

# 数据库技术-SQL基础

赵亚伟

[zhaoyw@ucas.ac.cn](mailto:zhaoyw@ucas.ac.cn)

中国科学院大学大数据分析技术实验室

2017.11.26

# 目录

- SQL产生背景
- 数据定义 (DDL)
- SQL查询基本结构 (DML)
- 集合运算
- 聚集函数
- 空值
- 嵌套子查询
- 复杂查询
- 视图
- 数据库修改
- 连接关系\*\*

# SQL产生的意义

- 关系代数提供了一种简洁的形式化的关系查询及操作方法，抽象层次高，但是并不友好。
- 商业化的数据库系统需要一种更加友好易用的查询语言。
- **SQL**是最具市场影响力的关系数据库查询语言。

# SQL产生背景

- 最早版本是由IBM的San Jose研究室（现在的Almaden研究中心）在System R中提出的，当时叫Sequel，后来名字演变为SQL（Structure Query Language，结构化查询语言），“SQL”读作“sequel”，也可以按单个字母的读音读作S—Q—L
- 版本：
  - ANSI和ISO：SQL-86，1986，100p，最初版，很简单，只有select，1987年ISO也通过了这一标准
  - SQL-89：1989，<300p，比较完善，DDL,DML,完整性
  - SQL-92：1992，600p，比较成功
  - SQL-99(SQL-3)：1999，<2000p，复杂
  - SQL-2003：是目前最高版本
- 关于Ingres的QUEL

# SQL的主要特点

- 综合统一：SQL语言集数据定义语言DDL、数据操纵语言DML、数据控制语言DCL的功能于一体，语言风格统一，可以独立完成数据库生命周期中的全部活动
- 高度非过程化：使用了关系代数和关系演算结构的组合，非过程化程度高
- 集合操作：SQL语言采用集合操作方式，不仅查找结果可以是元组的集合，而且一次插入、删除、更新操作的对象也可以是元组的集合。
- 两种使用方式：自含式和嵌入式
- 语言简洁易学易用：接近自然语言

# 自含式和嵌入式

- SQL语言既是自含式语言，又是嵌入式语言。
- 作为自含式语言，它能够独立地用于联机交互的使用方式，用户可以在终端键盘上直接键入SQL命令对数据库进行操作；
- 作为嵌入式语言，SQL语句能够嵌入到高级语言(例如C，COBOL，FORTRAN，PL / 1)程序中，供程序员设计程序时使用。
- 在两种不同的使用方式下，SQL语言的语法结构基本上是一致的。这种以统一的语法结构提供两种不同的使用方式的做法，提供了极大的灵活性与方便性。

# SQL的构成

- 1. **DDL**: 关系模式的定义、删除及修改
- 2. **DML**: 基于关系代数的查询、插入、删除及修改元组
- 3. **View**: 与DDL类似
- 4. **Integrity**
- 5. **Security**
- 6. **Embedded SQL and Dynamic SQL**
- 7. **Transaction Control**

# SQL的学习

## ■ 关于SQL的学习

- **问题**。针对某个问题或项目使用SQL
- **手册**。准备一个SQL的手册，一个DBMS，一个集成开发环境（能够支持SQL）
- **数据**。构建一些基本表及视图等
- **实验**。在DBMS中SQL的操作过程及结果分析，在某个集成开发环境中实现在应用程序的SQL嵌入。实现方法、过程要有记录，要有结论，或发现系统漏洞，或效率对比，或方案对比等均可。



这里讨论关系代数中的功能在  
现实中是如何实现的——SQL

# 目录

- SQL产生背景
- 数据定义 (**DDL**)
- SQL查询基本结构 (**DML**)
- 集合运算
- 聚集函数
- 空值
- 嵌套子查询
- 复杂查询
- 视图
- 数据库修改
- 连接关系\*\*

# 数据定义

- SQL中的DDL除了可以定义一个关系，还可以定义关系中的信息，包括：
  - 每个关系的模式
  - 每个属性的值域
  - 完整性约束
  - 每个关系维持的索引集合
  - 每个关系的安全性和权限信息
  - 每个关系的物理存储结构

# SQL主要功能： DDL

## ■ 1. 域类型

**smallint:** 16 位的整数。

**integer:** 32 位的整数。

**decimal(p,s):** p 精确值和 s 大小的十进位整数，精确值p是指全部有几个数(digits)大小值，s是指小数点后有几位数。如果没有特别指定，则系统会设为 p=5; s=0 。

**float:** 32位的实数。

**double:** 64位的实数。

**char(n):** n长度的字串，n不能超过 254

**varchar(n):** 长度不固定且其最大长度为n的字串，n不能超过 4000。

**date:** 包含了年份、月份、日期。

**time:** 包含了小时、分钟、秒。

**timestamp:** 包含 年、月、日、时、分、秒、千分之一秒。

- 2. 模式定义
- 模式定义的基本格式:
  - **create table r (A1 D1, A2 D2, ..., An Dn,**  
    **<integrity-consteaint1>,**  
    **<integrity-consteaint2>,**  
    **...,**  
    **<integrity-consteaintk>)**
  - **integrity-consteaint**表示完整性约束

# 例子：表定义

```
create table student  
(name char(15) not null, /*列级别完整性约束*/  
student-id char(10),  
Degree-level char(15),  
primary key(student-id), //实体完整性约束  
check (degree-level in  
('bachelors','masters','doctorate'))) //值约束
```

## ■ 表删除及模式修改

–**drop table r** 和 **delete table r**

前者删除整个表信息,后者保留关系r的模式,但元组信息被全部删除

–**alter table r add A D**:向关系r中添加属性A,域为D

–**alter table r drop A**: 将关系r中的属性A删除掉

## ■ 索引

如果数据已存在, 创建索引:

```
CREATE [ UNIQUE ] [ CLUSTERED |  
NONCLUSTERED ] INDEX index_name  
ON { table | view } ( column [ ASC | DESC ] ,...n )
```

索引的主要功能是能够有效地提高查询效率

索引删除

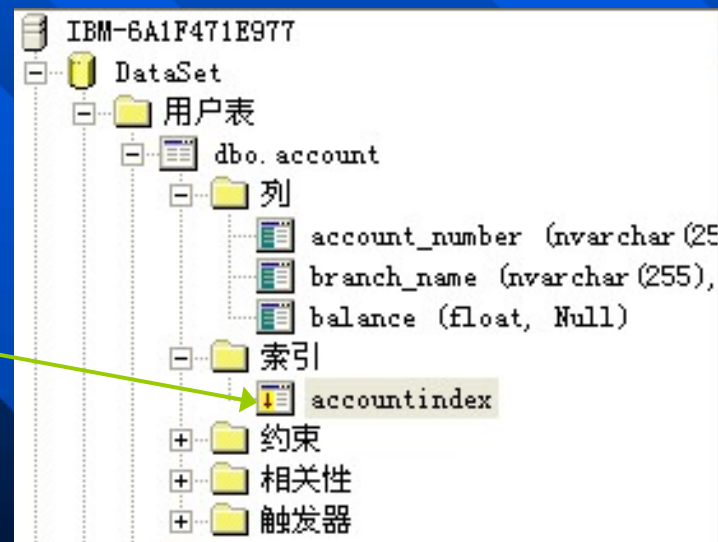
```
DROP INDEX 'table.index | view.index' [ ,...n ]
```

# 例子：索引

- 在表account的account\_number属性上建立唯一性降序索引

create unique index *accountindex*  
on *account* (*account\_number* DESC)

索引accountindex





## ■ Primary key:

- 表中经常有一个列或列的组合，其值能唯一地标识表中的每一行。这样的一列或多列称为表的主键（主码），通过它可强制表的实体完整性。当创建或更改表时可通过定义 PRIMARY KEY 约束来创建主键。
- 当为表指定 PRIMARY KEY 约束时，强制主码非空且唯一。
- 如果 PRIMARY KEY 约束定义在不止一列上，则一列中的值可以重复，但 PRIMARY KEY 约束定义中的所有列的组合的值必须唯一。
- 一个表只能有一个 PRIMARY KEY 约束
- 主码声明是可选的，指定更好

## SQLServer-在查询分析器中创建bank数据库以及相关关系

```
查询 — IBMSER.dbmis45.IBMSER\zhaoyawei — D:\教学工作\Database\Top

/*创建一个数据库bank*/
create database bank
use bank

/*创建表account*/
create table account
(account_number char(10) not null unique, /*完整性约束*/
 branch_name char(20),
 balance int,
 primary key (account_number)
)

/*创建表branch*/
create table branch
(branch_name char(20),
 branch_city char(20),
 assets int
)

/*创建表customer*/
create table customer
(customer_name char(20),
 customer_street char(20),
 customer_city char(10)
)

/*创建表depositor*/
create table depositor
(customer_name char(20),
 account_number char(10)
)

/*创建表loan*/
create table loan
/

命令已成功完成。
```

# SQLServer manager中查看创建的数据库和表

控制台根目录\Microsoft SQL Servers\SQL Server 组\ (local) (Windows NT)\数据库\bank\表

控制台根目录

- Microsoft SQL Servers
  - SQL Server 组
    - (local) (Windows NT)
      - 数据库
        - bank
          - 关系图
          - 表
          - 视图
          - 存储过程
          - 用户
          - 角色
          - 规则
          - 默认
          - 用户定义的数据类
          - 用户定义的函数
          - 全文目录
        - dbmis45
        - master
        - model
        - msdb
        - Northwind
        - pubs
        - tempdb
      - 数据转换服务
      - 管理
      - 复制
      - 安全性
      - 支持服务
      - Meta Data Services

表 26 个项目

名称	所有者	类型	创建日期
sysusers	dbo	系统	2000-8-6 1:29:12
sysstypes	dbo	系统	2000-8-6 1:29:12
sysreferences	dbo	系统	2000-8-6 1:29:12
sysprotects	dbo	系统	2000-8-6 1:29:12
sysproperties	dbo	系统	2000-8-6 1:29:12
syspermissions	dbo	系统	2000-8-6 1:29:12
sysobjects	dbo	系统	2000-8-6 1:29:12
sysmembers	dbo	系统	2000-8-6 1:29:12
sysindexkeys	dbo	系统	2000-8-6 1:29:12
sysindexes	dbo	系统	2000-8-6 1:29:12
sysfulltextnotify	dbo	系统	2000-8-6 1:29:12
sysfulltextcatalogs	dbo	系统	2000-8-6 1:29:12
sysforeignkeys	dbo	系统	2000-8-6 1:29:12
sysfiles1	dbo	系统	2000-8-6 1:29:12
sysfiles	dbo	系统	2000-8-6 1:29:12
sysfilegroups	dbo	系统	2000-8-6 1:29:12
sysdepends	dbo	系统	2000-8-6 1:29:12
syscomments	dbo	系统	2000-8-6 1:29:12
syscolumns	dbo	系统	2000-8-6 1:29:12
dtproperties	dbo	系统	2006-3-9 12:00:10
loan	dbo	用户	2006-3-9 12:42:47
depositor	dbo	用户	2006-3-9 12:42:47
customer	dbo	用户	2006-3-9 12:42:47
branch	dbo	用户	2006-3-9 12:42:47
borrower	dbo	用户	2006-3-9 12:42:47
account	dbo	用户	2006-3-9 12:42:47

## 系统表：数据库bank的相关信息

表 "sysfiles1" 中的数据，位置是 "bank" 中、 "(local..."

sysfiles1

☐ \* (所有列)  
☐ status  
☐ fileid  
☐ name  
☐ filename

列	别名	表	输出	排序类型	排序顺序	准
*			✓			

SELECT \*

	status	fileid	name	filename
▶	3	1	bank	C:\Program File
	49218	2	bank_log	C:\Program File
*				

## 系统表：数据库bank中的表的相关信息

表 "syscolumns" 中的数据，位置是 "bank" 中、 "(local..."

syscolumns

\* (所有列)

name

id

xtype

typestat

列	别名	表	输出	排序类型	排序顺序	准则
<pre>SELECT * FROM [dbo].[syscolumns]</pre>						
name	id	xtype	typestat	xusertype		
branch_name	1977058079	175	2	175		
balance	1977058079	56	0	56		
branch_name	2025058250	175	2	175		
branch_city	2025058250	175	2	175		

## 系统表：数据库bank中的索引信息

表 "sysindexes" 中的数据，位置是 "bank" 中、 "(lo...

sysindexes

☐ \* (所有列)  
☐ id  
☐ status  
☐ first  
☐ indid

列	别名	表	输出	排序类型	排序顺序
*			✓		

SELECT \*

keys	name	statblob	maxlen	rows
<Binary>	customer	<Binary>	8000	0
<Binary>	pk_dtproperties	<Binary>	8000	0
<Binary>	tdtproperties	<Binary>	8000	0



## 设置主键后的实体完整性错误提示信息



# 目录

- SQL产生背景
- 数据定义 (DDL)
- SQL查询基本结构 (DML)
- 集合运算
- 聚集函数
- 空值
- 嵌套子查询
- 复杂查询
- 视图
- 数据库修改
- 连接关系\*\*



# DML-查询结构

- 基本查询结构（基本查询运算，投影、选择、积）

**SELECT**  $A1, A2, \dots, An$

**FROM**  $r1, r2, \dots, rn$

**WHERE**  $P$

Select子句--- 投影( $\Pi$ )

From子句--- 广义笛卡尔积( $\times$ )

Where子句--- 选择( $\sigma$ )

$P$ 为谓词（条件表达式）

该查询语句等价于关系代数表达式:

$\Pi_{A1, A2, \dots, An}(\sigma_p(r1 \times r2 \times, \dots, \times rn))$

## 例子：基本查询

```
Select customer_name, borrower.loan_number, amount  
From borrower, loan  
Where borrower.loan_number=loan.loan_number
```

注意： `borrower.loan_number` 和 `loan.loan_number` 的写法与关系代数中的写法一样，使用前缀，避免混淆

# 更名运算-as

- SQL提供了为关系和属性重新命名的机制:

- 旧名 as 新名

- 例子:

```
select customer_name, b.loan_number as loan_id, amount  
from borrower as b, loan as l  
where b.loan_number = l.loan_number
```

更名前

更名后

# Tuple Variables 元组变量

- Tuple variables are defined in the from clause via the use of the as clause. 用 **as**, 引入关系名的 简称, 别名
- Find the customer names and their loan numbers for all customers having a loan at some branch. 找出在银行中贷款的所有客户的用户名、贷款号和贷款额

```
select customer-name, T.loan-number, S.amount  
from borrower as T, loan as S //引入别名或简称  
where T.loan-number = S.loan-number
```

- 用“关系名.属性名”隐含定义了元组变量
- 为什么称元组变量而不叫关系变量？比较的内容是元组

# String Operations

- SQL使用单引号标示字符串，常用的字符串运算符是**like**，表示模式匹配，模式描述：
  - **percent (%)**: 匹配任意子串
  - **underscore (\_)**: 匹配任意一个字符
- 所有街道含子串 “Main”的客户.  

```
select customer_name
from customer
where customer_street like '% Main%'
```
- SQL支持的其他字符串操作：
  - **concatenation (using “|”)** 串联
  - **upper()** 将字符串转为大写，**lower()** 转小写
  - 其他计算字符串长度、提取子串等等

# 排序与重复

- 在select中可以不使用索引而使用order by进行排序

Select \*

From *loan*

**Order by** *amount*

默认为升序，可以用desc表示降序，用asc表示升序

Select \*

From loan

**Order by** amount **desc**, loan\_number **asc**

- 重复

使用 DISTINCT 消除重复项，与关系代数的投影一致了

Select **DISTINCT** branch\_name

From loan

# The from Clause: 笛卡积

- Cartesian product 广义笛卡积

- Find the Cartesian product *borrower x loan*

**select \***

**from *borrower, loan***

- Find the name, loan number and loan amount of all customers having a loan at the Perryridge branch.

找出在该支行有贷款的账户信息

**select *customer-name, borrower.loan-number, amount***  
**from *borrower, loan***

**where *borrower.loan-number = loan.loan-number* and**  
***branch-name = 'Perryridge'***

- 保证正确性的 连接条件

# From子句：广义笛卡尔积

查询 — IBMUSER.bank.IBMUSER\zhaoyawei — D:\教学工作\...

```
/*use bank*/  
select count(*) from borrower  
select count(*) from loan  
select * from borrower, loan  
select count(*) from borrower, loan
```

	(无列名)	
1	8	
	(无列名)	
1	7	
	重复属性	
	customer_name	loan_number loan_number branch_name
1	Adams	L-16 L-11 Round Hill
2	Curry	L-93 L-11 Round Hill
3	Hayes	L-15 L-11 Round Hill
4	Jackson	L-14 L-11 Round Hill
	(无列名)	
1	56	



**Select的子句很多，功能强，一般的DBMS都能够支持**

**详细的功能及用法可以参考SQL标准或相应DBMS的帮助文档**

# 目录

- SQL产生背景
- 数据定义 (DDL)
- SQL查询基本结构 (DML)
- 集合运算
- 聚集函数
- 空值
- 嵌套子查询
- 复杂查询
- 视图
- 数据库修改
- 连接关系\*\*

# DML-集合运算

- 集合运算

- (1) **union**(并): 对应关系代数中的 $\cup$

- (2) **intersect**(交): 对应关系代数中的 $\cap$

- (3) **except**(差): 对应关系代数中的 $-$

一般的DBMS中支持三种操作，也有一些DBMS支持**union**，而**intersect**和**except**则通过其他方式实现。

- 参加集合运算的关系必须是**相容的**，即必须含有相同的属性集

# 集合运算的一般格式

(Select loan\_number  
from loan)

**Union** (or **intersect** or **except**)

(Select loan\_number  
from borrower)

# 例子：SQL-并

关系代数表达式：

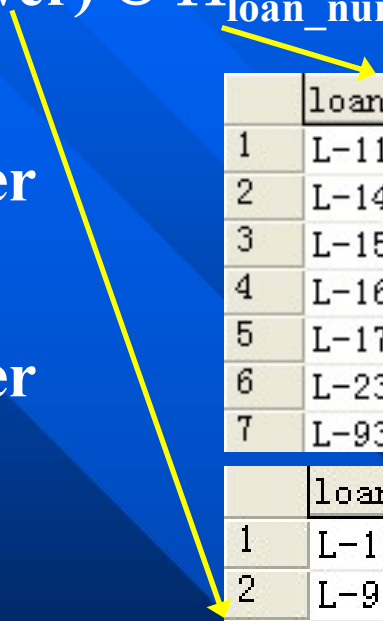
$$\Pi_{\text{loan\_number}}(\text{borrower}) \cup \Pi_{\text{loan\_number}}(\text{loan})$$

SQL语句：

(Select loan\_number  
from loan)

Union

(Select loan\_number  
from borrower)



	loan_number
1	L-11
2	L-14
3	L-15
4	L-16
5	L-17
6	L-23
7	L-93

	loan_number
1	L-16
2	L-93
3	L-15
4	L-14
5	L-17
6	L-11
7	L-23
8	L-17

	loan_number
1	L-16
2	L-93
3	L-15
4	L-14
5	L-17
6	L-11
7	L-23
8	L-17

# 例子：SQL-交

找出所有既有帐户又有贷款的客户  
关系代数表达式：

$$\Pi_{\text{customer\_name}}(\text{depositor}) \cap \Pi_{\text{customer\_name}}(\text{borrower})$$

SQL语句实现：

```
select * from depositer  
intersect  
select * from borrower
```

	ID	customer_name	loan_number
1	01	adams	L-16
2	02	curry	L-93
3	03	hayes	L-15
4	04	johnson	L-14
5	05	jones	L-17
6	06	smith	L-11
7	07	smith	L-23
8	08	williams	L-17

	ID	customer_name	account_number
1	01	hayes	A-102
2	02	johnson	A-101
3	03	johnson	A-201
4	04	jones	A-217
5	05	lindsay	A-222
6	06	smith	A-215
7	07	turner	A-305

	customer_name
1	hayes
2	johnson
3	jones
4	smith

- 上面的例子也可以通过测试是否为空（后面讲）求得交运算结果

```
select distinct customer_name
from depositor
where exists(      //元组不为空为true
  select customer_name
  from borrower
  where depositor.customer_name=borrower.customer_name)
```

# 例子：SQL-差

关系代数表达式：

$\Pi_{\text{customer\_name}}(\text{depositor}) - \Pi_{\text{customer\_name}}(\text{borrower})$

SQL语句实现：

(select customer\_name from depositor)  
except  
(select customer\_name from borrower)

	customer_name
1	lindsay
2	turner

	customer_name
1	hayes
2	johnson
3	johnson
4	jones
5	lindsay
6	smith
7	turner

	customer_name
1	adams
2	curry
3	hayes
4	johnson
5	jones
6	smith
7	smith
8	williams



# 其他实现方法

也可以采用以下语句实现

```
select distinct customer_name  
from depositor  
where customer_name not in (  
select customer_name  
from borrower)
```

也可以通过测试是否为空求得差运算结果

```
select distinct customer_name  
from depositor  
where not exists(  
select customer_name  
from borrower  
where depositor.customer_name=borrower.customer_name)
```

# 目录

- SQL产生背景
- 数据定义 (DDL)
- SQL查询基本结构 (DML)
- 集合运算
- 聚集函数
- 空值
- 嵌套子查询
- 复杂查询
- 视图
- 数据库修改
- 连接关系\*\*

# DML-聚集运算

- 聚集运算是SQL中的一个重要功能，在数据仓库的构建及查询中有重要作用
- 通过聚集函数实现，SQL预定义的聚集函数有：
  - Avg: 平均值
  - Min: 最小值
  - Max: 最大值
  - Sum: 总和
  - Count: 计数
- 常用子句: **group by**用于分组
- 除了**count(\*)**外所有的聚集函数都忽略输入的集合中的空值。**count(\*)** 计算 元组个数，含空值的元组也计数

# Aggregate Functions

- Find the average account balance at the Perryridge branch. 找出 Perryridge 支行的帐户余额的平均值

```
select avg (balance)
      from account
     where branch_name = 'Perryridge'
```

review: AS 子句可用  
来更改结果集列名或  
为导出列指定名称

Find the number of tuples in the *customer* relation. 计算个数

```
select count (*)
      from customer
```

Find the number of depositors in the bank. 计算无重复存款人个数

```
select count (distinct customer_name)
      from depositor
```

# Aggregate Functions – Group By

- Find the number of depositors for each branch. 每个支行的储户数量

```
select branch_name, count (distinct customer_name)
from depositor, account
where depositor.account_number = account.account_number
group by branch_name
```

**Note:** Attributes in select clause outside of aggregate functions must appear in group by list. 除聚集函数外的select子句的属性必出现在 group by list中

# Aggregate Functions – Having Clause

- Find the names of all branches where the average account balance is more than \$1,200. 帐户平均余额大于1200的

```
select branch_name, avg (balance)  
from account  
group by branch_name  
having avg (balance) > 1200
```

Note: predicates in the **having clause** are applied **after** the formation of **groups** whereas predicates in the **where clause** are applied before forming groups

# 聚集运算应用 (难)

- 问题：某零售商在做商品促销时，需要对促销商品进行评估，如某地区年龄在30岁以下、收入为中等水平的居民能购买某类商品的可能性有多大？

# 已知的历史信息-关系DCustomer

ID	AGE	INCOME	STUDENT	CREDIT_RAT	BAYS_COMPU
1	<30	High	T	Fair	F
2	<30	High	F	Excellent	F
3	30-40	High	F	Fair	T
4	>40	Medium	F	Fair	T
5	>40	Low	T	Fair	T
6	>40	Low	T	Excellent	F
7	30-40	Low	T	Excellent	T
8	<30	Medium	F	Fair	F
9	<30	High	T	Fair	T
10	>40	Medium	T	Fair	T
11	<30	Medium	T	Excellent	T
12	30-40	Medium	F	Excellent	T
13	30-40	Medium	T	Fair	T
14	>40	Low	F	Excellent	F



# 算法

- 这里只考察属性AGE和INCOME采用贝叶斯模型

$$\begin{aligned} \text{Confidence } (A \rightarrow C_i) &= P(C_i | A) = \frac{P(C_i \wedge A)}{P(A)} \\ &= \frac{(\prod_{k=1}^m P(a_k | C_i)) P(C_i)}{\sum_{j=1}^n P(A | C_j) P(C_j)} \\ &= \frac{(\prod_{k=1}^m P(a_k | C_i)) P(C_i)}{\sum_{j=1}^n (\prod_{k=1}^m P(a_k | C_j)) P(C_j)} \end{aligned}$$

# 实现过程

- 计算事前概率 $P(C_i)$ 
  - $P(\text{BAYS\_COMP}=\text{T})=9/14$
  - $P(\text{BAYS\_COMP}=\text{F})=5/14$
- 计算条件概率 $P(a_k|C_i)$ 
  - $P(\text{AGE}='<30'|\text{BAYS\_COMP}=\text{T})=2/9$
  - $P(\text{AGE}='<30'|\text{BAYS\_COMP}=\text{F})=3/5$
  - $P(\text{INCOME}='Medium'|\text{BAYS\_COMP}=\text{T})=5/9$
  - $P(\text{INCOME}='Medium'|\text{BAYS\_COMP}=\text{F})=1/5$
- 计算 $\Pi(P(a_k|C_i))$ 
  - $P(A|\text{BAYS\_COMP}=\text{T})=2/9*5/9$
  - $P(A|\text{BAYS\_COMP}=\text{F})=3/5*1/5$
- 计算 $\Pi(P(a_k|C_i))P(C_i)$ 
  - $P(A|\text{BAYS\_COMP}=\text{T}) * P(\text{BAYS\_COMP}=\text{T})=2/9*5/9*9/14=0.079$
  - $P(A|\text{BAYS\_COMP}=\text{F}) * P(\text{BAYS\_COMP}=\text{F})=3/5*1/5*5/14=0.042$
- 结论：年龄小于30岁中等收入的顾客购买电脑的可能性要大于不买的可能性，相差近一倍

# 聚集函数的实现方法

```
Select count(*) as TCOUNT from Dcustomer  
/*TCOUNT=14*/
```

```
Select count(*) as PTCOUNT from Dcustomer  
where BAYS_COMPU='T'  
/*PTCOUNT=9*/
```

```
Select count(*) as PFCOUNT from Dcustomer  
where BAYS_COMPU='F'  
/*PFCOUNT=5*/
```

```
Select count(*) as ACOUNT from Dcustomer  
where BAYS_COMPU='T' and age='<30'  
/*ACOUNT=2*/
```

...

# 一个查询结果



# 目录

- SQL产生背景
- 数据定义 (DDL)
- SQL查询基本结构 (DML)
- 集合运算
- 聚集函数
- 空值
- 嵌套子查询
- 复杂查询
- 视图
- 数据库修改
- 连接关系\*\*

# 关于空值null

- SQL允许使用null值表示关于某属性值的信息缺失
- 例子:

Example: Find all loan number which appear in the *loan* relation with null values for *amount*. 在loan 关系中amount为空值的贷款号

```
select loan_number  
from loan  
where amount is null
```

is null用于检测空值, is not null检测非空值

# 算术运算中的空值

- *null* signifies an unknown value or that a value does not exist.
- The result of any arithmetic expression involving *null* is *null*
  - Example:  $5 + \text{null}$  returns *null*
- However, aggregate functions simply ignore nulls, 聚集运算中会忽略一些空值，参考前面的内容

# 比较、逻辑运算中的空值

- Any comparison with *null* returns **unknown**
  - Example:  $5 < null$  or  $null <> null$  or  $null = null$
- Three-valued logic using the truth value *unknown*:
  - OR:  $(unknown \text{ or } true) = true$ ,  
 $(unknown \text{ or } false) = unknown$   
 $(unknown \text{ or } unknown) = unknown$
  - AND:  $(true \text{ and } unknown) = unknown$ ,  
 $(false \text{ and } unknown) = false$ ,  
 $(unknown \text{ and } unknown) = unknown$
  - NOT:  $(\text{not } unknown) = unknown$
  - “*P* is **unknown**” evaluates to true if predicate *P* evaluates to *unknown*
- Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*



# 目录

- SQL产生背景
- 数据定义 (DDL)
- SQL查询基本结构 (DML)
- 集合运算
- 聚集函数
- 空值
- 嵌套子查询
- 复杂查询
- 视图
- 数据库修改
- 连接关系\*\*

# DML-复杂查询

## ■ 嵌套子查询

- 子查询自身可以包括一个或多个子查询。一个语句中可以嵌套任意数量的子查询。
- 子查询的使用是为了对集合的**成员资格**、**集合的比较**以及**集合的基数**进行检查
- 嵌套子查询还可以作为用户构思查询时的中间结果，阶段性成果，**便于思考、化大为小，化繁为简**

## ■ 派生关系

- 允许**from**子句使用子查询表达式。

## ■ With子句

- 提供一种临时视图的方法，**create view**建立视图被一直保存，不是临时视图

# 嵌套子查询-集合成员资格

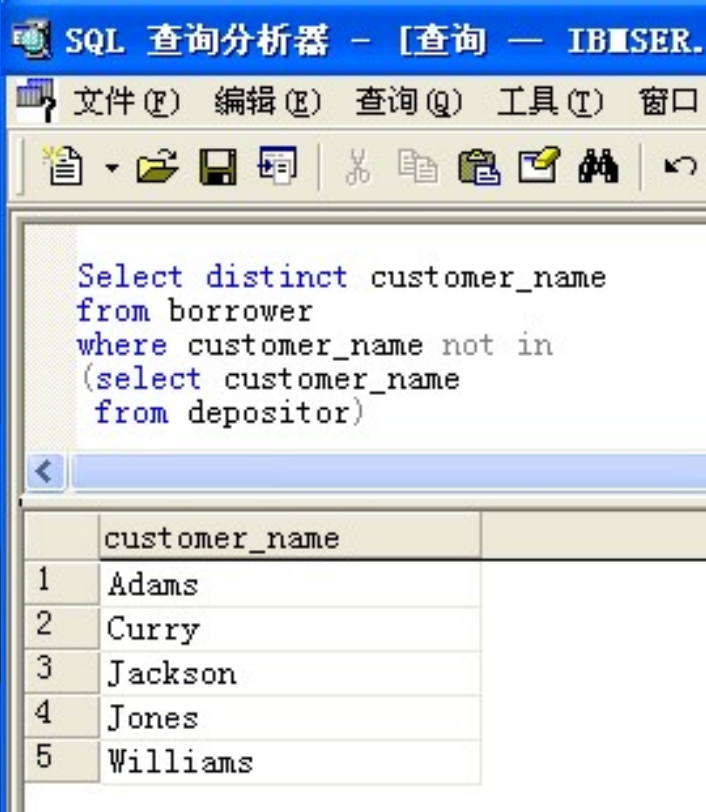
- 查询银行中有贷款的客户但无账户的客户

```
Select distinct customer_name  
from borrower  
where customer_name not in  
(select customer_name  
from depositor)  
/*子查询*/
```

一个  
属性

- 也可以考察一组属性
- 除了not in外还可以使用比较运算符，如>some表示至少比一个大，其他还有

=,!=,>,>=,!>,<,<=,!<  
SOME或ANY 等



The screenshot shows the 'SQL 查询分析器' (SQL Query Analyzer) window for the 'IBMSER.' database. The query entered is: `Select distinct customer_name from borrower where customer_name not in (select customer_name from depositor)`. The results are displayed in a table with one column, 'customer\_name', and five rows of data.

	customer_name
1	Adams
2	Curry
3	Jackson
4	Jones
5	Williams

# 嵌套子查询-集合比较

- Find all branches that have greater assets than some branch located in Brooklyn. 找出总资产至少比 **Brooklyn** 某一家支行高的支行的名字（**Brooklyn** 最高的和比 **Brooklyn** 某一家支行高的）

```
select distinct T.branch-name //不
from branch as T, branch as S
where T.assets > S.assets and S.br
```

	branch_name	branch_city	assets
1	brighton	brooklyn	7100000.0
2	downtown	brooklyn	9000000.0
3	mianus	horseneck	400000.0
4	north town	rye	3700000.0
5	perryridge	horseneck	1700000.0
6	pownal	bennington	300000.0
7	redwood	palo alto	2100000.0
8	round hill	horseneck	8000000.0

- Same query using > some clause

```
select branch-name
from branch
where assets > some //some相当于
(select assets
from branch
where branch-city = 'Brooklyn')
```

	branch_name	
1	downtown	
2	round hill	

# Some R 意义为 Some in R

- $F <\text{comp}> \text{some } r \Leftrightarrow \exists t \in r \text{ s.t. } (F <\text{comp}> t)$

Where  $<\text{comp}>$  can be:  $<, \leq, >, =, \neq$       谓词对某些值为真

$(5 < \text{some } \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}) = \text{true}$   
(read: 5 < some tuple in the relation)

$(5 < \text{some } \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline \end{array}) = \text{false}$

$(5 = \text{some } \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline \end{array}) = \text{true}$

$(5 \neq \text{some } \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline \end{array}) = \text{true} \quad (\text{since } 0 \neq 5)$

$(= \text{some}) \equiv \text{in}$

However,  $(\neq \text{some}) \not\equiv \text{not in}$

# Example Query

- Find the names of all branches that have greater assets than all branches located in Brooklyn. 总资产比 Brooklyn 任意一家支行都多的支行名

```
select branch_name
from branch
where assets > all
    (select assets
     from branch
     where branch_city = 'Brooklyn')
```

# All R 意义为 all in R, 或 all of R

- $F <\text{comp}> \text{all } r \Leftrightarrow \forall t \in r (F <\text{comp}> t)$

$$(5 < \text{all} \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}) = \text{false}$$

$$(5 < \text{all} \begin{array}{|c|} \hline 6 \\ \hline 10 \\ \hline \end{array}) = \text{true}$$

$$(5 = \text{all} \begin{array}{|c|} \hline 4 \\ \hline 5 \\ \hline \end{array}) = \text{false}$$

$$(5 \neq \text{all} \begin{array}{|c|} \hline 4 \\ \hline 6 \\ \hline \end{array}) = \text{true (since } 5 \neq 4 \text{ and } 5 \neq 6)$$

$(\neq \text{all}) \equiv \text{not in}$

However,  $(= \text{all}) \neq \text{in}$



# 嵌套子查询-测试空关系

- SQL中测试一个子查询（集合）是否为空用**exists**结构，即指定一个子查询，**检测行的存在**。
- **exists**使用了一个子查询作为条件，只有当子查询**返回行**的时候这个条件才**为真**，如果子查询**不返回**任何的行条件就**为假**。**not exists**则相反。
- 有些空值**null**测试 **exists**也为真，如在SQL Server中，**select null** 有元组返回

```
select *  
from account  
where exists ( select null )
```

	account_number	branch_name	balance
1	A-101	downtown	500.0
2	A-102	perryridge	400.0
3	A-201	brighton	900.0
4	A-215	mianus	700.0
5	A-217	brighton	750.0
6	A-222	redwood	700.0
7	A-305	round hill	350.0



# 例子：测试空关系

- 找出银行既有帐户又有贷款的客户（实质是交运算）

```
select customer_name  
from borrower  
where exists (  
  select *  
  from depositor  
  where depositor.customer_name=borrower.customer_name)
```

可以一个元组一个元组的试

存在这样的元组



# Example Query (难)

- Find all customers who have an account at all branches located in Brooklyn. 参考前面的除法例子

```
select distinct S.customer_name
from depositor as S
where not exists (
    (select branch_name
     from branch
     where branch_city = 'Brooklyn')
    except
    (select R.branch_name
     from depositor as T, account as R
     where T.account_number = R.account_number and
           S.customer_name = T.customer_name ))
```

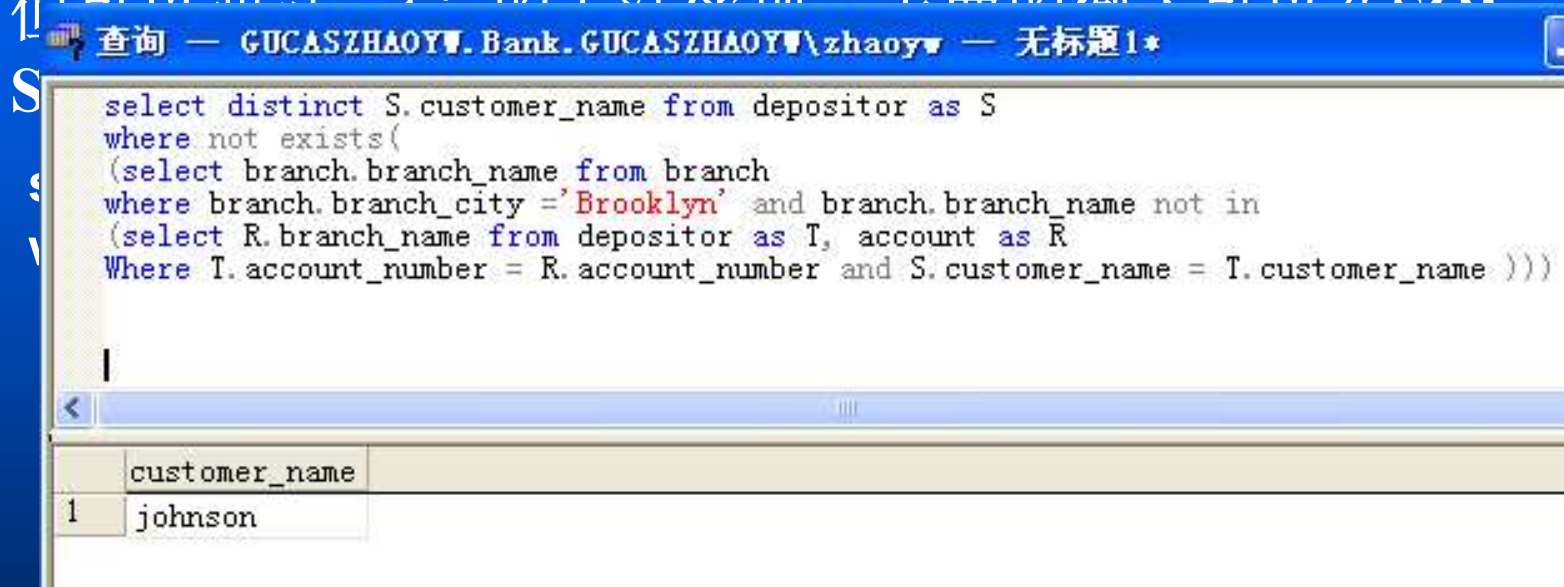
*S.customer\_name*='Johnson'  
时是空集, except, 差

Note that  $X - Y = \emptyset \Leftrightarrow X \subseteq Y$  注意是空集, 不是空值

Note: Cannot write this query using = all and its variants

# SQL Server 2000中实现上述例子

- 由于在SQL Server 2000不支持except(集合差, 减法), 但可以通过以下方法实现。上面的例子可以在SQL



The screenshot shows a SQL Server Enterprise Manager query window titled "查询 — GUCASZHAOYW.Bank.GUCASZHAOYW\zhaoyw — 无标题1\*". The query text is as follows:

```
select distinct S.customer_name from depositor as S
where not exists(
(select branch.branch_name from branch
where branch.branch_city = 'Brooklyn' and branch.branch_name not in
(select R.branch_name from depositor as T, account as R
Where T.account_number = R.account_number and S.customer_name = T.customer_name )))
```

Below the query window, the results are displayed in a table with two columns: an index and customer\_name.

	customer_name
1	johnson

*S.customer\_name = T.customer\_name )))*

## 嵌套子查询-测试是否存在重复元组

- The **unique** 谓词 construct tests whether a subquery has any duplicate tuples in its result.
- 如果子查询中没有重复元组则返回true，否则返回false
- 与exists类似，也有**unique**和**not unique**两种相反形式

# 例子: unique, not unique

- 测试是否存在重复元组(unique, not unique): Find all customers who have at most one account at the Perryridge branch. 找至多有一个账户的客户（一个元组一个元组的试）

```
select T.customer-name
  from depositor as T
 where unique (           //谓词，布尔函数
    select R.customer-name
  from account, depositor as R
  where T.customer-name = R.customer-name and
        R.account-number = account.account-number
  and account.branch-name = 'Perryridge')
```

# 目录

- SQL产生背景
- 数据定义 (DDL)
- SQL查询基本结构 (DML)
- 集合运算
- 聚集函数
- 空值
- 嵌套子查询
- 复杂查询
- 视图
- 数据库修改
- 连接关系\*\*

# 复杂查询

- 复杂查询通常很难或根本不能用一个SQL查询块或者几个SQL查询块并、交、差来解决，SQL提供了两种方式以简化复杂查询：派生关系和with子句，注意不能单独使用
- 派生关系：为子查询（派生的关系）重命名，as子句完成，一般产品都支持
- with子句：定义临时视图，SQL-99标准，一般的产品不支持
- 两种方式使得复杂查询模块化，增强了程序的可读性，这两种方式都可以找到等价的基本查询，但可能会复杂的多。

# 例子：派生关系-as子句

- 子查询产生关系result(branch\_name,avg\_balance)

```
(select branch_name,avg(balance)
from account
group by branch_name)
as branch_avg(branch_name,avg_balance)
```

- 派生关系查询

```
select branch_name, avg_balance
from (select branch_name,avg(balance)
from account
group by branch_name)
as branch_avg(branch_name,avg_balance)
```

派生关系branch\_avg  
的属性

Where **avg\_balance**>1200

找出平均账户余额大于1200元的支行的名字、平均账户余额



# 例子：With子句

产生临时视图

With *branch\_total*(*branch\_name*,*value*) as

    Select *branch\_name*,sum(*balance*)

    From *account*

    Group by *branch\_name*

产生临时视图

With *branch\_total\_avg*(*value*) as

    Select avg(*balance*)

    From *branch\_total*

    Select *branch\_name*

使用with子句产生的临时视图

    From *branch\_total*,*branch\_total\_avg*

    Where *branch\_total.value* >= *branch\_total\_avg.value*

# 目录

- SQL产生背景
- 数据定义 (DDL)
- SQL查询基本结构 (DML)
- 集合运算
- 聚集函数
- 空值
- 嵌套子查询
- 复杂查询
- 视图
- 数据库修改
- 连接关系\*\*

# SQL主要功能：视图

- DML中的语句一般都可用于视图操作
- 视图的数据来源于基本表，也可由视图创建视图
- 注意：Drop view是一个关联删除过程，即删除某视图，必须将由该视图产生的视图一同删除，否则这些视图会一直保存
- 注意：删除基本表时，其相关视图也应该进行删除或修改

# 视图定义

视图定义的格式:

**create view *v* as <query expression>**

## ■ 例子:

```
create view all_customer as
(select branch_name, customer_name
 from depositor, account
 where depositor.account_number = account.account_number)
union
(select branch_name, customer_name
 from borrower, loan
 where borrower.loan_number = loan.loan_number)
```

# 属性名处理

- 视图的属性名可以显示地指定:

```
create view branch_total_loan(branch_name, total_loan) as  
select branch_name, sum(amount)  
from loan  
group by branch_name
```

# 用其他视图定义视图

- 只要没有更新操作在视图上执行，视图可以出现在关系名可以出现的任何位置
- 因此，一个视图可能被用到定义另一个视图的表达式中
- 例如：

```
create view perryridge_customer as  
select customer_name  
from all_customer  
where branch_name='Perryridge'
```

其中， all\_customer本身也是视图

# 视图展开

- 如果一个视图是由其他视图定义的，那么，可以对视图进行展开，展开至表后为完全展开
- 视图展开揭示了视图的**来源或含义**
- 注意，如果视图不是由本身定义的，那么视图展开不是一个递归过程。相反，则是递归过程，如果是递归的，则通过所谓**不动点（fixed point）**来进行约束，避免迭代无休止地进行下去

# 例子：视图展开

```
select * from perryridge_customer  
where customer_name='John'
```

- 展开

```
select * from  
(select customer_name  
from all_customer  
where branch_name='Perryridge')  
where customer_name='John'
```

- **from** *all\_customer* 可以进一步展开



# 通过视图更新数据问题

- 对查询而言，视图是一个有用的工具，但如果通过视图来表达更新、插入和删除会导致严重的问题。
- 原因：用视图表达的数据库修改必须被翻译为对数据库逻辑模型中的实际关系的修改（**以虚务实**）
- 除了很少情况外，一般不允许通过视图对数据进行修改
- 有几种情况下视图是可更新的：
  - **from**子句中只有一个数据库关系
  - **select**子句中只包含关系的属性名，不包含表达式、聚集或**distinct**声明
  - 任何没有出现在**select**中的属性可以取空值
  - 查询中不含有**group by**子句和**having**子句（**having**子句在分组后才起作用，而**where**子句是可以更新的）

## e.g. Update of a View

- Create a view of all loan data in the *loan* relation, hiding the *amount* attribute 创建一个view，隐藏一些属性

```
create view loan_branch as  
select loan_number, branch_name  
from loan
```

- Add a new tuple to *branch\_loan* 通过视图更新

```
insert into branch_loan  
values ('L-37', 'Perryridge')
```

This insertion must be represented by the insertion of the tuple

(‘L-37’, ‘Perryridge’, *null*) 属性amount出现空值  
into the *loan* relation

# 两种解决办法

- 两种合理的解决办法：
  - 拒绝插入，并向用户返回一个错误信息
  - 向loan关系插入元组('L-37', 'Perryridge', *null*)
- SQL-99对于何时能够通过视图更新数据有一些更复杂的规则要求，这些规则允许很大一类视图上进行更新，但非常复杂

# 目录

- SQL产生背景
- 数据定义 (DDL)
- SQL查询基本结构 (DML)
- 集合运算
- 聚集函数
- 空值
- 嵌套子查询
- 复杂查询
- 视图
- 数据库修改
- 连接关系\*\*

# 数据库修改-删除

## ■ Delete

Delete from  $r$

Where  $P$

只对一个关系起作用，仅删除元组，模式将予以保留

## ■ Drop

Drop table  $r$

从数据库中删除一个关系，元组、模式均被删除

# Example- Deletion

- Delete all account tuples at the Perryridge branch 删除 Perryridge 支行的所有帐户

```
delete from account  
where branch_name = 'Perryridge'
```

- Delete all accounts at every branch located in the city 'Needham'. 删除所有位于Needham的支行的所有帐户

```
delete from account  
where branch_name in (select branch_name  
                        from branch  
                        where branch_city = 'Needham')
```

# 数据库修改-插入

- 可以直接插入元组，也可以插入经查询的元组集合
- **Insert**

```
Insert into account values('A-9732','Perryridge',1200)
```

/\*插入一个元组\*/

```
Insert into account
```

```
    select loan_number,branch_name,200
```

```
    from loan
```

```
    where branch_name='Perryridge'
```

/\*插入一个查询结果，一个集合\*/

# 数据库修改-更新

- 当希望不改变整个元组数量而只改变其中部分属性的值，则可以使用update语句。

- Update

**Update** account

**Set** balance=balance\*1.05

**Where** balance>(select avg(balance)  
from account)

根据查询结果进行更新



更新：所有存款余额大于平均数的账户增加5%的利息

- 通过视图修改数据库中数据（参考视图部分）



# Database – Updates

- Increase all accounts with balances over \$10,000 by 6%, all other accounts receive 5%.
  - Write two **update** statements:  

```
update account  
set balance = balance * 1.06  
where balance > 10000
```

```
update account  
set balance = balance * 1.05  
where balance ≤ 10000
```
  - The order is important
  - Can be done better using the **case** statement (next slide)
- 分两步完成，比较麻烦，有没有更好的方法？

# Case Statement for Conditional Updates

- 条件更新：满足条件的更新，多重条件
- 多步更新数据时，可以采用的条件更新
- 条件更新采用case语句，case的一般格式为：

```
case
  when pred 1 then result 1
  when pred 2 then result 2
  .....
  when pred n then result n
  else result 0
end
```

条件

结果

上面的都不满足的结果

# Example: 条件更新

- Same query as before: Increase all accounts with balances over \$10,000 by 6%, all other accounts receive 5%.  
超过10000加6%，其余的加5%

```
update account
  set balance = case
                    when balance <= 10000
                    then balance * 1.05
                    else balance * 1.06
                  end
```

- 与C语言类似, CASE 在多分支时特别方便.

# 事务

- 事务由查询和（或）更新语句序列组成。
- SQL标准规定当一个SQL语句被执行，就隐含地开始了一个事务，数据修改会被日志记录，其他可以不记录
- 事务定义需要说明起点与终点，如SQL99中：  
    **begin atomic**  
        ...加明显的起止符  
    **end**
- 在商业化产品中一般有对事务的定义说明，如在SQL Server中

```
BEGIN TRANSACTION M2  
UPDATE table2 ...  
SELECT * from table1  
COMMIT TRAN M2
```

# 事务结束

- 下列SQL语句之一结束一个事务
  - **commit work**: 提交当前事务，数据被永久保存，另一个新的事务自动开始
  - **rollback work**: 回滚当前事务，即撤销该事务中SQL语句对数据的更新，数据恢复到该事务的第一条语句之前的状态
- 事务保证了数据的一致性

# 目录

- SQL产生背景
- 数据定义 (DDL)
- SQL查询基本结构 (DML)
- 集合运算
- 聚集函数
- 空值
- 嵌套子查询
- 复杂查询
- 视图
- 数据库修改
- 连接关系\*\*

# SQL-连接关系\*\*

- SQL中的连接操作通常在from子句中完成
- SQL中的连接种类：
  - 内连接：对应关系代数中的 $\theta$ 连接（等值连接是 $\theta$ 连接的一种，自然连接是等值连接的一种， $\theta$ 不一定取“=”，取“!=”等其他运算符也可以，但等值连接，尤其是自然连接最为常用）
  - 外连接：对应关系代数中的扩展运算，外连接，但可以和自然连接组合，自然左外连接等
  - 何谓自然？重复属性自然应该消除，否则就不“自然”了

# 连接关系组成

- 每一种连接操作都包括：
  - **Join condition** – defines which tuples in the two relations match, and what attributes are present in the result of the join. 连接条件
  - **Join type** – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated. 连接类型（不能连接的如何处理）

## *Join types*

inner join  
left outer join  
right outer join  
full outer join

## *Join Conditions*

natural  
on <predicate>  
using  $(A_1, A_1, \dots, A_n)$



# 内连接 ( Inner join )

## ■ Inner join (内连接)

### – 等值连接 (inner join)

*loan* **Inner join** *borrower* **on** *loan.loan\_number =  
borrower.loan\_number* **as** *lb (loan\_number, branch,  
amount, cust, cust\_loan\_num)* 用**as**子句命名结果

### – 自然连接 (Natural inner join)

*loan* **natural inner join** *borrower*

### – **Natural**的含义是两个关系中元组匹配是显而易见的，也可以用于外连接，如**自然右外连接**，**natural right outer join**，**重复属性将被消除**

# 自然连接与等值连接

- **等值连接**: 返回连接的两个表中的所有列，但只返回在联接列中具有相等值的行
- **自然连接**: 由于重复相同的信息没有意义，因此可以通过更改选择列表消除两个相同列中的一个。其结果称为自然联接。
- **去掉重复列**的等值连接为自然连接

# SQL Server中的等值连接

**SELECT \***

**FROM** loan p **inner Join** account a

**ON** p.branch\_name = a.branch\_name /\*等值连接\*/

	loan_number	branch_name	amount	account_number	branch_name	balance
1	L-14	Downtown	1500	A-101	Downtown	500
2	L-17	Downtown	1000	A-101	Downtown	500
3	L-15	Perryridg...	1500	A-102	Perryridg...	400
4	L-16	Perryridg...	1300	A-102	Perryridg...	400
5	L-93	Mianus	500	A-215	Mianus	700
6	L-23	Redwood	2000	A-222	Redwood	700
7	L-11	Round Hil...	900	A-305	Round hil...	349

重复属性没有去除

# SQL Server中的自然连接

```
SELECT p.loan_number,p.amount,a.*  
FROM loan as p inner Join account a  
ON p.branch_name = a.branch_name /*自然连接，as可以省略*/
```

	loan_number	amount	account_number	branch_name	balance
1	L-14	1500	A-101	Downtown	500
2	L-17	1000	A-101	Downtown	500
3	L-15	1500	A-102	Perryridge	400
4	L-16	1300	A-102	Perryridge	400
5	L-93	500	A-215	Mianus	700
6	L-23	2000	A-222	Redwood	700
7	L-11	900	A-305	Round hill	349

重复属性已消除

# 外连接 ( outer join )

- 外联接会返回 **FROM** 子句中提到的至少一个表或视图的所有行，只要这些行符合搜索条件。将检索通过左外联接引用的左表的所有行，以及通过右外联接引用的右表的所有行。
- SQL中外连接表示
  - **Left outer join**(左外连接)
  - **Right outer join**(右外连接)
  - **Full outer join**(全外连接)
  - 所有外连接都可以和**natural**组合形成自然外连接

# 例子：外连接

*loan* **natural right outer join** *borrower* 自然右外连接

<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>	<i>customer-name</i>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	null	null	Hayes

*loan* **full outer join** *borrower* **using** (*loan-number*) 使用**using**  
相当于自然全外连接

<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>	<i>customer-name</i>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	null
L-155	null	null	Hayes

# 例子：外连接

*loan left outer join borrower on* //左外连, 保护左边, 无可连, 用空  
*loan.loan-number = borrower.loan-number*

<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>	<i>customer-name</i>	<i>loan-number</i>
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230
L-260	Perryridge	1700	<i>null</i>	<i>null</i>

重复属性没有去除

# SQL Server中的左外连接

**SELECT \* FROM account p left Outer Join loan a ON  
p.branch\_name = a.branch\_name /\*左外连接\*/**

	account_number	branch_name	balance	loan_number	branch_name	amount
1	100	汉武帝	1998	NULL	NULL	NULL
2	A-101	Downtown	500	L-14	Downtown...	1500
3	A-101	Downtown	500	L-17	Downtown...	1000
4	A-102	Perryridge	400	L-15	Perryrid...	1500
5	A-102	Perryridge	400	L-16	Perryrid...	1300
6	A-201	Brighton	900	NULL	NULL	NULL
7	A-215	Mianus	700	L-93	Mianus	500
8	A-217	Brighton	700	NULL	NULL	NULL



# SQL Server中的右外连接

```
SELECT * FROM loan p right Outer Join account a ON  
p.branch_name = a.branch_name /*右外连接*/
```

	loan_number	branch_name	amount	account_number	branch_name	balance
1	NULL	NULL	NULL	100	汉武帝	1998
2	L-14	Downtown	1500	A-101	Downtow...	500
3	L-17	Downtown	1000	A-101	Downtow...	500
4	L-15	Perryridg...	1500	A-102	Perryri...	400
5	L-16	Perryridg...	1300	A-102	Perryri...	400
6	NULL	NULL	NULL	A-201	Brighto...	900
7	L-93	Mianus	500	A-215	Mianus	700
8	NULL	NULL	NULL	A-217	Brighto...	700

# 全连接

- 完整外部联接(全连接)中两个表的所有行都将返回。
- SQL Server 2000中的全连接

SELECT \*

FROM loan p **full Outer Join** account a

ON p.branch\_name = a.branch\_name /\*全连接\*/

	loan_number	branch_name	amount	account_number	branch_name	balance
1	NULL	NULL	NULL	100	汉武帝	1998
2	L-14	Downtown	1500	A-101	Downtow...	500
3	L-17	Downtown	1000	A-101	Downtow...	500
4	L-15	Perryridg...	1500	A-102	Perryri...	400
5	L-16	Perryridg...	1300	A-102	Perryri...	400
6	NULL	NULL	NULL	A-201	Brighto...	900
7	L-93	Mianus	500	A-215	Mianus	700
8	NULL	NULL	NULL	A-217	Brighto...	700

# 其他连接

- **SQL-92**还提供另外两种类型的连接：
  - **Cross join**: 等价于没有连接条件的内连接，广义笛卡尔积
  - **Union join**: 等价于使用**false**条件的全外连接，即内连接结果  
为空
- 有些**SQL**标准在实际产品中不能得到完全支持，使用过程中可以参考产品的用户手册。

# 小结

- 商业数据库管理系统一般采用SQL标准，但语句做了一定的调整，如SQL Server中采用的SQL与Oracle中采用SQL是不完全相同的。
- SQL能够实现所有关系代数的操作并进行了扩展
- SQL的DDL和DML是核心，能够完成数据库操作的大部分功能