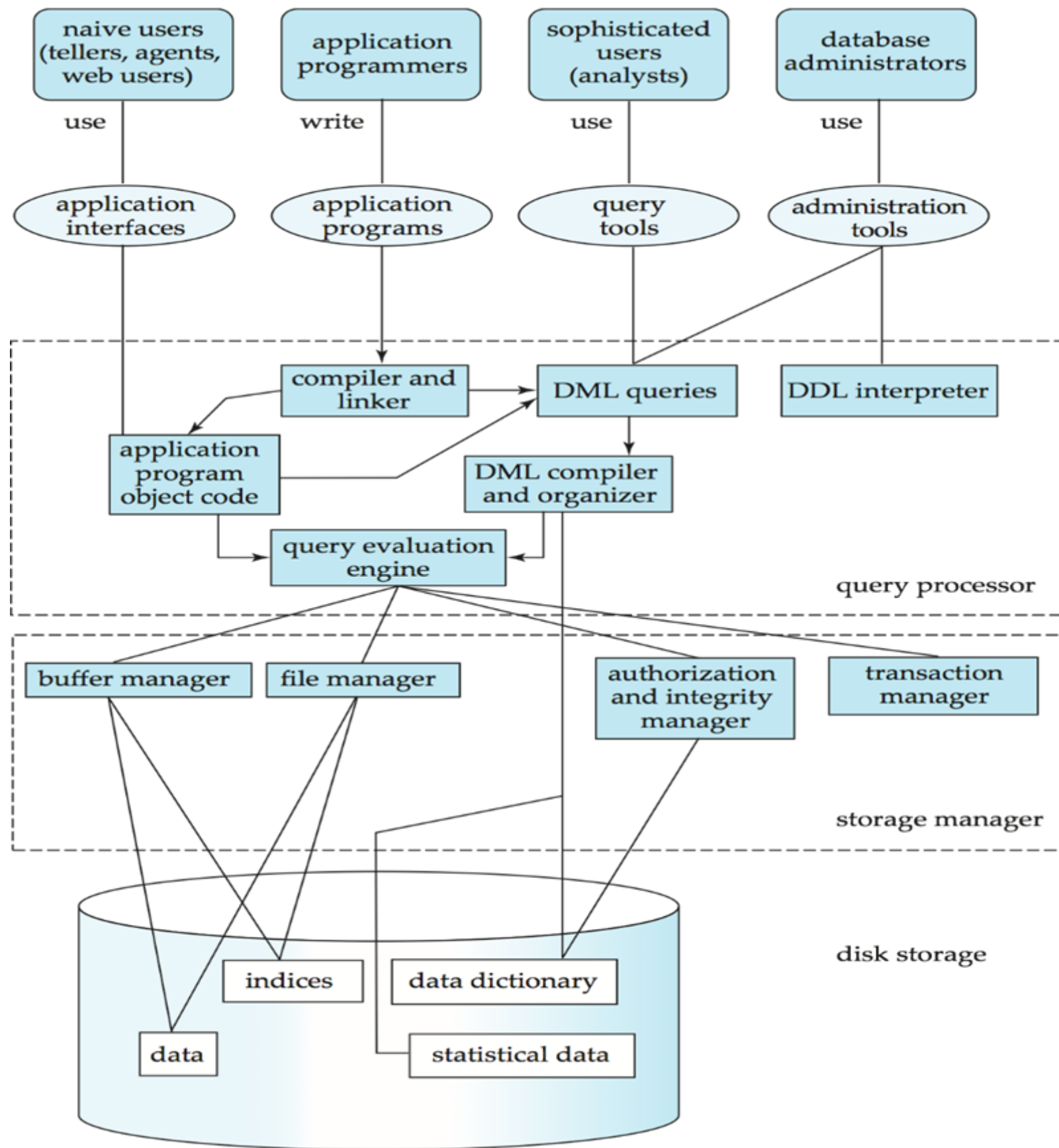


# 1 Database System Internals



## 1. DDL(**Data Definition Language**)

- Specification notation for defining the database schema
- DDL compiler generates a set of (table) templates stored in a **data dictionary**
- Data dictionary contains metadata (i.e., data about data)
  - Database schema
  - Integrity constraints
    - Primary key (ID uniquely identifies instructors)
  - Authorization
    - Who can access what

For example:

```
create table instructor (  
    ID          char(5),  
    name        varchar(20),  
    dept_name   varchar(20),  
    salary      numeric(8,2))
```

## 2. DML(**Data Manipulation Language**)

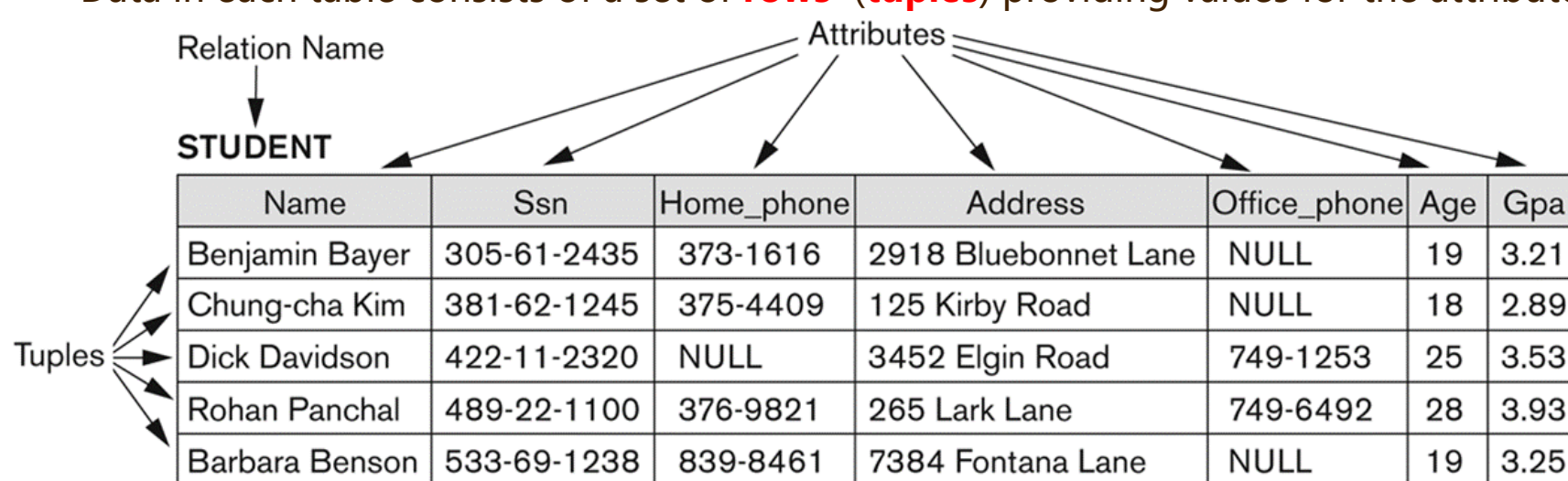
- Programming language for accessing and manipulating the data organized by the appropriate data model
  - a.k.a **query language**

## 3. **Database Engine**

- Storage manager
  - "how to store the data in (secondary) storage system?"
- Query processing
  - "how to minimize the cost and maximize the scalability?"
- Transaction manager
  - "how to ensure system integrity and consistency?"

## 2 The Relational Model

- Database consists of several **tables (relations)**
- Columns in each table are named by **attributes**
- Each attribute has an associated **domain** (set of allowed values)
- Data in each table consists of a set of **rows (tuples)** providing values for the attributes



### 2-1 Schema

#### 1. Schema defination in PPT

## Schema

✓ The **Schema** (or description) of a relation:

- Denoted by **R(A1, A2, .....An)**
  - R is the **name** of the relation
  - The **attributes** of the relation are A1, A2, ..., An

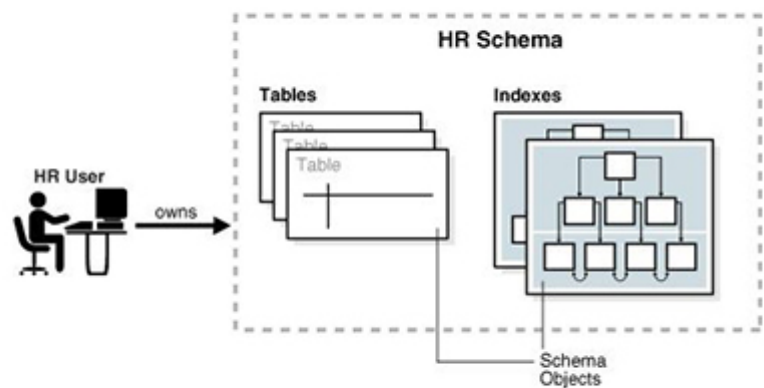
**CUSTOMER (Cust-id, Cust-name, Address, Phone#)**

- **CUSTOMER** is the relation name
- Defined over the four attributes: Cust-id, Cust-name, Address, Phone#

- Each **attribute** has a **domain** or a set of valid values.
- Integrity **constraints** (to be discussed later)

2. Schema Definition in Web

- schema在数据库中表示的是数据库对象集合，它包含了各种对象，比如：表，视图，存储过程，索引等等。
- 一般一个用户对应一个集合，所以为区分不同集合就需给不同集合起名。用户的schema名就相当于用户名，并作为该用户缺省schema。所以schema集合看上去像用户名。
- 例如当我们访问一个数据表时，若该表没有指明属于哪个schema，系统就会自动加上缺省的schema。



2-2 Attributes

Columns in each table are named by attributes

Attribute

✓

Each attribute has an **attribute name**: to interpret the meaning of the data elements corresponding to that attribute

✓

A set of allowed values for each attribute is called the **domain** of the attribute:

-The domain of "USA\_phone\_numbers" is the set of 10-digit phone numbers valid in the U.S.

-the domain of Cust-id is the set of 6-digit numbers

-A special value "NULL" for every domain

✓

A domain also has a data-type or a format.

-USA\_phone\_numbers:  
(ddd)ddd-dddd where each d is a decimal digit.

-Dates: yyyy-mm-dd, dd.mm.yyyy etc.

-"Cust-name": character strings of maximum length 25: varchar(25)

✓

Attribute values are (normally) required to be **atomic** – "indivisible"

2-3 Tuple

元组是关系数据库中的基本概念，关系是一张表，表中的每行（即数据库中的每条记录）就是一个元组，每列就是一个属性。 在二维表里，元组也称为记录。

Data in each table consists of a set of rows (tuples) providing values for the attributes

# Tuple

- ✓ A **tuple** is an **ordered n-tuple** of values (enclosed in angled brackets '< ... >')
  - Each value is derived from an appropriate **domain**.
- ✓ A relation is a **set** of such tuples (rows)

A row in the CUSTOMER relation is a 4-tuple:  
<632895, "John Smith", "101 Main St. Atlanta, GA 30332", "(404) 894-2000">

- ✓ We refer to component values of a tuple  $t$  by  $t(A_i) = v_i$  (the value of attribute  $A_i$  for tuple  $t$ )

## 2-4 State

- The **relation state (or instance)** is a subset of the **Cartesian product** of the domains of its attributes
- Each domain contains the set of all possible values the attribute can take

$$r(R) \subset \text{dom}(A_1) \times \text{dom}(A_2) \times \dots \times \text{dom}(A_n)$$

1. 对任意集合A, 根据定义有  $A \times \Phi = \Phi$ ,  $\Phi \times A = \Phi$
2. 一般地说, 笛卡尔积运算不满足交换律, 即  $A \times B \neq B \times A$  (当  $A \neq \Phi \wedge B \neq \Phi \wedge A \neq B$  时)
3. 笛卡尔积运算不满足结合律, 即  $(A \times B) \times C \neq A \times (B \times C)$  (当  $A \neq \Phi \wedge B \neq \Phi \wedge C \neq \Phi$  时)

Let  $R(A_1, A_2)$  be a relation schema

$$\text{dom}(A_1) = \{0, 1\}, \text{dom}(A_2) = \{a, b, c\}$$

Then:  $\text{dom}(A_1) \times \text{dom}(A_2)$  is a set of all possible combinations:

$$\{<0, a>, <0, b>, <0, c>, <1, a>, <1, b>, <1, c>\}$$

*How many combinations there are for a schema  $R$  with  $n$  attributes?  
(size of the Cartesian product?)*

$r(R)$  could be

$$\{<0, a>, <0, b>, <1, c>\}$$

*How many states there are in total?*

4. How many states there are in total?

For  $R(A_1, A_2, \dots, A_n)$ ,  $|\text{domain}(A_i)| = n_i$

笛卡尔积: 可以遍历出所有每个属性不空的所有属性结合情况

$$P(A_1, A_2, \dots, A_n) = \prod_{i=1}^N |A_i|$$

所以最终答案是这个!



$$\left(\prod_{i=1}^1 |A_i|\right) + \left(\prod_{i=1}^2 |A_i|\right) + \dots + \left(\prod_{i=1}^N |A_i|\right) = 2^{\sum_{i=1}^N |A_i|}$$

## 2-5 Formal Definitions

- Values in a tuple:
  - All values are considered **atomic** (indivisible).
  - Each value in a tuple must be from the domain of the attribute for that column
    - If tuple  $t = \langle v_1, v_2, \dots, v_n \rangle$  is a tuple (row) in the relation state  $r$  of  $R(A_1, A_2, \dots, A_n)$
    - Then each  $v_i$  must be a value from  $dom(A_i)$
  - **component values** of a tuple  $t$  :  $t[A_i]$  or  $t.A_i$
  - Similarly,  $t[A_u, A_v, \dots, A_w]$  refers to the **subtuple** of  $t$  containing the values of attributes  $A_u, A_v, \dots, A_w$ , respectively in  $t$
  - A special **null** value is used to represent values that are unknown or inapplicable to certain tuples.

<u>Informal Terms</u>		<u>Formal Terms</u>
Table (File)		Relation
Column (Field)		Attribute
All possible Column Values		Domain
Row (Record)		Tuple
Table Definition		Schema of a Relation
Populated Table		State of the Relation

## 2-6 Key Constraints

- **Superkey** of  $R$ :
  - a set of attributes  $SK$  of  $R$  with the following condition:
    - No two tuples in any valid relation state  $r(R)$  will have the same value for  $SK$
    - For any distinct tuples  $t_1$  and  $t_2$  in  $r(R)$ ,  $t_1[SK] \neq t_2[SK]$
    - This condition must hold in *any valid state*  $r(R)$
- **Key** of  $R$ :
  - A "minimal" superkey: a key is a superkey  $K$  such that removal of any attribute from  $K$  results in a set of attributes that is not a superkey
- In conclusion
  - Any *key* is a *superkey* (vice versa?)
  - Any set of attributes that *includes a key* is a *superkey*
  - A *minimal* superkey is a *key*

## 2-7 Functional Dependencies

定义：

一个数据库 R 的一组 FD 可以表示成： $X \rightarrow Y$ ，其中 X 和 Y 是数据库 R 中的两组属性集合，其含义是：对于 R 中的任意两个元组，只要他们的在集合 X 中的属性相等，那么他们在集合 Y 中的属性也相等。这里 X 称为决定因素 (Determinant)，Y 称为被决定因素 (Dependant)。

- A set of attributes X *functionally determines* a set of attributes Y if the value of X determines a unique value for Y

A FD  $X \rightarrow Y$  is a

- Full functional dependency** if removal of any attribute from X means the FD does not hold any more; otherwise it's a **Partial dependency**
- Transitive functional dependency**- if there a set of attributes Z that are neither a primary or candidate key and both  $X \rightarrow Z$  and  $Z \rightarrow Y$  holds.

我们常用  $\Sigma^*$  表示由函数依赖集合  $\Sigma$  所隐含的所有可能的函数依赖。

我们规定：如果  $\Sigma_1^* = \Sigma_2^*$ ，那么  $\Sigma_1$  和  $\Sigma_2$  是等价的。意思就是， $\Sigma_1$  和  $\Sigma_2$  可以不相同，只要他们对应的  $\Sigma^*$  是相等的，我们就可以认为这两个函数依赖集合的等价的。

函数依赖的最小覆盖 (Minimal cover)

通过定义函数依赖的最小覆盖，我们可以直接通过最小覆盖推理出数据库的所有函数依赖。一个函数依赖的最小覆盖具有以下特点：

$\Sigma_m$  与  $\Sigma$  是等价的。其中  $\Sigma_m$  是最小覆盖， $\Sigma$  是数据库给定的函数依赖集合；

Dependant：最小覆盖的每一条函数依赖，其右侧只存在单个的属性；

Determinant：最小覆盖的每一条函数依赖，其左侧可以存在多个属性；

任何冗余的函数依赖都会被移除。

### 2-7-1 Closure 闭包

1:

定义：通过一个属性 X 集合推理出来的所有属性集合，称之为该属性 X 的闭包 (Closure)，记作  $X^+$ 。

例1： $\Sigma \models X \rightarrow W$  等价于  $W \subseteq X^+$

例2：一个数据库  $R = \{A, B, C, D, E, F\}$  具有一下的函数依赖集合： $\Sigma = \{AC \rightarrow B, B \rightarrow CD, C \rightarrow E, AF \rightarrow B\}$ 。要判断  $\Sigma \models AC \rightarrow DE$  是否成立。我们首先需要找到属性 AC 的闭包：

1.	$(AC)^+$	$\supseteq AC$	初始化
2.		$\supseteq ACB$	根据依赖 $AC \rightarrow B$
3.		$\supseteq ACBD$	根据依赖 $B \rightarrow CD$
4.		$\supseteq ACBDE$	根据依赖 $C \rightarrow E$
5.		$= ACBDE$	

其中，我们发现  $DE \subseteq (AC)^+$ ，所以  $\Sigma \models AC \rightarrow DE$  是成立的。

2: 主键与函数依赖的关系

Candidate keys and FDs:

- Consider  $R(A_1, \dots, A_n)$  with FDs F
- X is a candidate key for R if X is from  $\{A_1, \dots, A_n\}$  and
  - $X \rightarrow A_1, \dots, A_n \in F^+$ ; and
  - there is no subset Y of X such that  $Y \rightarrow A_1, \dots, A_n \in F^+$ ;

### 2-7-2 Armstrong's axioms 阿姆斯特朗法则

1.

- Reflexivity(自反性): If  $Y$  is a subset of  $X$ , then  $X \rightarrow Y$
- 2.
- Augmentation(增广性): If  $X \rightarrow Y$ , then  $XZ \rightarrow YZ$  for any  $Z$
- 3.
- Transitivity(传递性): If  $X \rightarrow Y$  and  $Y \rightarrow Z$ , then  $X \rightarrow Z$
- 4.
- Decomposition(分解性): If  $X \rightarrow YZ$  then  $X \rightarrow Y$  and  $X \rightarrow Z$
- 5.
- Composition(结合性): if  $X \rightarrow Y$  and  $U \rightarrow V$  then  $XU \rightarrow YV$
- 6.
- Pseudo transitivity(伪传递性): if  $X \rightarrow Y$  and  $X \rightarrow Z$  then  $X \rightarrow YZ$
- 

## 2-7-3 通过 FD 寻找键

### 通过 FD 寻找键

在一个数据库中，一定存在这样的函数依赖关系： $K \rightarrow R$ ，其中  $K$  是超键， $R$  是该数据库所有属性的集合。

#### 算法

- 输入：数据库  $R$  的 FD 集合  $\Sigma$ 。
- 输出：数据库  $R$  所有超键的集合。
- 步骤：
  - 对于数据库  $R$  的每一个属性集合的子集  $X$ ，计算它的闭包  $X^+$ ；
  - 如果  $X^+ = R$ ，那么  $X$  就是一个超键；
  - 如果不存在  $X$  的真子集  $Y$  满足  $Y^+ = R$ ，那么  $X$  就是候选键（主键）。

在这个部分，我们把在候选键中出现的所有属性称为**主要属性（Prime attribute）**，其余的属性则称为**非主要属性（Non-prime attributes）**。

在寻找候选键的过程中，有一些比较好用的小技巧：

- 如果一个属性从来没有出现在任何 FD 的右侧，那么它肯定是候选键的一部分；
- 如果一个属性从来没有出现在任何 FD 的左侧，但它出现在某个 FD 的右侧，那么它肯定不是候选键的一部分；
- 如果某个集合  $X$  的真子集是候选键，那么  $X$  肯定不是一个候选键。

---

## 2-8 Relational Algebra

## 3-1 XML: Data model

### 1: 定义

XML 指可扩展标记语言（Extensible Markup Language）。

可扩展标记语言（英语：Extensible Markup Language，简称：XML）是一种标记语言，是从标准通用标记语言（SGML）中简化修改出来的。它主要用到的有可扩展标记语言、可扩展样式语言（XSL）、XBRL和XPath等。

XML **被设计用来传输和存储数据**。

HTML **被设计用来显示数据**。

### 2: XML Elements

**Element**: the segment between a start and its corresponding end tag

- subelement: the relation between an element and its component elements.

```
<person>
  <name> Yinghui Wu </name>
  <tel> (509) 335-7612 </tel>
  <email> yxw1650@case.edu </email>
  <email> yinghui.wu2@case.edu </email>
</person>
```

例如:

person: **Element**

name, tel, email: subelement

特点: **XML elements are ordered**

XML 文档中元素的顺序很重要, 会影响文档的处理和解释方式。在 XML 中, 元素的顺序可以传达有关元素之间的结构和关系的信息。例如, 元素之间的父子关系是由元素嵌套的顺序定义的。元素的顺序也可用于传达有关事件或过程中步骤顺序的信息。

### 3: XML attributes


A start tag may contain **attributes** describing certain "properties" of the element (e.g., dimension or type)

例如: 蓝色都是属性

```
<picture>
  <height dim="cm"> 2400</height>
  <width dim="in"> 96 </width>
  <data encoding="gif"> M05-+C$ ... </data>
</picture>
```

References(引用) (meaningful only when a **DTD** is present):

```
<person id = "011" pal="012">
  <name> Barack Obama </name>
</person>
<person id = "012" pal="011">
  <name> Hillary Clinton </name>
</person>
```



补充:

DTD 代表文档类型定义。在 XML (可扩展标记语言) 的上下文中, DTD 是一组定义 XML 文档的结构和元素的规则。DTD 指定可在 XML 文档中使用的有效元素和属性, 以及这些元素与其结构之间的关系。DTD 的目的是验证 XML 文档并确保其格式正确并遵循定义的结构。您提供的文本中对 DTD 的引用表明有一个 DTD 定义了"person"元素的结构, 包括属性"id"和"pal"。

特点:

- **XML attributes cannot be nested** -- flat
- The names of XML attributes of an element must be **unique**. 每个元素可以有多个属性, 但每个属性必须有一个唯一的名称。例如: <person pal="Blair" pal="Saddam"> 这就是错误的, pal重复了。
- XML attributes are **not ordered**, 例如:

1:

```
<person pal="012" id = "011">
  <name> Barack Obama </name>
</person>
```

2:

```
<person id = "011" pal="012" >
  <name> Barack Obama </name>
</person>
```

1、2就一样。

### 4: Compare elements to attributes

- Attributes vs. subelements: unordered vs. ordered

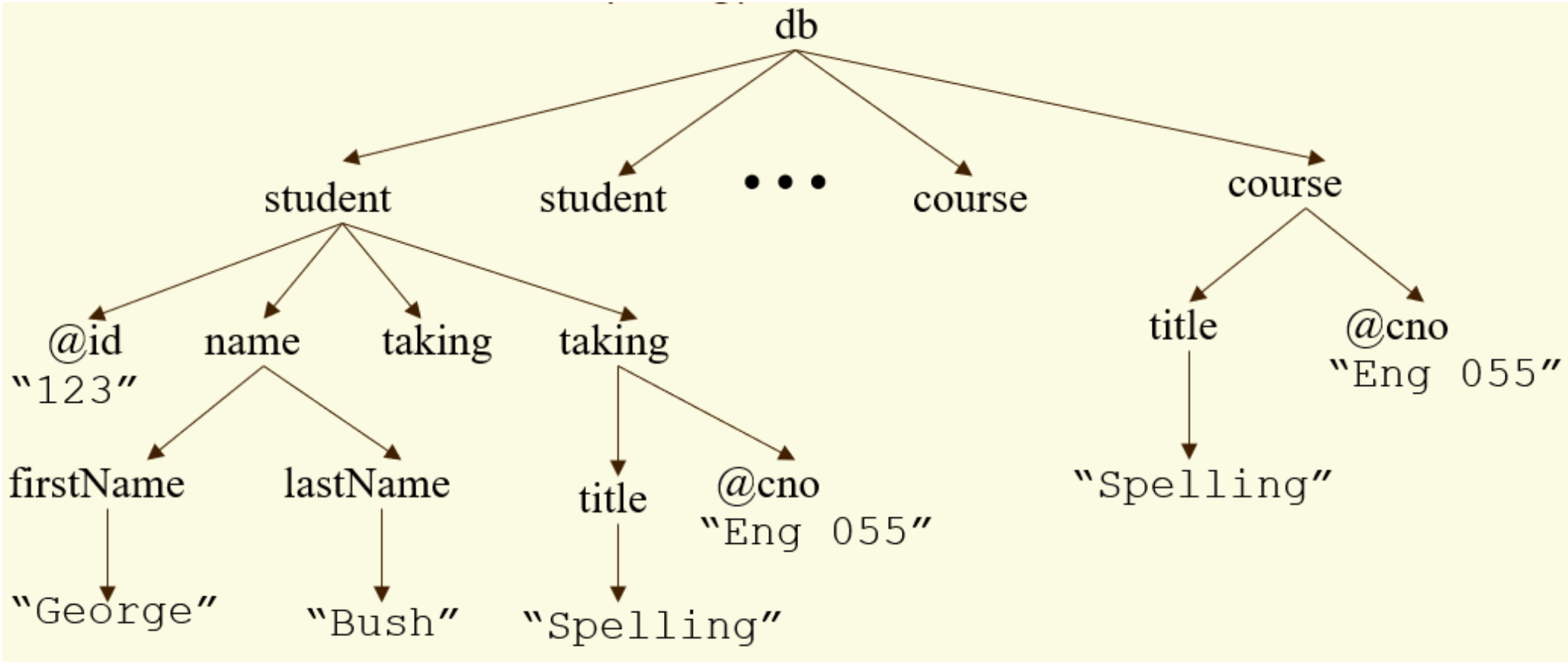


- attributes cannot be nested (flat structure)
- subelements cannot represent references

5: XML tree model

- **Element** node: typically internal, with a name (tag) and children (subelements and attributes), e.g., `student`, `name`.
- **Attribute** node: leaf with a name (tag) and text, e.g., `@id`.
- **Text** node: leaf with text (string) but without a name.

An XML document is modeled as a node-labeled ordered tree.



3-2 XML: DTD and XML Schema

Document Type Definition (DTD)

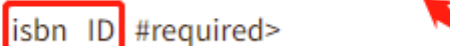
An XML document may come with an optional DTD – “schema”

```
<!DOCTYPE db [  
  <!ELEMENT db (book*)>  
  <!ELEMENT book (title, authors*, section*, ref*)>  
  <!ATTLIST book isbn ID #required>  
  <!ELEMENT section (text | section)*>  
  <!ELEMENT ref EMPTY>  
  <!ATTLIST ref to IDREFS #implied>  
  <!ELEMENT title #PCDATA>  
  <!ELEMENT author #PCDATA>  
  <!ELEMENT text #PCDATA>  
>]
```

- 解释：**
- <!DOCTYPE> - 定义文档类型声明并标识文档的根元素。
  - [] - 包含 DTD 声明，包括元素、属性和实体引用。
  - <!ELEMENT> - 定义 XML 文档中元素的结构，包括元素的名称及其内容模型。例如，(book\*) 表示“db”元素可以包含零个或多个“book”元素。
  - <!ATTLIST> - 定义元素的属性及其类型、默认值，以及它们是必需的还是可选的。例如，(isbn ID #required) 表示“book”元素必须具有 ID 类型的“isbn”属性，并且它是必需的。
  - #PCDATA - 表示该元素只能包含文本数据。
  - EMPTY - 表示该元素没有内容。
  - | - 表示选择运算符，这意味着元素可以包含一个或另一个。例如，(text | section)\* 表示“section”元素可以包含文本或更多“section”元素。
  - \* - 表示出现运算符，表示元素可以出现零次或多次。例如，(book\*) 表示“db”元素可以包含零个或多个“book”元素。
  - ID - 表示该属性是一个标识符并且在文档中必须是唯一的。
  - IDREFS - 指示属性包含引用文档中其他元素的标识符列表。

## 注意:

```
<!ELEMENT book (title, authors*, section*, ref*)>  
<!ATTLIST book isbn ID #required>
```



<!Element ...>层中只有可能是Element, 属性都存在<!ATTLIST ...>层中

## 对 <!ATTLIST ref to IDREFS #implied> 声明的解释:

<!ATTLIST> - 定义元素的属性列表。

ref - 指定属性所属的元素名称。

to - 指定属性的名称。

IDREFS - 将属性类型指定为 ID 引用列表, 用于引用 ID 值列表。

#implied - 指定该属性是可选的并且没有默认值。如果该属性不存在于元素中, 则其值被认为是空的。

因此, 在这种特定情况下, ref 元素可以有一个名为 to 的可选属性, 它是对文档中其他元素的 ID 引用列表。该属性不需要出现在元素中, 但如果出现, 它必须包含 ID 引用列表。如果该属性不存在, 则认为它是空的。

## 1. Element Type Definition (ETD)

在所提供的 DTD 中, <!ELEMENT E P> 声明使用正则表达式“P”定义了元素“E”及其内容模型。正则表达式可以具有以下形式之一:

<!ELEMENT E P> 可以用 E ---> P

EMPTY - 表示该元素没有内容。

ANY - 表示该元素可以包含任何内容, 包括其他元素和文本。

#PCDATA - 表示该元素只能包含文本数据。

E'——指的是DTD中定义的另一个元素, 它被用作“E”的子元素。

P1, P2 - 表示该元素可以依次包含 P1 和 P2。

P1 | P2 - 表示该元素可以包含 P1 或 P2, 但不能同时包含两者。

P? - 表示该元素可以包含零个或一次出现的 P。

P+ - 表示该元素必须包含一次或多次出现的 P。

P\* - 表示该元素可以包含零次或多次出现的 P。

例如, 正则表达式 (#PCDATA|b)\* 表示该元素可以包含一系列文本和“b”元素, 其中“b”元素在 DTD 的其他地方定义。正则表达式 (a|b)+ 表示该元素必须至少包含一次“a”元素或“b”元素, 但可以多次出现。

### 重中之重: Subelements are **ordered**.

single root: <!DOCTYPE db [ ... ] >; subelements are **ordered**.

#### 解释:

在 XML 中, 元素中子元素的顺序很重要并且会保留下来。这意味着, 如果 XML 文档指定了元素中子元素的特定顺序, 则在解析和处理文档时会保持该顺序。

子元素的顺序对于定义 XML 文档的结构和含义很重要。例如, 考虑一个代表一本书的多章文档。章节应该以特定的方式排序, 以保持本书的叙述流畅。如果不保留章节的顺序, 这本书的意义和连贯性就会丢失。

总之, 子元素的顺序在 XML 中很重要, 在创建、处理和修改 XML 文档时应予以维护。

### recursive definition 递归规定

递归定义是在自己的定义中引用自己的定义。在 XML 中, 递归定义可用于定义包含相同类型子元素的元素。

例如, 考虑以下“section”元素的 DTD 声明:

```
<!ELEMENT section (text | section)*>
```

此声明是一个递归定义, 因为“section”元素包含相同类型的子元素, 允许在节内嵌套节。

#### Thinking: recursive DTDs

```
<!ELEMENT person (name, father, mother)>
```

```
<!ELEMENT father (person)>
```

```
<!ELEMENT mother (person)>
```

**What is the problem with this? How to fix it?**

Answer:

递归 DTD 在某些情况下可能很有用, 但它们也可能导致无限循环和处理错误等问题。

```
<!ELEMENT person (name, father?, mother?)>
```

```
<!ELEMENT father (person) >
```

```
<!ELEMENT mother (person)>
```

定义具有适当终止条件的递归 DTD 非常重要，该终止条件将在某个点结束递归。

### In-class Quiz:

#### 1. How do we define “binary tree” with DTD?

binary tree:

```
<!ELEMENT node (leaf | (node, node))
```

```
<!ELEMENT leaf (#PCDATA)>
```

#### 2. The following two definitions are different. Why?

```
<!ELEMENT section (text | section)*>
```

```
<!ELEMENT section (text* | section* )>
```

#### 3. How to declare E to be an unordered pair (a, b)?

```
<!ELEMENT E ((a, b) | (b, a)) >
```

## 2.Attribute declarations

```
<!ATTLIST element_name
    attribute-name attribute-type default-declaration>
```

element\_name: 与属性关联的元素名称。

attribute\_name: 正在声明的属性的名称。

attribute\_type: 属性值的数据类型。可用的类型是 CDATA (字符数据)、ID (唯一标识符)、IDREF (对唯一标识符的引用)、IDREFS (对一个或多个唯一标识符的引用)、NMTOKEN (名称令牌)、NMTOKENS (名称令牌列表)、ENTITY (实体名称)、ENTITIES (实体名称列表)、NOTATION (符号名称) 和 ENUMERATION (一组预定义值)。

default\_declaration: 指定属性的默认值，可以是以下四个值之一：#REQUIRED (必须提供属性)、#IMPLIED (属性是可选的)、a default value (例如“red”) 或#FIXED (属性必须具有特定值)。

Note: it is OK for several element types to define an attribute of the same name, e.g.,

```
<!ATTLIST person name ID #required>
```

```
<!ATTLIST pet name ID #required>
```

## 3. XML reference mechanism

- ID attribute: **unique** within **the entire document**.
  - An element can have at most one ID attribute.
  - No default (fixed default) value is allowed.
    - #required: a value must be provided
    - #implied: a value is optional
- IDREF attribute: its value must be some other element's **ID** value in the document.
- IDREFS attribute: its value is a set, Each element of the set is the ID value of some other element in the document.

```
<!ATTLIST person
    id ID #required
    father IDREF #implied
    mother IDREF #implied
    children IDREFS #implied>
```

e.g.,

```
<person id="898" father="332" mother="336"
    children="982 984 986">
```

....

```
</person>
```

## 4.Valid XML documents

A **valid** XML document must have a DTD.

- It **conforms to** the DTD:
  - elements conform to the grammars of their type definitions (nested only in the way described by the DTD)
  - elements have all and only the attributes specified by the DTD
  - ID/IDREF attributes satisfy their constraints:
    - ID must be distinct
    - IDREF/IDREFS values must be existing ID values

建立了XML和DTD关系

# 3-3 XML Schema

## Official W3C Recommendation

W3C XML Schema 是万维网联盟 (W3C) 的官方推荐，它提供了用于定义 XML 文档的丰富类型系统。

XML Schema 的一些主要特性包括：

**1: Simple (atomic, basic) types 简单类型**

元素和属性的简单（原子、基本）类型：XML Schema 提供了广泛的内置简单类型，例如 xs:string、xs:integer、xs:boolean、xs:date 等，可以使用 定义 XML 文档中元素和属性的数据类型。

**2: Complex types 复杂类型**

元素的复杂类型：除了简单类型之外，XML Schema 还提供了复杂类型，可用于定义 XML 文档中元素的结构和内容。复杂类型可以包含多个元素和属性，还可以定义各种约束和方面。

**3: Inheritance 继承**

继承：XML Schema 还支持继承，允许一种类型从另一种类型派生。这对于定义相关类型的层次结构很有用，并且有助于减少 XML 文档中重复和重复的数量。

**4: Constraints 约束**

XML Schema 还允许您对 XML 文档中的数据定义各种约束，例如最小值和最大值、模式匹配和枚举。这些约束有助于确保 XML 文档中的数据有效并符合特定的规则和准则。

总体而言，XML Schema 提供的丰富的类型系统、继承和约束特性使其成为定义 XML 文档并确保其有效性和完整性的强大工具。

## 1. Atomic types

除了内置的原子类型外（见上面定义），XML模式还允许使用内置类型以及其他限制和方面的组合来创建用户定义的原子类型。 这些自定义类型可用于对XML元素和属性的值（例如最小值和最大值，模式和允许值列表）的值进行特定的约束。

Example: define an element or an attribute

```
<xs:element name="car" type="carType">
<xs:attribute name="car" type ="carType">
```

Define the type:

```
<xs:simpleType name="carType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Audi">
    <xs:enumeration value="BMW">
  </xs:restriction>
</xs:simpleType>
```

如何理解上面所定义的数据类型呢？

此 XML 架构定义定义了一个名为“carType”的简单类型，它使用 xs:enumeration facet 将允许的值限制为“Audi”或“BMW”。 以下是每行含义的细分：

xml Copy code

```
<xs:simpleType name="carType">
```

This line declares a new simple type with the name "carType".

xml Copy code

```
<xs:restriction base="xs:string">
```

This line indicates that this simple type is a restriction of the **base type** "xs:string". It means that the "carType" data type will **inherit** all the properties of **the string data type**, but it will also **add some additional restrictions**.

xml Copy code

```
<xs:enumeration value="Audi"/>
<xs:enumeration value="BMW"/>
```

These two lines define the allowed values for the "carType" data type. In this case, **the data type is restricted to only allow the values "Audi" or "BMW"**. Any value that is not "Audi" or "BMW" would be considered invalid for this data type.



xml

Copy code

```
</xs:restriction>
</xs:simpleType>
```

These two lines close the `xs:restriction` and `xs:simpleType` elements, respectively, effectively ending the definition of the "carType" data type.

Overall, this XML Schema definition provides a way to ensure that any element or attribute that is defined with the "carType" data type can only have a value of "Audi" or "BMW", while also inheriting all the properties of the string data type.

## 2. Complex Types

在 XML 模式中，复杂类型用于定义 XML 文档中包含**子元素和/或属性的元素的结构和内容**。与用于定义单个元素或属性的数据类型简单类型不同，**复杂类型可以定义多个子元素和属性**，并为它们指定规则和约束。

Sequence: `xs:sequence` 元素用于定义复杂类型中子元素的有序序列。子元素可以在序列中以任何顺序出现，但序列中的所有元素必须恰好出现一次。

All: `xs:all` 元素用于定义复杂类型中子元素的无序序列。`xs:all` 元素中的所有子元素必须恰好出现一次。

Choice: `xs:choice` 元素用于定义复杂类型中的变体类型。**它允许您指定一组子元素，其中只有一个可以出现在实例文档中。**

Occurrence constraint: `minOccurs` 和 `maxOccurs` 属性用于指定子元素在复杂类型中可以出现的最小和最大次数。

Group: `xs:group` 元素用于定义一组可在复杂类型定义中重复使用的子元素。

Any: `xs:any` 元素用于在复杂类型中定义“开放”类型。它允许您指定任何元素（包括模式中未定义的元素）都可以作为子元素出现。

```
<xs:complexType name="publicationType">
  <xs:sequence>
    <xs:choice>
      <xs:group ref="journalType">
        <xs:element name="conference" type="xs:string"/>
      </xs:choice>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="author" type="xs:string"
        minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:group name="journalType">
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="volume" type="xs:integer"/>
      <xs:element name="number" type="xs:integer"/>
    </xs:sequence>
  </xs:group>
```

## 3. Inheritance

**xs:extension** 元素用于通过扩展现有类型并添加其他字段来定义复杂类型中的子类型。

```
<xs:complexType name="datedPublicationType">
  <xs:complexContent>
    <xs:extension base="publicationType">
      <xs:sequence>
        <xs:element name="isbn" type="xs:string"/>
      </xs:sequence>
      <xs:attribute name="publicationDate" type="xs:date"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

解释: `datedPublicationType` 通过添加新的子元素 `isbn` 和属性 `publicationDate` 来扩展 `publicationType`。**xs:complexContent 元素用于指示复杂类型是通过扩展从另一个复杂类型派生的**，该复杂类型在 `base` 属性中指定。

`xs:complexContent` 元素用于指示复杂类型是通过扩展从另一个复杂类型派生的，该复杂类型在 `base` 属性中指定。

**xs:restriction** 元素用于通过限制或删除现有类型的某些字段来定义复杂类型中的子类型。

```
<xs:complexType name="anotherPublicationType">
  <xs:complexContent>
    <xs:restriction base="publicationType">
      <xs:sequence>
        <xs:choice>
          <xs:group ref="journalType"/>
          <xs:element name="conference" type="xs:string"/>
        </xs:choice>
        <xs:element name="author" type="xs:string"
          minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
```

anotherPublicationType 派生自 publicationType，方法是限制原始定义并添加新的会议元素和可选的作者元素。xs:complexContent 元素 anotherPublicationType 是根据另一个复杂类型的限制派生的，该复杂类型在 xs:restriction 元素的base属性中指定。通过使用 xs:restriction 元素，您可以定义一个新的复杂类型，它是原始类型的子集，删除或限制某些元素和属性，同时保留其他元素和属性。这允许您创建更具体的类型，这些类型仍然与原始类型兼容，从而更容易重用模式定义。

xs:sequence 元素定义新类型中元素的顺序。 **xs:choice 元素指定可以使用 journalType 组或 conference 元素，但不能同时使用两者。**

xs:element 元素将 author 元素指定为可选元素并允许它出现多次。

## 4. XML Constraints

### Keys and Foreign Keys:

```
<!ELEMENT db (student+, course+) >
<!ELEMENT student (id, name, gpa, taking*)>
<!ELEMENT course (cno, title, credit, taken_by*)>
<!ELEMENT taking (cno)>
<!ELEMENT taken_by (id)>
keys:
student.@id ® student, course.@cno ® course
foreign keys:
taking.@cno Í course.@cno, course.@cno ® course
taken_by.@id Í student.@id, student.@id ® student
```

### Constraints are important for XML

Constraints are a fundamental part of the semantics of the data; XML may not come with a DTD/type – thus constraints are often the only means to specify the semantics of the data

### The limitations of the XML standard (DTD)

ID and IDREF attributes in DTD vs. keys and foreign keys in RDBs(关系型数据库)

- Scoping:
  - ID unique within the entire document while a key needs only to uniquely identify a tuple within a relation
  - IDREF untyped: one has no control over what it points to -- you point to something, but you don't know what it is!
- keys can be multi-valued, while IDs must be single-valued (unary)
- a relation may have multiple keys, while an element can have at most one ID (primary)
- ID/IDREF can only be defined in a DTD, while XML data may not come with a DTD/schema
- ID/IDREF, even relational keys/foreign keys, fail to capture the semantics of hierarchical data – will be seen shortly

```
enroll (sid: string, cid: string, grade:string)
```

### Path expressions

ε 表示空字符串，即没有选择任何内容的路径。

l 代表标签或元素名称，例如“书”或“作者”。

/ 表示路径分离器，表示选择子元素或属性，例如“书/作者”。

// 表示后代或自我轴，允许在任何层次的嵌套中选择元素，例如“ //作者”。

使用这些符号，我们可以构造简单的路径表达式以导航XML树并选择特定元素或属性。

```
( //student, {@id})
( //student, {/name})
( //enroll, {@id, @cno})
( //, {@id})
```

Here are explanations for each of the path expressions:

- ( //student, {@id}) selects all "student" elements in the document and returns their "id" attributes.
- ( //student, {/name}) selects all "student" elements in the document and returns the text content of their "name" child elements.
- ( //enroll, {@id, @cno}) selects all "enroll" elements in the document and returns their "id" and "cno" attributes.
- ( //, {@id}) selects all elements in the document and returns their "id" attributes.

### Summary and Review

- XML is a prime data exchange format.
- DTD provides useful syntactic constraints on documents.
- XML Schema extends DTD by supporting a rich type system
- Integrity constraints are important for XML, yet are nontrivial

### Exercise:

- Design a DTD and an XML Schema to represent student, enroll and course relations. Give necessary XML constraints
- Convert student and course relations to an XML document based on your DTD/Schema
- Is XML capable of modeling an arbitrary relational/object-oriented database?

Designing a DTD and XML Schema for student, enroll and course relations:

### DTD:

```
php Copy code

<!ELEMENT university (student*, course*, enroll*)>
<!ELEMENT student (name, id, dob, address)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT id (#PCDATA)>
<!ELEMENT dob (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT course (cno, cname, instructor, credit)>
<!ELEMENT cno (#PCDATA)>
<!ELEMENT cname (#PCDATA)>
<!ELEMENT instructor (#PCDATA)>
<!ELEMENT credit (#PCDATA)>
<!ELEMENT enroll (eid, sid, cno, grade)>
<!ELEMENT eid (#PCDATA)>
<!ELEMENT sid (#PCDATA)>
<!ELEMENT grade (#PCDATA)>
```

### XML Schema:

```
xml Copy code

<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="university">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="student" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="name" type="xs:string"/>
              <xs:element name="id" type="xs:string"/>
              <xs:element name="dob" type="xs:string"/>
              <xs:element name="address" type="xs:string"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="course" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="cno" type="xs:string"/>
              <xs:element name="cname" type="xs:string"/>
              <xs:element name="instructor" type="xs:string"/>
              <xs:element name="credit" type="xs:integer"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="enroll" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="eid" type="xs:string"/>
              <xs:element name="sid" type="xs:string"/>
              <xs:element name="cno" type="xs:string"/>
              <xs:element name="grade" type="xs:string"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

XML document based on the DTD/Schema:

```
xml Copy code

<?xml version="1.0" encoding="UTF-8"?>
<university>
  <student>
    <name>John Doe</name>
    <id>001</id>
    <dob>1990-01-01</dob>
    <address>123 Main St.</address>
  </student>
  <course>
    <cno>CSE101</cno>
    <cname>Introduction to Computer Science</cname>
    <instructor>Jane Smith</instructor>
    <credit>3</credit>
  </course>
  <enroll>
    <eid>001</eid>
    <sid>001</sid>
    <cno>CSE101</cno>
    <grade>A</grade>
  </enroll>
</university>
```

XML is capable of modeling many aspects of a relational database, but it may not be able to model all aspects of an arbitrary relational or object-oriented database. For example, XML may not be able to model complex.

---



# 3-4 RDF Data

RDF -- Resource Description Framework,

它是 Web 上数据交换的标准模型。 它是一种灵活且可扩展的格式，用于以机器可读的方式表示有关资源的数据和元数据。

在 RDF 中，数据表示为节点和边的图形，其中节点表示资源，边表示它们之间的关系。 图中的每个节点和边都由唯一的 URI 标识，这使它们可以链接到 Web 上的其他资源。

RDF 数据通常以称为 RDF 三元组的格式存储，它由主语、谓语和宾语组成。 主语是被描述的资源，谓语是被断言的属性或关系，宾语是属性或相关资源的值。

RDF 数据用于各种应用程序，包括数据集成、语义搜索和链接数据。 它也是语义 Web 的一项基础技术，旨在通过向网页添加元数据和结构来使机器更容易访问 Web 内容。

<rdf:RDF

xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"

xmlns:xsd="http://www.w3.org/2001/XMLSchema#"

xmlns:uni="http://www.mydomain.org/uni-ns">

<rdf:Description rdf:about="949318">

<uni:name>Yinghui Wu</uni:name>

<uni:title> Professor</uni:title>

<uni:office rdf:datatype="xsd:string">Olin 515</uni:office>

</rdf:Description>

<rdf:Description rdf:about="EECS 433">

<uni:courseName>Database Systems</uni:courseName>

<uni:isTaughtBy>Yinghui Wu</uni:isTaughtBy>

</rdf:Description>

</rdf:RDF>

rdf:RDF 元素是 RDF/XML 文档的根元素，它包含整个 RDF 数据图。

xmlns:rdf 属性为 RDF 元素和属性定义名称空间。

xmlns:xsd 属性定义了 XML 模式数据类型的命名空间。

xmlns:uni 属性为特定于此 RDF 数据图的自定义词汇表定义名称空间。

第一个 rdf:Description 元素表示由 URI“949318”标识的实体。 **URI (Uniform Resource Identifier)**

The "rdf:about" attribute specifies the unique identifier for this resource, which is "949318".

该实体具有使用 uni: 命名空间定义三个属性：uni:name、uni:title 和 uni:office。

uni:name 属性的值为“Yinghui Wu”。

uni:title 属性的值为“Professor”。

uni:office 属性具有值“Olin 515”和指示该值是字符串 (xsd:string) 的 rdf:datatype 属性。

第二个 rdf:Description 元素表示由 URI“EECS 433”标识的实体。 该实体有两个使用 uni: 命名空间定义的属性：uni:courseName 和 uni:isTaughtBy。

uni:courseName 属性的值为“数据库系统”。

uni:isTaughtBy 属性的值为“Yinghui Wu”。

总的来说，这个 RDF 数据图代表了一个名叫“Yinghui Wu”的大学教授的信息，以及“Yinghui Wu”教授的一门名为“数据库系统”的课程。 uni: 命名空间用于为这些实体定义自定义属性，而 rdf:about 属性用于使用 URI 标识实体。

<rdf:Description rdf:about="EECS 433"> -----subject  
    <uni:courseName>Database system </uni:courseName>

```
<uni:isTaughtBy rdf:resource="949318"/> -----object
</rdf:Description>
```

```
<rdf:Description rdf:about="949318">
  <uni:name>Yinghui Wu</uni:name>
  <uni:title> Professor</uni:title>
</rdf:Description>
```

rdf:resource 属性用于指示属性的值是对由 URI 标识的另一个资源的引用。

在您提供的代码中，rdf:resource 属性用于指示 uni:isTaughtBy 属性的值是对 URI“949318”标识的资源的引用。 这表示课程“EECS 433”由 URI“949318”标识的人员教授，该 URI 在您提供的第二个 rdf:Description 元素中进行了描述。

通过在第一个 rdf:Description 元素的 rdf:about 属性和第二个 rdf:Description 元素的 rdf:resource 属性中使用相同的 URI，表明这两个元素描述相同的资源（即，识别的人 通过 URI“949318

**rdf:about** 是 RDF 元素的一个属性，用于指定三元组或语句的**主题(subject)**。 它指示元素正在描述或引用的资源。 例如，<rdf:Description rdf:about="http://example.org/book1"> ... </rdf:Description> 描述了 URI 为 http://example.org/book1 的资源。

**rdf:resource** 是 RDF 属性元素的属性，它指定三元组或语句的**对象(object)**。 它指示属性链接到或描述的资源。 例如，<dc:creator rdf:resource="http://example.org/author1">Author 1</dc:creator> 将 Dublin Core 属性 dc:creator 链接到具有 URI http://example 的资源 .org/author1。

# 4-1 Data Storage management

1 Fixed length records (FLR)