# 第三章 关系数据库标准语言SQL

# Structured Query Language

# 关于语言及SQL的几个问题

- 为什么需要语言?
- SQL是谁跟谁说的语言?
- 语法标准? 语汇集?
- 有没有方言?
- 在哪些场合可以说SQL?
- 有什么样的底稿? 谁负责翻译，口译还是笔译?

# 本章导读

## 教学内容

- SQL语言概述；
- 数据定义DDL，数据查询QL，数据更新DML和数据控制语言DCL的功能；
- 视图定义和使用；

## 要求掌握

- DDL，DML，DCL的语法结构；
- 会用SQL语言表达各种查询处理要求；
- 会使用视图

## 教学重点及难点

- 数据查询，子查询

# 本章内容

- SQL概述
- 数据定义
- 数据查询
- 数据更新
- 视图

# SQL概念

## 1、什么是SQL语言

SQL语言是结构化查询语言，Structured Query Language, 简称SQL。

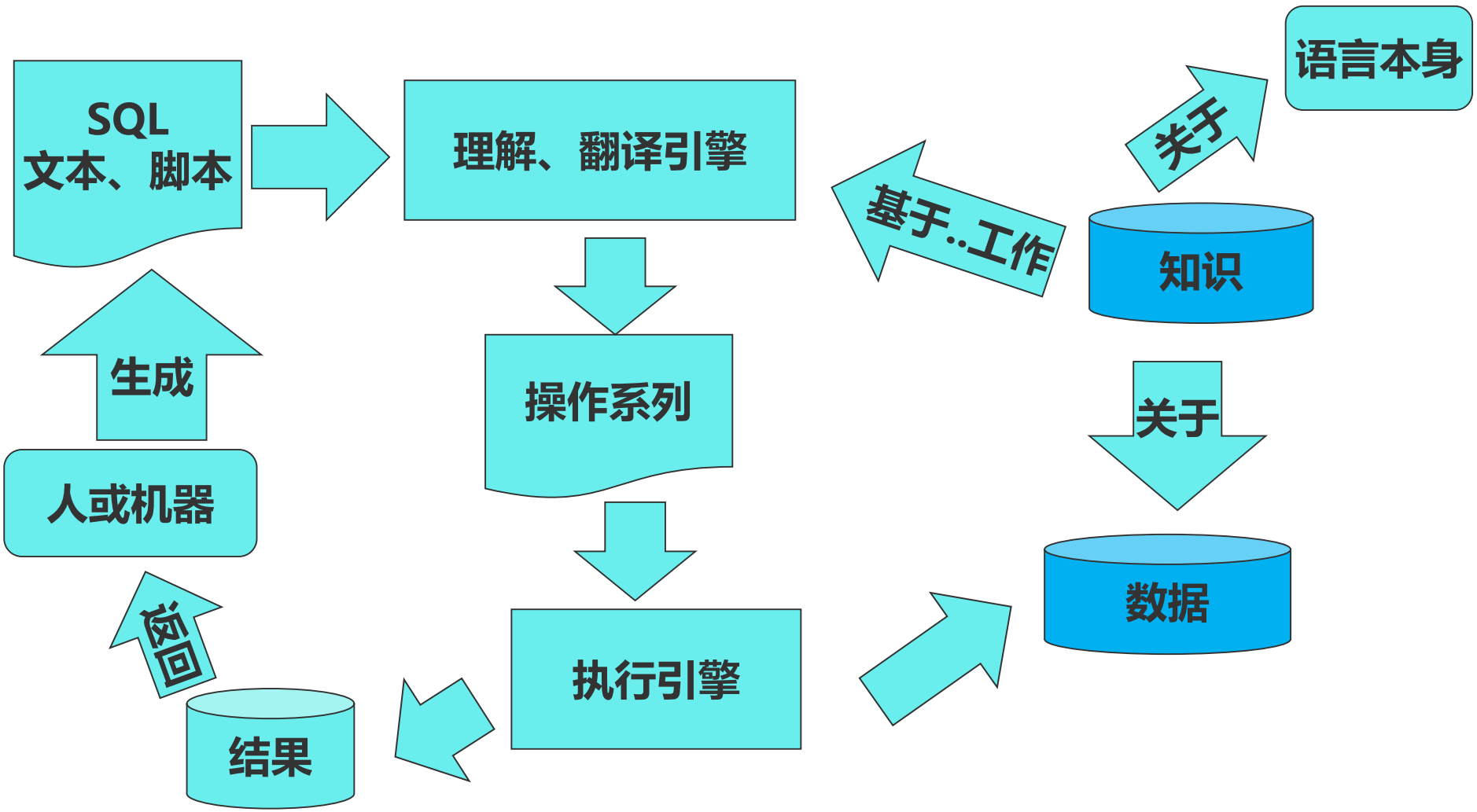
它是介于关系代数和关系演算之间的语言。

SQL is a domain-specific language used in programming and designed for managing data held in a relational database management system.

# Two main advantages of SQL

- In comparison to older read/write APIs like ISAM ( indexed sequential access method) or VSAM (Virtual storage access method), SQL offers two main advantages:
  - First, it introduced the concept of **accessing many records with one single command**
  - and second, it eliminates the need to specify ***how*** **to reach** a record, e.g.: **with or without an index**.

# 理解核心工作架构



# SQL概述—起源

- 1974年，IBM的Ray Boyce和Don Chamberlin将Codd关系数据库的12条准则的数学定义以简单的关键字语法表现出来，里程碑式地提出了SQL(Structured Query Language)语言。
- 1976年IBM E.F.Codd发表了一篇里程碑的论文“R系统: 数据库关系理论”，介绍了关系数据库理论和查询语言SQL。
- 1977年Oracle开发了第一个商用SQL关系数据库管理系统

# 2、SQL语言的发展

- 1974年，IBM的Boyce和Chamberlin为关系数据库原型系统System-R设计的一种查询语言；
- 1986年，ANSI公布第一个SQL标准：SQL86；
- 1987年，ISO通过SQL86标准；
- 1989年，ISO制定SQL89标准；
- 1990年，我国制定等同SQL89的国家标准；
- 1992年，ISO制定SQL92标准，即SQL2；
- 1999年，ANSI制定SQL3标准，即SQL3；
- 2003年，ANSI制定SQL2003标准。

SQL语言是关系数据库的标准语言

# 3. SQL的特点

## （1）综合统一

- 集数据定义语言（DDL），数据操纵语言（DML），数据控制语言（DCL）功能于一体。
  - definition, manipulation, and control
- 可以独立完成数据库生命周期中的全部活动：
  - 定义关系模式，插入数据，建立数据库；
  - 对数据库中的数据进行查询和更新；
  - 数据库重构和维护
  - 数据库安全性、完整性控制等
- 用户数据库投入运行后，可根据需要随时逐步修改模式，不影响数据的运行。
- 数据操作符统一

# 3. SQL的特点

## (2) 高度非过程化

- 非关系数据模型的数据操纵语言“面向过程”，必须制定存取路径
- SQL只要提出“做什么”—需求，无须了解存取路径—怎么做。
- 存取路径的选择以及SQL的操作过程由系统自动完成。

## (3) 面向集合的操作方式

- 非关系数据模型采用面向记录的操作方式，操作对象是一条记录
- SQL采用集合操作方式—关系代数
  - 操作对象、查找结果可以是元组的集合
  - 一次插入、删除、更新操作的对象可以是元组的集合

# 3、SQL的特点

## (4) 以同一种语法结构提供多种使用方式

- SQL是独立的语言

  能够独立地以联机交互的方式使用—online interactively

- SQL又是嵌入式语言--embedded

  SQL能够嵌入到高级语言（例如C，C++，Java）程序中，供程序员设计程序时使用

## (5) 语言简洁，易学易用

- SQL功能极强，完成核心功能只用了9个动词。

# SQL语言的动词

表 3.1　SQL 语言的动词

| SQL 功　能 | 动　　词 |
| --- | --- |
| 数 据 查 询 | SELECT |
| 数 据 定 义 | CREATE，DROP，ALTER |
| 数 据 操 纵 | INSERT，UPDATE，DELETE |
| 数 据 控 制 | GRANT，REVOKE |

**围绕数据生命周期，针对数据各种操作动作**

# SQL语言性质

(1)SQL语言是一种关系数据库语言

提供数据的定义、查询、更新和控制等功能。

(2)SQL语言不是一个应用程序开发语言，

只提供对数据库的操作能力，不能完成屏幕控制、菜单管理、报表生成等功能， 可成为应用开发语言的一部分.

(3)SQL语言不是一个DBMS

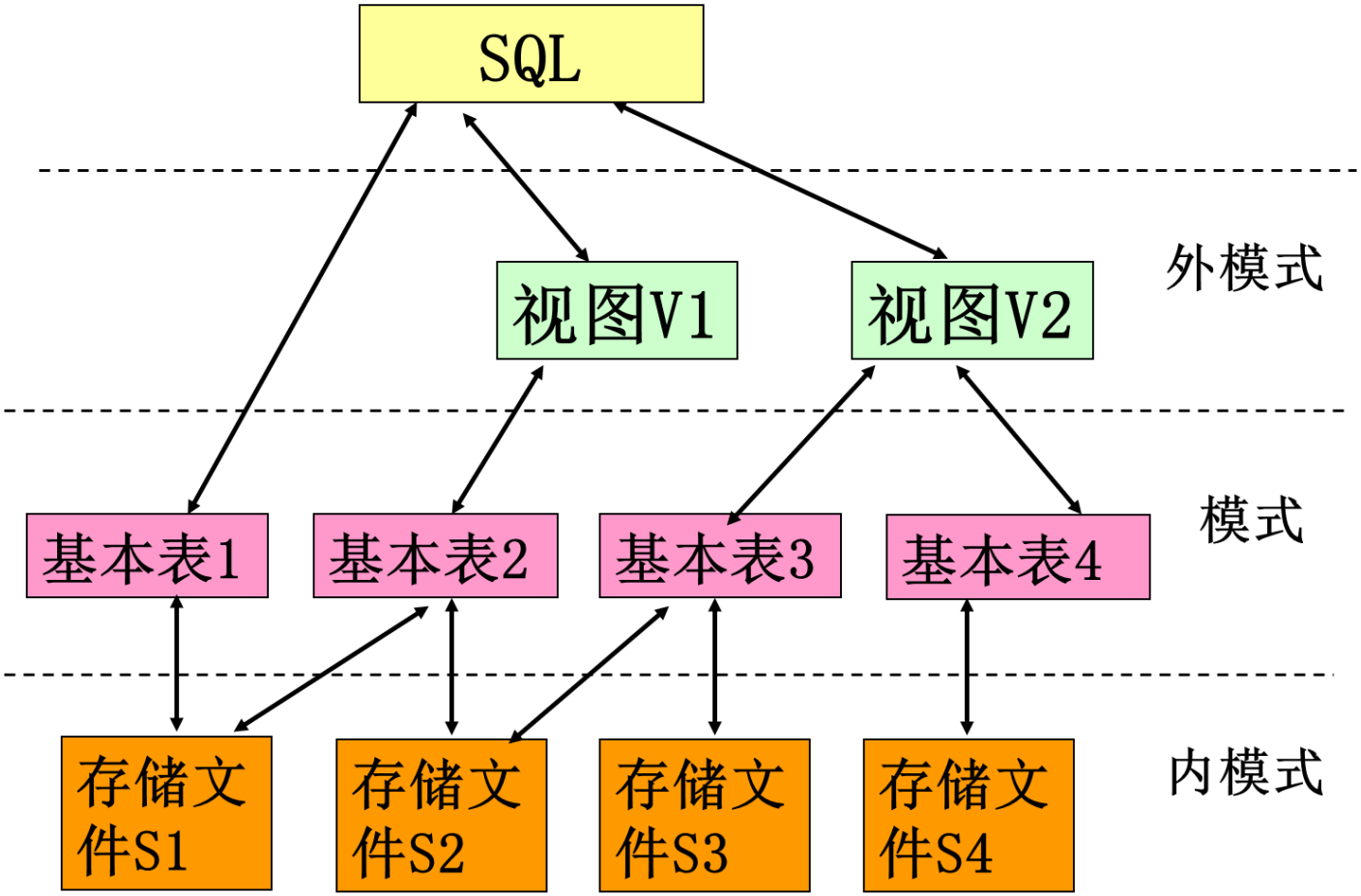
它是DBMS为用户提供的交互语言。

# 关系数据库的体系结构

SQL语言支持关系数据库三级模式结构，但术语与传统的关系模型术语不同。

| 在关系模型中 | 在SQL中 |
| --- | --- |
| 模式 | “基本表” |
| 内模式 | “存储文件” |
| 外模式 | “视图”或“基本表” |

# 关系数据库的体系结构




SQL对关系数据库模式的支持

# SQL语言的分类

SQL语言的命令通常分为四类

1）数据定义语言（ＤＤＬ）

2）查询语言（ＱＬ）

3）数据操纵语言（ＤＭＬ）

4）数据控制语言（ＤＣＬ）

# 数据定义语言

创建、修改或删除数据库中各种对象，包括SQL模式、基本表、视图、索引等。

**命令：**

（1）CREATE
- SCHEMA
- TABLE
- VIEW
- INDEX

（2）DROP
- SCHEMA
- TABLE
- VIEW
- INDEX

（3）ALTER TABLE

# 数据查询语言

按照指定的组合、条件表达式或排序检索已存在的数据库中数据，不改变数据库中数据。

**命令：**

SELECT … FROM … WHERE …

# 数据操纵与控制语言

## 数据操纵语言

对已经存在的数据库进行元组的插入、删除、修改等操作。

**命令：** INSERT、UPDATE、DELETE

## 数据控制语言

用来授予或收回访问数据库的某种特权、控制数据操纵事务的发生时间及效果、对数据库进行监视。

**命令：** GRANT、REVOKE、
COMMIT、ROLLBACK

# 数据定义

SQL的数据定义部分包括对

- SQL模式
- 基本表
- 视图
- 索引

创建和撤销操作。

# 一、SQL提供的基本数据类型

- 数值型
  - integer(int)：长整数。
  - smallint：短整数。
  - numeric（p, d）：定点数，共p位（不包括小数点），右边d位。
  - real：取决于机器精度的浮点数。
  - double precision：取决于机器精度的双精度浮点数。
  - float(n)：浮点数，精度至少为n位数字

# 一、 SQL提供的基本数据类型

- 字符型
  - char(n)固定长度为n的字符串。
  - varchar(n)最大长度为n的可变长字符串。
- 日期/时间型
  - date：日期（年、月、日），格式YYYY-MM-DD。
  - time：时间（小时、分、秒）,格式HH:MM:SS。

# 二、SQL模式的创建和删除

创建SQL模式即定义一个命名空间—工作环境

在这个空间中可以进一步**定义**该模式包含的**数据库对象**。

例如基本表、视图、索引等。

# SQL模式的创建和删除

## 1、创建模式：

CREATE SCHEMA <模式名> AUTHORIZATION <用户名>
如果没有指定<模式名>，<模式名>隐含为<用户名>

Create schema authorization wang

## 2、删除模式：

DROP SCHEMA <模式名> <CASCADE|RESTRICT>
Drop schema wang cascade

**CASCADE（级联）方式：**
删除模式的同时把模式中所有的数据库对象全部删除
**RESTRICT（约束）方式：**
只有当模式中没有任何下属对象时才能执行

# 建模式的同时嵌套创建其他对象

3、在CREATE SCHEMA中可以接受如下子句：

CREATE TABLE

CREATE VIEW

GRANT

CREATE SCHEMA <模式名>
AUTHORIZATION <用户名>
[<表定义子句>|<视图定义子句>|<授权定义子句>]

# 例

为用户ZHANG创建一个模式TEST，并在其中定义了一个表TAB1。

```
CREATE SCHEMA TEST
  AUTHORIZATION ZHANG
  CREATE TABLE TAB1
  (
    COL1 SMALLINT,
    COL2 INT,
    COL3 CHAR(20),
    COL4 NUMERIC(10, 3)
  );
```

# 三、基本表的创建、修改和撤销

## 1、创建表

别忘了关系模式的形式化表示：
R(U, D, DOM, F)

```
CREATE TABLE <基本表名>
  ( <列名 ><数据类型> [列级完整性约束条件],
   [<列名 ><数据类型>[列级完整性约束条件]],...
   [表级完整性约束条件] );
```

成功创建后，该表的模式信息将会被RDBMS存入数据字典中。有了登记信息，RDBMS将为针对该表的操作提供服务。

完整性约束条件也会被存入系统的数据字典中。如果完整性约束条件涉及到该表的多个属性列时，必须在表级定义该约束条件，否则既可以定义在列级，也可以定义在表级。

# 常见的完整性约束

- **主码**约束：PRIMARY KEY
- **唯一性**约束：UNIQUE
- **非空值**约束：NOT NULL
- **参照完整性**约束：FOREIGN KEY REFERENCES
- **值域检查**子句：CHECK

★保留字不能用作表名、列名等

# 例1：建立学生关系表

S(SNO, SN, AGE, SEX)

```
CREATE TABLE S
 (
    SNO CHAR(4) PRIMARY KEY,
    SN CHAR(8) NOT NULL,
    AGE SMALLINT,
    SEX CHAR(1)
 );
```

这个表对应的关系模式：

R(U, D, DOM, F)

各个成分分别是什么?

U = {SNO, SN, AGE, SEX}

D = {CHAR(4), CHAR(8), SMALLINT, CHAR(1)}

DOM = {(SNO, CHAR(4)), (SN, CHAR(8)), (AGE SMALLINT), (SEX, CHAR(1)) }

F = { (SNO→SN), …}

# 例2：建立课程关系表

C(CNO, CN, T)

```
CREATE TABLE C
(
    CNO CHAR(4) PRIMARY KEY,
    CN CHAR(8) UNIQUE,
    T CHAR(10)
);
```

# 例3：建立选课关系表

SC(SNO, CNO, G)

注意：表级的约束因为跟多个列有关，定义位置有要求

```
CREATE TABLE SC
(
        SNO CHAR(4),
        CNO  CHAR(4),
        G SMALLINT,
        PRIMARY KEY (SNO, CNO), /*表级约束只能放于此*/
        FOREIGN KEY(SNO) REFERENCES S(SNO),
        FOREIGN KEY(CNO) REFERENCES C(CNO),
        CHECK ((G IS NULL) OR (G BETWEEN 0 AND 100))
);
```

# 2. 基本表结构的修改

1）对表增加列：

ALTER TABLE <表名>
ADD <列名><数据类型>
[完整性约束],…;

例：ALTER TABLE S
ADD ADDR CHAR(20);

★ 不论原表中是否已存在数据，新增加的列一律为空值；

# 增加空列后

表S

| sno | sn | age | sex | addr |
|---|---|---|---|---|
| s1 | wang | 18 | F | null |
| s2 | li | 19 | M | null |
| ⋮ | ⋮ | ⋮ | ⋮ | null |

# 2）对表增加新的完整性约束条件

语法：ALTER TABLE <表名>
ADD <完整性约束>；

例：ALTER TABLE C
ADD UNIQUE (CN)；

# 3）删除指定的完整性约束条件

**语法：ALTER TABLE <表名>**
**DROP <完整性约束名>;**

**SQL不提供删除列的语法，Oracle中允许删除列。**

**通过实验回答以下问题：**

**（1）在SQL Server2008以上版本中，删除列是否允许?**

**（2）删除主键所在的列是否允许?**

# 4）修改原有列定义

语法：ALTER TABLE <表名>
ALTER COLUMN <列名> <数据类型>；

例：ALTER TABLE S
ALTER COLUMN age int;

# 3. 基本表的撤销

语法：DROP TABLE <表名>

[CASCADE|RESTRICT]

★RESTRICT：如存在依赖该表的对象（视图、索引、触发器、存储过程、约束等），此表不能被删除。

CASCADE：删除该表的同时，相关的依赖对象被同时删除。

例：DROP TABLE S CASCADE

当删除表时，表的数据、表上建立的索引和视图都自动被删除。

# DROP TABLE时，SQL2011 与 2个RDBMS的处理策略比较

| 序号 | 标准及主流数据库 / 依赖基本表的对象 | SQL2011 | | ORACLE 12c | | MS SQL SERVER 2012 |
|---|---|---|---|---|---|---|
| | | R | C | | C | |
| 1. | 索引 | 无规定 | | √ | √ | √ |
| 2. | 视图 | × | √ | √<br>保留 | √<br>保留 | √<br>保留 |
| 3. | DEFAULT，PRIMARY KEY，CHECK（只含该表的列）NOT NULL 等约束 | √ | √ | √ | √ | √ |
| 4. | Foreign Key | × | √ | × | √ | × |
| 5. | TRIGGER | × | √ | √ | √ | √ |
| 6. | 函数或存储过程 | × | √ | √<br>保留 | √<br>保留 | √<br>保留 |

R表示RESTRICT，C表示CASCADE

'×'表示不能删除基本表，'√'表示能删除基本表，‘保留’表示删除基本表后，还保留依赖对象

# 4. 模式与表间关系

1）每一个基本表都属于某一个模式

2）一个模式包含多个基本表

3）定义基本表所属模式


- 方法一：在表名中明显地给出模式名

```
/*模式名为 S_T*/
Create table “S_T”.Student（……）；
Create table “S_T”.Course（……）；
Create table “S_T”.SC（……）；
```

- 方法二：在创建模式语句中同时创建表

- 方法三：设置所属的模式

# 4. 模式与表间关系

4）创建基本表（其他数据库对象也一样）时，若没有指定模式，系统根据**搜索路径**来确定该对象所属的模式。

5）搜索路径包含**一组模式**的列表。RDBMS会使用该列表中**第一个存在的模式**作为数据库对象的模式名，若搜索路径中的模式名都不存在，系统给出错误。

6）搜索路径的当前默认值是：$user， PUBLIC；首先搜索与用户名相同的模式，如不存在，则使用PUBLIC模式

与操作系统的文件系统的搜索路径和当前路径的概念类似

# 4. 模式与表的关系

7） DBA用户可以设置搜索路径， 然后定义基本表

SET search_path TO “S_T”， PUBLIC;

Create table Student（……）；

结果建立了S_T.Student基本表。

RDBMS发现搜索路径中第一个模式名S_T存在，

就把该模式作为基本表Student所属的模式


# 研讨
# 看图说话

**关注功能**

**控制牵引**

**设施设备**

**套绳、鼻环**

**缰绳、手**

**效率成本**

**效率高**

**有成本**

**以上两个图前一页图的区别?**
**以上左右两个图有什么区别?**

**小规模vs大规模　　有管理vs无管理**

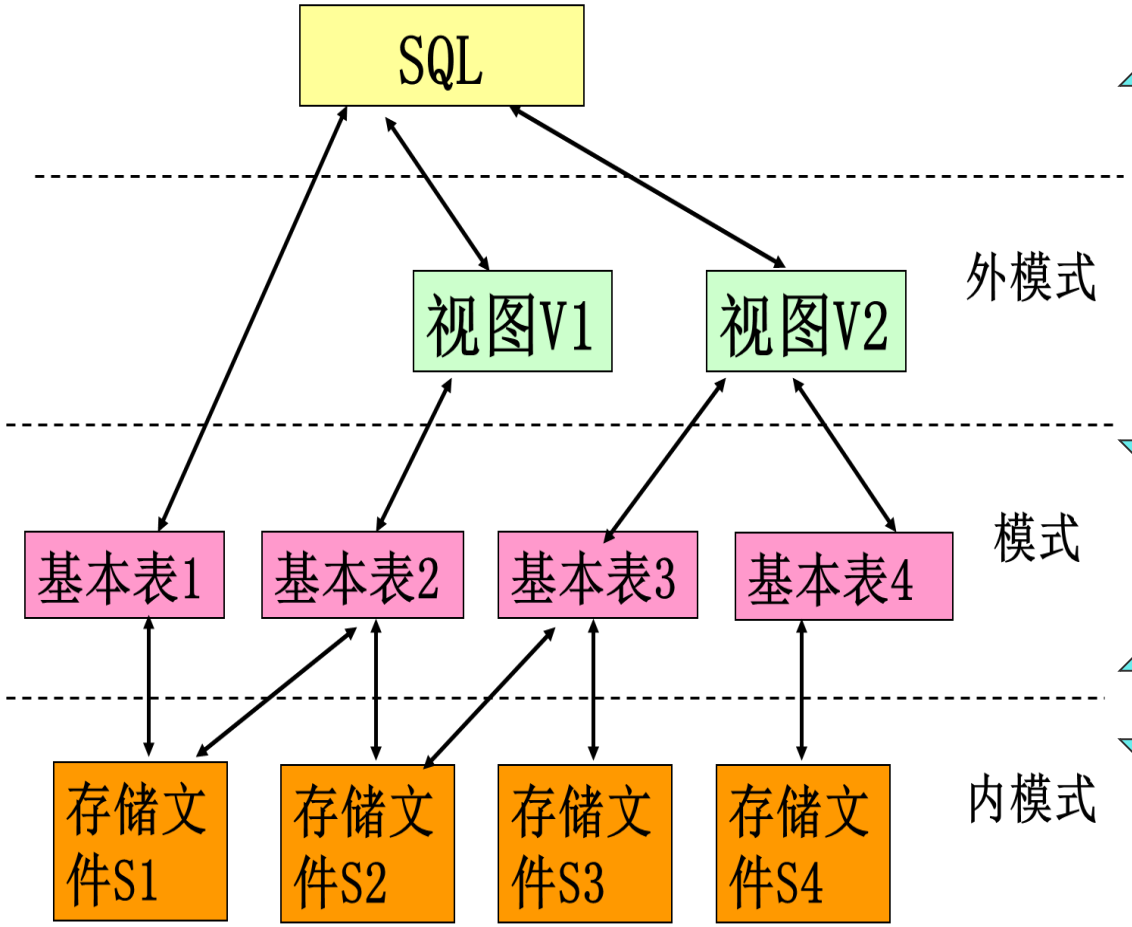
**还能或有必要用绳子吗?**


此图与前两个图的区别?
牛耳上挂着什么?
有什么用?
号相邻的有没有挨一起?
给定一个号，好不好找?
快速定位，还需要什么装备?
成本与效率?

# 图片背后的概念

- 管理主体：人
- 被管理对象：牛、牛群
- 业务：放牛、养牛
- 成本：时间、金钱、空间
- 效率：低、中、高
- 工具：绳、标签、电子标签
- 规模：一头、几头、一群、大规模、巨大规模
- 规模变化对工具的需求和管理的变化

# 关系型数据库的数据访问模式




SQL对关系数据库模式的支持

用户对关系型数据库的访问**不是基于指针，而是基于属性值提出查询**。用户**不需要关注**数据的具体存放位置。

**牺牲一定的效率，但是确保透明性。**

提供服务的关系型数据库管理系统**必须具有找到任何一条数据的能力**，基于分段、文件、分页、分块、指针等措施来**实现逻辑层到物理层的映射。** 通过**索引等机制来提升效率。**

# 四、索引及操作

- 索引的朴素解读
- 索+引
  - 索—绳索，便于操作，是工具
  - 引—效果，找到，不丢失，是目的
- 用绳索来引与不用绳索来引的区别?
- 索引的成本
  - 购买、建立、维护、维修、空间
- 索引的效率
  - 找得快、可控制、拴得住、不会丢失
- 无索引：
  - 控制不住、易丢失、丢了找得慢

# Database Index

- A database index is a **data structure** that improves **the speed** of data **retrieval operations** on a database table at the **cost** of **additional writes and storage space** to **maintain** the index data structure.
- Indexes are used to **quickly locate** data **without having to search every row** in a database table every time a database table is accessed.

# 建立、维护和使用索引

- 建立索引是加快查询速度的**有效手段**
- **建立索引**
  - DBA或表的属主（即建立表的人）根据需要建立
  - 有些DBMS自动建立以下列上的索引
    - PRIMARY KEY
    - UNIQUE
- **维护索引**
  - DBMS自动完成，用户不需要介入
- **使用索引**
  - DBMS自动选择**是否使用索引**以及**使用哪些索引**

# 索引的实现

- 索引是关系数据库的**内部实现技术**，属于**内模式**的范畴
- RDBMS中唯一索引一般采用**B+树、HASH索引**来实现
  - B+树索引具有动态平衡的优点
  - HASH索引具有查找速度快的特点
- 采用B+树，还是HASH索引则由具体的RDBMS来决定
- CREATE INDEX语句定义索引时，可以定义索引
  - **唯一索引**或**非唯一索引**
  - **非聚簇索引**或**聚簇索引**

# 1、索引的创建

语法：

```
CREATE [UNIQUE] [CLUSTER]
      INDEX <索引名>
      ON <基本表名>   /*谁的索引*/
           ( /*按什么索引*/
             <列名 1 >[ASC|DESC]
            [,<列名 2 >[ASC|DESC]]
            …
           )
```

UNIQUE—唯一，独特，每个索引键值只对一条记录

# Uinque Indexes

- Unique indexes are **indexes** that help **maintain data integrity** by ensuring that no two rows of data in a table have identical key values. –不同行关键码不允许存在重复值
- When you create a unique index for an existing table with data, values in the columns or expressions that comprise the index key are **checked for uniqueness**. If the table contains rows with **duplicate key values, the index creation process fails**.
- When a unique index is defined for a table, uniqueness is enforced whenever keys are added or changed within the index. This enforcement includes **insert, update, load, import, and set integrity**, to name a few. 各种操作受唯一索引的约束
- In addition to enforcing the uniqueness of data values, a unique index can also be used to **improve data retrieval performance during query processing**. 唯一索引的功用：提升查询效率

# Non-unique indexes

- Non-unique indexes are **not used to enforce constraints** on the tables with which they are associated. 非唯一不用于约束的实施
- Instead, non-unique indexes are used **solely to improve query performance** by maintaining a sorted order of data values that are **used frequently**. 只用于提升查询效率

# Non-clustered index

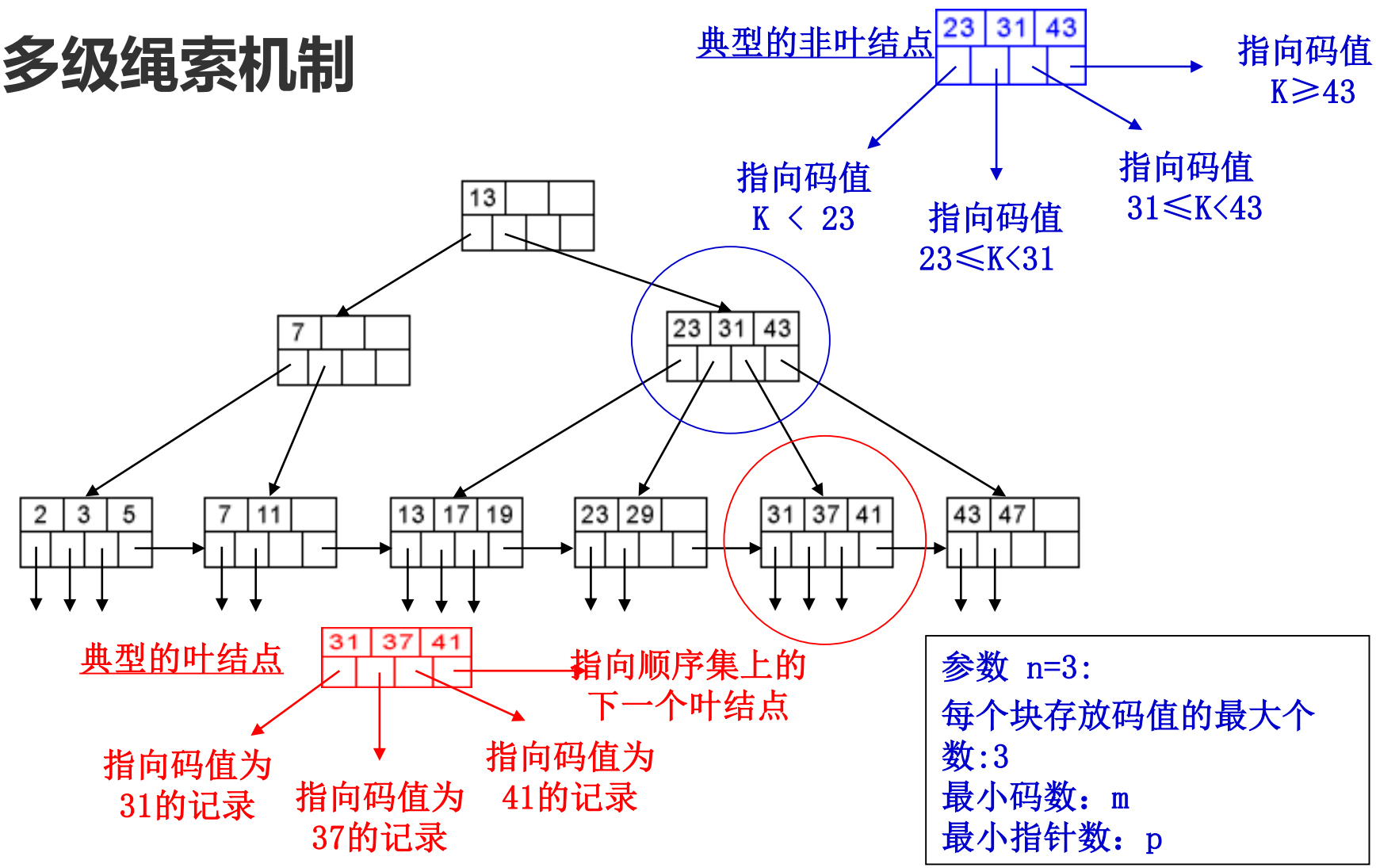
- The data is present in **arbitrary order**, but the **logical ordering** is specified by the index. The data rows may be **spread throughout the table** regardless of the value of the indexed column or expression.
- The non-clustered index tree contains the index keys in sorted order, with the **leaf level of the index containing the pointer to the record** (page and the row number in the data page in page-organized engines; row offset in file-organized engines).
- You can create **multiple nonclustered indexes** on a table or indexed view. Generally, nonclustered indexes are created to improve the performance of frequently used queries not covered by the clustered index.

# B trees 和 B+ Trees

- B树
  - 能自动保持与数据文件大小适应的索引层次
  - 平衡树
  - 能对所使用的存储空间进行管理，使每个块处于全满半满之间
- B+树
  - 特殊的B树，内部结点只存储索引块
  - 叶结点用指针指向数据

# B+树示意

## 多级绳索机制

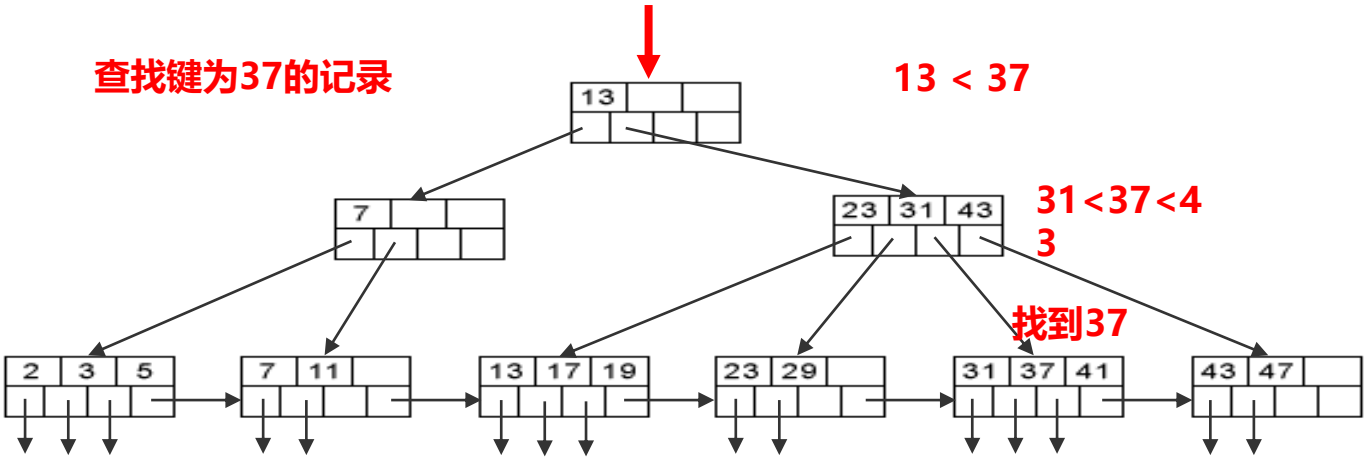



参数 n=3:
每个块存放码值的最大个数:3
最小码数：m
最小指针数：p

# B+树指针和码的数量

| | 最大指针数 | 最大码数 | 最小指针数(指向数据) | 最小码数 |
|---|---|---|---|---|
| 内部结点 | n + 1 | n | $\lceil (n+1)/2 \rceil$ | $\lceil (n+1)/2 \rceil - 1$ |
| 叶结点 | n + 1 | n | $\lfloor (n+1)/2 \rfloor$ | $\lfloor (n+1)/2 \rfloor$ |
| 根结点 | n + 1 | n | 1 | 1 |

# B+树中的查找



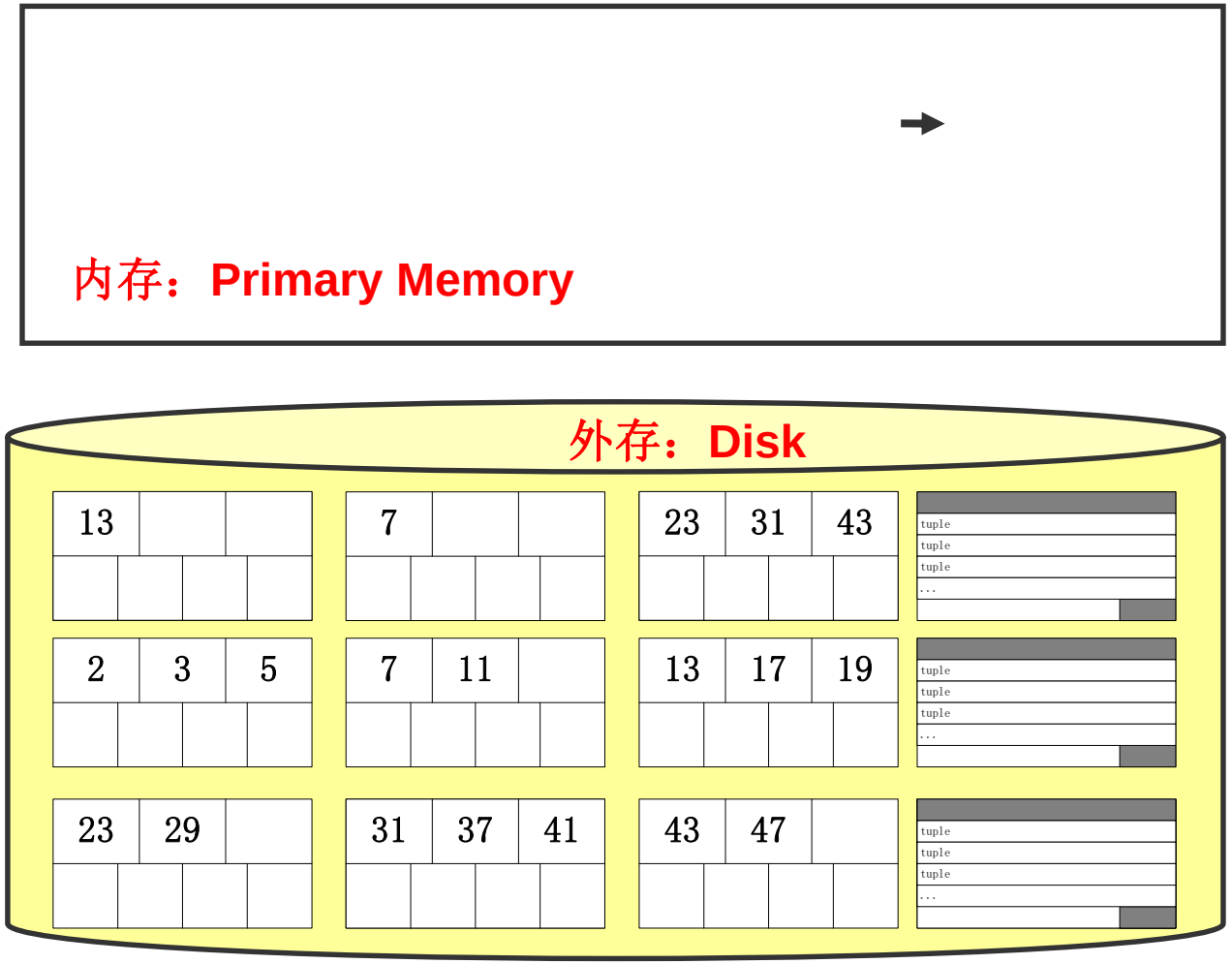

IO数：4
若把第一、第二层结点保存在缓冲区
IO数：2

键值为37的记录

**通过索引明确标明相应编号的牛的所关的地方**
**快速找到每一头牛**

# B+树中的查找(若索引块全在磁盘)

**查找键为37的记录**




读入root节点，IO=1
根据条件13 < 37，读取下一节点块

读入下一节点，IO=2
根据条件31 < 37 < 43，读取下一节点块

读入下一节点，IO=3，找到键值37,读取记录指针,读取记录所在的块

读入数据块，IO=4，根据记录指针找到键值为37的记录

# 例，创建唯一索引

例1:

CREATE **UNIQUE** INDEX ST ON S(SNO)

其中UNIQUE表示要求列SNO的值在基本表S中不重复。

例2:

CREATE **UNIQUE** INDEX SC_INDDEX

ON SC(SNO ASC, CNO DESC)

缺省时，表示升序。

# Clustered Index – 聚簇索引

- Clustering **alters** the data block into a certain distinct order to match the index, resulting in the row data being stored **in order**. Therefore, **only one** clustered index can be created on a given database table.
- Clustered indices can **greatly increase overall speed of retrieval**, but usually only where the data is accessed **sequentially** in the same or reverse order of the clustered index, or when a range of items is selected.

**聚簇索引、聚集索引：在某个或某些属性上有相同值的元组集中存在连续的物理块上。聚簇索引是指索引项的顺序与表中记录的物理顺序一致的索引组织。**

# 聚簇索引

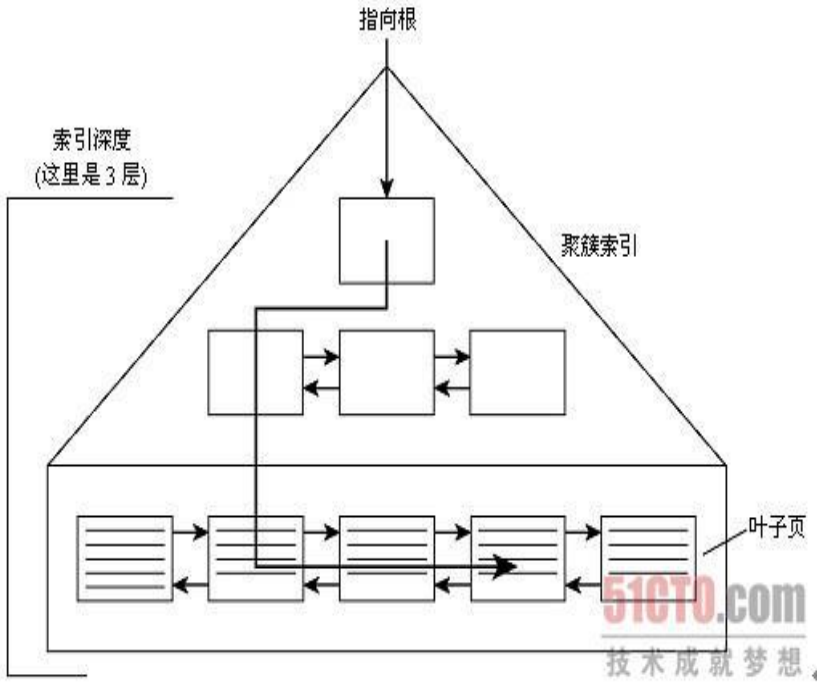
例3：在Student表的Sname（姓名）列上建立一个聚簇索引

```
CREATE CLUSTER INDEX Stusname
        ON  Student(Sname);
```

Student表中的记录将按照Sname值的升序存放

# SQL Server中的聚簇索引

一种特殊的平衡树，与前面的平衡树相比，差别在于索引的叶子级。在聚集索引中，**叶子级并不包括索引键和指针；它们就是数据本身**。这个差异意味着数据并不存储在**堆结构**中。它们存储在索引的叶子级，并按索引键进行排序。



# 建立聚集索引的好处

- 聚集索引对于那些经常要**搜索范围值的列**特别有效。使用聚集索引找到包含第一个值的行后，便可以确保包含后续索引值的行在物理相邻。
- 如果对从表中检索的数据进行**排序时经常要用到某一列**，则可以将该表在**该列上聚集**（物理排序），避免每次查询该列时都进行排序，从而节省成本。

# 聚簇索引与效率

- 在**最经常查询的列上**建立聚簇索引以提高查询效率
- 一个基本表上**最多只能建立一个**聚簇索引
- 经常**更新的列不宜建立聚簇索引**

# 2、索引的撤销

语法：DROP INDEX <索引名>

例: DROP INDEX SC_INDDEX

* 删除索引时，系统会同时从数据字典中删除有关该索引的描述。

* 索引建立后由系统维护，不需用户干预。

# 第三节 数据查询

## SQL的查询语句基本语法结构：

**SELECT- FROM- WHERE**

**SELECT $A_1$, $A_2$, …, $A_n$**
**FROM $R_1$, $R_2$, …, $R_n$**
**WHERE <查询条件表达式>**

SELECT用于检索和统计数据

# SELECT 语句完整的句法：

SELECT [DISTINCT] 列表达式[, 列表达式]

FROM 表名或视图名 [，表名或视图名]…

[WHERE 条件表达式] (条件子句)

[GROUP BY 列名1] (分组子句)

[HAVING 组条件表达式] (组条件子句)

[ORDER BY 列名2[ASC|DESC]..] (排序子句)

# 一、单表查询

## 1、选择表中若干列(SELECT)

### （1）查询指定列（或全部列）

**例1：查看学生全部信息**

```
select sno, sname, age, sex
from s;
```

或：
```
select *
from s;
```

# 一、单表查询

**例2：查询全部学生的学号和姓名。**

```
select sno, sname
from s ;
```

**例3：查询全部学生的姓名和学号。**

```
select sname, sno
from s ;
```

# （2）查询经过计算的列

**例4：查询学生学号、姓名和出生年份。**

```
select sno, sname,  2017 - age
from S ;
```

**若为新的属性取名，则可写为**

```
select sno, sname, 2017-age as 出生年份
from S
```

# 2. 选择表中若干元组(WHERE)

## （1）取消值重复的行

**例5：查询选修了课程的学生学号**

```
select sno
from sc;
```

| sno |
|---|
| 95001 |
| 95001 |
| 95001 |
| 95002 |
| 95002 |

**用DISTINCT取消重复的值**

```
select distinct sno
from sc;
```

| sno |
|---|
| 95001 |
| 95002 |

# （2）查询满足条件的元组

查询满足条件的元组是通过WHERE子句实现。在WHERE子句中常用的查询条件如表所示。

| 查询条件 | 谓词 |
| --- | --- |
| 比较 | =、>、<、>=、<=、<>、!=、!>、!< |
| 确定范围 | BETWEEN AND、NOT BETWEEN AND |
| 确定集合 | IN、NOT IN |
| 字符匹配 | LIKE、NOT LIKE |
| 空值 | IS NULL、IS NOT NULL |
| 多重条件 | AND、OR、NOT |

# 例：带条件的查询

## 例6：查询女学生的学号、姓名和年龄。

```
select sno, sname, age
from s
where sex = '女';
```

## 例7：查询所有年龄大于18岁的女生的姓名和学号。

```
select sname, sno
from s
where age > 18 and sex = '女';
```

# 常用谓词-like

用法: <列名〉like/not like 字符串

列必须为字符串;

例8：查询赵明同学的学号。

```
select sno
    from s
    where sname =?      '赵明' ;
```

★ 通配符： "_" 可代表任一单个字符;
"%" 可代表任意多个字符

# 通配符的使用

**例9：查询所有姓赵的学生的学号和姓名。**

```
Select sno,sname
From s
Where sname like  '赵%'
```

**例10：查询学号第二位是1的所有学生姓名。**

```
Select sname
From s
Where sno like  '_1%' ;
```

# 例11：查询课程名为“DB_DE”的课程号。

```
Select cno
From c
Where cname like  'DB_DE'
Where cname like  'DB\_DE'  ESCAPE  '\' ;
```


**注意**：上例中 ‘\’ 后面的 ‘_’ 不具有通配符含义

**问题**：like何时可用 = 代替?

# 无通配符时不需要用LIKE

如果LIKE后面的匹配串中不含通配符，则可用：

= 取代 LIKE

!= 或 <>取代 NOT LIKE

# 常用谓词—null

用法：where sno is NULL/not NULL

例12：查询选修了课程但没参加考试的学生的学号。

```
select sno
from sc
where grade is null;
```

# 常用谓词—null

例13：查询所有有成绩的学生学号和课程号

```
Select sno, cno
From sc
Where grade is not
null;
```

# 常用谓词—between

含义：查找属性值在指定范围内的元组

用法：

A Between B and C

等价于（A>=B and A<=C)

A not Between B and C

等价于（A>C OR A<B)

# 常用谓词—between

例14：查询所有年龄在17到18岁之间的学生的学号和姓名。

```
select sno,sname
    from s
    where age between 17 and 18
```

```
where age <= 18 and age >=17
```

# 常用谓词—between

**例15：查询所有年龄不在17到18岁之间的学生的学号和姓名。**

```
select sno, sname
from s
where age not between 17 and 18
```

```
where age > 18  or  age < 17
```

# 常用谓词—in

用法：用来查找属性值属于指定集合的元组

<元素> [not] in <结果集>

sno in ('95001', '95002', '95003')

IN操作比较耗时，核心业务尽量少用或不用。因为NOT IN 操作不能应用表的索引，强列推荐不要使用，可以其他方案代替。

为什么NOT IN索引不能用表的索引?

# 常用谓词—in

例16：查询选修了课程号1或2的学生的学号

```
Select sno
From sc
Where cno in ('1', '2');
```

```
where cno = '1'  or  cno = '2'
```

例17：查询年龄不是17岁和18岁的学生的学号

```
Select sno
From s
Where age not in (17, 18);
```

```
where age != 17  and  age != 18
```

# 3、对查询结果排序(Order by)

例18：查询学生学号、姓名和出生年份，并按出生年份的升序排列，出生年份相同时，按学号的降序排列

```
select sno, sname, 2017 - age as 出生年份
from S
order by 出生年份, sno desc
```

**注意**：对于排序列含空值

若按**升序**排列，含空值的元组将**最后**显示。

若按**降序**排列，含空值的元组将**最先**显示。

# 聚集函数—aggregate function

- In database management an **aggregate function** is a function where the values of multiple rows are **grouped** together as input on **certain criteria** to form a single value of more significant meaning or measurement such as a set, a bag or a list.

# 4、使用聚集函数（5个）

## 1）count

count ([distinct]<列名>):

统计一列中值的个数,不计算空值

distinct:表示计算时要取消指定列中的重复值

count ([distinct] *):

计算元组的个数，不管列值是否为空

什么叫聚集？aggregate, aggregation

# 使用聚集函数（5个）

2） sum([distinct]<列名>):

计算一列的总和(此列必须是数值型)

3） avg([distinct]<列名>):

计算一列的平均值(此列必须是数值型)

4） max([distinct]<列名>):

求一列值中的最大值

5） min([distinct]<列名>):

求一列值中的最小值

**除count(*)外，都跳过空值而只处理非空值。**

# 例题

| sno | cno | grade |
|---|---|---|
| s1 | c2 | 80 |
| s2 | c2 | 70 |
| s3 | c2 | |
| s4 | c2 | 90 |

Select avg(grade)

From sc → 此查询的结果为80

Where cno='c2'

# 例题

**例19：求男同学的总人数和平均年龄**

```
Select count(*), AVG(age)
From s
Where sex= ‘男’
```

**例20：统计选修了课程的学生人数**

```
Select count(distinct   sno)
From sc
```

# 例题

## 例21：求选修课程c2的学生的最高分和最低分。

```
Select   max(grade), min(grade)
From   sc
Where cno =  'c2'
```

# 5、分组聚集

## (group by与having)

Group by子句将查询结果按某一列或多列值分组，**值相等的为一组**。分组的**目的**是细化集合函数的作用对象，如果未分组，集合函数作用于整个查询结果。分组后作用于每一组，即每一组有一个函数值。

**例22：求每个同学平均分**

```
select sno , avg(grade)
from SC
group by sno;
```

# HAVING子句

如果分组后还要求按一定条件对这些组进行选择，最终只输出满足条件的组，则可以使用HAVING子句指定选择条件。

## 例23：查询选修课程在三门以上的同学学号

```
select sno
 from sc
 group by sno
 having count(cno)>3
```

# WHERE子句与HAVING子句的区别

(1) WHERE子句作用于基本表或视图，从中**选择满足条件的元组**。

(2) GROUP BY对WHERE的结果进行**分组**

(3) HAVING子句作用于组，从中**选择满足条件的组**，即对分组数据进一步筛选。

# 二、多表查询（连接查询）

1、**连接查询**：同时涉及多个表的查询

2、连接**条件**或连接**谓词**：用来连接两个表的条件

一般格式：

[<表名1>.]<列名1> <比较运算符> [<表名2>.]<列名2>

[<表名1>.]<列名1> BETWEEN [<表名2>.]<列名2> AND [<表名2>.]<列名3>

3、连接**字段**：连接谓词中的列名称

连接条件中的各连接**字段类型必须是可比的**，但名字不必是相同的

# 深入理解连接操作的执行过程

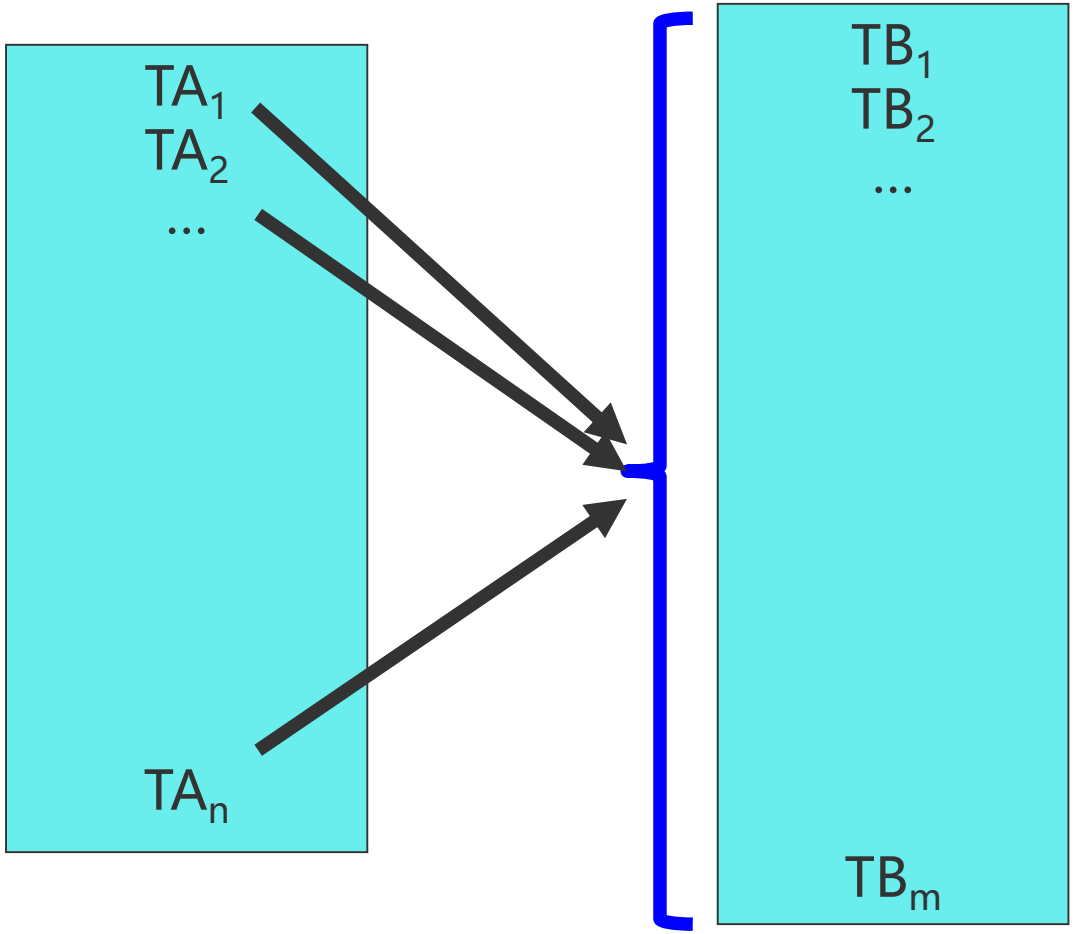
- 嵌套循环法(NESTED-LOOP)—**两重循环根据条件匹配**
  - 首先在**表1中找到第一个元组**，然后从头开始扫描表2，逐一查找满足连接条件的元组，找到后就将表1中的第一个元组与该元组拼接起来，形成结果表中一个元组。
  - 表2全部查找完后，**再找表1中第二个元组**，然后再从头开始扫描表2，逐一查找满足连接条件的元组，找到后就将表1中的第二个元组与该元组拼接起来，形成结果表中一个元组。
  - 重复上述操作，直到表1中的全部元组都处理完毕

**存在的问题?**

# 连接过程示意

n个元组

每次取两个表的两个元组判断是否符合连接条件，符合连接成新元组




m个元组

时间复杂度:

n*m

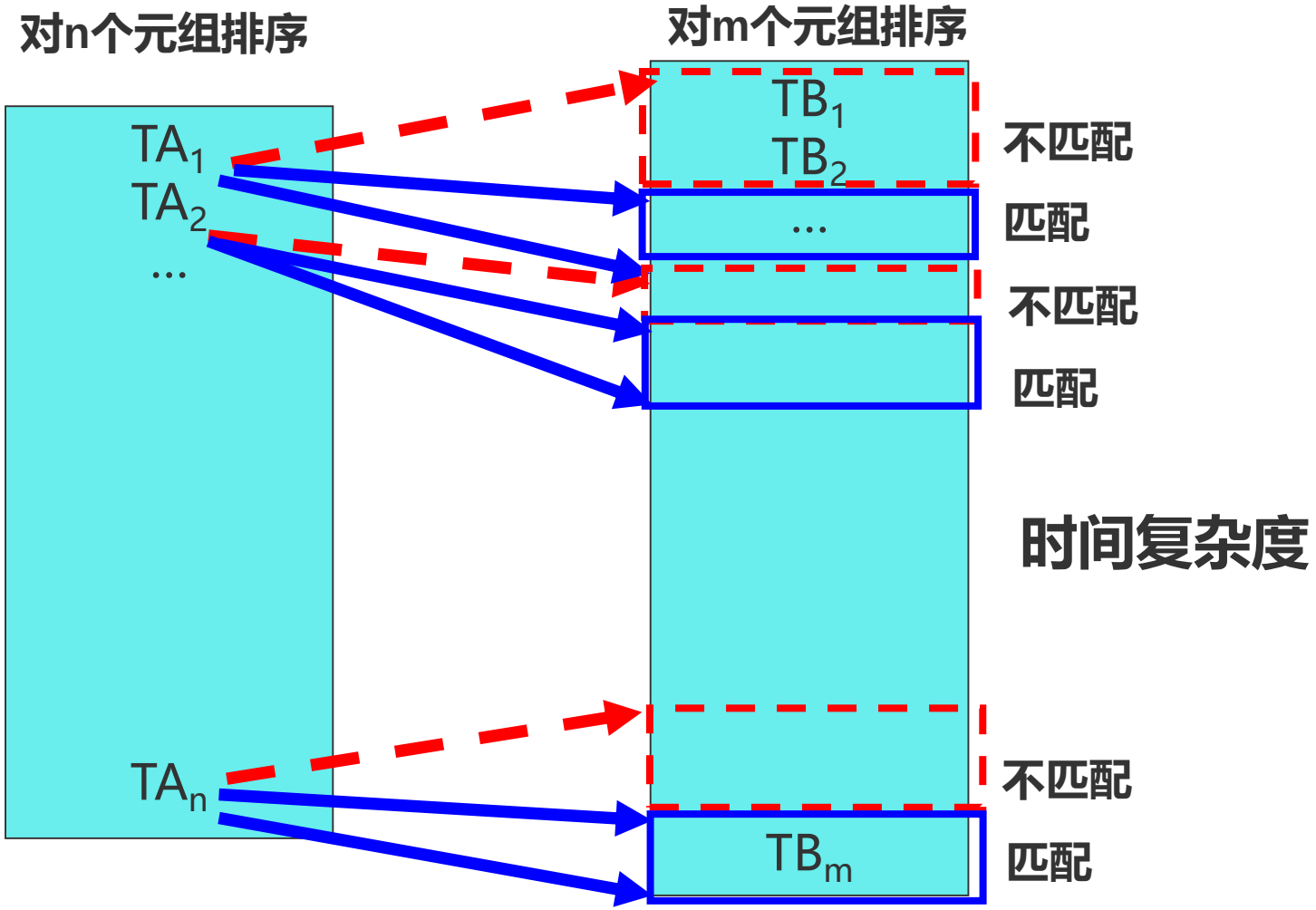
对于 $TA_i$，为什么都要重新从 $TB_1$ 开始?

# 排序合并法(SORT-MERGE)

- 常用于=连接
  - 首先按连接属性对表1和表2分别排序
  - 对表1的第一个元组，从头开始扫描表2，顺序查找满足连接条件的元组，找到后就将表1中的第一个元组与该元组拼接起来，形成结果表中一个元组。当遇到表2中第一条大于表1连接字段值的元组时，**对表2的查询不再继续**
  - 找到表1的第二条元组，然后从刚才的**中断点处继续**顺序扫描表2，查找满足连接条件的元组，找到后就将表1中的第一个元组与该元组拼接起来，形成结果表中一个元组。直接遇到表2中大于表1连接字段值的元组时，**对表2的查询不再继续**
  - 重复上述操作，直到表1或表2中的全部元组都处理完毕为止

# 排序合并法连接过程示意

每次取两个表的两个元组判断是否符合连接条件，符合连接成新元组




**效率如何？代价是什么？**

# 索引连接(INDEX-JOIN)

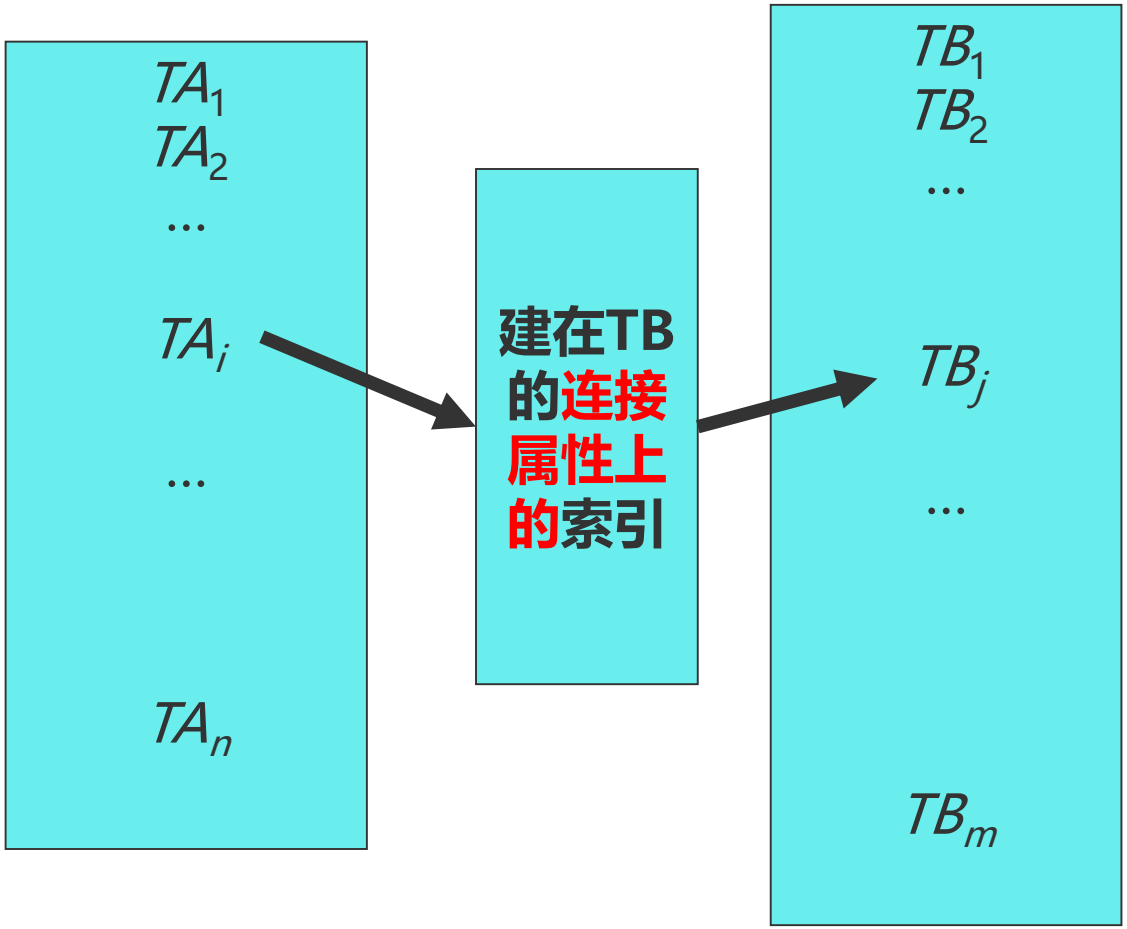
- 对表2按连接字段建立索引
- 挨个遍历表1中的每个元组，依次根据其连接字段值**查询表2的索引**，从表2中找到满足条件的元组，找到后就将表1中的当前元组与该元组拼接起来，形成结果表中一个元组

**效率、成本**

# 连接过程示意

n个元组

对TB，根据连接条件属性建立索引，每次通过计算索引直接找到与$TA_i$相应的TB中的元组集合。




m个元组

**每条元组的连接耗时是什么？**

# 连接查询的种类

- 等值与非等值连接查询
- 自身连接
- 外连接—outer join
- 复合条件连接

# 等值连接

## 1. 等值连接：连接运算符为=

例24：查询每个学生及其选修课程的情况

```
SELECT  Student.*，SC.*
FROM    Student，SC
   WHERE  Student.Sno = SC.Sno;
```

| Student.Sno | Sname | Ssex | Sage | Sdept | SC.Sno | Cno | Grade |
|---|---|---|---|---|---|---|---|
| 200215121 | 李勇 | 男 | 20 | CS | 200215121 | 1 | 92 |
| 200215121 | 李勇 | 男 | 20 | CS | 200215121 | 2 | 85 |
| 200215121 | 李勇 | 男 | 20 | CS | 200215121 | 3 | 88 |
| 200215122 | 刘晨 | 女 | 19 | CS | 200215122 | 2 | 90 |
| 200215122 | 刘晨 | 女 | 19 | CS | 200215122 | 3 | 80 |

# 等值连接

## 连接的两种语法：

### 1、ANSI SQL-89语法

```
SELECT  Student.*, SC.*
    FROM Student, SC
    WHERE Student.Sno = SC.Sno;
```

### 2、ANSI SQL-92语法

```
SELECT  Student.*, SC.*
    FROM  Student JOIN SC
    ON Student.Sno = SC.Sno;
```

# 等值连接举例

## 例25：查询女学生的学号、姓名、成绩：

关系代数：

$$\pi_{S.Sno,Sname,Grade}(\delta_{sex='女' \wedge s.sno=sc.sno}(S \times SC))$$

解：

```
SELECT S.SNO, SNAME, GRADE
FROM   S,  SC
WHERE  S.SNO = SC.SNO AND SEX = '女';
```

# 等值连接举例

## 例26: 查询选修c2或c4课程的学生学号和姓名

**关系代数:**

$$\pi_{sno,\ sname}(\delta_{cno = 'C2' \vee cno = 'C4'}(S \bowtie SC))$$

```
select s.sno, sname
from  s, sc
where  s.sno = sc.sno
    and (cno = 'c2'  or cno = 'c4');
```

# 等值连接举例

## 例27：找出平均成绩90以上的女生姓名

```
select sname
from s, sc
where s.sno = sc.sno and sex = '女'
    group by sname
    having avg(grade) > 90;
```

**Oracle中，GROUP BY子句中的列一定要出现在SELECT中**

# 等值连接举例

## 例27：找出平均成绩90以上的女生姓名

```
select sno,sname
from s，sc
where s.sno = sc.sno and sex = ‘女’
    group by sno
    having avg(grade)>90 ； (×)
```

**注意：如果使用了分组子句，则查询列表中的每个列要么是分组依据列（GROUP BY），要么是聚集函数**

# 等值连接举例

```
select sno, sname
from s, sc
where s.sno = sc.sno and sex = '女'
   group by sno, sname
   having avg(grade) > 90 ;
```

# 连接举例

**例28：求LIU老师所授课程的名称及每门课程的学生平均成绩。**

```
SELECT C.CNO, C.CNAME, AVG(Grade)
FROM SC, C
WHERE SC.CNO = C.CNO and TName = 'LIU'
GROUP BY C.CNO,C.CNAME
```

**注意：如果使用了分组子句，则查询列表中的每个列要么是分组依据列（GROUP BY），要么是聚集函数**

# 例29: 统计每一年龄选修课程的学生人数

```
select age, count(distinct s.sno)
from s, sc
where s.sno = sc.sno
group by age
```

# 2. 自身连接

同一个表的不同元组之间的连接称为自身连接。必须给表取别名，当作两个不同的表来处理。

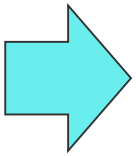
**例30：查询每一门课的间接先修课（即先修课的先修课）**

```
SELECT  FIRST.Cno, SECOND.Cpno
FROM  Course FIRST, Course SECOND
WHERE FIRST.Cpno = SECOND.Cno;
```

# 自身连接示例

FIRST表（Course表）　　SECOND表（Course表）

| Cno | Cname | Cpno | Ccredit |
|---|---|---|---|
| 1 | 数据库 | 5 | 4 |
| 2 | 数学 | | 2 |
| 3 | 信息系统 | 1 | 4 |
| 4 | 操作系统 | 6 | 3 |
| 5 | 数据结构 | 7 | 4 |
| 6 | 数据处理 | | 2 |
| 7 | PASCAL语言 | 6 | 4 |



| Cno | Pcno |
|---|---|
| 1 | 7 |
| 3 | 5 |
| 5 | 6 |

# 例31：查询至少选修c2和c4课程的学生学号。

**关系代数:** $\pi_{1}(\delta_{1=4 \wedge 2='C2' \wedge 5='C4'}(SC \times SC))$

```
select x.sno
from  sc as x, sc as y
where  x.sno = y.sno
        and x.cno = 'c2'
        and y.cno = 'c4' ;
```

# 3. 多表的连接

**例32：查询选修课程名为DB的学生学号和姓名。**

**关系代数：** $\pi_{Sno,Sname}(\delta_{Cname='DB'}(S \bowtie SC \bowtie C))$

```
select s.sno, sname
from  s,  sc, c
where  s.sno = sc.sno
       and sc.cno = c.cno
       and c.cname = 'DB' ;
```

# 4. 外连接

- 外连接与普通连接的区别
  - 普通连接操作只输出满足连接条件的元组
  - 外连接操作以指定表为连接主体，将主体表中不满足连接条件的元组一并输出

# 外连接（续）

- 左外连接
  - 列出左边关系中所有的元组，右边关系中未有匹配的列用NULL作为占位符
- 右外连接
  - 列出右边关系中所有的元组，左边关系中未有匹配的列用NULL作为占位符
- 全外连接
  - 列出两边关系中所有的元组，未有匹配的列用NULL作为占位符

# 外连接（续）

```
SELECT Student.Sno, Sname, Ssex, Sage, Sdept, Cno, Grade
FROM Student LEFT OUTER JOIN SC ON (Student.Sno = SC.Sno);
```

执行结果：

| Student.Sno | Sname | Ssex | Sage | Sdept | Cno | Grade |
|---|---|---|---|---|---|---|
| 200215121 | 李勇 | 男 | 20 | CS | 1 | 92 |
| 200215121 | 李勇 | 男 | 20 | CS | 2 | 85 |
| 200215121 | 李勇 | 男 | 20 | CS | 3 | 88 |
| 200215122 | 刘晨 | 女 | 19 | CS | 2 | 90 |
| 200215122 | 刘晨 | 女 | 19 | CS | 3 | 80 |
| 200215123 | 王敏 | 女 | 18 | MA | NULL | NULL |
| 200215125 | 张立 | 男 | 19 | IS | NULL | NULL |

在外连接中，可以在表名之间使用关键字LEFT OUTER JOIN、RIGHT OUTER JOIN 、以及FULL OUTER JOIN, **LEFT**关键字表示左边表的行是保留的， **RIGHT**关键字表示右边表的行是保留的， **FULL**关键字表示左右两边表的行都是保留的。

# 三、嵌套查询（子查询）

一个SELECT-FROM-WHERE语句称为一个查询块，将一个查询块**嵌套**在另一个查询块的WHERE子句的**条件**中的**查询称为嵌套查询**。

```
select sname
from s
where sno in
(
  select sno
  from sc
  where cno='C2'
);
```

外查询（父查询）

内查询（子查询）

**执行顺序**由里往外，子查询的结果作为父查询的条件。

# （一）子查询的分类

子查询可分成**不相关子查询**和**相关子查询**

## 1、不相关子查询（一般子查询）

**子查询**的查询条件**不依赖**于**父查询**，子查询可以**独立执行**，这类子查询称为**不相关子查询**。

不相关子查询的**执行过程**：

(1) **先执行子查询**，**子查询只执行一次**，子查询的结果作为父查询的条件。

(2)**根据子查询的结果再执行父查询。**

**不相关子查询一般使用谓词IN**

# 不相关子查询（一般子查询）

查询有一门课程成绩等于95分的学生的学号和姓名。

该查询可以采用三种不同的方法实现：

方法一：采用多表等值连接实现

方法二：采用不相关子查询实现

方法三：采用相关子查询实现

## （1）采用多表等值连接实现

```
select s.sno, sname
from s, sc
where s.sno = sc.sno and grade = 95
```

有什么问题?

存在效率问题　　解决方案?

# 采用不相关子查询实现

查询有一门课程成绩等于95分的学号和姓名。

```
select sno，sname  from s
 where sno in
      (
           select sno from sc
           where grade = 95
      );
```

父查询

子查询

特点:<1> 能独立运行,子查询条件不依赖父查询
<2> 只会运行一次
<3> 先执行子查询

# 回顾例27：找出平均成绩90以上的女生姓名

```
select sno, sname
from s,  sc
where s.sno = sc.sno
        and sex =  '女'
group by sno, sname
having avg(grade) > 90;
```

**代码短是否等于效率高?**

```
select sname
   from s
   where sno in
   (
      select sno
      from s,  sc
      where s.sno = sc.sno
              and sex =  '女 '
      group by sno
      having avg(grade)>90
   );
```

# 2. 关子查询

**子查询**的查询条件**依赖**于外层**父查询**的某个属性值，称这类查询为**相关子查询**。

**相关子查询一般使用谓词exists**

**例34:采用相关子查询实现查询有一门课程成绩等于95分的学生学号和姓名。**

```
select sno，sname
from s
where exists
(
   select *
   from sc
   where s.sno=sc.sno
      and grade = 95
);
```

相关子查询的**执行过程**:

首先取外层查询中表s的第1个元组，根据它与内层查询**相关的属性值**sno处理内层查询，若查询结果**非空**，则WHERE子句返回真，则取此元组放入结果表中，然后取外层查询中表的下一个元组，重复上述过程，直到外层查询中表全部检索完。

特点: <1>子查询不能独立运行。<2>子查询多次运行。
<3>先执行外查询

# 查询有一门课程成绩等于95分的学生学号和姓名

```
select sno，sname from s
 where exists
 ( select *
      from sc
     where s.sno=sc.sno
     and grade = 95);
```

结果集为：

| sno | sn |
|---|---|
| s3 | 李红 |
| S4 | 赵立 |

S

| sno | sn | age | sex |
|---|---|---|---|
| s1 | 丁岩 | 19 | M |
| s2 | 王爽 | 17 | F |
| s3 | 李红 | 18 | F |
| s4 | 赵立 | 21 | M |

SC

| sno | cno | Grade |
|---|---|---|
| s1 | c1 | 79 |
| s1 | c3 | 85 |
| s2 | c1 | 60 |
| s2 | c2 | 83 |
| s2 | c3 | 90 |
| s3 | c1 | 83 |
| s3 | c2 | 95 |
| s4 | c1 | 75 |
| s4 | c2 | 95 |

# （二）带有各种运算符的子查询

## 1. 带有 IN 谓词的子查询

当子查询的结果是一个集合时，经常使用带IN谓词的子查询。

<元素> in <子查询的结果集>

回顾例24：查询有一门课程成绩等于95分的学生学号和姓名。

```
select sno，sname
from s
where sno in
( select sno from sc  where grade = 95);
```

# 带有各种运算符的子查询

**回顾例32：查询选修了课程名为“DB”的学号和姓名**

**（1）用多表连接查询来实现**

```
select s.sno, sname
From s, sc, c
Where s.sno = sc.sno and
      c.cno = sc.cno and
      c.cname = ‘DB’ ;
```

# 带有各种运算符的子查询

## （2）用子查询实现例34：查询选修了课程名为“DB”的学号和姓名

```
select sno, sname
from s
where sno in
    (select sno
     from sc
     where cno in
          (select  cno
           from c
           where cname='DB'));
```

(3)最后在S表中取出SNO和SNAME

(2)然后在SC表中找出选修了3号课程的学生学号

(1)首先在C表中找出‘DB’的课程号3

# 2. 带有比较运算符的子查询

当用户能确切知道内层子查询返回的是单值时，可以用比较运算符 代替IN

例33：查询和“刘晨”相同年龄的学生学号和姓名。

用连接：

```
select y.sno,  y.sname
from s  x , s  y
where x.age = y.age
   and  x.sname =  ‘刘晨’
```

用子查询：

```
select sno, sname
from s
where age = (
     select age
     from s
    where sname =  ‘刘晨’
);
```

找出每个学生超过他选修课程平均成绩的课程号。


```
SELECT sno，  cno
FROM  SC  x
WHERE Grade >=
      (
       SELECT AVG(Grade)
        FROM  SC y
        WHERE y.cno = x.sno
      );
```

# 可能的执行过程：

1. 从外层查询中取出SC的一个元组x，将元组x的sno值（200215121）传送给内层查询。

```
SELECT AVG(Grade)
FROM SC y
WHERE y.sno = '200215121';
```

2. 执行内层查询，得到值88（近似值），用该值代替内层查询，得到外层查询：

```
SELECT sno， cno
FROM  SC x
WHERE SC.sno =  '200215121' and
Grade >= 88;
```

# 可能的执行过程：

3. 执行这个查询，得到

(200215121，1)

(200215121，3)

4. 外层查询取出下一个元组重复做上述1至3步骤，直到外层的SC元组全部处理完毕。结果为:

(200215121，1)

(200215121，3)

(200215122，2)

# 3. 带有ANY或ALL谓词的子查询

(1) any

<表达式> <比较运算符> any <子查询结果集>

子查询结果集的某个值

(2) all

<表达式> <比较运算符> all <子查询结果集>

子查询结果集的所有值

例如:

age > all(13, 15, 17)

age > any(13, 15, 17)

# 需要配合使用比较运算符

| | |
|---|---|
| > ANY | 大于子查询结果中的某个值 |
| > ALL | 大于子查询结果中的所有值 |
| < ANY | 小于子查询结果中的某个值 |
| < ALL | 小于子查询结果中的所有值 |
| >= ANY | 大于等于子查询结果中的某个值 |
| >= ALL | 大于等于子查询结果中的所有值 |
| <= ANY | 小于等于子查询结果中的某个值 |
| <= ALL | 小于等于子查询结果中的所有值 |
| = ANY | 等于子查询结果中的某个值 |
| =ALL | 等于子查询结果中的所有值（通常没有实际意义） |
| != （或<>） ANY | 不等于子查询结果中的某个值 |
| != （或<>） ALL | 不等于子查询结果中的任何一个值 |

# 例34:查询选修c2课程的学生学号和姓名

用in

```
select sno, sname
from s
where sno in
    (
        select sno from sc
        where cno =  'c2'
    );
```

用=any

```
select sno, sname
from s
where  sno = any
    (
        select sno from sc
        where cno =  'c2'
    );
```

# 例35：查询没选修c2课程的学生学号和姓名。

用 not in:

```
select sno, sname
from  s
where  sno not in
   (
      select sno  from sc
      where cno=  'c2'
   );
```

用<> all:

```
select sno, sname
from s
where  sno <> all
   (
     select sno  from sc
     where cno =   'c2'
   );
```

例36：查询男同学中比某一女生年龄小的学生姓名和年龄。

用any：

```
select sname, age  from s
where sex = ’男’  and
      age < any
                (
                     select age
                     from s
                     where sex = ‘女’
                );
```

用max函数：

```
select sname, age from s
where sex = ’男’  and
      age < (
                   select max(age)
                   from s
                   where sex = ‘女’
                );
```

例37：查询男生中比所有女生年龄都小的学生姓名和年龄。

用all：

```
select sname, age  from s
where sex = '男' and
      age <   all
            (     select age
                  from s
                  where sex=  '女'
            );
```

用min函数：

```
select sname, age from s
where sex = '男'   and
      age < (
                  select min(age)
                  from s
                  where sex = '女'
             );
```

# 4. 带有 EXISTS 谓词的子查询

EXISTS代表**存在量词**。带有EXISTS谓词的**子查询不返回任何数据**，只产生逻辑真值“true”或逻辑假值“false”。若**子查询结果为非空**，则**父查询**的WHERE子句返回**真值**，否则，返回假值。

**用法：**

**[not] exists (<子查询>)**

# 回顾例34:查询没选修c2课程的学生学号和姓名

```
select sno, sname
from s
where sno not in
   (select sno
    from sc
     where cno = 'c2');
```

```
select sno, sname
from s
where sno <>all
(select sno
 from sc
  where cno = 'c2');
```

```
select sno, sname from s
where not exists
 (select * from sc
  where s.sno = sc.sno
        and cno = 'c2');
```

next

# 图示：查询没选修c2课程的学生学号和姓名

```
select sno,sname
from  s
where  not exists
 (select *   from sc
     where s.sno = sc.sno
        and cno = 'c2');
```

结果集为：

| sno | sn |
|---|---|
| S1 | 丁岩 |

S

| sno | sn | age | sex |
|---|---|---|---|
| s1 | 丁岩 | 19 | M |
| s2 | 王爽 | 17 | F |
| s3 | 李红 | 18 | F |
| s4 | 赵立 | 21 | M |

SC

| sno | cno | G |
|---|---|---|
| s1 | c1 | 79 |
| s1 | c3 | 85 |
| s2 | c1 | 60 |
| s2 | c2 | 83 |
| s2 | c3 | 90 |
| s3 | c1 | 95 |
| s3 | c2 | 80 |
| s4 | c1 | 75 |
| s4 | c2 | 85 |

返回

SQL语言中没有提供**全称量词** $\forall$ 但可以把带有全称量词的谓词**转换**成等价的带有**存在量词**的谓词。

$$(\forall x)P = \neg(\exists x(\neg P))$$

例38和例39都表示全部课程，转换成等价的存在量词表示。

# 例38：检索选修全部课程的学生姓名

**(相当于查询这样的学生，没有一门课是他不选的)**

```
select sname    from  s
where  not exists
   ( select *  from c
      where not exists
          (select *  from sc
           where s.sno = sc.sno
              and sc.cno = c.cno));
```

**提示：请课后自己一定仔细理解，彻底弄懂**

# 例39.检索选修课程包含LIU老师所授全部课的学生学号

```
SELECT SNO
FROM SC  X
WHERE NOT EXISTS
        ( SELECT *
          FROM C
          WHERE T = 'LIU'
       AND NOT EXISTS
             ( SELECT *
               FROM SC  Y
               WHERE X.SNO = Y.SNO
                 AND Y.CNO = C.CNO ));
```

**相当于查询这样的学生，没有一门LIU老师所授的课是他不选的**

# 几点说明

- SQL语言中表名的顺序，条件顺序无关，SQL是非过程化语言。
- 查询条件包括:

  连接条件+选择条件
- 不相关子查询**不一定**能转化为多表的连接查询，而连接查询一定能用不相关子查询实现。

# 四、集合查询

SELECT 语句的查询结果是元组的集合，可以将多个SELECT语句的结果进行集合操作。

集合操作主要包括 UNION （并）、INTERSECT（交）、MINUS（差）。

注意：参加集合操作的各个结果表的列数必须相同，对应项的数据类型也必须相同。

# 1. UNION

## 例40：查询选修课程1或选修课程2的学生

```
(
  select sno
  from sc
  where cno = '1'
)
union
(
  select sno
  from sc
  where cno = '2'
);
```

```
select sno
from sc
where   cno = '1' or
        cno = '2';
```

# 2. INTERSECT

**例41：查询年龄不大于19岁的女学生。**

```
select *
from s
where
  sex = '女'   and  age <= 19;
```

```
(
  select *
  from s
  where sex = '女'
)
intersect
(
  select *
  from s
  where age<=19
);
```

# 3. MINUS (EXCEPT)

## 例42：查询年龄大于19岁的女学生

```
select *
from s
where sex = '女'    and
          age>19;
```

```
(
  select *
  from s
  where sex = '女'
)
minus
(
  select *
  from s
  where age <= 19
);
```

# 五、基于派生表的查询

子查询不仅可以出现在Where子句中，还可以出现在From子句中，这时子查询生成的临时派生表成为主查询的查询对象。

找出每个学生超过他选修课程平均成绩的课程号。

```
select sno, cno
from  SC  x
where Grade >= (
                  select AVG(Grade)
                   from  SC y
                   where y.sno = x.sno
                );
```

# 五、基于派生表的查询

（1）利用派生表Avg_sc记录每个学生的学号和平均成绩

（2）主查询将SC表与Avg_sc按学号相等连接，

（3）选出修课成绩大于平均成绩的课程号

```
SELECT Sno,  Cno
FROM  SC,(SELECT Sno,Avg(Grade) FROM  SC GROUPBY Sno)
            AS Avg_sc(avg_sno,avg_grade)
WHERE SC.Sno=Avg_sc.  avg_sno and
      SC.  Grade>= Avg_sc. avg_grade);
```

# 五、基于派生表的查询

如果子查询中没有聚集函数，派生表可以不指定属性列，但必须指定别名

查询所有选修了1号课程的学生姓名

```
SELECT Sname
FROM  Student,(SELECT Sno FROM  SC WHERE Cno= '1' )
                    AS SC1
WHERE Student.Sno=SC1.sno;
```

# 第四节 数据更新

## 一、插入数据

### 1、用Values子句向表中插数据

语法:insert into <基本表名>[(列名表)]
values (元组值);

例： insert into s(sno,sname,age)
values ( ‘s1’ , ‘李涛’ , 19)

insert into s
values ( ‘s1’ , ‘李涛’ ,19, ‘男’ )

# 2. 用子查询向表中插数据

语法：insert into 〈表名〉[(〈列名表〉)]

select 查询语句;

例43：把成绩不及格的学生的学号、姓名、课程号和成绩存入另一个已知基本表NOPASS(SNO , SNAME , CNO, GRADE)中。

```
insert  into  NoPass
select  s.sno , sname , cno, grade
from    s , sc
where s.sno = sc.sno  and
          grade < 60;
```

# 二、数据删除

语法:

DELETE FROM 基本表名
[WHERE 条件表达式];

★ DELETE语句只能从一个基本表中删除满足WHERE子句中的条件的元组，即其后只能有一个基本表名。

★ DELETE只删表中的数据，表的定义仍然在数据字典中。

# 案例

**例如：在基本表SC中删除无成绩的选课记录**

```
DELETE
FROM SC
WHERE Grade IS NULL;
```

**例如：删除所有女学生的选课记录**

```
Delete
from sc
where sno in(select sno
             from s
             where sex ='女');
```

# 例

例：在sc表中删除‘操作系统’课程成绩低于该课的平均成绩的所有元组。

```
DELETE FROM SC WHERE
CNO=(   SELECT CNO
        FROM C
        WHERE CNAME=‘操作系统’)
AND GRADE<
      ( SELECT AVG(GRADE)
        FROM SC,C
        WHERE SC.CNO=C.CNO AND
              CNAME=‘操作系统’)
```

# 三、数据修改

语法:

```
UPDATE 基本表名
SET 列名=值表达式
    [,列名=值表达式...]
[WHERE 条件表达式];
```

★ update语句只能修改一个基本表中满足where条件的元组的某些列值，即其后只能有一个基本表名。

# 数据修改举例

**例44: 把课程号为‘c5’的课程的任课教师名改为wu**

```
update c
set tname = ‘wu’
where cno = ‘c5’
```

**例45: 将c2课程的非空成绩提高10%**

```
UPDATE SC
SET GRADE = GRADE * (1 + 10%)
WHERE CNO = ‘C2’ AND GRADE IS NOT NULL
```

# 例46:把全体女学生的成绩置0

```
Update sc
Set grade = 0
where  sno in
    ( select sno
      from s
      where sex = '女'
    );
```

# 空值处理

空值出现的情况：

1. 属性应该有值，但目前不知道具体的值
2. 属性不应该有值

   缺考学生的成绩
3. 由于某种原因不便填写

   一个人的电话号码不想让人知道

# 空值的产生

1. 插入

```
INSERT INTO SC
        VALUES('201215126', '1',  NULL);
INSERT INTO SC(Sno, Cno)
        VALUES('201215126', '1');
```

2. 修改

```
UPDATE Student
   SET Sdept = NULL;
```

3. 外连接

# 空值的判断与约束条件

IS NULL

IS NOT NULL

**属性定义中有以下约束的，不能取空值：**

**1. NOT NULL约束条件**

**2. UNIQUE约束条件**

**3. PRIMARY KEY约束条件**

# 空值运算

1. 算术运算

   只要存在操作数为空值的情况，计算结果就是空值

2. 比较运算

   只要存在操作数为空值的情况，比较结果就是UNKNOWN

3. 逻辑运算

   三值逻辑(TRUE, FALSE, UNKNOWN)

# 逻辑运算真值表

| X | Y | X AND Y | X OR Y | NOT X |
|---|---|---|---|---|
| T | T | T | T | F |
| T | U | U | T | F |
| T | F | F | T | F |
| U | T | U | T | U |
| U | U | U | U | U |
| U | F | F | U | U |
| F | T | F | T | T |
| F | U | F | U | T |
| F | F | F | F | T |

# 例

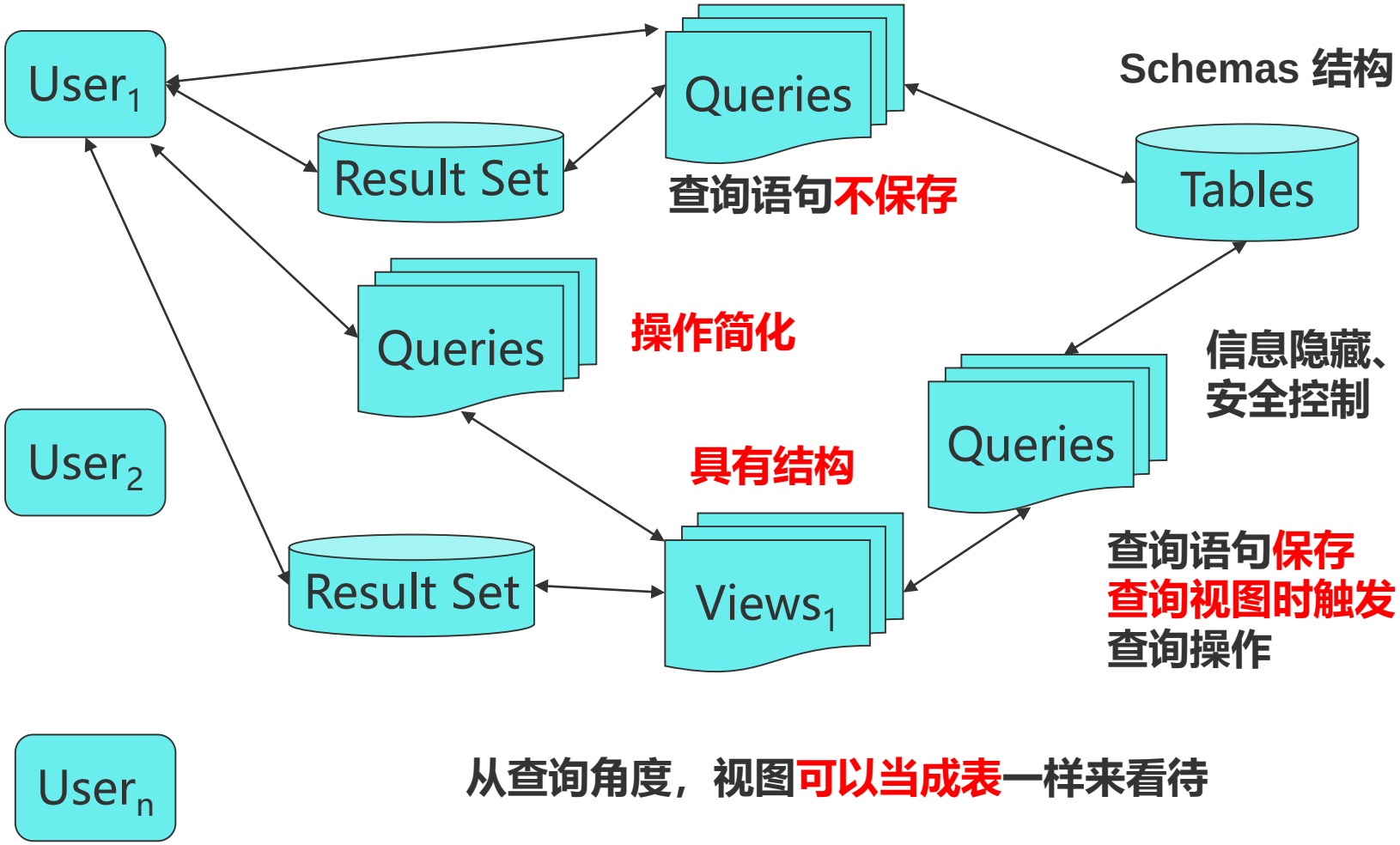
- 查询时，只有使WHERE和 HAVING子句中的选择条件为 TRUE的元组才输出

找出选修1号课程的不及格学生

```
SELECT sno
 FROM SC
 WHERE Grade < 60 AND cno = '1';
```

- 选出的是参加考试而不及格的学生

# 第五节 视图—view


User1
Queries
Schemas 结构
Result Set
查询语句不保存
Tables
Queries
操作简化
信息隐藏、
安全控制
User2
具有结构
Queries
查询语句保存
查询视图时触发
查询操作
Result Set
Views1
Usern
从查询角度，视图可以当成表一样来看待

# Definition of view

- In database theory, a view is **the result set** of **stored query** on the **data**, which the database users can **query just as** they would in a persistent database collection object.
- This **pre-established query command** is kept in the **database dictionary**.
- **Unlike ordinary base tables** in a relational database, a view **does not form** part of the **physical schema**: as a result set, it is a **virtual table** computed or collated **dynamically** from data in the database **when access to that view** is requested.
- **Changes** applied to the data in a relevant underlying table are **reflected** in the data shown in subsequent invocations of the view.

# 视图的特点

（1）**虚表**，是从一个或几个基本表（或视图）导出的表

（2）**只存放**视图的**定义**，**不存放**视图对应的**数据**

（3）基表中的数据发生**变化**，从视图中查询出的数据也**随之改变**

# 针对视图的操作

- 创建
- 查询
- 删除
- 受限更新
- 定义基于该视图的新视图

# 2. 创建视图

语法：

```
CREATE VIEW <视图名>
    [ (<列名1>,<列名2>,...) ]
    AS SELECT 查询语句
    [WITH CHECK OPTION];
```

WITH CHECK OPTION 表示：

对视图进行UPDATE,INSERT和DELETE操作时要保证更新、插入和删除的行满足视图定义中的谓词条件

# 创建视图案例

```
CREATE
      VIEW S_G(sno, sname, cname, grade)
AS
      SELECT  s.sno, sname, cname, grade
      FROM    S, SC, C
      WHERE S.SNO = SC.SNO
                  AND SC.CNO = C.CNO;
```

# 注意

- **组成视图的属性列名或者全部省略，或者全部指定，没有第三种选择。**
- 子查询不允许含有ORDER BY子句和DISTINCT短语
- 下列情况下必须明确指定组成视图的所有列名：
  (1)某个目标列是聚集函数或列表达式
  (2)多表连接时选出了几个同名列作为视图的字段
- RDBMS执行CREATE VIEW语句时只是把视图定义存入数据字典，并不执行其中的SELECT语句。
- 在对视图查询时，按视图的定义从基本表中将数据查出

# 视图的分类

（1）**行列子集视图**

若一个视图从**单个基本表导出**，并且只是去掉了基本表的某些行和某些列，但**保留了主码**，称这类视图为行列子集视图。

（2）**带表达式视图**

在定义视图时根据需要设置一些**派生列**，这些派生列在基本表中并不实际存在，所以称为**虚拟列**。带虚拟列的视图称为带表达式视图。

（3）**分组视图**

带有**聚集函数和GROUP BY子句**的查询来定义视图，这种视图称为分组视图。

# 例—行列子集视图

建立一个行列子集视图。建立一个男生的视图，并保证进行插入和修改操作时该视图只有男生。

```
create view m_student
as  select sno, sname, age
      from s
      where sex= ‘男’
      with check option;
```

由于在定义视图时加了with check option子句，以后对该视图进行插入、修改和删除操作时，RDBMS会自动加上sex= ‘男’ 的条件

# 对m_student视图的更新操作：

- 修改操作：自动加上sex=‘男’的条件
- 删除操作：自动加上sex=‘男’的条件
- 插入操作：自动检查sex属性值是否为‘男’
  - 如果不是，则拒绝该插入操作
  - 如果没有提供sex属性值，则自动定义sex为‘男’

例：建立一个带表达式的视图，反映学生出生年份的视图。

```
create view
        bt_s(sno, sname, birth)
as
    select sno, sname, 2017 - age
    from s;
```

例：建立一个**分组视图**，反映学生的学号和平均成绩。

```
create view
        s_avg_g(sno, gavg)
as
        select sno, avg(g)
        from sc
        group by sno;
```

# 基于视图的视图

(1) 建立信息系选修了1号课程的学生视图。

```
CREATE VIEW IS_S1(sno，sname，Grade) AS
SELECT student.sno，sname，Grade
FROM  student，SC
WHERE student.sno = SC.sno
      AND sdept = 'IS'
      AND SC.cno = '1';
```

(2) 建立信息系选修了1号课程且成绩在90分以上的学生的视图。

```
CREATE VIEW IS_S2 AS
SELECT sno，sname，Grade
FROM  IS_S1
WHERE  Grade >= 90;
```

# 3. 撤销视图

语法：DROP VIEW <视图名>[CASCADE]

用CASCADE语句，将某个视图及其导出的所有视图全部删除

例: DROP VIEW IS_S1 CASCADE

# 4. 查询视图

视图定义后，用户可以象对基本表一样对视图进行查询。

例: Select * from S_G where cname= ‘DB’ ;

视图定义为:

```
CREATE VIEW
S_G(sno, sname, cname, grade)
AS SELECT
   s.sno, sname, cname, grade
    FROM   s, sc, c
   WHERE  s.sno = sc.sno
        AND sc.cno = c.cno;
```

转化为:

```
SELECT s.sno, sname, cname, grade
    FROM   s,sc,c
  WHERE  s.sno = sc.sno
       AND sc.cno = c.cno
       AND cname = ‘DB’ ;
```

# 4. 查询视图

**例如：建立一个视图s_avg_g，包含每个同学的学号和平均成绩。**

```
create view s_avg_g(sno, gavg)
 as   select sno, avg(grade)
            from sc
            group by sno;
```

**在s_avg_g视图中查询平均成绩在90分以上的学生学号和平均成绩。**

```
select *
from s_avg_g
where gavg >= 90
```

将上述查询语句与定义视图的子查询结合后，转换后形成下列查询语句：

```
select sno, avg(grade)
from sc
where avg(grade) >= 90          (×)
group by sno;
```

正确的查询语句应该为：

```
select sno, avg(grade)
from sc
group by sno
having avg(grade)>=90;          (√)
```

# 5. 对视图的更新

对视图的查询是和基本表相同的，但是更新操作则受到下列**三条规则的限制**：

★如果视图是从多个**基本表使用连接**操作导出的，则**不允许更新**。

★如果导出的视图使用了**分组和聚集**操作，也**不允许更新**。

★如果视图是从单个基本表使用选择和投影操作导出的，并且包括了基本表的主键，即视图为**行列子集视图**，则**可以执行更新操作**。

# 5、对视图的更新

**建立如下视图**

```
CREATE VIEW S_G
        (sno, sname, cname, grade)
AS  SELECT S.SNO, SName, CName, Grade
    FROM  S, SC, C
    WHERE  S.SNO = SC.SNO
    AND   SC.CNO = C.CNO
```

**不能对上述视图进行更新；**

# 5. 对视图的更新

**建立视图:每个学生选课（Grade非空）的门数及平均成绩；**

```
CREATE VIEW S_GRADE(SNO, C_NUM, AVG_G)
  AS SELECT  SNO, COUNT(CNO), AVG(Grade)
     FROM SC
     WHERE Grade IS NOT NULL
     GROUP  BY  SNO
```

**不能对上述视图进行更新**

# 5. 对视图的更新

## 建立有关男同学的视图

```
CREATE VIEW S_MALE
 AS SELECT  SNO, SName, AGE
    FROM S
   WHERE SEX = ‘男’
   WITH CHECK OPTION
```

**可以对上述视图进行更新**

# 5. 对视图的更新

## 对视图S_MALE执行插入操作

INSERT INTO S_MALE

VALUES（‘S28’，‘WU’,18)

## 系统会自动把它转变为：

INSERT INTO S

VALUES（‘S28’，‘WU’,18，‘男’）

# 5. 对视图的更新

## 对视图S_MALE执行修改操作

```
UPDATE S_MALE
SET age=19 where sno= ‘S27’
```

## 系统会自动把它转变为：

```
UPDATE S
SET age=19 where sno = ‘S27’ and
 sex= ‘男’
```

# 6. 视图的作用

1. 视图能够简化用户的操作
2. 视图使用户能以多种角度看待同一数据
3. 视图能够对机密数据提供安全保护
4. 适当的利用视图可以更清晰的表达查询

拓展阅读：Materialized View

# ★第三章小结★

- **SQL数据库的体系结构：** 支持数据库三级模式结构，其中外模式对应于视图和部分基本表，模式对应于基本表，内模式对应于存储文件。
- **SQL的数据定义：** 定义包括模式定义与删除、表定义与删除、视图和索引的定义与删除等。
- **SQL的数据查询：** 数据查询用SELECT语句实现，可以完成单表操作、连接查询和嵌套查询。
- **SQL的数据更新：** 包括数据增加、修改和删除。对基本表的增、删、改操作有可能破坏参照完整性规则。

# ★第三章小结★

- **视图：** 视图是一种虚表，不是物理存在，对视图的操作实际上对基本表操作，引入视图有利于应用程序独立性、数据一致性、数据库的安全性，还能简化用户查询操作。