数据库系统概念-上

第一章 介绍

数据库管理系统由一个互相关联的数据的集合和一组用以访问这些数据的程序组成。这个数据集合通常称作数据库，其中包含了某个企业的信息。DBMS的主要目标是提供一种可以方便、高效地存取数据困信息的途径。

1.1、数据库系统的应用

- 企业信息
  - 销售
  - 会计
  - 人力资源

    - 生产制造
    - 联机销售
- 银行和金融
    - 银行业
    - 信用卡交易
    - 金融业
- 大学
- 航空业
- 电信业

1.2、数据库系统的目标

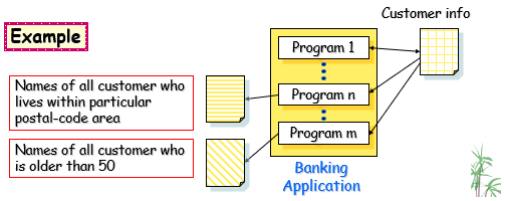
文件处理系统存储与操作信息的弊端：

- 数据的冗余和不一致`Data redundancy and inconsistency`：

  文件和程序由不同的程序员创建，不同文件有不同的结构，不同程序可能采用不同的程序设计语言。此外，相同的信息可能在几个文件中重复存储，这种冗余除了导致存储和访问开销增大外，还可能导致数据不一致。

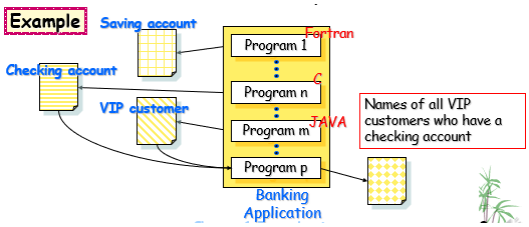
- 数据访问困难`Difficulty in accessing data`：

  Need to write a new program to carry out each new task.



- 数据孤立`Data isolation`：

  由于数据分散在不同的文件中，这些文件又可能具有不同的格式，编写新应用程序来检索适当数据是很困难的。



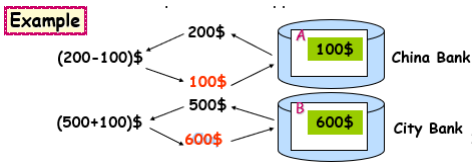
- 完整性问题`Integrity problems`：

  数据库中所存储数据的值必须满足某些特定的**一致性约束**。例如，某类银行账户的余额永远不能低于某个预定的值。开发者通过在各种不同应用程序中加入适当的代码来增强系统中的这些约束。当新的约束加入时，很难通过修改程序中加入适当的代码来增强系统中的这些约束。当约束设计不同文件中的多个数据项时，问题变得更加复杂。

- 原子性问题`Atomicity of updates`：

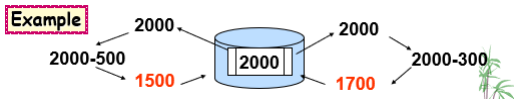
  Failures may leave data in an inconsistent state with partial updates carried out;

  Transfer of funds from one account to another should either complete or not happen at all.



- 多个用户的并发访问`Concurrent access by multiple users`：

为了提高系统的总体性能和加快响应速度，许多系统允许多个用户同时更新数据；并发的更新操作相互影响，可能导致数据的不一致。



- 安全性问题`Security problems`：

并非数据库系统的所有用户都可以访问所有数据。

1.3、数据视图

数据库系统时一些互相关联的数据以及一组使得用户可以访问和修改这些数据的程序的集合。数据库系统的一个主要目的是给用户提供数据的抽象视图，也就是手，系统隐藏关于数据存储和维护的某些细节。

1.3.1、 数据抽象

由于许多数据库系统的用户并为受过计算机的专业训练，系统开发人员通过如下几个层层次上的抽象来对用户屏蔽复杂性，以监护用户与系统的交互：

- 物理层：

最低层次的抽象，描述数据实际上是怎样存储的；物理层详细描述复杂的底层数据结构。

在物理层，记录可能被描述为连续存储单元组成的存储块。数据库系统为数据库程序设计人员屏蔽低层的存储细节。而数据库管理员可能需要了解数据物理组织的某些细节。

- 逻辑层：

比物理层层次稍高的抽象，描述数据库中存储什么数据及这些数据键存在什么关系。这样逻辑层就通过少量的相对简单的结构描述了整个数据库。

虽然逻辑层的简单结构的实现可能涉及浮渣的物理层结果，但逻辑层的用户不必知道这样的复杂性，这称作**物理数据独立性**`physical data independence`。应用程序如果不依赖于物理模式，就被称为是具有物理数据独立性，即使物理模式改变了，它们也无须重写。

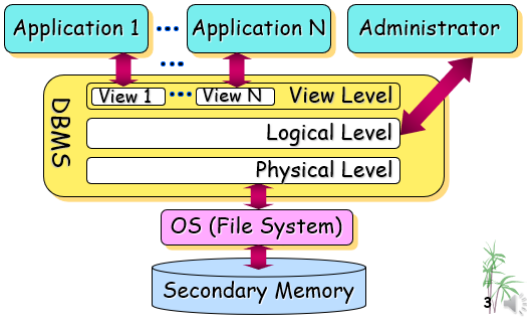
在逻辑层，记录通过类型定义描述，同时定义这些记录类型的相互关系。程序设计人员&DB管理员常基于此层次工作。

- 视图层：

最高层次的抽象，只描述整个数据库的某个部分。

数据库系统的很多用户并不需要关系所有的信息，而只需要访问数据库的一部分，系统可以为同一数据库提供多个视图。

在视图层，计算机用户看见的是对其屏蔽了数据类型细节的一组应用程序。此外，视图还提供了防止用户访问数据库的某些部分的安全性机制。



1.3.2、 实例和模式

- 模式 `Schema`：数据库的总体设计称作数据库模式。
- 实例 `Instance`：特定时刻存储在数据库中的信息的集合称作数据库的一个实例。

数据库模式和实例的概念可以通过与用程序设计云烟写出的程序进行类比来理解。数据库模式对应于程序设计语言中的变量声明（以及与之关联的类型的定义），每个变量在特定的时刻会有特定的值，程序中变量在某一时刻的值对应与数据库模式的一个实例。

根据不同的抽象层次，数据库系统可以分为几种不同模式：

- 物理模式：在物理层描述数据库的设计。
- 逻辑模式：在逻辑层描述数据库的设计。
- 子模式：在视图层描述了数据库的不同视图。

1.3.3、 数据模型 `Data Model`

数据库结构的基础是书籍模型。数据模型是一个描述数据、数据联系、数据语义以及一致性约束的概念工具的集合。数据模型提供了一种描述物理层、逻辑层以及视图层数据库设计的方式。数据模型可被划分为四类：

- 关系模型：关系模型用表的集合来表示数据和数据间的关系。每个表有多个列，每列有唯一的列名。关系模型是基于记录的模型的一种。基于记录的模型的名称的由来是因为数据库是由若干种固定格式的记录来构成的。每个表包含某种特定类型的记录，每个记录类型定义了固定数目的字段（或属性），表的列对应于记录类型的属性。

customer_id	customer_name	customer_street	customer_city
192-83-7465	Johnson	12 Alma St.	Palo Alto
677-89-9011	Hayes	3 Main St.	Harrison
182-73-6091	Turner	123 Putnam Ave.	Stamford
321-12-3123	Jones	100 Main St.	Harrison
336-66-9999	Lindsay	175 Park Ave.	Pittsfield
019-28-3746	Smith	72 North St.	Rye

(a) The *customer* table

account_number	balance
A-101	500
A-215	700
A-102	400
A-305	350
A-201	900
A-217	750
A-222	700

(b) The *account* table

customer_id	account_number
192-83-7465	A-101
192-83-7465	A-201
019-28-3746	A-215
677-89-9011	A-102
182-73-6091	A-305
321-12-3123	A-217
336-66-9999	A-222
019-28-3746	A-201

(c) The *depositor* table

- 实体-联系模型 `Entity-Relationship model`：实体-联系（E-R）数据模型基于对现实世界的这样一种认识——现实世界由一组称作实体的基本对象以及这些对象间的联系构成。实体是现实世界中可区别与其他对象的一件“事情”或一个“物体”。实体-联系模型被广泛用于数据库设计。
- 基于对象的数据模型 `Object-based data model`：面向对象的数据模型可以看成是E-R模型增加了封装、方法（函数）和对象表示等概念后的扩展。
- 对象-关系数据模型：结合了面向对象的数据模型和关系数据模型的特征。
- 半结构化数据模型 `Semistructured data model`：半结构化数据模型允许哪些相同类型的数据项含有不同的属性集数据定义（XML 可扩展性标记语言）。

- 网状数据模型`Network data model`和层次数据模型`Hierarchical data model`：与底层的实现联系紧密，并且是数据建模复杂化。

1.4、数据库语言

数据库系统提供**数据定义语言**`Data-Definition language`来定义数据库模式，以及**数据操纵语言**`Data-Manipulation language`来表达数据库的查询和更新。实际上，数据定义和数据操纵语言并不是两种分离的语言，相反，它们简单地构成了单一的数据库语言（如`SQL`）的不同部分。

1.4.1、 数据操纵语言`DML`

数据操作语言使得用户可以访问或操作那些按照某种适当的数据类型组织起来的数据，访问类型如上：

- 检索存储在数据库中的信息
- 插入新的信息到数据库
- 删除数据库的信息
- 修改数据库存储的信息

通常有两类基本的数据操纵语言：

- 过程化数据操作语言`Procedural DML`：要求用户指定需要什么数据以及如何获得这些数据。
- 声明式数据操作语言`Declarative DML`（非过程化）：要求用户指定需要什么数据，而不指明如何如何获得这些数据。

由于用户不必指明如何获得数据，数据库系统必须找出一种访问数据的高效途径。

查询`Query`是要求对信息进行检索的语句，***`DML`中涉及信息检索的部分称作查询语言`Query Language`***，`SQL`是使用最广泛的查询语言。

在物理层，必须定义可高效访问数据的算法；在更高的抽象层次，则强调易用性，目标是使人们能够更有效地和系统交互。数据库系统的查询处理器部件将`DML`的查询语句翻译成数据库物理层的动作序列。

1.4.2、数据定义语言

> 数据库模式是通过一系列定义来说明的，这些定义由一种称作数据定义语言的特殊语言来表达，`DDL`也可以用于定义数据的其它特征。

> 数据库系统所使用的存储结构和访问方式是通过一系列特殊的`DDL`语句来说明的，这种特殊的`DDL`称作数据存储和定义语言`Data storage and definition`语言，这些语句定义了数据库模式的实现细节。

存储在数据库中的数据值必须满足某些一致性约束`Consistency Constraint`，`DDL`语言提供了指定这种约束的工具。每当数据库更新时，数据库都会检查这些约束。通常，约束可以是关于数据库的任意谓词。然而，如果要测试任意谓词，可能代价比较高。因此，数据库系统实现可以以最小代价测试的完整型约束。

- 域约束`Domain Constraint`：

每个属性都必须对应于一个所有可能的取值构成的域（例如整数型、字符型、日期/时间型），声明一种属性属于某种具体的域就相当于约束它可以取得值。

- 参照完整性`Referencial Integrity`：

一个关系中给定属性集上的取值也在另一关系的某一属性值中出现。

- 断言`Assertion`：

一个断言就是数据库需要时刻满足的某一条件，域约束和参照性约束是断言的特殊形式。断言创建后，系统会检测其有效性，如果断言有效，则以后自由不破坏断言的数据库更新才被允许。

- 授权`Authorization`：不同的用户在数据库中不同数据值上允许不同的访问类型
    - 读权限
    - 插入权限
    - 更新权限
    - 删除权限

`DDL`以一些指令作为输入，生成一些输出，`DDL`的输出放在数据字典中，数据字典包含了元数据`metadata`，元数据是关于数据的数据。可以把数据字典看做一种特殊的表，这种表只能由数据库系统本身来访问和修改，在读取和修改实际的数据前，数据库系统先要参考数据字典。

1.5、关系数据库

关系数据库基于关系模型，使用一系列表来表达数据以及这些数据之间的联系。

1.5.1、表

记录、属性

1.5.2、数据操纵语言

`SQL`查询语言是非过程化的，它以几个表作为输入，总是仅返回一个表。

- SELECT table_name(.attribute_name) From table_name Where conditions;

1.5.3、数据定义语言

`SQL`提供了一个丰富的`DDL`语言，通过它，我们可以定义表、完整性约束、断言，等等。

- CREATE TABLE table_name(attribute_name data_type);

1.5.4、来自应用程序的数据库访问

为了访问数据库，`DML`语句需要由宿主语言来执行，有两种途径可以做到这一点：

- 通过提供应用程序接口（过程集），它可以用来将`DML`和`DDL`的语句发送给数据库，再取回结果。

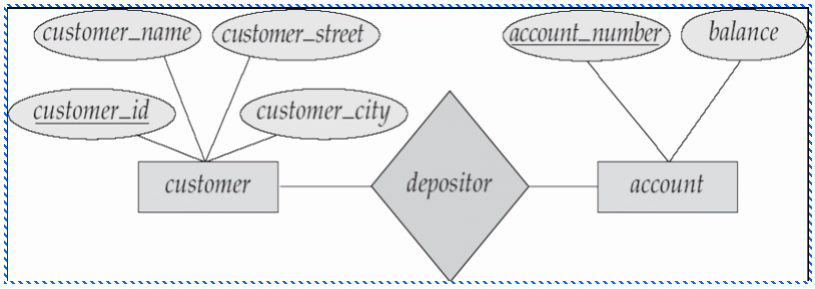
  与C语言一起使用的开放数据库连接`ODBC`，是一种常用的应用程序接口标准；Java数据库连接`JDBC`标准为Java语言提供了相应的特性。
- 通过扩展数宿主语言的语法，在宿主语言的程序中嵌入`DML`调用。通常用一个特殊字符作为`DML`调用的开始，并且通过预处理器，称为`DML`预编译器`Precompiler`，来将`DML`语句转变成宿主语言中的过程调用。

1.6、数据库设计

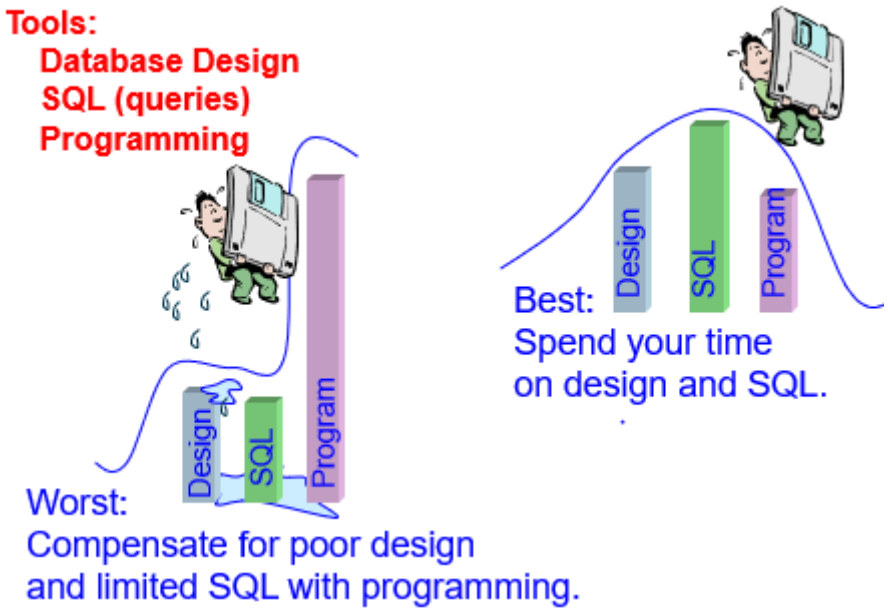
数据库系统被设计用来管理大量的信息，这些大量的信息不是孤立存在的；数据库设计的主要内容是数据库模式的设计

- 设计过程：高层的数据模型为数据库设计者提供了一个概念框架，区寿命数据库用户的数据需求，以及将怎样构造数据库结构以满足这些需求。
    - 初始阶段：全面刻画预期的数据库用户的数据需求，制定出用户需求的规格文档。
    - 选择数据模型：运用选定的数据模型的概念，转换需求为数据库的概念模式，这个**概念设计**阶段开发出来的模式提供了企业的详细概述；设计者再复审这个模式，确保所有的数据都满足并且相互之间没有冲突，这一阶段的重点是描述数据以及它们之间的联系，而不是指定物

理的存储细节；从关系模型的角度来看，概念设计阶段设计决定数据库中应该包括哪些属性，以及如何将这些属性组织到多个表中（主要方法：使用实体-联系模型或引入规范化算法）；一个开发完全的概念模式还将指出企业的功能需求，在功能需求说明中，用户描述数据之上的各种操作（或事物），例如更新数据、检索特定的数据、删除数据等。
    - **逻辑设计阶段** `Logical-design phrase`：将高层的概念模式映射到要使用的数据库系统的实现数据模型上。
    - **物理设计阶段** `Physical-design phrase`：指定数据库的物理特性（文件组织的形式以及内部的存储结构）
- 实体-联系模型
    - E-R数据模型使用一组称作实体的基本对象，以及这些对象间的联系；
    - 数据库中实体通过属性集合来描述；
    - 联系是几个实体之间的关联；
    - **Entity Set**: the set of all entities of the same type
    - **Relationship Set**: the set of all relationships of the same type
    - **E-R Diagram**：数据库的总体逻辑结构（模式）可以用实体-联系图进行图形化描述，`UML` 中，E-R图的实体集用矩形框表示，联系集用连接一对相关的实体集的菱形表示（联系名置于菱形内部），注意实体属性置于椭圆框中。



    - 除了实体和联系外，E-R模型还描绘了数据看必须遵守的对其内容的某些约束，一个重要的约束是**映射基数** `Mapping cardinality`，它表示某个联系集能与一实体进行关联的实体数目。
- 规范化：
    - 目标：生成一个关系模式集合，使我们存储信息是没有不必要的冗余，同时又能轻易的检索数据。



1.7、数据库用户和管理员

数据库系统的一个主要目标是从数据库中检索信息和往数据库中存储新信息，使用数据库的人员分为数据库用户和数据库管理员。

1.7.1、 数据库用户和用户界面

根据与系统交互方式的不同，数据库系统的用户可以方位四种不同类型，系统为不同类型的用户设计了不同类型的用户界面。

- 无经验的用户`**Naive users**`：通过激活事先已经写好的应用程序同系统进行交互。
- 应用程序员`**Application programmers**`：编写应用程序的计算机专业人员；快速应用开发`Rapid Application Development,RAD`。
- 老练的用户`**Sophisticated users**`：不通过编写程序来与系统交互，而是用数据库查询语言或数据分析工具来表达他们的要求。
- 专业在的用户：编写专门的、不适合于传统数据处理模式的数据库应用的富有经验的用户。这样的应用包括：计算机辅助设计系统，知识库和专家系统，存储复杂结构数据的系统，以及环境建模系统。

1.7.2、 数据库管理员

使用DBMS的一个主要原因是可以对数据和访问这些数据的程序进行集中控制，对系统进行集中控制的人称作**数据库管理员`DataBase Administrator,DBA`，`DBA`的作用包括：

- 模式定义`Schema definition`
- 存储结构及存取方式定义`Storage Structure`
- 模式及物理组织的修改`Schema and physical-organization modification`
- 数据访问授权`Granting of authorization for data access`
- 定义完整性约束`Specifying integrity constraints`
- 与用户进行联络`Acting as liaison with users`
- 日常维护`Routine maintenance`
    - 数据备份
    - 确保正常运转时所需的空域磁盘空间，并且在需要时升级磁盘空间
    - 监视数据库的运行及性能，并根据需求进行变更`Responding to changes in requirements`

1.8、 数据存储和查询

数据库系统划分为不同的模块，每个模块完成整个系统的一个功能，数据库系统的功能部件大致可分为存储管理`Storage Management`器和查询处理`Query Processing`部件。

1.8.1、 存储管理器

存储管理器是数据库系统负责在数据库中存储的低层数据与应用程序以及向系统提交的查询之间提供接口的程序模块或部件。存储管理器负责与文件管理器进行交互（原始数据通过文件系统存储在磁盘上，文件系统通常由传统的操作系统提供，存储管理器将不同的`DML`语句翻译为底层文件命令），负责数据库中的数据的存储、检索和更新。

存储管理器部件包括：

- 权限及完整性管理器`Authorization and integrity manager`：
    - 检测是否满足完整性约束；
    - 检查试图访问数据的用户的权限；
- 事务管理器`Transaction manager`：

    - 保证即使发生故障，数据库也保持一致的状态；
    - 保证并发事务的执行不发生冲突；
- 文件管理器`File manager`：
    - 管理磁盘空间的分配；
    - 管理用于表示磁盘上所存储信息的数据结构；
- 缓冲管理器`Buffer manager`

  （数据库系统的关键部分，使数据库可以处理比内存更大的数据）：
    - 负责将数据从磁盘上去到内存中，并决定哪些数据硬背缓冲存储到内存中；

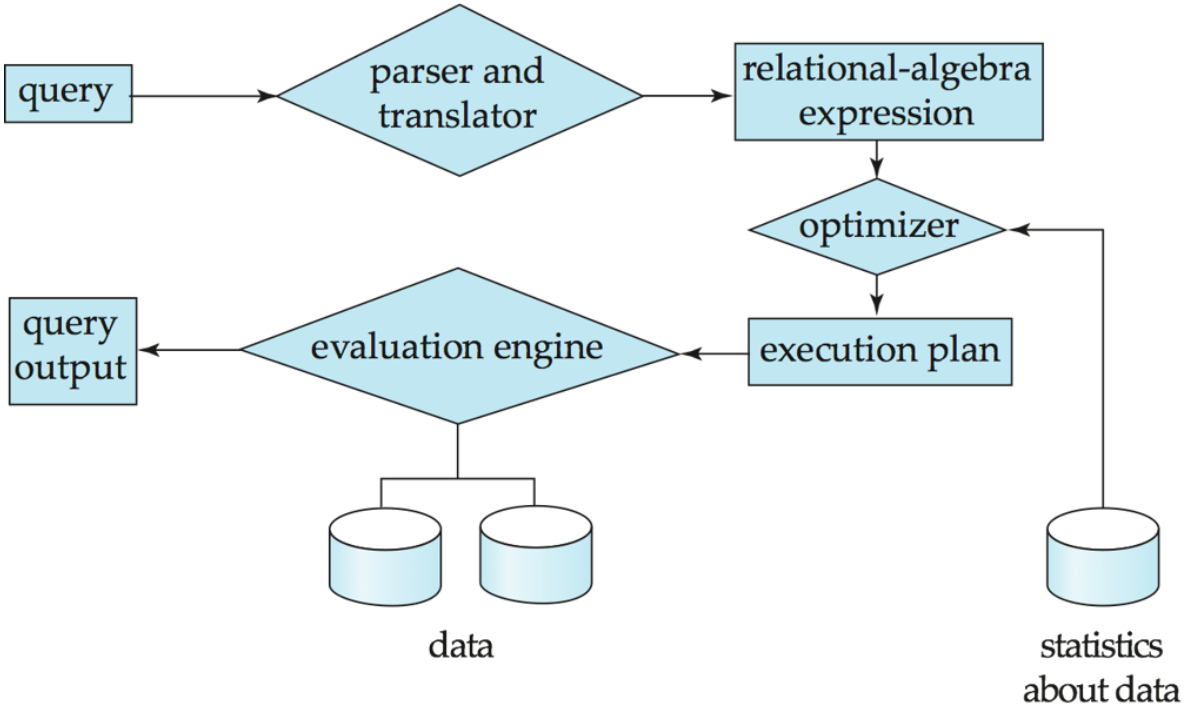
存储管理器实现了几种数据结构：

- 数据文件，存储数据库自身
- 数据字典，存储关于数据库结构的元数据，尤其是数据库模式
- 索引`Index`，提供对数据项的快速访问

1.8.2、查询处理器

查询处理器包括：

- `DDL`解释器，解释`DDL`语句并将这些定义记录在数据字典中
- `DML`编译器，将查询语言中`DML`语句翻译为一个计算方案，包括一系列查询计算引擎能理解的低级指令
- 查询计算引擎，执行由`DML`编译器产生的低级指令

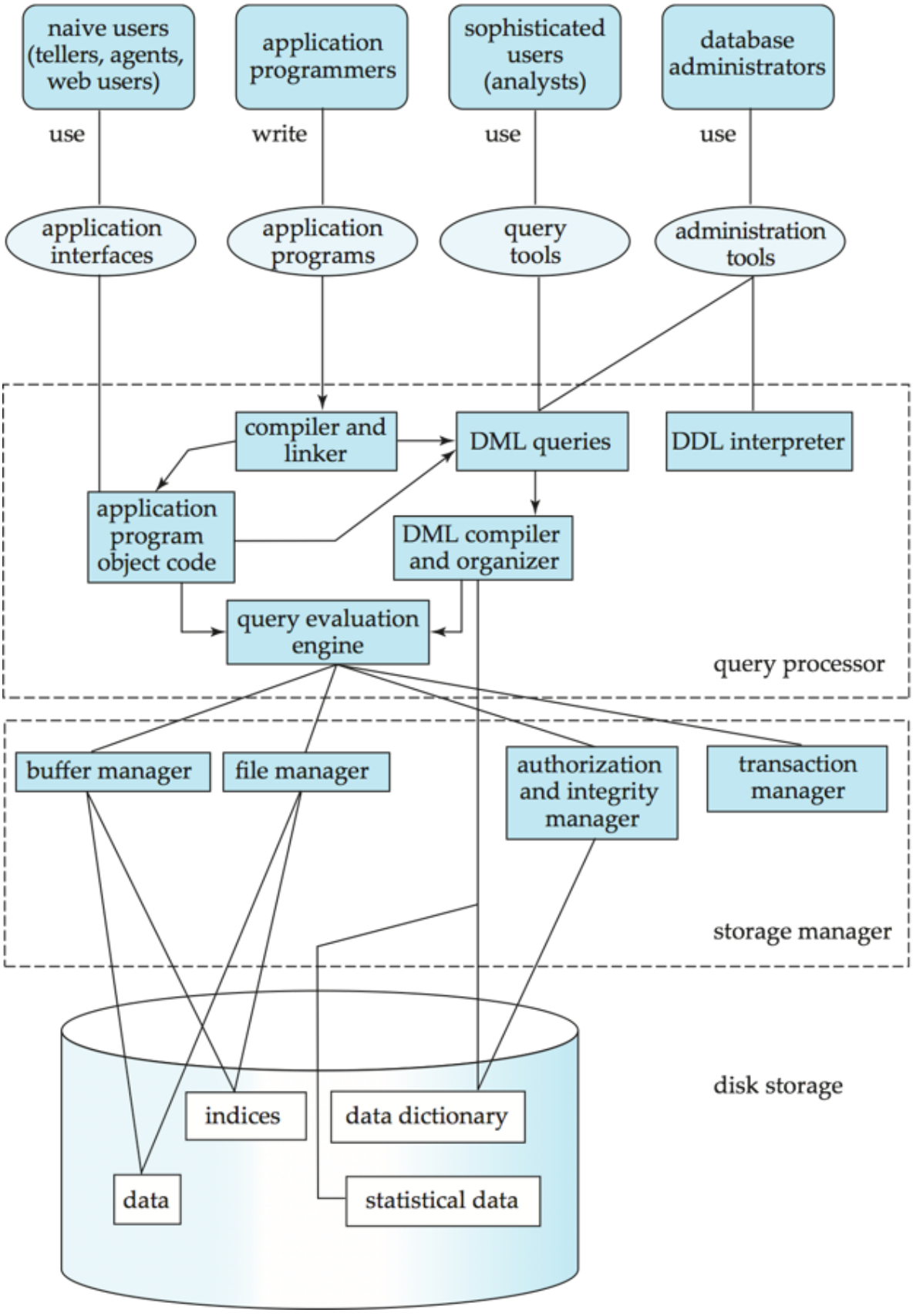


1.9、事物管理`Transaction Management`

- 事务是数据库应用中完成单一逻辑功能的操作集合，是一个既具有原子性又具有一致性的单元
- 事物管理器`Transaction Manager`包括：
    - 并发控制管理器`**Concurrency-control manager**`：控制并发事务间的相互影响，保证数据库的一致性；
    - 恢复管理器`Recovery maneger`：保证原子性和持久性（即使发生了故障和失败）

1.10、数据库体系结构`Database Architecture`

Database System Internals

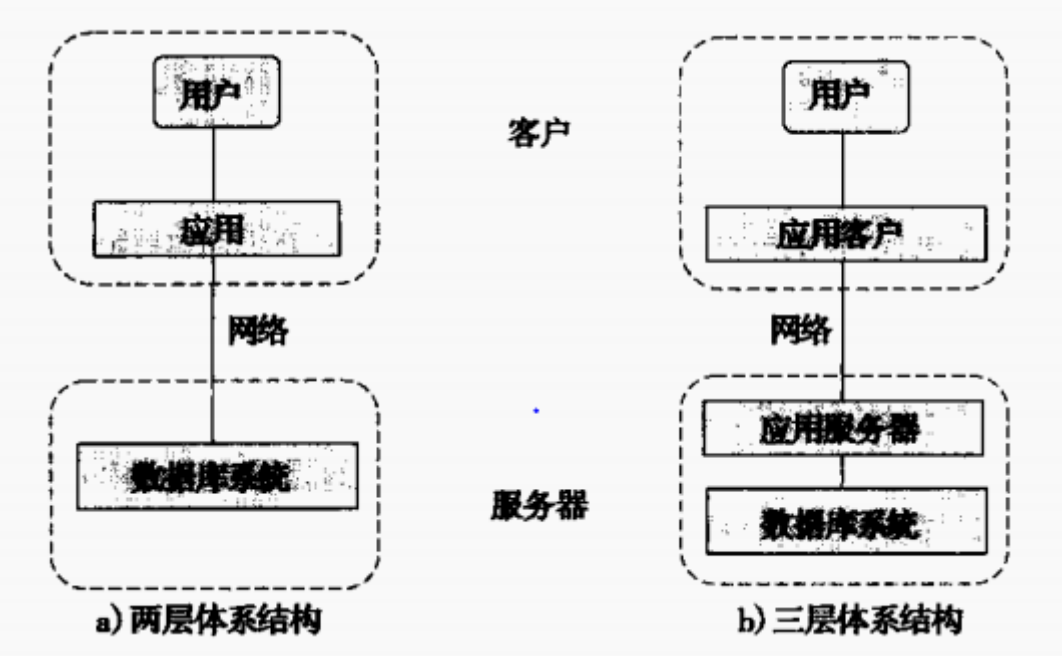


数据库体系结构很大程度上取决于数据库系统所运行的计算机系统，可以是：

- Centralized 集中式
- Client-server 客户-服务器式
- Parallel (multi-processor) 并行
- Distributed 分布式：包含地理上分离的多台计算机

数据库应用通常可分为两或三个部分：

- 在一个两层体系结构中，应用程序驻留在客户机上，通过查询语言表达式来调用服务器上的数据库系统功能，像`ODBC`和`JDBC`这样的应用程序接口标准被用于进行客户端和服务器的交互。
- 在一个三层体系结构中，客户机只作为一个前端并且不包含任何直接的数据库调用。客户端通常通过一个表单界面与应用服务器`Application server`进行通信，而应用服务器与数据库系统通信以访问数据。应用程序的业务逻辑，也就是说在何种条件下做出何种反应，被嵌入到应用服务器中，而不是分布在多个客户机上。三层结构的应用更适合大型应用和互联网上的应用。



1.11、数据库系统历史（略）

第二章 关系模型介绍

2.1、关系数据库的结构

- 关系数据库由表`Table`的集合构成，每个表有唯一的名字。
- 一般来说，表中一行代表了一组值之间的一种联系；由于一个表就是这种联系的一个集合，表这个概念和数学上的**关系**这个概念密切相关，而这正是关系数据模型名称的由来。
  - 关系定义：给定集合(属性域)$D_1, D_2, D_3, \ldots, D_n$，则关系就是$D_1 \times D_2 \times D_3 \times \ldots \times D_n$的一个子集。
- 数学术语中，**元祖**`tuple`只是一组值的序列（或列表）；在n个值之间的一种联系可以在数学上用关于这些值的一个n元祖来表示，换言之，它对应与表中的一行。
- 因此，在关系模型术语中，***关系***`Relation`**用来指代*表*，而*元祖*用来指代*行*，类似地属性指代的是表中的列。**
- 关系实例`Relation instance`：关系的特定实例，即所包含的一组特定的行。
- 对于关系的每个属性，都存在一个允许取值的集合，称为该属性的*域*`Domain`。
- 要求对所有关系r而言，r的所有属性的域都是原子的`atomic`。
- *空*`null`*值*是一个特殊的值，表示值未知或不存在

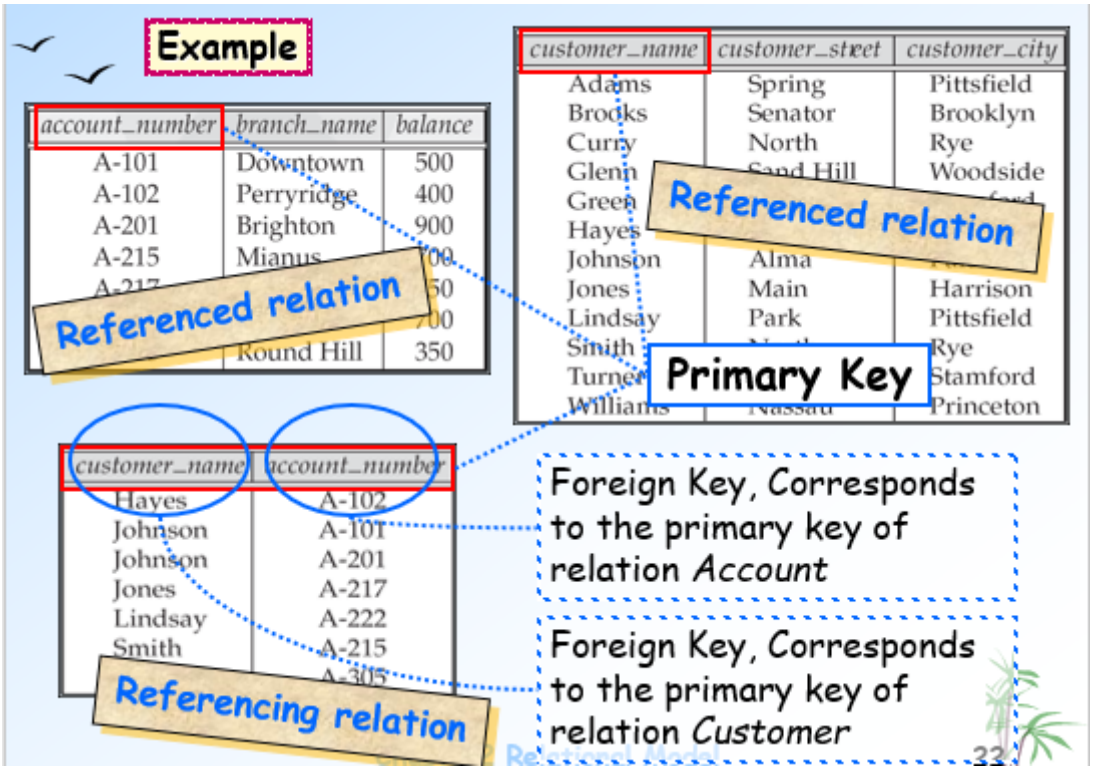
- 空值给数据库访问和1更新带来很多困难，因此因避免使用空值。

2.2、数据库模式

- 区分数据库模式和数据库实例
  - 前者是数据库的逻辑设计，后者是给定时刻数据库数据的一个快照
- 关系模式`Relation schema`的组成
  - 属性序列
  - 各属性对应域
- 关系模式------类型；关系------变量；关系实例------变量的值
- $R = (A_1, A_2, \ldots, A_n)$ is a **relation schema**，r(R) denotes a relation r on the relation schema R，relation instance of a relation are specified by a table.
- 如何把不同关系的元祖联系在一起？在关系模式中使用相同属性

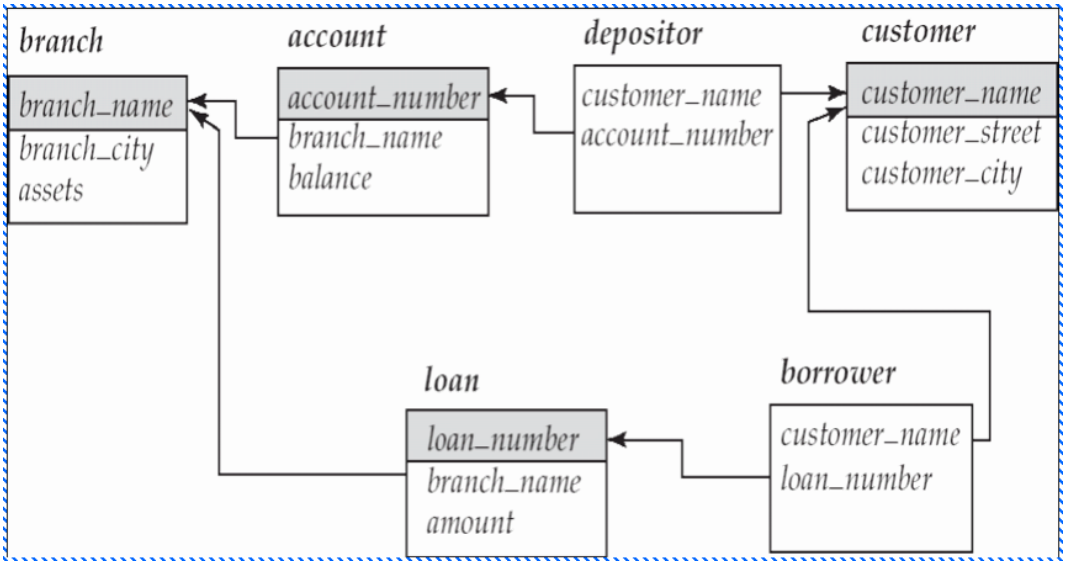
2.3、码

- 区分给定关系中的不同元祖的方法：通过属性表明——一个元祖的属性值必须能够唯一区分元祖
- ***超码*** `Superkey`：一个或多个属性的集合，使我们在一个关系中唯一的表示一个元祖
  - 形式化描述：
    - R表示关系r模式的属性集合
    - K是R的一个子集且是r的一个超码
    - 则关系r中任意两个不同元祖不会在K的所有属性上取值完全相等，即如果t_1与t_2在r中且$t_1 != t_2$，则$t_1.K$!=$t_2.K$
  - 超码可能包含无关重要的属性；如果K是一个超码，那么K的任意超集也是超码。
- ***候选码*** `candidate key`：任意真子集均不能成为超码的超码
  - 存在几个不同的属性集做候选码的情况
- ***主码*** `primary key`：被数据库设计者选中的、主要用来在一个关系中区分不同元祖的候选码。
  - 主码应该选择那些从不或极少变化的属性
  - 关系模式的主码属性一般列在其它属性前面
- **注意**：码是整个关系的一种性质，而不是单个元祖的性质
- 码的指定代表了被建模事物在现实世界中的约束
- 一个关系模式r_1可能在它的属性中包括另一个关系模式r_2的主码，则这个属性在r_1上称作参照r_2的***外码*** `foreign key`、关系r_1称为外码依赖的参照关系`referencing relation`、r_2称作外码的被参照关系`referenced relation`
  - 参照完整性约束`referential integrity constraint`：在参照关系中**任意**元祖在特定属性上的取值必然等于被参照关系中**某个**元祖在特定属性上的取值。
- Example:

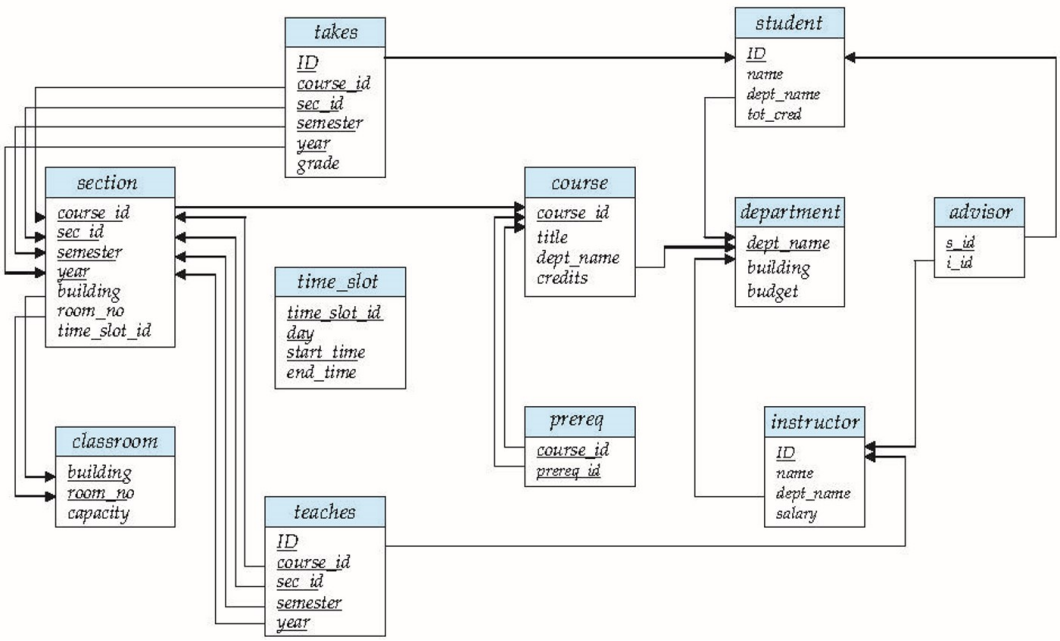


2.4、模式图

- 一个含有主码和外码依赖的数据库模式可以用*模式图* `Schema diagram` 来表示。
- 模式图：
  - 每一个关系用一个矩形表示；
  - 关系的名字显示在矩形上方，矩形内列出各属性，主码属性用下划线标注；
  - 外码依赖用从参照关系的外码属性到被参照关系的主码属性之间的箭头来表示；
  - 不表示参照完整性约束；
  - 银行数据库模式图



  - 大学数据库模式图：



2.5、关系查询语言

- 查询语言`Query language`：用户用来从数据库中请求获取信息的语言
- 分类：
  - 过程化语言`procedural language`：用户指导系统对数据执行一系列操作以计算出所需结果
  - 非过程化语言`nonprocedural language`：用户只需描述所需信息，而不用给出获取该信息的具体过程。
- 实际使用的查询语言既包含过程化方式的成分，又包含非过程化方式的成分。
- “纯”查询语言：
  - 关系代数`Relational algebra`：过程化
    - 包括一个运算的集合，运算以一个或两个关系为输入，产生一个新的关系作为结果
  - 元祖关系演算`Tuple relational calculus`：非过程化
  - 域关系演算`Domain relational calculus`：非过程化
  - 特点：
    - 语言简洁且形式化，默认商用语言的“句法修饰”，说明了从数据库中提取数据的基本技术;

2.6、关系运算

- 所有的过程化关系查询语言都提供了一组运算（施加于一个或一对关系上），运算结果总是单个的关系。
- 基本关系运算
  - 选择运算`Select Opration`：选择满足特定谓词的行
    - Define：

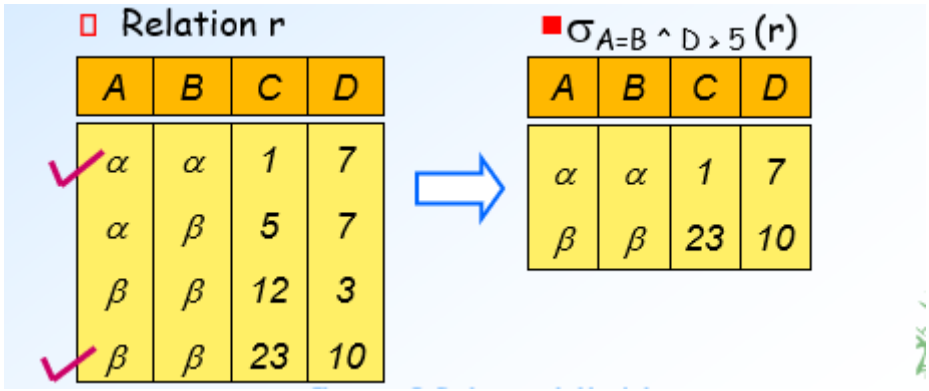
Select Operation Cont.

- Notation: $\sigma_p(r)$ — selection predicate
- Defined as:

$$\sigma_p(r) = \{t \mid t \in r \text{ and } p(t)\}$$

Where p is a formula in propositional calculus(命题演算) consisting of **terms** connected by : ∧ (**and**), ∨ (**or**), ¬ (**not**), Each **term** is one of:
<attribute> *op* <attribute> or <constant>
where *op* is one of: =, ≠, >, ≥. <. ≤

- Example:

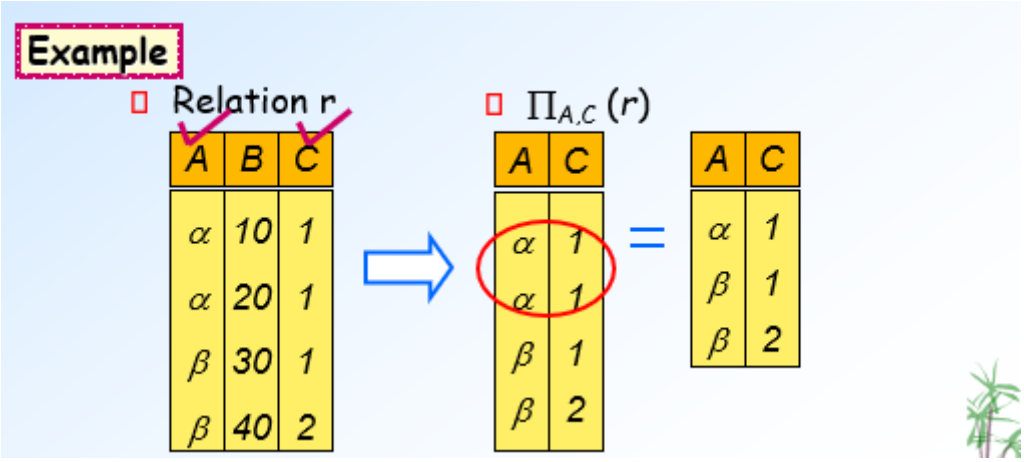


- 投影运算`Projec Operation`：选择特定的列
  - Define:

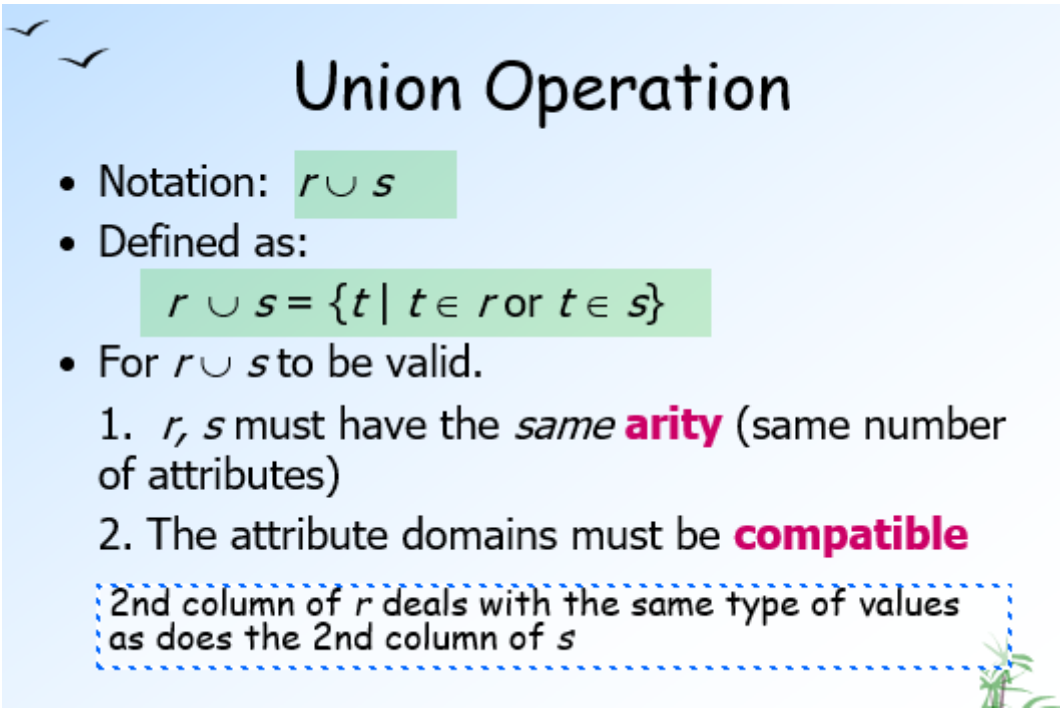
Project Operation Cont.

- Notation: $\prod_{A_1, A_2, \ldots, A_k}(r)$ — attribute names
- The result is defined as the relation of *k* columns obtained by erasing the columns that are not listed
- Duplicate rows removed from result, since relations are sets

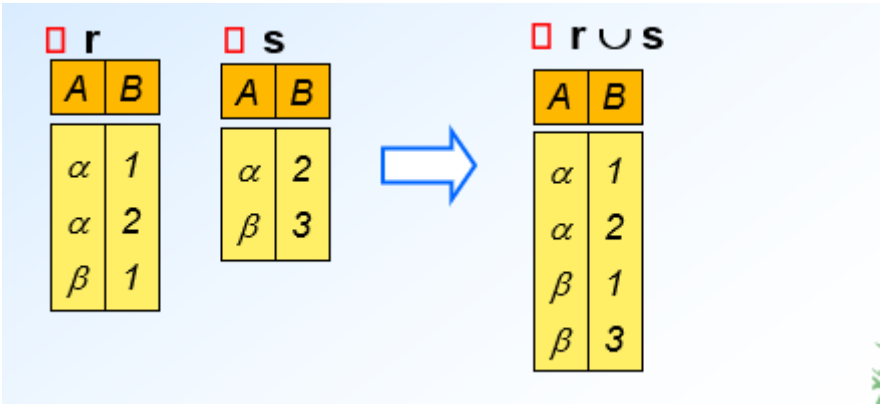
- Example:



- 并运算`Union Operation`：与并集运算类似
  - Define:



  - Example:



- 集合差运算`Set Difference Operation`：与差集类似
  - Define:

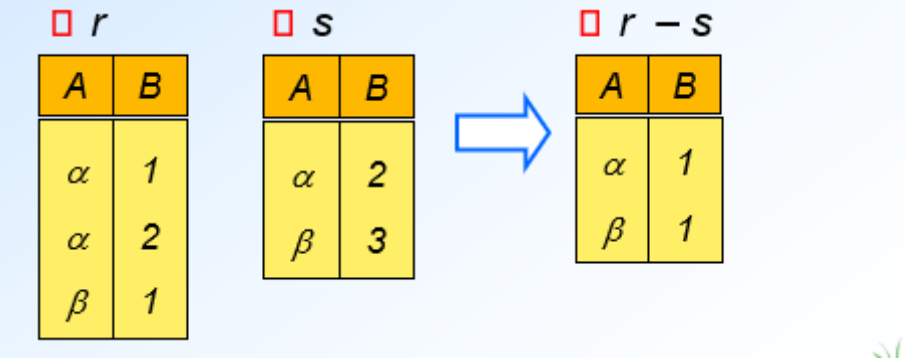
Set Difference Operation

- Notation $r - s$
- Defined as:

$$r - s = \{t \mid t \in r \text{ and } t \notin s\}$$

- Set differences must be taken between **compatible** relations.
  - r and s must have the same arity
  - attribute domains of r and s must be compatible

- Example:



- 笛卡尔积运算`Cartesian-Product Opeartion`:
  - Define: Pair each tuple of one relation with each tuple of another

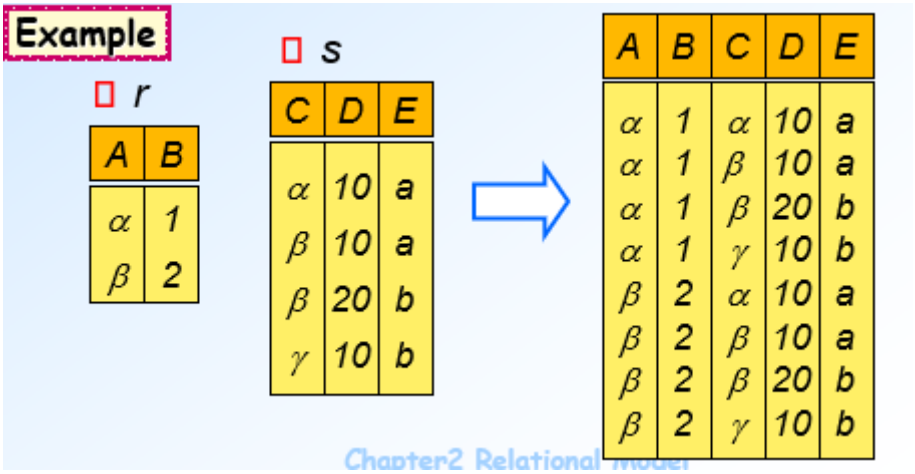
Cartesian-Product Operation

- Notation $r \times s$
- Defined as:

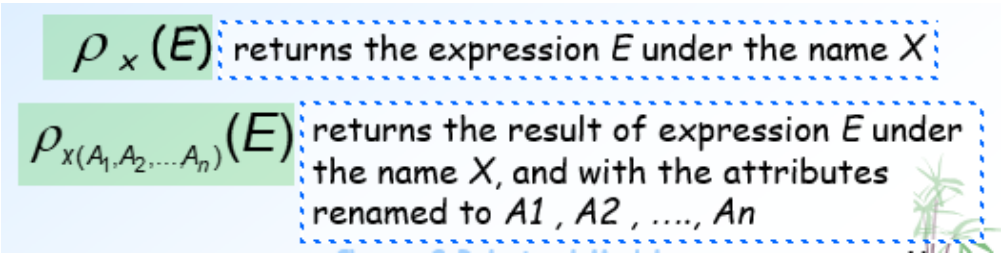
$$r \times s = \{t\, q \mid t \in r \text{ and } q \in s\}$$

- Assume that attributes of r(R) and s(S) are disjoint. (That is, $R \cap S = \varnothing$).
- If attributes of $r(R)$ and $s(S)$ are not disjoint, then renaming must be used

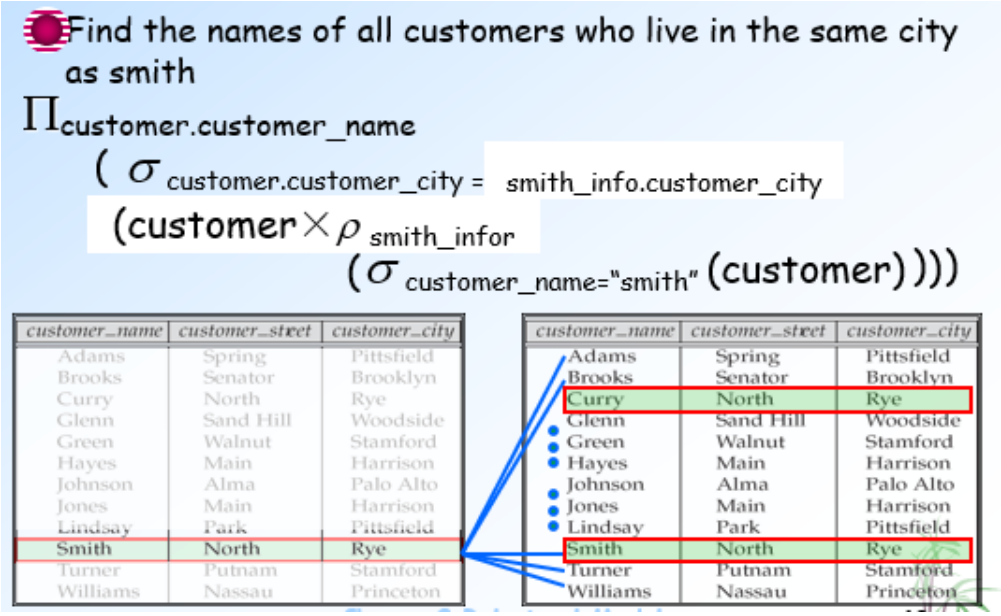
  - Example:



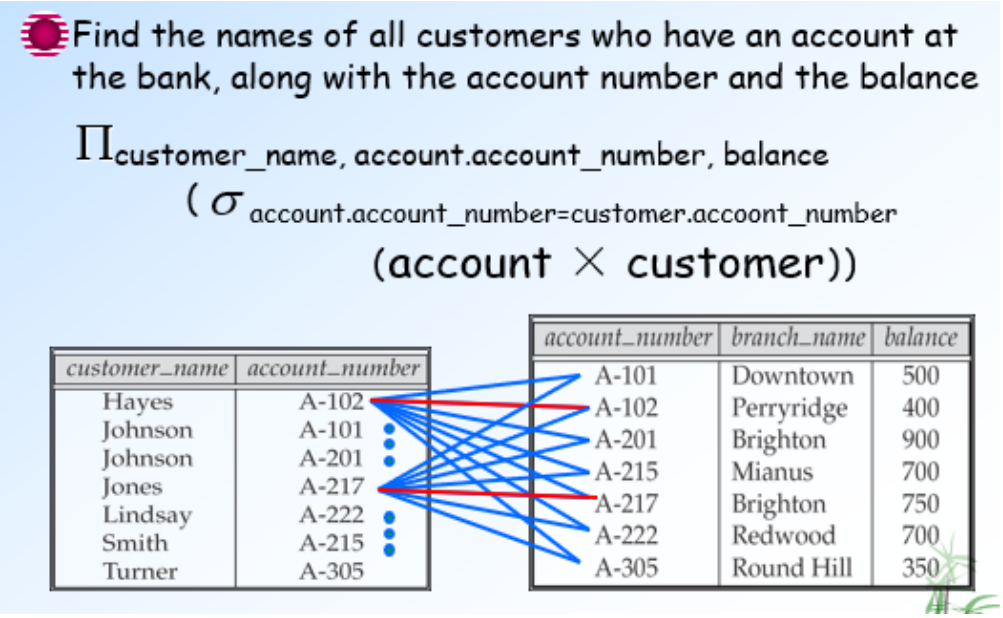
- 重命名运算:
  - Allow us to name , and therefore to refer to the results of relational-algebra expressions
    - Allow us to refer to a relation by more than one name



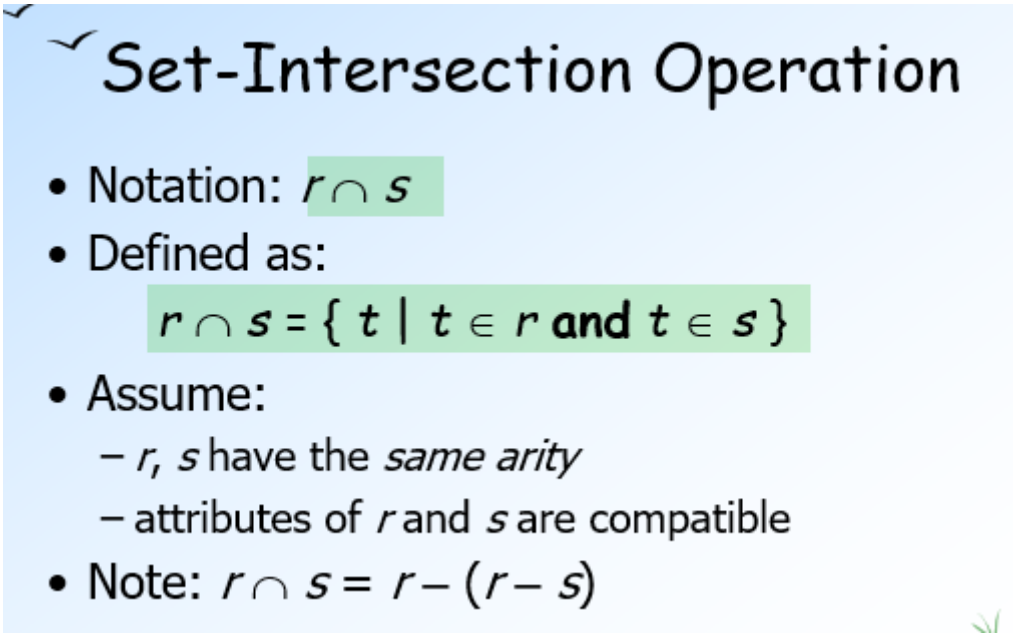
    - Example



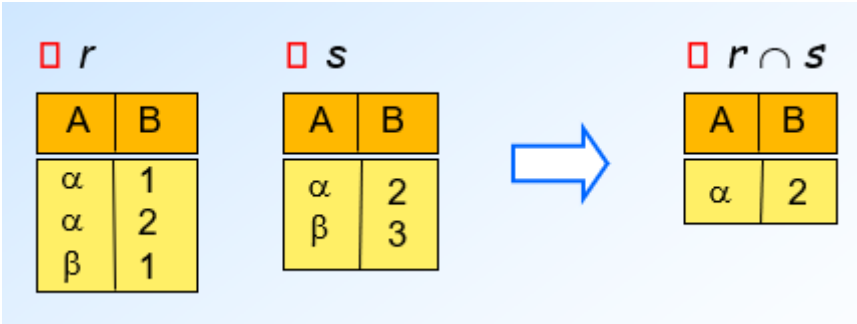
- 组合运算`Composition of Operations`***:
  - Define: Build expressions using multiple operations.
  - Example



- Additional Operations: that don't add any power to the relational algebra, but that simplify common queries.
  - Set intersection:
    - 定义：



    - Example:

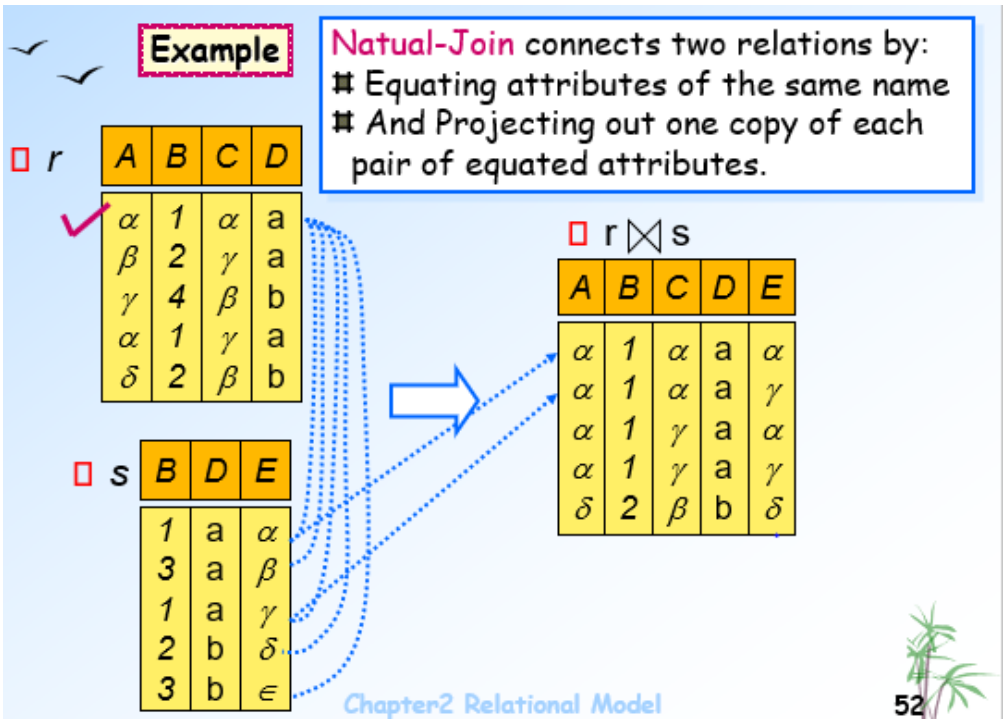


  - Natural Join:
    - 通常来说，两个关系上的自然连接运算所匹配的元祖在两个关系共用的所有属性上取值相同。
    - Define:

Natural-Join Operation

- Notation: r ⋈ s
- Let *r* and *s* be relations on schemas *R* and *S* respectively.
  Then, r ⋈ s is a relation on schema $R \cup S$ obtained as follows:
  - Consider each pair of tuples t_r from *r* and t_s from *s*.
  - If t_r and t_s have the same value on each of the attributes in $R \cap S$, add a tuple *t* to the result

- Example:



- Division:
  - Define:

Division Operation

- Notation: $r \div s$
- Let r and s be relations on schemas R and S respectively, where $R = (A_1, \ldots, A_m, B_1, \ldots, B_n)$ $S = (B_1, \ldots, B_n)$

  The result of r ÷ s is a relation on schema

  $R - S = (A_1, \ldots, A_m)$

$$r \div s = \{ t \mid t \in \Pi_{R-S}(r) \wedge \forall u \in s(tu \in r) \}$$

Where tu means the concatenation of tuples t and u to produce a single tuple

- Example:

r

A	B
α	1
α	2
α	3
β	1
γ	1
δ	1
δ	3
δ	4
∈	6
∈	1
β	2

s

B
1
2

r ÷ s

A
α
β

r

A	B	C	D	E
α	a	α	a	1
α	a	γ	a	1
α	a	γ	b	1
β	a	γ	a	1
β	a	γ	b	3
γ	a	γ	a	1
γ	a	γ	b	1
γ	a	β	b	1

s

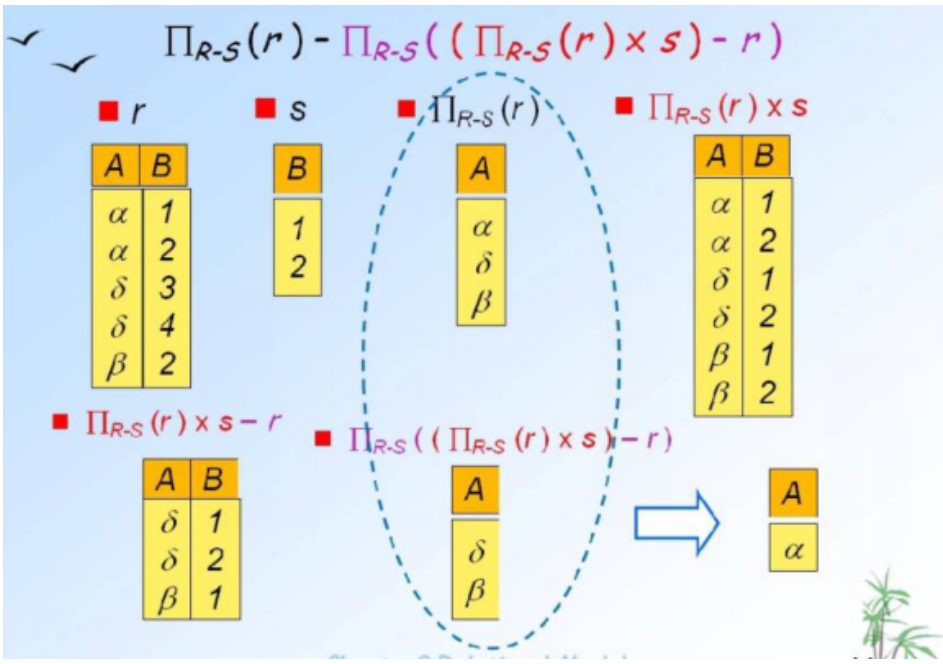
D	E
a	1
b	1

r ÷ s

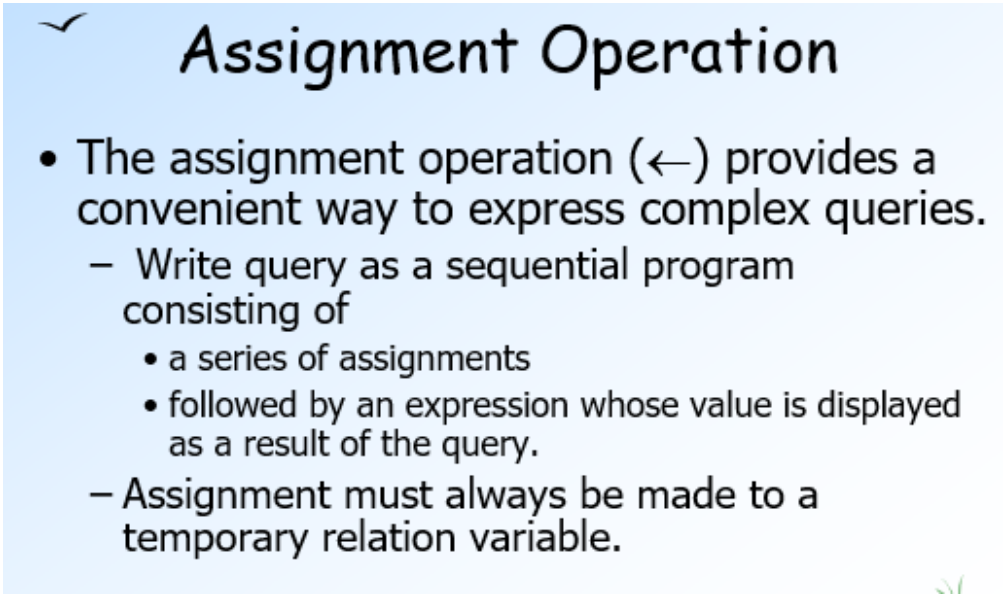
A	B	C
α	a	γ
γ	a	γ

- Division Operation ***

- Definition in terms of the basic algebra operation
  Let $r(R)$ and $s(S)$ be relations, and let $S \subseteq R$
  $r \div s =$
  $$\Pi_{R-S}(r) - \Pi_{R-S}((\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$$
  To see why
  - $\Pi_{R-S,S}(r)$ simply reorders attributes of r
  - $\Pi_{R-S}(\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$ gives those tuples t in $\Pi_{R-S}(r)$ such that for some tuple $u \in s$, $tu \notin r$



  - Assignment:
    - Define:



- Extended Operations:
  - Generalized Projection (广义投影)
    - Define:

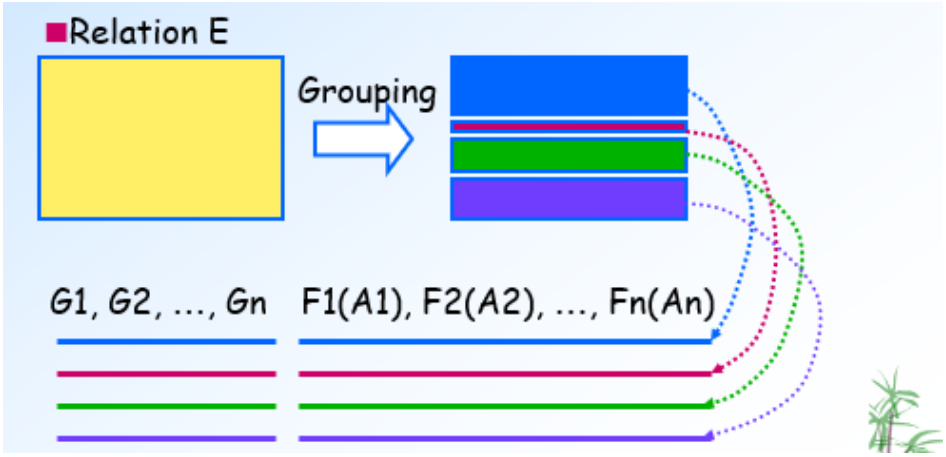
Generalized Projection

- Extends the projection operation by allowing arithmetic functions to be used in the projection list

$$\Pi_{F_1, F_2, \ldots, F_n}(E)$$

- E is any relational-algebra expression
- Each of F_1, F_2, …, F_n are arithmetic expressions involving constants and attributes in the schema of E.

- Aggregate Function
  - Takes a collection of values and returns a single value as a result
  - avg: average value
    - min: minimum value
    - max: maximum value
    - sum: sum of values
    - count: number of values
  - 表达式:
    - $\vartheta_{F_1(A_1),F_2(A_2),\ldots,F_n(A_n)}(E)$ F refer to function
    - $_{G_1,G_2,\ldots,G_n}\vartheta_{F_1(A_1),F_2(A_2),\ldots,F_n(A_n)}(E)$



  - Example:

branch_name	account_number	balance
Perryridge	A-102	400
Perryridge	A-201	900
Brighton	A-217	750
Brighton	A-215	750
Redwood	A-222	700

$branch\_name\ \mathcal{G}_{\mathbf{sum}(balance)}\ (account)$

branch_name	sum(balance)
Perryridge	1300
Brighton	1500
Redwood	700

- Outer Join:
  - An extension of the join operation that avoids loss of information
  - Define: Compute the join and then adds tuples form one relation that does not match tuples in the other relation to the result of the join.
  - Example:

Relation *loan*

loan_number	branch_name	amount
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

Relation *borrower*

customer_name	loan_number
Jones	L-170
Smith	L-230
Hayes	L-155

    - Join (Common):

Join

loan ⋈ *borrower*

loan_number	branch_name	amount	customer_name
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

    - Left Outer Join :

Left Outer Join

loan ⟕ *borrower*

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	*null*

- Right Outer Join:

Right Outer Join

loan ⟖ *borrower*

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	*null*	*null*	Hayes

- Full Outer Join：

Full Outer Join

loan ⟗ *borrower*

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	*null*
L-155	*null*	*null*	Hayes

- Null Values：
  - Signifies an unknown value or that a value does not exist
  - The results of any arithmetic expression involving null is null;
  - Aggregate functions simply ignore null values;
  - For duplicate elimination and grouping, null is treated like any other value, and two nulls are are assumed to be the same;
  - Comparisons with null values return the special truth value: *unknown*;
  - Result of select predicate (选择谓词) is treated as *false* if it evaluates to *unknown*;
  - Three-valued logic using the truth value *unknown*:
    - Or:
      - unknown or true = true
      - unknown or false = unknown
      - unknown or unknown = unknown
    - AND:
      - true and unknown = unknown
      - false and unknown = false
      - unknown and unknown = unknown
    - NOT:
      - not unknown = unknown
    - In `SQL` "P is unknown" evaluates to true if predicate P evaluates to unknown

2.7、Modification of Database

- The content of the database may be modified using the following operations:
  - Deletion:
    - A delete request is expressed similarly to a query, except instead of displaying tuples to the user, the selected tuples are removed from the database
    - Can delete only whole tuples, cannot delete values on only particular attributes
    - A deletion is expressed in relational algebra by: $r \leftarrow r - E$, E is relational algebra query
    - Example:
      - Delete all accounts at branches located in Needham
      - $r_1 \leftarrow \sigma_{branch-city="Needham"}(account \bowtie branch)$

        $r_2 \leftarrow \Pi_{account-number,branch-name,balance}(r_1)$

        $account \leftarrow account - r_2$

        $r_3 \leftarrow \Pi_{customer-name,account-number}(r_2 \bowtie depositor)$

        $depositor \leftarrow depositor - r_3$
- Insertion:
  - To insert data into a relation, we either:
    - specify a tuple to be inserted;
    - write a query whose result is a set of tuples to be inserted;
  - A insertion is expressed in relational algebra by: $r \leftarrow r \cup E$, E is relational algebra query.
  - The insertion of a single tuple is expressed by letting *E* be a constant relation containing one tuple.
  - Example:
    - Provide as a gift for all loan customers in the $Perryrridge$ branch, a $200 savings account. Let the loan number serve as the account number for the new savings account.
      - $r_1 \leftarrow (\sigma_{branch-name="Perryridge"}(borrower \bowtie loan))$

        $account \leftarrow account \cup \Pi_{loan-number,branch-name,200}(r_1)$

        $depositor \leftarrow depositor \cup \Pi_{customer-name,loan-number}(r_1)$
  - Updating:
    - A mechanism to change a value in a tuple without charging *all* values in the tuple.
    - Use the generalized projection operator to do this task: $r \leftarrow \Pi_{F_1,F_2,...,F_n}(r)$
      - Each F_i is
        - the I^{th} attribute of *r*, If the I^{th} attribute is not to be updated ;
        - an expression involving only constants and the attributes of *r*, which gives the new value for the attribute, if the attribute is to be updated.
    - Example :
      - Pay all accounts 5 percent interest
        - $account \leftarrow \Pi_{account-number,branch-name,balance*1.05}(amount)$
      - Pay all accounts with balances over $10,000 6 percent interest and pay all others 5 percent

- $account \leftarrow \Pi_{account-number,branch-name,balance*1.05}(\sigma_{balance>10000}(account)) \cup \Pi_{account-number,branch-name,balance*1.05}(\sigma_{balance>10000}(account))$