The Relational Model
Why Study the Relational Model?

• Most widely used model, by far: IBM, Microsoft, Oracle, MySQL, PostgreSQL, ...
• Older models (hierarchical, network) still around in legacy systems
• Other alternative models: object-oriented, object-relational, NoSQL, NewSQL, …
Relational Database: A Set of Relations

- A relation consists of 2 parts:
  - Schema: name of relation, name and type (or domain) of each column of the relation
    Students(sid: string, name: string, login: string, age: integer, gpa: real)
  - Instance: a table, with rows and columns
    # rows = cardinality, # fields (or attributes) = degree/arity

- Can think of a relation as a set of rows or tuples, without repetition (i.e., all rows are distinct)
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>1</td>
<td>Luis</td>
<td>l@cs</td>
<td>85</td>
<td>3.2</td>
</tr>
<tr>
<td>3</td>
<td>Panos</td>
<td>p@cs</td>
<td>30</td>
<td>4.0</td>
</tr>
<tr>
<td>4</td>
<td>Amelie</td>
<td>a@cs</td>
<td>30</td>
<td>4.1</td>
</tr>
<tr>
<td>5</td>
<td>Hila</td>
<td>h@cs</td>
<td>29</td>
<td>4.1</td>
</tr>
</tbody>
</table>

Cardinality=4, degree=5, all rows (or tuples) distinct

Do all columns in a relation instance have to be distinct?
Creating Relations in SQL

- Creates the Students relation; the type (domain) of each attribute is specified, and enforced by the DBMS when tuples are added or modified
  
  ```sql
  CREATE TABLE Students
  (sid CHAR(20),
   name CHAR(30),
   login CHAR(20),
   age INTEGER,
   gpa REAL)
  ```

- Creates the Courses relation, which holds information about available courses
  
  ```sql
  CREATE TABLE Courses
  (cid CHAR(10),
   cname CHAR(10),
   credits INTEGER)
  ```
Creating Relations in SQL

- Creates the Enrolled relation, which holds information about courses that students take, and the corresponding grades

```sql
CREATE TABLE Enrolled (sid CHAR(20), cid CHAR(10), grade CHAR(2))
```
Adding and Deleting Tuples

Can insert a single tuple:

```
INSERT INTO Students(sid, name, login, age, gpa)
VALUES (8, 'Nico', 'nico@cs', 24, 3.9)
```

Can delete all tuples satisfying some condition (e.g., name="Nico"): 

```
DELETE
FROM Students S
WHERE S.name = 'Nico'
```

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

- Integrity constraints are conditions (e.g., domain constraints) that must be true for any instance of the database
  - 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 will not allow illegal instances, so that stored data preserves intended real-world meaning and also data entry errors are avoided
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
  2. Minimality: Property 1 is not true for any strict subset of the key (i.e., we need all attributes in the set to satisfy the previous point)
- If a set of fields only satisfies Property 1, then the set is a superkey (e.g., \{sid\} is a key for Students, and \{sid, gpa\} is a superkey)
- If a relation has multiple keys, one is designated as the primary key and the rest are candidate keys
Primary and Candidate Keys in SQL

One PRIMARY KEY per relation; zero or more candidate keys specified with the UNIQUE keyword

Two different students cannot share the same value of sid or login; {sid} is the primary key, {login} is a candidate key

```
CREATE TABLE Students
(sid CHAR(20),
 name CHAR(30),
 login CHAR(20),
 age INTEGER,
gpa REAL,
PRIMARY KEY(sid),
UNIQUE(login))
```

Two different courses cannot share the same value of cid; {cid} is the primary key

```
CREATE TABLE Courses
(cid CHAR(10),
cname CHAR(10),
credits INTEGER,
PRIMARY KEY(cid))
```
Primary and Candidate Keys in SQL

One PRIMARY KEY per relation; zero or more candidate keys specified with the UNIQUE keyword

Each student can only take a given course once

CREATE TABLE Enrolled
(sid CHAR(20),
cid CHAR(10),
grade CHAR(2),
PRIMARY KEY(sid, cid))
<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Luis</td>
<td>l@cs</td>
<td>85</td>
<td>3.2</td>
</tr>
<tr>
<td>3</td>
<td>Panos</td>
<td>p@cs</td>
<td>30</td>
<td>4.0</td>
</tr>
<tr>
<td>4</td>
<td>Amelie</td>
<td>a@cs</td>
<td>30</td>
<td>4.1</td>
</tr>
<tr>
<td>5</td>
<td>Hila</td>
<td>h@cs</td>
<td>29</td>
<td>4.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cid</th>
<th>cname</th>
<th>credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DB</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>OS</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>PLT</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>C-</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>A+</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>A+</td>
</tr>
</tbody>
</table>
Primary and Candidate Keys in SQL: One Important Difference

- Primary key attributes cannot be “NULL”
  (much more on this later in the course)
- Candidate key attributes can be “NULL”
Foreign Keys, Referential Integrity

- Attributes of one relation can reference tuples of another relation.
- To avoid “dangling” references and guarantee referential integrity, we use “foreign keys”.
- A foreign key is a set of attributes in one relation that is used to refer to tuples in another relation.
- The foreign key attributes in a relation correspond to the primary key attributes in the referred to relation.
Foreign Keys in SQL

Only students listed in the Students relation should be allowed to enroll for courses; analogous constraint for Courses

CREATE TABLE Enrolled
(sid CHAR(20),
cid CHAR(10),
grade CHAR(2),
PRIMARY KEY (sid, cid),
FOREIGN KEY (sid) REFERENCES Students,
FOREIGN KEY (cid) REFERENCES Courses)
## Students

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Luis</td>
<td>l@cs</td>
<td>85</td>
<td>3.2</td>
</tr>
<tr>
<td>3</td>
<td>Panos</td>
<td>p@cs</td>
<td>30</td>
<td>4.0</td>
</tr>
<tr>
<td>4</td>
<td>Amelie</td>
<td>a@cs</td>
<td>30</td>
<td>4.1</td>
</tr>
<tr>
<td>5</td>
<td>Hila</td>
<td>h@cs</td>
<td>29</td>
<td>4.1</td>
</tr>
</tbody>
</table>

## Courses

<table>
<thead>
<tr>
<th>cid</th>
<th>cname</th>
<th>credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DB</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>OS</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>PLT</td>
<td>4</td>
</tr>
</tbody>
</table>

## Enrolled

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>C-</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>A+</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>A+</td>
</tr>
</tbody>
</table>
Enforcing Referential Integrity

• Consider Students and Enrolled: sid in Enrolled is a foreign key that references (the primary key attribute of) Students
• What should be done if an Enrolled tuple with a nonexistent student id is inserted? (Reject it!)
• What should be done if a Students tuple is deleted? Options:
  1. Reject deletion of the Students tuple if any Enrolled tuples refer to it
  2. Delete also all Enrolled tuples that refer to it
  3. Set sid in Enrolled tuples that refer to it to a “default” sid value, which should be present in Students
  4. Set sid in Enrolled tuples that refer to it to a special value “NULL,” which denotes “unknown” or “inapplicable”; NULL values don’t violate referential integrity
• What if the primary key attribute of a Students tuple is updated? Analogous options…
Foreign Keys in SQL

CREATE TABLE Enrolled
    (sid CHAR(20),
     cid CHAR(10),
     grade CHAR(2),
     PRIMARY KEY (sid, cid),
     FOREIGN KEY (sid) REFERENCES Students
       ON DELETE NO ACTION
       CASCADE
       SET DEFAULT
       SET NULL,
     FOREIGN KEY (cid) REFERENCES Courses
       ON UPDATE NO ACTION
       CASCADE
       SET DEFAULT
       SET NULL,
Where do ICs Come From?

- ICs are based on the semantics of the real-world enterprise described in the 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 a database instance. An IC is a statement about all possible instances.
- Primary/candidate key and foreign key ICs are the most common; more general ICs supported too, as we will see later.
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
  - Relational queries have precise semantics
  - The DBMS query optimizer can extensively reorder operations to answer a query, and still ensure that the query results are correct
The SQL Query Language

• SQL is the most widely used relational query language
• To find all 30-year-old students, with all their attributes, we can write:

```sql
SELECT * FROM Students S WHERE S.age=30
```

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Panos</td>
<td>p@cs</td>
<td>30</td>
<td>4.0</td>
</tr>
<tr>
<td>4</td>
<td>Amelie</td>
<td>a@cs</td>
<td>30</td>
<td>4.1</td>
</tr>
</tbody>
</table>

• To find just the name and login attributes, replace the first line with:

```sql
SELECT S.name, S.login
```
Querying Multiple Relations

What does the following query request?

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

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Luis</td>
<td>l@cs</td>
<td>85</td>
<td>3.2</td>
</tr>
<tr>
<td>3</td>
<td>Panos</td>
<td>p@cs</td>
<td>30</td>
<td>4.0</td>
</tr>
<tr>
<td>4</td>
<td>Amelie</td>
<td>a@cs</td>
<td>30</td>
<td>4.1</td>
</tr>
<tr>
<td>5</td>
<td>Hila</td>
<td>h@cs</td>
<td>29</td>
<td>4.1</td>
</tr>
</tbody>
</table>

### Courses

<table>
<thead>
<tr>
<th>cid</th>
<th>cname</th>
<th>credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DB</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>OS</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>PLT</td>
<td>4</td>
</tr>
</tbody>
</table>

### Enrolled

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>C-</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>A+</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>A+</td>
</tr>
</tbody>
</table>

SELECT S.name, E.cid
FROM Students S, Enrolled E
WHERE S.sid=E.sid AND E.grade='A'
Semantics of a SQL Query

A conceptual evaluation method for the previous query:

1. FROM clause: Compute cross-product of Students and Enrolled
2. WHERE clause: Check conditions, discard tuples that fail
3. SELECT clause: Delete unwanted fields

This is only conceptual, to specify the meaning of the query and what its results should be; an actual evaluation will typically be much more efficient, but must produce the same answers as the conceptual evaluation
### Cross-product of Students and Enrolled Instances

<table>
<thead>
<tr>
<th>S.sid</th>
<th>S.name</th>
<th>S.login</th>
<th>S.age</th>
<th>S.gpa</th>
<th>E.sid</th>
<th>E.cid</th>
<th>E.grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Luis</td>
<td>l@cs</td>
<td>85</td>
<td>3.2</td>
<td>1</td>
<td>1</td>
<td>C-</td>
</tr>
<tr>
<td>1</td>
<td>Luis</td>
<td>l@cs</td>
<td>85</td>
<td>3.2</td>
<td>4</td>
<td>1</td>
<td>A+</td>
</tr>
<tr>
<td>1</td>
<td>Luis</td>
<td>l@cs</td>
<td>85</td>
<td>3.2</td>
<td>3</td>
<td>1</td>
<td>A+</td>
</tr>
<tr>
<td>3</td>
<td>Panos</td>
<td>p@cs</td>
<td>30</td>
<td>4.0</td>
<td>1</td>
<td>1</td>
<td>C-</td>
</tr>
<tr>
<td>3</td>
<td>Panos</td>
<td>p@cs</td>
<td>30</td>
<td>4.0</td>
<td>4</td>
<td>1</td>
<td>A+</td>
</tr>
<tr>
<td>3</td>
<td>Panos</td>
<td>p@cs</td>
<td>30</td>
<td>4.0</td>
<td>3</td>
<td>1</td>
<td>A+</td>
</tr>
<tr>
<td>4</td>
<td>Amelie</td>
<td>a@cs</td>
<td>30</td>
<td>4.1</td>
<td>1</td>
<td>1</td>
<td>C-</td>
</tr>
<tr>
<td>4</td>
<td>Amelie</td>
<td>a@cs</td>
<td>30</td>
<td>4.1</td>
<td>5</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>Amelie</td>
<td>a@cs</td>
<td>30</td>
<td>4.1</td>
<td>4</td>
<td>1</td>
<td>A+</td>
</tr>
<tr>
<td>4</td>
<td>Amelie</td>
<td>a@cs</td>
<td>30</td>
<td>4.1</td>
<td>3</td>
<td>1</td>
<td>A+</td>
</tr>
<tr>
<td>5</td>
<td>Hila</td>
<td>h@cs</td>
<td>29</td>
<td>4.1</td>
<td>1</td>
<td>1</td>
<td>C-</td>
</tr>
<tr>
<td>5</td>
<td>Hila</td>
<td>h@cs</td>
<td>29</td>
<td>4.1</td>
<td>5</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>Hila</td>
<td>h@cs</td>
<td>29</td>
<td>4.1</td>
<td>4</td>
<td>1</td>
<td>A+</td>
</tr>
<tr>
<td>5</td>
<td>Hila</td>
<td>h@cs</td>
<td>29</td>
<td>4.1</td>
<td>3</td>
<td>1</td>
<td>A+</td>
</tr>
</tbody>
</table>
Mapping ER Diagrams to Relational Database Schemas

Entity sets get mapped to tables

CREATE TABLE Employees
(ssn CHAR(11),
name CHAR(20),
lot INTEGER,
PRIMARY KEY (ssn))

CREATE TABLE Departments
(did INTEGER,
dname CHAR(20),
budget REAL,
PRIMARY KEY (did))
Relationship sets without constraints also get mapped to a relation, with:

- Attributes for the primary key of each participating entity set, specified as foreign keys
- All descriptive attributes for the relationship set

CREATE TABLE Works_In(
    ssn CHAR(11),
    did INTEGER,
    since DATE,
    PRIMARY KEY (ssn, did),
    FOREIGN KEY (ssn) REFERENCES Employees,
    FOREIGN KEY (did) REFERENCES Departments
)
Translating ER Diagrams with Key Constraints
Translating ER Diagrams with Key Constraints

**Option 1:** Separate tables for Employees and Departments, plus table for Manages, with did as key (3 tables total)

**Option 2:** Table for Employees plus table for Departments + Manages together, with did as key (2 tables total): possible only because each department has at most one manager

CREATE TABLE `Manages`(
    ssn CHAR(11),
    did INTEGER,
    since DATE,
    PRIMARY KEY (did),
    FOREIGN KEY (ssn) REFERENCES Employees,
    FOREIGN KEY (did) REFERENCES Departments)

CREATE TABLE `Dept_Mgr`(
    did INTEGER,
    dname CHAR(20),
    budget REAL,
    ssn CHAR(11),
    since DATE,
    PRIMARY KEY (did),
    FOREIGN KEY (ssn) REFERENCES Employees)
Now let’s introduce a participation constraint...
Participation Constraints in SQL

We can capture participation constraints involving one entity set in a binary relationship if that entity set also has a key constraint, as in the previous slide, but little else (without resorting to CHECK constraints, which we will see much later)

```
CREATE TABLE Dept_Mgr(
    did INTEGER,
    dname CHAR(20),
    budget REAL,
    ssn CHAR(11) NOT NULL,
    since DATE,
    PRIMARY KEY (did),
    FOREIGN KEY (ssn) REFERENCES Employees,
    ON DELETE NO ACTION)
```
Participation Constraints Without Key Constraints?

Now Departments has **total participation** in `Works_In** (without a key constraint)**: Unfortunately, we cannot model such total participation case in SQL (yet). For now, when you map such diagrams to SQL in your homework and projects, just mention you cannot capture such participation constraints (yet) in SQL.
Translating Weak Entity Sets

Weak entity set and identifying relationship set are translated into a single table.

When the owner entity is deleted, all owned weak entities must also be deleted.

```
CREATE TABLE Dep_Policy (
    pname CHAR(20),
    age INTEGER,
    cost REAL,
    ssn CHAR(11),
    PRIMARY KEY (pname, ssn),
    FOREIGN KEY (ssn) REFERENCES Employees,
    ON DELETE CASCADE
)
```
Translating ISA Hierarchies?

- Employees
  - name
  - ssn
  - lot
  - ISA
  - Hourly_Emps
    - hourly_wages
    - hours_worked
  - Contract_Emps
    - contract_id
Translating ISA Hierarchies to Relations

**General approach uses 3 relations:** Employees, Hourly_Emps, and Contract_Emps

- Employees “records” every employee with the attributes of the Employees entity set.
- Hourly_Emps “records” the ssn of every hourly employee, which is a foreign key into Employees, plus attributes hourly_wages and hours_worked; must delete an Hourly_Emps tuple if the referenced Employees tuple is deleted (i.e., ON DELETE CASCADE).
- Contract_Emps is defined analogously as Hourly_Emps.
- Queries involving all employees are efficient, but those involving just Hourly_Emps or Contract_Emps require a join to get employee attributes beyond ssn, which can be costly.
CREATE TABLE Employees(
  ssn CHAR(11),
  name CHAR(20),
  lot INTEGER,
  PRIMARY KEY (ssn));
CREATE TABLE Hourly_Emps(
  ssn CHAR(11),
  hourly_wages REAL,
  hours_worked INTEGER,
  PRIMARY KEY (ssn),
  FOREIGN KEY (ssn) REFERENCES Employees
  ON DELETE CASCADE);
CREATE TABLE Contract_Emps(
  ssn CHAR(11),
  contract_id CHAR(10),
  PRIMARY KEY (ssn),
  FOREIGN KEY (ssn) REFERENCES Employees
  ON DELETE CASCADE);

Use this 3-table mapping if:
• Hourly_Emps and Contract_Emps can OVERLAP, or
• Hourly_Emps and Contract_Emps CANNOT OVERLAP but they do NOT COVER Employees, or
• Employees participates in one or more relationship sets or additional ISA hierarchies
What if:
1. Hourly_Emps and Contract_Emps CANNOT OVERLAP and
2. Hourly_Emps and Contract_Emps COVER Employees and
3. Employees does not participate in any relationship sets or additional ISA hierarchies?

If all of conditions 1 through 3 hold, then you can:
• Remove table Employees, and the foreign keys into it from Hourly_Emps and Contract_Emps
• Add name and lot into Hourly_Emps and Contract_Emps
Translating ISA Hierarchies to Relations

What if Hourly_Emps and Contract_Emps can OVERLAP?

• Follow the 3-table design in the earlier slide (otherwise name and lot would be redundantly stored twice for employees in the intersection, which is bad for data integrity)

What if Employees participates in a relationship set or another ISA hierarchy?

• Again, follow the 3-table design in the earlier slide (otherwise you couldn’t refer to “generic” employees with a single foreign key from other tables, which you need to do in this case)
Relational Model: Summary

• The relational model provides a tabular representation of data
• Simple and intuitive, currently the most widely used
• Integrity constraints can be specified based on application semantics
• DBMS prevents violations of the integrity constraints
  • Domain constraints
  • Primary and candidate keys, as well as foreign keys
  • Many others!
• A powerful and natural query language exists for the relational model, as we will see in detail