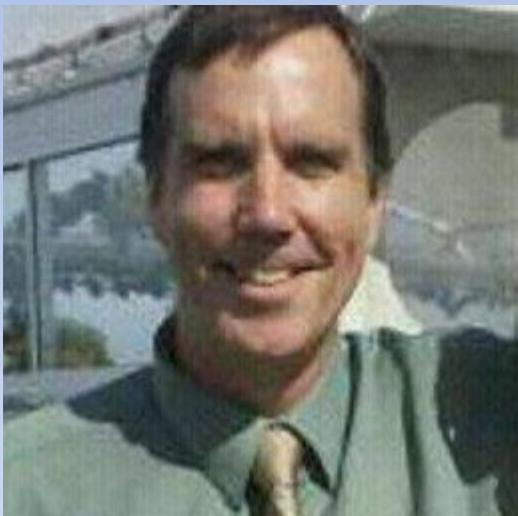


# Computer Science 226, Advanced Database Systems (3 units)



**Class Instructor:**

David Ruby

**Class Hours:**

TTh 5:30 – 6:45

**Office:**

Science II – 273

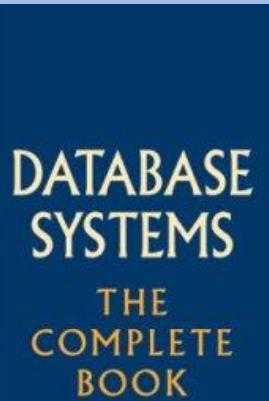
**Email:**

[druby@csufresno.edu](mailto:druby@csufresno.edu)

# *Database Systems: The Complete Book(2<sup>nd</sup>)*

by Hector Garcia-Molina,

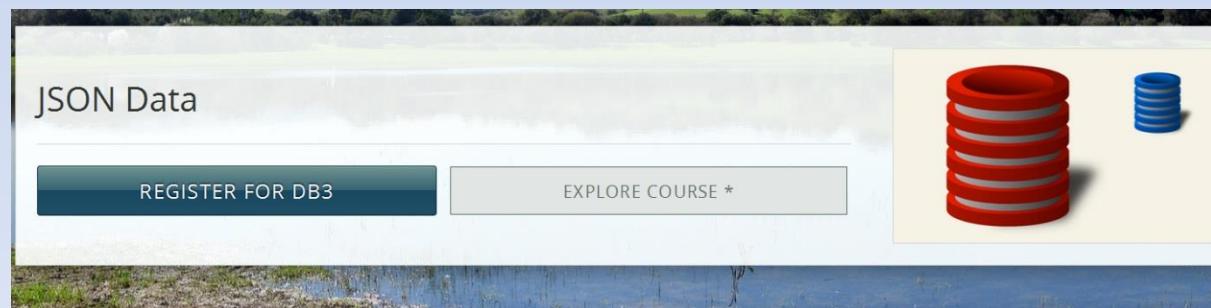
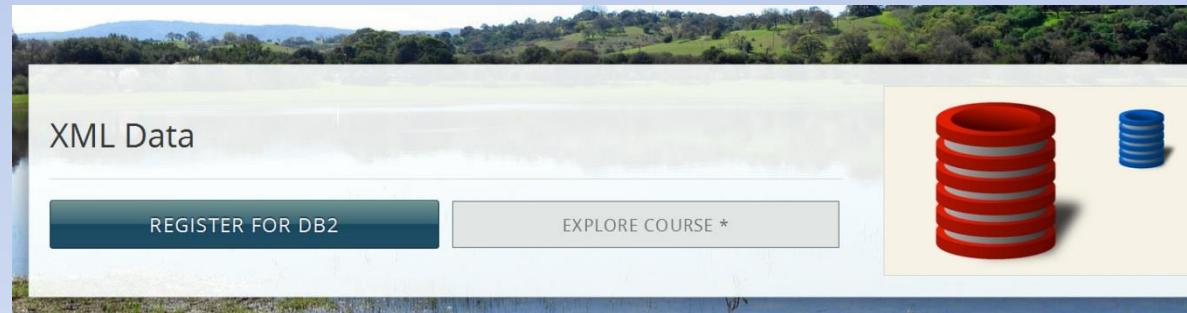
Jeffrey D. Ullman & Jennifer Widom



# Chapter 11:

## Semistructure Data

- XML
  - Well-Formed
  - DTD/XML Schema
- JSON
  - JSON Schema



# **Chapter 12:**

# **Querying**

# **Semistructured Data**

- Xpath/Xquery

# Chapter 2: Relation Algebra

- Operands:
  - Relations
  - Variables representing relations
- Operators:
  - Set operations
  - Slicing Relations
  - Gluing Relations
  - Renaming Relations
- Closure is Needed

# Chapter 5

## Chapter 5

### Algebraic and Logical Query Languages

206 CHAPTER 5. ALGEBRAIC AND LOGICAL QUERY LANGUAGES

We now switch our attention to bags. We start in databases. We start in languages, one algebraic and one logical.

A	B
1	2
3	4
1	2
1	2

Figure 5.1: A bag

#### 5.1.1 Why Bags?

# Chapter 6:

## Relational Data Model: Part 2

### Language: SQL

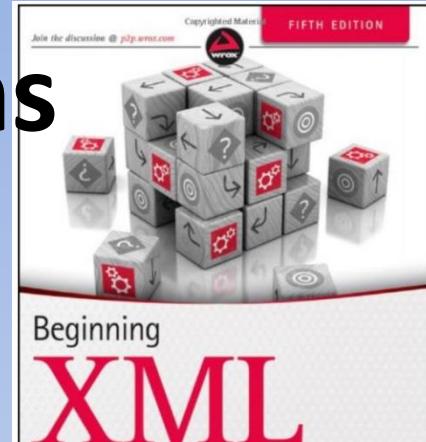
- Two Aspects to language
- Data Definition (ddl):
  - Declaring database schema: tables, constraints, indexes, views.
  - like declaring data/variables in programming language
- Data Manipulation (dml):
  - asking questions (querying) and modifying data.
  - Like executable code within programming language.



# DATABASE SYSTEMS THE COMPLETE BOOK

# Database Systems

- Chapter 11
  - Semistructured Data
  - XML



**Safari**  
Books  
Online

Entire Site ▾

Quin, Danny Ayers

Library

Help ▾ California State University User ▾

## This Book



Beginning XML,  
5th Edition

### Search

### Contents

• Table of Contents

Cover

### Contents

▶ Part I: Introducing XML

▶ Part II: Validation

▶ Part III: Processing

▶ Part IV: Databases

▶ Part V: Programming

▶ Part VI: Communication

▶ Part VII: Display

▶ Part VIII: Case Study

Appendix A: Answers to  
Exercises

< Return to Search Results



Contents

# CONTENTS

## Part I: Introducing XML

### Chapter 1: What is XML?

Steps Leading up to XML: Data Representation and Markups

The Birth of XML

More Advantages of XML

XML in Practice

Summary

# Semistructured Data

- Does not enforce a rigid structure
- Self-Describing
  - Does not have a separate schema declaration
- Lack of structure may make finding data more expensive
- Offers significant advantages for users.
  - Much simpler

# Semistructured Data Representation

- A database of semistructured data is a collection of nodes.
- Nodes are either:
  - Leaf
    - Atomic types like string or number
  - Interior
    - One or more arc outs
    - Arcs are labeled
  - Root node
    - Top level
    - Represents all the data

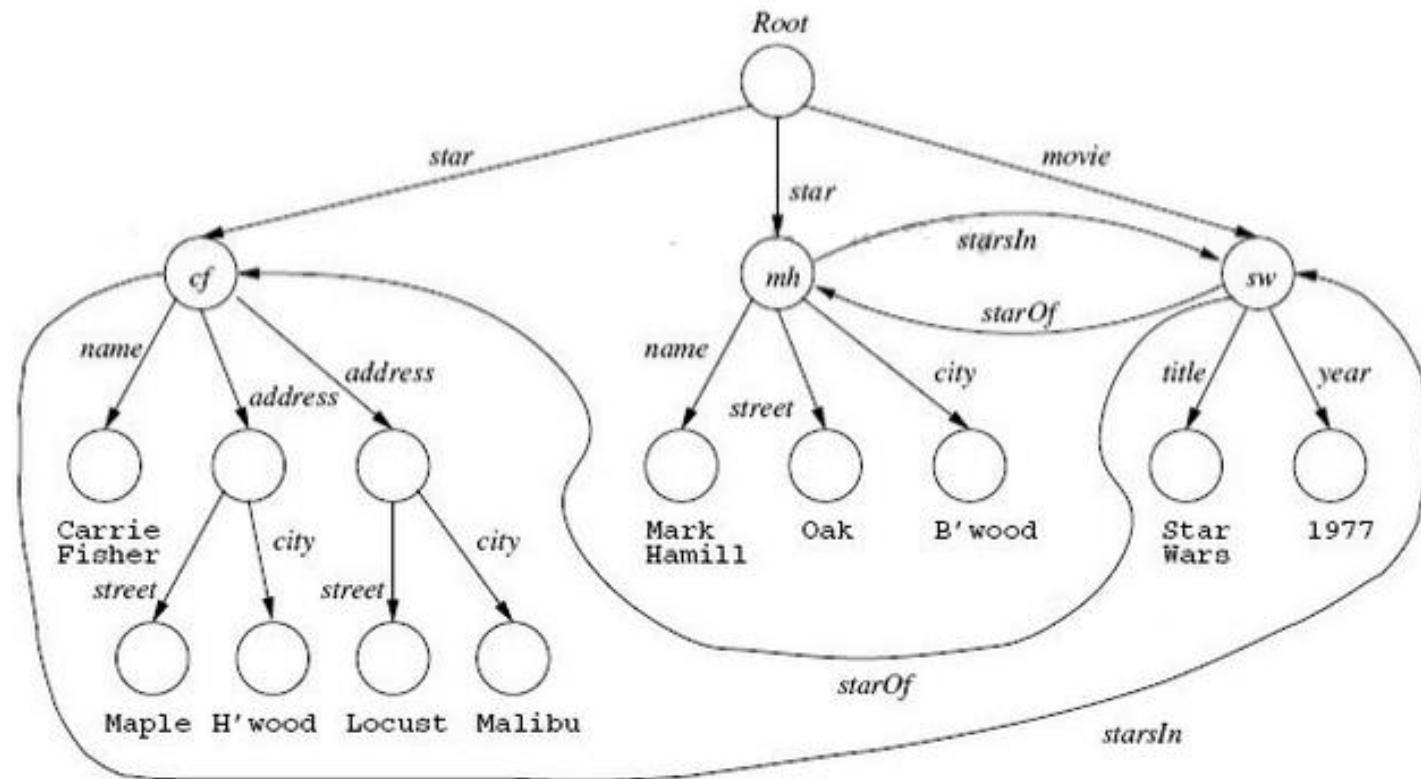


Figure 11.1: Semistructured data representing a movie and stars

- Semistructured data for Stars of Show
- Label L on arc from node N to node M can represent:
  - N is an object, and M is an attribute, L is attribute name
  - N and M objects and L is relationship between them.

# Semistructured Data

- Lets look at storing two actors from Glee in a text file:
  - Corey Monteith
  - Chris Colfer
- We know they are stars, but the computer would now.

```
<Stars updated="2014-07-18">
```

```
  <Star>Corey Monteith</Star>
```

```
  <Star>Chris Colfer</Star>
```

```
<Stars>
```

- XML lets us tag the data for understanding.

# Elements, Attributes, and Content

- The three most common parts of an XML document are
  - *elements*,
  - *attributes*
  - *content*.

- Again:

```
<Stars updated="2014-07-18">
  <Star>Chris Colfer</Star>
</Stars>
```

- Here, Stars & Star are XML elements;
- *updated* is an XML attribute whose value, 2014-07-18, applies to the <Stars> element;
- “Chris Colfer” is **content**, the actual text.

# Well-Formed XML

- An XML document that satisfies the minimum requirements is *well formed*.
- “well-formed” distinguishes basic XML documents from “valid” documents;
  - XML DTD validation is a stricter level of conformance than just “well formed.”
- A document not well-formed is, strictly speaking, NOT an XML document at all.

# The XML Declaration

- XML documents start with an optional *XML Declaration* which:
  - Tells software the document is trying to be in XML;
  - Identifies the version of XML in use;
  - Identifies the character encoding;
  - Tells the XML processor reading the document whether it must first fetch an external “Document Type Definition” (DTD) resource before continuing.

# XML Declaration

```
<?xml version="1.0" encoding="UTF-8"  
standalone="yes"?>
```

- `<?xml .... ?>` contains the XML Declaration.
- `version="1.0"` says the document is using XML version 1.0
- `encoding="UTF-8"` says the document is in the Unicode UTF-8 encoding.

# XML Declaration

```
<?xml version="1.0" encoding="UTF-8"  
standalone="yes"?>
```

- `standalone="yes"` is used only when there is a *Document Type Definition*, or DTD, present.
- So a more usual XML declaration looks like this:  
`<?xml version="1.0" encoding="UTF-8"?>`
- The XML declaration must be at the very start of the XML document
  - do not put blank lines or comments before it.

# XML Elements

- Elements are the basic building block of XML.
  - Every XML document contains at least one element.
- The beginning of an element is represented by a *start tag*:
  - an angle bracket immediately followed by the name of the element, and then a right angle bracket “>”
- The end of an element is </ followed by the name followed by > like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<Stars></Stars>
```

# XML Tags

- There are actually *three* kinds of XML tag:
  - A start tag, also called an open tag: <Stars>
  - An end tag, also called a close tag: </stars>
  - A self-closing tag, also called an empty tag: </stars>
- The self-closing tag is used instead of the start and end tags, and can be used only when an element has no content.
- There must be no space between the < or </ and the name.
- You can have space and newlines after the name and before the closing >-sign:
  - <Stars>

# Example

The screenshot shows the Atom code editor interface with the following details:

- Title Bar:** courses-noID.xml - D:\Documents\GitHub\c040f15 - Atom
- Menu Bar:** File, Edit, View, Selection, Find, Packages, Help
- Toolbars:** A horizontal toolbar at the top with various icons.
- File Tree:** On the left, a tree view of files and folders:
  - c040f15
  - .git
  - cpp
    - ch1
      - ch1.3.2.cpp
      - ch1.4.cpp
      - hw.cpp
    - ch2
    - ch3
    - ch4
      - ch4.ex.1.cpp
      - ch4.ex.2.cpp
      - ch4.ex.2.Spaces.cpp
      - ch4.ex.3.cpp
    - ch5
      - ch5.ex.1.cpp
      - ch5.ex.2.cpp
      - ch5.ex.2.txt
      - ch5.ex.3.cpp
      - ch5.ex.3.txt
      - ch5.ex.3b.cpp
    - addTable1.cpp
    - addTable2.cpp
    - cos.cpp
    - g.bat
    - noise.rand.cpp
    - sin.cpp
    - wumpus.1.cpp
    - zb.pa.1.2.1.cpp
  - doc
    - coding.md
    - cygwin.md
    - git.md
- Code Editor:** The main area displays XML code for a course catalog. The code includes sections for the department (CS), chair, professor, and two courses (CS106A and CS106B). The XML structure is as follows:

```
<?xml version="1.0" ?>
<Course_Catalog>
  <Department Code="CS">
    <Title>Computer Science</Title>
    <Chair>
      <Professor>
        <First_Name>Jennifer</First_Name>
        <Last_Name>Widom</Last_Name>
      </Professor>
    </Chair>
    <Course Number="CS106A" Enrollment="1070">
      <Title>Programming Methodology</Title>
      <Description>Introduction to the engineering of computer applications emphasizing modern software engineering principles</Description>
      <Instructors>
        <Lecturer>
          <First_Name>Jerry</First_Name>
          <Middle_Initial>R.</Middle_Initial>
          <Last_Name>Cain</Last_Name>
        </Lecturer>
        <Professor>
          <First_Name>Eric</First_Name>
          <Last_Name>Roberts</Last_Name>
        </Professor>
        <Professor>
          <First_Name>Mehran</First_Name>
          <Last_Name>Sahami</Last_Name>
        </Professor>
      </Instructors>
    </Course>
    <Course Number="CS106B" Enrollment="620">
      <Title>Programming Abstractions</Title>
    </Course>
  </Department>
</Course_Catalog>
```

- Status Bar:** At the bottom, it shows the file path D:\Documents\GitHub\Fall14\226\L2\courses-noID.xml, line 65, and status information: Unix(LF) UTF-8 XML 1 update.

# XML Names

- An XML name must start with one of:
  - an underscore (\_);
  - any Unicode upper or lower case letter;
- Characters after the first one can also be a dash or digit.
- The colon “:” is also allowed but only for use with XML namespaces, as explained later.
- You can never include spaces or punctuation in XML names, and you can't start names with “XML” (in any mixture of UPPER or lower case)

# Root Element

- All XML documents have exactly one element called the *root* element.
  - The root element start tag is the first start tag in the document, and its close tag is at the very end of the document.
- All the rest of the document is thus “inside” the root element.
- The term *root* comes because an XML document is often viewed as representing a “tree” of elements.

# Nested Elements

- Inside the root element you can have any mixture of free text and more elements.
- Elements must always “nest” properly. You cannot “overlap” tags like this:

```
<verse>
    <line>And they said, <quote>“We must</line>
    <line>take away his shoes!”</quote> but,</line>
</verse>
```

- Remember that there must always be a single outermost root element.

# XML Attributes

- Attributes are name-value pairs associated with elements. You put them inside start tags or inside self-closing tags.
- Attribute values *must* be in quotes. You can use " or '
- You can only have *one* attribute of a given name in any tag.
- There must always be a name, an = sign, and a value:
  - `<option selected = "selected"> . . . </option>`

# Entity References

- An *XML entity* is a named string that can be defined in a DTD and used anywhere.
- You use an entity by referencing it: an entity reference is an &-sign followed by a name and then a semicolon.
- **There are five built-in XML entities: &amp; (&), &apos; ('), &quot; ("), &lt; (<) and &gt; (>). You can always use these.**

# XML Comments

- XML comments are passed back to the parser, but applications usually ignore them. Use <!-- and --> to start and end comments.

```
<!-- This is a comment -->
```

- Comments are not allowed to contain “--” inside them.
- Comments do not nest.
- One comment convention uses stars for more visibility:

```
<!--* This is a comment  
    * over multiple  
    * lines  
    *-->
```

c040f15

.git

cpp

ch1

ch13.2.cpp

ch14.cpp

hw.cpp

ch2

ch3

ch4

ch4.ex1.cpp

ch4.ex2.cpp

ch4.ex.2.Spaces.cpp

ch4.ex3.cpp

ch5

ch5.ex1.cpp

ch5.ex2.cpp

ch5.ex2.txt

ch5.ex3.cpp

ch5.ex3.txt

ch5.ex3b.cpp

addTable1.cpp

addTable2.cpp

cos.cpp

g.bat

noise.rand.cpp

sin.cpp

wumpus.1.cpp

zb.pa.1.2.1.cpp

doc

coding.md

cygwin.md

git.md

README.md

```
1 <?xml version="1.0" ?>
2 <!--Bookstore with no DTD-->
3
4 <Bookstore>
5   <Book ISBN="ISBN-0-13-713526-2" Price="85" Edition="3rd">
6     <Title>A First Course in Database Systems</Title>
7     <Authors>
8       <Author>
9         <First_Name>Jeffrey</First_Name>
10        <Last_Name>Ullman</Last_Name>
11      </Author>
12      <Author>
13        <First_Name>Jennifer</First_Name>
14        <Last_Name>Widom</Last_Name>
15      </Author>
16    </Authors>
17  </Book>
18  <Book ISBN="ISBN-0-13-815504-6" Price="100">
19    <Remark>
20      Buy this book bundled with "A First Course" - a great deal!
21    </Remark>
22    <Title>Database Systems: The Complete Book</Title>
23    <Authors>
24      <Author>
25        <First_Name>Hector</First_Name>
26        <Last_Name>Garcia-Molina</Last_Name>
27      </Author>
28      <Author>
29        <First_Name>Jeffrey</First_Name>
30        <Last_Name>Ullman</Last_Name>
31      </Author>
32      <Author>
33        <First_Name>Jennifer</First_Name>
34        <Last_Name>Widom</Last_Name>
35      </Author>
36    </Authors>
37  </Book>
38 </Bookstore>
```

D:\Documents\GitHub\Fall14\226\L2\Bookstore-noDTD.xml 2:1

Unix(LF) UTF-8 XML 1 update

# XML Namespaces

- In XML, a namespace is a way to group some elements and attributes together under a common label
- Example: you hear people talking about a table:
  - Could be a relational database table...
  - Or a table in a Web page, with a border round it...
  - Or a wooden dining room table with extensible flaps!
- Namespaces give you some context.

# XML Namespaces and URIs

- XML Namespaces use URIs for namespace names.
- Two namespace URIs are considered the same if they are written the same, without unquoting them.
- An XML namespace name does not “point” to anything: it’s just a name.
- URIs were used because there’s already a good distributed mechanism for people to create their own URIs without any fear of conflicts.

# Namespace Example

- Let's edit the *months.xml* file, and make a namespace.
- The namespace identifies who owns the format;
  - we made up the months format, so let's call it ours:

```
<Months xmlns="http://wrox.com/ns/months/">
    <Month index="1">January</Month>
    <Month index="2">February</Month>
    . . .
</Months>
```

# The Default Namespace

- When you use  $xmlns = value$  like this, you are setting a “default namespace.”
- All elements inside the one with the namespace declaration are said to be “in” that namespace.
- Attributes are not in any namespace.
- In the absence of  $xmlns$ , elements and attributes are not in any namespace.
- **Note:** Do not think of  $xmlns$  as an XML attribute. It looks like an attribute, but it is not an attribute.

# Multiple Namespaces 1

- Very often you need to refer to more than one namespace in a document: part of the purpose of namespaces in XML is so you mix elements from different vocabularies.
- You can change the default namespace on any element, for all the elements beneath (inside), but what if you need two namespaces in one element, or if you need to use an attribute from a particular namespace?
- The solution is the *Namespace Prefix*, a short string that lets you work with multiple namespaces.

# Example Using Prefixes

```
<m:Months  
    xmlns:h="http://example.org/holidays/"  
    xmlns:m="http://wrox.com/ns/months/">  
  
<m:Month index="1">  
    <m:Name>January</m:Name>  
    <h:Holiday day="Hangover Day</h:Holiday>  
</m:Month>  
</m:Months>
```

- The prefixes (here *h* and *m*) are *not* part of the element names; they stands for the full namespace name.
- Using a prefix for Months has the same result as using the default namespace, and you can mix the two styles.

# Namespaces on Attributes

- Attributes are in no “no namespace” unless they are explicitly prefixed.
- Usually this is what you want, but some vocabularies define attributes to be used by other vocabularies: here is an example using XLink:

```
<svg version="1.0" xmlns="http://www.w3.org/2000/svg"
      xmlns:xlink="http://www.w3.org/1999/xlink">
  <image x="10" y="10" width="300px" height="310px"
         xlink:href="isaac-newton-300x310.jpg">
    <title>Sir Isaac Newton</title>
  </image>
</svg>
```

# In-Scope Namespaces

- A default namespace declaration affects all unprefixed elements inside it, as well the element with the declaration.
- A namespace declaration with a prefix (sometimes called a *binding*) makes that prefix available to all elements inside it as well as to the element with the declaration.
- For any element, the set of available namespaces is called the *in-scope namespaces*.
- The scope rules for namespaces are similar to variable names in programming languages.

# XML, XMLNS, XSD

- The *xml* prefix is always bound to <http://www.w3.org/XML/1998/namespace> - you don't need to declare it.
- The *xmlns* prefix is always bound to <http://www.w3.org/2000/xmlns/> (note the trailing slash); this one *must not* be declared in a document.
- W3C XML Schema uses <http://www.w3.org/2001/XMLSchema> - by convention usually bound to “*xs*” as a prefix.

# XSLT

- The XSL Transformation language, XSLT, and the XML Path Language, XPath, are the most widely used XML-based languages today, with XQuery a distant third.
- XSLT uses <http://www.w3.org/1999/XSL/Transform> and this is very commonly bound to an *xsl* prefix.
- XPath and XQuery do not use XML syntax, and do not have their own XML namespaces in the same way.
- XSLT also uses a *version* attribute, currently one of "1.0", "1.1", "2.0" or "3.0" in practice.

# XHTML

- XHTML is an XML-based version of HTML, the markup language used for the World Wide Web.
- XHTML uses `http://www.w3.org/1999/xhtml` as its namespace URI.
- XHTML documents can be served as HTML or as XML, but when served as HTML they must use a default namespace, not a prefix.
- XHTML is very widely used.

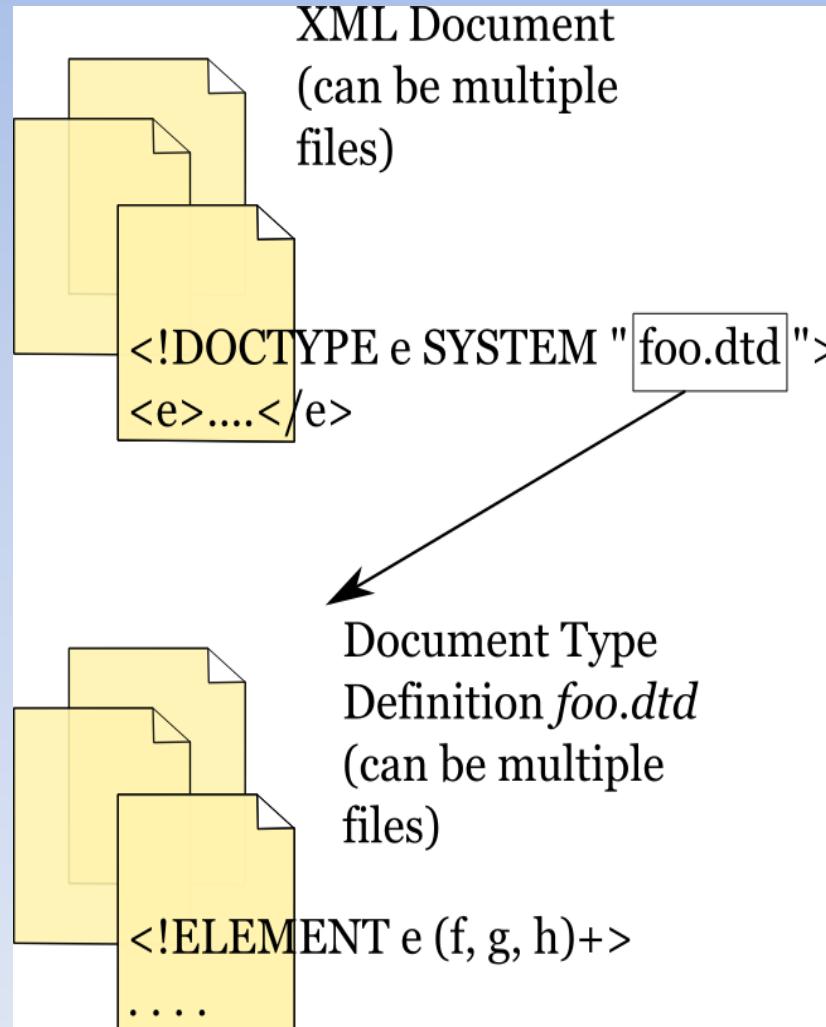
# Document Type Definitions

- A Document Type Definition defines an XML vocabulary, sometimes also loosely called a tagset.
- A DTD is just a plain text file with a particular syntax.
- XML files can contain a pointer to the DTD they use.
- The DTD defines elements, attributes and entities that can appear inside such an XML file.

# Validation

- A DTD defines a formal grammar, a set of rules for documents to follow. If a document follows the rules correctly it is said to *validate against the DTD*.
- The DTD constrains which XML element names can be used, and where
- The DTD can constrain where text goes but does *not constrain what that text can be; for that you need a Schema instead (e.g. XSD or RelaxNG)*
- DTDs are part of the XML specification and are very widely supported.

# Validation



# Linking XML to DTD

- When an XML document uses a DTD, it starts with a DOCTYPE declaration.

```
<!DOCTYPE contacts>
```

- This document must start with an element called contacts but has no explicit DTD.

```
<!DOCTYPE contacts SYSTEM "contacts.dtd">
```

- This document references contacts.dtd using a relative URI reference: the DTD is in the same place as the document, in a file called contacts.dtd.

```
<!DOCTYPE contacts PUBLIC "-//Beginning XML//DTD Contact Example//EN"  
"contacts.dtd">
```

- This document uses the same contacts.dtd file but gives it a name, a “public identifier.”

# Validation

- A validating processor starts by reading the XML document.
- It then encounters the DOCTYPE declaration and finds the link to the DTD.
- It downloads the DTD and reads it into memory.
- The processor then goes back to the XML document and applies the rules in the DTD to the rest of the XML document.

# SYSTEM identifiers

- A “SYSTEM Identifier” is a URI Reference; unlike a namespace, a system identifier can be a relative URI, in which case it’s resolved relative to the resource that contains it.

```
<!DOCTYPE contacts SYSTEM "contacts.dtd">  
  
<!DOCTYPE contacts  
  
    SYSTEM "http://www.example.org/contacts.dtd">
```

- The identifier is like a filename: the contents of the resource identified by the URI Reference are fetched and processed if the XML processor is doing DTD validation.
- Reminder: the separator in a URI is / and not \
- The resource pointed to be a system identifier must be a valid XML DTD; the syntax will be shown in this module.
- Later you will meet other uses for XML SYSTEM identifiers.

# PUBLIC identifiers

- A “Public Identifier” is a name. Public identifiers can’t be dereferenced directly, so they are less useful than system identifiers.

If you supply a public identifier for the DTD you must *also supply a system identifier to be used as a fall-back*:

```
<!DOCTYPE contacts  
      PUBLIC "-//Wiley Inc//DTD for contacts//EN"  
      "http://www.example.org/contacts.dtd">
```

- A public identifier has the syntax  
`-//owner//class description//language//version`
- The class for a Document Type Definition is always DTD, and the //Version part is optional.
- Use an XML Catalog to tell your processor how to dereference a public DTD; XML Catalogs will be covered later.

# DTD Syntax

- Once the XML processor has read the DOCTYPE declaration and used the PUBLIC and/or SYSTEM identifier to fetch the DTD, it reads and parses the DTD.
- The DTD contains:
  - element declarations
  - attribute declarations
  - entity declarations
  - notation declarations
  - comments and processing instructions

# Element Declarations

- An element declaration has three parts:
  - The !ELEMENT keyword
  - the name of the element being declared
  - what can go inside the element
- Example:
- <!ELEMENT student (name, age, height)>
- This declares an element called *student* that *must contain three sub-elements, name, age and height, in that order.*
- The part inside parentheses is called the *content model and is a kind of regular expression.*

# Element Content Models

- There are four kinds of content model:
  - Element Content
  - Mixed Content
  - Empty
  - Unrestricted (ANY)
- The following slides cover each type of content model in turn.

# Element Content 1:

- An XML element that is declared to contain only other elements is said to have *element content*.
- Whitespace is also allowed in element content, and is ignored by the XML parser if it is in validating mode.
- Element content models list the elements that are allowed inside the element being declared and specify their order using a regular expression:
- <!ELEMENT student (name, location, phone)>

# Element Content 2:

- `<!ELEMENT student (name, location, phone)>`
- Here, the element “student” must match the regular expression (name, location, phone).

```
<student>
  <name>Susan</name>
  <location>G51</location>
  <phone>555-9125</phone>
</student>
```

- The whitespace between the name, location and phone elements is ignored by a validating parser.

# Content Model Syntax 1

- Content models are made of *particles*, *each of which can be any of the following:*
  - an element name, a
  - a sequence: a, b, c, meaning the elements must follow each other with nothing except whitespace between them
  - a choice, a | b | c, meaning the input XML document must contain exactly one of an “a” element, a “b” element or a “c” element at that point; a, b and c
  - a particle inside ( parentheses )

# Content Model Syntax 2

- Particles can be followed by an *occurrence indicator*:
  - (none): the particle must occur exactly once in the input document;
  - \* (a star): zero or more times;
  - + (plus): one or more times
  - ? (question mark) zero or one times.

# Sequences

- Sequences use commas between particles:
- `<!ELEMENT name (first, middle, last)>`
- Note: This example is used in the coursebook, but in real applications you should use family-name and given-name, because not all cultures use the same order for their names!

```
<name>  
  <first>Liam</first>  
  <middle>R E</middle>  
  <last>Quin</last>  
</name>
```

- A name element will only validate if it has *exactly one each of first, middle and last elements inside it.*

# Choices

- Choices use vertical bars between particles:
- `<!ELEMENT location (address | GPS)>`
- A location element will validate if it has *exactly one address element or exactly one GPS element: it must have one or the other.*
- You can have long choices as there is no fixed limit to the number of choices:
- `<!ELEMENT a (b | c | d | e | f)>`
- Some DTDs have over 100 elements in a choice, but this can make them very difficult for authors to use.

# Combining Particles

- Content models particles can be combined, using (parentheses) where necessary.
- ```
<!ELEMENT location  
      (address | (latitude, longitude))>
```
- Here, a location element must contain *either exactly one address element or one latitude element followed by one longitude element*.
- If you make the pattern too complicated the XML parser might be happy but people creating documents will hate you, so be careful! It's often best to stick to simple sequences or choices.

# More than One

- Suppose an author has two middle initials:

```
<!ELEMENT name (first, middle, middle, last)>
```

- This works for that author, but a more general approach is usually better in computing:

```
<!ELEMENT name (given, middle*, family)>
```

- The star \* means the name element can contain any number of middle elements, from none at all to a million or more, as long as they are all between the given name and family name.
- There is no direct way to say you can have between zero and ten middle names (but there really *are people with 20 or more middle names!*)

# More than One

- You can also use occurrence indicators on groups:

```
<!ELEMENT name (given, (middle|initial)*, family)>
```

- This will match any number of middle elements or initial elements intermixed in any order (including none at all).
- You can also use + and ? after parentheses or after a name, to mean one or more, or zero or more.
- a+ is the same as (a, a\*)

# Mixed Content 1

- Mixed Content is the second of the four content model types (element, mixed, empty and any). Any element that can contain text is called *mixed*.

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

- This allows just text (#PCDATA, *parsed character data*) and no sub-elements. *The next example allows sub-elements too:*

```
<!ELEMENT p (#PCDATA|em|strong|a|i|b|span)*>
```

- The #PCDATA must be first in the content model, there must be a simple choice and there must be a \* at the end.
- All whitespace is significant inside the p element, because it matches #PCDATA.

# Mixed Content 2

```
<!ELEMENT p (#PCDATA|em|strong|a|i|b|span)*>
```

- Example:

```
<p>This p element has <em>embedded</em> sub-elements  
interspersed with <i>really</i> awesome text</p>
```

- Most document-like text will end up allowing sub-elements, e.g. for emphasis;
- Some languages (Japanese, Chinese and Korean especially) need sub-elements for “ruby” annotation elements, e.g. even in book titles.

# Mixed Content Summary

- Must use the choice separator (|, vertical bar) to separate elements, not the comma.
- #PCDATA must appear first in the list of elements
- There must be no inner content models: no (, ), \*, ?, + or comma inside the outer brackets
- If there are child elements, the \* must appear at the end of the model, outside the brackets.

```
<!ELEMENT a (#PCDATA)>  
<!ELEMENT b (#PCDATA)*>  
<!ELEMENT c (#PCDATA|d|e|f|g)*>
```

# Empty Content

- You can say that a particular element is not allowed to contain anything at all:

```
<!ELEMENT img EMPTY>
```

- Note: there are no parentheses - the keyword EMPTY replaces the usual content model.
- The keyword EMPTY (like #PCDATA and ELEMENT) must be in upper case.
- Empty elements can appear either as `<img/>` or as `<img></img>`
- Whitespace and text are not allowed inside an EMPTY element.

# Unrestricted Content

- An element declared with content model of ANY can contain any mix of elements and text:

```
<!ELEMENT trash ANY>
```

- Any elements that do occur inside an ANY element must themselves be declared in the DTD.
- It's usually a bad idea to use ANY: it can make it harder to write software to process the XML, and it can lead to mistakes in document creation.

# Attributes 1:

- Elements are ideal for containing human-readable content;
- Attributes are best used for:
  - a place for computer-readable information
  - information about elements (metadata)

```
<!ELEMENT a (#PCDATA)*>
<!ATTLIST a
    href CDATA #IMPLIED
>
```

- This declaration allows, e.g.:

```
<a href="I am an attribute">content here</a>
```

# Attributes 2:

- Attribute names have the same rules as element names,
  - except that an element can't have two attributes with the same name.
- There are several attribute types;
  - CDATA attributes (character data) contain text, and are the most common type.
- **Attributes can never contain elements.**
- Attributes can occur in any order on elements.

# Attribute Types

- Attributes can have a **default value** that will be supplied by DTD validation; most XML processing is done without DTDs today, so avoid this.
- Attributes can have a **fixed value**, e.g. to identify a version or even a namespace URI.
- Otherwise, an attribute is either required (#REQUIRED) or optional (#IMPLIED).

```
<!ATTLIST img  
        src CDATA #REQUIRED  
        style CDATA #IMPLIED  
>
```

# Attributes Revisited 1

- There are nine attribute types, not just CDATA that you saw in the previous module.
- Attribute values are *normalized* by turning each sequence of whitespace (including newlines) into a single space.

**1. CDATA: the value is text.**

**2. ID: the value is an XML name, an identifier, which must be unique in the whole document's ID attributes.**

**3. IDREF: the value is an identifier that also occurs as an ID value; the IDREF is said to point to, or refer to, that ID.**

# Attributes Revisited 2

- 4. IDREFS:** a whitespace-separated list of IDREF values.
- 5. ENTITY:** the attribute's value must be the *name* of an external unparsed entity.
- 6. ENTITIES:** a whitespace-separated list of entity names.
- 7. NMTOKEN:** a *name token*, like an ID but not required to be unique.
- 8. NMTOKENS:** a whitespace-separated list of NMTOKENs;
9. An enumeration (see next slide).

# Attributes Revisited 3

- An *enumeration* is a list of allowed values:

```
<!ELEMENT phone (#PCDATA)*>  
<!ATTLIST phone  
        kind (home|work|mobile|fax|unknown) "unknown"  
>
```

- This allows instances like:

```
<phone kind="fax">+33 12 23 45 67 99</phone>  
<phone>extension 7</phone> (kind is “unknown” here)
```

# DTDs: Why and Why Not

- Document Type Definitions are part of the XML standard and very widely supported.
- The content model notation is easy to understand.
- **But You can't control content (e.g. require digits in a phone number) with DTDs.**
- They are not namespace aware.
- No data typing (integer, hatsize...) so they are not useful for connecting XML to programming languages.
- As a result, DTDs are mostly replaced by XSD and RNG.

# XML Schema Overview

- W3C XML Schema is a more powerful and flexible alternative to DTDs.
- The W3C XML Schema specification
  - published in 2001 and the latest version, 1.1, in 2012;
  - most software implements either 1.1 or 1.0 second edition from 2004.
- The actual XML Schema Documents that users create are called XSDs.
- A *schema* is any document that defines the structure of something,
  - so a DTD is a schema;
  - W3C XML Schema is a particular schema language for XML;
- W3C XML Schema is the most widely used form of XML schema today.

# Benefits of W3C XML Schema

- Use basic XML element syntax, not DTD syntax.
- Supports XML Namespaces.
- Can constrain and validate the actual text, not just the elements and attributes.
- Better support for large-scale vocabularies, including abstract types and reusable content models.
- Object inheritance and type substitution closer to modern programming languages.
- User-defined types (e.g. Hatsize) become visible in APIs and languages for manipulating XML.

# The xs:schema element

```
<xs:schema  
    xmlns:xs="http://www.w3.org/2001/XMLSchema"  
    elementFormDefault="qualified"  
    attributeFormDefault="qualified"  
>  
    ...  
</xs:schema>
```

# The xs:schema element

- You can use any prefix, not just xs, but the namespace URI must be <http://www.w3.org/2001/XMLSchema>
- You can make this the default namespace but if you do, *it will bite you later*, so don't.
- Vocabularies defined by XML Schemas can be associated with a namespace;
  - use the *targetNamespace* attribute on the xs:schema element to identify the namespace.
- Use *elementFormDefault="qualified"* if the elements you are defining are in the target namespace, *unqualified* otherwise.
  - You almost always want *qualified* if you have a *targetNamespace*.
- Use *attributeFormDefault* in the same way; you probably want *unqualified* in most cases.

# Qualified vs unqualified

- Elements are said to be *qualified* if they are associated with a namespace.
- In an XML document this can done be with a prefix, e.g. html:div, or with a default namespace.
- Attributes are only qualified if they have a prefix, because unprefixed attributes are always in “no namespace.” Usually you will leave attributeFormDefault to unqualified.

# Content Models and Types

- XML Schema distinguishes *structures* and *data types*. Structures are the elements and attributes; the data types are the basic content such as strings, dates and numbers.
- Content models are based on DTDs, but extend them.
- A content model is part of an element's type.
- Elements that have both structure and content (e.g. subelements or attributes) have a *complex type*; An element that contains only text has a *simple type*.

# Content Models

- Schema provides three main building blocks for content models:
  1. xs:sequence (like , in a DTD)
  2. xs:choice (like | in a DTD)
  3. xs:all (not found in an XML DTD; SGML has & for this)
- You can intermix these building blocks (*particles*) with:
  4. xs:element to mark where elements occur in the input.
  5. xs:group, to make reusable content models.

# xs:sequence with cardinality

- Use *minOccurs* and *maxOccurs* attributes for cardinality:

```
<xs:sequence>
```

```
    <xs:element ref="title" minOccurs="0" maxOccurs="1"/>
```

```
    <xs:element ref="p" minOccurs="1" maxOccurs="unbounded"/>
```

```
</xs:sequence>
```

- This makes the title element optional and allows at most one title, followed by one or more p elements. In a DTD:

(title?, p+)

# xs:sequence with cardinality

- You can also use *minOccurs* and *maxOccurs* on xs:sequence and xs:choice elements:

```
<xs:sequence maxOccurs="unbounded">  
  <xs:element ref="title" minOccurs="0" maxOccurs="1"/>  
  <xs:element ref="p" minOccurs="1" maxOccurs="unbounded"/>  
</xs:sequence>
```

- This allows the whole sequence to repeat (but there must be at least one, as the default for minOccurs and maxOccurs is 1, not zero). In DTD notation: (title?, p+)+

# xs:choice

- Use xs:choice when the input can contain any of a number of elements: (title|p) becomes:

```
<xs:choice>  
  <xs:element ref="title"/>  
  <xs:element ref="p"/>  
</xs:choice>
```

- You can use minOccurs and maxOccurs on xs:choice too, e.g. to make (title|p)\*

# xs:all

- Use xs:all to say that everything listed must occur in the input but in any order
- xs:all must be the *only* content model element: you can't mix xs:all with choice, sequence or element at the same level.
- xs:all can only contain xs:element elements.
- minOccurs must be 0 or 1 on the elements inside, and maxOccurs must be 1.
- Use a choice or sequence instead of xs:all when possible.

# xs:element

- Declaring an element in XML Schema means associating the element name with a type, to say what the element must contain.

```
<xs:element name="name" type="type"  
            ref="global element declaration"  
            form="qualified|unqualified"  
            minOccurs="min" maxOccurs="max",  
            default="do not use", fixed="fixed value">
```

- The element name must be an XML name, starting with a Unicode letter or underscore, and not contain a colon.

# xs:element and @type

- An element can have a *simpleType* or a *complexType*.
  - A *simple type* is a sequence of characters, perhaps interpreted e.g. as a number or date, and maybe constrained in value, e.g. “an integer between 4 and 36 inclusive.”
  - An element with sub-elements or attributes has a *complex type*.
- You can specify the type of the element in three ways:
  1. Creating a *local* type (easiest but causes problems).
  2. Referring to a *global* type (usually simplest).
  3. Referring to another element you already declared globally.

# Global versus Local

- *Global declarations* are declarations that are direct children of the xs:schema element; they can be reused anywhere in the entire XML schema document.
- *Local declarations* are declarations that appear inside some other child or descendent of the xs:schema element; they can be used only in their specific context.
- In most cases in practice, life is simplest if you make all your element and type declarations global. Local types cannot easily be used in XSLT, XPath or XQuery.
- You can also have global elements with local types.

# Our First Complete Schema

```
<xs:schema  
    xmlns:xs="http://www.w3.org/2001/XMLSchema">  
    <xs:element name="entry" type="xs:string" />  
    <xs:element name="dictionary">  
        <xs:complexType>  
            <xs:sequence maxOccurs="unbounded">  
                <xs:element ref="entry"/>  
            </xs:sequence>  
        </xs:complexType>  
    </xs:element>  
</xs:schema>
```

# Understanding The Schema

Start by defining a plain text element called *entry*:

```
<xs:element name="entry" type="xs:string" />
```

Then define a *dictionary* element to be a sequence of *entry* elements: (entry)+

```
<xs:element name="dictionary">
  <xs:complexType>
    <xs:sequence maxOccurs="unbounded">
      <xs:element ref="entry"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

# Using The Schema

- Put the schema in a file called *dict.xsd*;
- Make an XML document :

```
<dictionary>
    <entry>this is an entry</entry>
    <entry>this is another entry</entry>
    <entry>this is also an entry</entry>
</dictionary>
```

- Call it *entries.xml*.

# Global Types

- To make the type in the previous schema be global:
  1. move the ComplexType element to the top level;
  2. then give it a name;
  3. then refer to that name from the *dictionary* element.
- The rest of the schema can stay unchanged, although it's common to put all the types at the beginning and then the element definitions, starting with the document-level element.

# Schema with Global Type

```
<xs:schema
```

```
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="dictType">
    <xs:sequence maxOccurs="unbounded">
      <xs:element ref="entry" />
    </xs:sequence>
  </xs:complexType>
  <xs:element name="dictionary" type="dictType" />
  <xs:element name="entry" type="xs:string" />
</xs:schema>
```

# Mixed Content

- Any element or type can have mixed="true" to say that all text, including whitespace, is significant inside it:

```
<xs:complexType name="entryType" mixed="true">
  <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:element ref="person"/>
    <xs:element ref="place"/>
  </xs:choice>
</xs:complexType>
<xs:element name="entry" type="entryType" />
<xs:element name="person" type="xs:string" />
<xs:element name="place" type="xs:string" />
```

# Mixed Content Instance Document

- This XML document validates with the changed XSD:

```
<dictionary
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-
  instance"
    xsi:noNamespaceSchemaLocation="dict.xsd">
  <entry>this is an entry</entry>
  <entry>This is about
  <place>Oxford</place></entry>
  <entry><person>Elijah</person> wrote
  this.</entry>
</dictionary>
```

# Empty Elements

- Use an empty xs:complexType to define an EMPTY XML element:

```
<xs:element name="pageBreak">  
  <xs:complexType />  
</xs:element>
```

# Declaring Attributes

- Use xs:attribute at the top level to define an attribute you can reference, or inside xs:complexType **as the last child** to add an attribute to an element:

```
<xs:complexType name="pageBreakType">
    <xs:attribute name="page" type="xs:integer" />
</xs:complexType>
<xs:element name="pageBreak" type="pageBreakType"
/>
```

- If you add pageBreak to the xs:choice in entryType, you can then include <pageBreak page="23"/> inside an entry element.

# Optional Attributes

- The xs:attribute element can have a *use* attribute to say whether the attribute is required:
  - use="optional" - the attribute can be omitted; you can also add default="some value" and a schema processor *may* supply the default value on validation; this is not reliable.
  - use="required" - every instance of the element in the input must have this attribute, with a value which can be empty if the attribute's type allows it.
  - use="prohibited" - this is for advanced use, and disallows the attribute, e.g. when one type is based on another.

# Built-in Schema Types 1

## String Types

xs:string	Any XML character data
xs:normalizedString	string with spaces collapsed (specialized)
xs:token	
xs:byte signed)	A number from -128 to 127 (8 bits)
xs:unsignedByte	A number from 0 to 255 (8 bits)
xs:base64Binary data	An ASCII representation of binary
xs:hexBinary data	Base 16 representation of binary

# Built-in Schema Types 2

## Integer Types

xs:integer	A whole number, arbitrary size
xs:positiveInteger	1, 2, 3, 4, etc.
xs:negativeInteger	-1, -2, -3, -4, etc
xs:nonNegativeInteger	0, 1, 2, ...
xs:nonPositiveInteger	0, -1, -2, -3, ...
xs:int, unsignedInt	(32-bit integer)
xs:long, unsignedLong	(64-bit integer)
xs:short, unsignedShort	(16-bit integer)

# Built-in Schema Types 3

## Floating-point Types

xs:decimal

arbitrary precision, e.g.

103.4242421

xs:float

IEEE 32-bit floating point

- In addition to numbers, you can use -0 (which is different from 0 in some systems) INF, -INF and NaN (Not a Number).
- Decimal and float can be positive or negative.
- There is no support for scientific notation (3.6E17).

# Built-in Schema Types 4

## Time and Date Types

xs:time                        e.g. 15:45:17.000 (this is 3:45 pm)

xs:dateTime                  e.g. 2016-07-17T15:45:17.000

xs:date                        e.g. 2016-03-24 (year-month-day)

The date and time formats are defined by ISO 8601;  
although this is not freely available,  
<http://www.w3.org/TR/NOTE-datetime> may be useful.

# Built-in Schema Types 8

## XML Types

- You can also use the XML DTD types: ID, IDREF, IDREFS, ENTITY, ENTITIES, NOTATION, NOTATIONS and NMTOKENS; see the DTD module for more information.
- There are many types to remember! Most important are:
  - xs:integer
  - xs:string
  - xs:decimal
  - xs:dateTime

# Making Your Own Types

- You can define a type based on one of the existing Schema types (or another type of your own) and then *restrict* the value space.
- You can also combine types to make a *union*;
- You can define space-separated *lists* of values.
- To do these, you use an xs:simpleType element and say whether you are making a restriction, a union or a list.

# Derivation By Restriction 1

- Restriction is the usual way to make new types; some examples:
  - A Canadian postal code (M16E 3J12) is a specialized string;
  - A US zip code is a 5-digit integer, or a string with digits and -.
  - A US state abbreviation is a list of allowed values.
- Some restrictions are too complex to represent in an XSD - e.g. that an attribute must contain a prime number, or that a username must be valid according to a database. Restrict the syntax in the Schema and check the value in an application.

# Facets for Numeric Values

- `minExclusive="12"` allows 13, 14, ...
- `minInclusive="12"` allows 12, 13, 14, ...
- `maxExclusive="12"` allows ...9, 10, 11
- `maxInclusive="12"` allows ...10, 11, 12
- `xs:totalDigits="3"` allows 999, but not 1000 (use `totalDigits` on `xs:integer` only)
- `xs:fractionDigits="2"` limits precision for fractions, e.g. 1.33

# A Derivation Example:

## xs:enumeration

- First we define a type that takes xs:string and restricts the set of values to a list of American states:

```
<xs:simpleType name="stateType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="AL" />
    <xs:enumeration value="AK" />
    <xs:enumeration value="AR" />
    <xs:enumeration value="CA" />
    <!-- many more entries here if you like -->
  </xs:restriction>
</xs:simpleType>
```

# A Derivation Example:

## xs:enumeration

- Now we define a type (entryType from previous slides) that includes an attribute of type “stateType”
- Note that xs:attribute must come at the *end* of complexType; putting it at the start is a common mistake!

```
<xs:complexType name="entryType" mixed="true">
  <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:element ref="person"/>
    <xs:element ref="pageBreak"/>
  </xs:choice>
  <xs:attribute name="state" type="stateType" />
</xs:complexType>
```

# A Derivation Example: Complete

```
<xs:simpleType name="stateType">
    <xs:restriction base="xs:string">
        <xs:enumeration value="AL" />
        <xs:enumeration value="AK" />
        <!-- many more entries here if you like -->
    </xs:restriction>
</xs:simpleType>
<xs:complexType name="entryType" mixed="true">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element ref="person"/>
        <xs:element ref="pageBreak"/>
    </xs:choice>
    <xs:attribute name="state" type="stateType" />
</xs:complexType>
<xs:element name="entry" type="entryType" />
```

# Derivations: List Types

- List types are the second of the three types of derivation (restriction with facets, lists, and unions).
- Lists are element or attribute values with multiple items separated by whitespace. Example:

```
<xs:simpleType name="statesVisitedType">
    <list itemType="stateType" />
</xs:simpleType>
```

- An element of this type might look like this:

```
<states-where-I-have-run-barefoot>
    AL NY AK CA MI HI
</states-where-I-have-run-barefoot>
```

# Facets for List Values

- `xs:length="7"` means the list must have contain items.
- A *list* is a value containing multiple items separated by white space (space and newlines).
- `xs:minLength` and `xs:maxLength` are a more flexible way of setting the number of items in a list.
- Setting `xs:length` is the same as setting both `xs:minLength` and `xs:maxLength`.
- `xs:enumeration` can be used to define list items.
- `xs:whiteSpace` can control how the items are separated.
- `xs:pattern` is tested *after* `xs:whiteSpace` treatment.

# Derivations: Union Types

- Union types are the third of the three types of derivation (restriction with facets, lists, and unions).
- Union types allow an instance element or attribute to contain an instance of any of the types you list.
- Example using a provinceType for Canadian provinces:

```
<xs:simpleType name="stateOrProvinceType">  
    <union memberTypes="stateType provinceType" />  
</xs:simpleType>
```

- You can also include the xs:simpleType definitions inside the union element instead of using memberTypes.

# Facets for Union Types

- Union types can be restricted using only two facets:
  - xs:pattern, applied *before* the schema processors determines the type of the supplied value;
  - xs:enumeration, to add a value to the union.
- Note, the value is checked against each type in the union in the order they are given; if you put xs:string first it will always match even if you have xs:integer afterwards. So, put xs:integer before xs:string.

# Namespaces and XML Schema

- W3C XML Schema has strong support for XML Namespaces, unlike DTDs.
- Identify the main “target namespace” for documents;
- bind that namespace to a prefix.
- define element names in that namespace.
- The namespace URI in the schema must match the namespace URI in the document **exactly**, byte for byte. The checking is done without unescaping anything.

# Identifying the Namespace

- Use the targetNamespace attribute on the xs:schema element and bind the namespace to a prefix (**e** in this example):

```
<xs:schema  
    xmlns:xs="http://www.w3.org/2001/XMLSchema"  
    elementFormDefault="qualified"  
    attributeFormDefault="unqualified"  
  
    targetNamespace="http://words.fromoldbooks.org/ns/entry/  
    /1.0"  
    xmlns:e="http://words.fromoldbooks.org/ns/entry/1.0">
```

# Qualified and Unqualified, Revisited

- Elements are *qualified* if they belong to a namespace; you can write

```
<xsd:element name="entry" elementForm="qualified" . . .
```

on every element definition, or use the top-level elementFormDefault to specify a default for all elements in the schema, unless you override it on any of them.

- If the element is marked as qualified it must be in the Schema's target namespace in the XML document.
- Attributes are in no namespace unless explicitly prefixed.

# Namespaces and Schema

- Once you have got a namespace prefix you can define elements as before. **However:**
- elements and types you define will now be in the target namespace; you will need your prefix to refer to them:

```
<xm:complexType name="dictType">
  <xm:sequence maxOccurs="unbounded">
    <xm:element ref="e:entry"/>
  </xm:sequence>
</xm:complexType>

<xm:element name="dictionary" type="xm:dictType" />
```

# Database Systems

- Chapter 12
  - Querying Semistructure Data

DATABASE  
SYSTEMS  
THE  
COMCourseSmart  
BC  
SECOND

A First Course in Database Systems, Third Edition

Exit Reader



Search



Page 517

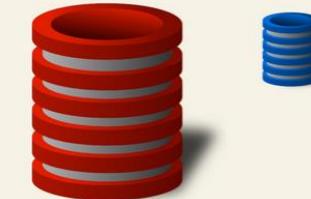
## Chapter 12

# Programming Languages

XPath and XQuery

YOU ARE REGISTERED FOR THIS COURSE

VIEW COURSEWARE



overview

[Return to main page for all Database mini-courses](#)

[ABOUT THIS MINI-COURSE](#)



i Course Number

DB6

¤ Price

Free

# DB6

Courseware Course Info Discussion Wiki Progress Readings Software Guide

Getting Started

Querying XML

XPath Introduction

XPath Demo

XQuery Introduction

XQuery Demo

XML Course-Catalog

XPath and XQuery

Exercises

Exercise

XML Course-Catalog

XPath and XQuery

Exercises Extras

XML World-Countries

XPath and XQuery

Exercises

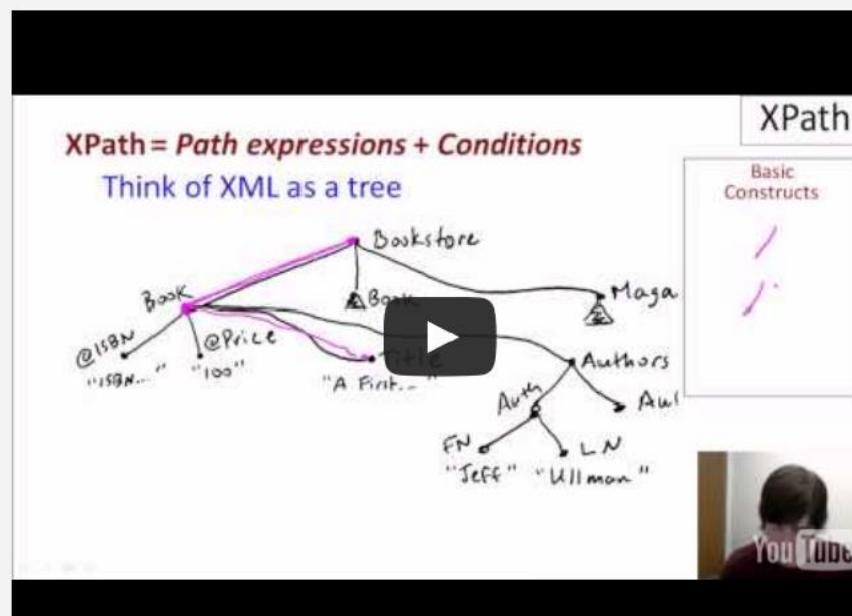
Exercise

XML World-Countries

XPath and XQuery



VIDEO



and I've introduced the concept of navigation axes.

But the real way to learn and understand XPath is to run some queries.

So I urge you to watch the next video which is a demo of XPath queries over our bookstore data and then try some queries yourself.

# XPath, the XML Path Language

- XPath was designed as a way to write down and interchange pointers into document trees from outside.
- XPath is a *domain specific language* (DSL) for XML.
- XPath 1 had its own data model, but many implementations use the DOM. XPath 2 uses the XDM.
- XPath is widely available: in Web browsers, in most programming languages, even in ISO SQL.
- XPath is compact, easy to learn and use, and powerful.

```
--<Course_Catalog>
  --<Department Code="CS">
    <Title>Computer Science</Title>
    +<Chair></Chair>
    +<Course Number="CS106A" Enrollment="1070"></Course>
    +<Course Number="CS106B" Enrollment="620"></Course>
    --<Course Number="CS107" Enrollment="500">
      <Title>Computer Organization and Systems</Title>
      +<Description></Description>
      --<Instructors>
        --<Lecturer>
          <First_Name>Julie</First_Name>
          <Last_Name>Zelenski</Last_Name>
        </Lecturer>
      </Instructors>
      +<Prerequisites></Prerequisites>
    </Course>
    +<Course Number="CS109" Enrollment="280"></Course>
    +<Course Number="CS124" Enrollment="60"></Course>
    +<Course Number="CS143" Enrollment="90"></Course>
    +<Course Number="CS145" Enrollment="130"></Course>
    +<Course Number="CS221" Enrollment="180"></Course>
    --<Course Number="CS228" Enrollment="110">
      +<Title></Title>
      +<Description></Description>
      --<Instructors>
        --<Professor>
          <First_Name>Daphne</First_Name>
          <Last_Name>Koller</Last_Name>
        </Professor>
```

# XPath Expressions

- XPath “expressions” are designed to be like Unix file paths:  
`/Course_Catalog/Department/Title`
- Unlike file paths, an XML element can have more than one child with the same name.
- The above example returns a *list* of all `<Title>` elements directly contained in `<Department>` elements that are directly inside `<Course_Catalog>` elements.
- `/Course_Catalog/Department[1]/Title` returns Title elements inside `<Department>` elements that are the first child of `<Course_Catalog>` elements.

# /Course\_Catalog/Department/Title

```
-<Course_Catalog>
  -<Department Code="CS">
    <Title>Computer Science</Title>
    +<Chair></Chair>
    +<Course Number="CS106A" Enrollment="1070"></Course>
    +<Course Number="CS106B" Enrollment="620"></Course>
    -<Course Number="CS107" Enrollment="500">
      <Title>Computer Organization and Systems</Title>
      +<Description></Description>
      -<Instructors>
        -<Lecturer>
          <First_Name>Julie</First_Name>
          <Last_Name>Zelenski</Last_Name>
        </Lecturer>
      </Instructors>
      +<Prerequisites></Prerequisites>
    </Course>
    +<Course Number="CS109" Enrollment="280"></Course>
    +<Course Number="CS124" Enrollment="60"></Course>
    +<Course Number="CS143" Enrollment="90"></Course>
    +<Course Number="CS145" Enrollment="130"></Course>
    +<Course Number="CS221" Enrollment="180"></Course>
    -<Course Number="CS228" Enrollment="110">
      +<Title></Title>
      +<Description></Description>
      -<Instructors>
        -<Professor>
          <First_Name>Daphne</First_Name>
          <Last_Name>Koller</Last_Name>
        </Professor>
```

Your Query Result:

```
<Title>Computer Science</Title>
<Title>Electrical Engineering</Title>
<Title>Linguistics</Title>
```

# doc("courses.xml")/Course\_Catalog/Department/Title

```
-<Course_Catalog>
  -<Department Code="CS">
    <Title>Computer Science</Title>
    +<Chair></Chair>
    +<Course Number="CS106A" Enrollment="1070"></Course>
    +<Course Number="CS106B" Enrollment="620"></Course>
    -<Course Number="CS107" Enrollment="500">
      <Title>Computer Organization and Systems</Title>
      +<Description></Description>
      -<Instructors>
        -<Lecturer>
          <First_Name>Julie</First_Name>
          <Last_Name>Zelenski</Last_Name>
        </Lecturer>
      </Instructors>
      +<Prerequisites></Prerequisites>
    </Course>
    +<Course Number="CS109" Enrollment="280"></Course>
    +<Course Number="CS124" Enrollment="60"></Course>
    +<Course Number="CS143" Enrollment="90"></Course>
    +<Course Number="CS145" Enrollment="130"></Course>
    +<Course Number="CS221" Enrollment="180"></Course>
    -<Course Number="CS228" Enrollment="110">
      +<Title></Title>
      +<Description></Description>
      -<Instructors>
        -<Professor>
          <First_Name>Daphne</First_Name>
          <Last_Name>Koller</Last_Name>
        </Professor>
```

Your Query Result:

```
<Title>Computer Science</Title>
<Title>Electrical Engineering</Title>
<Title>Linguistics</Title>
```

# XPath Node Types and their Tests

- Here's how to test for each XPath node type:
  - *name* An element of that name
  - *@attr* An **attribute** called *attr* on the current node
    - `data(@attr)`: used to output the @attr value.
  - *\** Any element

# XPath Contexts

- XPath expressions are evaluated in a *context*; this includes the idea of a *current node*.
- XPath isn't a full stand-alone language: it is always used from a “host” language such as PHP.
- The host language defines the context when XPath is invoked;
  - the `current()` function returns the initial context node.
- The context is also changed by starting an expression with `/` (the root node) or `//` (root or any descendant) and in predicates.

# XPath Predicates

- Any XPath sequence can be *filtered* by a *predicate*. A predicate is a test expression in [square brackets] evaluated in the context of some given node: if the predicate returns true than that node is kept.
  - A numeric predicate like [3] is short for [position() = 3]
  - Otherwise, the *effective boolean value* of the expression is determined, and the node is kept if the value is true.
- Example: Course entries for Department with title equal to Computer Science:

```
/Course_Catalog/Department[Title='Computer Science']/course/title
```

# XPath Predicates

- Example: Course entries for Department with title equal to Computer Science:

```
Doc("courses.xml")/  
    Course_Catalog/Department[Title='Computer Science']/Course/Title
```

Your Query Result:

```
<Title>Programming Methodology</Title>  
<Title>Programming Abstractions</Title>  
<Title>Computer Organization and Systems</Title>  
<Title>Introduction to Probability for Computer Scientists</Title>  
<Title>From Languages to Information</Title>  
<Title>Compilers</Title>  
<Title>Introduction to Databases</Title>  
<Title>Artificial Intelligence: Principles and Techniques</Title>  
<Title>Structured Probabilistic Models: Principles and Techniques</Title>  
<Title>Machine Learning</Title>
```

# Effective Boolean Value

- In an expression:
  - An empty sequence is false;
  - Any sequence whose first item is a node is true;
  - A boolean value - `true()`, `false()`, 0 or 1, returns its value;
  - A string (or URI) is false if it has zero length, otherwise true;
  - Numeric values are false if they are 0 (zero) or `NaN` (Not a Number), and true otherwise.
- Note, `true()` and `false()` are boolean constants; `true` and `false` are node tests for `<true>` and `<false>` element names!

# XPath Axes

- An XPath *axis* is a direction, like *up* or *forward*.
- Each / or // represents a “step” in the current direction, or axis; the default direction is “child” so that head/title refers to a title element that’s a child of a head element in the current context.
- You can change direction by naming an axis:
- `/html/body/h1/following-sibling::h2`
- This matches h2 elements in the “following-sibling” direction after the h1 element - *i.e.* at the same level in the tree.

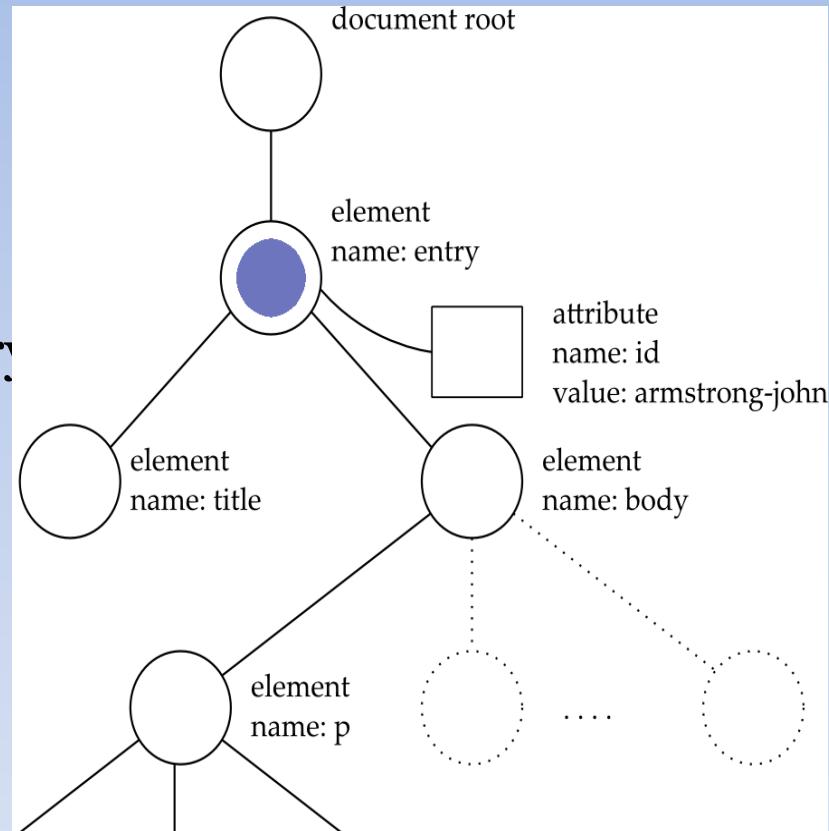
# XPath Axes

- child::            The default, /, “directly contained within”
- descendant::     Also written //, “anywhere underneath”
- attribute::      Also written @, e.g. attribute::\* or @\*
- self::            this node, e.g. self::p is true if this is a <p>
- following-sibling::       After the context node but at the same level in the tree and with the same parent;
- ancestor::        Goes upwards, e.g. ancestor::div returns a list of all the <div> elements anywhere directly above this node.

# Relative paths

If the context node is *entry*

- .. gets you to the document root;
- `body/p[1]` is the first paragraph;
- `title/ancestor::entry` is the entry node;
- `entry//p` is all the p elements;
- `entry/@id` is *armstrong-john*;
- `/entry` is the entry node;
- `//body/p` is the list of all p elements under `<body>`.



# Example

- doc("courses.xml")//Course/Title

Your Query Result:

```
<Title>Programming Methodology</Title>
<Title>Programming Abstractions</Title>
<Title>Computer Organization and Systems</Title>
<Title>Introduction to Probability for Computer Scientists</Title>
<Title>From Languages to Information</Title>
<Title>Compilers</Title>
<Title>Introduction to Databases</Title>
<Title>Artificial Intelligence: Principles and Techniques</Title>
<Title>Structured Probabilistic Models: Principles and Techniques</Title>
<Title>Machine Learning</Title>
<Title>Digital Systems I</Title>
<Title>Digital Systems II</Title>
<Title>From Languages to Information</Title>
```

# Example

doc("courses.xml")/Course\_Catalog//Course/ancestor::Department>Title

Your Query Result:

```
<Title>Computer Science</Title>
<Title>Electrical Engineering</Title>
<Title>Linguistics</Title>
```

# XPath Operators

- Operators take arbitrary XPath expressions. They are mostly the same as other languages:  $3 + 1$ ,  $3 * (2 + 5)$ 
  - Since `/` is “step”, divide is `div` or `idiv` for integer division;
  - `mod` is remainder after division (modulo):  $16 \text{ mod } 7$  gives 2.
- XPath 2 and later also have:
  - `to` gives a sequence of integers - e.g. `1 to 10`
  - `a << b, a >> b` is true if `a` occurs before (after) `b` in the document.
  - `(a, b, c)` builds a sequence.

# Operators Example

- Calculate Population Density

```
- <country name="Afghanistan" population="22664136" area="647500">
  <language percentage="11">Turkic</language>
  <language percentage="35">Pashtu</language>
  <language percentage="50">Afghan Persian</language>
</country>
- <country name="Albania" population="3249136" area="28750"/>
- <country name="Algeria" population="29183032" area="2381740">
  - <city>
    <name>Algiers</name>
    <population>1507241</population>
  </city>
</country>
- <country name="American Samoa" population="59566" area="199"/>
- <country name="Andorra" population="72766" area="450"/>
- <country name="Angola" population="10342899" area="1246700"/>
- <country name="Anguilla" population="10424" area="91">
  <language percentage="100">English</language>
</country>
- <country name="Antigua and Barbuda" population="65647" area="440">
  <language percentage="100">English</language>
</country>
- <country name="Argentina" population="34672996" area="2766890">
  - <city>
    <name>La Matanza</name>
    <population>1111811</population>
  </city>
  - <city>
    <name>Cordoba</name>
    <population>1208713</population>
  </city>
- <city>
```

# Density

```
doc("countries.xml")//country  
[(@population div @area)> 100]  
/data(@name)
```

Your Query Result:

```
Bermuda Gibraltar Holy See Macau Malta Monaco Singapore
```

Firefox

file:///D:/Documents/...xml.json/products.xml

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
- <Products>
  - <Maker name="A">
    - <PC model="1001" price="2114">
      <Speed>2.66</Speed>
      <RAM>1024</RAM>
      <HardDisk>250 </HardDisk>
    </PC>
    - <PC model="1002" price="995">
      <Speed>2.10</Speed>
      <RAM>512 </RAM>
      <HardDisk>250 </HardDisk>
    </PC>
    - <Laptop model="2004" price="1150">
      <Speed>2.00</Speed>
      <RAM>512 </RAM>
      <HardDisk>60</HardDisk>
      <Screen>13.3</Screen>
    </Laptop>
    - <Laptop model="2005" price="2500">
      <Speed>2.16</Speed>
      <RAM>1024</RAM>
      <HardDisk>120</HardDisk>
      <Screen>17.0</Screen>
    </Laptop>
  </Maker>
  - <Maker name="E">
    - <PC model="1011" price="959">
      <Speed>1.86</Speed>
      <RAM>2048</RAM>
      <HardDisk>160</HardDisk>
    </PC>
    - <PC model="1012" price="649">
      <Speed>2.80</Speed>
      <RAM>1024</RAM>
      <HardDisk>160</HardDisk>
    </PC>
    - <Laptop model="2001" price="3673">
      <Speed>2.00</Speed>
      <RAM>2048</RAM>
      <HardDisk>240</HardDisk>
      <Screen>20.1</Screen>
    </Laptop>
    - <Printer model="3002" price="239">
      <Color>false</Color>
      <Type>laser</Type>
    </Printer>
  </Maker>

```

Firefox

file:///D:/Documents/...xml.json/products.xml

```
- <Products>
  - <Maker name="E">
    - <PC model="1011" price="959">
      <Speed>1.86</Speed>
      <RAM>2048</RAM>
      <HardDisk>160</HardDisk>
    </PC>
    - <PC model="1012" price="649">
      <Speed>2.80</Speed>
      <RAM>1024</RAM>
      <HardDisk>160</HardDisk>
    </PC>
    - <Laptop model="2001" price="3673">
      <Speed>2.00</Speed>
      <RAM>2048</RAM>
      <HardDisk>240</HardDisk>
      <Screen>20.1</Screen>
    </Laptop>
    - <Printer model="3002" price="239">
      <Color>false</Color>
      <Type>laser</Type>
    </Printer>
  </Maker>
  - <Maker name="H">
    - <Printer model="3006" price="100">
      <Color>true</Color>
      <Type>ink-jet</Type>
    </Printer>
    - <Printer model="3007" price="200">
      <Color>true</Color>
      <Type>laser</Type>
    </Printer>
  </Maker>
</Products>
```

# Hello, XQuery!

- XQuery is a *domain specific language* for processing querying XML-native databases
  - and data structures that look like type-annotated sequences and trees.
- XQuery expressions operate over instances of the XQuery and XPath Data Model (XDM).
- XQuery implementations can be very fast (petabytes of data, response times in a few milliseconds), rather like ISO SQL used by relational databases.

# XQuery, XPath and XSLT

- XQuery *extends* XPath.
  - Every valid XPath expression is also an XQuery expression, in the right context.
- XSLT *uses* XPath:
  - you embed XPath expressions in XSLT elements.
- Today, XSLT, XPath and XQuery all use the same underlying data model, the XDM.

# XQuery and XPath Differences

- XQuery does not have a default context item: you have to identify the document or collection of documents explicitly.
- The following XPath expression:

```
/dictionary/entry[6]
```

might appear like this in XQuery:

```
doc("dictionary.xml")/dictionary/entry[6]
```

- XQuery is a *bigger* language, and, unlike XPath, can modify existing documents and construct new ones.

# How XQuery Is Used In Practice

- Standalone:
  - as a program that reads an XQuery expression and one or more XML documents and prints a result.
- As part of another language, such as SQL.
- As a library or API from Java, C++, PHP or other languages.
- As the query language for an XML-native database.
- Embedded inside an application or product, such as a customer relationship manager or a stock quote program, or even in mobile devices.

# First XQuery Example

- Pretty much any XQuery implementation can run this:

```
doc("courses.xml")//title
```

- This is a complete XQuery expression and returns all `<title>` elements from the sample document.

# FLWOR, Modules, Functions

- For XQuery the central part is the FLWOR expression (pronounced *flower expression*).
- User-defined functions, and function libraries (modules) are also major parts of XQuery in practice.

# FLWOR Expressions

- **FLWOR** is made up of the following parts:
- **For**: loop through items
- **Let**, to create bindings;
- **Where**, to filter results;
- **Order by**, to sort results;
- **Return**, to return results.
- You must have one or more *for* and/or *let*, and one *return*.

# FLWOR Example

```
for $country in doc("countries.xml")//country
where $country/@population<10000
order by $country/@population
return data($country/@name)
```

- This query returns the name attribute from each `<country>` element in the long sample document,
  - where the entry's *population* attribute is less than 10000.
  - The results are sorted by the *population* attribute as a string.

Your Query Result:

```
Niue Norfolk Island Falkland Islands Svalbard Pitcairn Islands Cocos Islands Saint Helena  
Saint Pierre and Miquelon Christmas Island Holy See
```

# FLWOR Example

```
for $country in doc("countries.xml")//country
where $country/@population<10000
order by xs:integer($country/@population)
return data($country/@name)
```

- This query returns the name attribute from each `<country>` element in the long sample document,
  - where the entry's *population* attribute is less than 10000.
  - The results are sorted by the *population* attribute as an integer.

Your Query Result:

```
Pitcairn Islands Cocos Islands Christmas Island Holy See Niue Norfolk Island Falkland
Islands Svalbard Saint Helena Saint Pierre and Miquelon
```

# FLWOR Example With Let

- The previous example did not use a *let*; here's a new version with a let clause:

```
for $country in doc("countries.xml")//country
let $pop := data($country/@population)
where $pop<10000
order by xs:integer($country/@population)
return data($country/@name)
```

- This version introduces a *binding*, a name for the expression `xs:integer($dude/@died)`
- Reusing the variable for the order by clause means the list is now sorted properly. Variables can help reduce bugs.

# FLWOR works on “tuples”

- The *for* expression generates a list of “tuples” -

```
for $a in 1 to 5, $b in ('a', 'b', 'c')
return <e a="{$a}" b="{$b}" />
```

- The {curly braces} are used to introduce nested XQuery expressions inside constructed elements and attributes.
- The query processor evaluates the *return* clause for every combination of \$a and \$b. It can do the evaluation in any order as long as the result is in the right order, like this:

```
<e a="1" b="a"/>  <e a="1" b="b"/>  <e a="1" b="c"/>
<e a="2" b="a"/>  <e a="2" b="b"/>  <e a="2" b="c"/>
<e a="3" b="a"/>  . . .
```

# Illegal Output

- The examples so far have produced sequences of elements, but XML requires a single outermost element in every document.
- You can use a wrapper *element constructor* and curly braces inside it, to make a legal document, like this:

```
<myNiceResults>{  
    for $a in 1 to 5, $b in ('a', 'b', 'c')  
        return <e a="{$a}" b="{$b}" />  
}</myNiceResults>
```

# Example

```
for $country in
doc("countries.xml")//country
where $country/@population<10000
order by $country/@population
return

<country>
<name>{$country/@name}</name>
<population>{$country/@population}</population>
</country>
```

## Your Query Result:

```
<country>
    <name name='Niue' />
    <population population='2174' />
</country>
<country>
    <name name='Norfolk Island' />
    <population population='2209' />
</country>
<country>
    <name name='Falkland Islands' />
    <population population='2374' />
</country>
<country>
    <name name='Svalbard' />
    <population population='2715' />
</country>
<country>
    <name name='Pitcairn Islands' />
    <population population='56' />
</country>
<country>
    <name name='Cocos Islands' />
    <population population='609' />
</country>
<country>
    <name name='Saint Helena' />
    <population population='6782' />
</country>
```

# Example

```
for $country in
doc("countries.xml")//country
where $country/@population<10000
order by
xs:integer($country/@population)
return

<country>
<name>{$country/@name}</name>
<population>{$country/@population}</population>
</country>
```

## Your Query Result:

```
<country>
  <name name='Pitcairn Islands'/>
  <population population='56'/>
</country>
<country>
  <name name='Cocos Islands'/>
  <population population='609'/>
</country>
<country>
  <name name='Christmas Island'/'>
  <population population='813'/'>
</country>
<country>
  <name name='Holy See'/'>
  <population population='840'/'>
</country>
<country>
  <name name='Niue'/'>
  <population population='2174'/'>
</country>
<country>
  <name name='Norfolk Island'/'>
  <population population='2209'/'>
</country>
<country>
  <name name='Falkland Islands'/'>
  <population population='2374'/'>
</country>
<country>
```

## Your Query Result:

```
<country>
  <name name='Niue'/'>
  <population population='2174'/'>
</country>
<country>
  <name name='Norfolk Island'/'>
  <population population='2209'/'>
</country>
<country>
  <name name='Falkland Islands'/'>
  <population population='2374'/'>
</country>
<country>
  <name name='Svalbard'/'>
  <population population='2715'/'>
</country>
<country>
  <name name='Pitcairn Islands'/'>
  <population population='56'/'>
</country>
<country>
  <name name='Cocos Islands'/'>
  <population population='609'/'>
</country>
<country>
  <name name='Saint Helena'/'>
  <population population='6782'/'>
</country>
```

# Some Actual XQuery Engines

- XQuery engines that parse the XML each time:
  - Saxon implements XSLT and XQuery in Java, .Net and JavaScript.
  - Qizx/open is a Java engine that does not use a database (in the free version).
- XML-Native databases:
  - BaseX, Sedna, dbxml, eXist are open source.
  - MarkLogic has a commercial native-XML database; EMC X-Hive is another.
- XQuery in Relational Databases:
  - Oracle, Microsoft SQL Server and IBM DB2 support XQuery directly.

# Element Constructors

- Element constructors in XQuery look like fragments of XML,
  - They make new nodes that are not in any document tree.
- XPath alone cannot do this:
  - XPath only returns pointers into an existing document tree, not new nodes.
  - This is why XPath can't return an element without its children.
- Inside element constructors, use {braces} to enclose XQuery expressions.
  - These expressions have no default context item.

# Element Constructor Example

- At its simplest, an element constructor just looks like XML.
- The following is a complete XML Query (with no prolog):

```
<entry id="jim">
  <title>Armstrong, James</title>
</entry>
```

- If you add braces, the constructor can contain expressions:

```
<entry id="jim" on="{ fn:current-date() }">
  <title>{
    doc("dates.xml")//entry[@id eq "jim"]/title
  }</title>
</entry>
```

# Element Constructor as a Value

- You can use an element constructor anywhere you can use any other value in XQuery:

```
let $jim := <entry id="jim">
    <title>Armstrong, James</title>
</entry>
return $jim//title
```

- You can include sub-expressions, too:

```
let $nums := <numbers>{
    for $i in 1 to 10 return <n>{$i * $i}</n>
}</numbers>
return $nums/n[position() gt 3]
```

# Computed Element Constructors

- Alternate syntax:

```
let $nums := element numbers {
    for $i in 1 to 10
        return element n {$i * $i}
}
return $nums/n[position() gt 3]
```

- You can mix the styles, but this can be confusing:

```
let $nums := element numbers {
    for $i in 1 to 10 return <n>{$i * $i}</n>
}
return $nums/n[position() gt 3]
```

# Computed Constructors

- The computed element constructors get their name because you can have an expression instead of a name:

```
let $name := "sock"  
return element { $name } { "argyle" }
```

- The braces show that it's an expression. You can also have computed attribute constructors:

```
element sock {  
    attribute clean { "yes" },  
    "argyle"  
}
```

Result: <sock clean="yes">argyle</sock>

# Where and Predicates

- If you have only one bound variable in your FLWOR expression, you could use a predicate:

```
for $a in (1 to 11)[. mod 2 eq 0]
return $a (: a sequence of even numbers :)
```

- Predicates affect a single sequence, but the where clause operates on tuples, so you can't easily use predicates if you have more than one bound variable.
- Some implementations optimize predicates and where clauses differently, so there may also be performance differences.

# The *order by* Clause 1

- Use the *order by* clause to sort the tuples. The syntax is:  
[stable] order by *expr* [ascending] [empty least]
- Use *ascending* or *descending* for smallest first or largest first.
- Use *empty least* or *empty greatest* to say where null values go in the result: all at the start or all at the end.
- The value of *expr* is used as a sort key for each tuple.
- If the expression has the same value for two tuples, the *stable* keyword forces those tuples to remain in their original order.

# The *return* Clause

- The *return* clause is evaluated once for each tuple that has not been filtered out by a false *where* clause.
- The result of FLWOR expression is a sequence of items, each item generated by one evaluation of *return*.
- XQuery sequences do not nest:
  - nested sequences are flattened into a single sequence containing all the items:

$(1, 2, 3, (4, (5), 6))$  is  $(1, 2, 3, 4, 5, 6)$

# Grouping 1

- Let's sort a dictionary so everyone born in the same year is together:
- We'll group by the *born* attribute on each entry, and also sort the groups by the same value:

```
for $e in /dictionary/letter/entry[@born]
let $born := xs:integer($e/@born)
group by $born stable order by $born ascending
return <g born="{$born}">{$e}</g>
```

- Now each time the *return* expression is evaluated, \$e will be a sequence of all entry elements with the same *born*.

# Grouping 2

- The results look like this (showing just entry titles here):

```
<g born="100">Palafox, John De</g>
```

```
<g born="102">Theophrastus</g>
```

```
· · ·
```

```
<g born="1614">
```

Annesley, Arthur

Carrenno De Miranda, Don Juan

Dod, John

```
· · ·
```

```
</g>
```

```
· · ·
```

# Try and Catch (XQuery 3)

- The *try* and *catch* expressions evaluate an expression and allow the *catch* expression to handle the error.
- You can only catch dynamic errors, not (for example) syntax errors or obvious static errors.

```
for $i in (1, 2, 3, 0, 12)
return try { 12 div $i }
catch * { 42 }
```

- result: 12 6 4 42 1

# *Switch* expressions (XQuery 3)

- Use *switch* to choose between multiple choices.

```
for $word in ("apple", "boy", "girl", "orange")
return concat(
    switch (substring($word, 1, 1))
        case "a" case "e" case "i" case "u" case "o"
            return "an"
        default return "a",
    " ", $word, "&#xa;")
```

- Each *case* is followed by a single expression.
- The *default* value is used if the switch expression is not `deep-equal()` to any of the case expressions.
- Result: an apple a boy a girl an orange

# Equality in XPath

- The expression `fred = george`, where *fred* and *george* are nodelists or sequences, is true if-and-only-if some item in *fred* is equal to some item in *george*.
- This is called *implicit existential quantification* in logic.
  - $(1, \textcolor{blue}{2}, 3) = (\textcolor{blue}{2}, 4, 6, 8)$  is true.
  - $(\textcolor{blue}{1}, 2, 3) \neq (\textcolor{blue}{2}, 4, 6, 8)$  is true because  $1 \neq 2$  ( $1$  not equal to  $2$ ).
  - $\text{not}( (1, 2, 3) = (2, 4, 6, 8) )$  is false -  $\text{not}()$  negates its argument.
  - $() = ()$  is false (empty sequences, no items in common).
- `/play/scene[character = "Caliban"]` finds all the scenes in which one or more of the characters is called Caliban (probably in *The Tempest*).

# Variables

```
for $born in /dictionary/entry/@born  
    return xs:integer(@born) idiv 100
```

```
let $np := count(//p) return //entry[@died = $np]
```

# Expressions

- “for”, “let” (already mentioned).
- if (e1) then e2 else e3; the else part is required but you can use the empty sequence, (), in many cases.
- “some...satisfies” and “every...satisfies” (boolean), e.g.

```
some entry in //entry satisfies @born = @died
```

- Define functions inline and also refer to functions.
- `matches()` and `replace()` use *regular expressions*.
- Many more functions added, e.g. `sqrt()`. See Book Appendix.

## Q6 (1/1 point)

Return the countries with the highest and lowest population densities. Note that because the "/" operator has its own meaning in XPath and XQuery, the division operator is infix "div". To compute population density use "(@population div @area)". You can assume density values are unique. The result should take the form:

```
<result>
  <highest density="value">country-name</highest>
  <lowest density="value">country-name</lowest>
</result>
```

**Note:** Your solution will need to reference **doc("countries.xml")** to access the data.

*(This problem is quite challenging; congratulations if you get it right.)*

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <countries>
4   <country name="Afghanistan" population="22664136" area="647500">
5     <language percentage="11">Turkic</language>
6     <language percentage="35">Pashtu</language>
7     <language percentage="50">Afghan Persian</language>
8   </country>
9   <country name="Albania" population="3249136" area="28750"/>
10  <country name="Algeria" population="29183032" area="2381740">
11    <city>
12      <name>Algiers</name>
13      <population>1507241</population>
14    </city>
15  </country>
16  <country name="American Samoa" population="59566" area="199"/>
17  <country name="Andorra" population="72766" area="450"/>
18  <country name="Angola" population="10342899" area="1246700"/>
19  <country name="Anguilla" population="10424" area="91">
20    <language percentage="100">English</language>
21  </country>
22  <country name="Antigua and Barbuda" population="65647" area="440">
23    <language percentage="100">English</language>
24  </country>
25  <country name="Argentina" population="34672996" area="2766890">
26    <city>
27      <name>La Matanza</name>
28      <population>1111811</population>
29    </city>
30    <city>
31      <name>Cordoba</name>
32      <population>1208713</population>
33    </city>
```

### Q6 (1/1 point)

Return the countries with the highest and lowest population densities. Note that because the "/" operator has its own meaning in XPath and XQuery, the division operator is infix "div". To compute population density use "(@population div @area)". You can assume density values are unique. The result should take the form:

```
<result>
  <highest density="value">country-name</highest>
  <lowest density="value">country-name</lowest>
</result>
```

**Note:** Your solution will need to reference `doc("countries.xml")` to access the data.

*(This problem is quite challenging; congratulations if you get it right.)*

- Calculate density
- Take max
- Take min

### Q6 (1/1 point)

Return the countries with the highest and lowest population densities. Note that because the "/" operator has its own meaning in XPath and XQuery, the division operator is infix "div". To compute population density use "(@population div @area)". You can assume density values are unique. The result should take the form:

```
<result>
  <highest density="value">country-name</highest>
  <lowest density="value">country-name</lowest>
</result>
```

**Note:** Your solution will need to reference `doc("countries.xml")` to access the data.  
*(This problem is quite challenging; congratulations if you get it right.)*

- Calculate density for all countries:

```
doc("countries.xml")//country/(@population div @area)
```

- Take max

- use the Xquery function: `max()`

```
max(doc("countries.xml")//country/(@population div @area))
```

- Assign the value to a XQuery Variable

```
let$x=max(doc("countries.xml")//country/(@population div @area))
```

# Element Constructors

Q6 (1/1 point)

Return the countries with the highest and lowest population densities. Note that because the "/" operator has its own meaning in XPath and XQuery, the division operator is infix "div". To compute population density use "`(@population div @area)`". You can assume density values are unique. The result should take the form:

```
<result>
  <highest density="value">country-name</highest>
  <lowest density="value">country-name</lowest>
</result>
```

**Note:** Your solution will need to reference `doc("countries.xml")` to access the data.

*(This problem is quite challenging; congratulations if you get it right.)*

- We need to construct XML to match the desired output:

`<highest density="value">country-name</highest>`

# Element Constructors

- We need to construct XML to match the desired output:

```
<highest density="value">country-name</highest>
```

```
<highest density='{$x}'>
  { doc("countries.xml")//country[(@population div @area)=$x]/data(@name) }
</highest>
```

# Element Constructors

- We need to construct XML to match the desired output:

```
<result>
  <highest density="value">country-name</highest>
  <lowest density="value">country-name</lowest>
</result>
```

PARTIAL SOLUTION IN XQUERY:

```
<result>
{
  let $x := max(doc("countries.xml")//country/(@population div @area))
  return
    <highest density='{$x}'>
      { doc("countries.xml")//country[@population div @area]=$x]/data(@name) }
    </highest>
}
</result>
```

# Partial Solution

```
1 <result>
2
3   {let $x := max(doc("countries.xml")//country/(@population div @area))
4   return
5     <highest density='{$x}'>
6       {doc("countries.xml")//country[@population div @area]=$x]/data(@name) }
7     </highest>
8
9   </result>
10
```



Incorrect

**Incorrect**

Your Query Result:

```
<result>
  <highest density='31052.3125'>Macau</highest>
</result>
```

# Hello, XSLT

- The Extensible Style Sheet Language (Transformations),
  - Designed so you could take an arbitrary XML document and transform it into one you knew how to format for print or screen rendering.
- XSLT is often used to turn XML into HTML, JSON, Comma Separated Values (CSV) files and more.
- Very common:
  - generate Web pages from XML;
  - convert from one application's format to another.

# XSLT is Declarative

- *procedural* language such as C, BASIC, Python or Java have you tell the computer how to do what you want.
- *declarative* languages have you describe what you want and the computer figures out the best way.
- SQL is also *declarative*.
  - both SQL and XSLT can be executed in parallel.
  - declarative languages can be highly optimized.

# Some XSLT Software

- Web browsers include XSLT
- On Linux, *xsltproc*, part of *libxml2*, implements XSLT
  - Included in most distributions.
- Most other programming languages use *libxml2* and *libxslt*
  - XSLT

# Sample XML Input Document

```
<People>
  <Person bornDate="1874-11-30" diedDate="1965-01-24">
    <Name>Winston Churchill</Name>
    <Description>
      Winston Churchill was a mid 20th century British politician who
      became famous as Prime Minister during the Second World War.
    </Description>
  </Person>
  <Person bornDate="1917-11-19" diedDate="1984-10-31">
    <Name>Indira Gandhi</Name>
    <Description>
      Indira Gandhi was India's first female prime minister and was
      assassinated in 1984.
    </Description>
  </Person>
</People>
```

# XSLT Syntax and Basic Elements

- XSLT uses XML syntax.
- The first (outermost, root) element of an XSLT stylesheet (as they are called) is:

```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <!--* rest of XSLT here *-->
</xsl:stylesheet>
```

- The version attribute can be 1.0, 2.0 or 3.0, depending on the XSLT version you're using.
- Be careful to get the namespace URI exactly right.

# XSLT Templates

- **The `xsl:template` element is central to XSLT.**

```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
        <!-- basic output here -->
    </xsl:template>
</xsl:stylesheet>
```

- This “XSLT instruction” says to the XSLT processor,
  - whenever you encounter a node matched by the *match* attribute,
  - evaluate the contents of this template in that context.
- The `match="/"` attribute means the template will be used (or *called*) when the XSLT processor reaches the root node (/) of the input XML document.
- The output will be empty:
  - the XML comment won’t even be seen by XSLT,
  - so the template is in effect empty.

# XSLT Context

- XSLT inherits the XPath concept of *context*:
  - there is always a current node list being processed and a current node called the *context node*.
- Relative expressions are evaluated in the current context.
- Within the template for / in the previous example,
  - the XPath expression Person/People would return a list of <People> elements
    - 2 in the example;
- The XPath expression Person would not return anything, as there is no Person child element of the root node, only People.

# Adding Output to a Template 1

- Anything you put inside <xsl:template> elements will be copied to the output.
- Consider the following version of the template for /

```
<xsl:template match="/">
  <html>
    <head>
      <title>Famous People</title>
    </head>
    <body>
      <h1>Famous People</h1>
      <hr />
    </body>
  </html>
</xsl:template>
```

# Adding Output to a Template 2

- The template in the example on the previous slide always generates the *same* output, regardless of its input:

```
<html>
  <head>
    <title>Famous People</title>
  </head>
  <body>
    <h1>Famous People</h1>
    <hr />
  </body>
</html>
```

# Chapter 12

544

CHAPTER 12. PROGRAMMING LANGUAGES FOR XML

where  $x$  is the name of a battle and  $y$  the name of a ship in the battle. There may be more than one `Ship` element in the sequence.

**! Exercise 12.2.3:** Solve the problem of Section 12.2.5; write a query that finds the star(s) living at a given address, even if they have several addresses, without finding stars that do not live at that address.

**! Exercise 12.2.4:** Do there exist expressions  $E$  and  $F$  such that the expression `every $x in E satisfies F` is true, but `some $x in E satisfies F` is false? Either give an example or explain why it is impossible.

## 12.3 Extensible Stylesheet Language

XSLT (Extensible Stylesheet Language for Transformations) is a standard of the World-Wide-Web Consortium. Its original purpose was to allow XML documents to be transformed into HTML or similar forms that allowed the document to be viewed or printed. However, in practice, XSLT is another query language for XML. Like XPath or XQuery, we can use XSLT to extract data from documents or turn one document form into another form.

```
<? xml version="1.0" encoding="utf-8" standalone="yes" ?>
<Movies>
    <Movie title = "King Kong">
        <Version year = "1933">
            <Star>Fay Wray</Star>
        </Version>
        <Version year = "1976">
            <Star>Jeff Bridges</Star>
            <Star>Jessica Lange</Star>
        </Version>
        <Version year = "2005" />
    </Movie>
    <Movie title = "Footloose">
        <Version year = "1984">
            <Star>Kevin Bacon</Star>
            <Star>John Lithgow</Star>
            <Star>Sarah Jessica Parker</Star>
        </Version>
    </Movie>
    ...
</Movies>
```

# Templates!

```
1)  <? xml version = "1.0" encoding = "utf-8" ?>
2)  <xsl:stylesheet xmlns:xsl =
3)      "http://www.w3.org/1999/XSL/Transform">
4)      <xsl:template match = "/">
5)          <HTML>
6)              <BODY>
7)                  <B>This is a document</b>
8)              </body>
9)          </html>
10)     </xsl:template>
11) </xsl:stylesheet>
```

# Adding Output to a Template 2

- The template in the example on the previous slide ALSO always generates the *same* output, regardless of its input:

```
<HTML>
  <BODY>
    <B>This is a document</b>
  </body>
</html>
```

# value-of

- The `xsl:value-of` instruction has a *select* attribute containing an XPath expression;
- it
  - evaluates the expression in the current context,
  - converts the result to plain text,
  - and returns it in place of the `value-of` instruction.

```
<? xml version="1.0" encoding="utf-8" standalone="yes" ?>
<Movies>
    <Movie title = "King Kong">
        <Version year = "1933">
            <Star>Fay Wray</Star>
        </Version>
        <Version year = "1976">
            <Star>Jeff Bridges</Star>
            <Star>Jessica Lange</Star>
        </Version>
        <Version year = "2005" />
    </Movie>
    <Movie title = "Footloose">
        <Version year = "1984">
            <Star>Kevin Bacon</Star>
    
```

## 12.3. EXTENSIBLE STYLESHEET LANGUAGE

547

- 1)   <? xml version = "1.0" encoding = "utf-8" ?>
- 2)   <xsl:stylesheet xmlns:xsl =
- 3)        "http://www.w3.org/1999/XSL/Transform">
- 4)        <xsl:template match = "/Movies/Movie">
- 5)           <xsl:value-of select = "@title" />
- 6)           <BR/>
- 7)        </xsl:template>
- 8)   </xsl:stylesheet>

OUTPUT:  
Movie Title &  
HTML Break

Tag

Figure 12.23: Printing the titles of movies

# Recursive Application

- Need to re-apply templates to subelements!

```
<xsl:apply-templates select = "expression" />
```

# The `xsl:apply-templates` instruction

- When the XSLT processor encounters an `xsl:apply-templates` instruction inside a template,
  1. It looks at the *select* attribute and uses it to generate a list of nodes
    - if no *select* attribute is given it uses `node()` to make a list of everything in the current context.
  2. For each node in that list, the processor looks for the most suitable `xsl:template` element to use based on the *match* attribute of each template in the stylesheet.
  3. The result of `apply-templates` is then the result of all of the templates that were used, joined together.

# Example of apply-templates

```
<xsl:template match="/">

<html>
  <head>
    <title>Famous People</title>
  </head>
  <body>
    <h1>Famous People</h1>
    <hr />
    <ul>
      <xsl:apply-templates select="People/Person" />
    </ul>
  </body>
</html>
</xsl:template>

<xsl:template match="Person">
  <li><!-- Person details here --></li>
</xsl:template>
```

```
1  <People>
2   <Person bornDate="1874-11-30" diedDate="1965-01-24">
3     <Name>Winston Churchill</Name>
4     <Description>
5       Winston Churchill was a mid 20th century British politician who
6       became famous as Prime Minister during the Second World War.
7     </Description>
8   </Person>
9   <Person bornDate="1917-11-19" diedDate="1984-10-31">
10    <Name>Indira Gandhi</Name>
11    <Description>
12      Indira Gandhi was India's first female prime minister and was
13      assassinated in 1984.
14    </Description>
15  </Person>
16 </People>
```

# The xsl:value-of Element

- A common mistake is to think `xsl:value-of select="Person"` will produce XML elements in the output;
- it won't, because value-of just takes the string value of tree nodes.

# Person template with value-of

```
<xsl:template match="Person">  
  <li><xsl:value-of select="Name"/></li>  
</xsl:template>
```

- Templates are evaluated in the context of their *match* expressions;
  - the current element is a `<Person>`, and has a `<Name>` sub-element, so `select="Name"` works.
- Reminder of the input:

```
<Person bornDate="1874-11-30" diedDate="1965-01-24">  
  <Name>Winston Churchill</Name>  
  <Description>  
    Winston Churchill was  
    . . .
```

# Result of apply-templates example

```
<html>
  <head>
    <title>Famous People</title>
  </head>
  <body>
    <h1>Famous People</h1>
    <hr>
    <ul>
      <li>winston Churchill</li>
      <li>Indira Gandhi</li>
      <li>John F. Kennedy</li>
    </ul>
  </body>
</html>
```

*(your output will probably not be indented by default)*

# Using xsl:apply-templates without select

- Change this:

```
<xsl:template match="Person">
  <li><xsl:value-of select="Name" /></li>
</xsl:template>
```

- To this:

```
<xsl:template match="Person">
  <li><xsl:apply-templates /></li>
</xsl:template>
```

- The output will now include the text from the Name and Description elements, all jumbled together. To sort it out, you'll need to add separate templates.

# Adding Templates

- Add the following two templates:

```
<xsl:template match="Name">
  <h2><xsl:apply-templates /></h2>
</xsl:template>

<xsl:template match="Description">
  <p><xsl:apply-templates /></p>
</xsl:template>
```

- Now the output looks neat and tidy.
- In addition, if the Description element is changed to contain sub-elements, you can handle them by writing a template for each sub-element, without changing the template for Description.

# What Happens?

- When the XSLT processor starts, it calls `apply-templates` on the document root node.
- It finds the template for the root node (/) that we wrote.
  - That template itself calls `apply-templates`.
  - The `People` element will be matched;
  - we don't have a template for it, so a default built-in template is used... which calls `apply-templates`...
    - Inside `<People>` we have a list of `<Person>` elements, which get processed one by one with the `Person` template...

# Templates, Functions and Recursion

- XSLT templates are really like recursive functions, with *apply-templates* being used to choose the next level of templates to use.
- The “input” to the function at each stage is a single node in the input tree together with the context.
- The result of the template is a partial tree of nodes and values, called in XSLT a *result tree*.
- You can also call templates explicitly, by giving them names.



# XSLT

Databases - DB7

Started - Jun 09, 2014



[View Course](#)

Your final grade: **100%**.

[Download Statement \(PDF\)](#)



# XPath and XQuery

Databases - DB6

Started - Jun 09, 2014



[View Course](#)

Your final grade: **100%**.

[Download Statement \(PDF\)](#)

# DB7

## Q1 (1/1 point)

Return a list of department titles.

Your solution should fill in the following stylesheet:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="...>
        ... template body ...
    </xsl:template>
    ... more templates as needed ...
</xsl:stylesheet>
```

**Note:** You do not need to use "doc(..)" in your solution. It will be executed on courses.xml.

# DB7

Q1 (1/1 point)

Return a list of department titles.

Your Query Result:

```
<Title>Computer Science</Title>
<Title>Electrical Engineering</Title>
<Title>Linguistics</Title>
```

```
</xsl:stylesheet>
```

**Note:** You do not need to use "doc(..)" in your solution. It will be executed on courses.xml.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <xsl:stylesheet version="1.0"
3 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4 <xsl:template match="/Course_Catalog/Department">
5   <Title> <xsl:value-of select="Title"/> </Title>
6 </xsl:template>
7 </xsl:stylesheet>
8
```

# Named Templates

- Instead of (or as well as) a *match* attribute, named templates have a *name* attribute.
- Named templates can also have `xsl:param` elements; these let you pass values into the template:

```
<xsl:template name="three-times">
  <xsl:param name="n">
    <xsl:value-of select="$n * 4">
</xsl:template>
<xsl:call-template name="three-times">
  <xsl:with-param name="three-times" select="17" />
</xsl:call-template>
```

- You can call a named template from another template or from inside `<xsl:variable>...</xsl:variable>`.

# The `xsl:for-each` Instruction

- The `<xsl:for-each>` element has a *select* attribute containing an XPath expression;
  - the contents of the for-each instruction element are evaluated for each item in the result of the expression.
- Despite the name, this is *not* a loop:
  - the XSLT processor may well process multiple `Person` elements in parallel, or in any order, as long as the result is in document order.
- `xsl:for-each` sets the XPath context to each node in turn just like `apply-templates`.

# for-each instead of apply-templates

```
<xsl:template match="/">
  <html>
    <head>
      <title>Famous People</title>
    </head>
    <body>
      <h1>Famous People</h1>
      <hr />
      <ul>
        <xsl:for-each select="People/Person">
          <li><xsl:value-of select="Name"/></li>
        </xsl:for-each>
      </ul>
    </body>
  </html>
</xsl:template>
```

# for-each (pull) or apply-templates (push)?

- Processing with **for-each** tends to be fragile:
  - if the input document structure changes, you have to update your **for-each** instructions.
- When you use separate templates to handle each input element, one template is used wherever that element appears
  - (unless you are more specific in its *match* element of course).
- **for-each** SHOULD NOT be used instead of **apply-templates**.

# XSLT and XPath

- Most XSLT instructions that can have a *select* attribute; use it as an XPath expression to generate a list of items.
  - The XSLT context item is available as `current()`;
  - The position in the list of nodes being evaluated is `position()`.
    - Example elements: `for-each`, `apply-templates`, `value-of`.
- Templates have a *match* attribute;
  - this takes an XSLT *match pattern*, a simplified XPath expression.

# XSLT Variables

- You can define “variables” in XSLT that can be used from within XPath expressions.
- A variable is defined from the `xsl:variable` instruction that creates it up to the end of the containing element.
- You can't redefine variables or change their value.

```
<xsl:variable name="n-people" select="count(/People/Person)" />
<xsl:value-of select="$n-people * 2" />
```

- In XSLT 2 you can add a type, which can give better error checking and also better performance:

```
<xsl:variable name="n" select="count(/e/Person)"
  as="xs:integer"/>
```

# XSLT Result Tree Fragments

- You can include XML content inside variables, too:

```
<xsl:variable name="churchill">
  <xsl:apply-templates select="/People/Person[1]" />
</xsl:variable>
<xsl:value-of select="$churchill//ul"/>
```

- XSLT creates a document node inside the variable when you do this;
  - it's always more efficient to use *select* if you can, and you also have to step over the internal document node in XPath expressions.

# Conditional Logic

- Use `xsl:if` for an optional item:

```
<xsl:if test="position() = 2">
  <xsl:text>I am the second one!</xsl:text>
</xsl:if>
```

- in XSLT 2 and later you can also use *if then else* from XPath:

```
<xsl:value-of select="if (position() eq 2) then 2 else 42" />
```

- use `xsl:choose` if you need an “otherwise” or if there are more than two choices.
- The `xsl:text` instruction shown here makes a text node in the output; you can just put the text there, but then all of the whitespace in your template is considered to be text too!

# Conditional in XSLT

- Branching is possible in XSLT

```
<xsl:if test = “boolean expressions” />
```

# Conditional in XSLT

```
1)  <? xml version = "1.0" encoding = "utf-8" ?>
2)  <xsl:stylesheet xmlns:xsl =
3)      "http://www.w3.org/1999/XSL/Transform">
4)      <xsl:template match = "/">
5)          <TABLE border = "5"><TR><TH>Stars</th></tr>
6)          <xsl:for-each select = "Stars/Star" />
7)              <xsl:if test = "Address/City = 'Hollywood'">
8)                  <TR><TD>
9)                      <xsl:value-of select = "Name" />
10)                 </td></tr>
11)             </xsl:if>
12)         </xsl:for-each>
13)     </table>
14)     </xsl:template>
15) </xsl:stylesheet>
```

Figure 12.28: Finding the names of the stars who live in Hollywood

# Use xsl:choose for multiple branches:

```
<xsl:choose>
    <xsl:when test="position() = 1">
        <xsl:text>One item</xsl:text>
    </xsl:when>
    <xsl:when test="count(/People/Person) = 2">
        <xsl:apply-templates select="(/People/Person)[2]" />
    </xsl:when>
    <xsl:otherwise>
        <xsl:text>No-one here but us chickens!</xsl:text>
    </xsl:otherwise>
</xsl:choose>
```

# <xsl:copy-of select = “path”/>

- sends to output a complete copy of the node specified by “path”

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <xsl:stylesheet version="2.0"
3   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4   <xsl:template match="/Course_Catalog/Department">
5     <xsl:copy-of select ="Chair" />
6   </xsl:template>
7 </xsl:stylesheet>
8 |
```

# <xsl:copy-of select = “path”/>

- S S

```
<Chair>
  <Professor>
    <First_Name>Jennifer</First_Name>
    <Last_Name>Widom</Last_Name>
  </Professor>
</Chair>
<Chair>
  <Professor>
    <First_Name>Mark</First_Name>
    <Middle_Initial>A.</Middle_Initial>
    <Last_Name>Horowitz</Last_Name>
  </Professor>
</Chair>
<Chair>
  <Professor>
    <First_Name>Beth</First_Name>
    <Last_Name>Levin</Last_Name>
  </Professor>
</Chair>
```

ode

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <xsl:stylesheet version="2.0"
3   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4   <xsl:template match="/Course_Catalog/Department">
5     <xsl:copy-of select ="Chair" />
6   </xsl:template>
7 </xsl:stylesheet>
8 |
```

# Summary

- XSLT is a functional, declarative language.
- XSLT supports recursion and conditional branching.
- Most XSLT “stylesheets” use `xsl:apply-templates` extensively.
  - It is the most important XSLT instruction.
- You can use all of XPath inside XSLT
  - (XPath was originally developed for XSLT).

# xsl-if

## Q1 (1/1 point)

Return all courses with enrollment greater than 500. Retain the structure of Course elements from the original data.

Your solution should fill in the following stylesheet:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="...">
        ... template body ...
    </xsl:template>
    ... more templates as needed ...
</xsl:stylesheet>
```

**Note:** You do not need to use "doc(..)" in your solution. It will be executed on courses.xml.

*(XSLT can be quite challenging to get right, and this problem is no exception. Congratulations if you succeed!)*

# xsl-if

## Q1 (1/1 point)

Return all courses with enrollment greater than 500. Retain the structure of Course elements from the original data.

Your solution should fill in the following stylesheet:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="...">
        ... template body ...
    </xsl:template>
    ... more templates as needed ...
</xsl:stylesheet>
```

**Note:** You do not need to use "doc(..)" in your solution. It will be executed on courses.xml.

*(XSLT can be quite challenging to get right, and this problem is no exception. Congratulations if you succeed!)*

<xsl:if test = “enrollment > 500”>

... body of if ...

</xsl:if>

# Loop Through Courses

Q1 (1/1 point)

Return all courses with enrollment greater than 500. Retain the structure of Course elements from the original data.

Your solution should fill in the following stylesheet:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="...">
        ... template body ...
    </xsl:template>
    ... more templates as needed ...
</xsl:stylesheet>
```

**Note:** You do not need to use "doc(..)" in your solution. It will be executed on courses.xml.

*(XSLT can be quite challenging to get right, and this problem is no exception. Congratulations if you succeed!)*

<xsl:for-each select = “course”>

... body of loop ...

</xsl:for-each>

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <xsl:stylesheet version="2.0"
3 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4 <xsl:template match="/Course_Catalog/Department">
5     <xsl:for-each select = "Course">
6         <xsl:if test = "@Enrollment > 500">
7             <xsl:copy-of select ="." />
8         </xsl:if>
9     </xsl:for-each>
10    </xsl:template>
11 </xsl:stylesheet>
12 |
```

Your Query Result:

```
<Course Enrollment='1070' Number='CS106A'>
    <Title>Programming Methodology</Title>
    <Description>Introduction to the engineering of computer applications emphasizing modern
software engineering principles.</Description>
    <Instructors>
        <Lecturer>
            <First_Name>Jerry</First_Name>
            <Middle_Initial>R.</Middle_Initial>
            <Last_Name>Cain</Last_Name>
        </Lecturer>
        <Professor>
            <First_Name>Eric</First_Name>
            <Last_Name>Roberts</Last_Name>
        </Professor>
        <Professor>
            <First_Name>Mehran</First_Name>
            <Last_Name>Sahami</Last_Name>
        </Professor>
    </Instructors>
</Course>
<Course Enrollment='620' Number='CS106B'>
    <Title>Programming Abstractions</Title>
```

# Relation Algebra

- Operands:
  - Relations
  - Variables representing relations
- Operators:
  - Set operations
  - Slicing Relations
  - Gluing Relations
  - Renaming Relations
- Closure is Needed

# Example Relation Schemas

- R1( K, A, B, C)
- R2( K, D, E)
- R3( A, A1, A2, A3)
- R4( B, B1, B2)
- R5( C, C1, C2, C3, C4, C5)
- w/ K a key value for R1 and R2.
- w/ A a key value for R3.
- w/ B a key value for R4.
- w/ C a key value for R5.

# Current Instances for Relation Examples

R1

K	A	B	C
4	2	0	6
5	2	0	5
1	1	3	8
2	1	3	7
3	2	3	3

R2

K	D	E
4	1	6
5	1	5
1	1	8
2	1	7
3	1	3

R5

B	B1	B2
0	0	0
3	9	27

R4

C	C1	C2	C3	C4	C5
4	2	0	6	1	6
5	2	0	5	1	5
1	1	3	8	1	8
2	1	3	7	1	7
3	2	3	3	1	3

# Set Operators

## Union/Intersection/Difference

- $X \cap Y$
- $X \cup Y$
- $Y - X$ 
  - Schemas must be identical

# Operators

## Union

- $X \cup Y$

K	D	E
1	1	8
2	1	7

X

K	D	E
4	1	6
5	1	5

Y

K	D	E
4	1	6
5	1	5
1	1	8
2	1	7

$X \cup Y$

# Operators

## Intersection

- $X \cap Y$

K	D	E
4	1	6
5	1	5
1	1	8
2	1	7

X

K	D	E
1	1	8
2	1	7
3	1	3

Y

K	D	E
1	1	8
2	1	7

X  $\cap$  Y

# Operators

## Difference

- $X - Y$ :
- Set of elements in X but NOT IN Y
- Element of Y ALSO IN X are removed

K	D	E
4	1	6
5	1	5
1	1	8
2	1	7

X

K	D	E
1	1	8
2	1	7
3	1	3

Y

K	D	E
4	1	6
5	1	5

X - Y

# Combining Operators

- Since each operation returns a Relation (closure) it can feed other operations.
- Can be viewed as an Expression Tree

# Operators

## Intersection as Difference

- $X \cap Y: X - (X - Y)$

K	D	E
4	1	6
5	1	5
1	1	8
2	1	7

X

K	D	E
4	1	6
5	1	5

K	D	E
1	1	8
2	1	7

K	D	E
1	1	8
2	1	7
3	1	3

Y

X - Y

X - (X - Y)

X ∩ Y

# Operators Selection

- $Y := \sigma_C(X)$ 
  - Select a set of rows of a relation
  - Based on conditional expression C
  - Operands in C are either attributes of relation X or constants.
  - Y includes only tuples that make C true.

# Operators Selection

- $Y := \sigma_{K<3}(X)$

X

K	A	B	C
4	2	0	6
5	2	0	5
1	1	3	8
2	1	3	7
3	2	3	3

$\sigma_{K<3}(X)$

K	A	B	C
1	1	3	8
2	1	3	7

# Operators

## Projection

- $Y := \Pi_L(X)$ 
  - Select a set of attributes/columns of relation

# Operators Projection

- $Y := \Pi_{C,C2}(X)$

C	C1	C2	C3	C4	C5
4	2	0	6	1	6
5	2	0	5	1	5
1	1	3	8	1	8
2	1	3	7	1	7
3	2	3	3	1	3

C	C2
4	0
5	0
1	3
2	3
3	3

X

$\Pi_{C,C2}(X)$

# Product

- $Z := X \times Y$ 
  - Each row of  $X$  attached to Each Possible row of  $Y$

# Product

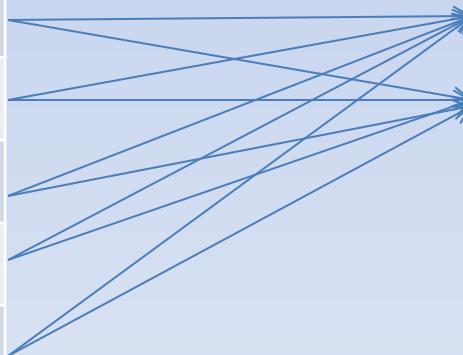
- $Z := R1 \times R4$

R1

K	A	B	C	
4	2	0	6	
5	2	0	5	
1	1	3	8	
2	1	3	7	
3	2	3	3	

R4

B	B1	B2
0	0	0
3	9	27



# Product

R1

K	A	B	C
4	2	0	6
5	2	0	5
1	1	3	8
2	1	3	7
3	2	3	3

R1 X R4

B	B1	B2
0	0	0
3	9	27

R4

K	A	B	C	B	B1	B2
4	2	0	6	0	0	0
4	2	0	6	3	9	27
5	2	0	5	0	0	0
5	2	0	5	3	9	27
1	1	3	8	0	0	0
1	1	3	8	3	9	27
2	1	3	7	0	0	0
2	1	3	7	3	9	27
3	2	3	3	0	0	0
3	2	3	3	3	9	27

# Product

$$Y3 := Y1 \times Y2$$

Y1(

A,	B
1	2
3	4

Y3(

A,	Y1.B,	Y2.B,	C
1	2	5	6
1	2	7	8
1	2	9	10
3	4	5	6
3	4	7	8
3	4	9	10

Y2(

B,	C
5	6
7	8
9	10

# Natural Join

- Usually want to join tuples that in some way *match*.
- Natural Join requires matching attributes have matching values.
- $R_3 := R_1 \bowtie R_2$ .
  - Take Product:  $R_1 \times R_2$
  - Take Result:  $\sigma_C$
  - C is same named attributes are equal
  - Remove redundant attributes
- Dangling Tuples are tuples from one relation that have no match in the other tuple.

# Product – Changed Instances

$$Y3 := Y1 \times Y2$$

$Y1($

A,	B
1	2
3	4

$Y2($

B,	C
2	11
5	9
7	2

$Y3($

A,	Y1.B,	Y2.B,	C
1	2	2	11
1	2	5	9
1	2	7	2
3	4	2	11
3	4	5	9
3	4	7	2

# Natural Join

$$Y3 := Y1 \bowtie Y2$$

$Y1($

A,	B
1	2
3	4

$Y3($

A,	B,	C )
1	2	11

$Y2($

B,	C
2	11
5	9
7	2

# Natural Join: Product & Selection & Projection

## $Y3 := Y1 \times Y2$

$Y1($

A,	B
1	2
3	4

$Y2($

B,	C
2	11
5	9
7	2

$Y3($

A,	$Y1.B,$	$Y2.B,$	C
1	2	2	11
1	2	5	9
1	2	7	2
3	4	2	11
3	4	5	9
3	4	7	2

STILL NEED:

1. Take Result where matching attributes are equal:  $Y4 = \sigma_{Y1.b=Y2.b}(Y3)$
2. Remove redundant attributes:  $\Pi_{A, Y1.B, C}(Y4)$

# Natural Join:

## Dangling Tuples

$Y3 := Y1 \times Y2$

$Y1($

A,	B)
1	2
3	4

$Y2($

B,	C)
2	11
5	9
7	2

$Y3($

A,	$Y1.B,$	$Y2.B,$	C
1	2	2	11
1	2	5	9
1	2	7	2
3	4	2	11
3	4	5	9
3	4	7	2

STILL NEED:

1. Take Result where matching attributes are equal:  $Y4 = \sigma_{Y1.b=Y2.b}(Y3)$
2. Remove redundant attributes:  $\Pi_{A, Y1.B, C}(Y4)$

# Theta-Join

- $R3 := R1 \bowtie_C R2$ 
  - Take Product:  $R1 \times R2$
  - Take Result:  $\sigma_C$
  - C is boolean condition

# Theta Join:

## Product & Selection

$$Y3 := \sigma_{A < C} (Y1 \times Y2)$$

$Y1($

A,	B
1	2
3	4

$Y2($

B,	C
2	11
5	9
7	2

$Y3($

A,	Y1.B,	Y2.B,	C
1	2	2	11
1	2	5	9
1	2	7	2
3	4	2	11
3	4	5	9
3	4	7	2

# Theta Join:

## Product & Selection

$$Y3 := \sigma_{A+B < C} (Y1 \times Y2)$$

Y1(

A,	B
1	2
3	4

Y2(

B,	C
2	11
5	9
7	2

Y3(

A,	Y1.B,	Y2.B,	C
1	2	2	11
1	2	5	9
1	2	7	2
3	4	2	11
3	4	5	9
3	4	7	2

# Renaming

- $R1 := \rho_{R1(A1, \dots, An)}(R2)$ 
  - makes R1 be a relation with attributes A1,...,An and the same tuples as R2.

# Renaming

$Y3 := \rho_{R1(A,B1,B2,C)}( Y1 \times Y2 )$

$Y1($

A,	B
1	2
3	4

$Y2($

B,	C
2	11
5	9
7	2

$R1($

A,	B1,	B2,	C
1	2	2	11
1	2	5	9
1	2	7	2
3	4	2	11
3	4	5	9
3	4	7	2

# Renaming & Set Operations

$$Y3 := \rho_{X,Y}(Y1) \cup \rho_{X,Y}(Y2) )$$

Y1(

A,	B )
1	2
3	4

Y2(

B,	C )
2	11
5	9
7	2

Y3(

X,	Y )
1	2
3	4
2	11
5	9
7	2

# Precedence

- Precedence of relational operators:
  1.  $[\sigma, \pi, \rho]$  (highest).
  2.  $[x, \bowtie]$ .
  3.  $\cap$ .
  4.  $[\cup, -]$

# Relational Algebra for Constraints

- Relation  $R \neq \emptyset$  OR  $R = \emptyset$
- Key Constraints
- Foreign Key Constraints
- Domain Value Constraints

# Example 1: Key Constraint

- $PC1 = PC2 = PC$

$Y := \sigma_{PC1.model=PC2.model \text{ AND } PC.maker \neq PC2.maker}(PC1 \times PC2)$

- Key Constraint:  $Y = \emptyset$

## Example 2

# Referential Integrity Constraint

$Y1 := \pi_{model}(\text{Product})$

$Y2 := \pi_{model}(\text{PC})$

- Referential Integrity Constraint:  $Y2 \subseteq Y1$

# DB4

Courseware Course Info Discussion Wiki Progress Readings Software Guide Extra Problems

Getting Started

Relational Algebra

Select, Project, Join

Set Operators, Renaming,  
Notation

Relational Algebra Quiz  
Quiz

Relational Algebra  
Exercises

Exercise

Course Completion



In this assignment you are to write relational algebra queries over a small database, executed using our RA Workbench. Behind the scenes, the RA workbench translates relational algebra expressions into SQL queries over the database stored in SQLite. Since relational algebra symbols aren't readily available on most keyboards, RA uses a special syntax described in our [RA Relational Algebra Syntax](#) guide.

We've created a small sample database to use for this assignment. It contains four relations:

```
Person(name, age, gender)      // name is a key
Frequents(name, pizzeria)      // [name,pizzeria] is a key
Eats(name, pizza)              // [name,pizza] is a key
Serves(pizzeria, pizza, price) // [pizzeria,pizza] is a key
```

# Chapter 5

## Chapter 5

### Algebraic and Logical Query Languages

206 CHAPTER 5. ALGEBRAIC AND LOGICAL QUERY LANGUAGES

We now switch our attention to bags. We start in databases. We start in languages, one algebraic and one logical.

A	B
1	2
3	4
1	2
1	2

Figure 5.1: A bag

#### 5.1.1 Why Bags?

# Bag Semantics

- Bags (or Multisets):
  - Generalization of the notion of a set.
  - Members are allowed to appear more than once.
- Commercial DB's implement relations as Bags.
- Some relational operations are much more efficient when done on bags.

# Projection w/ Bags

R1

K	A	B	C
4	2	0	6
5	2	0	5
1	1	3	8
2	1	3	7
3	2	3	3

- $\Pi_{A,B}(R1)$  -- efficiency

A	B
2	0
2	0
1	3
1	3
2	3

# Bag Semantics

- Bags let us calculate some types of values more naturally.
  - Find the average B value
  - Projection onto B with sets yields (0,3)
    - Average =  $3/2 = 1.5$
  - Projection onto B with bags yields (0, 0, 3, 3, 3)
    - Average =  $9/5 = 1.8$

A	B
2	0
2	0
1	3
1	3
2	3

# Bag Semantics

## Selection

- Selection – Applied to each tuple independently.

# Bag Semantics

## Set Operations

- Union – Tuples from each relation added to bag.
- Intersection – Tuples in output are the min of the number of times they appear in each relation.
- Difference – Tuples are removed one for one with a minimum value of 0.

# Bag Semantics

## Joins

- Product – Tuples treated independently
- Joins – With multiple tuples having same values, we get multiple join possibilities.

# Bag Semantics

## Example

- create table t1 (x char(1), y int);
- create table t2 (x char(1), z int);
- insert into t1 values ('A',2), ('B',3), ('C',4), ('B',7);
- insert into t2 values ('B', 0), ('C',1), ('B', 4);

x	y
A	2
B	3
C	4
B	7

t1

x	z
B	0
C	1
B	4

t2

# Bag Semantics

## Joins

- select \* from t1 natural join t2;

x	y	z
B	3	0
B	3	4
C	4	1
B	7	0
B	7	4

x	y
A	2
B	3
C	4
B	7

t1

x	z
B	0
C	1
B	4

t2

# Bag Semantics – Slightly Modified Joins

- select \* from t1 natural join t2;

x	y	z
B	3	0
B	3	4
C	4	1
B	3	0
B	3	4

x	y
A	2
B	3
C	4
B	3

t1

x	z
B	0
C	1
B	4

t2

# Relational Algebra – Advanced (Extended)

$\delta$  = eliminate duplicates from bags.

$\tau$  = sort tuples.

$\gamma$  = grouping and aggregation.

Outerjoin :

avoids “dangling tuples” = tuples that do not join with anything.

# Duplicate Elimination

- $R1 = \delta(R2)$ 
  - Out relation  $R1$  with a single copy of each tuple in  $R2$ .

# Duplicate Elimination

- $R1 = \delta(R2)$

$R2$

x	y	z
B	3	0
B	3	4
C	4	1
B	3	0
B	3	4

$R1 = \delta(R2)$

x	y	z
B	3	0
C	4	1
B	3	4

# Sort Tuples

- $\mathsf{T} = \text{sort tuples}$
- $R1 := \mathsf{T}_L(R2)$ .
  - $L$  is a list of attributes from  $R2$ .
- $R1$ :
  - List of tuples of  $R2$
  - sorted first on the value of the first attribute on  $L$
  - Sorted second on the second attribute of  $L$
  - ....

# Sort Tuples

- $R1 := \tau_{x,z}(R2)$ .

$R2$

	x	y	z
	B	3	0
	C	4	1
	B	3	4

$\tau_{x,z}(R2)$

	x	y	z
	B	3	0
	B	3	4
	C	4	1

# Aggregation Operators

- Aggregation Operators:
  - are applied to a entire column of data.
  - Return a single value.
  - i.e., SUM, AVG, COUNT, MIN, and MAX.

# Aggregation Operators

- Aggregation Operators:

- $\text{SUM}(K) = 15$
- $\text{AVG}(K) = 3$
- $\text{COUNT}(K) = 5$
- $\text{MIN}(K) = 1$
- $\text{MAX}(K) = 5$

R1

K	A	B	C
4	2	0	6
5	2	0	5
1	1	3	8
2	1	3	7
3	2	3	3

# $\gamma$ - Grouping

- $\gamma_L$ : (lowercase gamma)
  - Grouping Operator
- $L$  is list of:
  - Individual Attributes (used for grouping)
  - Grouping Operators
    - (i.e., COUNT(), SUM(), ...)

# $\gamma_L(R)$ – Grouping

- Group tuples in R:
  - Form one group for every set of values of attributes from L in R.
  - For each aggregation operator in AGG() in L, apply AGG() to each group formed.
- Outputs:
  - One tuple for each set of values of attributes from L in R.
  - A single value for each AGG() applied to that group.

# Grouping: $\gamma_{B, count(*)}(R1)$

R1

K	A	B	C
4	2	0	6
5	2	0	5
1	1	3	8
2	1	3	7
3	2	3	3

- $\gamma_{B, count(*)}(R1)$
- Group tuples in R1:
  - Form one group for every set of values of attribute ‘B’ in R1 (0, 3).
  - Apply Count(\*) to each group formed.

# Grouping: $\gamma_{B, count(*)}(R1)$

R1

K	A	B	C
4	2	0	6
5	2	0	5
1	1	3	8
2	1	3	7
3	2	3	3

- $\gamma_B(R1)$
- Group tuples in R1:
  - Form one group for every set of values of attribute ‘B’ in R1 (0, 3).
  - Apply Count(\*) to each group formed.

# Grouping: $\gamma_{B, count(*)}(R1)$

R1

B	Count(*)
0	2
3	3

- $\gamma_{B, count(*)}(R1)$
- Group tuples in R1:
  - Form one group for every set of values of attribute ‘B’ in R1 (0, 3).
  - Apply Count(\*) to each group formed.

# Outerjoins

- $R1 \bowtie R2$
- Dangling Tuple:
  - A tuple of  $R1$  with no corresponding tuple from  $R2$  is said to be dangling.
  - A tuple of  $R2$  with no corresponding tuple from  $R1$  is also said to be dangling
  - Outerjoin preserves these tuples by padding them with null.

# Outerjoins

- movie\_list (mid, myear, mname);
- movie\_ratings(pid, mid, rating);
- movie\_list  $\bowtie$  movie\_ratings

mid	myear	mname
979	1979	Movie 1
1079	1979	Movie 2
393	1993	Movie 3
1293	1993	Movie 4
300	2004	Movie 5

pid	mid	Rating
200	979	2
200	393	2
304	300	4

- movie\_list

- movie\_ratings

# Outerjoins

- $\text{movie\_list} \bowtie \text{movie\_ratings}$

mid	myear	mname
979	1979	Movie 1
1079	1979	Movie 2
393	1993	Movie 3
1293	1993	Movie 4
300	2004	Movie 5

pid	mid	Rating
200	979	2
200	393	2
304	300	4

- $\text{movie\_ratings}$

- $\text{movie\_list}$

mid	myear	mname	pid	mid	Rating
979	1979	Movie 1	200	979	2
393	1993	Movie 3	200	393	2
300	2004	Movie 5	304	300	4

# Outerjoins

- movie\_list OUTERJOIN movie\_ratings

mid	myear	mname
979	1979	Movie 1
1079	1979	Movie 2
393	1993	Movie 3
1293	1993	Movie 4
300	2004	Movie 5

pid	mid	Rating
200	979	2
200	393	2
304	300	4

- movie\_ratings

- movie\_list

mid	myear	mname	pid	mid	Rating
979	1979	Movie 1	200	979	2
1079	1979	Movie 2	null	null	null
393	1993	Movie 3	200	393	2
1293	1993	Movie 4	null	null	null
300	2004	Movie 5	304	300	4

# Relational Data Model: Part 2

## Language: SQL

- Two Aspects to language
- Data Definition (ddl):
  - Declaring database schema: tables, constraints, indexes, views.
  - like declaring data/variables in programming language
- Data Manipulation (dml):
  - asking questions (querying) and modifying data.
  - Like executable code within programming language.

# Relational Data Model

## SQL (DDL aspects)

- Create table NAME(att1 type, att2 type ...)
  - NAME: name of the newly created table
  - att1, att2, att3, att4, ...: Attributes for table.
  - Type: data type for each of the attributes
    - See next slide!

# Relational Data Model

## SQL (DDL aspects)

- Create table NAME(att1 type, att2 type ...)
  - NAME: name of the newly created table
  - att1, att2, att3, att4, ...: Attributes for table.
  - Type: data type for each of the attributes
    - See next slide!

# Relational Data Model: SQL

## Creating Tables

```
CREATE TABLE Person(
```

```
    pid int,
```

```
    lName varchar(20),
```

```
    fName varchar(20),
```

```
    gender char(1),
```

```
    birth date);
```

# Tables w/ Keys

- Method 1:
  - ‘PRIMARY KEY’ included after attribute declaration
  - Good only when key is a single attributed
- Method 2:
  - ‘PRIMARY KEY (att1, att2, ... )’ included after all attribute declarations
- UNIQUE can be used instead of PRIMARY KEY:
  - UNIQUE keys can have NULL value.
  - PRIMARY KEY keys cannot be NULL.

# Relational Data Model: SQL

## Creating Tables w/ Keys

### Method 1

```
CREATE TABLE Person(  
    pid int primary key,  
    lName varchar(20),  
    fName varchar(20),  
    gender char(1),  
    birth date);
```

# Relational Data Model: SQL

## Creating Tables w/ Keys

### Method 2

```
CREATE TABLE Person(
```

```
    pid int,
```

```
    lName varchar(20),
```

```
    fName varchar(20),
```

```
    gender char(1),
```

```
    birth date,
```

```
primary key (pid);
```

# StudentDB Example

Transcript(sid, semester, year,  
CourseID, CourseDesc, units, grade)

sid	semester	year	courseID	courseDesc	units	grade
500	Fall	1980	English 1	Composition	3	B
500	Fall	1980	Chem 1A	Gen Qual Anal	5	C
500	Fall	1980	Math 20	Intro Comp Prog	2	A
500	Fall	1980	Math 75	Math Analysis I	4	A
500	Fall	1980	Hist 11	Amer Hst to 1865	3	A
500	Spring	1981	QM 64	Compu Lang - COBOL	3	A
500	Spring	1981	Phil 1	Into to Phil	4	B
500	Spring	1981	Chem 8	Elem Org Chem	3	C
500	Spring	1981	Math 76	Math Analysis II	4	B
500	Spring	1981	Math 114	Discrete Struct	3	B
500	Fall	1981	Art H 20	Modern World	3	B
500	Fall	1981	Fin 34	Personal Investing	3	A
500	Fall	1981	Math 77	Math Anal III	0	F
500	Fall	1981	Math 107	Intro Prob + Stat	3	A
500	Winter	1981	Econ 1a	Prin of Econ	3	A

# Delete a Table

- To delete a table use DROP:

`DROP TABLE tablename;`

- To just delete the data (see chapter 6):

`DELETE FROM tablename;`

# Delete a Table: Example

## StudentDB.sql

- Create Table:

```
CREATE TABLE Person(  
    pid int primary key,  
    lName varchar(20), fName varchar(20),  
    gender char(1), birth date);
```

- Drop table:

```
DROP TABLE Person
```

# Delete a Table: Example

## StudentDB.sql

```
CREATE TABLE Transcript(  
    sid int, semester varchar(20), year int,  
    CourseID varchar(20), CourseDesc varchar(20),  
    units int, grade char(2))
```

- To delete a table use DROP:  
`DROP transcript;`
- To just delete the data (see chapter 6):  
`DELETE FROM transcript;`

# Modifying Tables: ALTER

- The command ALTER is used to modify tables:
  - `ALTER tablename ADD att1 type1`
    - Adds the attribute *att1* with type *type1* to table *tablename*.
  - `ALTER tablename DROP attribute`

# Chapter 6: SQL

- Discussed Data Definition Language (DDL) within SQL.
- Now looking at Data Manipulation Language (DML) within SQL.
- Implementation of Relational Algebra
- SQL Allows DBMS to optimize actual implementation.

# But First: Getting Data

## A Brief Intro

- SQL Modify Commands are not like Queries , in that:
  - they do not return a result.
  - they do modify the contents of database.
- SQL Modify Commands:
  - Insert
  - Update
  - Delete

# Database Mods:

## Basic Insert

- `INSERT INTO table VALUE (...);`
- `INSERT INTO table VALUES (...), (...), (...), ...`
- LATER:
  - `INSERT INTO table (SUBQUERY)`

# Insert Example

- Inserting a single value:

Relation Schema: product (maker, model, ctype);

```
INSERT INTO product VALUE  
('Z', 5005, 'laptop' );
```

# Insert Example

- Inserting multiple values:

Relation Schema: product (maker, model, ctype);

```
#insert the data from 3rd Edition Fig 2.20 pg #53
```

```
insert into product values
```

```
('A',      1001,    'pc'),  
('A',      1002,    'pc'),  
('A',      1003,    'pc'),  
('A',      2004 ,   'laptop'),  
('A'     ,2005 ,   'laptop'),  
('A'     ,2006 ,   'laptop'),  
('B',      1004, 'pc'),  
('B',      1005, 'pc'),  
('B',      1006, 'pc'),  
('B',      2007 , 'laptop'),  
('C',      1007, 'pc');
```

# Database Mods: Delete

- `DELETE FROM table;`
- `DELETE FROM table WHERE <cond>;`

# Database Mods: Update

- UPDATE table SET

att1=value,

att2=value

WHERE <cond>;

# Database Mods: Update

- R1(K int, B float, C float, primary key (K));  
select \* from r1;

K	B	C
2	1	7
3	2	0
5	1	7
7	2	2
9	1	8

# Database Mods: Update

- R1(K int, B float, C float, primary key (K));

UPDATE r1 SET b = b\*10, c = c/10.0 WHERE k < 5;

K	B	C
2	10	0.7
3	20	0
5	1	7
7	2	2
9	1	8

# Back to Chapter 6: SQL

- Discussed Data Definition Language (DDL) within SQL.
- Now looking at Data Manipulation Language (DML) within SQL.
- Implementation of Relational Algebra
- SQL Allows DBMS to optimize actual implementation.
- One SQL Question at End;

# Example Tables

- R1( K, A, B, C)
- R2( K, D, E)
- R3( A, A1, A2, A3)
- R4( B, B1, B2)
- R5( C, C1, C2, C3, C4, C5)
- w/ K a key value for R1 and R2.
- w/ A a key value for R3.
- w/ B a key value for R4.
- w/ C a key value for R5.

# Example Tables:

## Data

R1

K	A	B	C
4	2	0	6
5	2	0	5
1	1	3	8
2	1	3	7
3	2	3	3

R2

K	D	E
4	1	6
5	1	5
1	1	8
2	1	7
3	1	3

R5

B	B1	B2
0	0	0
3	9	27

R4

C	C1	C2	C3	C4	C5
4	2	0	6	1	6
5	2	0	5	1	5
1	1	3	8	1	8
2	1	3	7	1	7
3	2	3	3	1	3

# Simplest SQL Operation: Projection from Relational Algebra

- $Y := \Pi_L(X)$ 
  - Select a set of attributes/columns of relation

# Projection w/ SQL

## SELECT-FROM

**SELECT** attribute-names

**FROM** table-name ;

# Operators

## Projection w/ SQL

- $Y := \Pi_{C,C2}(X)$
- SQL:  
SELECT C, C2  
FROM X;

C	C1	C2	C3	C4	C5
4	2	0	6	1	6
5	2	0	5	1	5
1	1	3	8	1	8
2	1	3	7	1	7
3	2	3	3	1	3

X

C	C2
4	0
5	0
1	3
2	3
3	3

SELECT C, C2  
FROM X;

# Operators

## Projection w/ SQL

- Listing all tuples
- SQL:
  - SELECT \*
  - FROM X

C	C1	C2	C3	C4	C5
4	2	0	6	1	6
5	2	0	5	1	5
1	1	3	8	1	8
2	1	3	7	1	7
3	2	3	3	1	3

SELECT \* FROM X

# Operators Selection

- $Y := \sigma_C(X)$ 
  - Select a set of rows of a relation
  - Based on conditional expression C
  - Operands in C are either attributes of relation X or constants.
  - Y includes only tuples that make C true.

# Projection w/ SQL

## SELECT-FROM-WHERE

**SELECT** attribute-names

**FROM** table-name

**WHERE** condition;

# Selection w/ SQL: WHERE Clause

- $Y := \sigma_{K<3}(X)$
- SQL:
  - SELECT \*
  - FROM X
  - WHERE K<3;

X

K	A	B	C	
4	2	0	6	
5	2	0	5	
1	1	3	8	
2	1	3	7	
3	2	3	3	

$\sigma_{K<3}(X)$

K	A	B	C	
1	1	3	8	
2	1	3	7	

SELECT \* FROM X WHERE K<3;

# Selection & Projection w/ SQL: Select-From-Where Statements

- $Y := \sigma_{K<3}(X)$
- SQL:
  - SELECT K, A
  - FROM X
  - WHERE K<3;

X

K	A	B	C	
4	2	0		6
5	2	0		5
1	1	3		8
2	1	3		7
3	2	3		3

$\pi_{K,A}(\sigma_{K<3}(X))$

K	A
1	1
2	1

SELECT K, A FROM X WHERE K<3;

# Selection & Projection w/ SQL: Select-From-Where Statements

- $Y := \sigma_{K<3}(X)$
- SQL:
  - SELECT K, A ( $\pi_{K,A}$ )
  - FROM X
  - WHERE K<3 ( $\sigma_{K<3}$ );

X

K	A	B	C	
4	2	0		6
5	2	0		5
1	1	3		8
2	1	3		7
3	2	3		3

$\pi_{K,A}(\sigma_{K<3}(X))$

K	A
1	1
2	1

SELECT K, A  
FROM X  
WHERE K<3;

# SQL: Select-From-Where Statements

- Always remember, and never forget:
  - `SELECT * FROM table ;`
    - `SELECT` attributes
    - `FROM` table
    - `WHERE attribute = value ;`

`SELECT` desired attributes

`FROM` one or more tables

`WHERE` condition about tuples of the tables

# Select-From-Where Statements

R5

C	C1	C2	C3	C4	C5
4	2	0	6	1	6
5	2	0	5	1	5
1	1	3	8	1	8
2	1	3	7	1	7
3	2	3	3	1	3

- Select C, C5 from R5;

C	C5
4	6
5	5
1	8
2	7
3	3

# Select-From-Where Statements

R5

C	C1	C2	C3	C4	C5
4	2	0	6	1	6
5	2	0	5	1	5
1	1	3	8	1	8
2	1	3	7	1	7
3	2	3	3	1	3

- Select C, C5 from R5;
- $\pi_{C,C5}(R5)$

C	C5
4	6
5	5
1	8
2	7
3	3

# Select-From-Where Statements

R5

C	C1	C2	C3	C4	C5
4	2	0	6	1	6
5	2	0	5	1	5
1	1	3	8	1	8
2	1	3	7	1	7
3	2	3	3	1	3

- SELECT C, C5
- FROM R5
- WHERE C > 3;

C	C5
4	6
5	5

# Select-From-Where Statements

R5

C	C1	C2	C3	C4	C5
4	2	0	6	1	6
5	2	0	5	1	5
1	1	3	8	1	8
2	1	3	7	1	7
3	2	3	3	1	3

- SELECT C, C5
- FROM R5
- WHERE C > 3;
- $\sigma_{C>3}(\pi_{C,C5}(R5))$

C	C2
4	6
5	5

# Example:

## Exercise – 2.4.1

- Product(maker, model, type)
  - PC(model, speed, ram, hd, price)
  - Laptop(model, speed, ram, hd, screen, price)
  - Printer(model, color, type, price)
- a) What PC models have a speed of at least 3.00?

# a) What PC models have a speed of at least 3.00?

- $R1 := \sigma_{\text{speed} \geq 3.00}(\text{PC})$
- $R2 := \pi_{\text{model}}(R1)$

model
1005
1006
1013

# a) What PC models have a speed of at least 3.00?

- Step 1: Sele

```
1 #testing
2 • use computers;
3
4
5 • select * from pc;
```

Result Set Filter:  Export: Wrap Cell Content:

model	speed	ram	hd	price
1001	2.66	1024	250	2114
1002	2.1	512	250	995
1003	1.42	512	80	478
1004	2.8	1024	250	649
1005	3.2	512	250	630
1006	3.2	1024	250	1049
1007	2.2	1024	250	510
1008	2.2	2048	250	770
1009	2	1024	250	650
1010	2.8	2048	300	770
1011	1.86	2048	160	959
1012	2.8	1024	160	649
1013	3.06	512	80	529

# a) What PC models have a speed of at least 3.00?

- Step 2: Relational Algebra

- $\sigma_{\text{speed} \geq 3.00}(\text{PC})$

The screenshot shows a database interface with a command-line area and a results grid.

Command-line area (top half):

```
1 #testing
2 • use computers;
3
4
5 • select * from pc;
6 • select * from pc where speed >= 3.00;
7
```

Results grid (bottom half):

model	speed	ram	hd	price
1005	3.2	512	250	630
1006	3.2	1024	250	1049
1013	3.06	512	80	529

# a) What PC models have a speed of at least 3.00?

- Step 3: Relational Algebra

- $R1 := \sigma_{\text{speed} \geq 3.00}(\text{PC})$
- $R2 := \pi_{\text{model}}(R1)$

The screenshot shows a database management system interface with the following details:

- Toolbar:** Includes icons for file operations, search, and various database functions.
- Tab Bar:** Shows tabs for "testing\*", "assign.2.music", "movies", "ratings", "SQL File 5", "assign.2.movies", "q.1", "movies", and "StudentDB".
- Code Editor:** Displays the following SQL code:

```
1 #testing
2 • use computers;
3
4
5 • select * from pc;
6 • select * from pc where speed >= 3.00;
7 • select model from pc where speed >= 3.00;
```
- Result Set Filter:** A text input field for filtering results.
- Result Grid:** Shows a table with one column labeled "model". The data rows are:

model
1005
1006
1013
- Page Number:** In the bottom right corner, it says "321".

# Example:

## Exercise – 6.1.3

- Product(maker, model, type)
  - PC(model, speed, ram, hd, price)
  - Laptop(model, speed, ram, hd, screen, price)
  - Printer(model, color, type, price)
- a) Find the model number, speed, and hard-disk size for all PC's whose price is under \$1000?

a) Find the model number, speed, and hard-disk size for all PC's whose price is under \$1000?

- Step 1: Selection:
  - SELECT \*
  - FROM PC
  - WHERE price < 1000 ;

a) Find the model number, speed, and hard-disk size for all PC's whose price

- Step

- SE

- FR

- W

- Step

- SE

- FR

- W

The screenshot shows the MySQL Workbench interface with a query editor and results pane.

**Query Editor:**

```
5 • SELECT model,
6      speed,
7      hd
8  FROM PC
9 WHERE price < 1000 ;
10
11
12 select * from pc;
```

**Results Table:**

model	speed	hd
1002	2.1	250
1003	1.42	80
1004	2.8	250
1005	3.2	250
1007	2.2	250
1008	2.2	250
1009	2	250
1010	2.8	300
1011	1.86	160
1012	2.8	160
1013	3.06	80

**Output Log:**

Action	Time	Message
use computers	13:44:48	0 row(s) affected
SELECT model, speed, hd FROM PC WHERE price < 1000 LIMIT 0, 1000	13:44:54	11 row(s) returned

# Renaming Attributes

- Attributes can be renamed:
  - Within Select Clause
  - Using Keyword: AS

```
SELECT old-name AS new-name  
FROM table-name
```

# Renaming Attributes

R5

C	C1	C2	C3	C4	C5
4	2	0	6	1	6
5	2	0	5	1	5
1	1	3	8	1	8
2	1	3	7	1	7
3	2	3	3	1	3

SELECT C AS x, C2 AS y

FROM R5;

- $\rho_{x,y}(\pi_{c,c2}(R5))$

X	Y
4	0
5	0
1	3
2	3
3	3

# Example:

## Exercise – 6.1.3

- Product(maker, model, type)
  - PC(model, speed, ram, hd, price)
  - Laptop(model, speed, ram, hd, screen, price)
  - Printer(model, color, type, price)
- b) Find the model number, speed, and hard-disk size for all PC's whose price is under \$1000, but rename the *speed* column *gigahertz* and the *hd* column *gigabytes*?

b) Find the model number, speed, and hard-disk size for all PC's whose price is under \$1000, but rename the *speed* column *gigahertz* and the *hd* column *gigabytes*?

- Step

The screenshot shows the MySQL Workbench interface. The top menu bar has tabs for 'testing\*' (selected), 'Triggers.2', 'StudentDB', 'iris.2D', 'assign.2.computers', 'searching', 'assign.2.computers', 'assign.2.music', and 'movies'. Below the menu is a toolbar with various icons. The main area contains a SQL editor window with the following code:

```
5 • SELECT model,
6      speed AS gigahertz,
7      hd     AS gigabytes
8 FROM PC
9 WHERE price < 1000 ;
10
11
12 select * from pc;
```

Below the SQL editor is a result grid titled 'Result Set Filter' with three columns: 'model', 'gigahertz', and 'gigabytes'. The data is as follows:

model	gigahertz	gigabytes
1002	2.1	250
1003	1.42	80
1004	2.8	250
1005	3.2	250
1007	2.2	250
1008	2.2	250
1009	2	250
1010	2.8	300
1011	1.86	160
1012	2.8	160
1013	3.06	80

At the bottom, there is an 'Output' tab showing the log of actions:

Action Output	Time	Action	Message
1	13:44:48	use computers	0 row(s) affected
2	13:44:54	SELECT model, speed, hd FROM PC WHERE price < 1000 LIMIT 0, 1000	11 row(s) returned
3	13:55:27	SELECT model, speed AS gigahertz, hd AS gigabytes FROM PC WHERE pri...	11 row(s) returned

# Constants & Expression w/ S

## Select Clause

- Select 'title', c + c1 AS total;

# Example:

## Exercise – 6.1.3

- Product(maker, model, type)
- PC(model, speed, ram, hd, price)
- Laptop(model, speed, ram, hd, screen, price)
- Printer(model, color, type, price)

Find the model number and hard-disk size in megabytes for all PC's whose price is under \$1000, and rename the *hd* column *megabytes*?

Find the model number and hard-disk size in megabytes for all PC's whose price is under \$1000, and rename the *hd* column *megabytes*?

The screenshot shows a MySQL Workbench interface. The query editor window contains the following SQL code:

```
3
4 • SELECT model,
5      hd*1000 AS megabytes
6  FROM PC
7 WHERE price < 1000 ;
8
```

The results grid displays the following data:

model	megabytes
1002	250000
1003	80000
1004	250000
1005	250000
1007	250000
1008	250000
1009	250000
1010	300000
1011	160000
1012	160000
1013	80000

# Matching Strings w/ LIKE's

- Comparing a string to a pattern:  
`<Attribute> LIKE <pattern>`  
`<Attribute> NOT LIKE <pattern>`
- *Pattern* is a quoted string with
  - % = “any string.”
  - \_ = “any character.”
- `SELECT * FROM person WHERE name LIKE “D%”`
- `SELECT * FROM person WHERE name LIKE “D_”`

# Matching Strings w/ Like

The screenshot shows a MySQL Workbench interface with the following details:

- Query Editor:** Contains the following SQL code:

```
91
92 • select * from person;
93 • select * from person where fname like 'Seedorf%';
94
```
- Results Grid:** Displays a table of 18 rows with columns: pid, lname, fname, gender, birth. The 'fname' column values all start with 'Seedorf'.

pid	lname	fname	gender	birth
116	Pete	Seedorf116	M	1985-06-13
120	Mary	Seedorf120	F	1947-04-30
122	Marius	Seedorf122	M	1960-07-21
124	Habib	Seedorf124	M	1941-12-23
132	Raj	Seedorf132	M	1972-02-22
133	Ginger	Seedorf133	F	1955-07-09
134	Shirley	Seedorf134	F	1947-05-21
139	Mary	Seedorf139	F	1943-01-18
143	Raj	Seedorf143	M	1964-10-18
162	Atha	Seedorf162	F	1957-09-21
167	Jane	Seedorf167	F	1981-09-06
169	Pete	Seedorf169	M	1969-02-16
170	Floyd	Seedorf170	M	1975-10-09
176	John	Seedorf176	M	1983-05-19
177	Mary	Seedorf177	F	1953-09-08
185	Jack	Seedorf185	M	1979-10-21
- Action History:** Shows the following log:

Time	Action	Message
16 09:53:49	alter table person change firstname lname varchar(20)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0
17 09:53:52	select * from person where fname like 'Seedorf%' LIMIT 0..1000	135 row(s) returned

# Matching strings w/ Like

- Find all movies that have ‘action’ in title.

SELECT \*

```
95 • select * from movies where mname like '%action%';
```

Result Set Filter:  Edit: Export/Import: Wrap Cell Content:

	mid	myear	mname
▶	391	1993	Last Action Hero
	930	1996	Chain Reaction
*	NULL	NULL	NULL

# Matching strings w/ Like

- Find all movies that have ‘action’ in title.

SELECT \*

The screenshot shows a MySQL query interface with the following details:

- Query Bar:** The query entered is: `select * from movies where mname like '% action %';`
- Result Set Filter:** An empty text input field for filtering results.
- Toolbar:** Includes buttons for Edit, Export/Import, and Wrap Cell Content.
- Table Results:** A table with columns: mid, myear, mname.

	mid	myear	mname
▶	391	1993	Last Action Hero
*	NULL	NULL	NULL

MySQL Workbench

cs126

File Edit View Query Database Server Tools Scripting Help

SOL SQL Data Export Data Import/Restore

Navigator

MANAGEMENT

- Server Status
- Client Connections
- Users and Privileges
- Status and System Variables
- Data Export
- Data Import/Restore

INSTANCE

- Startup / Shutdown
- Server Logs
- Options File

SCHEMAS

- Filter objects
- computers
- iris2d
- movies**
- music
- sakila
- studentdb
- test
- world

Information

No object selected

Object Info Session

Query 1 test\* assign.2.computers

```
37 #delete from example
38 use computers;
39 • delete from product where maker = 'A';
40
41 • use movies;
42 • select * from person where fname like 'Seedorf_22';
```

Result Set Filter: | Edit: | Export/Import: | Wrap Cell Content: |

pid	IName	fName	gender	birth
122	Marius	Seedorf122	M	1960-07-21
222	Lorie	Seedorf222	F	1966-04-11
622	Pete	Seedorf622	M	1969-08-12
*				

person 7 x

Output

Action Output

Time	Action	Message	Duration / Fetch
9 10:17:43	select * from person where fname like 'Seedorf22_' LIMIT...	4 row(s) returned	0.000 sec / 0.000 sec
10 10:20:58	select * from person where fname like 'Seedorf_22_' LIMI...	0 row(s) returned	0.000 sec / 0.000 sec
11 10:21:09	select * from person where fname like 'Seedorf_22' LIMIT...	3 row(s) returned	0.000 sec / 0.000 sec

# DB5

Stanford University



Databases: DB5 SQL

Courseware

Course Info

Discussion

Wiki

Progress

Readings

Software Guide

▶ Getting Started

▼ SQL

Introduction to SQL

Basic SELECT Statement

Table Variables and Set Operators

Subqueries in WHERE Clause

Subqueries in FROM and SELECT



VIDEO

SQL: Intro

Data Definition Language (DDL)

Create table ...  
drop table ...

Data Manipulation Language (DML)

# DB5

SQL

## Introduction to SQL

### Basic SELECT Statement

### Table Variables and Set Operators

### Subqueries in WHERE Clause

### Subqueries in FROM and SELECT

### The JOIN Family of Operators

### Aggregation

### NULL Values

### Data Modification Statements

### SQL Movie-Rating Query Exercises

Exercise

### SQL Movie-Rating Query Exercises Extras

## VIDEO

A screenshot of a video player interface. The main window shows a database query in a SQL editor. The query is:

```
1 select sname, GPA, decision
2 from Student, Apply
3 where Student.sID = Apply.sID
4 and sizeHS < 1000 and major = 'CS' and cname = 'Stanford';
```

Below the editor is an 'Output' tab with columns for Type, Info, Description, and Time. A large red play button is centered over the video area. In the bottom right corner of the video frame, there is a small thumbnail of a person's face and the YouTube logo.

This is the first of seven videos where we're going to learn the SQL language.

The videos are largely going to be live demos of SQL queries and updates running on an actual database.

The first video is going to focus on the basics of the `SELECT` statement.

# DB5



You've started a new movie-rating website, and you've been collecting data on reviewers' ratings of various movies. There's not much data yet, but you can still try out some interesting queries. Here's the schema:

Movie ( mID, title, year, director )

English: There is a movie with ID number *mID*, a *title*, a release *year*, and a *director*.

Reviewer ( rID, name )

English: The reviewer with ID number *rID* has a certain *name*.

Rating ( rID, mID, stars, ratingDate )

English: The reviewer *rID* gave the movie *mID* a number of *stars* rating (1-5) on a certain *ratingDate*.

Your queries will run over a small data set conforming to the schema. [View the database](#). (You can also [download the schema and data](#).)

# DB5

## Q1 (1 point possible)

Find the titles of all movies directed by Steven Spielberg.

**Note:** Your queries are executed using SQLite, so you must conform to the SQL constructs supported by SQLite.

```
1 select * from movie;
```



Incorrect

### Q1 (1 point possible)

Find the titles of all movies directed by Steven Spielberg.

**Note:** Your queries are executed using SQLite, so you must conform to the SQL constructs supported by SQLite.

```
1 select * from movie;
```

**Incorrect**

Your Query Result:

101	Gone with the Wind	1939	Victor Fleming
102	Star Wars	1977	George Lucas
103	The Sound of Music	1965	Robert Wise
104	E.T.	1982	Steven Spielberg
105	Titanic	1997	James Cameron
106	Snow White	1937	<NULL>
107	Avatar	2009	James Cameron
108	Raiders of the Lost Ark	1981	Steven Spielberg

**X** Incorrect

**Expected Query Result:**

E.T.

Raiders of the Lost Ark

# Matching Strings w/ LIKE's

- Comparing a string to a pattern:  
`<Attribute> LIKE <pattern>`  
`<Attribute> NOT LIKE <pattern>`
- *Pattern* is a quoted string with
  - % = “any string.”
  - \_ = “any character.”
- `SELECT * FROM person WHERE name LIKE “D%”`
- `SELECT * FROM person WHERE name LIKE “D_”`

# Matching Strings w/ Like

The screenshot shows a MySQL Workbench interface with the following details:

- Query Editor:** Contains the following SQL code:

```
91
92 • select * from person;
93 • select * from person where fname like 'Seedorf%';
94
```
- Results Grid:** Displays a table of 18 rows with columns: pid, lname, fname, gender, birth. The 'fname' column values all start with 'Seedorf'.

pid	lname	fname	gender	birth
116	Pete	Seedorf116	M	1985-06-13
120	Mary	Seedorf120	F	1947-04-30
122	Marius	Seedorf122	M	1960-07-21
124	Habib	Seedorf124	M	1941-12-23
132	Raj	Seedorf132	M	1972-02-22
133	Ginger	Seedorf133	F	1955-07-09
134	Shirley	Seedorf134	F	1947-05-21
139	Mary	Seedorf139	F	1943-01-18
143	Raj	Seedorf143	M	1964-10-18
162	Atha	Seedorf162	F	1957-09-21
167	Jane	Seedorf167	F	1981-09-06
169	Pete	Seedorf169	M	1969-02-16
170	Floyd	Seedorf170	M	1975-10-09
176	John	Seedorf176	M	1983-05-19
177	Mary	Seedorf177	F	1953-09-08
185	Jack	Seedorf185	M	1979-10-21
- Action History:** Shows the following log:

Time	Action	Message
16 09:53:49	alter table person change firstname lname varchar(20)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0
17 09:53:52	select * from person where fname like 'Seedorf%' LIMIT 0..1000	135 row(s) returned

# Matching strings w/ Like

- Find all movies that have ‘action’ in title.

SELECT \*

The screenshot shows a MySQL query interface with the following details:

- Query ID: 95
- Query: `select * from movies where mname like '%action%';`
- Result Set Filter: (empty)
- Table Headers: mid, myear, mname
- Results:

	mid	myear	mname
▶	391	1993	Last Action Hero
	930	1996	Chain Reaction
*	NUL	NUL	NUL

# Matching strings w/ Like

- Find all movies that have ‘action’ in title.

SELECT \*

The screenshot shows a MySQL query interface with the following details:

- Query Bar:** The query entered is `select * from movies where mname like '% action %';`.
- Result Set Filter:** An empty text input field for filtering results.
- Toolbar:** Includes buttons for Edit, Export/Import, and Wrap Cell Content.
- Table Results:** A table with columns `mid`, `myear`, and `mname`. The data row is:

	mid	myear	mname
▶	391	1993	Last Action Hero
*	NULL	NULL	NULL

MySQL Workbench

cs126

File Edit View Query Database Server Tools Scripting Help

SOL SQL Data Export Data Import/Restore

Navigator

MANAGEMENT

- Server Status
- Client Connections
- Users and Privileges
- Status and System Variables
- Data Export
- Data Import/Restore

INSTANCE

- Startup / Shutdown
- Server Logs
- Options File

SCHEMAS

- Filter objects
- computers
- iris2d
- movies**
- music
- sakila
- studentdb
- test
- world

Information

No object selected

Object Info Session

Query 1 test\* assign.2.computers

```
37 #delete from example
38 use computers;
39 • delete from product where maker = 'A';
40
41 • use movies;
42 • select * from person where fname like 'Seedorf_22';
```

Result Set Filter: | Edit: | Export/Import: | Wrap Cell Content: |

pid	IName	fName	gender	birth
122	Marius	Seedorf122	M	1960-07-21
222	Lorie	Seedorf222	F	1966-04-11
622	Pete	Seedorf622	M	1969-08-12
*				

person 7 x

Output

Action Output

Time	Action	Message	Duration / Fetch
9 10:17:43	select * from person where fname like 'Seedorf22_' LIMIT...	4 row(s) returned	0.000 sec / 0.000 sec
10 10:20:58	select * from person where fname like 'Seedorf_22_' LIMI...	0 row(s) returned	0.000 sec / 0.000 sec
11 10:21:09	select * from person where fname like 'Seedorf_22' LIMIT...	3 row(s) returned	0.000 sec / 0.000 sec

# Working w/ Dates

- DATE is a special string type in SQL:
  - ‘2014-02-04’
- YEAR( DATE ):
  - Yields the year from the date value
  - SELECT YEAR(DATE '2014-02-04');
    - Yields ‘2014’

# Assign.2.movies.sql

- CREATE TABLE Person(  
    pid int primary key,  
    lName varchar(20),  
    fName varchar(20),  
    gender char(1),  
    birth date);

# Assign.2.movies.sql

- Query: Find everyone born in 1970:

The screenshot shows a MySQL Workbench interface with a query editor and a results grid.

Query Editor (top half):

```
testing* x Triggers.2 StudentDB iris.2D assign.2.computers searching assign.2.computers assign.2.music movies ratings SQL File 11 assign.  
29  
30 • SELECT *  
31 FROM person  
32 WHERE YEAR(birth) = 1970;  
33  
34
```

Results Grid (bottom half):

pid	IName	fName	gender	birth
117	Sandy	Toyama117	F	1970-11-12
137	Jackie	Jones137	F	1970-04-06
238	Marius	Halperin238	M	1970-09-05
279	John	Nichols279	M	1970-11-29
302	Marius	Nichols302	M	1970-02-20
424	Nelson	Singh424	M	1970-03-15
456	Shirley	Nichols456	F	1970-07-17
565	Nelson	Halperin565	M	1970-03-22
635	Jack	Singh635	M	1970-06-17
669	Ginger	Seedorf669	F	1970-10-09
691	Burly	Jones691	M	1970-09-11
736	Atha	Seedorf736	F	1970-01-05
775	Habib	Seedorf775	M	1970-03-08

# Ordering Output

- If we want the output sorted:

The screenshot shows a MySQL Workbench interface with a query editor and a results grid.

**Query Editor:**

```
testing* x Triggers 2 StudentDB iris.2D assign.2.computers searching assign.2.computers assign.2.music movies ratings SQL File 11 assign
29
30 • SELECT *
31   FROM person
32 WHERE YEAR(birth) = 1970
33 order by birth;
34
```

**Result Set Grid:**

	pid	IName	fName	gender	birth
▶	736	Atha	Seedorf736	F	1970-01-05
	302	Marius	Nichols302	M	1970-02-20
	775	Habib	Seedorf775	M	1970-03-08
	424	Nelson	Singh424	M	1970-03-15
	565	Nelson	Halperin565	M	1970-03-22
	137	Jackie	Jones137	F	1970-04-06
	911	Mary	Seedorf911	F	1970-04-19
	1033	Raymond	Singh1033	M	1970-04-26
	1064	Raymond	Halperin1064	M	1970-06-15
	635	Jack	Singh635	M	1970-06-17
	456	Shirley	Nichols456	F	1970-07-17
	782	Lorie	Toyama782	F	1970-08-09
	238	Marius	Halperin238	M	1970-09-05

# Joining Tables

- SQL offers the ability to connect tables in several different ways.
- PRODUCT is the simplest, by including each relation in the FROM clause list (separated by commas)

# Select-From-Where Statements

- **SELECT \* FROM R1, R4;**

B	B1	B2
0	0	0
3	9	27

R4

R1

K	A	B	C
4	2	0	6
5	2	0	5
1	1	3	8
2	1	3	7
3	2	3	3

R1 X R4

K	A	B	C	B	B1	B2
4	2	0	6	0	0	0
4	2	0	6	3	9	27
5	2	0	5	0	0	0
5	2	0	5	3	9	27
1	1	3	8	0	0	0
1	1	3	8	3	9	27
2	1	3	7	0	0	0
2	1	3	7	3	9	27
3	2	3	3	0	0	0
3	2	3	3	3	9	27

# Select-From-Where Statements

- **SELECT K, B2  
FROM R1, R4;**

R1

K	A	B	C
4	2	0	6
5	2	0	5
1	1	3	8
2	1	3	7
3	2	3	3

$\pi_{K,B2}(R1 \times R4)$

B	B1	B2
0	0	0
3	9	27

R4

K	B2
4	0
4	27
5	0
5	27
1	0
1	27
2	0
2	27
3	0
3	27

# Product

Y1 X Y2

IN SQL:

```
SELECT *\nFROM Y1, Y2;
```

Y1(

A,	B
1	2
3	4

Y2(

B,	C
5	6
7	8
9	10

(	A,	Y1.B,	Y2.B,	C	)
1	2	5	6		
1	2	7	8		
1	2	9	10		
3	4	5	6		
3	4	7	8		
3	4	9	10		

# Select-From-Where Statements

- **SELECT K, R1.B, R4.B, B2  
FROM R1, R4  
WHERE K < 3 ;**

R1

K	A	B	C
4	2	0	6
5	2	0	5
1	1	3	8
2	1	3	7
3	2	3	3

R4

B	B1	B2
0	0	0
3	9	27

$$\sigma_{K<3} (\pi_{K,R1.B,R4.B,B2} (R1 \times R4))$$

K	R1.B	R4.B	B2
1	3	0	0
1	3	3	27
2	3	0	0
2	3	3	27

# Select-From-Where Statements

- **SELECT K, B2  
FROM R1, R4  
WHERE K < 3 AND  
R1.B = R4.B ;**

$$\sigma_{K<3 \text{ and } R1.B=R4.B}(\pi_{K,R1.B,R4.B,B2}(R1 \times R4))$$

R1

K	A	B	C
4	2	0	6
5	2	0	5
1	1	3	8
2	1	3	7
3	2	3	3

R4

B	B1	B2
0	0	0
3	9	27

K	R1.B	R2.B	B2
4	2	0	0
5	2	0	0
1	3	3	27
2	3	0	0
3	3	3	27

# Select-From-Where Statements

- **SELECT K AS x, B2 AS y,  
FROM R1, R4  
WHERE K < 3 AND R1.B=R4.B;**

R1

K	A	B	C
4	2	0	6
5	2	0	5
1	1	3	8
2	1	3	7
3	2	3	3

R4

B	B1	B2
0	0	0
3	9	27

$$\rho_{x,y} ( \sigma_{K<3} ( \pi_{K,B2} (R1 \times R4) ) )$$

X	Y
1	27
2	27

# Select-From-Where Statements: Tuple Variables

- **SELECT \***  
**FROM R1A, R1 B**  
**WHERE B.K < 3 AND A.K=B.C;**

R1

K	A	B	C
4	2	0	6
5	2	0	5
1	1	3	8
2	1	3	7
3	2	3	3

$$\sigma_{B.K<3} ( \rho_{A(K,A,B,C)}(R1) \times \rho_{B(K,A,B,C)}(R1) )$$

# Example:

## Exercise – 2.4.1

- Product(maker, model, type)
  - PC(model, speed, ram, hd, price)
  - Laptop(model, speed, ram, hd, screen, price)
  - Printer(model, color, type, price)
- b) Which manufacturers make laptops with a hard disk of at least 100gb

b) Which manufacturers make laptops with a hard disk of at least 100gb

```
97 •   SELECT Product.maker  
98     FROM Product, Laptop  
99 WHERE Product.Model = Laptop.model AND  
100    hd >= 100;  
101
```

maker
E
A
B
F
G

Result Set Filter:  Export: Wrap Cell Content:

maker
A
B
E
F
G

```
FROM Product, Laptop  
WHERE Product.Model = Laptop.model AND  
hd >= 100;
```

# **Example:**

## **Exercise – 2.4.1**

- Product(maker, model, type)
  - PC(model, speed, ram, hd, price)
  - Laptop(model, speed, ram, hd, screen, price)
  - Printer(model, color, type, price)
- d) Find the model numbers of all color laser printers

# Example

- Product(maker, model, type)
- PC(model, speed, ram, hd, price)
- Laptop(model, speed, ram, hd, screen, price)
- Printer(model, color, type, price)
- **Write the Relational Algebra to:**

Ex: 2.4.1.d) Find the model numbers of all color laser printers

$R1 := \sigma_{\text{color} = \text{true AND type} = \text{laser}} (\text{Printer})$

$R2 := \pi_{\text{model}} (R1)$

IN SQL:

```
SELECT model FROM printer  
WHERE color and ctype='laser';
```

model
3003
3007

List all printer models and type,  
along with whether color or B&W

SELECT model, ctype,

CASE color

WHEN true THEN 'color'

WHEN false THEN 'B&W'

ELSE 'error'

END as Color

FROM printer;

model	ctype	color
3001	ink-jet	color
3002	laser	B&W
3003	laser	color
3004	ink-jet	color
3005	laser	B&W
3006	ink-jet	color
3007	laser	color

# List all printer makers, models, & prices

```
SELECT product.maker, product.model, printer.price  
FROM product, printer
```

```
101  
102 • SELECT product.maker, product.model, printer.price  
103 FROM product, printer  
104 WHERE product.model = printer.model;  
105
```

Result Set Filter:  Export: Wrap Cell Content:

maker	model	price
D	3004	120
D	3005	120
E	3001	99
E	3002	239
E	3003	899
H	3006	100
H	3007	200

# List all printer makers, models, & prices w/ TUPLE VARIABLES

SELECT m.maker, m.model, pr.price

```
101
102 • SELECT product.maker, product.model, printer.price
103   FROM product, printer
104 WHERE product.model = printer.model;
105
106 • SELECT m.maker, m.model, pr.price
107   FROM product m, printer pr
108 WHERE m.model = pr.model;
109
```

Result Set Filter:  Export: Wrap Cell Content:

	maker	model	price
▶	D	3004	120
	D	3005	120
	E	3001	99
	E	3002	239
	E	3003	899
	H	3006	100
	H	3007	200

# SQL - Advanced

- Outer Joins
- Aggregations
- Eliminating Duplicates
- Grouping
- Having Clause
- Database Modifications

# Joins - Revisited

- Join On (Theta Join)

SELECT \*

FROM relation1 JOIN relation2 ON <condition1>

WHERE <condition2>;

# Joins - Revisited

- Join Using

SELECT \*

FROM relation1 JOIN relation2 USING (att1, ...)

WHERE <condition>;

# Outerjoins w/ SQL

- `SELECT * from R1 NATURAL OUTER JOIN R2;`
- Dangling Tuple:
  - A tuple of R1 with no corresponding tuple from R2 is said to be dangling.
  - A tuple of R2 with no corresponding tuple from R1 is also said to be dangling
  - Outer Join preserves these tuples by padding them with null.

# Outerjoins

- Full outer join not available in MySQL - emulated
- `SELECT * from R1 NATURAL LEFT JOIN R2 UNION  
SELECT * FROM R1 NATURAL RIGHT JOIN R2 ;`
- Dangling Tuple:
  - A tuple of R1 with no corresponding tuple from R2 is said to be dangling. Left Join preserves these.
  - A tuple of R2 with no corresponding tuple from R1 is also said to be dangling. Right Join preserves these.
  - Tuples preserved by padding them with null.
  - NOTE: Second Query needs extra where clause.
    - (Reader Exercise)

# Outerjoins

- movie\_list (mid, myear, mname);
- movie\_ratings(pid, mid, rating);
- Select \* from

movie\_list natural join movie\_ratings

mid	myear	mname
979	1979	Movie 1
1079	1979	Movie 2
393	1993	Movie 3
1293	1993	Movie 4
300	2004	Movie 5

pid	mid	Rating
200	979	2
200	393	2
304	300	4

- movie\_list

- movie\_ratings

# Outerjoins

- Select \* from movie\_list natural join movie\_ratings;

mid	myear	mname
979	1979	Movie 1
1079	1979	Movie 2
393	1993	Movie 3
1293	1993	Movie 4
300	2004	Movie 5

pid	mid	Rating
200	979	2
200	393	2
304	300	4

- movie\_ratings

- movie\_list

mid	myear	mname	pid	mid	Rating
979	1979	Movie 1	200	979	2
393	1993	Movie 3	200	393	2
300	2004	Movie 5	304	300	4

# Outerjoins

- Select \* from movie\_list LEFT JOIN movie\_ratings;

mid	myear	mname
979	1979	Movie 1
1079	1979	Movie 2
393	1993	Movie 3
1293	1993	Movie 4
300	2004	Movie 5

pid	mid	Rating
200	979	2
200	393	2
304	300	4

- movie\_ratings

- movie\_list

mid	myear	mname	pid	mid	Rating
979	1979	Movie 1	200	979	2
1079	1979	Movie 2	null	null	null
393	1993	Movie 3	200	393	2
1293	1993	Movie 4	null	null	null
300	2004	Movie 5	304	300	4

# Relational Algebra – Extended

## Now In SQL

$\delta$  = eliminate duplicates from bags.

$\tau$  = sort tuples.

$\gamma$  = grouping and aggregation.

*Outerjoin* : avoids “dangling tuples” = tuples that do not join with anything.

# Bag Semantics

## Example

- create table t1 (x char(1), y int);
- create table t2 (x char(1), z int);
- insert into t1 values ('A',2), ('B',3), ('C',4), ('B',3);
- insert into t2 values ('B', 0), ('C',1), ('B', 4);

x	y
A	2
B	3
C	4
B	3

t1

x	z
B	0
C	1
B	4

t2

# Bag Semantics

## Joins

- select \* from t1 natural join t2;

R2

x	y	z
B	3	0
B	3	4
C	4	1
B	3	0
B	3	4

x	y
A	2
B	3
C	4
B	7

t1

x	z
B	0
C	1
B	4

t2

# Duplicate Elimination: Distinct

- SELECT \* FROM R2;

R2

x	y	z
B	3	0
B	3	4
C	4	1
B	3	0
B	3	4

SELECT DISTINCT \* FROM R2;

$\delta_{(R2)}$

x	y	z
B	3	0
C	4	1
B	3	4

# Sort Tuples

- $\mathsf{T} = \text{sort tuples}$
- $R1 := \mathsf{T}_L(R2)$ .
  - $L$  is a list of attributes from  $R2$ .
- $R1$ :
  - List of tuples of  $R2$
  - sorted first on the value of the first attribute on  $L$
  - Sorted second on the second attribute of  $L$
  - ....

# Sort Tuples: Order By

SELECT DISTINCT \* FROM R2;

x	y	z
B	3	0
C	4	1
B	3	4

SELECT DISTINCT \* FROM R2 ORDER BY x,z;

$\tau_{x,z}(R2)$

x	y	z
B	3	0
B	3	4
C	4	1

# Aggregation Operators

- Aggregation Operators:
  - are applied to a entire column of data.
  - Return a single value.
  - i.e., SUM, AVG, COUNT, MIN, and MAX.

# Aggregation Operators

- Aggregation Operators:

➤ SUM(K)	=	15
➤ AVG(K)	=	3
➤ COUNT(K)	=	5
➤ MIN(K)	=	1
➤ MAX(K)	=	5

R1			
K	A	B	C
4	2	0	6
5	2	0	5
1	1	3	8
2	1	3	7
3	2	3	3

SELECT sum(k), avg(k), count(k),  
min(k), max(k)  
FROM R1;

sum(k)	avg(k)	count(k)	min(k)	max(k)
15	3	5	1	5

# $\gamma$ - Grouping

- $\gamma_L$ : (lowercase gamma)
  - Grouping Operator
- $L$  is list of:
  - Individual Attributes (used for grouping)
  - Grouping Operators
    - (i.e., COUNT(), SUM(), ...)

# $\gamma_L(R)$ – Grouping

- Group tuples in R:
  - Form one group for every set of values of attributes from L in R.
  - For each aggregation operator in AGG() in L, apply AGG() to each group formed.
- Outputs:
  - One tuple for each set of values of attributes from L in R.
  - A single value for each AGG() applied to that group.

# Grouping: $\gamma_{B, \text{count}(K)}(R1)$

R1

K	A	B	C
4	2	0	6
5	2	0	5
1	1	3	8
2	1	3	7
3	2	3	3

- $\gamma_B(R1)$
- Group tuples in R1:
  - Form one group for every set of values of attribute ‘B’ in R1 (0, 3).
  - Apply Count(K) to each group formed.

# Grouping: $\gamma_{B, count(K)}(R1)$

R1

K	A	B	C
4	2	0	6
5	2	0	5
1	1	3	8
2	1	3	7
3	2	3	3

- $\gamma_{B, count(*)}(R1)$
- Group tuples in R1:
  - Form one group for every set of values of attribute ‘B’ in R1 (0, 3).
  - Apply Count(K) to each group formed.

# Grouping: $\gamma_{B, count(K)}(R1)$

R1

K	A	B	C
4	2	0	6
5	2	0	5
1	1	3	8
2	1	3	7
3	2	3	3

- $\gamma_{B, count(*)}(R1)$ :
  - SELECT B, Count(K) FROM R1 GROUP BY B;
    - Group tuples in R1:
    - Form one group for every set of values of attribute ‘B’ in R1 (0, 3).
    - Apply Count(K) to each group formed.

# Grouping: $\gamma_{B, count(*)}(R1)$

R1

B	Count(K)
0	2
3	3

- $\gamma_{B, count(*)}(R1)$ 
  - SELECT B, Count(K) FROM R1  
GROUP BY B;
    - Group tuples in R1:
    - Form one group for every set of values of attribute ‘B’ in R1 (0, 3).
    - Apply Count(K) to each group formed.

# Having Clause

- HAVING <condition>
  - Can follow a GROUP BY clause
  - Condition is applied to each group
  - Groups not satisfying condition are not included in query.

# Having Clause

- Same Constraints as Select w/ Group By
  - Anything goes w/ a subquery
  - Grouping Attributes w/ condition.
  - Grouping Operators w/ condition.

# Questions 6.4.6

- Product(maker, model, type)
- PC(model, speed, ram, hd, price)
- Laptop(model, speed, ram, hd, screen, price)
- Printer(model, color, type, price)
- a) Find the average speed of PC's .

# Questions 6.4.6

```
698 • use computers;  
699 • select * from product;  
700 • select avg(speed) as avg_speed from pc;  
701 )
```

Result Set Filter:  Export: Wrap Cell Content:

	avg_speed
▶	2.4846153809474063

select avg(speed) as avg\_speed from pc;

# Questions 6.4.6

- Product(maker, model, type)
- PC(model, speed, ram, hd, price)
- Laptop(model, speed, ram, hd, screen, price)
- Printer(model, color, type, price)
- b) Find the average speed of laptops costing over \$1000.

# Questions 6.4.6

```
702 • select avg(speed) as avg_speed from laptop  
703   where price>1000;
```

Result Set Filter:  Export: Wrap Cell Content:

avg_speed
1.9983333547910054

- b) Find the average speed of laptops costing over \$1000.

```
select avg(speed) as avg_speed from laptop  
      where price>1000;
```

# Questions 6.4.6

- Product(maker, model, type)
- PC(model, speed, ram, hd, price)
- Laptop(model, speed, ram, hd, screen, price)
- Printer(model, color, type, price)
- c) Find the average price of PC's made by manufacturer "A ."

# Questions 6.4.6

```
704 • select avg(price) as avg_speed from PC, product  
705      where product.model=pc.model and maker='A';  
706  
707
```

Result Set Filter:  Export: Wrap Cell Content:

avg_speed
-----------

▶ 1195.6667
-------------

c, find the average price of PCs made by manufacturer “A .”

```
select avg(price) as avg_price from PC, product  
where product.model=pc.model and maker='A';
```

# Questions 6.4.6

- Product(maker, model, type)
- PC(model, speed, ram, hd, price)
- Laptop(model, speed, ram, hd, screen, price)
- Printer(model, color, type, price)
- c) Find , for each different speed, the average price of a PC .

- Product
- PC(n)
- Laptop
- Printer
- c) Find the average price of each speed

```
707 •    SELECT SPEED,
708          AVG(price) AS AVG_PRICE
709      FROM PC
710     GROUP BY speed ;
```

Result Set Filter: Export: Wrap Cell

SPEED	AVG_PRICE
1.42	478.0000
1.86	959.0000
2	650.0000
2.1	995.0000
2.2	640.0000
2.66	2114.0000
2.8	689.3333
3.06	529.0000
3.2	839.5000

Result 8 ×

select speed, avg(price) as avg\_price from PC  
group by speed ;

# Questions 6.4.6

- Product(maker, model, type)
- PC(model, speed, ram, hd, price)
- Laptop(model, speed, ram, hd, screen, price)
- Printer(model, color, type, price)
- g) Find the manufacturers that make at least three different models of PC .

- Product(maker)
- PC(model, spec)
- Laptop(model, spec)
- Printer(model, spec)
- g) Find the makers of PC. different models

```

712 •   SELECT R.maker
713     FROM Product R,
714          PC P
715     WHERE R.model = P.model
716     GROUP BY R.maker
717     HAVING COUNT(R.model) >=3 ;
718
719

```

Result Set Filter:  Export:

	maker
▶	A
	B
	D
	E

```

SELECT R.maker
FROM Product R,
      PC P
WHERE R.model = P.model
GROUP BY R.maker
HAVING COUNT(R.model) >=3 ;

```

```
719 •    SELECT  maker
720      FROM  Product
721     WHERE  ctype='pc'
722     GROUP BY maker
723    HAVING COUNT(model) >=3 ;
724
```

Result Set Filter: Export:

maker
A
B
C
D
E

- Product(maker)
- PC(model, size)
- Laptop(model, processor, price)
- Printer(model, type)
- Find the manufacturers that make at least three different models of PC.

```
SELECT  maker
FROM    Product
WHERE   ctype='pc'
GROUP BY maker
HAVING COUNT(model) >=3 ;
```

# In-Class

- h) Find for each manufacturer who sells PC's the maximum price of a PC.

```
725 •    SELECT R.maker,  
726          MAX(P.price) AS Max_Price  
727      FROM Product R,  
728          PC P  
729      WHERE R.model = P.model  
730      GROUP BY R.maker ;  
731  
732
```

• It returns the maximum price for each manufacturer.

	maker	Max_Price
▶	A	2114
▶	B	1049
▶	C	510
▶	D	770
▶	E	959

GROUP BY R.maker ;

# Chapter 20

## Distributed Databases

- Database Models using Parallel Computing
- Map-Reduce Programming Model
- Peer-To-Peer
- 0 Chance of Question