CSE532
Theory of Database Systems
The Relational Data Model

CSE 532, Theory of Database Systems

Stony Brook University

http://www.cs.stonybrook.edu/~cse532

Table

- For relational databases data is stored in tables.
- Set of rows (no duplicates)
- Each *row* describes a different entity
- Each *column* states a particular fact about each entity
  - Each column has an associated *domain*
    - Domain of *Status* = {fresh, soph, junior, senior}

Id	*Name*	*Address*	*Status*
1111	John	123 Main	fresh
2222	Mary	321 Oak	soph
1234	Bob	444 Pine	soph
9999	Joan	777 Grand	senior

Relation

- Mathematical entity corresponding to a table
  - row ~ *tuple*
  - column ~ *attribute*
- Values in a tuple are *related* to each other
  - John lives at 123 Main
- Relation **R** can be thought of as predicate **R**
  - **R**(x,y,z) is true *iff* tuple (x,y,z) is in **R**
    - a tuple is an ordered fixed-sized sequence
  - **R** is a set of tuples: **R** = {(John, 111 Main St, fresh), …}

Creating Tables

```
CREATE TABLE Student (
   Id  INTEGER,
   Name CHAR(20),
   Address  CHAR(50),
   Status  CHAR(10),
   PRIMARY KEY (Id)  )
```

Constraint

Operations

- Operations on relations are precisely defined
  - Take relation(s) as argument, produce new relation as result
    - Unary (*e.g.*, delete certain rows)
    - Binary (*e.g.*, **union, Cartesian product**)
- Corresponding operations defined on tables as well
- Using mathematical properties, *equivalence* can be decided
  - Important for ***query optimization***:

$$\mathrm{op1}(T1,\mathrm{op2}(T2)) \overset{?}{=} \mathrm{op3}(\mathrm{op2}(T1),T2)$$

Structured Query Language: SQL

- Language for manipulating tables
- ***Declarative* – Statement specifies *what* needs to be obtained, *not how* it is to be achieved** (*e.g.*, how to access data, the order of operations)
- Due to declarativity of SQL, <u>DBMS determines evaluation strategy</u>
  - This greatly simplifies application programs
  - *But DBMS is not infallible*: programmers should have an idea of strategies used by DBMS so they can design better tables, indices, statements, in such a way that DBMS can evaluate statements efficiently

Structured Query Language (SQL)

```
SELECT <attribute list>
FROM  <table list >
WHERE <condition>
```

- Language for constructing a new table from argument table(s)
  - FROM indicates source tables
  - WHERE indicates which *rows* to retain
    - It acts as a filter
  - SELECT indicates which *columns* to extract from retained rows
    - Projection
- The result is a table.

Simple Example

```
SELECT Name
FROM   Student
WHERE  Id > 4999
```

Id	*Name*	*Address*	*Status*
1234	John	123 Main	fresh
5522	Mary	77 Pine	senior
9876	Bill	83 Oak	junior

Student

Name
Mary
Bill

Result

More Examples

SELECT *Id*, *Name* FROM Student

SELECT *Id*, *Name* FROM Student
    WHERE *Status* = ‘senior’

SELECT * FROM Student
    WHERE *Status* = ‘senior’

result is a table with one column and one row

SELECT COUNT(*) FROM Student
    WHERE *Status* = ‘senior’

Complex Example

- **Goal:** table in which each row names a senior and gives a course taken and grade
- Combines information in two tables:
  - Student: *Id*, *Name*, *Address*, *Status*
  - Transcript: *StudId*, *CrsCode*, *Semester*, *Grade*

```
SELECT Name, CrsCode, Grade
FROM  Student, Transcript
WHERE  StudId = Id AND Status = 'senior'
```

Join

SELECT *a1*, *b1*
FROM T1, T2
WHERE *a2* = *b2*

T1

a1	*a2*	*a3*
A	1	xxy
B	17	rst

T2

b1	*b2*
3.2	17
4.8	17

FROM T1, T2
yields:

a1	*a2*	*a3*	*b1*	*b2*
A	1	xxy	3.2	17
A	1	xxy	4.8	17
B	17	rst	3.2	17
B	17	rst	4.8	17

WHERE *a2* = *b2*
yields:

B	17	rst	3.2	17
B	17	rst	4.8	17

SELECT *a1*, *b1*
yields result:

B	3.2
B	4.8

Modifying Tables

UPDATE *Student*
SET *Status* = ‘soph’
WHERE *Id* = 111111111

INSERT INTO Student (*Id*, *Name*, *Address*, *Status*)
VALUES (999999999, ‘Bill’, ‘432 Pine’, ‘senior’)

DELETE FROM Student
WHERE *Id* = 111111111

Transactions

- Many enterprises use databases to store information about their state
  - *E.g.*, balances of all depositors
- The occurrence of a real-world event that changes the enterprise state requires the execution of a program that changes the database state in a corresponding way
  - *E.g.*, balance must be updated when you deposit
- A *transaction* is a program that accesses the database in response to real-world events

Transactions

- Transactions are not just ordinary programs
- Additional requirements are placed on transactions (and particularly their execution environment) that go beyond the requirements placed on ordinary programs.
  - **A**tomicity
  - **C**onsistency
  - **I**solation
  - **D**urability

$\left.\begin{array}{l}\\ \\ \\ \end{array}\right\}$ *ACID properties*

Atomicity

- A real-world event either happens or does not happen.
  - Student either registers or does not register.
- Similarly, the system must ensure that either the transaction runs to completion (*commits*) or, if it does not complete, it has no effect at all (*aborts*).
  - This is not true of ordinary programs. A hardware or software failure could leave files partially updated.

Integrity Constraints

- Rules of the enterprise generally limit the occurrence of certain real-world events.
  - **Student cannot register for a course if current number of registrants = maximum allowed**
- Correspondingly, allowable database states are restricted.
  - *cur_reg* <= *max_reg*
- These limitations are expressed as *integrity constraints,* which are assertions that must be satisfied by the database state.

Consistency

- Transaction designer must ensure that

  IF the database is in a state that satisfies all integrity constraints when execution of a transaction is started

  THEN when the transaction completes:

  - All integrity constraints are once again satisfied (constraints can be violated in intermediate states)
  - New database state satisfies specifications of transaction

Isolation

- Deals with the execution of multiple transactions concurrently.
- If the initial database state is consistent and accurately reflects the real-world state, then the *serial* (one after another) execution of a set of consistent transactions preserves consistency.
- But serial execution is *inadequate* from a performance perspective.

Durability

- The system must ensure that once a transaction commits its effect on the database state is not lost in spite of subsequent failures.
    - Not true of ordinary systems. For example, a media failure after a program terminates could cause the file system to be restored to a state that preceded the execution of the program.

More on Isolation

- Concurrent (interleaved) execution of a set of transactions offers performance benefits, but might not be correct.
- **Example**: Two students execute the course registration transaction at about the same time

(*cur_reg* is the number of current registrants)

T_1:	read(*cur_reg* : 29)			write(*cur_reg* : 30)
T_2:		read(*cur_reg* : 29)	write(*cur_reg* : 30)	
time $\rightarrow$				

Result: Database state no longer corresponds to real-world state, integrity constraint violated.

More on Isolation

- The effect of concurrently executing a set of transactions must be the same as if they had executed serially in some order
  - The execution is thus *not* serial, but *serializable*
- Serializable execution has better performance than serial, but performance might still be inadequate. Database systems offer several isolation levels with different performance characteristics (but some guarantee correctness only for certain kinds of transactions – not in general)

ACID Properties

- The transaction monitor is responsible for ensuring atomicity, durability, and (the requested level of) isolation.
  - Hence it provides the abstraction of failure-free, non-concurrent environment, greatly simplifying the task of the transaction designer.
- The transaction designer is responsible for ensuring the consistency of each transaction, but doesn't need to worry about concurrency and system failures.

Data and Its Structure

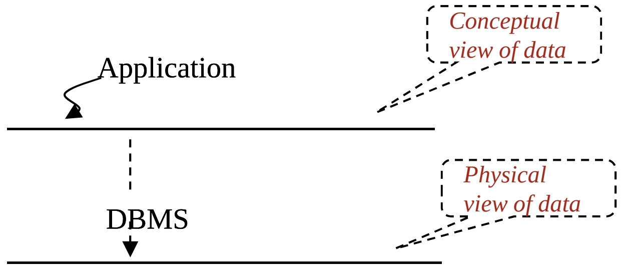
- Data is actually stored as bits, but it is difficult to work with data at this level.
- It is convenient to view data at different *levels of abstraction*.
- ***Schema***: Description of data at some abstraction level. Each level has its own schema.
- We will be concerned with three schemas: *physical*, *conceptual*, and *external*.

Physical Data Level

- *Physical schema* describes details of how data is stored: tracks, cylinders, indices etc.
- Early applications worked at this level – explicitly dealt with details.
- **Problem:** Routines were hard-coded to deal with physical representation.
    - Changes to data structure difficult to make.
    - Application code becomes complex since it must deal with details.
    - Rapid implementation of new features impossible.

Conceptual Data Level

- Hides details.
  - In the relational model, the conceptual schema presents data as a set of tables.
- DBMS maps from conceptual to physical schema automatically.



- Physical schema can be changed without changing application:
  - DBMS would change mapping from conceptual to physical transparently
  - This property is referred to as ***physical data independence***

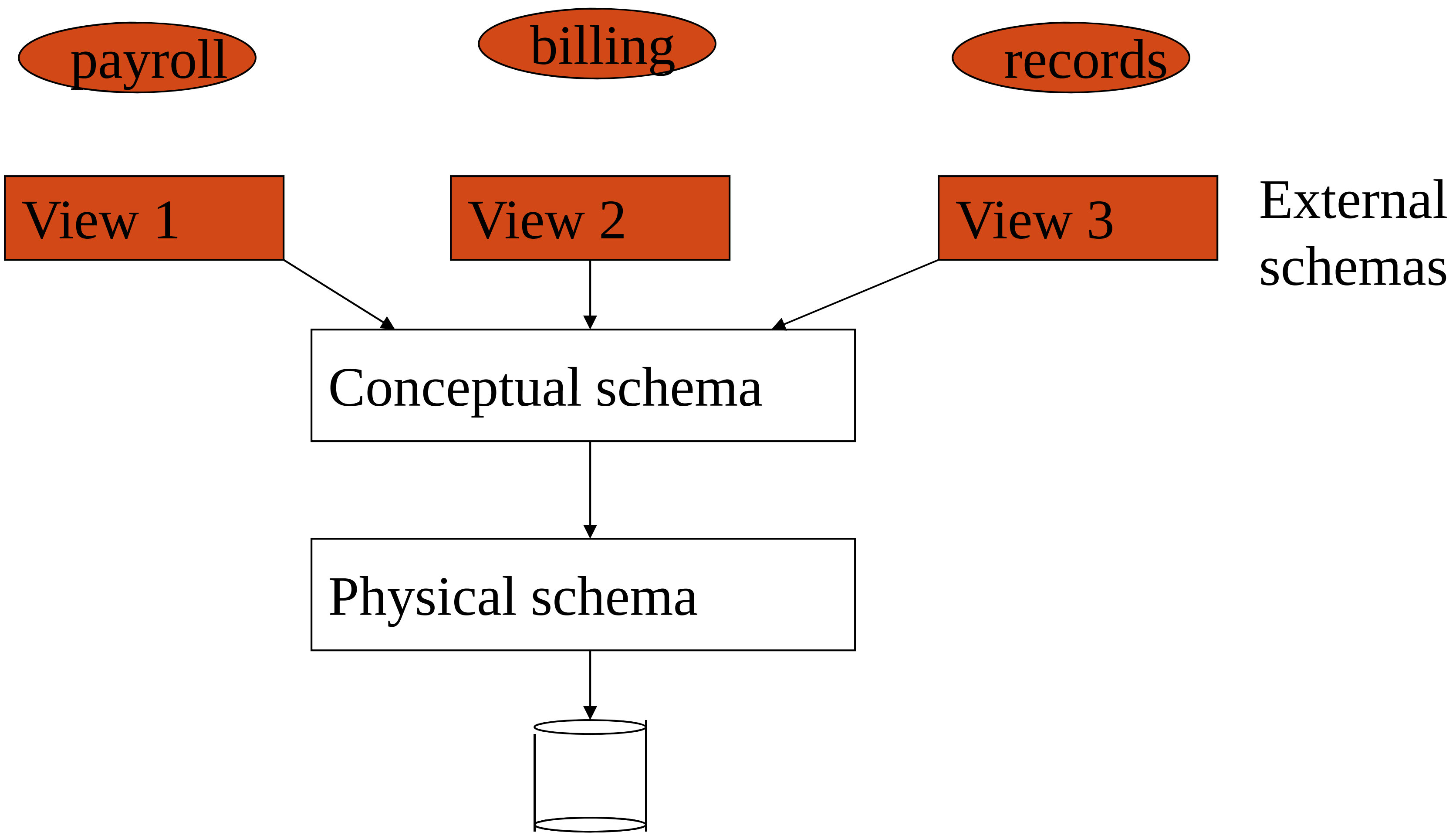
External Data Level

- In the relational model, the *external schema* also presents data as a set of relations.
- An external schema specifies a *view* of the data in terms of the conceptual level. It is tailored to the needs of a particular category of users.
  - Portions of stored data should not be seen by some users.
    - Students should not see their files in full.
    - Faculty should not see billing data.
  - Information that can be derived from stored data might be viewed as if it were stored.
    - GPA not stored, but calculated when needed.

External Data Level (con't)

- Application is written in terms of an external schema.
- A view is computed when accessed (not stored).
- Different external schemas can be provided to different categories of users.
- Translation from external to conceptual done automatically by DBMS at run time.
- Conceptual schema can be changed without changing application:
  - Mapping from external to conceptual must be changed.
- Referred to as ***conceptual data independence***.

Levels of Abstraction



Data Model

- **Schema**: description of data at some level (*e.g.*, tables, attributes, constraints, domains)
- **Model**: tools and language for describing:
  - Conceptual and external schema
    - *Data definition language* (DDL)
  - Integrity constraints, domains (DDL)
  - Operations on data
    - *Data manipulation language* (DML)
  - Directives that influence the physical schema (affects performance, not semantics)
    - *Storage definition language* (SDL)

Relational Model

- A particular way of structuring data (using relations)
- Simple
- Mathematically based
  - Expressions (≡ *queries*) can be analyzed by DBMS
  - Queries are transformed to equivalent expressions automatically (*query optimization*)
    - Optimizers have limits (=> programmer needs to know how queries are evaluated and optimized)

Relation Instance

- Relation is a set of tuples
  - Tuple ordering immaterial
  - No duplicates
  - *Cardinality* of relation = number of tuples
- All tuples in a relation have the same structure; constructed from the same set of attributes
  - Attributes are named (ordering is immaterial)
  - Value of an attribute is drawn from the attribute's *domain*
    - There is also a special value **null** (value unknown or undefined), which belongs to no domain
  - *Arity* of relation = number of attributes

Relation Schema

- Relation name
- Attribute names & domains
- Integrity constraints like
  - The values of a particular attribute in all tuples are unique
  - The values of a particular attribute in all tuples are greater than 0
- Default values

Relational Database

- Finite set of relations
- Each relation consists of a schema and an instance
- *Database schema* = set of relation schemas constraints among relations (*inter-relational* constraints)
- *Database instance* = set of (corresponding) relation instances

Database Schema (Example)

- Student (*Id*: INT, *Name*: STRING, *Address*: STRING, *Status*: STRING)
- Professor (*Id*: INT, *Name*: STRING, *DeptId*: DEPTS)
- Course (*DeptId*: DEPTS, *CrsName*: STRING, *CrsCode*: COURSES)
- Transcript (*CrsCode*: COURSES, *StudId*: INT, *Grade*: GRADES, *Semester*: SEMESTERS)
- Department(*DeptId*: DEPTS, *Name*: STRING)

Integrity Constraints

- Part of schema
- Restriction on state (or of sequence of states) of data base
- Enforced by DBMS
- *Intra-relational* - involve only one relation
  - Part of relation schema
  - e.g., all *Id*s are unique
- *Inter-relational* - involve several relations
  - Part of relation schema or database schema

Constraint Checking

- Automatically checked by DBMS
- Protects database from errors
- Enforces enterprise rules

Kinds of Integrity Constraints

- Static – restricts legal states of database
  - Syntactic (structural)
    - e.g., all values in a column must be unique
  - Semantic (involve meaning of attributes)
    - e.g., cannot register for more than 18 credits
- Dynamic – limitation on sequences of database states
    - e.g., cannot raise salary by more than 5%

Key Constraint

- A ***key constraint*** is a sequence of attributes $A_1,\ldots,A_n$ (n=1 possible) of a relation schema, **S**, with the following property:
  - A relation instance **s** of **S** satisfies the key constraint iff at most one row in **s** can contain a particular set of values, $a_1,\ldots,a_n$, for the attributes $A_1,\ldots,A_n$
  - *Minimality*: no subset of $A_1,\ldots,A_n$ is a key constraint
- ***Key***
  - Set of attributes mentioned in a key constraint
    - e.g., *Id* in Student,
    - e.g., (*StudId*, *CrsCode*, *Semester*) in Transcript
  - It is *minimal*: no subset of a key is a key
    - (*Id*, *Name*) is not a key of Student

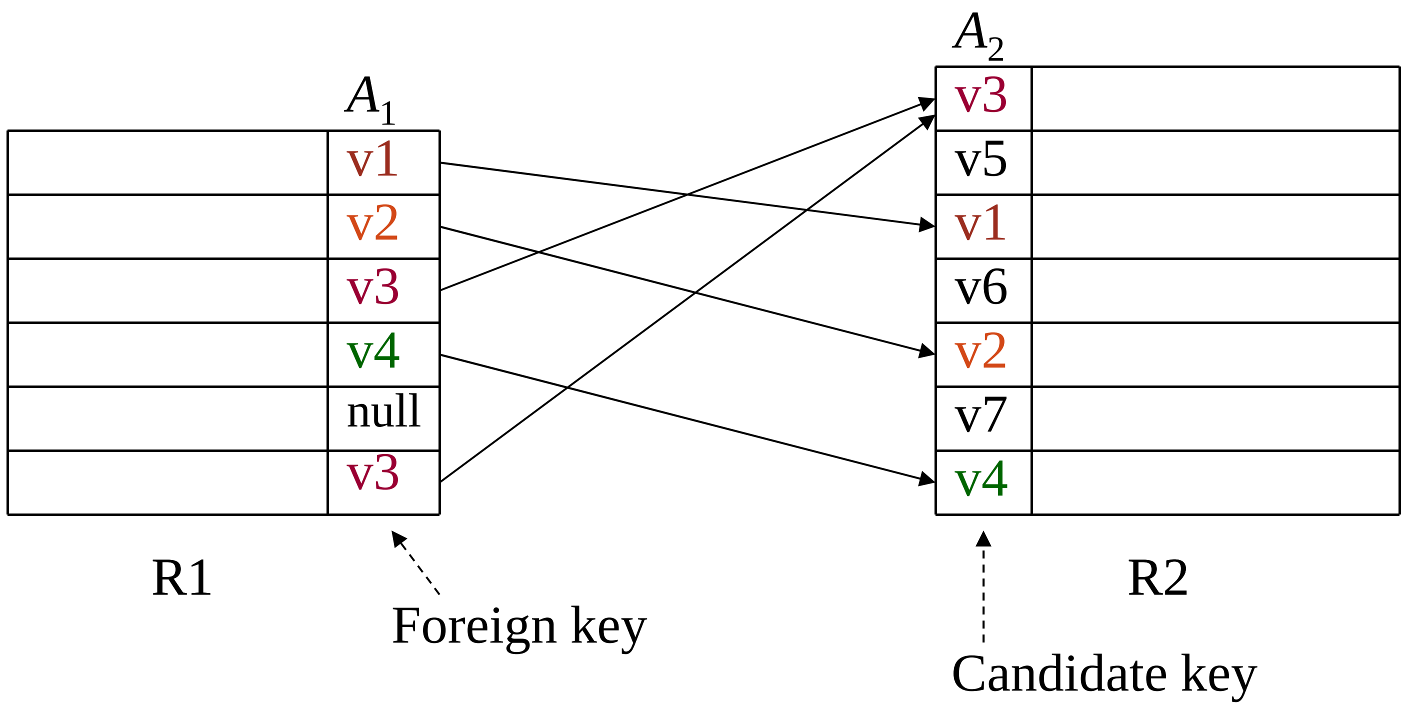
Key Constraint (cont'd)

- ***Superkey*** - set of attributes containing key
  - (*Id*, *Name*) is a superkey of Student
- Every relation has a key
- Relation can have several keys:
  - ***primary key:*** *Id* in Student (can't be **null**)
  - ***candidate key*****:** (*Name*, *Address*) in Student

Foreign Key Constraint

- ***Referential integrity:*** Item named in one relation must refer to tuples that describe that item in another
  - Transcript (*CrsCode*) references Course(*CrsCode*)
  - Professor(*DeptId*) references Department(*DeptId*)
- Attribute A_1 is a ***foreign key*** of R1 referring to attribute A_2 in R2, if whenever there is a value v of A_1, there is a tuple of R2 in which A_2 has value v, and A_2 is a key of R2
  - This is a special case of referential integrity: A_2 must be a candidate key of R2 (e.g., *CrsCode* is a key of Course in the above)
  - If no row exists in R2 => violation of referential integrity
  - Not all rows of R2 need to be referenced: relationship is not symmetric (e.g., some course might not be taught)
  - Value of a foreign key might not be specified (*DeptId* column of some professor might be **null**)

Foreign Key Constraint (Example)



Foreign Key (cont'd)

- Names of the attrs A_1 and A_2 need not be the same.
  - With tables:

    Teaching(*CrsCode*: COURSES, *Sem*: SEMESTERS, *ProfId*: INT)
    Professor(*Id*: INT, *Name*: STRING, *DeptId*: DEPTS)

  ProfId attribute of Teaching references *Id* attribute of Professor
- R1 and R2 need not be distinct.
  - Employee(*Id*:INT, *MgrId*:INT, ….)
    - Employee(*MgrId*) references Employee(*Id*)
  - Every manager is also an employee and hence has a unique row in Employee

Foreign Key (cont'd)

- Foreign key might consist of several columns
  - (*CrsCode*, *Semester*) of Transcript references
    (*CrsCode*, *Semester*) of Teaching
- R1(A_1, …A_n) <u>references</u> R2(B_1, …B_n)
  - A_i and B_i must have same domains (although not necessarily the same names)
  - B_1,…,B_n must be a candidate key of R2

Inclusion Dependency

- Referential integrity constraint that is not a foreign key constraint
- Teaching(*CrsCode*, *Semester*) references
  Transcript(*CrsCode*, *Semester*)

  (no empty classes allowed)
- Target attributes do not form a candidate key in Transcript (*StudId* missing)
- No simple enforcement mechanism for inclusion dependencies in SQL (requires *assertions -- later*)

SQL

- Language for describing database schema and operations on tables
- ***Data Definition Language*** **(**DDL**):** sublanguage of SQL for describing schema

Tables

- SQL entity that corresponds to a relation
- An element of the database schema
- SQL-92 is currently the most supported standard but is now superseded by SQL:1999 and SQL:2003
- Database vendors generally deviate from the standard, but eventually converge

Table Declaration

```
CREATE TABLE  Student (
  Id:  INTEGER,
  Name:  CHAR(20),
  Address:  CHAR(50),
  Status:  CHAR(10)
)
```

Id	*Name*	*Address*	*Status*
101222333	John	10 Cedar St	Freshman
234567890	Mary	22 Main St	Sophomore

Student

Primary/Candidate Keys

```
CREATE TABLE Course (
   CrsCode CHAR(6),
   CrsName CHAR(20),
   DeptId CHAR(4),
   Descr CHAR(100),
   PRIMARY KEY (CrsCode),
   UNIQUE (DeptId, CrsName)   -- candidate key
)
```

Comments start with 2 dashes

Null

- **Problem**: Not all information might be known when row is inserted (e.g., *Grade* might be missing from Transcript)
- A column might not be applicable for a particular row (e.g., *MaidenName* if row describes a male)
- ***Solution***: Use place holder – **null**
  - Not a value of any domain (although called null value)
    - Indicates the absence of a value
  - Not allowed in certain situations
    - Primary keys and columns constrained by NOT NULL

Default Value

-Value to be assigned if attribute value in a row is not specified

```
CREATE TABLE Student (
    Id  INTEGER,
    Name  CHAR(20) NOT NULL,
    Address  CHAR(50),
    Status  CHAR(10) DEFAULT 'freshman',
    PRIMARY KEY (Id) )
```

Semantic Constraints in SQL

- Primary key and foreign key are examples of *structural* constraints
- **Semantic constraints**
  - Express the logic of the application at hand:
    - e.g., number of registered students ≤ maximum enrollment

Semantic Constraints (cont'd)

- Used for application dependent conditions
- *Example*: limit attribute values

```
CREATE TABLE Transcript (
   StudId  INTEGER,
   CrsCode  CHAR(6),
   Semester  CHAR(6),
   Grade  CHAR(1),
   CHECK (Grade IN ('A', 'B', 'C', 'D', 'F')),
   CHECK (StudId > 0 AND StudId < 1000000000) )
```

- Each row in table must satisfy condition

Semantic Constraints (cont'd)

- *Example*: relate values of attributes in different columns

```
CREATE TABLE Employee (
    Id   INTEGER,
    Name   CHAR(20),
    Salary   INTEGER,
    MngrSalary   INTEGER,
    CHECK ( MngrSalary > Salary) )
```

Constraints – Problems

- **Problem 1**: Empty table always satisfies all CHECK constraints (an idiosyncrasy of the SQL standard)

```
CREATE TABLE Employee (
    Id  INTEGER,
    Name  CHAR(20),
    Salary  INTEGER,
    MngrSalary  INTEGER,
    CHECK ( 0 < (SELECT COUNT (*) FROM Employee)) )
```

  - If Employee is empty, there are no rows on which to evaluate the CHECK condition.

Constraints – Problems

- **Problem 2**: Inter-relational constraints should be symmetric

```
CREATE TABLE Employee (
   Id  INTEGER,
   Name  CHAR(20),
   Salary  INTEGER,
   MngrSalary  INTEGER,
    CHECK ((SELECT  COUNT (*) FROM Manager) <
                 (SELECT  COUNT (*) FROM Employee)) )
```

  - Why should constraint be in Employee an not Manager?
  - What if Employee is empty?

Assertion

- Element of schema (like table)
- Symmetrically specifies an inter-relational constraint
- Applies to entire database (not just the individual rows of a single table)
    - hence it works even if Employee is empty

```
CREATE ASSERTION DontFireEveryone
  CHECK (0 < SELECT  COUNT (*)  FROM Employee)
```

Assertion

```
CREATE ASSERTION KeepEmployeeSalariesDown
    CHECK (NOT EXISTS(
                SELECT * FROM Employee E
                WHERE E.Salary > E.MngrSalary))
```

Assertions and Inclusion Dependency

```
CREATE ASSERTION  NoEmptyCourses
   CHECK  (NOT  EXISTS (
                  SELECT *  FROM Teaching T
                   WHERE   -- for each row T check
                            -- the following condition
                      NOT  EXISTS (
                      SELECT * FROM Transcript  R
                      WHERE T.CrsCode = R.CrsCode
                        AND T.Semester = R.Semester)
                ))
```

Courses with no students

Students in a particular course

Domains

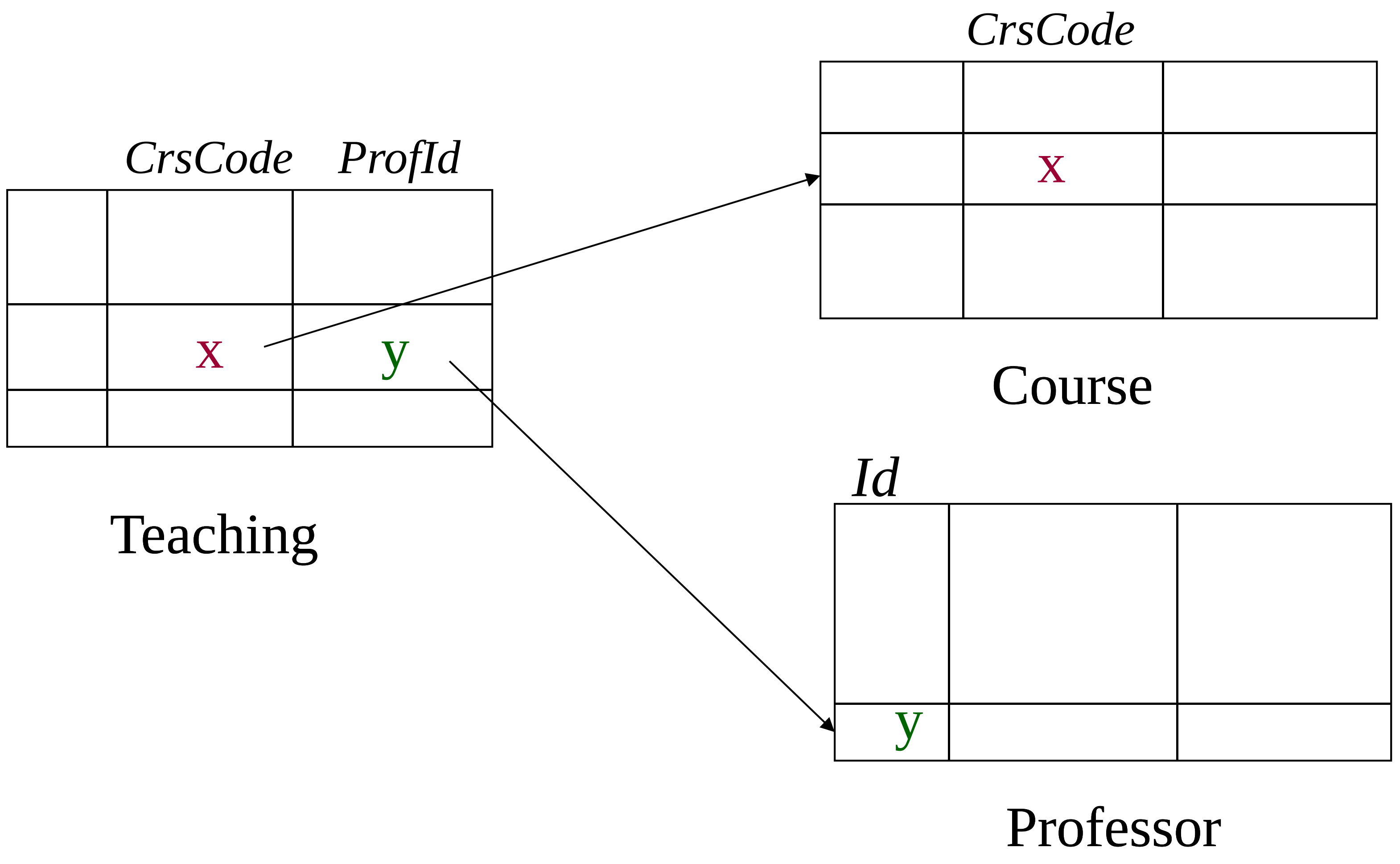
- Possible attribute values can be specified
  - Using a CHECK constraint or
  - Creating a new domain
- Domain can be used in several declarations
- Domain is a schema element

```
CREATE DOMAIN Grades CHAR (1)
   CHECK  (VALUE IN (‘A’, ‘B’, ‘C’, ‘D’, ‘F’))
CREATE TABLE Transcript (
    ….,
    Grade: Grades,
    …  )
```

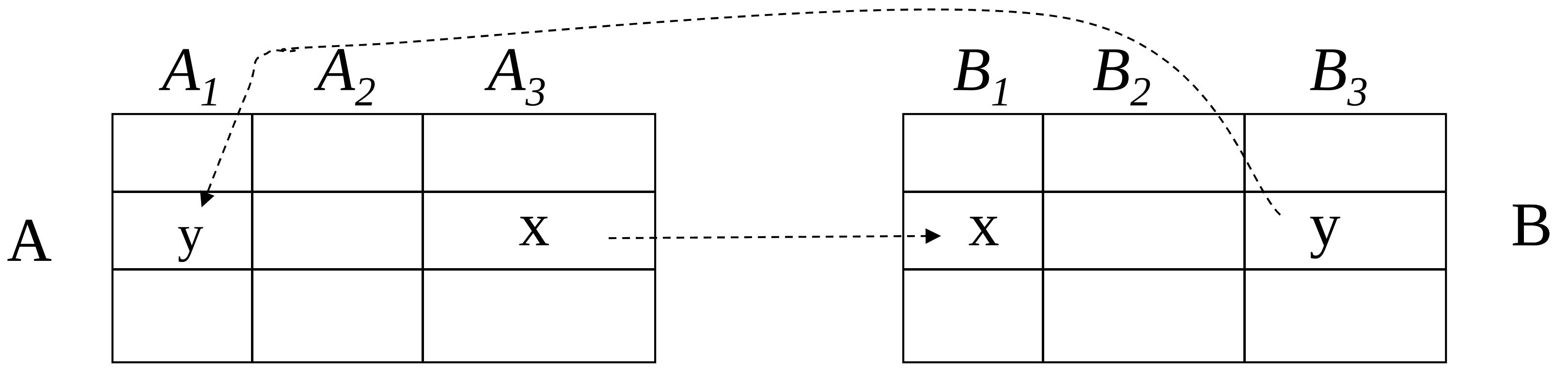
Foreign Key Constraint

```
CREATE TABLE Teaching (
    ProfId   INTEGER,
    CrsCode   CHAR (6),
    Semester   CHAR (6),
    PRIMARY KEY (CrsCode, Semester),
    FOREIGN KEY (CrsCode) REFERENCES Course,
    FOREIGN KEY (ProfId) REFERENCES Professor (Id) )
```

Foreign Key Constraint



Circularity in Foreign Key Constraint



candidate key: A_1
foreign key: A_3 references B(B_1)

candidate key: B_1
foreign key: B_3 references A(A_1)

Problem 1: Creation of A requires existence of B and vice versa

Solution:

```
CREATE TABLE  A ( ……)    -- no foreign key
CREATE TABLE  B ( ……)    -- include foreign key
ALTER TABLE  A
   ADD CONSTRAINT cons
      FOREIGN KEY (A3) REFERENCES B (B1)
```

Circularity in Foreign Key Constraint (cont'd)

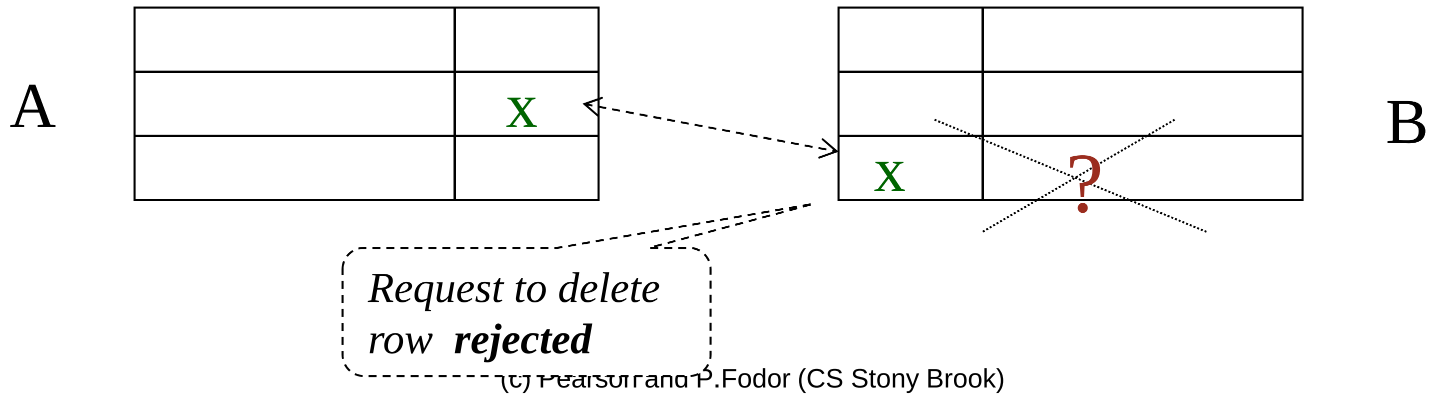
- Problem 2: Insertion of row in A requires prior existence of row in B and vice versa
- Solution: use appropriate ***constraint checking mode***:
  - `IMMEDIATE` checking
  - `DEFERRED` checking

Reactive Constraints

- Constraints enable DBMS to recognize a bad state and reject the statement or transaction that creates it
- More generally, it would be nice to have a mechanism that allows a user to specify how to *react* to a violation of a constraint
- SQL-92 provides a limited form of such a reactive mechanism for foreign key violations

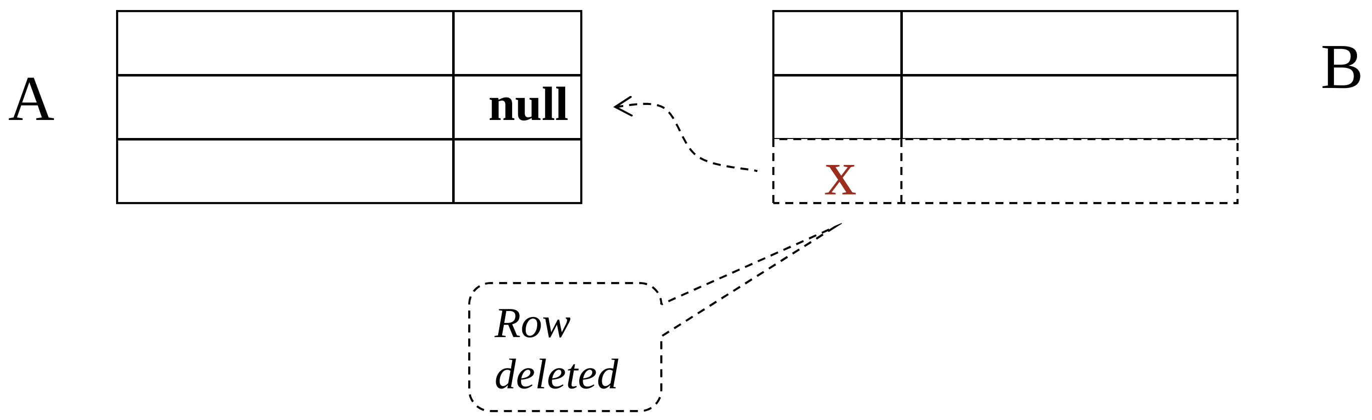
Handling Foreign Key Violations

- Insertion into A: Reject if no row exists in B containing foreign key of inserted row
- Deletion from B:
  - NO ACTION: Reject if row(s) in A references row to be deleted (default response)



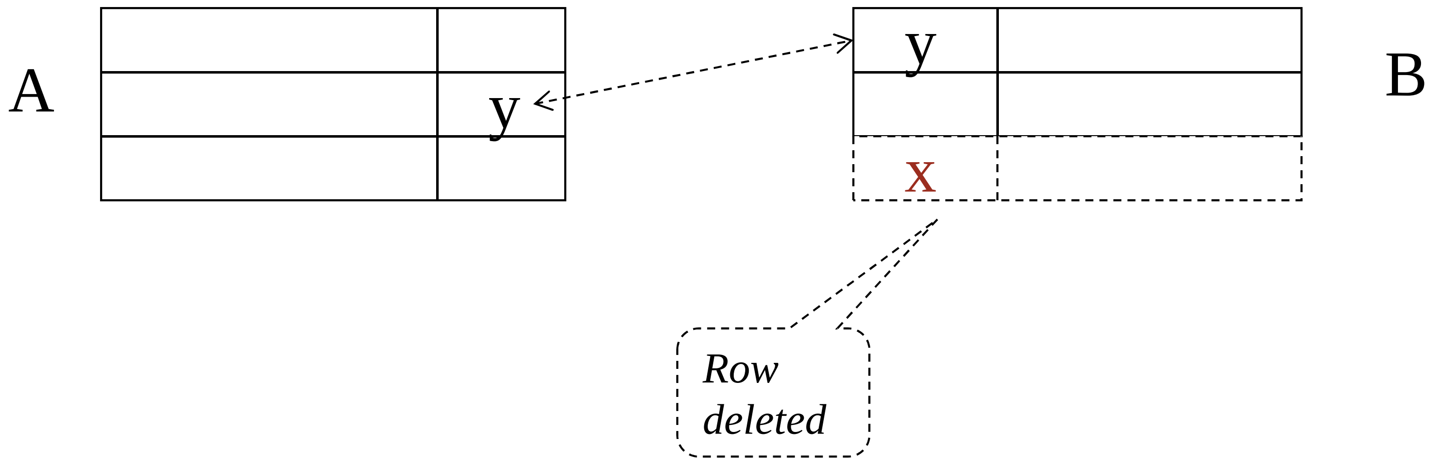
Handling Foreign Key Violations (cont'd)

- Deletion from B (cont'd):
  - SET NULL: Set value of foreign key in referencing row(s) in A to **null**



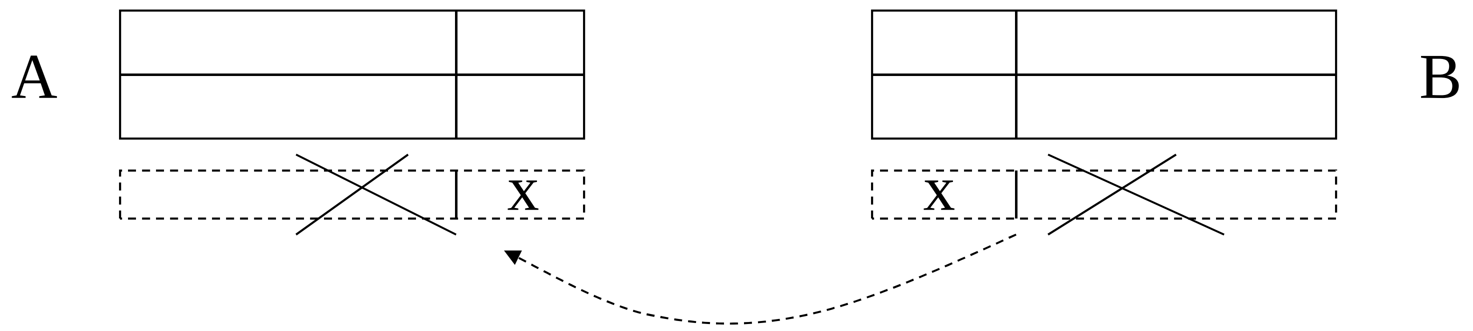
Handling Foreign Key Violations (cont'd)

- Deletion from B (cont'd):
  - SET DEFAULT: Set value of foreign key in referencing row(s) in A to default value (y) which must exist in B



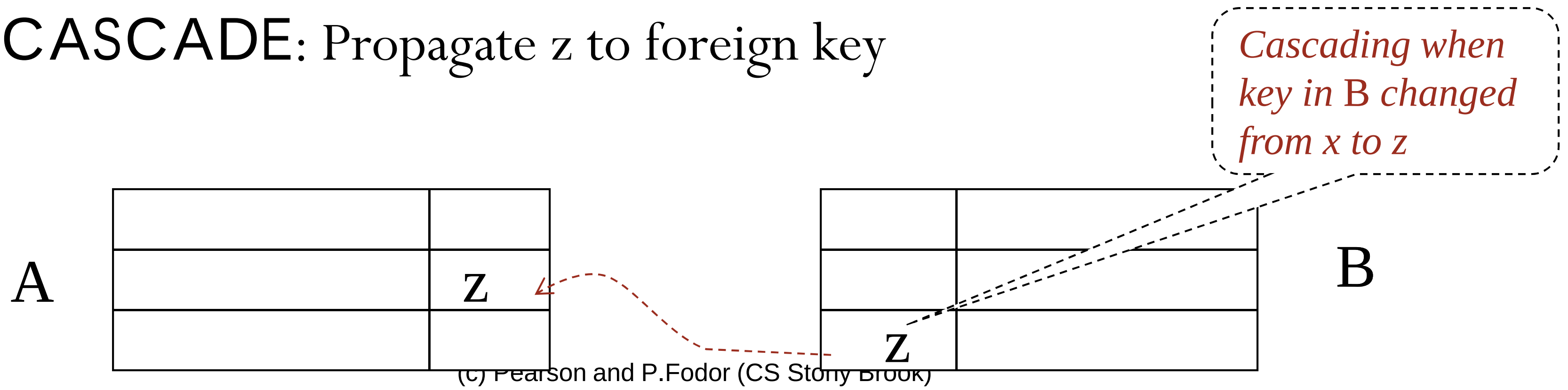
Handling Foreign Key Violations (cont'd)

- Deletion from B (cont'd):
  - CASCADE: Delete referencing row(s) in A as well



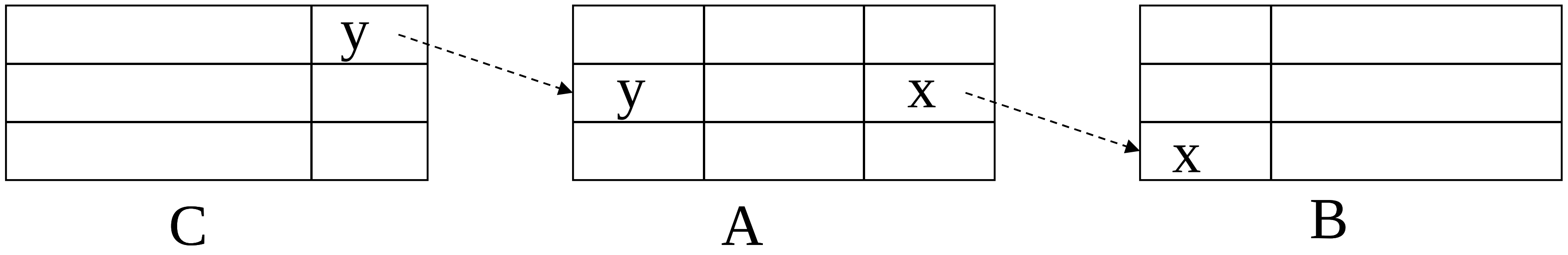
Handling Foreign Key Violations (cont'd)

- Update (change) foreign key in A: Reject if no row exists in B containing new foreign key
- Update candidate key in B (to z) – same actions as with deletion:
  - NO ACTION: Reject if row(s) in A references row to be updated (default response)
  - SET NULL: Set value of foreign key to **null**
  - SET DEFAULT: Set value of foreign key to default
  - CASCADE: Propagate z to foreign key



Handling Foreign Key Violations (cont'd)

- The action taken to repair the violation of a foreign key constraint in A may cause a violation of a foreign key constraint in C
  - The action specified in C controls how that violation is handled;
  - If the entire chain of violations cannot be resolved, the initial deletion from B is rejected.



Specifying Actions

```
CREATE TABLE Teaching (
   ProfId    INTEGER,
   CrsCode CHAR (6),
   Semester CHAR (6),
   PRIMARY KEY (CrsCode, Semester),

   FOREIGN KEY (ProfId) REFERENCES Professor (Id)
      ON DELETE NO ACTION
       ON UPDATE CASCADE,

   FOREIGN KEY (CrsCode) REFERENCES Course (CrsCode)
      ON DELETE SET NULL
       ON UPDATE CASCADE )
```

Triggers

- A more general mechanism for handling events
  - Not in SQL-92, but is in SQL:1999
- Trigger is a schema element (like table, assertion, …)

```
CREATE TRIGGER CrsChange
    AFTER UPDATE OF CrsCode, Semester ON Transcript
    WHEN  (Grade IS NOT NULL)
        ROLLBACK
```

Views

- Schema element
- Part of external schema
- A *virtual* table constructed from actual tables on the fly
  - Can be accessed in queries like any other table
  - Not materialized, constructed when accessed
  - Similar to a subroutine in ordinary programming

Views - Examples

Part of external schema suitable for use in Bursar's office:

```
CREATE VIEW CoursesTaken (StudId, CrsCode, Semester) AS
   SELECT  T.StudId, T.CrsCode, T.Semester
   FROM Transcript T
```

Part of external schema suitable for student with Id 123456789:

```
CREATE VIEW CoursesITook (CrsCode, Semester, Grade) AS
    SELECT  T.CrsCode, T.Semester, T.Grade
    FROM  Transcript T
    WHERE  T.StudId = '123456789'
```

Modifying the Schema

```
ALTER TABLE Student
   ADD COLUMN  Gpa  INTEGER DEFAULT 0

ALTER TABLE Student
   ADD CONSTRAINT GpaRange
      CHECK (Gpa >= 0 AND Gpa <= 4)

ALTER TABLE  Transcript
   DROP CONSTRAINT Cons     -- constraint names are useful

DROP TABLE  Employee

DROP ASSERTION  DontFireEveryone
```

Access Control

- Databases might contain sensitive information
- Access has to be limited:
  - Users have to be identified – ***authentication***
    - Generally done with passwords
  - Each user must be limited to modes of access appropriate to that user - ***authorization***
- SQL:92 provides tools for specifying an authorization policy but does not support authentication (vendor specific)

Controlling Authorization in SQL

GRANT *access_list*
  ON *table*
  TO *user_list*

access modes: SELECT, INSERT, DELETE, UPDATE, REFERENCES

User name

GRANT UPDATE (*Grade*) ON Transcript TO prof_smith
- Only the *Grade* column can be updated by prof_smith

GRANT SELECT ON Transcript TO joe
- Individual columns cannot be specified for SELECT access (in the SQL standard) – all columns of Transcript can be read
- *But* SELECT access control to individual columns can be *simulated* through views (next)

Controlling Authorization in SQL Using Views

GRANT *access*
ON *view*
TO *user_list*

GRANT SELECT ON CoursesTaken TO joe

– Thus views can be used to simulate access control to individual columns of a table

Authorization Mode REFERENCES

- Foreign key constraint enforces relationship between tables that can be exploited to
  - *Control access*: can enable perpetrator prevent deletion of rows

```
CREATE TABLE DontDismissMe (
   Id INTEGER,
   FOREIGN KEY (Id) REFERENCES Student
      ON DELETE NO ACTION )
```

  - *Reveal information*: successful insertion into DontDissmissMe means a row with foreign key value exists in Student

```
INSERT INTO DontDismissMe ('111111111')
```

REFERENCE Access mode (cont'd)

```
GRANT REFERENCES
ON Student
TO joe
```