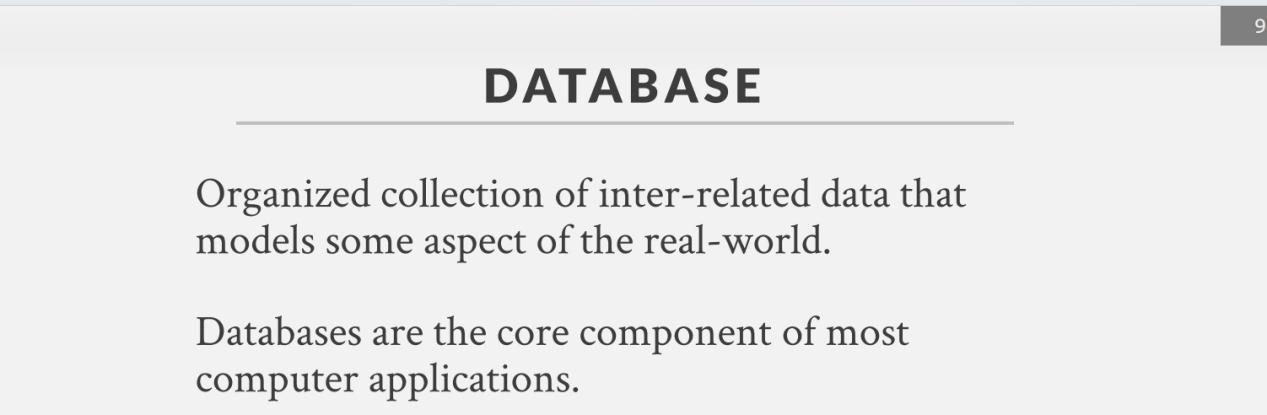
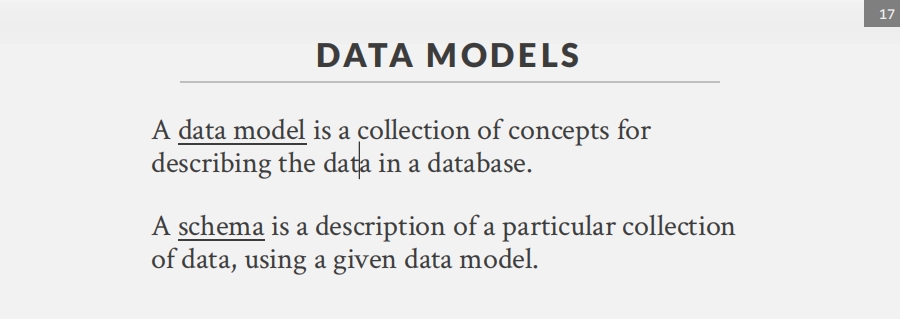
‘ ’ 

进入数据库之前，我们先来进行对于数据库的简单了解。

数据库是一个有组织的数据集合。这些数据之间以某种方式进行联系，并且这些数据我们希望它们基于一种现实世界的某些方面进行建模。也就是说这些数据一般都是我们现实生活中一些数据的映射。也就是说一个数据库应该捕获显示世界中的约束和各个方面。

需要注意的是，数据库是一个广泛的概念，广泛分布于各个领域中，我们这里来进行主要了解的是计算机领域方面的数据库概念，当然，这个概念也是可以广泛套用到各个领域的数据库中去的。



接下来先来了解下数据库的模式。首先来看”**模型**”，这是一种比较高级的抽象概念。这个模型可以用来表示数据库中的数据分布方式。一般来说，一个数据库中只支持一种模型，就比如层次模型，网状模型，关系模型等。

对于一个数据库，给定一个模型，数据库将以模式来定义数据。当然，模式只是指定这个数据库拥有这些个表或者这些数据集合。而这些又具有着各自的域名等名称和此数据类型相应的属性。模式允许为数据提供结构，以方便我们的查询。没有模式定义的数据将是没有意义的，将是一串无意义的二进制位罢了。就比如我们的一系列的日志文件，一般而言都保有一种叫做时间戳的东西以方便我们对我们想要找的文件进行一定的定位。相反，如果我们在这些文件上面没有任何的结构，我将无法找到我们想要的东西，毕竟这些在外面看来将会是一系列的01串，我们无法解析它们。

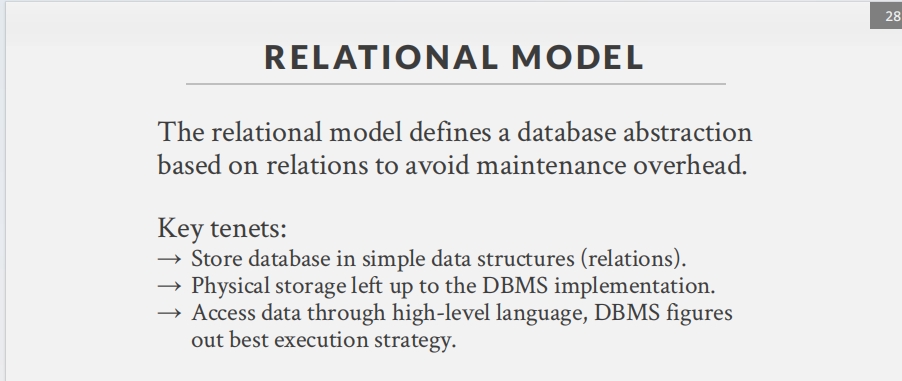
数据模型是关于如何定义数据集合的抽象，以及各个数据间关系的抽象。至于一个数据模型中的模式，可以认为是在一个数据模型中，一个对象具有这一系列的属性。抽象以C++的设计来看，数据模型可以时设计的类，用以定义一个抽象对象中所有数据的含义，毕竟本质上只是一些01串，当我们给他们附加上了一系列解析方式，这些数据才具有意义。至于模式，可以看做是一个实例化的对象，具有确实的数据，是一个真正意义上可以用来进行分析数据的对象。

可以这么看，模型是一系列的属性栏构成的集合，这些个集合是用来匹配我们接下来的数据的分布的，就比如这一块数据是对应的集合中的元素，而模式就是这底层的数据，需要模式来进行这些数据的解析。

接下来进入关系数据库的简介。

关系是一种数据模型，将基于**关系**为数据库定义一个抽象，以避免我们人工去维护数据库系统中各个数据的组成方式。

因此，关系模型有几个关键思想。

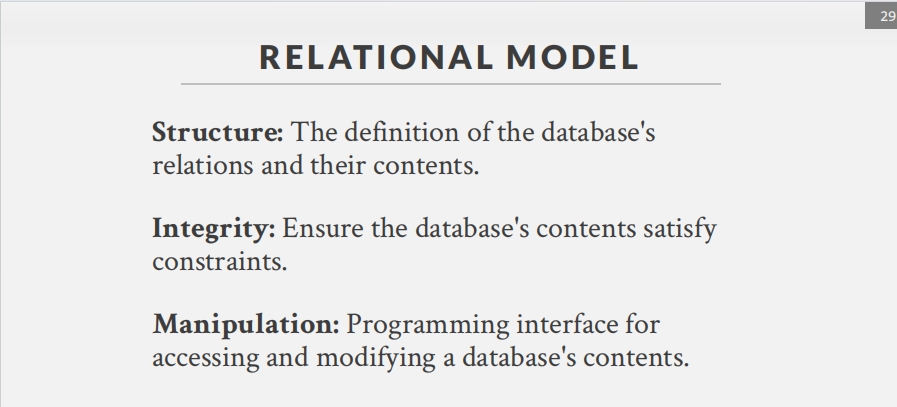


1~将我们的数据库储存为简单的数据结构，就比如关系或者表等。并且不会将这些图或者表中的数据运行应用程序访问。因此，这在用户层得到的效果就差不多相当于C++中类设计的私有成员和公有api。

2~数据的储存地址由数据库系统来进行维护(DBMS等)，其实这就很像我们的虚拟内存系统的设计。我们一般是将数据库的数据放在磁盘这种非易失的储存容器上的，但是我们在使用时是想要将这些数据放在内存上的，而这个步骤不需要我们去进行维护，数据库系统将会自动帮我们完成这个行为。当然，当我们数据库的数据的存储位置发生改变时，我们用户层也不应该需要去对查询操作进行相应的更改，这部分操作应该有数据库系统来代替我们完成。就像虚拟内存系统中的映射。

3~一个关系模型数据库应该支持高级语言的编写，就比如我们的查询操作，我们不希望我们需要去使用数据库系统的操作而是能够使用高级语言去简化我们的设计。

当然，事实上，还有许多一系列的规范是一个关系模型需要遵守的。这里就不列了。而且在现实中，其实很少有一个关系模型数据库能够遵守所有的规范的。

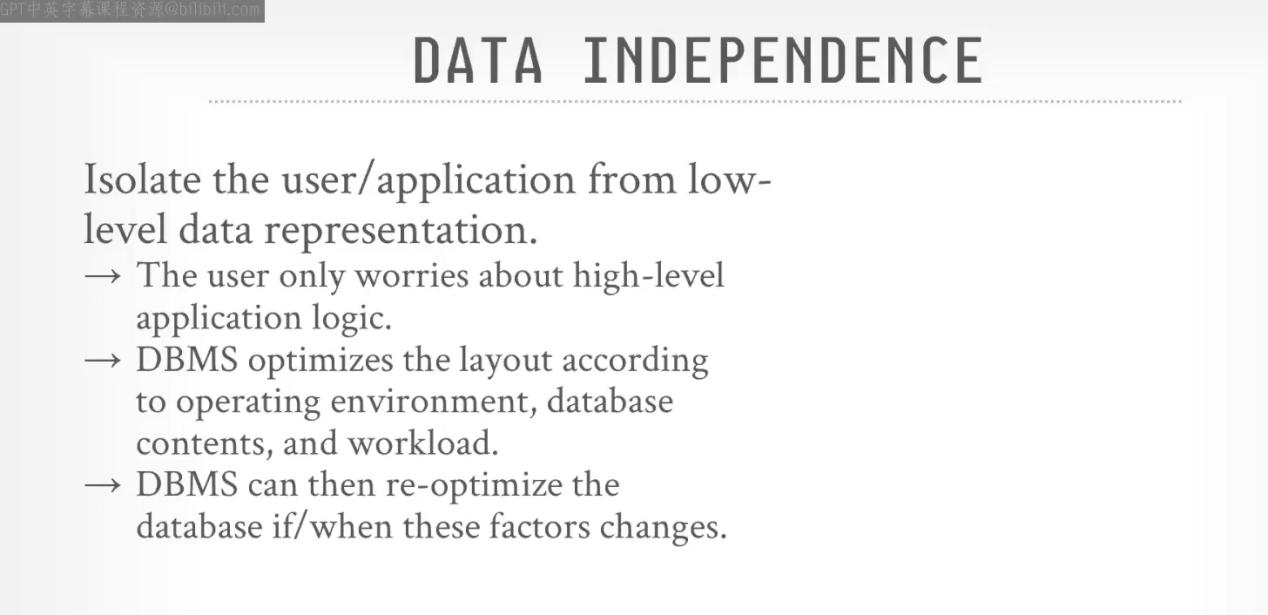


接下来在来看一下关系模式的几个设计

1~Structure(结构)这是一个数据库的结构，这里包含着我们拥有的表的定义，或者，更确切的说，我们拥有的关系的定义。设计着通过一种独立于物理储存模式的方法来对定义这些数据。同时，还可以进行一系列的约束的定义，当然远不止于此。

2~Intergrity我们如果在数据库中添加一个人的年龄，我们应该保证这个属性项不能为负数，而这个就是一种约束。

3~Mainpulation(操作机制)这个基本就是api的定义，通过这些我们能够去高效的查询等对数据库数据的操作。



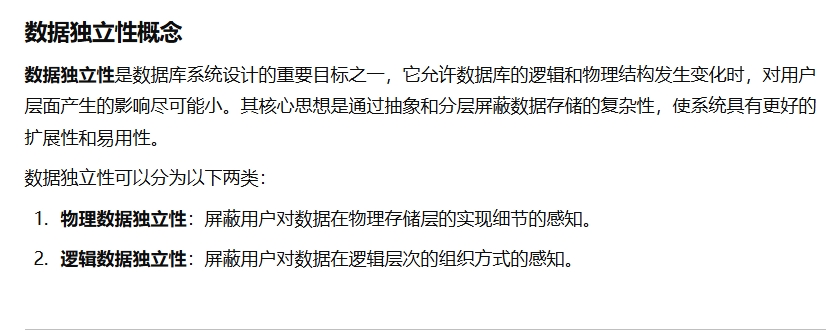
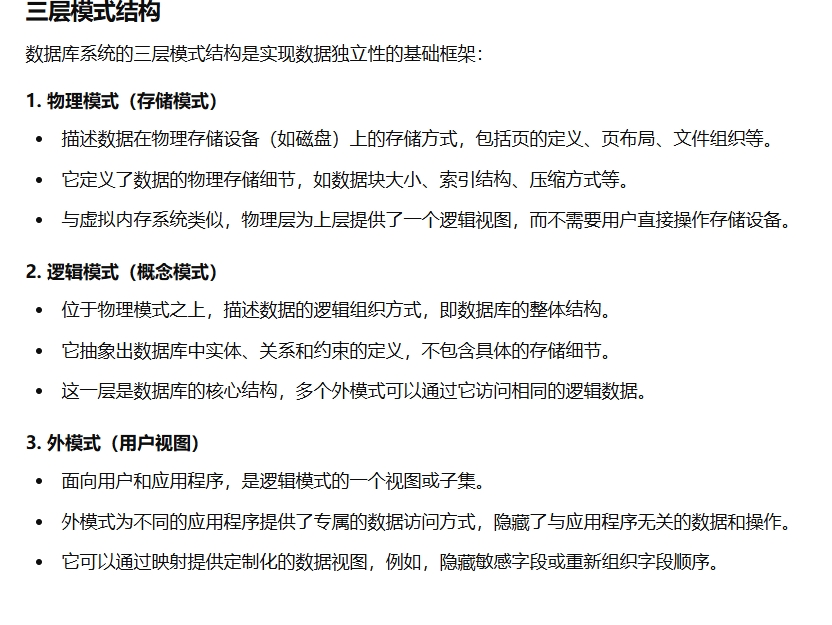
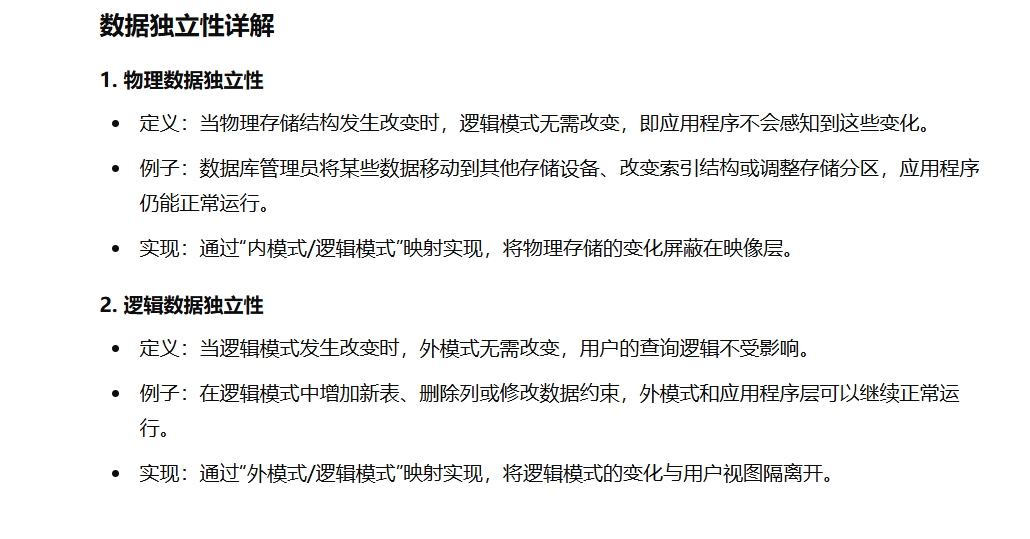
接下来看一下数据库中关于**数据独立性**的定义

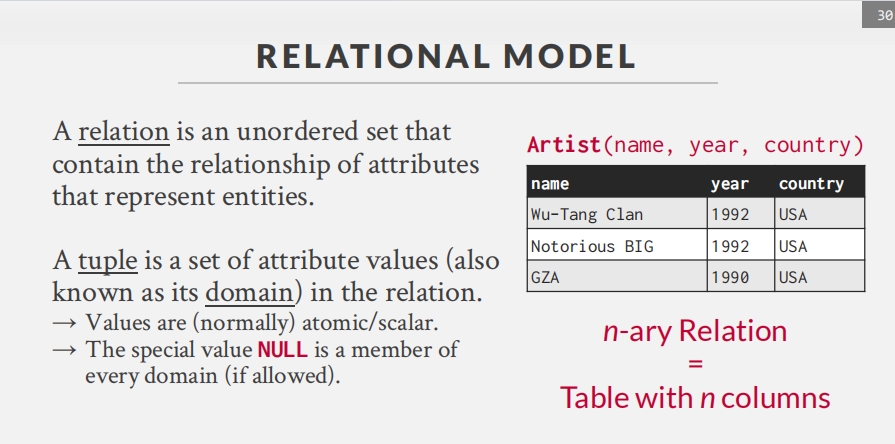
在数据库中，数据独立性有着多个方面。首先，我们不想提供数据库中储存的数据在物理储存上的信息，毕竟这对于设计过程来说是一个相对来说比较复杂的。我们想要通过一种高级抽象能够简化我们的设计。就比如我们计算机的虚拟内存系统一般。

这里通常就设计到了数据库系统的三层模式结构。最底层的是实际数据储存区域，通常是在磁盘等非易失储存器上的。在这个上一层的是内模式，在这一层将会定义页面，页面的布局。就比如这个物理储存页是否储存了数据，这个物理储存页上的一个个二进制位实际上是我们的哪些抽象数据，文件是如何布局的，以及数据库的种种属性(怎么个文件构成等等)。

再之上，是一个”**模式**”，这个是应用程序如何定义对应的数据库中的内容的层次。再往上，就是（**外模式**）了。这个是面向应用程序的层面，是一个相对来说比较高级的封装了。这个外模式中的每一个都能有对应的应用程序且被对应的应用程序所使用，即使可能不会使用到这个外模式的所有数据。当然，也存在多个应用程序共享一个表，外模式通过不同的访问方式来向不同的应用程序提供访问表中不同数据的功能。

再往上，就是我们的应用程序层了，这些由高级语言编写的代码在这个层次中并不是最重要的一环。

所谓的物理独立性其实指的是**内模式/模式映像**，通过这个映像，我们不必去关注实际上数据的物理储存。与此同时，还有一种更高级别的逻辑独立性，这个通常处于**外模式/模式映像**层次，通过这种映像，用户可以认为拥有这些逻辑表，然后可以为不同的应用程序提供不同的数据。实际上就是一种分流，将模式对应的数据给逻辑上分包为一个个应用程序需要的小包，而用户不需要理解这些操作的具体实现以及在这下面的逻辑级别的实际储存方式。就比如你win等系统的一系列文件，你只需要去使用它，至于文件的读写操作，文件的实际储存地址等，都是已经封装好了的。



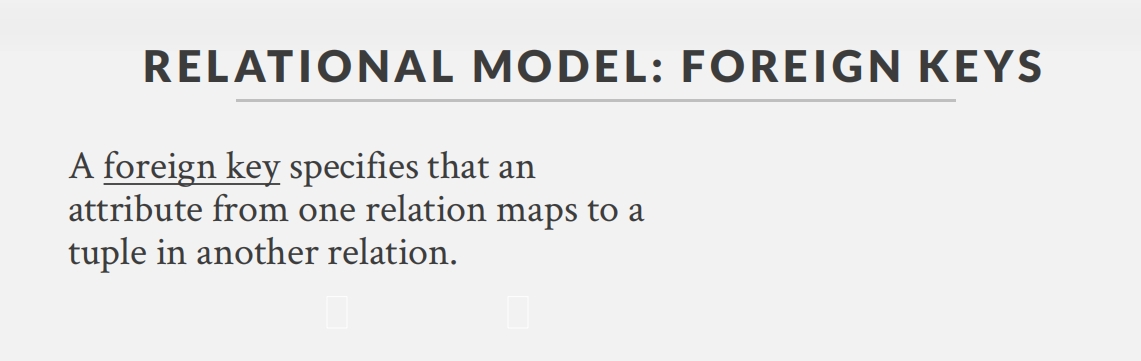
接下来我们来看一下关系。首先，关系应该是一组无序的数其中包含着现实世界中的一些信息。一个关系应该是现实生活中一些数据的映射。请注意，在表中，数据应该是无序的，也就是说，这些数据不应该存在先后的逻辑顺序

在上面那个表中，一行的数据可以表示一个元组。一个元组可以认为是一个基本组成单位。其实用行来说明也比较好，毕竟这样能更凸显出它**行储存**的这个性质(至少你逻辑上看起来像行储存)。

在原始的关系模型中，你必须定义在关系中拥有的所有的属性。

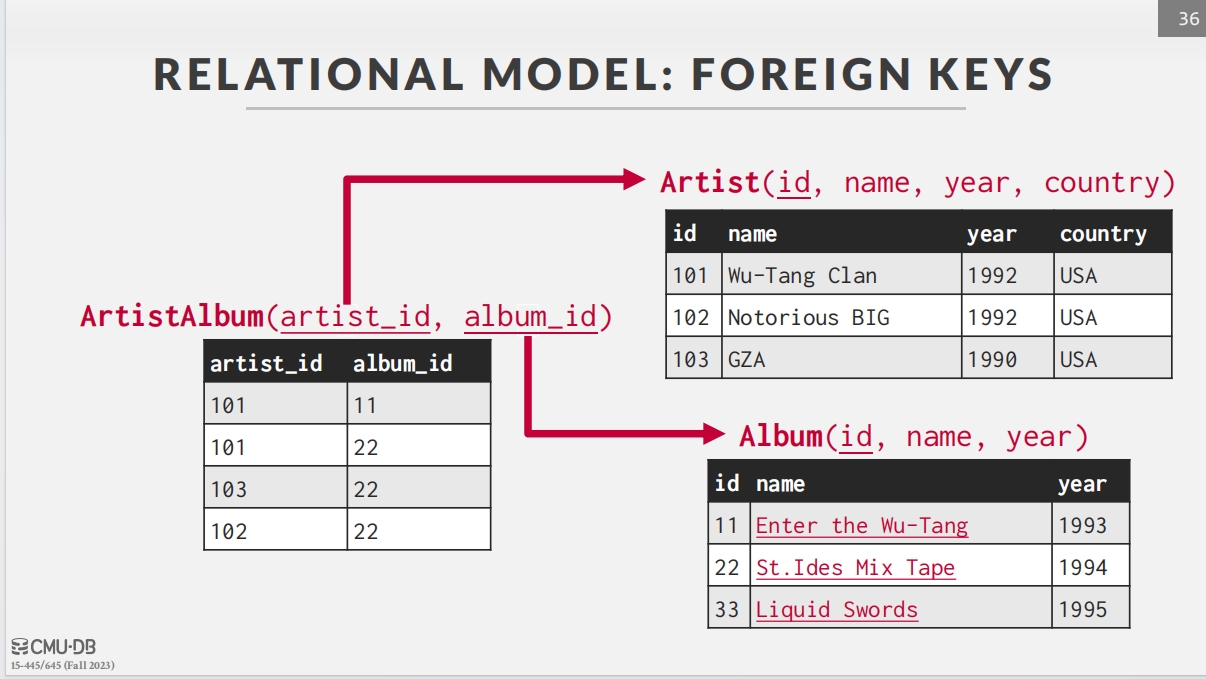
接下来还有一个关系中相当重要的一个属性。主键是一组中唯一代表该组元素的一个实体的属性。它的地位其实就相当于你的身份证号。这个主键在所有的元组中应该是唯一的。通过这个具有唯一性的主键，我们可以类似的像switch一样去构建一个跳转表去进行高效的查找。

事实上，对于主键这种东西，很多的数据库系统都有着自己隐式的维护着的一套主键体系。即使我们不给我们的数据提供一个主键元素，系统也会自动分配一个主键给我们创建的元组。并且，即使我们分配了一个主键元素，系统也可能不会采用这个元素，只是显示的保存着我们提供的主键元素，但在内部还是使用着自己的主键体系。就比如QSLitye通过自己递增的值作为主见，即使我们将我们提供的一个元组中的一个元素定义为主键。

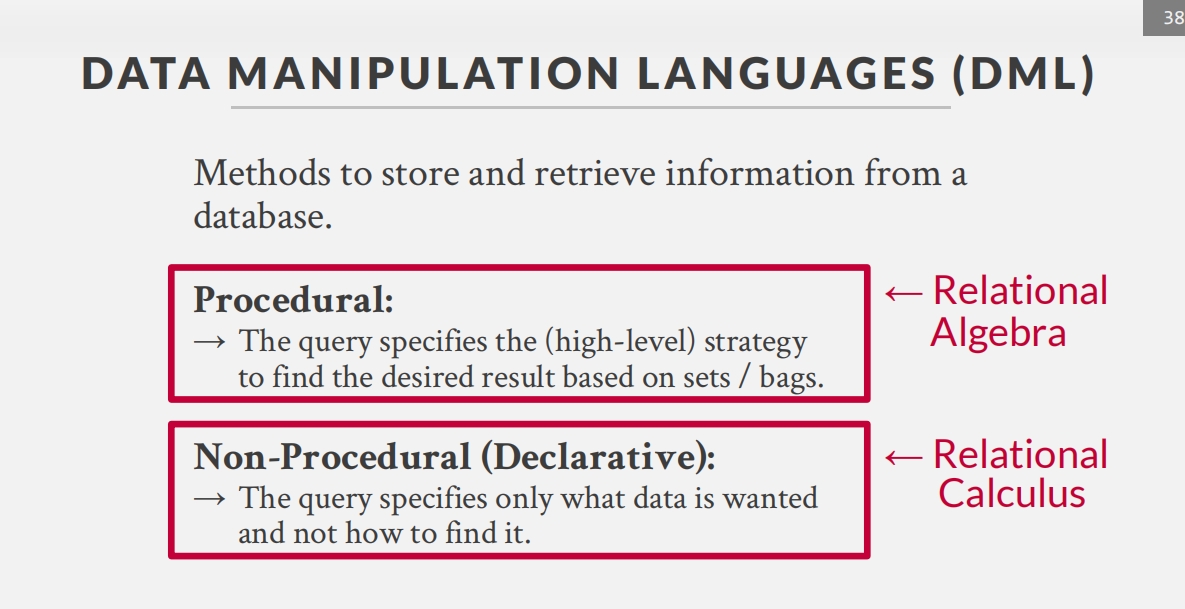


有主键，那么可以想象应该有外键这种东西。但事实上，主键和外键之间并不存在什么直接的联系。

外键在关系中用于一个关系中的元素与另一个关系中的元素进行连接。也就是说，外键用于连接不在同一个关系中的俩个元素，以此来定义着外键俩段的失误是相互关联的。



在这个图中出现了一个外键的实例。ArtistAlbum表是相对于另外俩个表想独立的。这个表的每一个元组可以说储存了俩个外键，其中一个外键指向了Aritist表的一个元组，另一个外键指向了Album的一个元组，通过这种设计。这个表沟通了俩个本来不会有交集的表，而且使得结构更加清晰。在这种架构下，我们如果想要去查找一个特定元素，我们都可以将这个AritistAlbum表看做一个中间表来进行跳转。

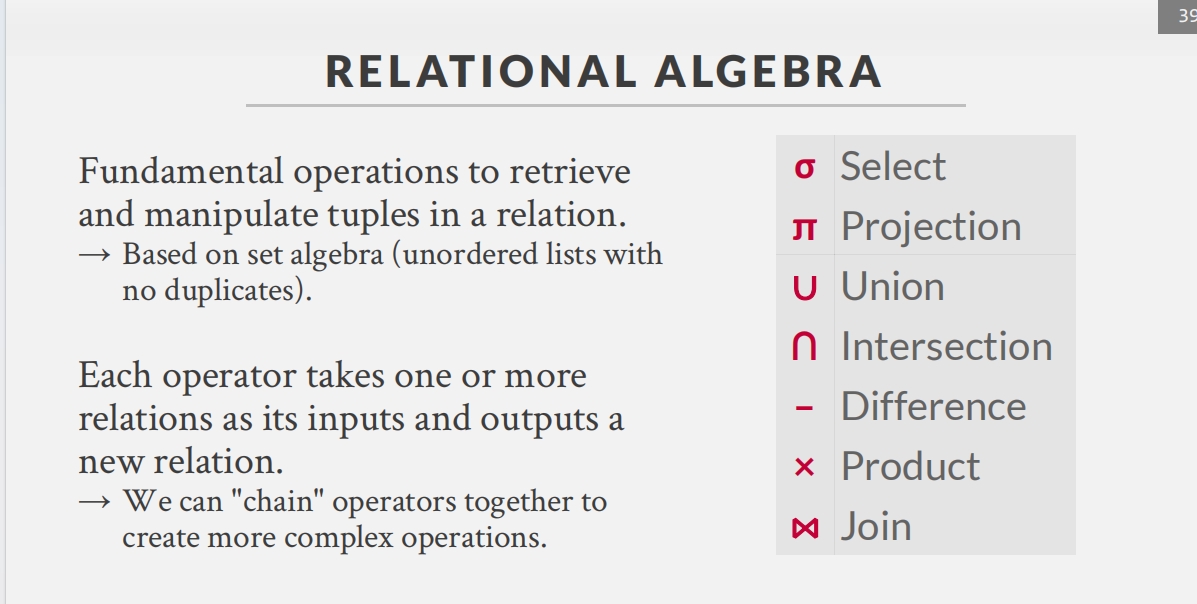


接下来看一下DML(DATA MAN IPU LAT ION LANGUAGES 数据操作语言)按照语言不同的查询方式，可以分为俩种

第一种是一种过程式的。在过程式的查询语言中。查询这个操作需要明确我们要怎么查询，如何处理数据等细致的操作。需要我们去明确**过程**。**关系代数**将是这种实现的一个示例。

第二种是一种非过程式的。这种相对于上面一种来说使用相对简单。这中查询语言应该是隐藏了实现细节的。通过提供需要检索的数据内容由底层的封装函数来帮助我们进行查询的。**关系演算**将是这种实现的一个示例。

在接下来的过程中，我们将会更注重于关系代数的学习。

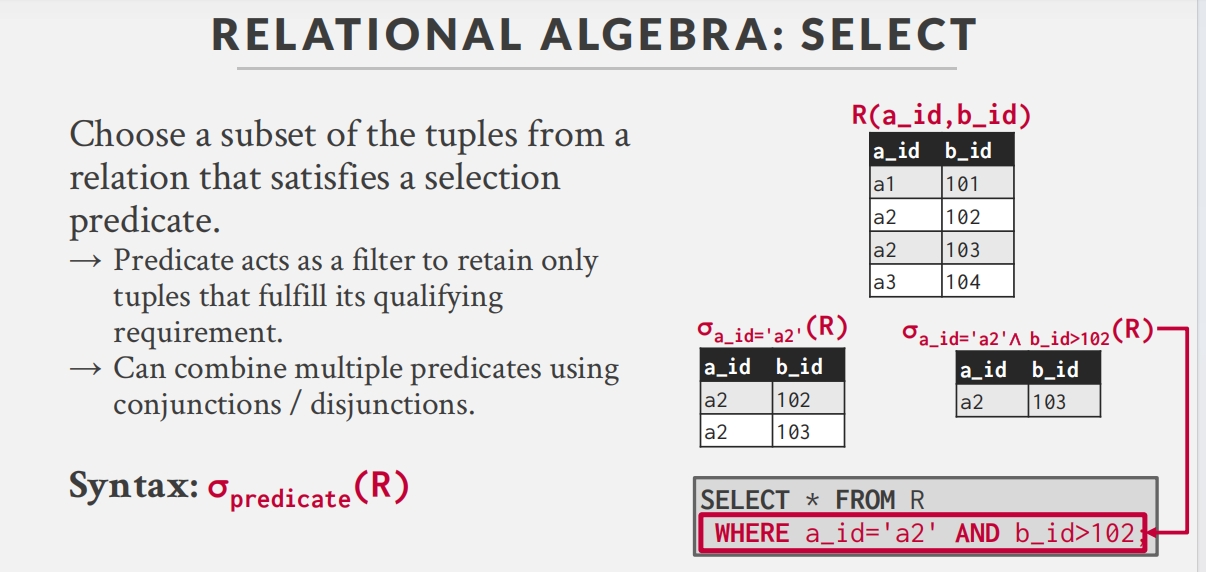


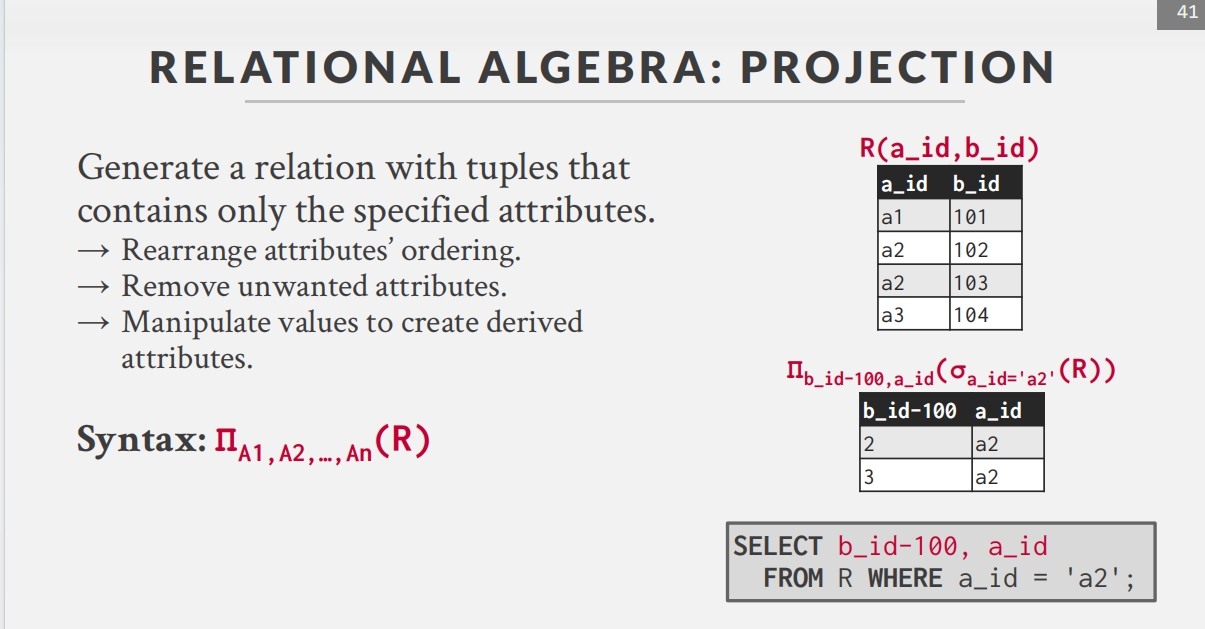
在过程式数据操作语言中，提供或者说定义了多个原语，或者说操作符。我们是希望可以通过这些来构建我们在数据库中所有可能的查询操作的。当然，现在存在着一系列的扩展以来适应越来越复杂的操作。

对于这些个操作符，我们可以认为是对于关系的一种筛选。通过操作符，我们可以对数据库中的一系列表，或者说一系列的关系，当然，也有可能是来自于其他操作符的输出(将输出作为另一个的输入应该很熟悉了)。也就是说，这个其实就是找到基于布尔逻辑的谓词来筛选符合我们理想条件的数据，然后任何符合条件的元组都将被作为输出接受。

就比如下图中选择语句的示例。其实总总这些都很更离散数学有关，但是不深就是了。或者说，暂时不需要用到很深的离散数学。在一个操作符中，我们还可以使用合取和析取等操作将多个条件组合以实现单个操作符无法实现的功能。

总之，这些个操作符其实像一个筛子(谓词逻辑)。

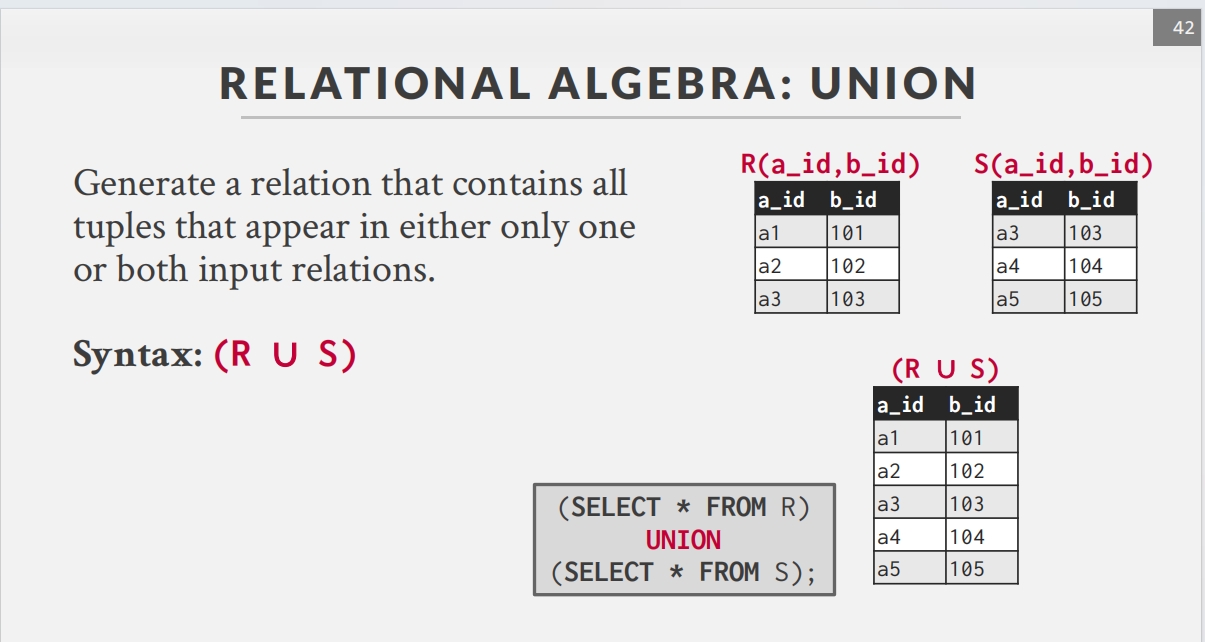




接下来看到**投影**操作符。通过这个操作符，我们可以去重新定义我们的输出应该是一种什么样的顺序，移除我们不想要的属性，以及我们期望以一种什么样的数据去对数据进行处理等等。

注意看上面的一个投影与选择的复合使用。将选择操作符得到的一个元组作为了投影操作符的定义域。并通过对这个R的操作得到了一个相对于原来的元组不同的结果。

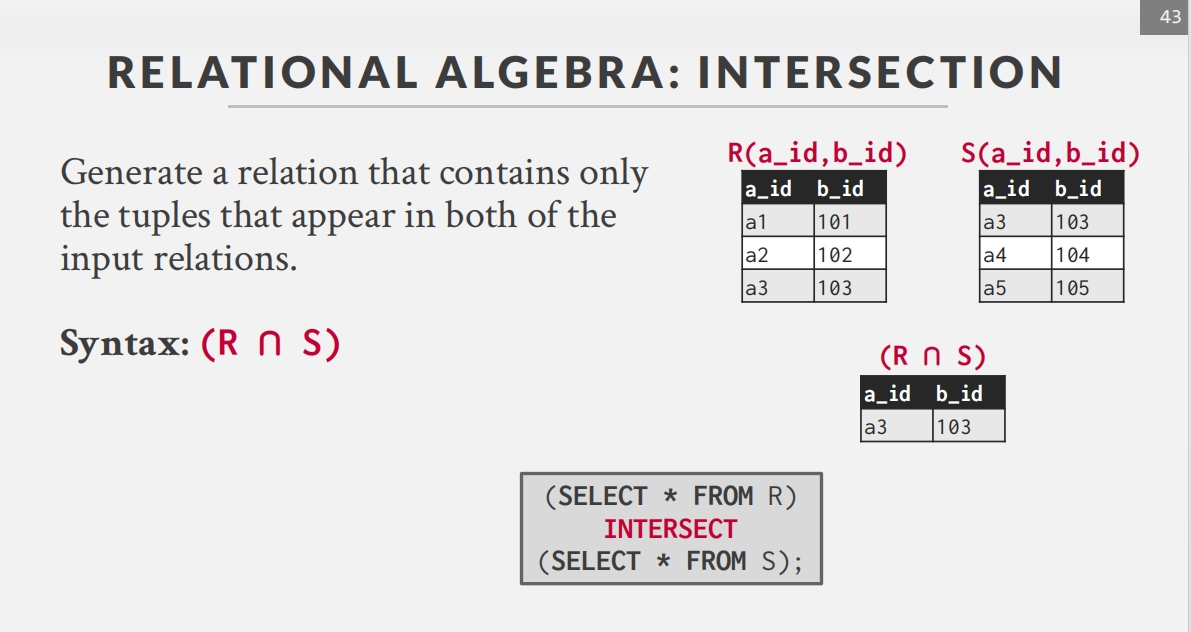
可以看出，通过投影操作符，我们不仅可以改变一个元组中元素的位置，大小，还可以通过一些字符串函数，算术运算等来现自己想要的结果。我们需要做的就是定义对应的操作。

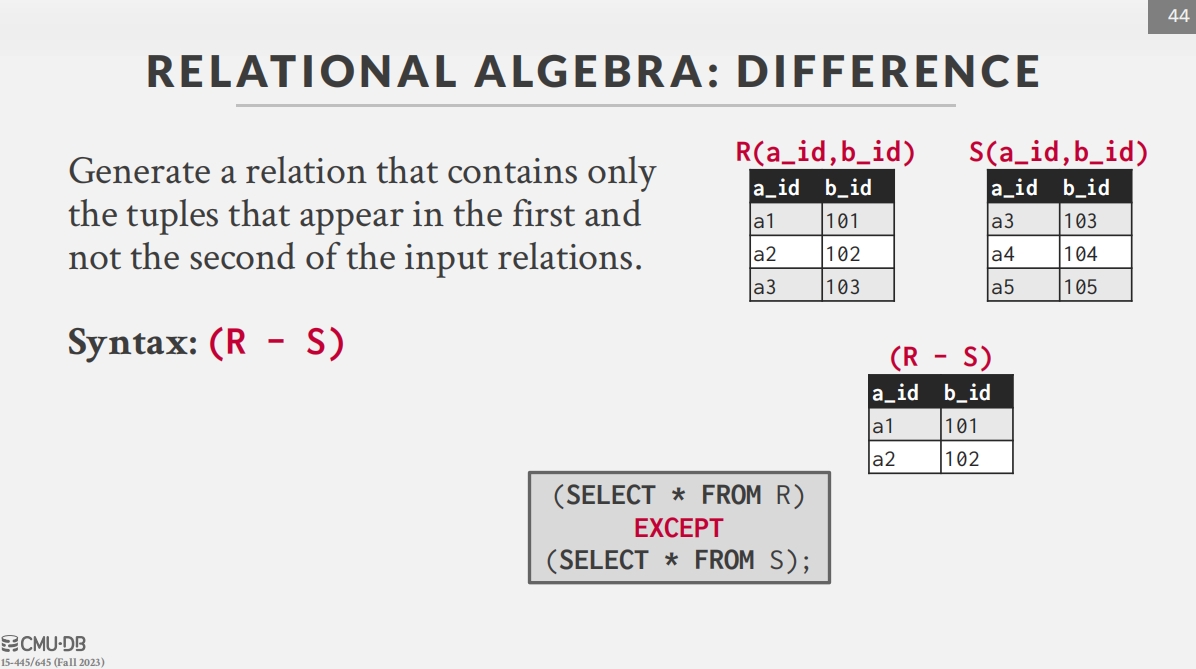


接下来看合并操作符(UNION)。通过这个操作符，我们可以基于已有的关系来生成额外的关系。当然，还有系列功能等待探索。人话：不会。

事实上，简单的合并语句就像上面一个，只是将俩个小表合并为了一张大表。在这种情况下，我们要求进行合并的俩个表需要有相同的属性。

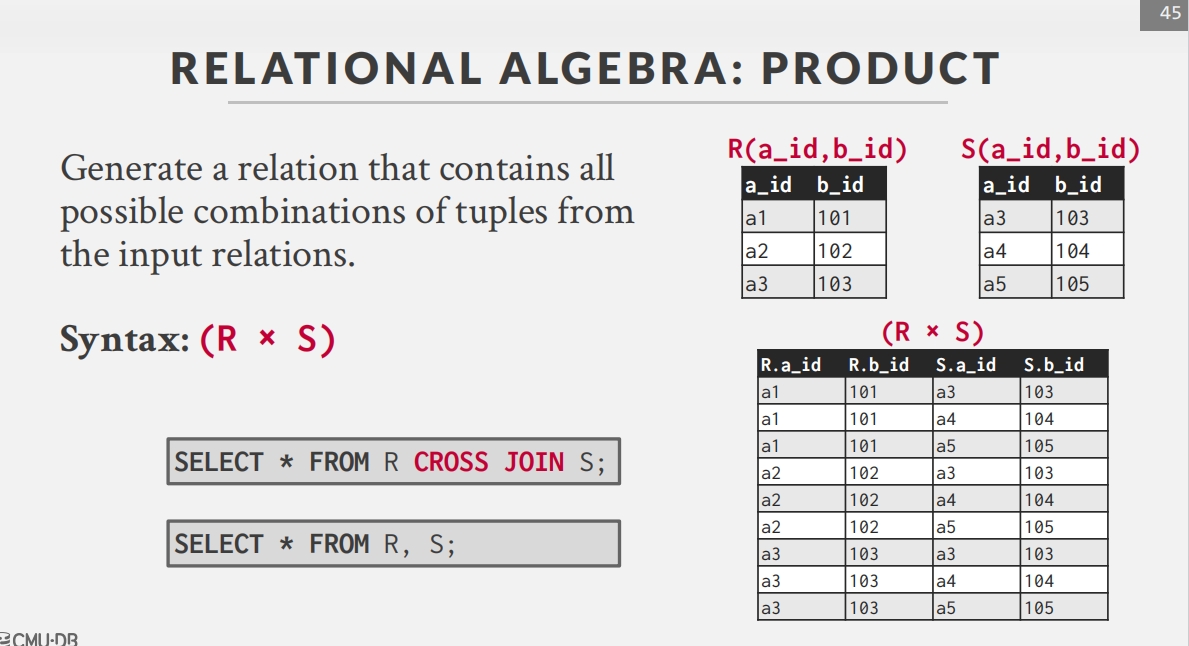
有交就有并，这俩个其实都是老朋友了，就不加赘述了。



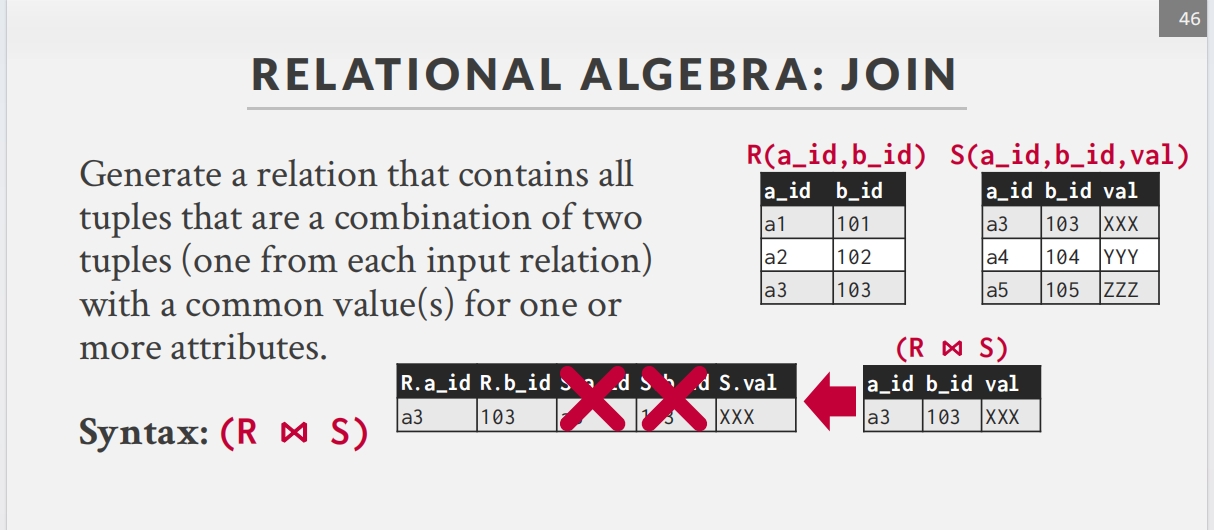


接下来看到差集。差集的结构就是属于第一个关系但是不属于第二个关系的所有元组。

这些其实都只是些集合论的东西，很无趣。



笛卡尔乘积，也是熟悉的东西，略过。



接下来看一个有趣的东西 ，连接(**join**)。

连接将识别这些个列表中的属性，找到其中的相同属性，基于这些相同的属性，将多个元组连接为一个元组并舍去其中的重复项。

这个一般是自然连接，即会通过连接字的名称来进行查找和连接