INDEX ( deptname )
UNIQUE INDEX ( deptno );
```

27.12 可变表和临时表

可变临时表和全局临时表可以使用CREATE AS WITH NO DATA创建；可变临时表可以使用CREATE WITH DATA创建，但不会复制数据；全局临时表不能使用CREATE WITH DATA创建。

可变临时表和全局临时表必须使用单独的数据操作语句装载数据。

例

创建一个全局临时表，并复制department表的数据。

```
CREATE GLOBAL TEMPORARY TABLE dept1 AS (SELECT * FROM department) WITH NO DATA;

SHOW TABLE dept1;

CREATE SET GLOBAL TEMPORARY TABLE Customer_Service.dept1 ,NO FALLBACK ,
  LOG
  (
  department_number SMALLINT,
  department_name CHAR(30) CHARACTER SET LATIN NOT CASESPECIFIC NOT NULL,
  budget_amount DECIMAL(10,2),
  manager_employee_number INTEGER)
PRIMARY INDEX ( department_number )
ON COMMIT DELETE ROWS;
```

注：临时表的缺省选项是ON COMMIT DELETE。这意味着使用数据操作语句装载数据后，数据会立即被删除。为了装载数据，需要修改表的缺省设置。

修改临时表，可以保留记录。

ALTER TABLE dept1, ON COMMIT PRESERVE ROWS;

再装载数据。

INSERT INTO dept1 SELECT * FROM department;

记住：可变临时表和全局临时表不能使用WITH DATA选项来装载数据，必须使用INSERT 或INSERT SELECT来装载数据。

27.13 使用缺省值

查看department表的数据，注意预算有空(null)值。

SELECT department_number, budget_amount FROM department;

department_number	budget_amount
501	308000.00
301	465600.00
201	293800.00

600	?
100	400000.00
402	308000.00
403	932000.00
302	226000.00
401	982300.00

列定义中可以说明缺省值，不论是否有WITH DATA 或WITH NO DATA选项。

例

使用CREATE WITH NO DATA 创建表，显示department_number 和budget_amount。Budget_amount必须有值(NOT NULL)，缺省值为0(DEFAULT 0)。

```
CREATE TABLE dept1 (deptno, budget NOT NULL DEFAULT 0)
AS (SELECT department_number
      ,budget_amount FROM department) WITH NO DATA;

SHOW TABLE dept1;

CREATE SET TABLE Customer_Service.dept1 ,NO FALLBACK ,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL
  (
  deptno SMALLINT,
  budget DECIMAL(10,2) NOT NULL DEFAULT 0.00 )
PRIMARY INDEX ( deptno );
```

装入数据

INSERT INTO dept1 SELECT department_number, budget_amount FROM department;

结果

*** Failure 3604 Cannot place a null value in a NOT NULL field.

可以使用COALESCE函数解决上面的问题，将空值转变成0。

INSERT INTO dept1 SELECT department_number, COALESCE(budget_amount,0) FROM department;

SELECT * FROM dept1;

deptno	budget
501	308000.00
301	465600.00
201	293800.00
600	.00
100	400000.00
402	308000.00
403	932000.00
302	226000.00
401	982300.00

例

使用CREATE AS完成上面同样的任务，创建表并装入数据。将budget_amount转变成DEC(10,2)类型。

```
CREATE TABLE dept1 (deptno, budget NOT NULL DEFAULT 0)
AS (SELECT department_number
      ,budget_amount (DEC(10,2))
   FROM department) WITH DATA;
```

结果

```
*** Failure 3604 Cannot place a null value in a NOT NULL field.
```

失败是由于试图往NOT NULL列插入空值。使用COALESCE函数解决上面的问题。

```
CREATE TABLE dept1 (deptno, budget NOT NULL DEFAULT 0)
AS (SELECT department_number
      ,COALESCE(budget_amount,0) (DEC(10,2))
   FROM department) WITH DATA;

SHOW TABLE dept1;

CREATE SET TABLE Customer_Service.dept1 ,NO FALLBACK ,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL
```

```
 (
  deptno SMALLINT,
  budget DECIMAL(10,2) NOT NULL DEFAULT 0.00 )
PRIMARY INDEX ( deptno );
```

练习

Lab 27_1

基于employee表创建新表，并装入数据。新表包含部门403的雇员，只有下面3列。

雇员号 - 名字为empno，数据类型为SMALLINT，有UNIQUE约束

雇员last name - 名字为lastnm，数据类型为CHAR(10)

雇员周薪水 - 名字为weekly_sal

创建表后，SHOW查看表，SELECT查询所有记录，按雇员号排序。

Lab 27_2

创建一个表并装入数据，显示每个部门的最高薪水。将部门号改为dept，最高薪水改为maxsal，并缺省的标题'Max//Sal'。使用一条查询employee表的语句完成，SHOW查看表，SELECT查询所有记录。

Lab 27_3

创建一个表并装入数据，该表叫做emp_age，包含下面列：

雇员号 - 名字为empno

部门名 - 名字为deptname

雇员年龄 - 名字为age，计算值为'(DATE - birthdate) YEAR'

要创建该表，需要连接employee表和department表。

创建表后，SHOW查看表，SELECT查询年龄大于50的雇员记录，按雇员号排序。

第二十八章　存储过程(一)

完成本章学习后，将能够：

- 创建、编译、执行存储过程；
- 在存储过程中实现游标处理；
- 创建带参数的存储过程。

28.1 存储过程基础

28.1.1 什么是存储过程

存储过程是定义在Teradata数据库或用户空间中的对象，是可以执行的，包含两种类型的语句：

- SQL 语句 (Structured Query Language)
- SPL 语句 (Stored Procedure Language)

SQL语句用于访问Teradata数据库中一个或多个表中的记录。SPL语句对执行SQL语句增加了过程控制。

SPL提供了大多数第三代语言的功能：

- 分支逻辑
- 条件逻辑
- 错误处理逻辑
- 退出逻辑

Teradata存储过程的其他特点包括：

- 可以包含输入/输出参数
- 可以包含处理例外情况的例外处理程序
- 可以包含说明的局部变量
- 通常包含 SQL 语句，但不要求

28.1.2 存储过程的特点

存储过程有下列特点：

- 存储在 Teradata 服务器端。
- 在 Teradata 服务器端编译和执行。
- 编译后的对象存储在存储过程表中。
- 源代码也可以存储在存储过程表中。
- 要求占用永久空间(perm space)。

使用SQL， 用户可以对存储过程执行下列操作：

- CREATE PROCEDURE
- REPLACE PROCEDURE
- RENAME PROCEDURE
- DROP PROCEDURE
- SHOW PROCEDURE

下列工具支持存储过程：

- BTEQ
- CLIv2
- PP2
- ODBC
- QUERYMAN

- TeqTalk(DMTEQ)
- JDBC
- ANSI SQL99 (SQL3)
- PSM-96

28.1.3 存储过程的优势

传统的SQL请求来自客户机端，与之不同，存储过程包含的SQL请求来自服务器端，并且在服务器端处理。SQL请求被服务器端的数据库管理软件激活，传递给解析器(Parser)，请求和应答被创建、处理，并可以返回给存储过程。存储过程将结果集、状态码或计算结果返回给客户端应用。

使用存储过程有许多好处：

- 存储过程减少了客户机和服务器之间的网络流量；
- 将请求处理移到服务器端，所有处理到在服务器本地进行；
- 允许在服务器端定义和执行业务规则；
- 提供更好的交易控制；
- 提供更好的应用安全性。

28.2 调用存储过程

28.2.1 带参数的存储过程

在存储过程中，可以定义3类参数：

- IN - 执行时，给存储过程提供值

- OUT - 调用完成后，值从存储过程返回给用户
- INOUT - 既给存储过程提供值，也从存储过程返回值

存储过程最多可以包含1024个参数，参数可以定义为任何Teradata的数据类型。

例

下列存储过程测试一个输入参数，输出计算结果。

```
CREATE PROCEDURE test_value
  (IN p1 INTEGER, OUT pmsg CHAR(30))
BEGIN
  IF p1 > 0 THEN
    SET pmsg = 'Positive value';
  ELSEIF p1 = 0 THEN
    SET pmsg = 'Zero Value';
  ELSE
    SET pmsg = 'Negative Value';
  END IF;
END;
```

这个例子示范了创建存储过程的某些规则：

- IN、OUT、INOUT 参数必须在过程名后，说明数据类型。
- 存储过程主体必须包含在 BEGIN 和 END 语句之间。
- IF 语句可以有多个 ELSEIF 和一个 ELSE 条件。
- IF 语句必须以 END IF 语句结束。

- SET 语句用于给参数或说明的变量赋值。
- SET、END IF 和 END 语句都以分号";"结束。

28.2.2 调用存储过程

要执行存储过程，使用CALL语句。

例

执行存储过程test_value。

CALL test_value(3, pmsg);

结果

pmsg

Positive value

上面的例子示范了执行存储过程的某些例子：

- 所有定义的参数，在 CALL 语句中都必须表示。
- 使用过程中定义的参数名说明输出参数。
- 输出参数被输出，就象表中的一列。

CALL test_value(0, pmsg);

pmsg

Zero Value

CALL test_value(-1, pmsg);

pmsg

Negative Value

Set语句

Set语句用于给说明的变量和参数赋值。Set语句的格式如下：

```
SET A = B;
```

A或B 都可以是说明的变量。

A 或B都可以是INOUT参数。

A 不能是IN参数。

B不能是OUT参数。

例

假设下面的过程：

```
CREATE PROCEDURE test_set (IN a INT, INOUT b INT, OUT c INT)
BEGIN DECLARE d INTEGER;

SET a = b; (invalid)
SET b = c + 1; (invalid)
SET d = b + c; (invalid)

SET b = a; (valid)
SET b = b + 1; (valid)
SET c = b + a; (valid)
SET d = b + a; (valid)
```

28.2.3 调用参数选项

存储过程被CALL语句激活，过程要求的参数在CALL语句中说明，并且与过程中定义的参数一一对应。

考虑下列过程：

```
CREATE PROCEDURE call_test1 (IN V1 INTEGER, INOUT V2 INTEGER,
OUT V3 INTEGER)
BEGIN
  SET V3 = V1 + 2;
  SET V2 = V2 * V2;
END;
```

CALL call_test1 (3,4,V3);

4	V3
16	5

参数可以使用常量、变量和表达式。

CALL call_test1 (3+2, 5*3, V3);

(5*3)	V3
225	7

28.2.4 转换调用参数

IN、INOUT或OUT参数使用CAST转换，获得下列属性：

- 数据类型
- FORMAT
- TITLE
- NAMED

看下列参数转换的例子。

CALL call_test1 (10, CAST(5 AS TITLE 'INOUT Output'), CAST(V3 AS CHAR(4)));

```
INOUT   Output V3
----------- ----
25          12
```

CALL call_test1 (10, CAST(5 AS FORMAT '99.99'), CAST(V3 AS NAMED V3_Output));

```
5           V3_Output
----        ---------
25.00       12
```

参数中可以使用空值。可以使用NULL，或者不对参数初始化。

CALL call_test1 (3,NULL,V3);

```
Null        V3
-----       ----
?           5
```

CALL call_test1 (3,V2,V3);

```
V2          V3
----        ----
?           5
```

28.2.5 使用宿主变量作为参数

参数值可以从数据文件中输入。USING子句能够从外部文件输入数据值，传递给宿主变量。

宿主变量的数据来自外部的客户端请求。典型地，是一个外部文件，其数据可以用于SQL参数。宿主变量前面都有一个冒号":"。

看下列BTEQ脚本，使用带参数的CALL语句，从外部文件获得参数的数据，

```
.IMPORT DATA FILE = xyz; /* Designates the external file */
.REPEAT *; /* Read each record in the file sequentially */

USING (a INT, b INT) CALL call_test_1(:a, :b, V3);
        /* Execute the procedure for each record read */
```

28.2.6 过程调用过程

过程能够包含CALL语句调用其他过程，两个过程之间允许传递参数值。

例1

创建一个过程，接收一个输入值，计算输出值。

```
REPLACE PROCEDURE test_call6(IN w1 INTEGER, OUT w2 INTEGER)
BEGIN
  SET w2 = 100/w1 + 100;
```

```
END;
```

测试过程：

```
CALL test_call6 (4, w2);

v2
--------
125
```

例2

创建一个过程，接收一个输入值，加2后，传递给过程test_call6。

```
REPLACE PROCEDURE test_call5(IN v1 INTEGER, OUT v2 INTEGER)
BEGIN
  DECLARE a INTEGER;
  SET a = v1 + 2;
  CALL test_call6(:a,:v2);
END;
```

测试过程：

```
CALL test_call5 (2, v2);

v2
--------
125
```

如果被调用过程发生了错误，被调用过程和调用过程都终止。

```
CALL test_call5 (0, v2);
*** Failure 2618 test_call6:Invalid calculation: division by zero.
```

被调用的过程必须在调用过程之前创建，否则出错。

```
.compile file=test_call5;
Error 5495 Stored Procedure test_call6 does not exist. 5495;
```

例3

创建一个过程，接收一个输入值，加100后，输出结果。

```
REPLACE PROCEDURE test_call2(IN w1 INTEGER, OUT w2 INTEGER)
BEGIN
  SET w2 = w1 + 100;
END;
```

例4

创建一个过程，接收一个输入值，加2后，传递给过程test_call2。

```
REPLACE PROCEDURE test_call4(IN v1 INTEGER, OUT v2 INTEGER, OUT
v3 INTEGER)
```

```
BEGIN
  SET v2 = v1 + 2;
  CALL test_call2(:v2,:v3);
END;
```

```
CALL test_call4 (3, v2, v3);
```

```
v2          v3
----        ----
5           105
```

28.2.7 CALL语句的其他考虑

- 存储过程只能在创建的平台(Unix 或 Windows)上调用。
- 存储过程不能调用自己。
- CALL 能够在 ANSI 模式或 Teradata 模式— 非两阶段提交模式下执行。
- CALL 只能够在创建过程的会话模式(ANSI 或 Teradata)下执行。
- 一条 CALL 语句必须是宏内的语句。
- 一条 CALL 语句不能是多语句请求的一部分。
- CALL 语句嵌套数目限制为 15。

28.2.8 存储过程包含SQL

例

下列存储过程在条件满足时，往表中插入记录。

```
CREATE PROCEDURE sp_log (INOUT p1 INTEGER)
BEGIN
Label_One:
  WHILE p1 > 0 DO
    SET p1 = p1 - 1;
    IF p1 > 5 THEN ITERATE Label_One;
    END IF;
    INSERT logtable (:p1);
  END WHILE Label_One;
END;
```

上面的例子示范了创建存储过程的一些规则：

- 当条件满足时，WHILE 语句用于控制后面的语句执行。
- END WHILE 语句用于定义 WHILE 条件的界限。
- 可以随意定义标签(如 Label_One)，说明过程的逻辑部分。
- 在此例中，标签定义了整个 WHILE 循环。
- ITERATE 语句的作用象'GO TO'语句。
- 在此例中，ITERATE 用于回到 WHILE 条件前。
- 当 SQL 语句中使用参数(p1)时，它被当作一个宿主变量，前面带冒号。
- ITERATE 和 END WHILE 语句用分号结束。

28.2.9 调用例子

例1

执行存储过程sp_log，输入参数大于5。

```
CREATE PROCEDURE sp_log (INOUT p1 INTEGER)
BEGIN
Label_One:
  WHILE p1 > 0 DO
    SET p1 = p1 - 1;
    IF p1 > 5 THEN ITERATE Label_One;
    END IF;
    INSERT logtable (:p1);
  END WHILE Label_One;
END;
```

```
CALL sp_log(10);
```

结果

```
10
----
0
```

上面的调用示范了执行存储过程的规则：

- INOUT 参数在执行时说明，在完成时返回。
- INOUT 输出的标题对应调用时说明的参数。
- 在此存储过程中，如果 p1 大于等于 0，则 p1 最后的结果是 0。
- 所有的 SQL 语句必须以分号结束。

- 过程内的 SQL 语句每次经过时，都会执行一次。

确认日志表中插入的记录：

```
SELECT * from LOGTABLE ORDER BY 1;
```

```
x
----
0
1
2
3
4
5
```

例2

执行存储过程sp_log，输入参数小于5。

```
CALL sp_log(3);
```

结果

```
3
----
0
```

```
SELECT * FROM logtable ORDER BY 1;
```

x

0

1

2

28.3 存储过程的权力和权限

新的存储过程

要使用CREATE PROCEDURE创建新的过程，需要在创建过程的数据库上有CREATE PROCEDURE的权限。

用户DBC或拥有带WITH GRANT OPTION的CREATE PROCEDURE权限的用户能够给你授权。

创建者还必须拥有过程中引用的数据库对象的适当的权限。

已有的存储过程

要使用REPLACE PROCEDURE修改已有的存储过程，需要对数据库或过程有DROP PROCEDURE权限。

DROP PROCEDURE权限可以在数据库、用户或存储过程一级授予或取消。DROP权限自动给所有用户。

要调用存储过程，需要有EXECUTE PROCEDURE权限。该权限可以在数据库、用户或过程一级授予。

例

给用户SQL01授予在数据库Sales中创建过程的权力。

GRANT CREATE PROCEDURE ON Sales TO SQL01;

给用户SQL01授予执行存储过程Sales_App_1的权力。

GRANT EXECUTE PROCEDURE ON PROCEDURE Sales.Sales_App_1 TO SQL01;

取消用户SQL01从数据库Sales中删除过程的权力。

REVOKE DROP PROCEDURE ON Sales FROM SQL01;

28.4 使用LOOP语句

- 可以使用 LOOP 语句，来定义重复执行的顺序。
- 可以选择定义一个标签。
- LEAVE 语句允许退出循环。
- 没有 LEAVE 语句，循环是无限的循环。

- 如果在循环中出现例外条件，并且没有定义例外处理，则循环和存储过程被终止。
- DECLARE 语句用于说明局部变量及其数据类型和可选的缺省值。

例

创建一个循环的过程，往日志表中插入记录。

```
CREATE PROCEDURE sp_loop ()
BEGIN
  DECLARE vcount INTEGER DEFAULT 0;
  loop_label: LOOP
    SET vcount = vcount + 1;
    IF vcount > 5 THEN
        LEAVE loop_label;
    END IF;
    INSERT logtable (:vcount);
  END LOOP loop_label;
END;
```

此过程没有定义输入和输出参数。执行过程：

```
CALL sp_loop();

*** Procedure has been executed.
*** Total elapsed time was 1 second.
```

```
SELECT * FROM logtable ORDER BY 1;
```

```
x
----
1
2
3
4
5
```

28.5 游标

28.5.1 声明游标

在存储过程内，当SQL语句希望返回多行数据记录时，有必要声明游标。游标能够一次一行地处理返回的结果。下面的例子中显示了游标的语法：

```
FOR loopvar AS cur1 CURSOR FOR
  SELECT employee_number, department_number FROM employee
DO PRINT 'EmpNo:', loopvar.employee_number;
END FOR;
```

上面的例子中示范了游标的规则：

- 声明游标，需要使用 FOR 语句。
- 要赋予游标一个名字，例子中的名字为 cur1。
- 赋予循环一个名字 loopvar。

- SELECT 语句返回多行。
- DO 语句定义对返回的每行进行的操作。
- PRINT 语句表明打印到数据库窗口中。主要用于调式存储过程。
- 列名必须以循环名为限定词。(如 loopvar.employee_number)
- END FOR 说明 FOR 循环的边界。

28.5.2 游标的例子

例

创建一个存储过程，返回所有雇员号小于1005的雇员，并返回记录行数。

```
REPLACE PROCEDURE sp_cur5 (OUT prowcount INTEGER)
BEGIN
  DECLARE newvar BYTEINT DEFAULT 0;
  FOR loopvar AS cur1 CURSOR FOR
    SELECT employee_number AS emp, department_number AS dept FROM
employee
    WHERE employee_number < 1005 ORDER BY 1
  DO
    SET newvar = newvar + 1;
    PRINT 'Row Count:' , newvar , 'EmpNo:', loopvar.emp;
  END FOR;
  SET prowcount = newvar;
END;
```

上面的例子示范了存储过程中使用游标的规则：

- END FOR 语句以分号结束。
- SQL 语句是游标声明的一部分，不以分号结束。
- 游标内不允许使用多语句请求，只能是单 SQL 语句。

执行过程：

CALL sp_cur5(prowcount);

prowcount

5

输出变量(prowcount)在调用时必须使用定义的名字，否则出错。过程返回5行记录。

为了查看PRINT语句的输出结果，需要访问数据库窗口。这要求有权运行控制台实用程序。如果有权限，你可以在UNIX系统上使用下列命令访问数据库窗口：

cnsterm 5

结果

PED1 | 0 1251 | PED1.SP_CUR5 | 9 : 2000-06-19 22:33:37.760000-03:00| Row Count: 1

EmpNo: 801

PED1 | 0 1251 | PED1.SP_CUR5 | 9 : 2000-06-19 23:33:37.780000-03:00| Row Count: 2

EmpNo: 1001

PED1 | 0 1251 | PED1.SP_CUR5 | 9 : 2000-06-20 00:34:37.800000-03:00| Row Count: 3

EmpNo: 1002

PED1 | 0 1251 | PED1.SP_CUR5 | 9 : 2000-06-20 01:35:37.830000-03:00| Row Count: 4

EmpNo: 1003

PED1 | 0 1251 | PED1.SP_CUR5 | 9 : 2000-06-20 02:36:37.860000-03:00| Row Count: 5

EmpNo: 1004

PRINT语句输出的顺序如下：

用户名 - (PED1)

会话号 - (01251)

过程名 - (SP_CUR5)

PRINT语句行号 - (9)

时间戳 - (2000-06-20 02:36:37.860000-03:00)

用户的输出串 - (Row Count: 5 EmpNo: 1004)

在开发存储过程时，PRINT语句是非常有用的调试工具。

28.5.3 使用ACTIVITY_COUNT

获得游标处理的记录行数的另外一个方法是，使用保留字ACTIVITY_COUNT。这是一个系统变量，表示任何时刻SQL语句在该时刻处理的记录数。

例

创建一个存储过程，返回所有雇员号小于1005的雇员，并返回记录行数。

```
REPLACE PROCEDURE sp_cur5a (OUT prowcount INTEGER)
BEGIN
  FOR loopvar AS cur1 CURSOR FOR
    SELECT employee_number AS emp, department_number AS dept FROM
employee
    WHERE employee_number < 1005 ORDER BY 1
  DO
    PRINT 'Row Count:' , ACTIVITY_COUNT , 'EmpNo:', loopvar.emp;
  END FOR;
  SET prowcount = ACTIVITY_COUNT;
END;

CALL sp_cur5(prowcount);

prowcount
-----------
```

5

查看数据库窗口中的显示结果。

cnsterm 5

PED1 | 0 1255 | PED1.SP_CUR5 | 9 : 2000-06-22 22:33:40.260000-03:00|
Row Count: 1 EmpNo: 801
PED1 | 0 1255 | PED1.SP_CUR5 | 9 : 2000-06-22 23:33:40.290000-03:00|
Row Count: 2 EmpNo: 1001
PED1 | 0 1255 | PED1.SP_CUR5 | 9 : 2000-06-23 00:34:40.320000-03:00|
Row Count: 3 EmpNo: 1002
PED1 | 0 1255 | PED1.SP_CUR5 | 9 : 2000-06-23 01:35:40.350000-03:00|
Row Count: 4 EmpNo: 1003
PED1 | 0 1255 | PED1.SP_CUR5 | 9 : 2000-06-23 02:36:40.380000-03:00|
Row Count: 5 EmpNo: 1004

28.6 SELECT INTO - 返回一行

当SELECT语句只返回一行数据时，没有必要使用游标。因为游标提供了一个浏览机制，能够顺序处理返回的数据记录。单行记录，没有必要使用这种机制。

SELECT INTO设计用于处理单行数据记录。

REPLACE PROCEDURE sel_into (IN empno INTEGER, OUT deptnum INTEGER, OUT lastnm CHAR(20))

```
BEGIN
  SELECT department_number , last_name
  INTO :deptnum, :lastnm
  FROM employee
  WHERE employee_number = :empno;
END;
```

上面的例子中示范了存储过程中使用SELECT INTO的规则：

- INOUT 和 OUT 参数可以用作 INTO 变量(不能使用说明的变量)。
- IN 和 INOUT 参数可以用作 WHERE 子句的变量(不能使用说明的变量)。
- INTO 和 WHERE 子句变量是宿主变量，前面必须带冒号。
- SQL 以分号结束。

执行过程：

```
CALL sel_into (1008,deptnum,lastnm);
```

结果

```
deptnum   lastnm
-------   -------
301   Kanieski
```

如果没有记录返回，输出参数为空值。

如果SELECT INTO中返回多行数据记录，将发生错误。

例

返回给定部门的所有雇员记录。

```
REPLACE PROCEDURE sel_many_into (IN deptnum INTEGER, OUT lastnm
CHAR(20))
BEGIN
  SELECT last_name INTO :lastnm FROM employee
  WHERE department_number = :deptnum;
END;

CALL sel_many_into(301,lastnm);
*** Failure 7627 sel_many_into:SELECT-INTO returned more than one row.
```

注：使用SELECT INTO，当返回多行记录时出错。

28.7 编译

28.7.1 编译存储过程

存储过程必须被驻留在服务器端的SPL编译器编译，并要求使用C编译器。编译输出的结果存储在用户数据库的表中，也可以选择将源代码存储到表中。在BTEQ中，编译命令如下：

```
.COMPILE FILE = sp_source;
```

执行命令后，会出现表示过程创建了的信息，或返回语法错误。缺省地，源代码回作为编译输出的一部分存储起来。如果不要求将源代码存储在服务器上，则使用下列编译命令：

```
.COMPILE FILE = sp_source WITH NOSPL;
```

注：如果没有保存SPL源代码，SHOW PROCEDURE命令不能返回源代码。

如果源代码包含PRINT语句，编译命令如下：

```
COMPILE FILE = sp_source WITH PRINT;
```

缺省，PRINT语句是禁止的。

28.7.2 编译和权限

创建存储过程

为了存储过程编译成功，要检查SQL语句的语法和创建者的访问权限。如果有语法或权限的问题，编译器会返回错误。

调用存储过程

调用存储过程的用户，不需要有被调用过程引用的对象的权限。但用户必须对存储过程或数据库有EXECUTE PROCEDURE权限。

当调用一个过程时，检查数据库对象拥有者(不是调用者或创建者)的权限。拥有者必须对引用的对象有带WITH GRANT OPTION的权限，允许其他用户访问或操作被引用的对象。

存储过程编译成功了，并不意味着执行也成功。下面的任何事件都可能在编译和执行之间发生：

- 表、视图、数据库的权限可能被取消。
- 表或视图中的列可能被删掉。
- 表、视图、数据库可能被删除。

另外，也可能出现运行错误，这是编译时无法检测的(如，被0除)。

28.7.3 编译使用ODBC和JDBC的存储过程

当创建使用ODBC或JDBC的存储过程时，编译在创建过程时自动进行。用户只提交CREATE PROCEDURE语句，如果创建成功，存储过程就可以执行了。在BTEQ中，用户必须显式提交一个编译步骤。

因为PRINT和SPL选项是编译时的选项，不能在创建时使用。例如，在ODBC环境下，这些选项可以被ODBC管理员设置如下：

ProcedureWithPrintStmt (N/P)
ProcedureWithSPLSource (Y/N)

缺省设置是：

ProcedureWithPrintStmt =N

ProcedureWithSPLSource=Y

这些选项不能在用户级被替换。如果要求不同的选项，必须由ODBC管理员实现。

注：Queryman 5.1.0是支持存储过程的ODBC客户端软件。

28.8 例外发生

缺省，当过程中发生例外时，过程调用被终止。如果我们稍微修改以下前面的过程，增加一条PRINT语句(在引起错误的SELECT之后)，PRINT语句不能被执行。

```
REPLACE PROCEDURE sel_many_into_2 (IN deptnum INTEGER, OUT lastnm CHAR(20))
BEGIN
  SELECT last_name
  INTO :lastnm
  FROM employee
  WHERE department_number = :deptnum;

  PRINT 'Successful Completion';
END;
```

```
CALL sel_many_into_2(501,lastnm);
```

** Failure 7627 sel_many_into_2:SELECT-INTO returned more than one row.

注：PRINT没有执行。如果过程使用WITH PRINT选项编译，再次调用时，只执行PRINT语句。PRINT语句只用于调试。

28.9 得到存储过程的帮助

HELP PROCEDURE命令返回存储过程定义的信息。命令有两种形式：

```
HELP PROCEDURE procedurename
HELP PROCEDURE procedurename ATTRIBUTES
```

前面使用的存储过程：

```
REPLACE PROCEDURE sel_many_into_2 (IN deptnum INTEGER, OUT lastnm CHAR(20))
BEGIN
  SELECT last_name INTO :lastnm FROM employee
  WHERE department_number = :deptnum;

  PRINT 'Successful Completion';
END;
```

在BTEQ环境使用HELP命令。

```
.FOLDLINE ON
```

```
.SIDETITLES ON
HELP PROCEDURE sel_many_into_2;
```

Parameter Name	deptnum	(1st Parameter)
Type	I	(Integer)
Comment	?	(No Comments)
Nullable	Y	(Allows Nulls)
Format	-(10)9	(Default Format)
Title	?	(No Title)
Max Length	4	(Four Bytes Max Length)
Decimal Total Digits	?	(Not A Decimal)
Decimal Fractional Digits	?	(Not A Decimal)
Range Low	?	(No Range)
Range High	?	(No Range)
UpperCase	N	(Not Defined Uppercase)
Table/View?	P	(Procedure Table Type)
Default value	?	(Defaults As Null)
Char Type	0	(Not Character Type)
Parameter Type	I	(IN Type)
Parameter Name	lastnm	(2nd Parameter)
Type	CF	(Character Fixed)
Comment	?	
Nullable	Y	
Format	X(20)	
Title	?	
Max Length	20	
Decimal Total Digits	?	(Not A Decimal)
Decimal Fractional Digits	?	(Not A Decimal)
Range Low	?	(No Range)
Range High	?	(No Range)

UpperCase	N	
Table/View?	P	
Default value	?	
Char Type	1	(Latin Character type)
Parameter Type	O	(OUT Type)

HELP PROCEDURE sel_many_into_2 ATTRIBUTES;

Transaction Semantics	TERADATA	(Created In Teradata Mode)
Print Mode	N	(Created Without Print Option)
Platform	UNIX MP-RAS	(Created On Unix MP-RAS Platform)
Character Set	ASCII	
Default Character DataType	LATIN	
Collation	ASCII	
Text	Y	(Source SPL Is Stored)

练习

Lab 28_1

(1) 创建存储过程lab_sp_1，要求两个输入(一个是INTEGER ，另一个是DECIMAL(5,2)) 和一个输出(DECIMAL(5,2))。过程将两个输入加起来，输出结果。

(2) 测试过程。

Lab 28_2

(1) 创建存储过程lab_sp_2，要求一个整数输入和一个DECIMAL(5,2)输出。过程计算输入的平方，输出结果。

(2) 测试过程。

Lab 28_3

(1) 修改lab_sp_1，调用lab_sp_2。过程返回输入输入和的平方。

(2) 测试过程。

Lab 28_4

(1) 创建存储过程lab_sp_3，接收两个整数输入(high和low) 和一个整数输出(empcnt)。过程输入一个范围，找出雇员号在这一范围内(包含)的所有雇员。

如果雇员号在此范围，使用PRINT命令在数据库窗口中输出department_number和last_name。查找到的雇员总数作为输出。

建议：

不使用游标。

使用WHILE循环从小到大处理范围。

使用SELECT INTO - 一次仅处理一行，不需要游标。

使用IF ACTIVITY_COUNT = 1来测试是否有此号码的雇员。

说明一个计数器变量(cnt)，计数有多少雇员在此范围。

当前有效的范围是1001到1025。(建议的范围：low = 1020, high = 1030)

(2) 测试过程。

注：访问数据库窗口去查看PRINT语句的输出。

Lab 28_5

(1) 创建一个存储过程lab_sp_4，完成过程lab_sp_3的同样的功能，但使用游标。

(2) 测试过程。

建议：

说明一个游标。

使用ACTIVITY_COUNT返回总记录数。

使用PRINT语句输出department_number 和last_name。

第二十九章　存储过程(二)

完成本章学习后，将能够：

- 创建包括存储过程的交易；
- 在存储过程中创建例外处理程序，处理例外情况；
- 预测包含交易和例外处理的存储过程的动作；
- 在存储过程中创建可更新的游标，进行修改和删除操作。

29.1 可更新游标

29.1.1 使用可更新游标修改

可更新的游标是存储过程的一个特点，通过前面的SELECT语句定位，在适当的位置修改和删除记录。These are referred to as 'positioned' updates and deletes.

假设要修改雇员的薪水，提高的百分比取决于薪水所处的范围。换句话说，范围规定薪水提高的百分比。

没有游标的话，要定位记录两次，一次查找雇员的薪水(SELECT) ，确定合适的范围，一次基于范围增加薪水(UPDATE)。不论是使用索引，还是全表扫描，每次操作都必须先找到记录。

可更新游标允许使用SELECT语句读记录，保持记录的位置，然后使用'CURRENT OF CURSOR'对当前记录进行修改。使用一个游标指针，按照当前指针的位置，读记录，修改和删除记录。

例

使用游标，修改指定部门内所有雇员的薪水。按照下列规则：

- 如果雇员的收入小于$30,000，则增加 10%。
- 如果雇员的收入在$30,000 和$39,999 之间，则增加 8% 。
- 如果雇员的收入在$40,000 和$49,999 之间，则增加 5%。

```
CREATE PROCEDURE upd_cur1 (IN deptnum INT)
BEGIN
  DECLARE salary DEC(7,2);
  FOR for_loop AS cursor1 CURSOR FOR
    SELECT employee_number, salary_amount FROM employee
    WHERE department_number = :deptnum
  DO
  SET salary = for_loop.salary_amount;
  IF salary < 30000 THEN UPDATE employee
    SET salary_amount = :salary * 1.10 WHERE CURRENT OF cursor1;
  ELSEIF salary < 40000 THEN UPDATE employee
    SET salary_amount = :salary * 1.08 WHERE CURRENT OF cursor1;
  ELSEIF salary < 50000 THEN UPDATE employee
    SET salary_amount = :salary * 1.05 WHERE CURRENT OF cursor1;
  END IF;
  END FOR;
END;
```

上面的例子示范了可更新游标的下列规则：

- 通过 SELECT 语句检索到所有记录。
- 每条记录单独测试。

- 对测试的记录可以直接修改，不需重新定位。
- SELECT 语句申请了读级锁。
- UPDATE 语句申请了行哈稀(row-hash)写级锁。
- 创建者对 employee 表有读和写的权限。

29.1.2 使用可更新游标删除

使用可更新游标，可以进行定位删除。

例

给当前预算低于$500,000的部门，增加10%的预算。给当前预算低于$1,000,000的部门，增加5%的预算。删除预算是0或空值的部门。

```
CREATE PROCEDURE upd_cur2 ()
BEGIN
  FOR for_loop AS cursor2 CURSOR FOR
    SELECT COALESCE(budget_amount,0) AS bud_amt FROM department
  DO
  SET budget = for_loop.bud_amt;
  IF budget = 0 THEN
    DELETE FROM department WHERE CURRENT OF cursor2;
  ELSEIF budget < 500000 THEN UPDATE department
    SET budget_amount = budget_amount*1.10

    WHERE CURRENT OF cursor2;
```

```
  ELSEIF budget < 1000000 THEN UPDATE department
    SET budget_amount = budget_amount*1.05
    WHERE CURRENT OF cursor2;
  END IF;
  END FOR;
END;
```

对上面的例子说明如下：

- COALESCE 函数将空值转换成 0。
- 如果预算超过$1,000,000，不采取任何动作。
- 每个满足条件的记录进行修改或删除操作。

29.1.3 可更新游标的其他规则

可更新游标符合下列规则：

- 能够在 ANSI 模式下创建和执行。
- 允许对当前位置进行多次修改。
- 在修改后，还允许对当前位置执行删除操作。
- 定义触发器的表不能使用游标。

29.2 存储过程的例外处理

29.2.1 例外处理

例外处理是一种译码构造，根据遇到的例外的类型执行一个或多个动作。如果存储过程中没有例外处理，遇到例外情况时，过程在例外点终止。例外处理使用DECLARE HANDLER 语法说明，有两种形式：

- EXIT 处理 - 在执行了例外处理的动作后，过程终止。
- CONTINUE 处理 - 在执行了例外处理的动作后，过程从例外语句的下一条语句继续执行。

处理例外时，可以使用两个保留字：

- SQLSTATE - 返回特定的错误码。
- SQLEXCEPTION - 告诉处理程序处理所有的例外。

例

在过程sel_many_into_2中增加一个EXIT例外处理，处理任何的例外情况。创建一个错误日志表，记录遇到的例外。

首先，创建错误日志表。

```
CREATE TABLE error_log
(Sql_code INT
(Sql_State CHAR(5)
,time_of_day TIME(0));
```

现在，增加例外处理，过程命名为handler_2。

```
REPLACE PROCEDURE handler_2 (IN deptnum INTEGER, OUT lastnm
CHAR(20))
BEGIN
  DECLARE EXIT HANDLER FOR SQLEXCEPTION
  BEGIN
  PRINT 'EXCEPTION CONDITION OCCURRED';
  INSERT INTO error_log VALUES (:SQLCODE, :SQLSTATE,
CURRENT_TIME);
  END;
    SELECT last_name INTO :lastnm FROM employee
    WHERE department_number = :deptnum;

    PRINT 'Successful Completion';
END;
```

上面的例子示范了例外处理的下列规则：

- 使用 DECLARE ... HANDLER 语句定义例外处理的类型和处理条件。
- 使用 BEGIN 和 END 来定义处理的动作。
- 如果处理只执行一条语句，则 BEGIN 和 END 是可选项。
- 当处理动作完成后，EXIT 处理将终止过程。
- SQLEXCEPTION 关键字表示处理所有例外情况。

29.2.2 EXIT例外处理

存储过程handler_2中的例外处理是EXIT。

执行过程：

CALL handler_2(301,lastnm);

lastnm

?

下面说明过程调用的结果：

- SELECT 语句执行发生例外(Failure 7627 returned more than one row).
- 例外处理中的 PRINT 和 INSERT 语句被执行。
- 因为是 EXIT 处理，SELECT 语句后的 PRINT 语句没有执行。
- 输出参数返回空值。

29.2.3 CONTINUE例外处理

考虑同样的存储过程，但使用CONTINUE例外处理：

```
REPLACE PROCEDURE handler_2 (IN deptnum INTEGER, OUT lastnm
CHAR(20))
BEGIN
  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
  BEGIN
  PRINT 'EXCEPTION CONDITION OCCURRED';
  INSERT INTO error_log VALUES (:SQLCODE, :SQLSTATE,
CURRENT_TIME);
  END;
```

```
    SELECT last_name INTO :lastnm FROM employee
    WHERE department_number = :deptnum;

    PRINT 'Successful Completion';
END;
```

下面说明过程调用的结果：

- SELECT 语句执行发生例外(Failure 7627 returned more than one row).
- 例外处理中的 PRINT 和 INSERT 语句被执行。
- 因为是 CONTINUE 处理，SELECT 语句后的 PRINT 语句被执行。
- 输出参数返回空值。

检查错误日志：

```
SELECT * FROM error_log;
```

Sql_Code	Sql_State	time_of_day
7627	21000	17:58:12

注：保留字SQLCODE包含与例外相关的Teradata的错误码(7627)。SQLSTATE是ANSI标准的例外错误码，类型为CHAR(5)。Teradata错误码与SQLSTATE使用的错误码有一个对应，例外处理只识别SQLSTATE，不识别SQLCODE。

例外处理成功完成后，下列3个系统变量都复位：

```
SQLSTATE = '00000'
SQLCODE = 0
ACTIVITY_COUNT = 0
```

29.2.4 多个例外处理

在一个过程中可能有多个例外处理，每个处理不同的例外条件。

例

存储过程修改如下：创建EXIT处理错误21000和错误42000。其他错误，使用CONTINUE处理，给出警告信息。

```
REPLACE PROCEDURE handler_2 (IN deptnum INTEGER, OUT lastnm
CHAR(20))
BEGIN
  DECLARE EXIT HANDLER FOR SQLSTATE '21000', SQLSTATE '42000'
  BEGIN
  PRINT 'EXCEPTION 21000 OCCURRED';
  INSERT INTO error_log VALUES (:SQLCODE, :SQLSTATE,
CURRENT_TIME);
  END;

  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
  BEGIN
  PRINT 'WARNING - EXCEPTION CONDITION OCCURRED';
  INSERT INTO error_log VALUES (:SQLCODE, :SQLSTATE,
CURRENT_TIME);
  END;
   SELECT last_name INTO :lastnm FROM employee
   WHERE department_number = :deptnum;
```

```
   PRINT 'Successful Completion';
END;
```

对上面的例子说明如下：

- EXIT 处理程序处理 SQLSTATE 21000 和 42000 的错误。
- CONTINUE 处理程序处理其他任何例外错误。
- 单个例外处理不能将 SQLSTATE 和 SQLEXCEPTION 混合使用。

执行过程。

```
CALL handler_4(501,lastnm);

lastnm
------
?
```

查看数据库窗口。

```
cnsterm 5

PED1 | 0 1258 | PED1.HANDLER_4 | 5 : 2000-06-27 00:00:39.050000-00:00|
EXCEPTION 21000 OCCURRED
```

检查错误日志。

```
SELECT * FROM error_log;

Sql_Code        Sql_State       time_of_day
-------         ---------       ----------
```

7627 21000 13:56:58

29.3 存储过程和交易

29.3.1 存储过程和ANSI模式的交易

存储过程可以在ANSI-mode或Teradata-mode下创建，必须在同样的模式下执行。

在ANSI模式下，存储过程执行第一条SQL语句，开始一个交易。执行后续的SQL语句，交易继续，直到下列情况发生：

- COMMIT - 结束交易，提交所有对数据库的改变。
- ABORT - 结束交易，回滚交易开始后的所有改变。
- ROLLBACK - 与 ABORT 相同。
- SQL 语句失败 - 与 ABORT 相同。

下列3种情况允许交易继续：

- SQL 语句成功 - 当前 SQL 语句成功完成，交易继续。
- SQL 语句错误 - 当前 SQL 语句中断，交易继续。
- SQL 语句警告 - 当前 SQL 语句完成，但有警告信息，交易继续。

在ANSI模式下，错误或失败条件发生时，SQL语句不能成功完成。

在ANSI模式下，一条SQL语句有4种可能的结果：

成功条件：

- SQL成功完成
- SQLSTATE = '00000'

- ACTIVITY_COUNT = # rows affected
- 交易继续

警告条件：

- SQL完成，显示警告信息
- SQLSTATE > '00000'
- ACTIVITY_COUNT = # rows affected
- 交易继续

错误条件：

- SQL遇到错误条件
- SQLSTATE > '00000'
- 请求被回滚
- 交易继续

失败条件：

- 发生下列一个条件：
    - 检测到死锁
    - DDL语句中断了
    - 遇到一条ROLLBACK或ABORT语句
- SQLSTATE > '00000'
- 交易被回滚并结束

29.3.2 ANSI模式交易的例子

在ANSI模式下创建存储过程Proc_1，包含3条UPDATE语句(UPD1, UPD2, UPD3)，跟着1条COMMIT语句。使用CONTINUE例外处理，允许过程运行完。

```
CREATE PROCEDURE Proc_1 ()
BEGIN
  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
    UPDATE t1 SET col_a = col_a /2; /* UPD1 */
    UPDATE t2 SET col_b = col_b /3; /* UPD2 */
    UPDATE t3 SET col_c = col_c /4; /* UPD3 */
  COMMIT;
END;
```

执行过程：

```
CALL Proc_1();
```

例1 (错误条件 - ANSI-mode)

假设：

- 在 BTEQ ANSI 模式下创建和执行存储过程。
- 在 UPD2 中发生一个错误。

ANSI模式下调用，错误引起语句回滚，但交易继续。

执行过程 (使用CONTINUE例外处理)	
UPD1	- 执行成功
UPD2	- 错误条件
	- 引用已经删除了的表

	- 语句被回滚
UPD3	- 执行成功
COMMIT	- UPD1和UPD3被提交(commit)
过程执行结果	
UPD1	- 提交
UPD2	- 回滚
UPD3	- 提交

例2 - (失败条件)

假设：

- 在 BTEQ ANSI 模式下创建和执行存储过程。
- 在 UPD2 中发生一个失败条件。

执行过程 (使用CONTINUE例外处理)	
UPD1	- 成功执行
UPD2	- 失败条件，遇到死锁
	- 交易被回滚
UPD3	- 开始新的交易
	- 成功执行
COMMIT	- UPD3被提交
过程执行结果	
UPD1	- 回滚
UPD2	- 回滚
UPD3	- 提交

29.3.3 存储过程和Teradata模式交易

在Teradata (BTET) 模式，存储过程提交的每条SQL语句代表一个交易。如果几条SQL语句组成单个交易，必须使用BEGIN TRANSACTION (缩写BT) 和END TRANSACTION (缩写ET)，这叫做显式交易。显式的交易一直继续，直到下列情况发生：

- END TRANSACTION - 结束交易，提交对数据库的改变。
- ABORT - 结束交易，回滚交易开始后的改变。
- ROLLBACK - 与 ABORT 相同
- SQL 语句失败 - 与 ABORT 相同

在Teradata模式，错误和失败条件之间没有区别。

下列两种情况允许交易继续：

- SQL 语句成功 - 当前 SQL 语句成功完成，交易继续。
- SQL 语句警告 - 当前 SQL 语句完成，但有警告信息，交易继续。

在Teradata模式下，1条SQL语句有3种可能的结果：

成功条件：

- SQL成功完成
- SQLSTATE = '00000'
- ACTIVITY_COUNT = # rows affected
- 交易继续

警告条件：

- SQL完成，有警告信息
- SQLSTATE > '00000'
- ACTIVITY_COUNT = # rows affected
- 交易继续

错误或失败条件：

- SQL遇到错误或失败条件
- SQLSTATE > '00000'
- 交易被回滚并结束

29.3.4 Teradata模式交易的例子

在Teradata模式下创建与前面相同的存储过程Proc_1。

```
CREATE PROCEDURE Proc_1 ()
BEGIN
  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
    UPDATE t1 SET col_a = col_a /2; /* UPD1 */
    UPDATE t2 SET col_b = col_b /3; /* UPD2 */
    UPDATE t3 SET col_c = col_c /4; /* UPD3 */
END;
```

执行过程：

```
CALL Proc_1();
```

例1 - (失败条件 - 隐式交易)

执行过程 (使用CONTINUE例外处理)	
UPD1	- 成功执行
UPD2	- 错误/失败条件
	- 引用已经删除的表
	- 语句被回滚
UPD3	- 成功执行
过程执行结果	
UPD2	作为一个交易被回滚 (每条SQL语句都是一个交易)
UPD1	- 提交
UPD3	- 提交

在Teradata模式下创建与前面相同的存储过程Proc_1，过程的语句使用BT和ET括起来。

例2 - (失败条件 - 显式交易)

执行过程 (使用CONTINUE例外处理)	
BT	- 交易开始
UPD1	- 执行成功
UPD2	- 错误/失败条件
	- 遇到死锁

	- 交易被回滚
UPD3	- 执行成功
ET	- 交易被提交
过程执行结果	
UPD1	- 回滚
UPD2	- 回滚
UPD3	- 提交

ANSI模式允许存储过程包括下列SQL语句：

- COMMIT
- ABORT
- ROLLBACK

Teradata模式允许存储过程包括下列SQL语句：

- BEGIN TRANSACTION 或 BT
- END TRANSACTION 或 ET
- ABORT
- ROLLBACK

29.4 例外处理中的例外

29.4.1 例外处理中的例外(ANSI-mode)

在存储过程中出现例外情况，引起过程终止或激活一个例外处理程序。在例外处理程序中也可能发生例外情况。

创建一个存储过程Sp_1，包含几条UPDATE语句和一个CONTINUE例外处理程序。

```
CREATE PROCEDURE Sp_1 ()
BEGIN
 DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
 BEGIN
 UPDATE t3 SET col_c = col_c /4; /* UPD3 */
 END
  UPDATE t4 SET col_d = col_d /2; /* UPD4 */
  UPDATE t5 SET col_e = col_e /3; /* UPD5 */
  UPDATE t6 SET col_f = col_f /4; /* UPD6 */
END;
```

例1 (在过程主体中发生错误条件)

假设：

- 在 BTEQ ANSI 模式下创建和执行存储过程。
- 在 UPD5 中发生一个错误条件。

在ANSI模式下调用过程，错误条件引起语句回滚，但交易继续。

```
CALL Sp_1; /* ANSI mode*/
COMMIT;
```

执行过程	
UPD4	- 执行成功
UPD5	- UPD5发生错误42000
	- 请求被回滚

	- 激活例外处理程序
UPD3	- 执行成功
	- CONTINUE处理回到主过程
UPD6	- 执行成功
COMMIT	- 除UPD5以外，所有改变被提交

在ANSI模式下，交易遇到COMMIT 或ROLLBACK语句才会结束。

例2 (在过程主体和例外处理程序中发生错误条件)

假设：

- 在 BTEQ ANSI 模式下创建和执行存储过程。
- 在 UPD5 中发生一个错误条件。
- 在例外处理程序的 UPD3 也发生一个错误。

```
CALL Sp_1; /* ANSI mode*/
COMMIT;
```

执行过程	
UPD4	- 执行成功
UPD5	- UPD5发生错误42000
	- 请求被回滚
	- 激活例外处理程序
UPD3	- UPD3发生错误53015
	- 过程被终止
UPD6	- 没有执行

COMMIT	- 只提交了UPD4
过程执行结果	
UPD5	- 报告错误代码42000

注：在例外处理程序中发生例外，即使定义了对这种例外的处理，过程也将终止。

在上面的例子中，通用的SQLEXCEPTION例外处理程序是有效的，但不能处理由它自己引起的例外。过程终止后，报告主过程中发生的错误代码，例外处理程序中发生的错误代码不报告。

29.4.2 例外处理中的例外(Teradata-mode)

仍然使用前面的存储过程Sp_1。

例2 (在过程主体和例外处理程序中发生失败条件)

假设：

- 在 BTEQ Teradata 模式下创建和执行存储过程。
- 在 UPD5 中发生一个错误条件。
- 在例外处理程序的 UPD3 也发生一个错误。

在Teradata模式下执行过程，发生失败条件，引起当前交易回滚。在此例中，每个单独的请求都是一个交易。

CALL Sp_1; /* Teradata mode*/

执行过程	
UPD4	- 执行成功并提交
UPD5	- UPD5发生错误42000
	- 激活例外处理程序
UPD3	- UPD3发生错误53015
	- 过程被终止
UPD6	- 没有执行
过程执行结果	
UPD5	- 报告错误代码42000

例2 (同样的失败 - 显式交易)

假设同样的条件，但是过程在显式交易中执行。在此例中，所有请求都回滚。

```
BT;
CALL Sp_1;
ET;
```

执行过程	
UPD4	- 执行成功
UPD5	- UPD5发生错误42000
	- 激活例外处理程序
UPD3	- UPD3发生错误53015
	- 过程被终止

UPD6	- 没有执行
过程执行结果	
UPD4	- 与交易一起被回滚
UPD5	- 报告错误代码42000

29.5 包含存储过程的交易

29.5.1 包含存储过程的交易(ANSI-mode)

除了在存储过程内的交易外，存储过程本身也可以是交易的一部分。并且，存储过程还可以有例外处理程序。

这样，在交易的下列地方都可能发生失败：

- 存储过程外
- 存储过程内的主体部分
- 存储过程内的例外处理程序

仍然使用前面的存储过程Sp_1。

例1 (错误条件 - ANSI-mode)

假设：

- 在 BTEQ ANSI 模式下创建和执行存储过程。
- 在 UPD5 中发生一个错误条件。
- UPD1 和 UPD2 在存储过程外执行。

在ANSI模式下调用过程，发生错误条件，引起语句回滚，但交易继续。

```
UPDATE t1 SET col_a = col_a /2; /* UPD1 */
          /* not part of stored procedure */
UPDATE t2 SET col_b = col_b /3; /* UPD2 */
          /* not part of stored procedure */
CALL Sp_1;
COMMIT;
```

执行过程	
UPD1	- 执行成功
UPD2	- 执行成功
UPD4	- 执行成功
UPD5	- 发生错误条件
	- UPD5被回滚
	- 激活例外处理程序
UPD3	- 执行贯彻
	- CONTINUE处理返回到主过程
UPD6	- 执行成功
COMMIT	- UPD1, UPD2, UPD4, UPD3, UPD6被提交

例2 (失败条件 - ANSI-mode)

假设：

- 在 BTEQ ANSI 模式下创建和执行存储过程。
- 在 UPD5 中发生一个失败条件。

```
UPDATE t1 SET col_a = col_a /2; /* UPD1 */
     /* not part of stored procedure */
```

```
UPDATE t2 SET col_b = col_b /3; /* UPD2 */
    /* not part of stored procedure */
CALL Sp_1;
COMMIT;
```

执行过程	
UPD1	- 执行成功
UPD2	- 执行成功
UPD4	- 执行成功
UPD5	- 发生失败条件
	- UPD5, UPD4, UPD2, UPD1被回滚
	- 激活例外处理程序
UPD3	- 执行成功
	- CONTINUE处理返回主过程
UPD6	- 执行成功
COMMIT	- UPD3, UPD6被提交

29.5.2 包含存储过程的交易(Teradata-mode)

在Teradata模式下，每个请求都是一个独立的交易，失败条件只影响一个交易。在Teradata模式下，错误或失败条件是一样的，处理相同。

仍然使用存储过程Sp_1。

例1 (Teradata 失败条件)

假设：

- 在 BTEQ Teradata 模式下创建和执行存储过程。
- 在 UPD5 中发生一个失败条件。

```
UPDATE t1 SET col_a = col_a /2; /* UPD1 */
     /* not part of stored procedure */
UPDATE t2 SET col_b = col_b /3; /* UPD2 */
     /* not part of stored procedure */
CALL Sp_1;
```

执行过程	
UPD1	- 执行成功并提交
UPD2	- 执行成功并提交
UPD4	- 执行成功并提交
UPD5	- 发生失败条件
	- UPD5被回滚
	- 激活例外处理程序
UPD3	- 执行成功并提交
	- CONTINUE处理返回主过程
UPD6	- 执行成功并提交

例2 (Teradata失败条件，显式交易)

假设：

- 在 BTEQ Teradata 模式下创建和执行存储过程。
- 在 UPD5 中发生一个失败条件。
- 使用 BT 和 ET 定义交易。

```
BT;
UPDATE t1 SET col_a = col_a /2; /* UPD1 */
    /* not part of stored procedure */
UPDATE t2 SET col_b = col_b /3; /* UPD2 */
    /* not part of stored procedure */
CALL Sp_1;
ET;
```

执行过程	
BT	- 开始显式交易
UPD1	- 执行成功
UPD2	- 执行成功
UPD4	- 执行成功
UPD5	- 发生失败条件
	- UPD5, UPD4, UPD2, UPD1被回滚
	- 激活例外处理程序
	- 显式交易结束
UPD3	- 执行成功并提交 (新交易)
	- CONTINUE处理返回主过程
UPD6	- 执行成功并提交 (新交易)
ET	- 忽略

在此例中，UPD5的失败回滚整个交易。这个交易结束后，后面的两个请求被作为单独的交易处理。遇到ET语句时，忽略。

练习

Lab 29_1

(1)在创建过程之前，需要创建雇员的视图：

```
DATABASE sqlxxxx;
CREATE VIEW emp_view AS SELECT * FROM Customer_Service.employee;
```

并且，也要创建error_log表：

```
CREATE TABLE error_log
(Sql_code INT
,Sql_State CHAR(5)
,time_of_day TIME(0));
```

(2) 使用存储过程检索雇员信息，根据雇员号得到雇员的last_name，类型为CHAR(20)。

建议：

- 访问视图 emp_view。
- 定义 EXIT 例外处理程序处理任何 SQL 例外情况。
- 当 SQLSTATE 是'42000' 时，输出'Object Not Found'；其他输出'Unplanned Exception'。
- 定义 CONTINUE 例外处理程序处理 SQLSTATE 是'02000'的情况，输出'No Rows Found'。
- 出现任何例外情况，往错误日志表 error_log 中插入 1 条记录。

- 在 SELECT 语句后，使用 PRINT 语句输出'Successful Completion'和 ACTIVITY_COUNT。
- 在 BTEQ 内查看数据库窗口。

(3) 使用雇员号1008测试存储过程。

(4) 使用不存在的雇员1099测试存储过程。

(5) 删除视图emp_view，再使用雇员号1008运行过程。

Lab 29_2

(1) 首先，创建一个雇员表emp_lab，并复制数据。该表只有employee_number和salary_amount列。

CREATE TABLE emp_lab AS (SELECT employee_number, salary_amount FROM Customer_Service.employee) WITH DATA;

(2) 创建salary_log表和错误日志error_log表。

```
CREATE TABLE salary_log
(employee_number INT
,oldsal    DEC(9,2)
,newsal   DEC(9,2))
```

```
PRIMARY INDEX (employee_number);

CREATE TABLE error_log
(Sql_code INT ,Sql_State CHAR(5)
,time_of_day TIME(0));
```

(3) 创建一个存储过程，以雇员号和百分比为输入，按照增加的百分比更新雇员的薪水。

建议：

- 过程输出原来的薪水和新的薪水。
- 如果更新成功，
  - 在salary_log表中增加该雇员的记录；
  - 打印ACTIVITY_COUNT。
- 设置CONTINUE例外处理程序，
  - 对任何例外，往错误日志error_log表中插入1条记录；
  - 打印SQLSTATE。
- 在过程结束处，使用PRINT输出信息'Procedure Completion'。

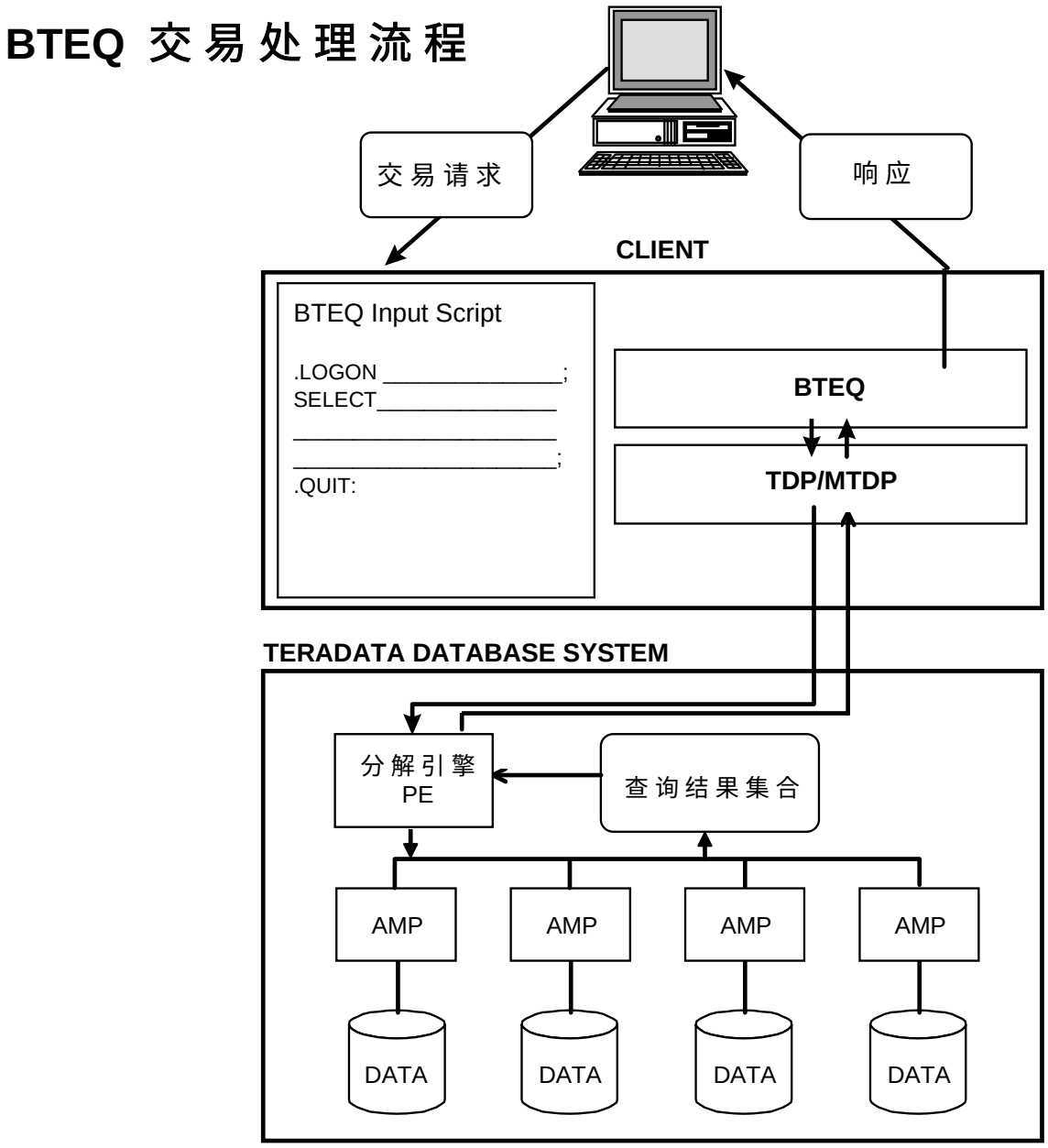
(4) 测试过程，给雇员1010的薪水提高10%。

(5) 测试过程，给不存在的雇员(1000)提高薪水。

附录一　BTEQ基础

1.1 BTEQ基础

BTEQ代表Basic Teradata Query，是随Teradata版本一(Version 1)发行时就有的一个用于提交SQL查询的前端工具。功能很强，速度也很快，因此在后来每次进行移植时都将BTEQ包含进来。现在，BTEQ可以既可以在封闭主机环境下运行，也可以在UNIX、Windows或者DOS环境下运行。BTEQ的交易处理流程可以用图A1-1来表示。



图A1-1 BTEQ的交易处理流程

在UNIX环境下，BTEQ是随Teradata基本系统一起发行的一个软件包，不需要另外购买。在UNXI提示符下键入下面的命令：

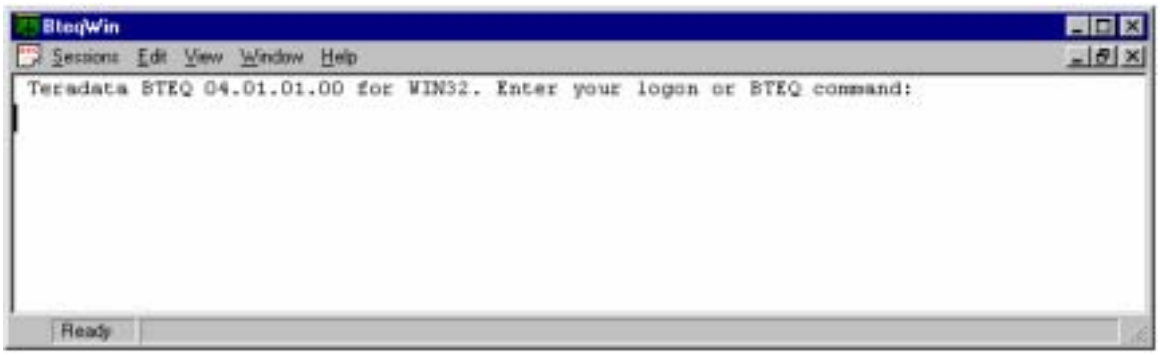
bteq

系统出现类似下面的提示：

Teradata BTEQ 04.01.00.00 for UNIX5. Enter your logon or BTEQ command:

表示已进入BTEQ环境，等待用户登录。

如果是在Windows或DOS环境下，BTEQ是单独的一个软件包，安装时一般都放置在Teradata程序组下。双击"BTEQ for Windows"按钮，出现图A1-2所示的提示：



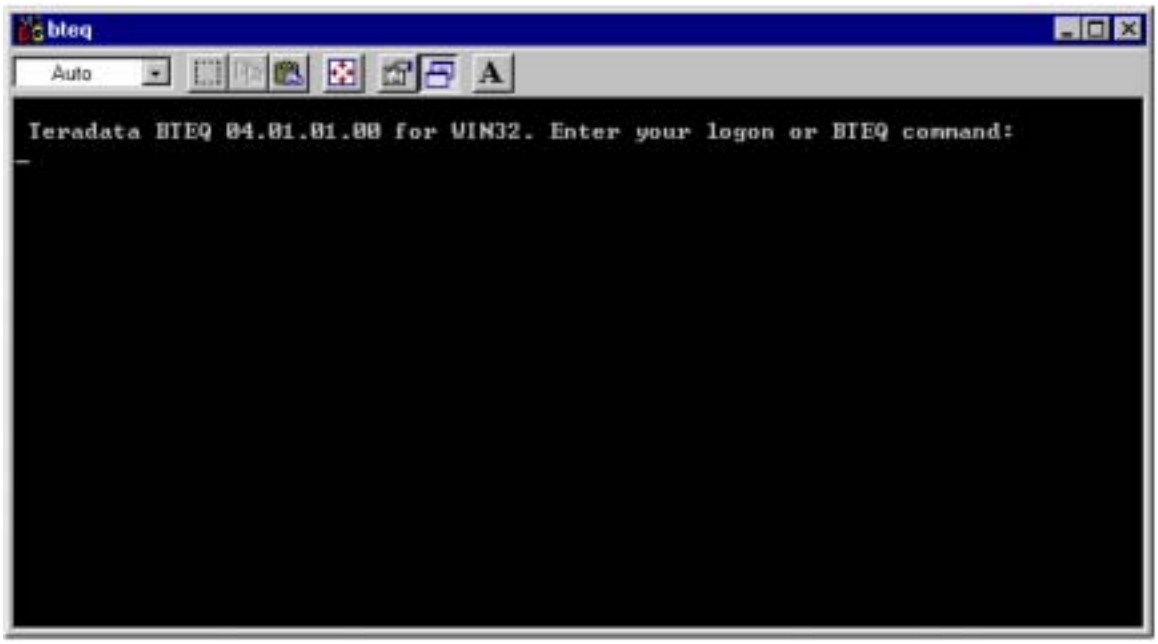
图A1-2 BTEQ for Windows界面

注意此时登录时要指定系统名字，系统名字与用户名之间用"/"隔开。假设这个Teradata数据库的名字是DEMO (该名字必须在HOSTS文件中加以定义)，以SQL01用户名登录，可以键入下面的命令：

.logon demo/sql01

系统将弹出一个窗口，提示用户输入密码。密码校验正确后，即可以递交SQL交易请求了。

如果点击Teradata程序组下的BTEQ按钮，则进入了BTEQ的DOS环境，如图A1-3所示：



图A1-3 BTEQ for DOS界面

和BTEQ for Windows一样，登录时也要指定系统名字。

BTEQ既可以以交互式方式运行，也可以以批处理方式运行。在批处理方式下，必须先编写一个脚本(SCRIPT)文件，交互式方式下的所有命令均可以写到脚本中。批处理方式运行一个脚本的方法是：

bteq <input_script >output_file

或者先进入BTEQ，然后执行：

.run file = input_script

下面简单介绍BTEQ在交互式方式下的常用命令和使用方法，这些命令都可以写到脚本文件中，以批处理的方式运行。

所有的BTEQ命令都必须用句点"."开头，结尾可以用分号";"结束，也可以不用。所有的SQL语句不需要用句点开头，但一定要用分号结束。可以把多个SQL语句组合成一个交易，分号用来表示一个交易的结束。

用户登录的命令是

.logon systemname/username

如果是在运行Teradata的UNIX机器的终端上执行BTEQ，可以不需要提供系统名。

终止一个Teradata连接的命令是

.logoff

这只是退出了Teradata，没有退出BTEQ。

退出BTEQ的命令是

.quit

QUIT命令将先终止Teradata连接(如果还没有的话)，并返回到操作系统提示符。

如果要显示所有的BTEQ命令，可执行：

.help bteq

如果在在BTEQ中执行UNIX命令，可执行：

.os unix_command

如果要将BTEQ的输出保存到一个文件中，可使用下面的EXPORT命令：

.export file = output_file

然后可使用下面的命令将BTEQ的输出恢复到标准输出：

.export reset

在编写BTEQ的脚本时，可以插入注释。BTEQ中的注释和C语言中的注释一样，以"/*"开头，以" */"结尾，可以跨行，如下所示：

```
/* You can include comments in a BTEQ
   script by enclosing the comment text in "/*" and "*/"
   and the comment may span multiple lines
*/
.logon sql01, sql01
select * from customer_service.department
.quit
```

如果按照ANSI的标准，注释是以"--"开头，到行尾自动结束，不可以跨行。

注意在脚本文件中，必须把用户的密码在LOGON命令中提供。假设编写如下内容的脚本文件：

```
.SET SESSION TRANSACTION ANSI
.LOGON sql01, sql01;

--  Obtain a list of the department numbers and names from the
--  department table.

SELECT  department_number
              ,department_name
FROM    department;

.QUIT
```

该文件名为testscript，执行该脚本及返回结果的过程如下所示：

```
# bteq
 Teradata BTEQ 04.00.01.00 for UNIX5. Enter your logon or BTEQ command:
.run file  = testscript       -注： 键入此命令后，下面的内容均为系统输出
.run file = testscript
 Teradata BTEQ 04.00.01.00 for UNIX5. Enter your logon or BTEQ command:
.SET SESSION TRANSACTION ANSI

 Teradata BTEQ 04.00.01.00 for UNIX5. Enter your logon or BTEQ command:
.LOGON  sql01,    -注：尽管在脚本文件中提供了用户密码，但这里不会显示出来

 *** Logon successfully completed.
 *** Transaction Semantics are ANSI.
 *** Character Set Name is 'ASCII'.
 *** Total elapsed time was 1 second.

 BTEQ -- Enter your DBC/SQL request or BTEQ command:
-- Obtain a list of the department numbers and names from the
-- department table.

SELECT  department_number
 ,department_name
FROM department;

 *** Query completed. 9 rows found. 2 columns returned.
 *** Total elapsed time was 1 second.
```

```
department_number       department_name
-----------------       ------------------------------
              401       customer support


BTEQ -- Enter your DBC/SQL request or BTEQ command:

.QUIT
 *** You are now logged off from the DBC.
 *** Exiting BTEQ...
 *** RC (return code) = 0
# exit
```

1.2 BTEQ的报表输出功能

BTEQ还提供了较丰富的用于编写报表的命令。前面谈到，要把BTEQ的输出转换到一个文件中，要使用EXPORT命令。事实上，EXPORT命令的选项很多，它的基本形式是：

```
.EXPORT { DATA                { FILE = filename                    } }
          { INDICDATA         { DDNAME = ddname                    } }
          { REPORT              (此 项 仅 用 于 封 闭 主 机)        } }
          { DIF [datalables]                                       } }
    [,LIMIT = n] [,CLOSE | OPEN]
```

其中几个选项分别说明如下：

- DATA：输出为二进制数据文件，具体数据与每个字段的类型定义有关。每行为一条记录，输出中不包含格式信息。对于变长字段，如

VARCHAR、VARBYTE 等，输出时该字段前有一个二字节长的二进制数，用来表示实际的数据长度。

- INDICDATA：与 DATA 基本相同，但当字段中有 NULL 值时，将有定义好的指示器(INDICTATOR)来表示。
- REPORT：输出为报表方式，可以定义字段头等，这也是缺省的输出方式。每行为一条记录，当字段值为 NULL 时，用问号"?"来表示。每个字段的输出可以进行格式化。
- DIF：用于 PC 上应用的数据交换格式(Data Interchange Format)，如 LOTUS 1-2-3 等。

BTEQ具有较强的报表格式化输出功能，下面简单归纳一下常用的格式化命令，各命令中带下划线的表示缺省值。

.[SET] DEFAULTS

将输出格式定义复位成刚进入BTEQ时的值。

.[SET] ECHOREQ [ON/OFF]

定义是否将SQL请求以及BTEQ命令的内容复制到输出报表中，缺省是ON。

.[SET] FOLDLINE [ON/OFF] [,/n/ALL]

输出时从指定的字段开始换行。假设输出包含四个字段，则

.SET FOLDLINE ON 1

将使输出的第一个字段换行，而第二、三、四字段并列在下面的一行。进一步发出下面的指令：

.SET FOLDLINE ON 3

则将使输出的第一个字段在第一行(上面的命令仍有效)，第二、三字段在第二行(因为从第三个字段换行)，第四个字段在第三行。

.[SET] FOOTING [NULL, 'string']

定义每页下面的脚注，字符串中可以包含控制字符&DATE、&TIME、&PAGE和&n等来表示当前的日期、时间、页号以及SELECT语句中的第n个字段。该命令要求FORMAT置成ON。

.[SET] FORMAT [ON/OFF]

用来控制那些操作页面格式的命令，当它为OFF时，BTEQ将忽略FOOTING、FORMCHAR、HEADINAG、PAGEBREAK、PAGELENGTH、RTITLE的设置。

.[SET] FORMCHAR [ON/OFF/'HEX sequence 'xb]

用来控制报表的打印。在有些打印机中必须使用一个特殊的十进制数来定义换行，可以通过此命令来实现这一功能。如.SET FORMCHAR '0C' xb表示将0C设置成换行符。

.[SET] HEADING [NULL, 'string']

定义每页输出的头，也支持&DATE、&TIME、&PAGE和&n等。

.[SET] NULL [AS] ['string']

对于NULL值，系统缺省用问号来表示，这个命令可用来改变此缺省值。

.[SET] OMIT [ON/OFF] [,/n/ALL]

在SELECT出来的所有字段中，将指定的某些字段不包含进报表中。这些字段的值可能用到页头或脚注中。

.[SET] PAGEBREAK [ON/OFF] [,/n/ALL]

当指定的某些字段的值发生变化时，插入一个分页符，开始新的一页。

.[SET] PAGELENGTH n

定义页面的长度，缺省为55行。

.[SET] RTITLE ['string']

定义每页上方的标题，它将自动包含日期和页号。

.[SET] SEPARATOR ['string' / n]

定义输出时字段之间的分隔符，如可以设成" |"等符号。如果是数字n，表示字段之间的空格数，缺省是2，最多可设成254个空格。

.[SET] SESSION CHARSET [code / 'charstring']

定义当前连接的字符集。字符集可以代码来表示，也可以用实际的字符串来表示，如'German ASCII'。缺省值是'ASCII'。

.[SET] SIDETITLES [ON/OFF] [0, withlist, ALL]

在输出时每个字段前加上字段名作为副标题。withlist表示WITH子句的列表，如数字1表示第一个WITH子句，数字2表示第二个WITH子句。

.[SET] SKIPDOUBLE [ON/OFF] [, /n/ALL]

当指定字段的值发生变化时，插入两个空行。

.[SET] SKIPLINE [ON/OFF] [, /n/ALL]

当指定字段的值发生变化时，插入一个空行。

.[SET] SUPPRESS [ON/OFF] [, /n/ALL]

对于指定的字段，在列表时如果有连续的重复值，则用空格代替。

.[SET] TITLEDASHES [ON/OFF] [0, withlist, ALL]

在WITH子句中，是否加点划线进行分隔。withlist表示WITH子句的列表。

.[SET] UNDERLINE [ON/OFF] [, /n/ALL]

在每行输出中对指定的字段加下划线。

.[SET] WIDTH n

设置报表宽度。缺省为75个字符，设置范围为20至254个字符。

下面综合使用刚刚列举的格式化命令，写了一个脚本文件。列表如下：

```
.LOGON  userid,password;
.SET       FORMAT ON;
.SET       PAGEBREAK ON 1;
.SET       RTITLE    ALARIES BY JOB CODE// CODE &1//JOB TITLE &2';
.SET       OMIT ON 1,2;
.SET       SUPPRESS ON 3,4;
.SET       WIDTH 72;
SELECT job.job_code
```

```
   ,description
   ,department_number   (TITLE     EPT ')
            (FORMAT '怜zz9')
   ,manager_employee_number (TITLE 惭GR  ')
            (FORMAT '怜zz9')
   ,employee_number     (TITLE     MP')
            (FORMAT '怜zz9')
   ,last_name       (TITLE     AST  ')
            (FORMAT     (9)')
   ,first_name      (TITLE     IRST  ')
            (FORMAT     (9)')
   ,salary_amount          (TITLE     ALARY ')
            (FORMAT  '$,$$$,$$9.99')
FROM    job
   ,employee
WHERE  job.job_code = employee.job_code
WITH     SUM (salary_amount)  (TITLE 惯otal Salaries')
   (FORMAT  '$,$$$,$$99.99')
   BY job.job_code
WITH     SUM (salary_amount)(TITLE     rand Total')
   (FORMAT  '$$,$$$,$$9.99')
ORDER BY       3, 4, 5;
.QUIT;
```

该脚本文件产生的输出如下所示：

Sample BTEQ Report

90/11/11 SALARIES BY JOB CODE PAGE 1

CODE 432101

JOB TITLE Instructor

DEPT	MGR	EMP	LAST	FIRST	SALARY
403	1005	1007	Villegas	Arnando	$49,700.00
		1009	Lombardo	Domingus	$31,000.00
		1012	Hopkins	Paulene	$37,900.00
		1020	Charles	John	$39,500.00
		1024	Brown	Allen	$43,700.00
				Total Salaries	$201,800.00

90/11/11 SALARIES BY JOB CODE PAGE 2

CODE 512101

JOB TITLE Sales Rep

DEPT	MGR	EMP	LAST	FIRST	SALARY
501	1017	1015	Wilson	Edward	$53,625.00
		1018	Ratzlaff	Larry	$54,000.00
		1023	Rabbit	Peter	$26,500.00
				Total Salaries	$134,125.00
				Grand Total	$1,067,350.00

附录二　创建测试数据库试验环境的脚本文件

在UNIX或NT下分别创建下面各个脚本文件：

1、INTED00.TXT

```
.LOGON DBC,DBC
/*************************************************/
/* Create a "super user" for Teradata Education  */
/* Classroom Lab database/userid installation    */
/*************************************************/
/* Note: This script must be run by a user who   */
/*       has Create Database/User authority and  */
/*       the right to grant or re-grant select   */
/*       on the dictionary tables.
/*************************************************/
/* Space limits and account codes may be         */
/* modified by the user              (RPK 3/97) */
/*************************************************/

CREATE USER Teradata_Education AS PASSWORD = educate
      PERM = 20000000 SPOOL = 5000000
      ACCOUNT = ('$M_D2102');

GRANT ALL ON Teradata_Education TO Teradata_Education
            WITH GRANT OPTION;
```

```
GRANT SELECT ON DBC TO Teradata_Education
        WITH GRANT OPTION;

.LOGOFF
```

2、INSQL01.TXT

```
.SET SESSIONS 8
.SET QUIET ON
.LOGON Teradata_Education,Educate
/*********************************************************/
/* Creates Database Customer_Service, defines and       */
/* populates nine (9) sample tables          (RPK 3/97) */
/*********************************************************/

SELECT * FROM DBC.Databases
WHERE    DatabaseName = 'Customer_Service';

.IF ActivityCount = 0 THEN .GOTO CreateCS

GRANT DROP DATABASE ON Customer_Service TO Teradata_Education;
DELETE DATABASE Customer_Service;
DROP   DATABASE Customer_Service;

.LABEL CreateCS

CREATE DATABASE Customer_Service FROM Teradata_Education AS
     PERM=500000 ACCOUNT = ('$M_P0623');
```

```
DATABASE Customer_Service;

CREATE TABLE contact, FALLBACK
    (contact_number INTEGER
    ,contact_name CHAR(30) NOT NULL
    ,area_code SMALLINT NOT NULL
    ,phone INTEGER NOT NULL
    ,extension INTEGER
    ,last_call_date DATE NOT NULL)
     UNIQUE PRIMARY INDEX (contact_number);

CREATE TABLE customer, FALLBACK
    (customer_number INTEGER
    ,customer_name CHAR(30) NOT NULL
    ,parent_customer_number INTEGER
    ,sales_employee_number INTEGER
    )
     UNIQUE PRIMARY INDEX (customer_number);

CREATE TABLE department, FALLBACK
    (department_number SMALLINT
    ,department_name CHAR(30) NOT NULL
    ,budget_amount DECIMAL(10,2)
    ,manager_employee_number INTEGER
    )
     UNIQUE PRIMARY INDEX (department_number)
    ,UNIQUE        INDEX (department_name);

CREATE TABLE employee, FALLBACK
   (employee_number INTEGER
```

```
    ,manager_employee_number INTEGER
    ,department_number INTEGER
    ,job_code INTEGER
    ,last_name CHAR(20) NOT NULL
    ,first_name VARCHAR(30) NOT NULL
    ,hire_date DATE NOT NULL
    ,birthdate DATE NOT NULL
    ,salary_amount DECIMAL(10,2) NOT NULL
    )
     UNIQUE PRIMARY INDEX (employee_number);

CREATE TABLE employee_phone, FALLBACK
    (employee_number INTEGER NOT NULL
    ,area_code SMALLINT NOT NULL
    ,phone INTEGER NOT NULL
    ,extension INTEGER
    ,comment_line CHAR(72)
    )
     PRIMARY INDEX (employee_number);

CREATE TABLE job, FALLBACK
    (job_code INTEGER
    ,description VARCHAR(40) NOT NULL
    ,hourly_billing_rate DECIMAL(6,2)
    ,hourly_cost_rate DECIMAL(6,2)
    )
     UNIQUE PRIMARY INDEX (job_code)
    ,UNIQUE        INDEX (description);

CREATE TABLE location, FALLBACK
    (location_number INTEGER
```

```
   ,customer_number INTEGER NOT NULL
   ,first_address_line CHAR(30) NOT NULL
   ,city VARCHAR(30) NOT NULL
   ,state CHAR(15) NOT NULL
   ,zip_code INTEGER NOT NULL
   ,second_address_line CHAR(30)
   ,third_address_line CHAR(30)
   )
    PRIMARY INDEX (customer_number);


CREATE TABLE location_employee, FALLBACK
   (location_number INTEGER NOT NULL
   ,employee_number INTEGER NOT NULL
   )
    PRIMARY INDEX (employee_number);

CREATE TABLE location_phone, FALLBACK
   (location_number INTEGER
   ,area_code SMALLINT NOT NULL
   ,phone INTEGER NOT NULL
   ,extension INTEGER
   ,description VARCHAR(40) NOT NULL
   ,comment_line LONG VARCHAR
   )
    PRIMARY INDEX (location_number);


INSERT INTO contact     VALUES
  (8010,'Brayman, Connie',408,1112345,112,870721);
INSERT INTO contact     VALUES
```

```
  (8001,'Leblanc, James',805,2213456,221,870801);
INSERT INTO contact     VALUES
  (8005,'Hughes, Jack',212,5432126,710,870805);
INSERT INTO contact     VALUES
  (8007,'Smith, Ginny',408,3792152,333,870801);
INSERT INTO contact     VALUES
  (8008,'Torres, Alison',802,5487890,444,880814);
INSERT INTO contact     VALUES
  (8009,'Dibble, Nancy',602,2713387,652,880809);

INSERT INTO customer VALUES
  (00,'Corporate Headquarters',NULL,NULL);
INSERT INTO customer VALUES
  (01,'A to Z Communications, Inc.',NULL,1015);
INSERT INTO customer VALUES
  (02,'Simple Instruments Co.',1,1015);
INSERT INTO customer VALUES
  (03,'First American Bank',NULL,1023);
INSERT INTO customer VALUES
  (04,'Sum Bank',3,1023);
INSERT INTO customer VALUES
  (05,'Federal Bureau of Rules',NULL,1018);
INSERT INTO customer VALUES
  (06,'Liberty Tours',NULL,1023);
INSERT INTO customer VALUES
  (07,'Cream of the Crop',NULL,1018);
INSERT INTO customer VALUES
  (08,'Colby Co.',NULL,1018);
INSERT INTO customer VALUES
  (09,'More Data Enterprise',NULL,1023);
INSERT INTO customer VALUES
```

```
  (10,'Graduates Job Service',NULL,1015);
INSERT INTO customer VALUES
  (11,'Hotel California',NULL,1015);
INSERT INTO customer VALUES
  (12,'Cheap Rentals',NULL,1018);
INSERT INTO customer VALUES
  (13,'First American Bank',3,1023);
INSERT INTO customer VALUES
  (14,'Metro Savings',NULL,1018);
INSERT INTO customer VALUES
  (15,'Cates Modeling',NULL,1015);
INSERT INTO customer VALUES
  (16,'VIP Investments',3,1023);
INSERT INTO customer VALUES
  (17,'East Coast Dating Service',NULL,1023);
INSERT INTO customer VALUES
  (18,'Wall Street Connection',NULL,1023);
INSERT INTO customer VALUES
  (19,'More Data Enterprise',9,1015);
INSERT INTO customer VALUES
  (20,'Metro Savings',14,1018);

INSERT INTO department  VALUES
  (401,'customer support',982300,1003);
INSERT INTO department  VALUES
  (201,'technical operations',293800,1025);
INSERT INTO department  VALUES
  (301,'research and development',465600,1019);
INSERT INTO department  VALUES
  (302,'product planning',226000,1016);
INSERT INTO department  VALUES
```

```
  (403,'education',932000,1005);
INSERT INTO department  VALUES
  (402,'software support',308000,1011);
INSERT INTO department  VALUES
  (501,'marketing sales',308000,1017);
INSERT INTO department  VALUES
  (100,'president',400000,0801);
INSERT INTO department  VALUES
  (600,'None',,1099);

INSERT INTO employee VALUES
  (0801,0801,100,111100,'Trainer','I.B.',730301,450811,100000);
INSERT INTO employee VALUES
  (1001,1003,401,412101,'Hoover','William',760618,500114,25525);
INSERT INTO employee VALUES
  (1002,1003,401,413201,'Brown','Alan',760731,440809,43100);
INSERT INTO employee VALUES
  (1003,0801,401,411100,'Trader','James',760731,470619,37850);
INSERT INTO employee VALUES
  (1004,1003,401,412101,'Johnson','Darlene',761015,460423,36300);
INSERT INTO employee VALUES
  (1005,0801,403,431100,'Ryan','Loretta',761015,550910,31200);
INSERT INTO employee VALUES
  (1006,1019,301,312101,'Stein','John',761015,531015,29450);
INSERT INTO employee VALUES
  (1007,1005,403,432101,'Villegas','Arnando',770102,370131,49700);
INSERT INTO employee VALUES
  (1008,1019,301,312102,'Kanieski','Carol',770201,580517,29250);
INSERT INTO employee VALUES
  (1009,1005,403,432101,'Lombardo','Domingus',770201,451115,31000);
INSERT INTO employee VALUES
```

```
  (1010,1003,401,412101,'Rogers','Frank',770301,350423,46000);
INSERT INTO employee VALUES
  (1011,0801,402,421100,'Daly','James',770315,491211,52500);
INSERT INTO employee VALUES
  (1012,1005,403,432101,'Hopkins','Paulene',770315,420218,37900);
INSERT INTO employee VALUES
  (1013,1003,401,412102,'Phillips','Charles',770401,630810,24500);
INSERT INTO employee VALUES
  (1014,1011,402,422101,'Crane','Robert',780115,600704,24500);
INSERT INTO employee VALUES
  (1015,1017,501,512101,'Wilson','Edward',780301,570304,53625);
INSERT INTO employee VALUES
  (1016,0801,302,321100,'Rogers','Nora',780301,590904,56500);
INSERT INTO employee VALUES
  (1017,0801,501,511100,'Runyon','Irene',780501,511110,66000);
INSERT INTO employee VALUES
  (1018,1017,501,512101,'Ratzlaff','Larry',780715,540531,54000);
INSERT INTO employee VALUES
  (1019,0801,301,311100,'Kubic','Ron',780801,421211,57700);
INSERT INTO employee VALUES
  (1020,1005,403,432101,'Charles','John',781001,490621,39500);
INSERT INTO employee VALUES
  (1021,1025,201,222101,'Morrissey','Jim',781001,430429,38750);
INSERT INTO employee VALUES
  (1022,1003,401,412102,'Machado','Albert',790301,570714,32300);
INSERT INTO employee VALUES
  (1023,1017,501,512101,'Rabbit','Peter',790301,621029,26500);
INSERT INTO employee VALUES
  (1024,1005,403,432101,'Brown','Allen',790501,540116,43700);
INSERT INTO employee VALUES
  (1025,0801,201,211100,'Short','Michael',790501,470707,34700);
```

```
INSERT INTO employee_phone VALUES
  (0801,213,8278777,101,'Corporate President');
INSERT INTO employee_phone VALUES
  (1001,415,2412021,NULL,'Graduates Job Service');
INSERT INTO employee_phone VALUES
  (1001,415,3563560,NULL,'Hotel California');
INSERT INTO employee_phone VALUES
  (1001,213,2872019,NULL,'Cates Modeling');
INSERT INTO employee_phone VALUES
  (1001,415,6567000,NULL,'More Data');
INSERT INTO employee_phone VALUES
  (1001,415,4491221,NULL,'A TO Z office');
INSERT INTO employee_phone VALUES
  (1001,415,4491225,NULL,'A TO Z System Manager');
INSERT INTO employee_phone VALUES
  (1001,415,4491244,NULL,'A TO Z Secretary');
INSERT INTO employee_phone VALUES
  (1001,415,9234864,NULL,'residence/office');
INSERT INTO employee_phone VALUES
  (1001,415,9237892,NULL,'Simple Instruments');
INSERT INTO employee_phone VALUES
  (1002,213,2721606,NULL,'residence');
INSERT INTO employee_phone VALUES
  (1002,213,8278777,439,'office');
INSERT INTO employee_phone VALUES
  (1003,213,3774534,NULL,'residence');
INSERT INTO employee_phone VALUES
  (1003,213,8278777,401,'office');
INSERT INTO employee_phone VALUES
  (1004,212,7230101,NULL,'First Am. Bank computer room');
```

```
INSERT INTO employee_phone VALUES
  (1004,212,7232121,NULL,'First Am. Bank system manager');
INSERT INTO employee_phone VALUES
  (1004,609,5591011,213,'Sum Bank system manager');
INSERT INTO employee_phone VALUES
  (1004,609,5591011,224,'Sum Bank computer room');
INSERT INTO employee_phone VALUES
  (1004,609,5591011,225,'Sum Bank secretary');
INSERT INTO employee_phone VALUES
  (1004,609,5785781,NULL,'residence/office');
INSERT INTO employee_phone VALUES
  (1004,212,5786099,NULL,'Liberty Tours main number');
INSERT INTO employee_phone VALUES
  (1004,919,9789000,NULL,'More Data System Manager');
INSERT INTO employee_phone VALUES
  (1004,617,7567676,NULL,'First Am. Bank Manager');
INSERT INTO employee_phone VALUES
  (1004,212,8282828,NULL,'VIP Investments');
INSERT INTO employee_phone VALUES
  (1004,718,2243283,NULL,'East Coast Dating');
INSERT INTO employee_phone VALUES
  (1004,212,4909190,NULL,'Wall Street Connection');
INSERT INTO employee_phone VALUES
  (1005,213,2514189,NULL,'residence');
INSERT INTO employee_phone VALUES
  (1005,213,8278777,415,'office');
INSERT INTO employee_phone VALUES
  (1006,213,8278777,410,'office');
INSERT INTO employee_phone VALUES
  (1006,213,3716087,NULL,'residence');
INSERT INTO employee_phone VALUES
```

```
  (1007,213,2274764,NULL,'residence');
INSERT INTO employee_phone VALUES
  (1007,213,8278777,440,'office');
INSERT INTO employee_phone VALUES
  (1008,213,3788092,NULL,'residence');
INSERT INTO employee_phone VALUES
  (1008,213,8278777,429,'office');
INSERT INTO employee_phone VALUES
  (1009,213,2482619,NULL,'residence');
INSERT INTO employee_phone VALUES
  (1009,213,8278777,413,'office');
INSERT INTO employee_phone VALUES
  (1010,202,5456187,NULL,'Residence/office');
INSERT INTO employee_phone VALUES
  (1010,213,8278777,NULL,'office');
INSERT INTO employee_phone VALUES
  (1010,202,3239119,NULL,'Fed Bureau of Rules request name');
INSERT INTO employee_phone VALUES
  (1010,804,9230911,NULL,'Cream of the Crop');
INSERT INTO employee_phone VALUES
  (1010,313,4630300,NULL,'Colby Co');
INSERT INTO employee_phone VALUES
  (1010,312,0990988,NULL,'Cheap Rentals');
INSERT INTO employee_phone VALUES
  (1010,804,4563000,370,'Metro Savings');
INSERT INTO employee_phone VALUES
  (1010,804,4563000,375,'Metro Savings');
INSERT INTO employee_phone VALUES
  (1010,312,5692122,NULL,'Metro Savings');
INSERT INTO employee_phone VALUES
  (1011,213,8278777,422,'office');
```

```
INSERT INTO employee_phone VALUES
  (1011,213,3549138,NULL,'residence');
INSERT INTO employee_phone VALUES
  (1012,213,8278777,418,'office');
INSERT INTO employee_phone VALUES
  (1012,213,9788422,NULL,'residence');
INSERT INTO employee_phone VALUES
  (1013,213,8278777,411,'office');
INSERT INTO employee_phone VALUES
  (1013,213,9857506,NULL,'residence');
INSERT INTO employee_phone VALUES
  (1014,213,8278777,442,'office');
INSERT INTO employee_phone VALUES
  (1014,213,2528809,NULL,'residence');
INSERT INTO employee_phone VALUES
  (1015,213,8278777,436,'office');
INSERT INTO employee_phone VALUES
  (1015,213,3012906,NULL,'residence');
INSERT INTO employee_phone VALUES
  (1015,415,4491225,NULL,'A to Z system manager');
INSERT INTO employee_phone VALUES
  (1015,415,9237892,NULL,'Simple Instruments receptionist');
INSERT INTO employee_phone VALUES
  (1016,213,8278777,412,'office');
INSERT INTO employee_phone VALUES
  (1016,213,2925224,NULL,'residence');
INSERT INTO employee_phone VALUES
  (1017,213,8278777,425,'office');
INSERT INTO employee_phone VALUES
  (1017,213,9231070,NULL,'residence');
INSERT INTO employee_phone VALUES
```

```
  (1018,202,3239119,NULL,'Fed Bureau of Rules');
INSERT INTO employee_phone VALUES
  (1018,804,2989791,NULL,'residence/office');
INSERT INTO employee_phone VALUES
  (1019,213,8278777,418,'office');
INSERT INTO employee_phone VALUES
  (1019,213,2640855,NULL,'residence');
INSERT INTO employee_phone VALUES
  (1020,213,8278777,433,'office');
INSERT INTO employee_phone VALUES
  (1020,213,2248513,NULL,'residence');
INSERT INTO employee_phone VALUES
  (1021,213,8278777,428,'office');
INSERT INTO employee_phone VALUES
  (1021,213,2659291,NULL,'residence');
INSERT INTO employee_phone VALUES
  (1022,213,8278777,416,'office');
INSERT INTO employee_phone VALUES
  (1022,213,4982012,NULL,'residence');
INSERT INTO employee_phone VALUES
  (1023,212,7232121,NULL,'First Am. Bank system manager');
INSERT INTO employee_phone VALUES
  (1023,212,8283747,NULL,'residence/office');
INSERT INTO employee_phone VALUES
  (1023,609,5591011,213,'Sum Bank receptionist');
INSERT INTO employee_phone VALUES
  (1024,213,8278777,417,'office');
INSERT INTO employee_phone VALUES
  (1024,213,2724743,NULL,'residence');
INSERT INTO employee_phone VALUES
  (1025,213,8278777,429,'office');
```

```
INSERT INTO employee_phone VALUES
  (1025,213,2964652,NULL,'residence');

INSERT INTO job VALUES
  (111100,'Corporate President',0,0);
INSERT INTO job VALUES
  (412102,'Product Specialist',0,0);
INSERT INTO job VALUES
  (412103,'System Support Analyst',0,0);
INSERT INTO job VALUES
  (413201,'Dispatcher',0,0);
INSERT INTO job VALUES
  (412101,'Field Engineer',0,0);
INSERT INTO job VALUES
  (512101,'Sales Rep',0,0);
INSERT INTO job VALUES
  (312101,'Software Engineer',0,0);
INSERT INTO job VALUES
  (312102,'Hardware Engineer',0,0);
INSERT INTO job VALUES
  (322101,'Planning Specialist',0,0);
INSERT INTO job VALUES
  (432101,'Instructor',0,0);
INSERT INTO job VALUES
  (222101,'System Analyst',0,0);
INSERT INTO job VALUES
  (422101,'Software Analyst',0,0);
INSERT INTO job VALUES
  (104201,'Electronic Assembler',0,0);
INSERT INTO job VALUES
  (104202,'Mechanical Assembler',0,0);
```

```
INSERT INTO job VALUES
 (411100,'Manager - Customer Support',0,0);
INSERT INTO job VALUES
 (421100,'Manager - Software Support',0,0);
INSERT INTO job VALUES
 (431100,'Manager - Education',0,0);
INSERT INTO job VALUES
 (321100,'Manager - Product Planning',0,0);
INSERT INTO job VALUES
 (311100,'Manager - Research and Development',0,0);
INSERT INTO job VALUES
 (511100,'Manager - Marketing Sales',0,0);

INSERT INTO location VALUES
(05000000,0,'1294 Jefferson Blvd','Los Angeles','California',
951604032,NULL,NULL);
INSERT INTO location VALUES
(05000001,1,'101 Middlefield Rd','Palo
Alto','California',951604032,NULL,NULL);
INSERT INTO location VALUES
(05000002,2,'49 Fourth St','San Francisco','California',941031066,NULL,NULL);
INSERT INTO location VALUES
(33000003,3,'10366 25th St','New York City','New
York',105293045,NULL,NULL);
INSERT INTO location VALUES
(31000004,4,'55 Madison Av','Trenton','New Jersey',123419199,NULL,NULL);
INSERT INTO location VALUES
(09000005,5,'1 Lincoln Square','Washington','DC',156075555,NULL,NULL);
INSERT INTO location VALUES
(33000006,6,'10 River Rd','Schenectady','New York',123016166,NULL,NULL);
INSERT INTO location VALUES
```

```
(47000007,7,'4035 South 35th Av','Arlington','Virginia',222061016,NULL,NULL);
INSERT INTO location VALUES
(23000008,8,'1100 State St','Detroit','MI',484107888,NULL,NULL);
INSERT INTO location VALUES
(34000009,9,'4400 Greenwood Rd','Wilmington','NC',284031199,NULL,NULL);
INSERT INTO location VALUES
(05000010,10,'5171 El Camino Real','Palo Alto','California',94071,NULL,NULL);
INSERT INTO location VALUES
(05000011,11,'770 Hotel Dr','Menlo Park','California',940585151,NULL,NULL);
INSERT INTO location VALUES
(14000012,12,'510 Benton Av','Chicago','Illinois',606483930,NULL,NULL);
INSERT INTO location VALUES
(22000013,13,'1059 Kings Rd','Boston','Massachusetts',012104091,NULL,NULL);
INSERT INTO location VALUES
(47000014,14,'1690 Miller Av','Richmond','Virginia',223104121,NULL,NULL);
INSERT INTO location VALUES
(05000015,15,'687 Culver Blvd','Culver
City','California',900513965,NULL,NULL);
INSERT INTO location VALUES
(33000016,16,'2255 16th Av','New York City','New
York',105293033,NULL,NULL);
INSERT INTO location VALUES
(33000017,17,'4001 Harbor Blvd','Brooklyn','New
York',105431915,NULL,NULL);
INSERT INTO location VALUES
(33000018,18,'105 Time Square','New York City','New
York',105082682,NULL,NULL);
INSERT INTO location VALUES
(05000019,19,'567 El Camino Real','San
Mateo','California',942153219,NULL,NULL)
;
```

```
INSERT INTO location VALUES
(14000020,20,'876 Winston St','Chicago','Illinois',606316166,NULL,NULL);

INSERT INTO location_employee VALUES
 (05000001,1001);
INSERT INTO location_employee VALUES
 (05000002,1001);
INSERT INTO location_employee VALUES
 (33000003,1004);
INSERT INTO location_employee VALUES
 (31000004,1004);
INSERT INTO location_employee VALUES
 (09000005,1010);
INSERT INTO location_employee VALUES
 (33000006,1004);
INSERT INTO location_employee VALUES
 (47000007,1010);
INSERT INTO location_employee VALUES
 (23000008,1010);
INSERT INTO location_employee VALUES
 (34000009,1004);
INSERT INTO location_employee VALUES
 (05000010,1001);
INSERT INTO location_employee VALUES
 (05000011,1001);
INSERT INTO location_employee VALUES
 (14000012,1010);
INSERT INTO location_employee VALUES
 (22000013,1004);
INSERT INTO location_employee VALUES
 (47000014,1010);
```

```
INSERT INTO location_employee VALUES
  (05000015,1001);
INSERT INTO location_employee VALUES
  (33000016,1004);
INSERT INTO location_employee VALUES
  (33000017,1004);
INSERT INTO location_employee VALUES
  (33000018,1004);
INSERT INTO location_employee VALUES
  (33000019,1001);
INSERT INTO location_employee VALUES
  (14000020,1010);

INSERT INTO location_phone VALUES
  (05000000,213,8278777,101,'Corporate Presidents office',NULL);
INSERT INTO location_phone VALUES
  (05000001,415,4491221,NULL,'FEs office',NULL);
INSERT INTO location_phone VALUES
  (05000001,415,4491225,NULL,'System Manager',NULL);
INSERT INTO location_phone VALUES
  (05000001,415,4491244,NULL,'Secretary','available 9:00 to 5:00');
INSERT INTO location_phone VALUES
  (05000002,415,9237892,NULL,'Receptionist','ask for page');
INSERT INTO location_phone VALUES
  (33000003,212,7230101,NULL,'Computer Room',NULL);
INSERT INTO location_phone VALUES
  (33000003,212,7232121,NULL,'System Manager',NULL);
INSERT INTO location_phone VALUES
  (31000004,609,5591011,213,'Receptionist','leave message');
INSERT INTO location_phone VALUES
  (31000004,609,5591011,224,'System Manager',NULL);
```

```
INSERT INTO location_phone VALUES
  (31000004,609,5591011,225,'Computer Room',NULL);
INSERT INTO location_phone VALUES
  (09000005,202,3239119,NULL,'Switchboard',NULL);
INSERT INTO location_phone VALUES
  (33000006,212,5786099,NULL,'Small office',NULL);
INSERT INTO location_phone VALUES
  (47000007,804,9230911,NULL,'Switchboard',NULL);
INSERT INTO location_phone VALUES
  (23000008,313,4630300,NULL,'Receptionist',NULL);
INSERT INTO location_phone VALUES
  (34000009,919,9789000,NULL,'Receptionist',NULL);
INSERT INTO location_phone VALUES
  (34000009,919,9789000,601,'John Moore','Vice President');
INSERT INTO location_phone VALUES
  (05000010,415,2412021,NULL,'Alice Hamm','President');
INSERT INTO location_phone VALUES
  (05000011,415,3563560,NULL,'J.R. Stern','Owner');
INSERT INTO location_phone VALUES
  (14000012,312,9880988,NULL,'Tom Thumb','Owner');
INSERT INTO location_phone VALUES
  (22000013,617,7567676,NULL,'Computer Room',NULL);
INSERT INTO location_phone VALUES
  (22000013,617,7562918,NULL,'System Manager',NULL);
INSERT INTO location_phone VALUES
  (47000014,804,4563000,370,'Alan Monday','System Manager');
INSERT INTO location_phone VALUES
  (47000014,804,4563000,375,'Receptionist',NULL);
INSERT INTO location_phone VALUES
  (05000015,213,2872019,NULL,'Charles Cates','Owner');
INSERT INTO location_phone VALUES
```

```
  (33000016,212,8282828,NULL,'Andy Moore',NULL);
INSERT INTO location_phone VALUES
  (33000017,718,2243283,NULL,'various contacts',NULL);
INSERT INTO location_phone VALUES
  (33000018,212,4909190,NULL,'Tom Sellers',NULL);
INSERT INTO location_phone VALUES
  (05000019,415,6567000,NULL,'Receptionist',NULL);
INSERT INTO location_phone VALUES
  (14000020,312,5692122,NULL,'Receptionist',NULL);
INSERT INTO location_phone VALUES
  (14000020,312,5692136,NULL,'System Manager',NULL);

.LOGOFF
```

3、INSQL02.TXT

```
.LOGON Teradata_Education,Educate
/*********************************************************/
/* Creates Database CS_VIEWS and populates it with ten   */
/* views of the Customer_Service tables.    (RPK 3/97)  */
/*                                                       */
/* Update by P. Derouin 8/19/97                          */
/*********************************************************/

SELECT DatabaseName FROM DBC.Databases
WHERE  DatabaseName = 'CS_VIEWS';

.IF ActivityCount = 0 THEN .GoTo CreateVM
```

```
GRANT DROP DATABASE ON CS_VIEWS TO Teradata_Education;
DELETE DATABASE CS_VIEWS;
DROP   DATABASE CS_VIEWS;

.LABEL CreateVM

CREATE DATABASE CS_VIEWS FROM Teradata_Education AS PERM = 0
            ACCOUNT = ('$M_P0623');

GRANT SELECT ON Customer_Service to CS_VIEWS WITH GRANT OPTION;

DATABASE CS_VIEWS;

 CREATE VIEW contact
     (contact_number
     ,contact_name
     ,area_code
     ,phone
     ,extension
     ,last_call_date)
   AS
   SELECT
      contact_number
     ,contact_name
     ,area_code
     ,phone
     ,extension
     ,last_call_date
   FROM CUSTOMER_SERVICE.contact;
```

```
CREATE VIEW customer
   (customer_number
   ,customer_name
   ,parent_customer_number
   ,sales_employee_number)
 AS
 SELECT
    customer_number
   ,customer_name
   ,parent_customer_number
   ,sales_employee_number
 FROM CUSTOMER_SERVICE.customer;

CREATE VIEW department
   (department_number
   ,department_name
   ,budget_amount
   ,manager_employee_number)
 AS
 SELECT
    department_number
   ,department_name
   ,budget_amount
   ,manager_employee_number
 FROM CUSTOMER_SERVICE.department;

CREATE VIEW employee
   (employee_number
   ,manager_employee_number
   ,department_number
   ,job_code
```

```
  ,last_name
  ,first_name
  ,hire_date
  ,birthdate
  ,salary_amount)
 AS
 SELECT
   employee_number
  ,manager_employee_number
  ,department_number
  ,job_code
  ,last_name
  ,first_name
  ,hire_date
  ,birthdate
  ,salary_amount
 FROM CUSTOMER_SERVICE.employee;

CREATE VIEW employee_phone
  (employee_number
  ,area_code
  ,phone
  ,extension
  ,comment_line)
 AS
 SELECT
   employee_number
  ,area_code
  ,phone
  ,extension
  ,comment_line
```

```
 FROM CUSTOMER_SERVICE.employee_phone;

CREATE VIEW job
  (job_code
  ,description
  ,hourly_billing_rate
  ,hourly_cost_rate)
 AS
 SELECT
   job_code
  ,description
  ,hourly_billing_rate
  ,hourly_cost_rate
 FROM CUSTOMER_SERVICE.job;

CREATE VIEW location
  (location_number
  ,customer_number
  ,first_address_line
  ,city
  ,state
  ,zip_code
  ,second_address_line
  ,third_address_line)
 AS
 SELECT
   location_number
  ,customer_number
  ,first_address_line
  ,city
  ,state
```

```
  ,zip_code
  ,second_address_line
  ,third_address_line
 FROM CUSTOMER_SERVICE.location;

CREATE VIEW location_employee
  (location_number
  ,employee_number)
 AS
 SELECT
   location_number
  ,employee_number
 FROM CUSTOMER_SERVICE.location_employee;

CREATE VIEW location_phone
  (location_number
  ,area_code
  ,phone
  ,extension
  ,description
  ,comment_line)
 AS
 SELECT
   location_number
  ,area_code
  ,phone
  ,extension
  ,description
  ,comment_line
 FROM CUSTOMER_SERVICE.location_phone;
```

```
CREATE VIEW emp
  (emp
  ,mgr
  ,dept
  ,job
  ,last
  ,first
  ,hire
  ,birth
  ,sal)
 AS
 SELECT
   employee_number
  ,manager_employee_number
  ,department_number
  ,job_code
  ,last_name
  ,first_name
  ,hire_date
  ,birthdate
  ,salary_amount
 FROM CUSTOMER_SERVICE.employee;

.LOGOFF
```

4、INSQL03.TXT

```
.LOGON Teradata_Education,Educate
```

```
/************************************************/
/* Create SQL Class Userids: SQL00 TO SQL20     */
/*       from newly created userid SQL          */
/************************************************/
/* Note: This script will fail if these userids  */
/*       are already defined. To cleanup old SQL */
/*       userids, run the script: rmsql01.txt    */
/************************************************/
/* Spool limit, account code values, and also    */
/* access rights to the sample views and tables  */
/* will be inherited from user SQL.   (RPK 3/97) */
/*                                               */
/* Updated by P.Derouin (8/19/97)                */
/************************************************/

CREATE USER SQL FROM Teradata_Education AS
     PASSWORD = SQL PERM = 12000000
     SPOOL = 500000 ACCOUNT = ('$M_P0623');

GRANT SELECT ON Customer_Service TO ALL SQL;
GRANT SELECT ON CS_VIEWS         TO ALL SQL;

 CREATE USER SQL00 FROM SQL AS  PASSWORD = SQL00
      PERM = 250000 DEFAULT DATABASE = CS_VIEWS;
 CREATE USER SQL01 FROM SQL AS  PASSWORD = SQL01
      PERM = 250000 DEFAULT DATABASE = CS_VIEWS;
 CREATE USER SQL02 FROM SQL AS  PASSWORD = SQL02
      PERM = 250000 DEFAULT DATABASE = CS_VIEWS;
 CREATE USER SQL03 FROM SQL AS  PASSWORD = SQL03
      PERM = 250000 DEFAULT DATABASE = CS_VIEWS;
 CREATE USER SQL04 FROM SQL AS  PASSWORD = SQL04
```

```
    PERM = 250000 DEFAULT DATABASE = CS_VIEWS;
CREATE USER SQL05 FROM SQL AS  PASSWORD = SQL05
    PERM = 250000 DEFAULT DATABASE = CS_VIEWS;
CREATE USER SQL06 FROM SQL AS  PASSWORD = SQL06
    PERM = 250000 DEFAULT DATABASE = CS_VIEWS;
CREATE USER SQL07 FROM SQL AS  PASSWORD = SQL07
    PERM = 250000 DEFAULT DATABASE = CS_VIEWS;
CREATE USER SQL08 FROM SQL AS  PASSWORD = SQL08
    PERM = 250000 DEFAULT DATABASE = CS_VIEWS;
CREATE USER SQL09 FROM SQL AS  PASSWORD = SQL09
    PERM = 250000 DEFAULT DATABASE = CS_VIEWS;
CREATE USER SQL10 FROM SQL AS  PASSWORD = SQL10
    PERM = 250000 DEFAULT DATABASE = CS_VIEWS;
CREATE USER SQL11 FROM SQL AS  PASSWORD = SQL11
    PERM = 250000 DEFAULT DATABASE = CS_VIEWS;
CREATE USER SQL12 FROM SQL AS  PASSWORD = SQL12
    PERM = 250000 DEFAULT DATABASE = CS_VIEWS;
CREATE USER SQL13 FROM SQL AS  PASSWORD = SQL13
    PERM = 250000 DEFAULT DATABASE = CS_VIEWS;
CREATE USER SQL14 FROM SQL AS  PASSWORD = SQL14
    PERM = 250000 DEFAULT DATABASE = CS_VIEWS;
CREATE USER SQL15 FROM SQL AS  PASSWORD = SQL15
    PERM = 250000 DEFAULT DATABASE = CS_VIEWS;
CREATE USER SQL16 FROM SQL AS  PASSWORD = SQL16
    PERM = 250000 DEFAULT DATABASE = CS_VIEWS;
CREATE USER SQL17 FROM SQL AS  PASSWORD = SQL17
    PERM = 250000 DEFAULT DATABASE = CS_VIEWS;
CREATE USER SQL18 FROM SQL AS  PASSWORD = SQL18
    PERM = 250000 DEFAULT DATABASE = CS_VIEWS;
CREATE USER SQL19 FROM SQL AS  PASSWORD = SQL19
    PERM = 250000 DEFAULT DATABASE = CS_VIEWS;
```

```
CREATE USER SQL20 FROM SQL AS  PASSWORD = SQL20
    PERM = 250000 DEFAULT DATABASE = CS_VIEWS;

.LOGOFF
```

然后利用BTEQ或Queryman等工具来按秩序运行上面的脚本文件。以Windows BTEQ为例，假设上述脚本文件存放在\TNT_Sql目录下，则执行方法如下：

```
.run file=\TNT_Sql\INTED00.TXT
.run file=\TNT_Sql\INSQL01.TXT
.run file=\TNT_Sql\INSQL02.TXT
.run file=\TNT_Sql\INSQL03.TXT
```

这样就创建了测试用的数据库，可以利用BTEQ或Queryman等前端工具来学习Teradata SQL，完成本书中所介绍的例子和练习。

附录三　习题答案

Lab 4_1

```
HELP DATABASE CS_VIEWS;
HELP DATABASE customer_service;
```

Lab 4_2

```
HELP TABLE customer_service.department;
```

Lab 4_3

```
SHOW VIEW CS_VIEWS.emp;
```

Lab 4_4

```
EXPLAIN SELECT      *
   FROM department
   ORDER BY department_name;
```

Lab 4_5

```
SHOW TABLE  customer_service.employee_phone;
```

Lab 4_6

```
HELP INDEX customer_service.customer;
```

Lab 4_7

```
DATABASE customer_service;
SELECT * FROM emp;
```

(该语句将出错，因为视图EMP不在数据库customer_service中)

```
SELECT * FROM CS_VIEWS.emp;
```

(也可以改变缺省数据库后再执行前面的SQL语句)

```
DATABASE  CS_VIEWS;
SELECT * FROM EMP;
```

Lab 5_1

```
SELECT employee_number
   ,last_name
   ,salary_amount
   ,job_code
FROM    employee
WHERE  salary_amount > =      40001
AND       salary_amount < =      50000
ORDER BY       last_name;
```

```
SELECT employee_number
   ,last_name
   ,salary_amount
   ,job_code
FROM    employee
WHERE  salary_amount BETWEEN 40001 AND 50000
ORDER BY salary_amount DESC;
```

Lab 5_2

```
SELECT employee_number
   ,department_number
   ,salary_amount
FROM    employee
WHERE  salary_amount > 30000
AND       department_number    IN       (501,301,201)
ORDER BY department_number
   ,salary_amount;
```

Lab 5_3

```
SELECT location_number
   ,area_code
   ,phone
   ,extension
   ,description
FROM    location_phone
WHERE  description  LIKE  '%Manager%'
ORDER BY       location_number;
```

Lab 5_4

```
SELECT customer_number
   ,customer_name
   ,sales_employee_number
FROM    customer
WHERE  parent_customer_number IS NULL
AND      sales_employee_number  IS NOT NULL
ORDER BY 1;
```

Lab 5_5

```
SELECT location_number
   ,area_code
   ,phone
   ,description
FROM    location_phone
WHERE  area_code  =  415
AND      description  NOT  LIKE  '%Manager%'
ORDER BY       phone;
```

Lab 5_6

```
SELECT job_code
   ,description
FROM    job
WHERE  (job_code BETWEEN  400000  AND  499999
AND      description LIKE '%manager%')
OR         (description LIKE '%analyst%')
ORDER BY       description;
```

Lab 5_7

```
SELECT location_number
   ,area_code
   ,phone
   ,extension
FROM    location_phone
WHERE  extension IS NULL
AND       area_code IN (415,619)
ORDER BY       area_code
   ,phone;
```

Lab 6_1

```
SELECT DATE
   , TIME
   , USER;

SELECT DATE + 365;

SELECT EXTRACT (MONTH FROM DATE + 60);
```

Lab 6_2

```
SELECT 'Math Function'
   ,1
   ,2*3+4*5;

SELECT 'Math Function'
```

```
   ,1
   ,2*(3+4)*5;
```

Lab 6_3

```
SELECT CAST (last_name AS CHAR(10))
   ,salary_amount
   ,salary_amount * 1.1
FROM    employee
WHERE  department_number = 401
ORDER BY       salary_amount DESC;
```

Lab 6_4

```
SELECT EXTRACT (YEAR FROM (ADD_MONTHS (DATE, 15)));
```

Lab 6_5

```
SELECT CAST (last_name AS CHAR(10))
   ,salary_amount * 1.1
   ,salary_amount + 500
   ,(salary_amount * 1.1) - (salary_amount + 500)
FROM    employee
WHERE  department_number = 401
ORDER BY       salary_amount  DESC;
```

Lab 6_6

```
SELECT CAST (last_name  AS CHAR(10) )
```

```
   ,department_number
   ,hire_date
   ,(date - birthdate) / 365
FROM    employee
WHERE  (date - birthdate) / 365 <  40
ORDER BY birthdate DESC;
```

Lab 7_1

```
CREATE MACRO  yourID.LAB7_1  AS
   (SELECT location_number
   ,customer_number
   ,CAST (city AS CHAR(14))
   ,CAST (state AS CHAR(14))
   FROM location
   ORDER BY state
   ,city;
   );
EXEC yourID.LAB7_1;
```

Lab 7_2

```
SHOW MACRO yourID.LAB7_1;
*** Text of DDL statement returned.

CREATE MACRO  SQLT4.LAB7_1  AS
   (SELECT       location_number
   ,customer_number
   ,CAST (city AS         CHAR(14))
   ,CAST (state AS CHAR(14))
```

```
   FROM location
   ORDER BY    state
   ,city;
   );

CREATE MACRO  LAB7_2  AS
   (SELECT       customer_number
   ,CAST (city AS CHAR(14))
   ,CAST (state AS CHAR(14))
   ,zip_code
   FROM location
   WHERE         state = 'Illinois'
   ORDER BY    state
   ,city
   ,zip_code;
   );

EXEC yourID.LAB7_2;
```

Lab 7_3

```
HELP DATABASE yourID;
```

Lab 7_4

```
REPLACE   MACRO   LAB7_1   AS
   (SELECT       location_number
   ,customer_number
   ,CAST (city AS CHAR(14))
   ,first_address_line
```

```
   FROM location
   WHERE        city = 'Chicago'
   ORDER BY    customer_number DESC;
   );

EXEC LAB7_1;
```

Lab 7_5

```
DROP   MACRO   yourID.LAB7_1;
```

Lab 8_1

```
SELECT 'TRUE'
WHERE EXISTS
   (SELECT job_code FROM job
   WHERE job_code NOT IN
            (SELECT job_code FROM employee));
```

Lab 8_2

```
SELECT last_name
   ,first_name
   ,job_code
FROM    employee
WHERE  job_code NOT IN
   (SELECT job_code
   FROM job);
```

```
SELECT last_name
   ,first_name
   ,job_code
FROM    employee
WHERE  job_code NOT = ALL
   (SELECT job_code
   FROM job);
```

Lab 8_3

```
SELECT employee_number
   ,last_name
   ,department_number
FROM    employee
WHERE  department_number IS NOT NULL
AND      department_number NOT IN
   (SELECT department_number
   FROM department);
```

Lab 8_4

```
SELECT customer_number
   ,customer_name
   ,sales_employee_number
FROM    customer
WHERE  customer_number IN
   (SELECT customer_number
   FROM location
   WHERE state = 'California')
ORDER BY sales_employee_number;
```

Lab 8_5

```
SELECT employee_number
   ,last_name
   ,department_number
   ,job_code
FROM    employee
WHERE  job_code = 211100
AND      employee_number IN
   (SELECT manager_employee_number
   FROM department);
```

Lab 8_6

```
SELECT employee_number
   ,last_name
   ,department_number
   ,job_code
FROM    employee
WHERE  manager_employee_number IN
(SELECT manager_employee_number
FROM    department
WHERE  manager_employee_number IN
   (SELECT employee_number
   FROM employee
   WHERE job_code = 211100) );

SELECT employee_number
   ,last_name
```

```
   ,department_number
   ,job_code
FROM     employee
WHERE  manager_employee_number = ANY
   (SELECT manager_employee_number
   FROM department
   WHERE manager_employee_number = ANY
            (SELECT employee_number
            FROM employee
            WHERE job_code   =   211100) );
```

Lab 9_1

```
SELECT last_name
   ,hire_date (FORMAT 'mmmbdd,byyyy')
   ,salary_amount/26 (FORMAT '$$$,$$9.99')
   (TITLE 'Salary Amount')
FROM     employee
WHERE  department_number   =   401
ORDER BY salary_amount DESC;
```

Lab 9_2

```
SELECT employee_number      (TITLE 'Employee')
   ,phone (FORMAT '999-9999')   (TITLE 'Phone')
FROM     employee_phone
ORDER BY employee_number;
```

Lab 9_3

```
SELECT employee_number
   ,last_name      (TITLE 'Last Name')
   ,first_name     (TITLE 'First Name') (CHAR (9))
   ,(DATE - birthdate)/365 (NAMED Age)
FROM    employee
WHERE  department_number = 301
ORDER BY       age;
```

Lab 9_4

```
SELECT CAST (last_name AS char(10))
   ,department_number
   ,hire_date
FROM    employee
WHERE  (EXTRACT (MONTH FROM hire_date)) =
   (EXTRACT (MONTH FROM DATE))
ORDER BY       hire_date;

SELECT CAST (last_name AS char(10))
   ,department_number
   ,hire_date
FROM    employee
WHERE  (EXTRACT (MONTH FROM hire_date)) =
   (EXTRACT (MONTH FROM DATE)) - 4
ORDER BY       hire_date;
```

Lab 9_5

```
SELECT CAST (last_name AS char(10))
   ,department_number
```

```
   ,hire_date
   ,hire_date (FORMAT 'mmm') (TITLE 'month')
FROM    employee
WHERE  EXTRACT (MONTH FROM hire_date) =
            EXTRACT (MONTH FROM DATE)
ORDER BY EXTRACT (MONTH FROM hire_date);

SELECT CAST (last_name AS char(10))
   ,department_number
   ,hire_date
   ,hire_date (FORMAT 'mmm') (TITLE 'month')
FROM    employee
WHERE  EXTRACT (MONTH FROM hire_date) =
            EXTRACT (MONTH FROM DATE)-4
ORDER BY EXTRACT (MONTH FROM hire_date);
```

Lab 9_6

```
SELECT last_name       (FORMAT 'x(8)')
   ,first_name     (FORMAT 'x(8)')
   ,EXTRACT (MONTH FROM hire_date) (TITLE 'month')
   ,hire_date       (FORMAT 'mmm') (TITLE ' ')
   ,hire_date  (FORMAT 'yyyy')(TITLE 'year')
FROM    employee
WHERE  department_number  =  501
ORDER BY       3;

SELECT last_name       (FORMAT   'x(8) ' )
   ,first_name (FORMAT    'x(8) ' )
   , EXTRACT (MONTH FROM hire_date) (TITLE  'month' )
   ,hire_date (FORMAT    'mmm' ) (TITLE  '  ')
```

```
   ,hire_date (FORMAT    'yyyy' ) (TITLE  'year' )
FROM    employee
WHERE  department_number    =  501
ORDER BY       4;
```

Lab 10_1

```
SELECT  e.employee_number
            ,CAST (e.last_name AS CHAR(9))
            ,e.salary_amount
            ,CAST (j.description AS CHAR(20))
FROM    employee e
            INNER JOIN  job j
ON          e.job_code = j.job_code
WHERE  e.department_number <> 100
ORDER BY j.description
            ,e.employee_number;
```

Lab 10_2

```
SELECT l.customer_number
   ,c.customer_name
   ,c.sales_employee_number
FROM    customer c
            INNER JOIN  location l
ON          c.customer_number = l.customer_number
WHERE  l.state   = 'California'
ORDER BY 1;
```

Lab 10_3

```
DATABASE customer_service;

SELECT e.employee_number
   ,e.department_number
   ,m.last_name
FROM     employee e
            INNER JOIN   employee m
ON e.manager_employee_number = m.employee_number
WHERE e.hire_date < 780101
ORDER BY 2;
```

Lab 10_4

```
EXPLAIN
SELECT  e.job_code
            ,description
            ,last_name
FROM     employee e CROSS JOIN job j;
```

Lab 10_5

```
SELECT e.last_name
   ,d.department_name
   ,j.description
   ,p.phone
FROM     employee e
            INNER JOIN department d
ON          e.department_number = d.department_number
            INNER JOIN  job j
```

```
ON        e.job_code = j.job_code
          INNER JOIN  employee_phone p
ON        e.employee_number = p.employee_number
WHERE  p.area_code = 213
AND       e.job_code = 412101
ORDER BY e.last_name;
```

另一种方法：

```
SELECT e.last_name
   ,d.department_name
   ,j.description
   ,p.phone
FROM    employee e
   ,department d
   ,job j
   ,employee_phone p
WHERE  p.area_code = 213
AND       e.job_code = 412101
AND       e.job_code = j.job_code
AND       e.department_number = d.department_number
AND       e.employee_number  = p.employee_number
ORDER BY e.last_name;
```

Lab 11_1

```
CREATE TABLE yourID.emp_new ,FALLBACK
          ,NO BEFORE JOURNAL
          ,NO AFTER JOURNAL
   (employee_number     INTEGER
   ,manager_employee_number INTEGER
```

```
   ,department_number   INTEGER
   ,job_code        INTEGER
   ,last_name       CHAR(20) NOT NULL
   ,first_name      VARCHAR(30) NOT NULL
   ,hire_date       DATE NOT NULL
   ,birthdate       DATE NOT NULL
   ,salary_amount          DECIMAL(10,2) NOT NULL
   )
UNIQUE PRIMARY INDEX (employee_number);
```

Lab 11_2

```
DATABASE yourID;
CREATE TABLE dept_new, FALLBACK
            NO BEFORE JOURNAL, NO AFTER JOURNAL
   (department_number NOT NULL
                              CONSTRAINT primary_1 PRIMARY KEY,
    department_name CHAR (30) NOT CS NOT NULL,
    budget_amount DEC (10,2),
    manager_employee_number INT);

SHOW TABLE dept_new;

ALTER TABLE dept_new
ADD CONSTRAINT unique_nm UNIQUE (department_name);

SHOW TABLE dept_new;

ALTER TABLE dept_new
ADD CONSTRAINT hundred_plus CHECK (department_number > 99);
```

```
SHOW TABLE dept_new;

ALTER TABLE dept_new
ADD CONSTRAINT mgr_ref FOREIGN KEY (manager_employee_number)
                        REFERENCES emp_new (employee_number);
```

Lab 11_3

```
HELP CONSTRAINT dept_new.unique_nm;

HELP CONSTRAINT dept_new.hundred_plus;

HELP CONSTRAINT dept_new.mgr_ref;
```

Lab 11_4

```
ALTER TABLE userid.emp_new,         NO FALLBACK
   ADD salary_level BYTEINT BETWEEN 3 AND 120 ;

CREATE  UNIQUE  INDEX   (salary_level)   ON   userid.emp_new;
```

Lab 11_5

```
HELP INDEX userid.emp_new;
```

Lab 11_6

```
DROP INDEX (salary_level) ON userid.emp_new;
```

```
ALTER TABLE emp_new       ,FALLBACK
   DROP salary_level ;
```

Lab 12_1

```
INSERT  INTO userid.emp_new
SELECT *
FROM    employee ;
```

Lab 12_2

```
SELECT employee_number
   ,last_name
   ,job_code
   ,salary_amount
FROM    userid.emp_new
WHERE  department_number = 301;
```

Lab 12_2, CONTINUED

```
UPDATEuserid.emp_new
SET       salary_amount   =   salary_amount   *   1.15
WHERE  department_number   =   301;

SELECT employee_number
   ,last_name
   ,job_code
   ,salary_amount
FROM    emp_new
WHERE  department_number   =   301;
```

Lab 12_3

```
UPDATEuserid.emp_new
SET       salary_amount   = salary_amount   /  1.15
WHERE  department_number = 301;

SELECT employee_number
   ,last_name
   ,job_code
   ,salary_amount
FROM    userid.emp_new
WHERE  department_number = 301;
```

Lab 12_4

```
INSERT INTO userid.dept_new
SELECT * FROM department;
```

该语句将出错，因为与约束有冲突

Lab 12_5

```
INSERT INTO dept_new
SELECT * FROM department
WHERE department.manager_employee_number
          = employee.employee_number;
```

Lab 12_6

```
DATABASE userid;
```

INSERT INTO dept_new
VALUES (99, ew dept', 900000,1021);

返回：Fails: violates hundred_plus constraint.

INSERT INTO dept_new
VALUES (400, ducation', 900000,1021);

返回：Fails: violates unique_nm constraint.

INSERT INTO dept_new
VALUES (400, ew dept', 900000,1099);

返回：Fails: violates mgr_ref constraint.

INSERT INTO dept_new
VALUES (400, ew dept', 900000,1021);

返回：Succeeds.

Lab 13_1

```
DATABASE userid;
CREATE MACRO       new_dept
                   ( dept   INTEGER
                   , budget         DEC(10,2)
                   , name CHAR(30)
                   , mgr    INTEGER)
AS
(  INSERT INTO         dept_new
```

```
                    ( department_number
                    , department_name
                    , budget_amount
                    , manager_employee_number)
          VALUES    (  :dept
                    ,  :name
                    ,  :budget
                    ,  :mgr );
  SELECT        department_number   (TITLE    'number')
                ,department_name              (TITLE 'name')
                ,budget_amount                (TITLE 'budget)
                ,manager_employee_number (TITLE 'manager')
  FROM dept_new
  WHERE         department_number = :dept;
);

EXEC   new_dept   (601,    'Shipping', 800000, NULL) ;

EXEC   new_dept   (701, 'Credit', 1200000, 1018) ;

EXEC   new_dept   (905,    ' President Staff', 5000000, 801) ;
```

Lab 13_2

```
EXPLAIN EXEC Lab13_1
  (601, 'Shipping', 800000, NULL) ;
```

Lab 13_3

```
DATABASE userid;
```

```
CREATE MACRO        lab13_3
   (dept    INTEGER
   ,raise   DECIMAL   (5))
AS
(SELECT          employee_number
   ,last_name       (char   (10)  )
   ,salary_amount (format   ' $$, $$$, $$9.99'  )
                    (title   'Current//Salary'  )
   ,salary_amount + :raise
   (format   '$$, $$$, $$9.99'  )
   (title   'New//Salary'  )
FROM    emp_new
WHERE  department_number = :dept
ORDER BY        2;
UPDATEemp_new
SET        salary_amount   =   salary_amount   +   :raise
WHERE  department_number    =         :dept;
);

EXEC     Lab13_3   (401,1000) ;
EXEC     Lab13_3   (401, -1000) ;
```

Lab 13_4

```
DATABASE userid;
CREATE MACRO        lab13_4
   (low    DECIMAL (10,2)
   ,high    DECIMAL (10,2))
AS
(SELECT          last_name
   ,department_number
```

```
   ,salary_amount
FROM    emp_new
WHERE salary_amount BETWEEN    :low    AND   :high
ORDER BY       3;
);

EXEC     Lab13_4 (50001,99999);
EXEC     Lab13_4   (high   =   29999,   low   =   0) ;
```

Lab 14_1

```
SELECT e.department_number (TITLE 'Dept//Nbr')
   ,d.department_name   (TITLE 'Dept//Name')
            (FORMAT 'X(9)')
   ,MIN(e.salary_amount)         (TITLE 'Minimum//Salary')
            (FORMAT 'Z,ZZZ,ZZ9.99')
   ,MAX(e.salary_amount)         (TITLE 'Maximum//Salary')
            (FORMAT 'Z,ZZZ,ZZ9.99')
   ,SUM(e.salary_amount)         (TITLE 'Current//Salary')
            (FORMAT 'Z,ZZZ,ZZ9.99')
   ,SUM(e.salary_amount * 1.1)(TITLE 'Adjusted//Salary')
            (FORMAT 'Z,ZZZ,ZZ9.99')
FROM    employee e
            INNER JOIN  department d
ON         e.department_number = d.department_number
GROUP BY       e.department_number
            ,d.department_name
ORDER BY       e.department_number;
```

Lab 14_2

```
SELECT department_number
   ,job_code
   ,COUNT(job_code)
   ,AVG(salary_amount)
FROM    employee
GROUP BY       department_number
   ,job_code
ORDER BY       department_number
   ,job_code;
```

Lab 15_1

```
SELECT c.customer_name
   ,l.customer_number
   ,c.sales_employee_number
FROM    customer c
            INNER JOIN  location l
ON          c.customer_number = l.customer_number
WHERE  l.state   = 'California'
ORDER BY       sales_employee_number
WITH    COUNT(l.customer_number) (TITLE 'Total Cust');
```

Lab 15_2

```
SELECT last_name       (FORMAT 'X(10)')
   ,first_name     (FORMAT 'X(10)')
   ,department_number (NAMED Dept)
   ,job_code       (NAMED Job)
   ,salary_amount          (NAMED Salary)
```

```
FROM    employee
WHERE   department_number   IN      (401, 403)
AND     job_code    NOT   IN    (411100, 413201, 431100)
WITH    SUM(salary)   (TITLE 'Job Total')    BY Job
WITH    AVG(salary)   (TITLE 'Job Avg')     BY Job
WITH    SUM(salary)   (TITLE 'Dept Total')  BY Dept
WITH    AVG(salary)   (TITLE 'Dept Avg')    BY Dept
WITH    SUM(salary)   (TITLE 'Grand Total')
        ,AVG(salary)  (TITLE 'Grand Average')
ORDER BY      last_name;
```

Lab 16_1

```
SELECT phone  (FORMAT '999-9999')
  ,'Employee'     (TITLE 'Owner')
FROM    employee_phone
WHERE   area_code = 415
UNION
SELECT phone
  ,'Location'
FROM    location_phone
WHERE   area_code = 415
ORDER  BY 1 ;
```

Lab 16_2

```
SELECT employee_number
FROM    employee
WHERE   employee_number     IN
  (SELECT       manager_employee_number
```

```
   FROM     department);

SELECT employee_number
FROM     employee
INTERSECT
SELECT manager_employee_number
FROM     department;
```

Lab 16_3

```
SELECT job_code        (title 'INVALID JOB CODES')
FROM     employee
EXCEPT
SELECT job_code
FROM     job;
```

Lab 17_1

```
DATABASE userid;
CREATE VIEW  sales_territory AS
   SELECT        l.customer_number
   ,c.customer_name
   ,c.sales_employee_number
   FROM customer c
   ,location l
   WHERE c.customer_number = l.customer_number
   AND   l.state =_'California';
```

是否可以通过该视图更新数据？不行，因为同时访问了多个表。

Lab 17_2

```
DATABASE userid;
SELECT *
FROM    sales_territory;
```

Lab 17_3

```
DATABASE userid;
CREATE VIEW dept_sal (dep, sumsal)
AS
SELECT  department_number
        ,SUM(salary_amount)
FROM customer_service.employee
GROUP BY 1;

SELECT * FROM dept_sal;
```

Lab 17_4

```
DATABASE      yourID;
SELECT department_name
        , sumsal
FROM    departtment d
        INNER JOIN  dept_sal ds
ON      ds.dep = d.department_number
WHERE  sumsal > 100000;
```

Lab 17_5

```
DATABASE userid;
CREATE VIEW cust AS
   SELECT        customer_number      (FORMAT '9(4)'  ) (TITLE 'Cust #')
   ,customer_name        (TITLE 'Name'  )
   ,parent_customer_number     (FORMAT '9(4)'  ) (TITLE 'Parent')
   ,sales_employee_number      (FORMAT '9(4)'  ) (TITLE 'Rep')
   FROM customer_service.customer;
SELECT *
FROM    cust;
```

Lab 18_1

```
SELECT e.last_name
   ,e.first_name   (FORMAT 'X(10)')
   ,'('|| SUBSTRING(p.area_code FROM 4 FOR 3) || ')' ||
   SUBSTRING(p.phone FROM 5 FOR 3)  || '-' ||
   SUBSTRING(p.phone FROM 8)      (TITLE 'Phone Number')
FROM    employee e
   ,employee_phone p
WHERE  p.area_code = 609
AND      e.employee_number = p.employee_number;
```

Lab 18_2

```
SELECT description
   ,INDEX(description,'Analyst')
FROM    job
WHERE  INDEX(description,'Analyst')  >  0;
```

Lab 18_3

```
SELECT e.last_name
   ,e.first_name   (FORMAT 'X(10)')
   ,'(' || SUBSTRING(p.area_code FROM 4 FOR 3) || ')' ||
   SUBSTRING(p.phone FROM 5 FOR 3) || '-' || SUBSTRING(p.phone FROM 8) ||
' ' || (SUBSTRING(COALESCE(p.extension,0)) FROM 1)
   (TITLE 'Phone Number         Ext')
FROM    employee e
   ,employee_phone p
WHERE  p.area_code = 609
AND       e.employee_number = p.employee_number;
```

Lab 18_4

```
SELECT SUBSTRING(contact_name FROM INDEX(contact_name,', ')+2)
   (TITLE 'First Name')
   ,SUBSTRING(contact_name FROM 1 FOR INDEX(contact_name,',')-1)
   (TITLE 'Last Name')
   ,phone
FROM    contact
WHERE  area_code = 408
ORDER BY       1;
```

Lab 19_1

```
SELECT  e.job_code
            ,description
            ,last_name
```

```
FROM    employee e
        LEFT JOIN job j
ON      e.job_code = j.job_code;
```

Lab 19_2

```
SELECT e.last_name
   ,j.description
   ,d.department_name   (FROMAT   (13)')
FROM    employee e
        INNER JOIN department d
ON e.department_number = d.department_number
        INNER JOIN job j
ON e.job_code = j.job_code
ORDER BY 1;
```

Lab 19_3

```
SELECT e.last_name
   ,j.description
   ,d.department_name
FROM    employee e
        INNER JOIN department d
ON      e.department_number = d.department_number
        LEFT JOIN job j
ON e.job_code = j.job_code
ORDER BY  1;
```

Lab 19_4

```
SELECT e.last_name
   ,j.description
   ,d.department_name
FROM    employee e
            INNER JOIN department d
ON e.department_number = d.department_number
            RIGHT JOIN job j
ON e.job_code = j.job_code
ORDER BY 1;
```

Lab 20_1

```
DATABASE customer_service;

SELECT last_name, salary_amount, department_number
FROM employee ee
WHERE salary_amount >
            (SELECT AVG(salary_amount) FROM employee em
             WHERE ee.department_number = em.department_number)
ORDER BY 3, 2 DESC;
```

Lab 20_2

```
SELECT  last_name       AS last
            ,salary_amount          (FORMAT '$,$$$,$99.99') AS sal
            ,department_number   AS dep
            ,avgsal             FORMAT ( '$,$$$,$99.99')
FROM (SELECT AVG(salary_amount),
                    department_number
                    FROM employee
```

```
                    GROUP BY department_number)
    my_temp (avgsal, deptno)
   ,employee ee
WHERE  sal > avgsal
     AND dep = deptno
ORDER BY dep, sal DESC;
```

Lab 20_3

```
SELECT department_number
FROM department
WHERE department_number NOT IN
          (SELECT department_number FROM employee);

SELECT department_number
FROM department de
WHERE NOT EXISTS
          (SELECT * from employee EE
           WHERE ee.department_number = de.department_number);
```

Lab 20_4

```
SELECT last_name, salary_amount, e.department_number,
d.department_name (FORMAT 'X(12)',TITLE 'department')
FROM    employee e
          INNER JOIN department d
ON        e.department_number = d.department_number
WHERE  salary_amount >
          (SELECT AVG(salary_amount) FROM employee em
           WHERE e.department_number = em.department_number)
```

```
ORDER BY 3, 2 DESC;
```

Lab 21_1

```
SELECT CAST(SUM(
   CASE department_number
   WHEN 401 THEN budget_amount
   WHEN 403 THEN budget_amount
   ELSE 0
   END) / SUM(budget_amount) * 100 as numeric(2,0))
   (title '401/403//Bgt Ratio', FORMAT '99%')
 FROM department;
```

Lab 21_2

```
SELECT CAST(SUM(
   CASE
   WHEN department_number = 401
         THEN budget_amount
   WHEN department_number = 403
         THEN budget_amount * 1.05
   ELSE 0
   END) / SUM(CASE   WHEN department_number = 403
                     THEN budget_amount * 1.05
                     ELSE budget_amount
                     END ) AS NUMERIC (2,2) )
AS Dept_401_403_bgtl_ratio
FROM department;
```

Lab 21_3

```
SELECT SUM (COALESCE (budget_amount,0))       (TITLE   'Total')
   ,AVG (COALESCE (budget_amount,0))      (TITLE   'Avg')
   ,MIN (COALESCE (budget_amount,0))      (TITLE   'Min')
   ,MAX (COALESCE (budget_amount,0))      (TITLE   'Max')
FROM    department;

SELECT SUM(budget_amount)          (TITLE   'Total')
   ,AVG(budget_amount)         (TITLE   'Avg')
   ,MIN(budget_amount)         (TITLE   'Min')
   ,MAX(budget_amount)         (TITLE   'Max')
FROM    department;
```

Lab 21_4

```
SELECT department_number    (TITLE   'Dept' ) (FORMAT  'ZZZ9')
   ,COUNT(salary_amount)      (TITLE   'Tot #' ) (FORMAT  'ZZZ9')
   ,SUM(salary_amount)(TITLE   'Tot Sal' )
   ,AVG(salary_amount)(TITLE   'Avg Sal' )
   ,COUNT(COALESCE(salary_amount,0)) (TITLE 'ZIN #') (FORMAT 'ZZZ9')
   ,AVG(COALESCE(salary_amount,0))  (TITLE   'ZIN Avg' )
   ,COUNT(NULLIF(salary_amount,0)) (TITLE 'NIZ #') (FORMAT 'ZZZ9' )
   ,AVG(NULLIF(salary_amount,0))     (TITLE 'NIZ Avg' )
FROM    employee
GROUP BY       1
ORDER BY       1;
```

Lab 22_1

```
DATABASE Customer_Service;
```

```
SELECT sc.year_of_calendar AS "year"
  ,sc.quarter_of_year AS "Quarter"
  ,ds.itemid
  ,SUM(ds.sales)
FROM sys_calendar.calendar sc
    ,daily_sales ds
WHERE sc.calendar_date = ds.salesdate
AND   sc.quarter_of_year = 3
AND   sc.year_of_calendar BETWEEN 1997 AND 1998
AND ds.itemid = 10
GROUP BY 1,2,3
ORDER BY 1,2;
```

Lab 22_2

```
CREATE TABLE day_of_week
(numeric_day INTEGER
,char_day CHAR(9)
)
UNIQUE PRIMARY INDEX (numeric_day);

INSERT INTO day_of_week VALUES (1, 'Sunday');
INSERT INTO day_of_week VALUES (2, 'Monday');
INSERT INTO day_of_week VALUES (3, 'Tuesday');
INSERT INTO day_of_week VALUES (4, 'Wednesday');
INSERT INTO day_of_week VALUES (5, 'Thursday');
INSERT INTO day_of_week VALUES (6, 'Friday');
INSERT INTO day_of_week VALUES (7, 'Saturday');

SELECT dw.char_day "Day of//  Week"
FROM sys_calendar.calendar sc
```

```
      ,day_of_week       dw
WHERE sc.calendar_date = 580320
AND   sc.day_of_week = dw.numeric_day
;
```

Lab 22_3

```
SELECT dw.char_day "Day of//  Week"
   ,temp.avgsal
FROM day_of_week dw
    ,(sel sc.day_of_week
   ,AVG(ds.sales)
   FROM daily_sales ds, sys_calendar.calendar sc
   WHERE sc.calendar_date =  ds.salesdate
   HAVING avgsal > 300
   GROUP BY 1)
   temp(dayofweek,avgsal)
WHERE temp.dayofweek = dw.numeric_day
;
```

Lab 22_4

```
SELECT dw.char_day "Day of//  Week"
 ,SUM(ds.sales) avgsal
 FROM    daily_sales ds
   ,sys_calendar.calendar sc
   ,day_of_week dw
 WHERE sc.calendar_date =  ds.salesdate
 AND  sc.day_of_week = dw.numeric_day
 GROUP BY 1, dw.numeric_day
```

```
 ORDER BY dw.numeric_day
;
```

Lab 23_1

```
SELECT salesdate
      ,CAST(itemid AS BYTEINT) (FORMAT '99') AS "item"
   ,sales
   ,CSUM(sales, salesdate)AS "Csum"
   ,MSUM(sales, 3, salesdate) AS "Msum"
   ,MDIFF(sales, 3, salesdate) AS "Mdiff"
   ,MAVG(sales, 3, salesdate) AS "Mavg"
FROM daily_sales
WHERE salesdate BETWEEN 980101 AND 980228;
```

Lab 23_2

```
SELECT department_number
   ,budget_amount
   ,RANK(budget_amount)AS Rank
FROM  department
QUALIFY RANK(budget_amount) <=3;
```

Lab 23_3

```
SELECT deptno,
   salamt, RANK(salamt)AS Ranked
FROM    (SELECT department_number, SUM(salary_amount)
FROM    employee GROUP BY 1) temp(deptno,salamt)
```

```
QUALIFY RANK(salamt) <= 3;
```

Lab 23_4

```
SELECT employee_number
   ,salary_amount
   ,QUANTILE (100, salary_amount)  AS "Quantile"
FROM employee
QUALIFY QUANTILE(100, salary_amount) > 79;
```

Lab 23_5

```
SELECT employee_number
   ,salary_amount
   ,QUANTILE(100, salary_amount) AS "Quantile"
FROM employee
QUALIFY QUANTILE(100, salary_amount) < 21
ORDER BY 2 DESC;
```

Lab 23_6

```
SELECT itemid
      ,CAST(salesdate AS CHAR(10)) AS chardate
   ,sales
   ,MLINREG(sales, 14, chardate) AS "Mlinreg"
FROM jan_sales js
WHERE itemid = 10
AND salesdate BETWEEN 980101 AND 980114;
```

Lab 23_7

创建表：

```
CREATE TABLE username.empsamp
        ( empno INTEGER,
        deptno INTEGER,
        job INTEGER,
        sampid BYTEINT)
UNIQUE PRIMARY INDEX ( empno );
```

装入采样数据，并查询新表的内容：

```
INSERT into empsamp
SELECT employee_number
        ,department_number
        ,job_code
        ,sampleid
FROM employee sample .33, .33, .33;

SELECT * FROM empsamp ORDER BY sampid, empno;
```

Lab 23_8

```
INSERT INTO empsamp
SELECT employee_number
        ,department_number
        ,job_code
        ,SAMPLEID
FROM    employee SAMPLE 15, 15, 15;

SELECT * FROM empsamp ORDER BY sampid, empno;
```

Lab 23_9

```
SELECT DISTINCT(dept)
FROM (SELECT department_number
   FROM employee
   QUALIFY quantile(100, salary_amount) > 79)
   temp (dept);
```

Lab 24_1

创建表，装入数据：

```
SHOW TABLE customer_service.employee;
DATABASE sqlxxxx;
```

将表名改为'sqlxxxx.emp1'，提交CREATE TABLE语句。

```
INSERT INTO emp1 SELECT * FROM Customer_Service.employee;
SHOW TABLE customer_service.department
```

将表名改为'sqlxxxx.dept1'，提交CREATE TABLE语句。

```
INSERT INTO SQL00.dept1  SELECT * FROM Customer_Service.department;
```

创建触发器：

```
   CREATE TRIGGER .NullForKey
   BEFORE DELETE on emp1
   REFERENCING OLD_TABLE AS oldtable
   FOR EACH STATEMENT
   (UPDATE dept1 SET manager_employee_number = NULL
   WHERE manager_employee_number = oldtable.employee_number;);
```

Lab 24_2

```
DELETE FROM emp1 WHERE employee_number = 801;
```

```
SELECT employee_number FROM emp1 WHERE employee_number = 801;

SELECT manager_employee_number AS mgr
    ,department_number        AS dept
FROM dept1
WHERE department_number = 100;
```

Lab 24_3

创建日志表：

```
CREATE TABLE exceed_10_pcent
   (empnum     INT
   ,sal_date   DATE
   ,oldsal     DEC(10,2)
   ,newsal     DEC(10,2)
   ,percent_of_budget DEC(4,1)
   );
```

创建触发器：

```
CREATE TRIGGER AvgSalTrig
   AFTER UPDATE OF (salary_amount) ON emp1
REFERENCING OLD_TABLE AS oldtable
      NEW_TABLE AS newtable
FOR EACH STATEMENT
(INSERT INTO exceed_10_pcent
SELECT o.employee_number,DATE,o.salary_amount,
n.salary_amount,NULL FROM oldtable o, newtable n
WHERE n.salary_amount  >
   (SELECT AVG(budget_amount) * .10 FROM dept1)
```

```
AND o.employee_number = n.employee_number;
);
```

Lab 24_4

```
UPDATE emp1 SET salary_amount = salary_amount * 1.03
WHERE employee_number = 1015;

SELECT * FROM exceed_10_pcent;
```

Lab 24_5

```
REPLACE TRIGGER AvgSalTrig
   AFTER UPDATE OF (salary_amount) ON emp1
REFERENCING OLD_TABLE AS oldtable
        NEW_TABLE AS newtable
FOR EACH STATEMENT
(INSERT INTO exceed_10_pcent
SELECT o.employee_number,DATE,o.salary_amount, n.salary_amount,NULL
FROM oldtable o, newtable n
WHERE n.salary_amount * 1.10 >
   (SELECT AVG(budget_amount) FROM dept1 )
AND o.employee_number = n.employee_number;
UPDATE exceed_10_pcent FROM dept1 d
SET percent_of_budget = (newsal/d.budget_amount) * 100
WHERE empnum = oldtable.employee_number
AND oldtable.department_number = newtable.department_number;
);
```

Lab 24_6

```
DELETE FROM exceed_10_pcent;
UPDATE emp1 SET salary_amount = salary_amount * 1.03
WHERE employee_number = 1015;

SELECT * FROM exceed_10_pcent;
```

Lab 25_1

```
DATABASE sqlxxxx;
CREATE GLOBAL TEMPORARY TABLE gt_deptsal, NO LOG
(deptno   SMALLINT
,avgsal   DEC(9,2)
,maxsal   DEC(9,2)
,minsal   DEC(9,2)
,sumsal   DEC(9,2)
,empcnt   SMALLINT)
ON COMMIT PRESERVE ROWS;
```

Lab 25_2

```
INSERT INTO gt_deptsal
SELECT  department_number
     ,avg(salary_amount)
      ,max(salaly_amount)
     ,min(salary_amount)
      ,sum(salary_amount)
     ,count(employee_number)
FROM Customer_Service.employee GROUP BY 1;
```

Lab 25_3

```
CREATE VOLATILE TABLE vt_emp_job_dept, NO LOG(empno  INT,
 deptname CHAR(15),
 job_desc CHAR(15))
UNIQUE PRIMARY INDEX (empno)
ON COMMIT PRESERVE ROWS;
```

Lab 25_4

```
INSERT INTO vt_emp_job_dept
   SEL employee_number
      ,department_name
      ,description
   FROM (Customer_Service.employee emp
            LEFT OUTER JOIN Customer_Service.department  dep
   ON dep.department_number = emp.department_number)
            LEFT OUTER JOIN Customer_Service.job
   ON job.job_code = emp.job_code;
```

Lab 25_5

```
SELECT deptname  AS CHAR(16)
   ,job_desc  AS CHAR(15)
FROM vt_emp_job_dept
            INNER JOIN Customer_Service.employee emp
ON emp.employee_number = empno
            INNER JOIN gt_deptsal
ON emp.department_number = deptno AND emp.salary_amount = maxsal
```

```
ORDER BY 1;
```

Lab 25_6

```
SELECT deptname  CHAR(15) ,job_desc  CHAR(15)
FROM    Customer_Service.employee emp
            INNER JOIN gt_deptsal
ON      emp.department_number = deptno
            INNER JOIN vt_emp_job_dept
ON      emp.employee_number = empno
QUALIFY RANK(sal ASC) <= 6
ORDER BY 1;
```

Lab 25_7

```
SELECT deptname  CHAR(15)
   ,job_desc  CHAR(15)
FROM Customer_Service.employee emp
            INNER JOIN gt_deptsal
ON emp.department_number = deptno
            INNER JOIN vt_emp_job_dept
ON emp.employee_number = empno
QUALIFY QUANTILE(100,sal) >= 75
;
```

Lab 25_8

```
SELECT deptname
   ,job_desc
      ,salary_amount       , QUANTILE(100, sal)
```

```
FROM Customer_Service.employee emp
        INNER JOIN gt_deptsal
ON dept = deptno
        INNER JOIN vt_emp_job_dept
ON emp.employee_number = empno
QUALIFY QUANTILE(100,sal) >= 75
;
```

Lab 25_9

```
SELECT department_name  CHAR(16)
   ,avgsal
   ,empcnt
FROM Customer_Service.department
        INNER JOIN gt_deptsal
ON department_number = deptno
AND empcnt < 6
QUALIFY QUANTILE (100, avgsal) >= 50;
```

Lab 26_1

(1)

```
SELECT cityname
,citypop
,citystate
,statepop
FROM city INNER JOIN state
ON city.citystate=state.stateid
ORDER BY statepop, citypop
;
```

(2)

```
CREATE JOIN INDEX sqlxxxx.citystateidx AS
SELECT (city.citystate,state.statepop)
,(city.cityname,city.citypop)
FROM city INNER JOIN state
ON city.citystate=state.stateid
;
```

(3)

```
EXPLAIN
SELECT cityname
  ,citypop
  ,citystate
  ,statepop
FROM city INNER JOIN state
ON city.citystate=state.stateid
ORDER BY statepop, citypop
;
```

使用了连接索引表sqlxxxx.citystateidx。

Lab 26_2

(1)

```
DROP JOIN INDEX sqlxxxx.citystateidx;
```

(2)

```
SELECT cityname
```

```
,citypop
,citystate
,statepop
FROM city INNER JOIN state
ON city.citystate=state.stateid
WHERE statepop between 1000000 and 3000000
ORDER BY statepop, citypop
;
```

(3)

```
CREATE JOIN INDEX sqlxxxx.citystateidx AS
SELECT
(city.citystate,state.statepop)
,(city.cityname,city.citypop)
FROM city INNER JOIN state
ON city.citystate=state.stateid
ORDER BY statepop
;
```

(4)

```
EXPLAIN
SELECT cityname
,citypop
,citystate
,statepop
FROM city INNER JOIN state
ON city.citystate=state.stateid
WHERE statepop BETWEEN 1000000 AND 3000000
```

ORDER BY statepop, citypop

;

使用了连接索引表sqlxxxx.citystateidx。

Lab 26_3

(1)

CREATE INDEX (citypop) ON citystateidx ORDER BY VALUES (citypop);

(2)

SELECT cityname

,citypop

,citystate

,statepop

FROM city INNER JOIN state

ON city.citystate=state.stateid

WHERE citypop BETWEEN 500000 AND 1000000

ORDER BY statepop, citypop

;

注：对于非常小的表，优化器忽略非唯一的次索引，而扫描整个索引表。

Lab 26_4

(1)

DATABASE Customer_Service;

CREATE JOIN INDEX sqlxxxx.dept_sals

AS SELECT department_number AS Deptno

,SUM(salary_amount)

```
AS Dept_Sal FROM employee
GROUP BY 1;
```

(2)

```
SELECT department_number AS DeptNm
 ,SUM(salary_amount) AS Dept_Sal
FROM employee
GROUP BY 1
ORDER BY 1;
```

(3)

```
EXPLAIN SELECT department_number AS DeptNm
 ,SUM(salary_amount) AS Dept_Sal
FROM employee
GROUP BY 1
ORDER BY 1;
```

使用了连接索引表sqlxxxx.dept_sals。

(4)

```
SELECT department_number AS DeptNm
  ,SUM(salary_amount) AS Dept_Sal
  ,AVG(salary_amount) AS Avg_Sal
FROM employee
GROUP BY 1
ORDER BY 1;
```

(5)

```
EXPLAIN
SELECT department_number AS DeptNm
 ,SUM(salary_amount) AS Dept_Sal
 ,AVG(salary_amount) AS Avg_Sal
FROM employee
GROUP BY 1
ORDER BY 1;
```

使用了连接索引表sqlxxxx.dept_sals。

Lab 27_1

```
DATABASE sqlxxxx;
CREATE TABLE emp_sal(empno UNIQUE NOT NULL,lastnm,weekly_sal) AS
(SELECT employee_number (SMALLINT)
 ,last_name (CHAR(10))
 ,salary_amount/52
FROM Customer_Service.employee
WHERE department_number = 403)
WITH DATA;

SHOW TABLE emp_sal;

SELECT * FROM emp_sal ORDER BY 1;
```

Lab 27_2

```
CREATE TABLE emp_max_sals(dept, maxsal TITLE 'Max//Sal')AS
(SELECT department_number, MAX(salary_amount)
FROM Customer_Service.employee
```

```
GROUP BY 1)
WITH DATA;

SHOW TABLE emp_max_sals;

SELECT * FROM emp_max_sals ORDER BY 1;
```

Lab 27_3

```
CREATE TABLE emp_age (empno,deptname,age) AS
 (SELECT e.employee_number
 ,d.department_name
 ,(DATE - birthdate)YEAR
FROM Customer_Service.employee e
INNER JOIN Customer_Service.department d
ON e.department_number = d.department_number)
WITH DATA;

SHOW TABLE emp_age;

SELECT * FROM emp_age
WHERE age > (INTERVAL '50' YEAR) ORDER BY 1;
```

Lab 28_1

```
DATABASE sqlxxxx;
CREATE PROCEDURE lab_sp_1 (IN a_in INTEGER, IN b_in DEC(5,2), OUT
c_out DEC(5,2))
BEGIN
 SET c_out = a_in + b_in;
```

```
END;

CALL lab_sp_1(3,4,c_out);
```

Lab 28_2

```
CREATE PROCEDURE lab_sp_2 (IN x_in INTEGER, OUT y_out DEC(5,2))
BEGIN
 SET y_out = x_in * x_in;
END;

CALL lab_sp_2(5,y_out);
```

Lab 28_3

```
REPLACE PROCEDURE lab_sp_1 (IN a_in INTEGER, IN b_in DEC(5,2), OUT
c_out DEC(5,2))
BEGIN
 SET c_out = a_in + b_in;
 CALL lab_sp_2(:c_out,:c_out);
END;

CALL lab_sp_1(3,4,c_out);
```

Lab 28_4

```
CREATE PROCEDURE lab_sp_3 (IN low INTEGER, IN high INTEGER, OUT
empcnt INTEGER)
BEGIN
 DECLARE x INT;
```

```
 DECLARE cnt INT DEFAULT 0;
 DECLARE deptnum SMALLINT;
 DECLARE lastnm CHAR(10);
 SET x = low;
Label_One:
WHILE x < high + 1
 DO SELECT department_number , last_name
  INTO :deptnum, :lastnm
  FROM Customer_Service.employee
  WHERE employee_number = :x;
  IF ACTIVITY_COUNT = 1 THEN
  SET cnt = cnt + 1;
  PRINT 'DEPT NUM:', deptnum,'LAST NAME:', lastnm;
  END IF;
 SET x = x + 1;
 END WHILE Label_One;
 SET empcnt = cnt;
END;

CALL lab_sp_3(1020,1030,empcnt);
cnsterm 5
```

Lab 28_5

```
REPLACE PROCEDURE lab_sp_4 (IN low INTEGER, IN high INTEGER, OUT
empcnt INTEGER)
BEGIN
 FOR empvar AS cur1 CURSOR FOR
  SELECT department_number AS deptnum, last_name AS lastnm
  FROM Customer_Service.employee
  WHERE employee_number BETWEEN :low AND :high
```

```
 ORDER BY employee_number
DO
 PRINT 'DEPT NUM:', empvar.deptnum;
 PRINT 'LAST NAME:', empvar.lastnm;
END FOR;
SET empcnt = ACTIVITY_COUNT;
END;

CALL lab_sp_4(1020,1030,empcnt);
cnsterm 5
```

Lab 29_1

(2)

```
REPLACE PROCEDURE lab_sp_5 (IN empnum INTEGER, OUT lastnm
CHAR(20))
BEGIN
 DECLARE EXIT HANDLER FOR SQLEXCEPTION
 /* Handles SQLSTATE '42000' 'Object not found condition' and any others*/

 BEGIN
 IF SQLSTATE = '42000' THEN
 PRINT 'Warning - Object Not Found';
 ELSE
 PRINT 'Unplanned Exception';
 END IF;
 INSERT INTO error_log VALUES (:SQLCODE, :SQLSTATE,
CURRENT_TIME);
 END;
```

```
DECLARE CONTINUE HANDLER FOR SQLSTATE '02000'
/* Handles 'no rows returned' condition */

BEGIN
PRINT 'Warning - No Rows Found';
INSERT INTO error_log VALUES (:SQLCODE, :SQLSTATE,
CURRENT_TIME);
END;
 SELECT last_name INTO :lastnm FROM emp_view
 WHERE employee_number = :empnum;
 PRINT 'Successful Completion - ACTIVITY COUNT = ', ACTIVITY_COUNT;
END;
```

(3)

```
CALL lab_sp_5 (1008,lastnm);

.os cnsterm 5
```

(4)

```
CALL lab_sp_5 (1099,lastnm);

SELECT * FROM error_log;

.os cnsterm 5
```

(5)

```
DROP view emp_view;
CALL lab_sp_5 (1008,lastnm);
```

```
SELECT * FROM error_log;

.os cnsterm 5
```

Lab 29_2

(3)

```
REPLACE PROCEDURE lab_sp_6 (IN empnum INTEGER, IN sal_increase
DEC(3,2),
 INOUT old_sal DEC(9,2), OUT new_sal DEC(9,2))
BEGIN
 DECLARE new_salary DEC(9,2);
 DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
  BEGIN
  INSERT INTO error_log VALUES (:SQLCODE, :SQLSTATE,
CURRENT_TIME);
  PRINT 'SQLSTATE:' ,SQLSTATE;
  END;
 BT;
 SELECT salary_amount INTO :old_sal FROM emp_lab
  WHERE employee_number = :empnum;
 SET new_sal = old_sal * (1 + sal_increase);
 UPDATE emp_lab SET salary_amount = :new_sal
  WHERE employee_number = :empnum;
 PRINT 'ACTIVITY COUNT = ', ACTIVITY_COUNT;
 IF ACTIVITY_COUNT > 0 THEN
 INSERT INTO salary_log VALUES(:empnum, :old_sal, :new_sal);
 END IF;
 ET;
 PRINT 'Procedure Completion';
```

```
END;
```

(4)

```
CALL lab_sp_6(1010,.10,old_sal,new_sal);

SELECT * FROM salary_log;

.os cnsterm 5
```

(5)

```
CALL lab_sp_6(1000,.80,old_sal,new_sal);

.os cnsterm 5
```