# ISYS2120 – Data & Information Management

**Week 3A:** SQL DDL, and DML for data changes

(Kifer/Bernstein/Lewis - Chapter 3; Ramakrishnan/Gehrke - Chapter 3)

Based on slides from Kifer/Bernstein/Lewis (2006) “Database Systems” and from Ramakrishnan/Gehrke (2003) “Database Management Systems”, and including material from Fekete and Röhm.

Prof Alan Fekete

# Schema definition

- A relational dbms knows the schema of the database it manages
  - Indeed, data cannot be stored except when the schema is already known
    - This is in contrast to some other platforms (eg JSON stores), which can accept data and then be told (or even figure out) the schema
- The schema is described in SQL statements (part of Data Definition Language, also abbreviated as DDL)
  - Give names of tables, names of columns, datatypes of columns, and also various constraints
- There are also commands that modify the schema
- Schema descriptions are stored in the system catalogue, which can be queried
  - Eg a table which gives information about the attributes of all tables

# Creating Tables in SQL

- Creation of tables:

```
CREATE TABLE name ( list-of-columns-with-types )
```

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

```
CREATE TABLE Instructor (lname VARCHAR(20),
                         fname  VARCHAR(20),
                         salary INTEGER,
                         birth  DATE,
                         hired  DATE );
```

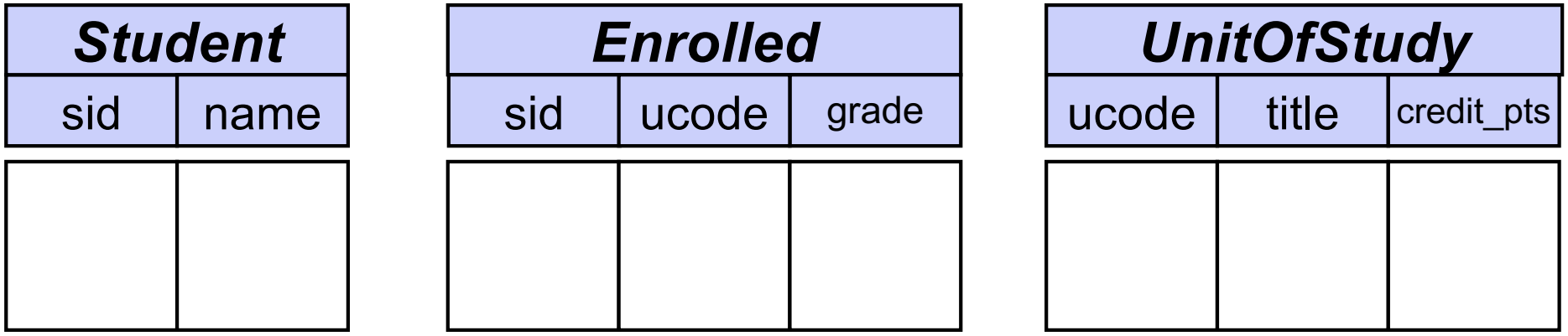
  - Remember that table and column names are SQL identifiers (and case-insensitive)
  - Remember not to use any keyword as an identifier (eg, don't try to have a column called `date`)
  - After performing this, the table exists, but it starts empty (contains no rows yet)

# Base Datatypes of SQL

| Base Datatypes | Description | Example Values |
|---|---|---|
| SMALLINT<br>INTEGER<br>BIGINT | Integer values | 1704, 4070 |
| DECIMAL(p,q)<br>NUMERIC(p,q) | Fixed-point numbers with precision *p* and *q* decimal places | 1003.44, 160139.9 |
| FLOAT(p)<br>REAL<br>DOUBLE PRECISION | floating point numbers with precision *p* | 1.5E-4, 10E20 |
| CHAR(q)<br>VARCHAR(q)<br>CLOB(q) | alphanumerical character string types<br>of fixed size *q* respectively<br>of variable length of up to *q* chars | 'The quick brown fox jumps…', 'ISYS2120' |
| BLOB(r) | binary string of size r | B'01101', X'9E' |
| DATE | date | DATE '1997-06-19', DATE '2001-08-23' |
| TIME | time | TIME '20:30:45', TIME '00:15:30' |
| TIMESTAMP | timestamp | TIMESTAMP '2002-08-23 14:15:00' |
| INTERVAL | time interval | INTERVAL '11:15' HOUR TO MINUTE |
| | | |

*(cf. Türker, ORDBMS 2004/2005)*

# Create Table Example




-- Note: layout on lines is not significant!

```
CREATE TABLE Student (
    sid     INTEGER,
    name VARCHAR(20)
);
CREATE TABLE UnitOfStudy
(
    ucode CHAR(8),
    title VARCHAR(30),
    creditPoints INTEGER
);
CREATE TABLE Enrolled (
     sid INTEGER, ucode CHAR(8), grade INTEGER
);
```

# The Special NULL 'Value'

- SQL-based RDBMS allows a special entry ***NULL*** in a column to represent facts that are not relevant, or not yet known
  - This can occur in a column of any datatype
  - This is different than the VARCHAR 'Null'
- Eg a new employee has not yet been allocated to a department
- Eg salary, hired may not be meaningful for adjunct lecturers
- Eg INSTRUCTOR table in a university

| lname | fname | salary | birth | hired |
|---|---|---|---|---|
| Jones | Peter | 35000 | 1970 | 1998 |
| Smith | Susan | null | 1983 | null |
| Smith | Alan | 35000 | 1975 | 2000 |

# Evaluation of use of NULL

- Advantage:
  NULL is useful because using an ordinary value with special meaning does not always work
  - Eg if salary=-1 is used for “unknown” in the previous example, then averages won’t be sensible

- Disadvantage:
  NULL causes complications in the definition of many operations (eg is 27 < NULL? Or 27 > NULL?)

# SQL Schemas & Table Modifications

- Database Servers typically shared by multiple users
  - We want separate schemas per user (naming: `schema.tablename`)
  - **`CREATE SCHEMA`** … command
    - E.g. http://download.oracle.com/docs/cd/B19306_01/server.102/b14200/statements_6014.htm or http://www.postgresql.org/docs/8.3/static/sql-createschema.html
  - If not provided, automatic schema by user name (cf. Oracle)

- Several base data types available in SQL
  - E.g. `INTEGER, REAL, CHAR, VARCHAR, DATE, ...`
  - but each system has its specialities such as specific BLOB types or value range restrictions
    - E.g. Oracle calls a string for historical reasons `VARCHAR2`
  - cf. online documentation

- Existing schemas can be changed

  **`ALTER TABLE`** *name* **`ADD COLUMN`** … | **`ADD CONSTRAINT`**… | …
  - Huge variety of vendor-specific options; cf. online documentation

# Integrity Constraints

- Integrity Constraint (IC): a condition that must be true for *any* instance of the database; e.g., *domain constraints, uniqueness constraints* and more complicated ones too.
  - ICs can be declared in the schema
    - They are specified when schema is defined.
    - All declared ICs are checked whenever relations are modified.
- A *legal* instance of a relation is one that satisfies all specified ICs.
  - If ICs are declared, DBMS will not allow illegal instances.
- When the DBMS checks ICs, stored data is more faithful to real-world meaning.
  - Can avoid some data entry errors, too!

# Non-Null Columns

- One domain constraint is to insist that no value in a given column can be null
  - The value can't be unknown; The concept can't be inapplicable

- In SQL, append NOT NULL to the field declaration

```
CREATE TABLE Instructor
                    ( lname VARCHAR(20) NOT NULL,
                      fname  VARCHAR(20) NOT NULL,
                      salary INTEGER,
                      birth  DATE NOT NULL,
                      hired  DATE );
```

# Primary Key

- Recall that many entities have identifiers, which can be used to distinguish one entity from another in the entity set, even when many attributes are the same
  - Often, an identifier is completely artificial, and the value has no meaning at all
  - Sometimes, the identifier is a sequence number
- In a relational schema, we can declare that a column is a Primary Key (often abbreviated as PK in textbooks or diagrams)
  - This allows it to be used as identifier in other tables, to connect information
  - It will enforce that every row has a different value for this column
  - It will enforce that no row has NULL as value for this column

# SQL ways to indicate a PK

- Append to end of column definition

```
CREATE TABLE Student (
    sid     INTEGER PRIMARY
    KEY,
    name VARCHAR(20)
);
```

- Extra clause in table creation

```
CREATE TABLE Student (
    sid     INTEGER,
    name VARCHAR(20),
   PRIMARY KEY (sid)
);
```

- Named constraint in table creation

```
CREATE TABLE Student (
    sid     INTEGER,
    name VARCHAR(20),
   CONSTRAINT Student_PK
            PRIMARY KEY (sid)
);
```

- Separate named constraint added later

```
CREATE TABLE Student (
    sid     INTEGER,
    name VARCHAR(20)
);
ALTER TABLE Student (
   ADD CONSTRAINT Student_PK
    PRIMARY KEY (sid)
);
```

# Composite PK

- In some tables, no single column is an identifier, but a combination of columns may be able to distinguish the rows
- Eg, sometimes, each manufacturer has its own set of serial numbers,
  - it is possible to have two rows with the same serialno, when they have different manufacturer
  - And it is possible to have two rows with the same manufacturer, when they have different serialno
  - But the combination (manufacturer, serialno) is suitable as an identifier
  - This is called a composite primary key
- In SQL

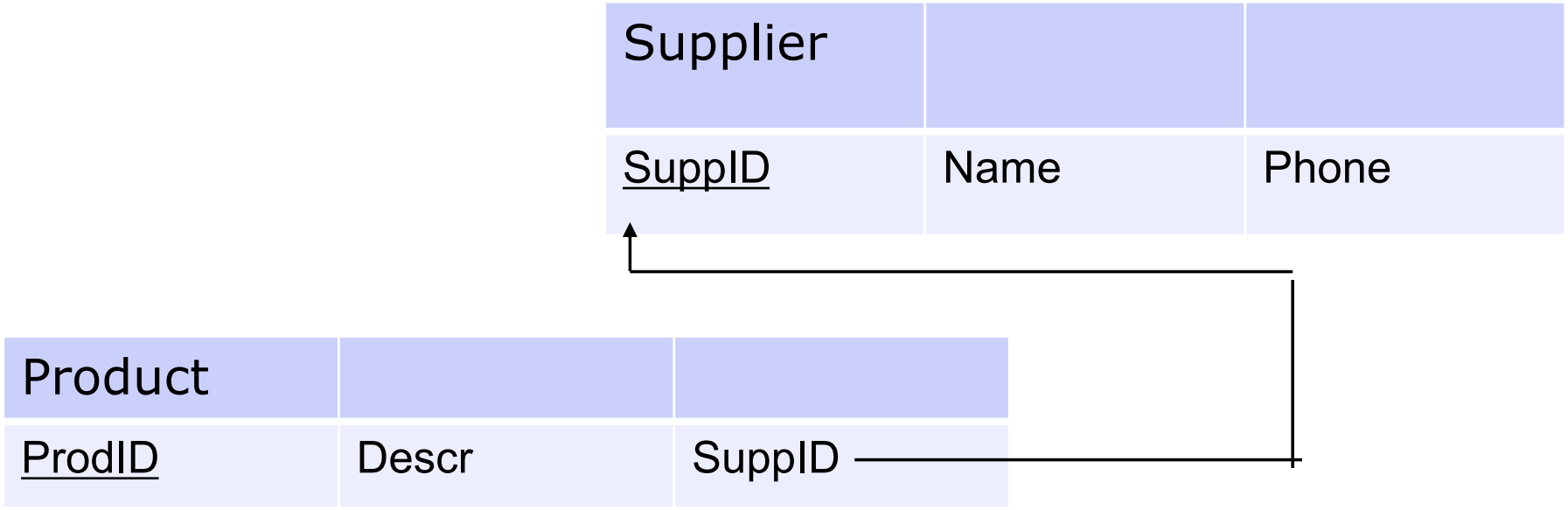
```
CREATE TABLE Product (
    manufacturer VARCHAR(20),
    serialno INTEGER,
    PRIMARY KEY (manufacturer, serialno)
);
```

# Uniqueness

- A table can have only one primary key declared (may be a single column, or a combination of columns)
- Sometimes, there are other columns or combinations that are necessarily different among rows
  - We call any combination of columns which can distinguish the rows, as a candidate key
- SQL allows multiple constraints on a table, which say a column or combination is UNIQUE

# Foreign Key

- When we use values to connect information across tables, we expect that there will be something with that value in the referenced table
- We can make this explicit with a FOREIGN KEY constraint (often abbreviated FK in text or diagrams)
- Recall the relational schema diagram



# Foreign Key Constraint in SQL

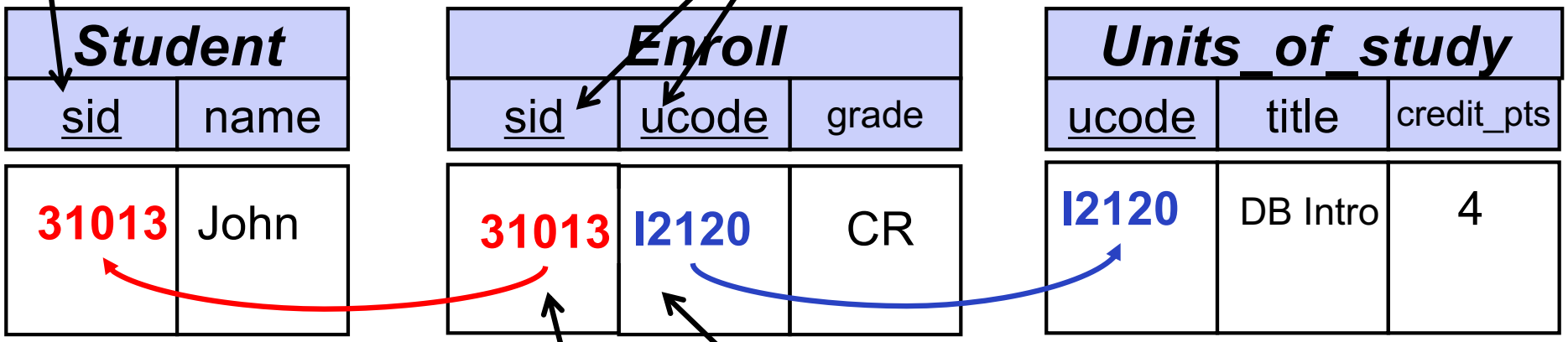
- Several ways to express this, among them

```
CREATE TABLE Product ( ProdID INTEGER,
    descr VARCHAR(10),
    SuppID INTEGER REFERENCES Supplier(SuppID));
Or
CREATE TABLE Product ( ProdID INTEGER,
    descr VARCHAR(10),
    SuppID INTEGER REFERENCES Supplier);
Or      -- When no column is mentioned for referenced table, implicit reference to its primary key
CREATE TABLE Product ( ProdID INTEGER,
    descr VARCHAR(10),
    SuppID INTEGER,
    FOREIGN KEY (SuppID) REFERENCES Supplier(SuppID));
Or
CREATE TABLE Product ( ProdID INTEGER,
    descr VARCHAR(10),
    SuppID INTEGER,
    CONSTRAINT Product_FK FOREIGN KEY (SuppID) REFERENCES
     Supplier(SuppID));
```

# Summary of terms

**Primary key** identifies each tuple of a relation.

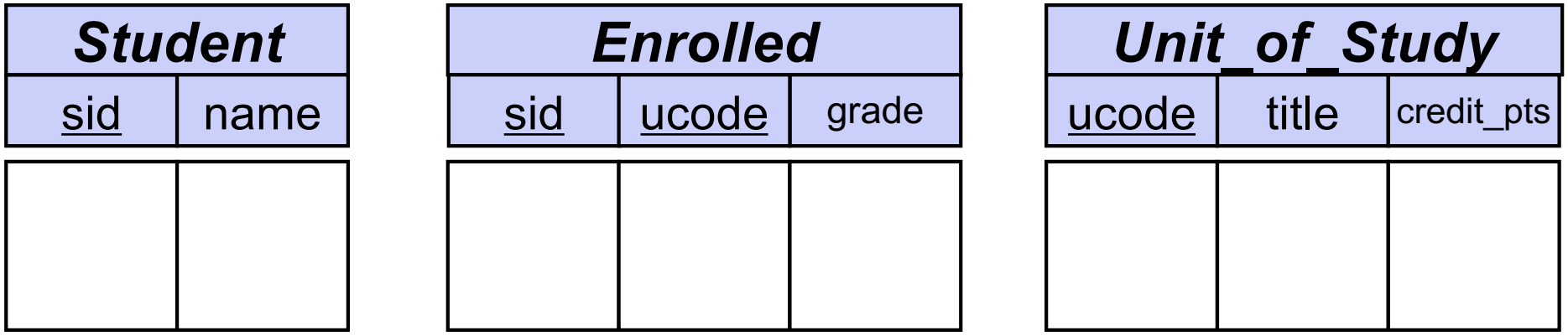
**Composite Primary Key** consisting of more than one attribute.




**Foreign key** is a (set of) attribute(s) in one relation that `refers' to a tuple in another relation by giving the value of its key (like a `logical pointer').

Another foreign key

# Create Table Example with PKs/FKs




```
CREATE TABLE Student ( sid INTEGER, … ,
    CONSTRAINT Student_PK PRIMARY KEY (sid)
);
CREATE TABLE UoS  ( ucode CHAR(8), … ,
    CONSTRAINT UoS_PK PRIMARY KEY (ucode)
);
CREATE TABLE Enrolled ( sid INTEGER, ucode CHAR(8), grade CHAR(2),
    CONSTRAINT Enrolled_FK1 FOREIGN KEY (sid) REFERENCES Student,
    CONSTRAINT Enrolled_FK2 FOREIGN KEY (ucode) REFERENCES UoS,
    CONSTRAINT Enrolled_PK  PRIMARY KEY (sid,ucode)
);
```

# Choosing the Correct Key Constraints

- Careful: Used carelessly, an IC can prevent the storage of database instances that arise in practice!
- Example:
  Attempt to model that a student can get only a single grade per course.

```
CREATE TABLE Enrolled  (
  sid   INTEGER,
  cid   CHAR(8),
  grade CHAR(2),
  PRIMARY KEY (sid,cid) )
```

vs.

```
CREATE TABLE Enrolled  (
    sid   INTEGER,
    cid   CHAR(8),
    grade CHAR(2),
  PRIMARY KEY (sid, cid),
  UNIQUE (sid, grade) )
```

- "For a given student and course, there is a single grade; the same grade can be achieved by several students in a course."
- "For a given student and course, there is a single grade; but a student can achieve *a certain grade only once*."

# Brainteaser

- Given the following example a table

```
CREATE TABLE Test (
        a INTEGER,
        b INTEGER UNIQUE,
        PRIMARY KEY (a,b)
    );
```

- Would the following be a legal database instance?

{ (1, 1) ,
(1, 2) ,
(1, 3) ,
(2, 1) ,
(2, 4) }

# Keys and NULLs

- PRIMARY KEY
  - Must be unique and do not allow NULL values
- UNIQUE (candidate key)
  - Possibly many *candidate keys* (specified using UNIQUE)
  - According to the ANSI standards SQL:92, SQL:1999, and SQL:2003, a UNIQUE constraint should disallow duplicate non-NULL values, but allow multiple NULL values.
  - Many DBMS (e.g. Oracle or SQL Server) implement only a crippled version of this, allowing a single NULL but disallowing multiple NULL values….
- FOREIGN KEY
  - By default allows nulls
  - If there must be a parent tuple, then must combine with NOT NULL constraint

# Foreign Keys & Referential Integrity

- **Referential Integrity**:
  for each tuple in the referring relation whose foreign key value is **X**, there must be a tuple in the referred relation with a candidate key that also has value **X**
  - e.g. *sid* is a foreign key referring to Student:
    Enrolled(*sid*: integer, ucode: string, grade: string)
- When all foreign key constraints are enforced, referential integrity is achieved, i.e., no dangling references

Q: Can you name a data model w/o referential integrity?

# How to enforce Referential Integrity

- Consider Student and Enrolled;
  *sid* in Enrolled is a foreign key that references Student.

- What should be done if an Enrolled tuple with a non-existent student *sid* is inserted? (*Reject it!*)

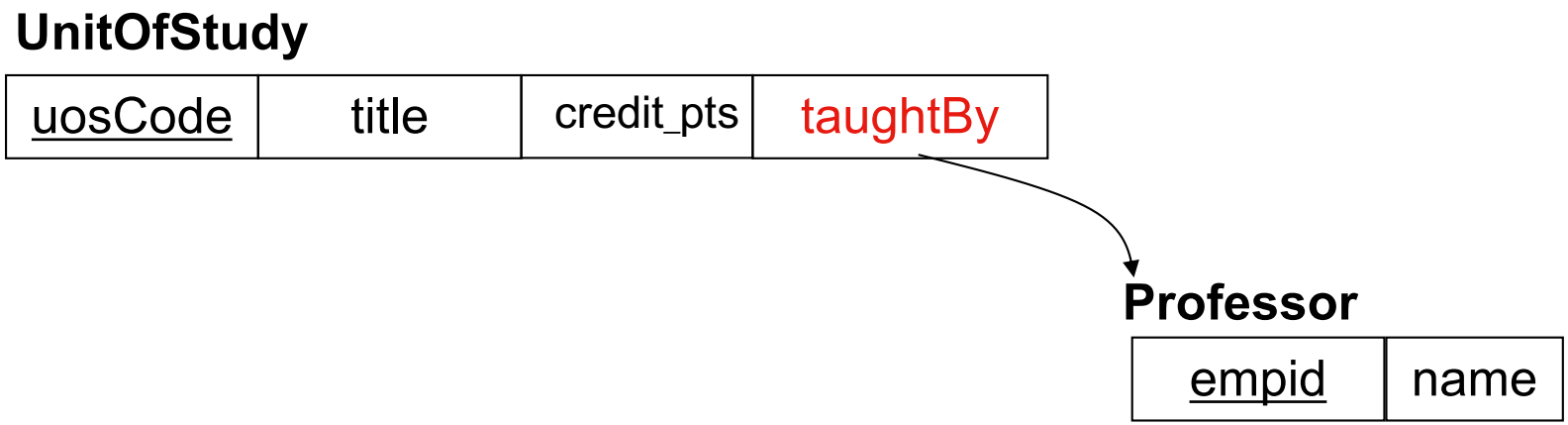
- What should be done if a Student tuple is deleted? Choices:
  - Also delete all Enrolled tuples that refer to it.
  - Disallow deletion of a Student tuple that is referred to.
  - Set sid in Enrolled tuples that refer to it to a *default sid*.
  - (In SQL, also: Set sid in Enrolled tuples that refer to it to a special value *null*, denoting *`unknown'* or *`inapplicable'*.)

- Similar if primary key of Student tuple is updated.

# Referential Integrity in SQL

- SQL/92, SQL:1999 and SQL:2003 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)

```
CREATE TABLE UnitOfStudy
( uosCode    CHAR(8),
  title      VARCHAR(80),
  credit_pts  INTEGER,
  taughtBy INTEGER DEFAULT 1,
    PRIMARY KEY (uosCode),
    FOREIGN KEY (taughtBy)
    REFERENCES  Professor
      ON UPDATE CASCADE
      ON DELETE SET DEFAULT )
```



# CHECK Constraints

- In SQL the domain of a column is given as a simple datatype
- Sometimes, we know more about the valid values that can be there
  - Eg Salary should be positive
  - Eg UoSCode should be 4 letters then 4 digits
- We can define a Boolean property that is required to always be true for every row

```
CREATE TABLE Employee
( empID  INTEGER,
  …
  salary INTEGER CHECK (salary > 0), …);
```

- If the property involves several columns, it needs to be a separate clause in the SQL, not just appended to the column declaration

# Changing schema in SQL

- Deletion of a tables:

```
DROP TABLE name
```

  - the schema information and the tuples in the table, are all removed.
  - Example: Destroy the Instructor relation

```
DROP TABLE Instructor
```

- Existing schemas can be changed

```
ALTER TABLE name ADD COLUMN ... | ADD CONSTRAINT... | ...
```

  - Huge variety of vendor-specific options; cf. online documentation

# Modifying Instances using SQL DML

- Insertion of new data into a table / relation
  - **Syntax:**
    **INSERT INTO** *table* [“(”*list-of-columns*“)”] **VALUES** “(“ list-of-*expression* “)”
  - Example:

```
INSERT INTO Student (sid, name) VALUES (12345678, 'Smith')
```

- Updating of tuples in a table / relation
  - **Syntax:**
    **UPDATE** *table* **SET** *column*“=“*expression* {“,”*column*“=“*expression*}
    [ **WHERE** *search_condition* ]
  - Example:

```
UPDATE Student
   SET address = '4711 Water Street'
 WHERE sid = 123456789
```

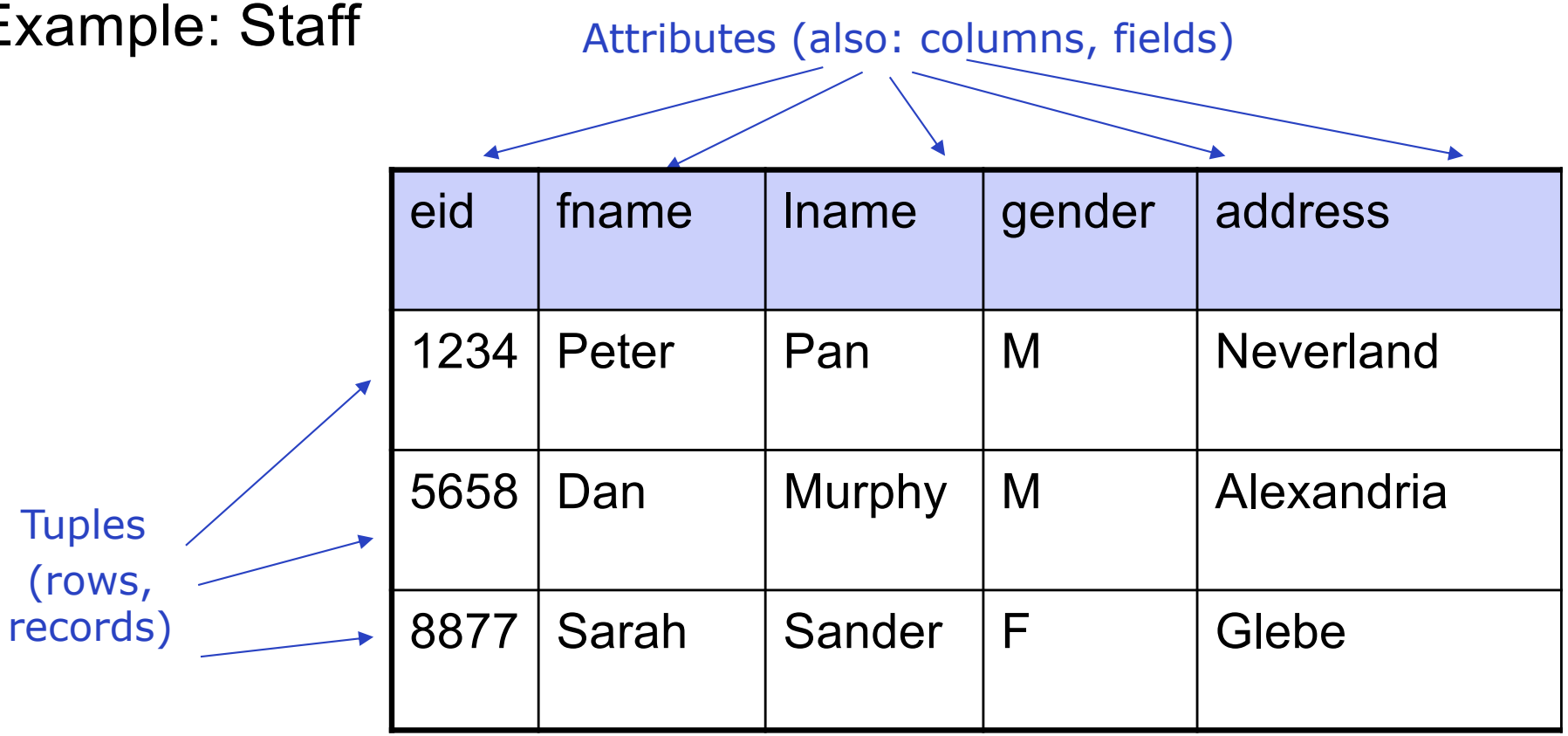
- Deleting of tuples from a table / relation
  - **Syntax:**
    **DELETE FROM** *table* [ **WHERE** *search_condition* ]
  - Example:

```
DELETE FROM Student WHERE name = 'Smith'
```

WHERE clause
as in SELECT statement

# Recall terminology

- **Informal Definition:**
  A ***relation*** is a named, two-dimensional table of data
  - Table consists of rows (record) and columns (attribute or field)
- Example: Staff




| eid | fname | lname | gender | address |
|---|---|---|---|---|
| 1234 | Peter | Pan | M | Neverland |
| 5658 | Dan | Murphy | M | Alexandria |
| 8877 | Sarah | Sander | F | Glebe |

# Maths Definition of a Relation

- **A Relation is a mathematical concept from discrete maths based on the ideas of sets.**
  - ***Relation R***
    Given sets $D_1$, $D_2$, …, $D_n$, a relation $R$ is a subset of $D_1$ x $D_2$ x…x $D_n$
    Thus, a relation is a set of $n$-tuples ($a_1$, $a_2$, …, $a_n$) where $a_i \in D_i$
- Example:

If

*studentid = {12345678, 23456789, 345354345, 44455666, etc}*

*name = {Jones, Smith, Kerry, Lindsay, etc}*

*date = {1985-11-09, 1984-07-15, 1984-12-01, 1986-01-01, etc}*

then

$R$ = { (12345678, Jones, 1984-07-15),
(345354345, Lindsay, 1986-01-01),
(44455666, Kerry, 1985-11-09),
(23456789, Kerry, 1994-07-15) }

is a relation over *studentid* x *name* x *date*

# Theory vs. Technology

- A relational DBMS supports data in a form close to, but not exactly, matching the mathematical relation
  - RDBMS allows null values for unknown information
  - RDBMS gives a name for each position among the columns
  - RDBMS insists that datatype for any column is from a limited variety, all simple
  - RDBMS considers the columns as being arranged in an order
  - RDBMS considers the rows as being arranged in an order
  - RDBMS allows duplicate rows

# References

- Kifer/Bernstein/Lewis (2nd edition)
  - Chapter 3
- Ramakrishnan/Gehrke (3rd edition - the 'Cow' book)
  - Chapter 3.1-3.4 and 3.6-3.7, plus Chapter 1.5
- Ullman/Widom (3rd edition)
  - Chapter 2.1 - 2.3, Section 7.1 and Chapter 8.1-8.2
  - *views and foreign keys come later, instead relational algebra is introduced very early on; also briefly compares RDM with XML,*
- Silberschatz/Korth/Sudarshan (5th edition - 'sailing boat')
  - Chapter 2.1 - 2.2; Chapter 3.1-3.2 and 3.9
  - *starts with relational algebra early on which we do later*
- Elmasri/Navathe (5th edition)
  - Chapter 2.1-2.3; Chapter 5; Section 8.8
  - *talks first more about system architectures and conceptual modeling*