The Relational Model and Basic SQL

Chapter 3
(Except 3.5)

Why Use a DBMS?

- Data independence and efficient access.
- Reduced application development time.
- Data integrity and security.
- Uniform data administration.
- Concurrent access, recovery from crashes.
Data Independence *

- Applications insulated from how data is structured and stored.
- *Logical data independence*: Protection from changes in logical structure of data.
- *Physical data independence*: Protection from changes in physical structure of data.

- One of the most important benefits of using a DBMS!

Levels of Abstraction

- Many views, single conceptual (logical) schema and physical schema.
  - Views describe how users see the data.
  - Conceptual schema defines logical structure
  - Physical schema describes the files and indexes used.

- Schemas are defined using DDL; data is modified/queried using DML.
Relational Database: Definitions

- **Relational database**: a set of relations
- **Relation**: made up of 2 parts:
  - **Instance**: a table, with rows and columns.
    #Rows = cardinality, #fields = degree / arity.
  - **Schema**: specifies name of relation, plus name and type of each column.
    • E.G. Students(sid: string, name: string, login: string, age: integer, gpa: real).
- Can think of a relation as a set of rows or tuples (i.e., all rows are distinct).

Example: University Database

- **Conceptual schema**:
  - **Students**: sid: string, name: string, login: string, age: integer, gpa: real
  - **Courses**: cid: string, cname: string, credits: integer
  - **Enrolled**: sid: string, cid: string, grade: string
- **Physical schema**:
  - Relations stored as unordered files.
  - Index on first column of Students.
- **External Schema (View)**:
  - **Course_info**: cid: string, enrollment: integer
Example Instance of Students Relation

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@eecs</td>
<td>18</td>
<td>3.2</td>
</tr>
<tr>
<td>53650</td>
<td>Smith</td>
<td>smith@math</td>
<td>19</td>
<td>3.8</td>
</tr>
</tbody>
</table>

- Cardinality = 3, degree = 5, all rows distinct
- Do all columns in a relation instance have to be distinct?

Relational Query Languages

- A major strength of the relational model: supports simple, powerful querying of data.
- Queries can be written intuitively, and the DBMS is responsible for efficient evaluation.
  - The key: precise semantics for relational queries.
  - Allows the optimizer to extensively re-order operations, and still ensure that the answer does not change.
The SQL Language

- Developed by IBM (system R) in the 1970s
- Need for a standard since it is used by many vendors
- Standards:
  - SQL-86
  - SQL-89 (minor revision)
  - SQL-92 (major revision)
  - SQL-99 (major extensions, current standard)
- Both a DDL and a DML

The SQL Query Language

- To find all 18 year old students, we can write:
  
  ```sql
  SELECT * 
  FROM Students S 
  WHERE S.age=18 
  ```

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>18</td>
<td>3.2</td>
</tr>
</tbody>
</table>

  • To find just names and logins, replace the first line:
  ```sql
  SELECT S.name, S.login
  ```
Querying Multiple Relations

- What does the following query compute?

```
SELECT S.name, E.cid
FROM Students S, Enrolled E
WHERE S.sid=E.sid AND E.grade="A"
```

Given the following instances of Enrolled and Students:

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@eecs</td>
<td>18</td>
<td>3.2</td>
</tr>
<tr>
<td>53650</td>
<td>Smith</td>
<td>smith@math</td>
<td>19</td>
<td>3.8</td>
</tr>
</tbody>
</table>

![Table](sid|cid|grade)

- We get:

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53831</td>
<td>Carnatic101</td>
<td>C</td>
</tr>
<tr>
<td>53831</td>
<td>Reggae203</td>
<td>B</td>
</tr>
<tr>
<td>53650</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>53666</td>
<td>History105</td>
<td>B</td>
</tr>
</tbody>
</table>

Creating Relations in SQL

- Creates the Students relation. Observe that the type (domain) of each field is specified, and enforced by the DBMS whenever tuples are added or modified.

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

- As another example, the Enrolled table holds information about courses that students take.

```
CREATE TABLE Enrolled
(sid: CHAR(20),
 cid: CHAR(20),
 grade: CHAR(2))
```
**Some Attribute Domains in SQL**

- INTEGER
- DECIMAL(p,q): p-bit numbers, with q bits right of decimal
- FLOAT(p): p-bit floating point numbers
- CHAR(n): fixed length character string, length n
- VARCHAR(n): variable length character string, max. length n
- DATE: describes a year, month, day
- TIME: describes an hour, minute, second
- TIMESTAMP: describes a date and a time on that date
- YEAR/MONTH INTERVAL: time interval
- DAY/TIME INTERVAL: time interval
- … and many more

**Destroying and Altering Relations**

**DROP TABLE Students**

- Destroys the relation Students. The schema information *and* the tuples are deleted.

**ALTER TABLE Students**

```
  ADD COLUMN firstYear: integer
```

- 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:
  
  ```sql
  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):
  
  ```sql
  DELETE
  FROM Students S
  WHERE S.name = 'Smith'
  ```

Updating Tuples

- Can update all tuples using:
  
  ```sql
  UPDATE Students
  SET gpa = gpa * 1.1
  ```

- Can update all tuples satisfying some condition (e.g., name = Smith):
  
  ```sql
  UPDATE Students
  SET gpa = gpa * 1.1
  WHERE name = 'Smith'
  ```

* Powerful variants of these commands are available; more later!
**Integrity Constraints (ICs)**

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

**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’s >1 key for a relation, one of the keys is chosen (by DBA) to be the *primary key*.
- E.g., *sid* is a key for Students. (What about *name*)? The set \{sid, gpa\} is a superkey.
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.” vs. “Students can take only one course, and receive a single grade for that course; further, no two students in a course receive the same grade.”

- Used carelessly, an IC can prevent the storage of database instances that arise in practice!

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

---

Foreign Keys, Referential Integrity

- **Foreign key**: Set of fields in one relation that is used to `refer` to a tuple in another relation. (Must correspond to primary key of the second relation.) Like a `logical pointer`.

- E.g. *sid* is a foreign key referring to Students:
  - Enrolled(*sid*: string, *cid*: string, *grade*: string)
  - If all foreign key constraints are enforced, **referential integrity** is achieved, i.e., no dangling references.
  - Can you name a data model w/o referential integrity?
    - • Links in HTML!
**Foreign Keys in SQL**

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

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

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Carnatic101</td>
<td>C</td>
</tr>
<tr>
<td>53666</td>
<td>Reggae203</td>
<td>B</td>
</tr>
<tr>
<td>53650</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>53666</td>
<td>History105</td>
<td>B</td>
</tr>
</tbody>
</table>

**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?
  - Also delete all Enrolled tuples that refer to it.
  - Disallow deletion of a Students tuple that is referred to.
  - Set *sid* in Enrolled tuples that refer to it to a *default sid*.
  - *(In SQL, also: Set *sid* in Enrolled tuples that refer to it to a special value `null`, denoting 'unknown' or 'inapplicable'.)*
- Similar if primary key of Students tuple is updated.
**Referential Integrity in SQL**

- SQL/92 and SQL:1999 support all 4 options on deletes and updates.
  - Default is **NO ACTION** (delete/update is rejected)
  - **CASCADE** (also delete all tuples that refer to deleted tuple)
  - **SET NULL** / **SET DEFAULT** (sets foreign key value of referencing tuple)

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

**Where do ICs Come From?**

- ICs are based upon the semantics of the real-world enterprise that is being described in the database relations.

- 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.
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.sid = E.sid and S.age<21
  ```

- Views can be dropped using the **DROP VIEW** command.
  - How to handle **DROP TABLE** if there’s a view on the table?
    - **DROP TABLE** command has options to let the user specify this.

Views and Security

- Views can be used to present necessary information (or a summary), while hiding details in underlying relation(s).
  - Given YoungStudents, but not Students or Enrolled, we can find students s who have are enrolled, but not the cid’s of the courses they are enrolled in.