Announcements
• If you are enrolled to the class, but have not received the email from Piazza, please send me an email
• 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.

Why use a DBMS?
What does a DBMS provide?
Why use a DBMS?

Levels of Abstractions in a DBMS

- Physical schema
  - Storage as files, row vs. column store, indexes
  - will discuss these in later lectures

- Logical schema
  - describes the stored data in the physical schema
  - Decided by conceptual schema design
    - e.g. ER Diagram
    - will not be covered in this course

- External schema
  - different “views” of the database to different users (later)
  - One physical and logical schema but there can be multiple external schemas

When NOT to use a DBMS?

Levels of Abstractions in a DBMS

- External Schema
- Logical Schema
- Physical Schema
- Disk

Students(sid: string, name: string, login: string, age: integer, gpa: real)
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

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

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?

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

XML: Example

```
<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)
- 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>Garcia-Molina</author>
  <author>Ullman</author>
  <author>Widom</author>
  <title>Database Systems – The Complete Book</title>
  <publisher>Prentice Hall</publisher>
</book>

Does not work

Title Publisher Author1 Author2

XML to Relational Model

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

SELECT S.age, age1=S.age-5, 2*S.age AS age2
FROM Sailors S
WHERE S.sname LIKE 'B_%B'

• 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

Find sid's of sailors who've reserved a red or a green boat

• Assume a Boats relation
• 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?)

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

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:

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

- **EXISTS** is another set comparison operator, like **IN**
- Illustrates why, in general, subquery must be recomputed for each Sailors tuple

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

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

- If **UNIQUE** is used, and * is replaced by \(R.bid\), finds sailors with at most one reservation for boat #103
  - **UNIQUE** checks for duplicate tuples

Recall: Aggregate Operators

Check yourself:
What do these queries compute?

\[
\begin{align*}
\text{SELECT COUNT(*) FROM Sailors S WHERE S.rating>10} \\
\text{SELECT AVG(S.age) FROM Sailors S WHERE S.rating=10} \\
\text{SELECT COUNT(DISTINCT S.rating) FROM Sailors S WHERE S.sname='Bob'} \\
\text{SELECT AVG(DISTINCT S.age) FROM Sailors S WHERE S.rating=10}
\end{align*}
\]

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):
    \[
    \text{SELECT MIN(S.age) FROM Sailors S WHERE S.rating=}i
    \]
  - For \(i = 1, 2, \ldots, 10\):
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

<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>10</td>
<td>35.0</td>
</tr>
<tr>
<td>25.5</td>
<td>35.0</td>
</tr>
<tr>
<td>35.0</td>
<td>35.0</td>
</tr>
<tr>
<td>25.5</td>
<td>35.0</td>
</tr>
<tr>
<td>33.5</td>
<td>35.0</td>
</tr>
<tr>
<td>25.5</td>
<td>35.0</td>
</tr>
</tbody>
</table>

**Step 2:** Apply WHERE clause

<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>10</td>
<td>35.0</td>
</tr>
<tr>
<td>25.5</td>
<td>35.0</td>
</tr>
<tr>
<td>35.0</td>
<td>35.0</td>
</tr>
<tr>
<td>25.5</td>
<td>35.0</td>
</tr>
<tr>
<td>33.5</td>
<td>35.0</td>
</tr>
<tr>
<td>25.5</td>
<td>35.0</td>
</tr>
</tbody>
</table>

**Step 3:** Apply GROUP BY according to the listed attributes

<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>10</td>
<td>35.0</td>
</tr>
<tr>
<td>25.5</td>
<td>35.0</td>
</tr>
<tr>
<td>35.0</td>
<td>35.0</td>
</tr>
<tr>
<td>25.5</td>
<td>35.0</td>
</tr>
<tr>
<td>33.5</td>
<td>35.0</td>
</tr>
<tr>
<td>25.5</td>
<td>35.0</td>
</tr>
</tbody>
</table>

**Step 4:** Apply HAVING clause

<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>10</td>
<td>35.0</td>
</tr>
<tr>
<td>25.5</td>
<td>35.0</td>
</tr>
<tr>
<td>35.0</td>
<td>35.0</td>
</tr>
<tr>
<td>25.5</td>
<td>35.0</td>
</tr>
<tr>
<td>33.5</td>
<td>35.0</td>
</tr>
<tr>
<td>25.5</td>
<td>35.0</td>
</tr>
</tbody>
</table>

Answer relation:

- rating: 7
- age: 35.0
- rating: 8
- age: 25.5
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

<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** Ratings, MIN(S. age) AS minage
**FROM** Sailors S
**WHERE** S.age \( \geq 18 \)
**GROUP BY** S.rating
**HAVING** COUNT(*) > 1

### 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
  - \((X < 100) \land (X > 1)\) is \(T \land T = T\)
  - \((X > 100) \lor (X > 1)\) is \(F \lor T = T\)
  - \((X > 100) \land (X > 1)\) is \(F \land T = F\)
  - NOT\((X = 5)\) is \(\neg T = F\)
- Intuitively,
  - \(T = 1, F = 0\)
  - For \(V_1, V_2 \in \{1, 0\}\)
  - \(V_1 \land V_2 = \min(V_1, V_2)\)
  - \(V_1 \lor V_2 = \max(V_1, V_2)\)
  - \(\neg(V_1) = 1 - V_1\)

### 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 \(V_1, V_2 \in \{1, 0, 0.5\}\)
  - \(V_1 \land V_2 = \min(V_1, V_2)\)
  - \(V_1 \lor V_2 = \max(V_1, V_2)\)
  - \(\neg(V_1) = 1 - V_1\)
- Therefore,
  - NOT UNKNOWN = UNKNOWN
  - UNKNOWN OR TRUE = TRUE
  - UNKNOWN AND TRUE = UNKNOWN
  - UNKNOWN AND FALSE = FALSE
  - UNKNOWN OR FALSE = UNKNOWN
New issues for Null

- The presence of null complicates many issues. E.g.:
  - Special operators needed to check if value IS/IS NOT NULL
  - Be careful!
  - "WHERE X = NULL" does not work!
  - Need to write "WHERE X IS NULL"
- Meaning of constructs must be defined carefully
  - e.g., WHERE clause eliminates rows that don’t evaluate to true
  - So not only FALSE, but UNKNOWNS are eliminated too
  - very important to remember!
- But NULL allows new operators (e.g. outer joins)
- Arithmetic with NULL
  - all of +, *, / return null if any argument is null
- Can force “no nulls” while creating a table
  - sname char(20) NOT NULL
  - primary key is always not null

Aggregates with NULL

- What do you get for
  - SELECT count(*) from R1?
  - SELECT count(rating) from R1?
  - Ans: 3 for both

<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</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>

R1

- What do you get for
  - SELECT count(*) from R2?
  - SELECT count(rating) from R2?
  - Ans: First 3, then 2

<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</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>null</td>
<td>55</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>

R2

- What do you get for
  - COUNT, SUM, AVG, MIN, MAX (with or without DISTINCT)
    - Discards null values first
    - Then applies the aggregate
    - Except count(*)
  - If only applied to null values, the result is null

<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>null</td>
<td>45</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>null</td>
<td>55</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>null</td>
<td>35</td>
</tr>
</tbody>
</table>

R3

<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>null</td>
<td>45</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>null</td>
<td>55</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>null</td>
<td>35</td>
</tr>
</tbody>
</table>

R3
Creating Relations in SQL

- Creates the "Students" relation
  - the type (domain) of each field is specified
  - enforced by the DBMS whenever tuples are added or modified
- As another example, the "Enrolled" table holds information about courses that students take

```
CREATE TABLE Students
(sid CHAR(20),
 name CHAR(20),
 login CHAR(10),
 age INTEGER,
 gpa REAL)
```

```
CREATE TABLE Enrolled
(sid CHAR(20),
 cid CHAR(20),
 grade CHAR(2))
```

Destroying and Altering Relations

- Destroys the relation Students
  - The schema information and the tuples are deleted.
- The schema of Students is altered by adding a new field; every tuple in the current instance is extended with a NULL value in the new field.

Adding and Deleting Tuples

- Can insert a single tuple using:
  
  ```
  INSERT INTO Students (sid, name, login, age, gpa)
  VALUES (53688, 'Smith', 'smith@ee', 18, 3.2)
  ```
- Can delete all tuples satisfying some condition (e.g., name = Smith):
  
  ```
  DELETE FROM Students S
  WHERE S.name = 'Smith'
  ```

Integrity Constraints (ICs)

- IC: condition that must be true for any instance of the database
  - e.g., domain constraints
  - ICs are specified when the schema is defined
  - ICs are checked when relations are modified
- A legal instance of a relation is one that satisfies all specified ICs
  - DBMS will not allow illegal instances
- If the DBMS checks ICs, stored data is more faithful to real-world meaning
  - Avoids data entry errors, too!

Keys in a Database

- Key / Candidate Key
- Primary Key
- Super Key
- Foreign Key

- Primary key attributes are underlined in a schema
  - Person(pid, address, name)
  - Person2(address, name, age, job)

Primary Key Constraints

- A set of fields is a key for a relation if:
  1. No two distinct tuples can have same values in all key fields, and
  2. This is not true for any subset of the key
- Part 2 false? A superkey
- If there are > 1 keys for a relation, one of the keys is chosen (by DBA = DB admin) to be the primary key
  - E.g., sid is a key for Students
  - The set (sid, gpa) is a superkey.
- Any possible benefit to refer to a tuple using primary key (than any key)?
Primary and Candidate Keys in SQL

- Possibly many candidate keys
  - specified using `UNIQUE`
  - one of which is chosen as the primary key.

- “For a given student and course, there is a single grade.”

CREATE TABLE Enrolled
  (sid CHAR(20),
   cid CHAR(20),
   grade CHAR(2),
   PRIMARY KEY ???)

Possibly many candidate keys – specified using `UNIQUE` – one of which is chosen as the primary key.

- “For a given student and course, there is a single grade.”

CREATE TABLE Enrolled
  (sid CHAR(20),
   cid CHAR(20),
   grade CHAR(2),
   PRIMARY KEY (sid, cid))

Possibly many candidate keys – specified using `UNIQUE` – one of which is chosen as the primary key.

- “For a given student and course, there is a single grade.”

CREATE TABLE Enrolled
  (sid CHAR(20),
   cid CHAR(20),
   grade CHAR(2),
   PRIMARY KEY (sid, cid),
   UNIQUE (cid, grade))

Possibly many candidate keys – specified using `UNIQUE` – one of which is chosen as the primary key.

- “For a given student and course, there is a single grade.”

CREATE TABLE Enrolled
  (sid CHAR(20),
   cid CHAR(20),
   grade CHAR(2),
   PRIMARY KEY sid,
   UNIQUE (cid, grade))

Possibly many candidate keys – specified using `UNIQUE` – one of which is chosen as the primary key.

- “For a given student and course, there is a single grade.”

CREATE TABLE Enrolled
  (sid CHAR(20),
   cid CHAR(20),
   grade CHAR(2),
   PRIMARY KEY sid, cid)

Possibly many candidate keys – specified using `UNIQUE` – one of which is chosen as the primary key.

- “For a given student and course, there is a single grade.”

CREATE TABLE Enrolled
  (sid CHAR(20),
   cid CHAR(20),
   grade CHAR(2),
   PRIMARY KEY (?)
   UNIQUE ???)

Possibly many candidate keys – specified using `UNIQUE` – one of which is chosen as the primary key.

- “For a given student and course, there is a single grade.”

CREATE TABLE Enrolled
  (sid CHAR(20),
   cid CHAR(20),
   grade CHAR(2),
   PRIMARY KEY (sid, cid)
   UNIQUE ???)

Possibly many candidate keys – specified using `UNIQUE` – one of which is chosen as the primary key.

- “For a given student and course, there is a single grade.”

CREATE TABLE Enrolled
  (sid CHAR(20),
   cid CHAR(20),
   grade CHAR(2),
   PRIMARY KEY (sid, cid))

Possibly many candidate keys – specified using `UNIQUE` – one of which is chosen as the primary key.

- “For a given student and course, there is a single grade.”

CREATE TABLE Enrolled
  (sid CHAR(20),
   cid CHAR(20),
   grade CHAR(2),
   PRIMARY KEY sid, cid)
**Foreign Keys in SQL**

- Only students listed in the Students relation should be allowed to enroll for courses

```sql
CREATE TABLE Enrolled
    (sid CHAR(20),
     cid CHAR(20),
     grade CHAR(2),
     PRIMARY KEY (sid,cid),
     FOREIGN KEY (sid) REFERENCES Students )
```

**Enforcing Referential Integrity**

- Consider Students and Enrolled
  - sid in Enrolled is a foreign key that references Students.
- What should be done if an Enrolled tuple with a non-existent student id is inserted?
  - Reject it!
- What should be done if a Students tuple is deleted?
  - Three semantics allowed by SQL
    1. Also delete all Enrolled tuples that refer to it (cascade delete)
    2. Disallow deletion of a Students tuple that is referred to
    3. Set sid in Enrolled tuples that refer to it to a default sid
    4. (in addition in SQL): Set sid in Enrolled tuples that refer to it to a special value null, denoting 'unknown' or 'inapplicable'

**Where do ICs Come From?**

- ICs are based upon the semantics of the real-world enterprise that is being described in the database relations
- Can we infer ICs from an instance?
  - We can check a database instance to see if an IC is violated, but we CAN NEVER infer that an IC is true by looking at an instance.
  - An IC is a statement about all possible instances!
- From example, we know name is not a key, but the assertion that sid is a key is given to us.
- Key and foreign key ICs are the most common; more general ICs supported too

**Example Instances**

- What does the key (sid, bid, day) in Reserves mean?
- If the key for the Reserves relation contained only the attributes (sid, bid), how would the semantics differ?

```sql
CREATE TABLE Enrolled
    (sid CHAR(20) DEFAULT '000',
     cid CHAR(20),
     grade CHAR(2),
     PRIMARY KEY (sid,cid),
     FOREIGN KEY (sid) REFERENCES Students ON DELETE CASCADE ON UPDATE SET DEFAULT )
```

**Views**

- A view is just a relation, but we store a definition, rather than a set of tuples

```sql
CREATE VIEW YoungActiveStudents
    (name, grade)
AS
    SELECT S.name, E.grade
    FROM Students S, Enrolled E
    WHERE S.age < 21
```

- Views can be dropped using the DROP VIEW command
- Views and Security: Views can be used to present necessary information (or a summary), while hiding details in underlying relation(s)
  - the above view hides courses "cid" from E
- More on views later in the course
Can create a new table from a query on other tables too

```
SELECT ... INTO ... FROM ... WHERE
```

Example:

```
SELECT S.name, E.grade
INTO YoungActiveStudents
FROM Students S, Enrolled E
WHERE S.sid = E.sid and S.age < 21
```

“WITH” clause – very useful!

- You will find “WITH” clause very useful!
- CAN SIMPLIFY COMPLEX NESTED QUERIES

```
WITH Temp1 AS
(SELECT ...)
Temp2 AS
(SELECT ...)
SELECT X, Y
FROM Temp1, Temp2
WHERE ...
```

Overview: General Constraints

- **Useful when more general ICs than keys are involved**
- **There are also ASSERTIONS to specify constraints that span across multiple tables**
- **There are TRIGGERS too:** procedure that starts automatically if specified changes occur to the DBMS

```
CREATE TABLE Sailors
    ( sid INTEGER,
      sname CHAR(10),
      rating INTEGER,
      age REAL,
      PRIMARY KEY (sid),
      CHECK (rating >= 1 AND rating <= 10))
```

```
CREATE TABLE Reserves
    ( sname CHAR(10),
      bid INTEGER,
      day DATE,
      PRIMARY KEY (bid,day),
      CONSTRAINT noInterlakeRes
      CHECK (sname <> 'Interlake'))
```

Triggers

- **Trigger:** procedure that starts automatically if specified changes occur to the DBMS
- **Three parts:**
  - **Event** (activates the trigger)
  - **Condition** (tests whether the trigger should run)
  - **Action** (what happens if the trigger runs)

```
CREATE TRIGGER youngSailorUpdate
AFTER INSERT ON Sailors
REFERENCING NEW TABLE NewSailors
FOR EACH STATEMENT
INSERT INTO YoungSailors
(sid, name, age, rating)
SELECT sid, name, age, rating
FROM NewSailors
WHERE age <= 18
```

Summary: SQL

- SQL has a huge number of constructs and possibilities
  - You need to learn and practice it on your own
  - Given a problem, you should be able to write a SQL query and verify whether a given one is correct
- **Pay attention to NULLS**
- **Can limit answers using “LIMIT” or “TOP” clauses**
  - e.g. to output TOP 20 results according to an aggregate
  - also can sort using ASC or DESC keywords