Announcements

• If you are enrolled to the class, but
  – have not received the email from Piazza, please email me
  – If you missed Thursday’s lab, please email me

• HW1 will be released this week

• Project ideas will be posted by next week
Today’s topic

• Physical and Logical Data Independence
• Data Model and XML
• More SQL
  – Aggregates (Group-by)!
  – Creating/modifying relations/data
  – Constraints

Acknowledgement:
The following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke.
Physical and Logical Data Independence
What does a DBMS provide?
Why use a DBMS?

1. **Data Independence**
   - Application programs should not be exposed to the data representation and storage
   - DBMS provides an abstract view of the data

2. **Efficient Data Access**
   - A DBMS utilizes a variety of sophisticated techniques to store and retrieve data (from disk) efficiently
Why use a DBMS?

3. Data Integrity and Security
   - DBMS enforces “integrity constraints” – e.g. check whether total salary is less than the budget
   - DBMS enforces “access controls” – whether salary information can be accesses by a particular user

4. Data Administration
   - Centralized professional data administration by experienced users can manage data access, organize data representation to minimize redundancy, and fine tune the storage
Why use a DBMS?

5. Concurrent Access and Crash Recovery
   – DBMS schedules concurrent accesses to the data such that the users think that the data is being accessed by only one user at a time
   – DBMS protects data from system failures

6. Reduced Application Development Time
   – Supports many functions that are common to a number of applications accessing data
   – Provides high-level interface
   – Facilitates quick and robust application development
When NOT to use a DBMS?

- DBMS is optimized for certain kind of workloads and manipulations

- There may be applications with tight real-time constraints or a few well-defined critical operations

- Abstract view of the data provided by DBMS may not suffice

- To run complex, statistical/ML analytics on large datasets
Levels of Abstractions in a DBMS

- Physical schema
  - Storage as files, row vs. column store, indexes
  - will discuss these in later lectures
Levels of Abstractions in a DBMS

- **Logical/Conceptual schema**
  - describes the stored data in the physical schema

- **Decided by conceptual schema design**
  - e.g. ER Diagram
    - not covered in this course
  - Normalization
    - will be covered

Students(sid: string, name: string, login: string, age: integer, gpa: real)
Levels of Abstractions in a DBMS

• **External schema**
  - different “views” of the database to different users (later)

• **One physical and logical schema but there can be multiple external schemas**
Data Independence

- Application programs are insulated from changes in the way the data is structured and stored

- A very important property of a DBMS

- Logical and Physical
Logical Data Independence

- Users can be shielded from changes in the logical structure of data
- e.g. Students:
  Students(sid: string, name: string, login: string, age: integer, gpa: real)
- Divide into two relations
  Students_public(sid: string, name: string, login: string)
  Students_private(sid: string, age: integer, gpa: real)
- Still a “view” Students can be obtained using the above new relations
  – by “joining” them with sid
- A user who queries this view Students will get the same answer as before
Physical Data Independence

• The logical/conceptual schema insulates users from changes in physical storage details
  – how the data is stored on disk
  – the file structure
  – the choice of indexes

• The application remains unaltered
  – But the performance may be affected by such changes
Data Model and XML (an overview)
Data Model

• Applications need to model some real world units

• Entities:
  – Students, Departments, Courses, Faculty, Organization, Employee, ...

• Relationships:
  – Course enrollments by students, Product sales by an organization

• A data model is a collection of high-level data description constructs that hide many low-level storage details
Data Model

Can Specify:

1. Structure of the data
   – like arrays or structs in a programming language
   – but at a higher level (conceptual model)

2. Operations on the data
   – unlike a programming language, not any operation can be performed
   – allow limited sets of queries and modifications
   – a strength, not a weakness!

3. Constraints on the data
   – what the data can be
   – e.g. a movie has exactly one title
Important Data Models

• Structured Data
• Semi-structured Data
• Unstructured Data

What are these?
Important Data Models

• **Structured Data**
  – All elements have a fixed format
  – Relational Model (table)

• **Semi-structured Data**
  – Some structure but not fixed
  – Hierarchically nested tagged-elements in tree structure
  – XML

• **Unstructured Data**
  – No structure
  – text, image, audio, video
Semi-structured Data and XML

- **XML**: Extensible Markup Language

- **Will not be covered in detail in class, but many datasets available to download are in this form**
  - You will download the DBLP dataset in XML format and transform into relational form (in HW1!)

- **Data does not have a fixed schema**
  - “Attributes” are part of the data
  - The data is “self-describing”
  - Tree-structured
<article mdate="2011-01-11" key="journals/acta/Saxena96">
  <author>Sanjeev Saxena</author>
  <title>Parallel Integer Sorting and Simulation Amongst CRCW Models.</title>
  <pages>607-619</pages>
  <year>1996</year>
  <volume>33</volume>
  <journal>Acta Inf.</journal>
  <number>7</number>
  <url>db/journals/acta/acta33.html#Saxena96</url>
  <ee>http://dx.doi.org/10.1007/BF03036466</ee>
</article>
Attribute vs. Elements

• Elements can be repeated and nested
• Attributes are unique and atomic
XML vs. Relational Databases

+ Serves as a model suitable for integration of databases containing similar data with different schemas
  - e.g. try to integrate two student databases: $S1(sid, name, gpa)$ and $S2(sid, dept, year)$
  - Many "nulls" if done in relational model, very easy in XML
• NULL = A keyword to denote missing or unknown values

+ Flexible – easy to change the schema and data

- Makes query processing more difficult

Which one is easier?
• XML (semi-structured) to relational (structured) or
• relational (structured) to XML (semi-structured)?
XML to Relational Model

• Problem 1: Repeated attributes

<book>
  <author>Ramakrishnan</author>
  <author>Gehrke</author>
  <title>Database Management Systems</title>
  <publisher>McGraw Hill</publisher>
</book>

What is a good relational schema?
XML to Relational Model

• Problem 1: Repeated attributes

<book>
    <author>Ramakrishnan</author>
    <author>Gehrke</author>
    <title>Database Management Systems</title>
    <publisher>McGraw Hill</publisher>
</book>

What if the paper has a single author?
XML to Relational Model

• Problem 1: Repeated attributes

<book>
  <author>Garcia-Molina</author>
  <author>Ullman</author>
  <author>Widom</author>
  <title>Database Systems – The Complete Book</title>
  <publisher>Prentice Hall</publisher>
</book> Does not work

<table>
<thead>
<tr>
<th>Title</th>
<th>Publisher</th>
<th>Author1</th>
<th>Author2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## XML to Relational Model

<table>
<thead>
<tr>
<th>BookId</th>
<th>Title</th>
<th>Publisher</th>
</tr>
</thead>
<tbody>
<tr>
<td>b1</td>
<td>Database Management Systems</td>
<td>McGraw Hill</td>
</tr>
<tr>
<td>b2</td>
<td>Database Systems – The Complete Book</td>
<td>Prentice Hall</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BookAuthoredBy</th>
</tr>
</thead>
<tbody>
<tr>
<td>BookId</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>b1</td>
</tr>
<tr>
<td>b1</td>
</tr>
<tr>
<td>b2</td>
</tr>
<tr>
<td>b2</td>
</tr>
<tr>
<td>b2</td>
</tr>
</tbody>
</table>
XML to Relational Model

• Problem 2: Missing attributes

```
<book>
  <author>Ramakrishnan</author>
  <author>Gehrke</author>
  <title>Database Management Systems</title>
  <publisher>McGraw Hill</publisher>
  <edition>Third</edition>
</book>
<book>
  <author>Garcia-Molina</author>
  <author>Ullman</author>
  <author>Widom</author>
  <title>Database Systems – The Complete Book</title>
  <publisher>Prentice Hall</publisher>
</book>
```

<table>
<thead>
<tr>
<th>BookId</th>
<th>Title</th>
<th>Publisher</th>
<th>Edition</th>
</tr>
</thead>
<tbody>
<tr>
<td>b1</td>
<td>Database Management Systems</td>
<td>McGraw Hill</td>
<td>Third</td>
</tr>
<tr>
<td>b2</td>
<td>Database Systems – The Complete Book</td>
<td>Prentice Hall</td>
<td>null</td>
</tr>
</tbody>
</table>
Summary: Data Models

• Relational data model is the most standard for database managements
  – and is the main focus of this course

• Semi-structured model/XML is also used in practice – you will use them in hw assignments

• Unstructured data (text/photo/video) is unavoidable, but won’t be covered in this class
Back to SQL!
Expressions and Strings

- Illustrates use of arithmetic expressions and string pattern matching
- **Find triples (of ages of sailors and two fields defined by expressions) for sailors**
  - whose names begin and end with B and contain at least three characters
- **LIKE** is used for string matching. `_' stands for any one character and `%' stands for 0 or more arbitrary characters
  - You will need these often

```sql
SELECT S.age, age1=S.age-5, 2*S.age AS age2 
FROM Sailors S 
WHERE S.sname LIKE 'B_%B'
```
Find sid’s of sailors who’ve reserved a red or a green boat

- **UNION**: Can be used to compute the union of any two *union-compatible* sets of tuples
  - can themselves be the result of SQL queries
- If we replace OR by AND in the first version, what do we get?
- Also available: **EXCEPT** (What do we get if we replace **UNION** by **EXCEPT**?)

```sql
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
AND (B.color='red' OR B.color='green')
UNION
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
AND B.color='red'
UNION
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
AND B.color='green'
```
Find sid’s of sailors who’ve reserved a red and a green boat
Find sid’s of sailors who’ve reserved a red and a green boat

- **INTERSECT**: Can be used to compute the intersection of any two union-compatible sets of tuples.
  
  - Included in the SQL/92 standard, but some systems don’t support it

```
SELECT S.sid
FROM Sailors S, Boats B1, Reserves R1,
     Boats B2, Reserves R2
WHERE S.sid=R1.sid AND R1.bid=B1.bid
     AND S.sid=R2.sid AND R2.bid=B2.bid
     AND (B1.color='red' AND B2.color='green')
```

```
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
     AND B.color='red'
INTERSECT
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
     AND B.color='green'
```
Nested Queries

Find names of sailors who’ve reserved boat #103:

```
SELECT S.sname 
FROM Sailors S 
WHERE S.sid IN (SELECT R.sid 
    FROM Reserves R 
    WHERE R.bid=103)
```

- A very powerful feature of SQL:
  - a WHERE/FROM/HAVING clause can itself contain an SQL query
- To find sailors who’ve not reserved #103, use NOT IN.
- To understand semantics of nested queries, think of a nested loops evaluation
  - For each Sailors tuple, check the qualification by computing the subquery
Nested Queries with Correlation

Find names of sailors who’ve reserved boat #103:

```
SELECT S.sname
FROM Sailors S
WHERE EXISTS (SELECT *
  FROM Reserves R
  WHERE R.bid=103 AND S.sid=R.sid)
```

- **EXISTS** is another set comparison operator, like **IN**
- Illustrates why, in general, subquery must be re-computed for each Sailors tuple
Nested Queries with Correlation

Find names of sailors who’ve reserved boat #103 at most once:

\[
\text{SELECT } S\text{.sname} \\
\text{FROM } \text{Sailors } S \\
\text{WHERE } \text{UNIQUE (SELECT } R\text{.bid} \\
\text{FROM } \text{Reserves } R \\
\text{WHERE } R\text{.bid}=103 \text{ AND } S\text{.sid}=R\text{.sid)}
\]

• If \text{UNIQUE} is used, and * is replaced by \text{R.bid}, finds sailors with at most one reservation for boat #103
  – \text{UNIQUE} checks for duplicate tuples
More on Set-Comparison Operators

• We’ve already seen IN, EXISTS and UNIQUE
• Can also use NOT IN, NOT EXISTS and NOT UNIQUE.
• Also available: op ANY, op ALL, op IN
  — where op : >, <, =, <=, >=
• Find sailors whose rating is greater than that of some sailor called Horatio
  — similarly ALL

```
SELECT *
FROM Sailors S
WHERE S.rating > ANY (SELECT S2.rating
FROM Sailors S2
WHERE S2.sname='Horatio')
```
**Recall: Aggregate Operators**

Check yourself:
What do these queries compute?

- \( \text{SELECT COUNT (*) FROM Sailors S} \)
- \( \text{SELECT AVG (S.age) FROM Sailors S WHERE S.rating=10} \)
- \( \text{SELECT COUNT (DISTINCT S.rating) FROM Sailors S WHERE S.sname='Bob'} \)
- \( \text{SELECT AVG (DISTINCT S.age) FROM Sailors S WHERE S.rating=10} \)

- COUNT (*)
- COUNT ([DISTINCT] A)
- SUM ([DISTINCT] A)
- AVG ([DISTINCT] A)
- MAX (A)
- MIN (A)
Motivation for Grouping

• So far, we’ve applied aggregate operators to all (qualifying) tuples
  – Sometimes, we want to apply them to each of several groups of tuples
• Consider: Find the age of the youngest sailor for each rating level
  – In general, we don’t know how many rating levels exist, and what the rating values for these levels are!
  – Suppose we know that rating values go from 1 to 10; we can write 10 queries that look like this (need to replace i by num):

\[
\text{For } i = 1, 2, \ldots, 10:\quad \text{SELECT MIN (S.age) FROM Sailors S WHERE S.rating = i}
\]
Queries With GROUP BY and HAVING

```
SELECT [DISTINCT] target-list
FROM relation-list
WHERE qualification
GROUP BY grouping-list
HAVING group-qualification
```

- The target-list contains
  - (i) attribute names
  - (ii) terms with aggregate operations (e.g., MIN (S.age))

- The attribute list (i) must be a subset of grouping-list
  - Intuitively, each answer tuple corresponds to a group, and these attributes must have a single value per group
  - Here a group is a set of tuples that have the same value for all attributes in grouping-list
Conceptual Evaluation

• The cross-product of relation-list is computed
• Tuples that fail qualification are discarded
• ‘Unnecessary’ fields are deleted
• The remaining tuples are partitioned into groups by the value of attributes in grouping-list
• The group-qualification is then applied to eliminate some groups
• Expressions in group-qualification must have a single value per group
  – In effect, an attribute in group-qualification that is not an argument of an aggregate op also appears in grouping-list
  – like “…GROUP BY bid, sid HAVING bid = 3”
• One answer tuple is generated per qualifying group

First go over the examples in the following slides
Then come back to this slide and study yourself
Find age of the youngest sailor with age $\geq 18$, for each rating with at least 2 such sailors.

```sql
SELECT S.rating, MIN(S.age) AS minage
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating
HAVING COUNT(*) > 1
```

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>29</td>
<td>brutus</td>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>32</td>
<td>andy</td>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>64</td>
<td>horatio</td>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>71</td>
<td>zorba</td>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>74</td>
<td>horatio</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>85</td>
<td>art</td>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>95</td>
<td>bob</td>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>96</td>
<td>frodo</td>
<td>3</td>
<td>25.5</td>
</tr>
</tbody>
</table>

**Answer relation:**

<table>
<thead>
<tr>
<th>rating</th>
<th>minage</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
</tbody>
</table>
Find age of the youngest sailor with age $\geq 18$, for each rating with at least 2 such sailors.

Step 1: Form the cross product: FROM clause
(some attributes are omitted for simplicity)

<table>
<thead>
<tr>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
</tbody>
</table>

SELECT S.rating, MIN (S.age) AS minage
FROM   Sailors S
WHERE  S.age $\geq$ 18
GROUP BY S.rating
HAVING COUNT (*) > 1
Find age of the youngest sailor with age >= 18, for each rating with at least 2 such sailors.

Step 2: Apply WHERE clause

```
SELECT S.rating, MIN(S.age) AS minage
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating
HAVING COUNT(*) > 1
```
Find age of the youngest sailor with age >= 18, for each rating with at least 2 such sailors.

Step 3: Apply GROUP BY according to the listed attributes

<table>
<thead>
<tr>
<th>rating</th>
<th>age</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>45.0</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>1</td>
<td>33.0</td>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>8</td>
<td>55.5</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>10</td>
<td>35.0</td>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>10</td>
<td>16.0</td>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>9</td>
<td>35.0</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>3</td>
<td>63.5</td>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
<td>3</td>
<td>25.5</td>
</tr>
</tbody>
</table>

### SQL Query

```sql
SELECT S.rating, MIN(S.age) AS minage
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating
HAVING COUNT(*) > 1
```
Find age of the youngest sailor with age $\geq 18$, for each rating with at least 2 such sailors.

**Step 4: Apply HAVING clause**

The *group-qualification* is applied to eliminate some groups.

```sql
SELECT S.rating, MIN(S.age) AS minage
FROM Sailors S
WHERE S.age $\geq$ 18
GROUP BY S.rating
HAVING COUNT(*) > 1
```
Find age of the youngest sailor with age $\geq 18$, for each rating with at least 2 such sailors.

**Step 5: Apply SELECT clause**

Apply the aggregate operator

At the end, one tuple per group

```
SELECT S.rating, MIN(S.age) AS minage
FROM Sailors S
WHERE S.age $\geq$ 18
GROUP BY S.rating
HAVING COUNT(*) > 1
```
Nulls and Views in SQL
Null Values

• Field values in a tuple are sometimes
  – unknown, e.g., a rating has not been assigned, or
  – inapplicable, e.g., no spouse’s name
  – SQL provides a special value null for such situations.
Standard Boolean 2-valued logic

- True = 1, False = 0
- Suppose X = 5
  - (X < 100) AND (X >= 1) is $T \land T = T$
  - (X > 100) OR (X >= 1) is $F \lor T = T$
  - (X > 100) AND (X >= 1) is $F \land T = F$
  - NOT(X = 5) is $\neg T = F$

- Intuitively,
  - $T = 1, F = 0$
  - For $V1, V2 \in \{1, 0\}$
    - $V1 \land V2 = \text{MIN}(V1, V2)$
    - $V1 \lor V2 = \text{MAX}(V1, V2)$
    - $\neg(V1) = 1 - V1$
2-valued logic does not work for nulls

• Suppose rating = null, X = 5
• Is rating > 8 true or false?
• What about AND, OR and NOT connectives?
  – (rating > 8) AND (X = 5)?
• What if we have such a condition in the WHERE clause?
3-Valued Logic For Null

- TRUE (= 1), FALSE (= 0), UNKNOWN (= 0.5)
  - unknown is treated as 0.5

- Now you can apply rules from 2-valued logic!
  - For \( V_1, V_2 \in \{1, 0, 0.5\} \)
  - \( V_1 \land V_2 = \text{MIN}(V_1, V_2) \)
  - \( V_1 \lor V_2 = \text{MAX}(V_1, V_2) \)
  - \( \neg (V_1) = 1 - V_1 \)

- Therefore,
  - NOT UNKNOWN = UNKNOWN
  - UNKNOWN OR TRUE = TRUE
  - UNKNOWN AND TRUE = UNKNOWN
  - UNKNOWN AND FALSE = FALSE
  - UNKNOWN OR FALSE = UNKNOWN