



Database Management Systems

Module 11: Advanced SQL

Partha Pratim Das

*Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur*

ppd@cse.iitkgp.ernet.in

**Srijoni Majumdar
Himadri B G S Bhuyan
Gurunath Reddy M**



Database System Concepts, 6th Ed.

©Silberschatz, Korth and Sudarshan
www.db-book.com

Slides marked with 'PPD' are new or edited



Week 02 Recap

- **Module 06: Introduction to SQL/1**
 - History of SQL
 - Data Definition Language (DDL)
 - Basic Query Structure (DML)
- **Module 07: Introduction to SQL/2**
 - Additional Basic Operations
 - Set Operations
 - Null Values
 - Aggregate Functions
- **Module 08: Introduction to SQL/3**
 - Nested Subqueries
 - Modification of the Database
- **Module 09: Intermediate SQL/1**
 - Join Expressions
 - Views
 - Transactions
- **Module 10: Intermediate SQL/2**
 - Integrity Constraints
 - SQL Data Types and Schemas
 - Authorization



Module Objectives

- To understand how to use SQL from a programming language
- To familiarize with functions and procedures in SQL
- To understand the triggers



Module Outline

- Accessing SQL From a Programming Language
- Functions and Procedural Constructs
- Triggers



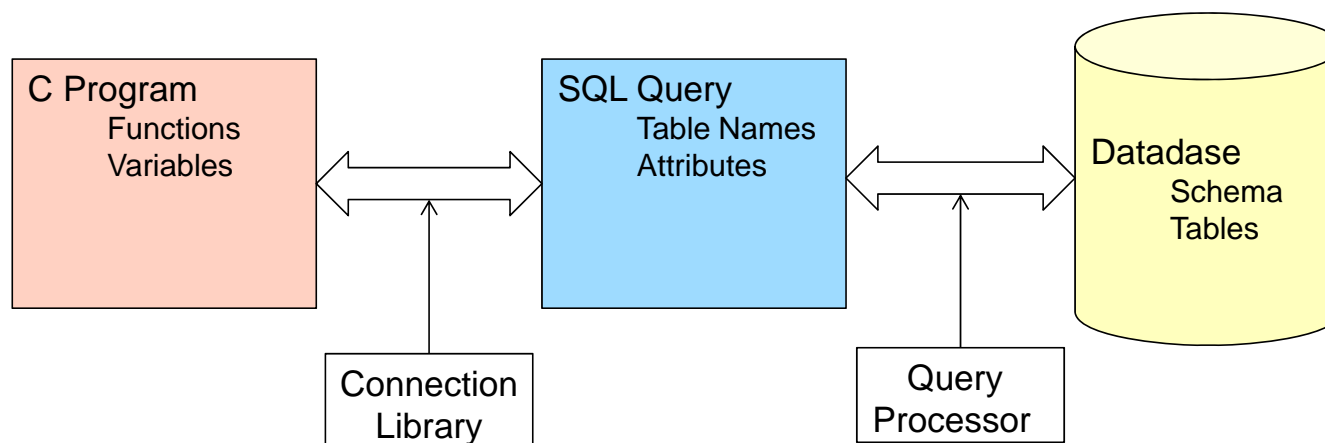
PPD

- **Accessing SQL From a Programming Language**
- Functions and Procedural Constructs
- Triggers

ACCESSING SQL FROM A PROGRAMMING LANGUAGE



Native Language \leftrightarrow Query Language





Accessing SQL From a Programming Language

- API (application-program interface) for a program to interact with a database server
- Application makes calls to
 - Connect with the database server
 - Send SQL commands to the database server
 - Fetch tuples of result one-by-one into program variables
- Various tools:
 - JDBC (Java Database Connectivity) works with Java
 - ODBC (Open Database Connectivity) works with C, C++, C#, Visual Basic, and Python
 - Other API's such as ADO.NET sit on top of ODBC
 - Embedded SQL



JDBC

- **JDBC** is a Java API for communicating with database systems supporting SQL
- JDBC supports a variety of features for querying and updating data, and for retrieving query results.
- JDBC also supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes
- Model for communicating with the database:
 - Open a connection
 - Create a “statement” object
 - Execute queries using the Statement object to send queries and fetch results
 - Exception mechanism to handle errors



ODBC

- Open DataBase Connectivity (ODBC) standard
 - standard for application program to communicate with a database server
 - application program interface (API) to
 - open a connection with a database,
 - send queries and updates,
 - get back results
- Applications such as GUI, spreadsheets, etc. can use ODBC



ODBC – Python Example

- The code uses a data source named “SQLS” from the odbc.ini file to connect and issue a query.
- It creates a table, inserts data using literal and parameterized statements and fetches the data

```
import pyodbc

conn = pyodbc.connect('DSN=SQLS;UID=test01;PWD=test01')
cursor=conn.cursor()
cursor.execute("create table rvtest (col1 int, col2 float, col3 varchar(10))")
cursor.execute("insert into rvtest values(1, 10.0, \"ABC\")")
cursor.execute("select * from rvtest")

while True:
    row=cursor.fetchone()
    if not row:
        break
    print(row)

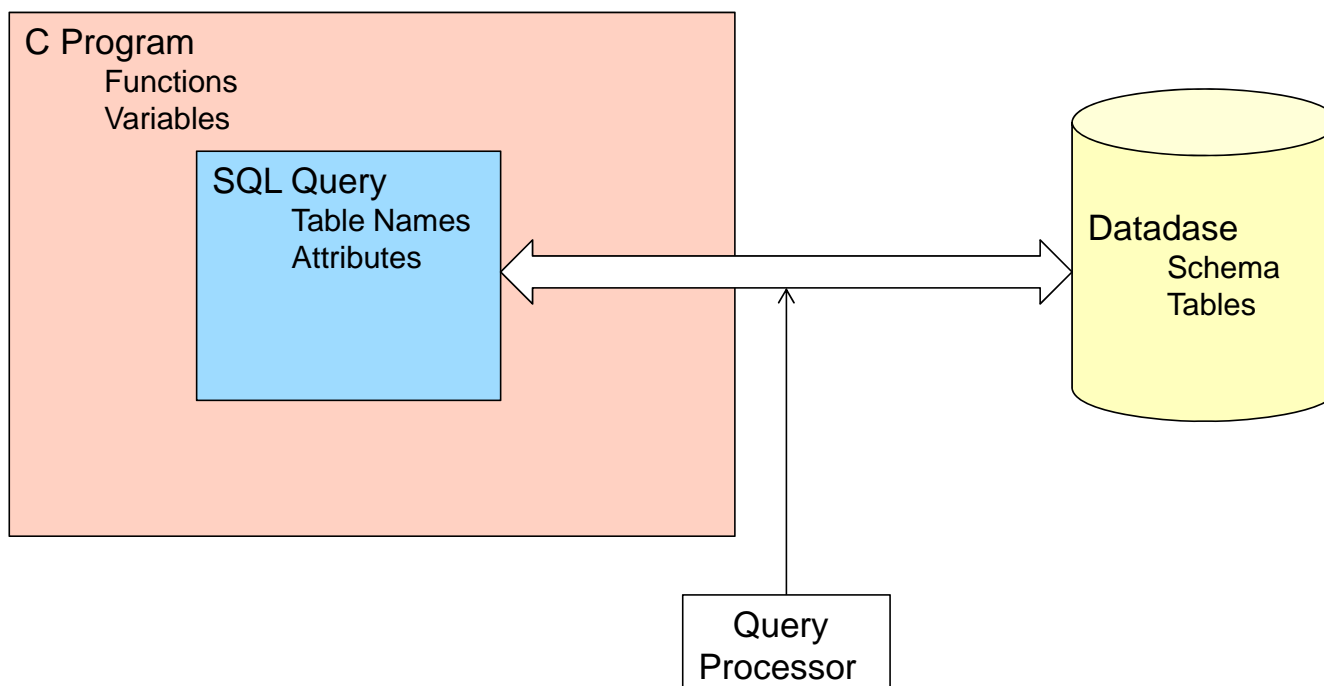
cursor.execute("delete from rvtest")
cursor.execute("insert into rvtest values (?, ?, ?)", 2, 20.0, 'XYZ')
cursor.execute("select * from rvtest")

while True:
    row=cursor.fetchone()
    if not row:
        break
    print(row)
```

Source: <https://dzone.com/articles/tutorial-connecting-to-odbc-data-sources-with-pyth>



Native Language \leftrightarrow Query Language





Embedded SQL

- The SQL standard defines embeddings of SQL in a variety of programming languages such as C, C++, Java, Fortran, and PL/1
- A language to which SQL queries are embedded is referred to as a **host language**, and the SQL structures permitted in the host language comprise *embedded SQL*
- The basic form of these languages follows that of the System R embedding of SQL into PL/1
- **EXEC SQL** statement is used to identify embedded SQL request to the preprocessor

EXEC SQL <embedded SQL statement >;

Note: this varies by language:

- In some languages, like COBOL, the semicolon is replaced with END-EXEC
- In Java embedding uses `# SQL { };`



Embedded SQL (Cont.)

- Before executing any SQL statements, the program must first connect to the database. This is done using:

EXEC-SQL **connect to** *server* **user** *user-name* **using** *password*;

Here, *server* identifies the server to which a connection is to be established

- Variables of the host language can be used within embedded SQL statements. They are preceded by a colon (:) to distinguish from SQL variables (e.g., *:credit_amount*)
- Variables used as above must be declared within DECLARE section, as illustrated below. The syntax for declaring the variables, however, follows the usual host language syntax

EXEC-SQL BEGIN DECLARE SECTION

int *credit-amount*;

EXEC-SQL END DECLARE SECTION;



Embedded SQL (Cont.)

- To write an embedded SQL query, we use the
declare c cursor for <SQL query>
statement. The variable *c* is used to identify the query
- Example:
 - From within a host language, find the ID and name of students who have completed more than the number of credits stored in variable **credit_amount** in the host language
 - Specify the query in SQL as follows:

EXEC SQL

```
declare c cursor for  
select ID, name  
from student  
where tot_cred > :credit_amount
```

END_EXEC



Embedded SQL (Cont.)

- Example:
 - From within a host language, find the ID and name of students who have completed more than the number of credits stored in variable `credit_amount` in the host language
- Specify the query in SQL as follows:

EXEC SQL

```
declare c cursor for  
select ID, name  
from student  
where tot_cred > :credit_amount
```

END_EXEC

- The variable `c` (used in the cursor declaration) is used to identify the query



Embedded SQL (Cont.)

- The **open** statement for our example is as follows:

```
EXEC SQL open c ;
```

This statement causes the database system to execute the query and to save the results within a temporary relation. The query uses the value of the host-language variable *credit-amount* at the time the **open** statement is executed.

- The fetch statement causes the values of one tuple in the query result to be placed on host language variables.

```
EXEC SQL fetch c into :si, :sn END_EXEC
```

Repeated calls to fetch get successive tuples in the query result



Embedded SQL (Cont.)

- A variable called SQLSTATE in the SQL communication area (SQLCA) gets set to '02000' to indicate no more data is available
- The **close** statement causes the database system to delete the temporary relation that holds the result of the query.

EXEC SQL close c ;

Note: above details vary with language. For example, the Java embedding defines Java iterators to step through result tuples.



Embedded SQL – C Example

- The program prompts the user for an order number, retrieves the customer number, salesperson, and status of the order, and displays the retrieved information on the screen

```
int main() {
    EXEC SQL INCLUDE SQLCA;
    EXEC SQL BEGIN DECLARE SECTION;
        int OrderID;          /* Employee ID (from user)      */
        int CustID;           /* Retrieved customer ID   */
        char SalesPerson[10]  /* Retrieved salesperson name */
        char Status[6]        /* Retrieved order status   */
    EXEC SQL END DECLARE SECTION;

    /* Set up error processing */
    EXEC SQL WHENEVER SQLERROR GOTO query_error;
    EXEC SQL WHENEVER NOT FOUND GOTO bad_number;

    /* Prompt the user for order number */
    printf ("Enter order number: ");
    scanf_s("%d", &OrderID);
```

```
    /* Execute the SQL query */
    EXEC SQL SELECT CustID, SalesPerson, Status
        FROM Orders
        WHERE OrderID = :OrderID
        INTO :CustID, :SalesPerson, :Status;

    /* Display the results */
    printf ("Customer number: %d\n", CustID);
    printf ("Salesperson: %s\n", SalesPerson);
    printf ("Status: %s\n", Status);
    exit();

query_error:
    printf ("SQL error: %ld\n", sqlca->sqlcode);
    exit();

bad_number:
    printf ("Invalid order number.\n");
    exit();
}
```

- The statement used to return the data is a singleton SELECT statement; that is, it returns only a single row of data. Therefore, the code example does not declare or use cursors

Source: <https://docs.microsoft.com/en-us/sql/odbc/reference/embedded-sql-example>



Updates Through Embedded SQL

- Embedded SQL expressions for database modification (**update**, **insert**, and **delete**)
- Can update tuples fetched by cursor by declaring that the cursor is for update

EXEC SQL

```
declare c cursor for  
select *  
from instructor  
where dept_name = 'Music'  
for update
```

- We then iterate through the tuples by performing **fetch** operations on the cursor (as illustrated earlier), and after fetching each tuple we execute the following code:

```
update instructor  
set salary = salary + 1000  
where current of c
```



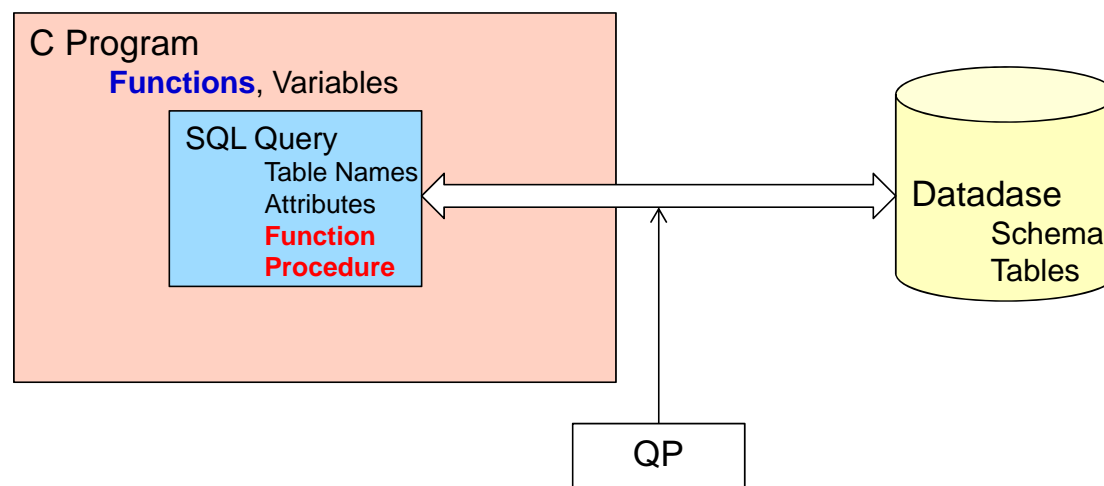
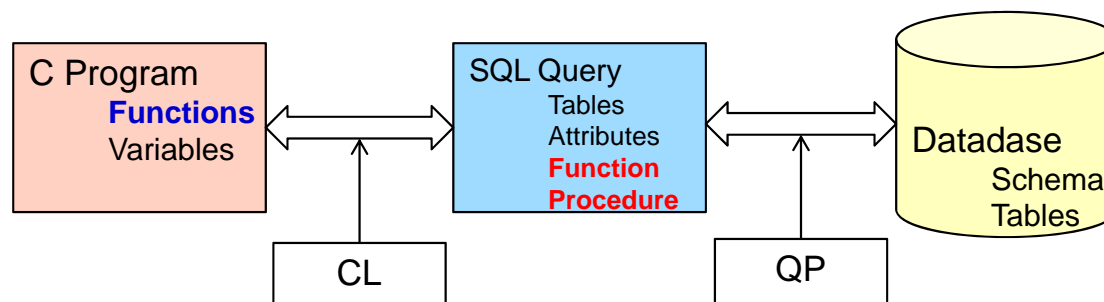
PPD

- Accessing SQL From a Programming Language
- **Functions and Procedural Constructs**
- Triggers

FUNCTIONS AND PROCEDURAL CONSTRUCTS



Native Language \leftrightarrow Query Language





Functions and Procedures

- SQL:1999 supports functions and procedures
 - Functions/procedures can be written in SQL itself, or in an external programming language (e.g., C, Java)
 - Functions written in an external languages are particularly useful with specialized data types such as images and geometric objects.
 - Example: functions to check if polygons overlap, or to compare images for similarity
 - Some database systems support **table-valued functions**, which can return a relation as a result
- SQL:1999 also supports a rich set of imperative constructs, including
 - Loops, if-then-else, assignment
- Many databases have proprietary procedural extensions to SQL that differ from SQL:1999



SQL Functions

- Define a function that, given the name of a department, returns the count of the number of instructors in that department.

```
create function dept_count (dept_name varchar(20))  
  returns integer  
  begin  
    declare d_count integer;  
    select count ( * ) into d_count  
    from instructor  
    where instructor.dept_name = dept_name  
    return d_count;  
end
```

- The function *dept_count* can be used to find the department names and budget of all departments with more than 12 instructors.

```
select dept_name, budget  
from department  
where dept_count (dept_name) > 12
```



SQL functions (Cont.)

- Compound statement: **begin ... end**
 - May contain multiple SQL statements between **begin** and **end**.
- **returns** – indicates the variable-type that is returned (e.g., integer)
- **return** – specifies the values that are to be returned as result of invoking the function
- SQL function are in fact **parameterized views** that generalize the regular notion of views by allowing parameters



Table Functions

- SQL:2003 added functions that return a relation as a result
- Example: Return all instructors in a given department

```
create function instructor_of (dept_name char(20))  
    returns table (  
        ID varchar(5),  
        name varchar(20),  
        dept_name varchar(20),  
        salary numeric(8,2))  
  
    return table  
    (select ID, name, dept_name, salary  
     from instructor  
     where instructor.dept_name = instructor_of.dept_name)
```

- Usage

```
select *  
from table (instructor_of ('Music'))
```



SQL Procedures

- The *dept_count* function could instead be written as procedure:

```
create procedure dept_count_proc (  
                                in dept_name varchar(20),  
                                out d_count integer)  
  
begin  
    select count(*) into d_count  
    from instructor  
    where instructor.dept_name = dept_count_proc.dept_name  
  
end
```

- Procedures can be invoked either from an SQL procedure or from embedded SQL, using the **call** statement.

```
declare d_count integer;  
call dept_count_proc( 'Physics', d_count);
```

- Procedures and functions can be invoked also from dynamic SQL
- SQL:1999 allows more than one function/procedure of the same name (called name **overloading**), as long as the number of arguments differ, or at least the types of the arguments differ



Language Constructs for Procedures & Functions

- SQL supports constructs that gives it almost all the power of a general-purpose programming language.
 - Warning: most database systems implement their own variant of the standard syntax below.
- Compound statement: **begin ... end**,
 - May contain multiple SQL statements between **begin** and **end**.
 - Local variables can be declared within a compound statements
- **While** and **repeat** statements:

while *boolean expression* **do**
 sequence of statements ;
end while

repeat
 sequence of statements ;
until *boolean expression*
end repeat



Language Constructs (Cont.)

- **For loop**
 - Permits iteration over all results of a query
- Example: Find the budget of all departments

```
declare n integer default 0;  
for r as  
    select budget from department  
do  
    set n = n + r.budget  
end for
```



Language Constructs (Cont.)

- Conditional statements (**if-then-else**)
SQL:1999 also supports a **case** statement similar to C case statement
- Example procedure: registers student after ensuring classroom capacity is not exceeded
 - Returns 0 on success and -1 if capacity is exceeded
 - See book (page 177) for details
- Signaling of exception conditions, and declaring handlers for exceptions

```
declare out_of_classroom_seats condition
declare exit handler for out_of_classroom_seats
begin
...
.. signal out_of_classroom_seats
end
```

 - The handler here is **exit** -- causes enclosing **begin..end** to be exited
 - Other actions possible on exception



External Language Routines*

- SQL:1999 permits the use of functions and procedures written in other languages such as C or C++
- Declaring external language procedures and functions

```
create procedure dept_count_proc(in dept_name varchar(20),  
                                out count integer)
```

```
language C
```

```
external name ' /usr/avi/bin/dept_count_proc'
```

```
create function dept_count(dept_name varchar(20))
```

```
returns integer
```

```
language C
```

```
external name '/usr/avi/bin/dept_count'
```



External Language Routines (Contd.)*

- SQL:1999 allows the definition of procedures in an imperative programming language, (Java, C#, C or C++) which can be invoked from SQL queries.
- Functions defined in this fashion can be more efficient than functions defined in SQL, and computations that cannot be carried out in SQL can be executed by these functions.
- Declaring external language procedures and functions

```
create procedure dept_count_proc(in dept_name varchar(20),  
                                out count integer)
```

```
language C
```

```
external name ' /usr/avi/bin/dept_count_proc'
```

```
create function dept_count(dept_name varchar(20))
```

```
returns integer
```

```
language C
```

```
external name ' /usr/avi/bin/dept_count'
```



External Language Routines (Cont.)*

- Benefits of external language functions/procedures:
 - more efficient for many operations, and more expressive power
- Drawbacks
 - Code to implement function may need to be loaded into database system and executed in the database system's address space.
 - risk of accidental corruption of database structures
 - security risk, allowing users access to unauthorized data
 - There are alternatives, which give good security at the cost of potentially worse performance
 - Direct execution in the database system's space is used when efficiency is more important than security



Security with External Language Routines*

- To deal with security problems, we can do one of the following:
 - Use **sandbox** techniques
 - That is, use a safe language like Java, which cannot be used to access/damage other parts of the database code.
 - Run external language functions/procedures in a separate process, with no access to the database process' memory.
 - Parameters and results communicated via inter-process communication
- Both have performance overheads
- Many database systems support both above approaches as well as direct executing in database system address space.



- Accessing SQL From a Programming Language
- Functions and Procedural Constructs
- **Triggers**

TRIGGERS



Triggers

- A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database
- To design a trigger mechanism, we must:
 - Specify the conditions under which the trigger is to be executed.
 - Specify the actions to be taken when the trigger executes.
- Triggers introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases.
 - Syntax illustrated here may not work exactly on your database system; check the system manuals



Triggering Events and Actions in SQL

- Triggering event can be **insert**, **delete** or **update**
- Triggers on update can be restricted to specific attributes
 - For example, **after update of *takes* on *grade***
- Values of attributes before and after an update can be referenced
 - **referencing old row as** : for deletes and updates
 - **referencing new row as** : for inserts and updates
- Triggers can be activated before an event, which can serve as extra constraints. For example, convert blank grades to null.

```
create trigger setnull_trigger before update of takes  
referencing new row as nrow  
for each row  
when (nrow.grade = ' ')  
begin atomic  
    set nrow.grade = null;  
end;
```



Trigger to Maintain `credits_earned` value

```
create trigger credits_earned after update of takes on (grade)  
referencing new row as nrow  
referencing old row as orow  
for each row  
when nrow.grade <> 'F' and nrow.grade is not null  
    and (orow.grade = 'F' or orow.grade is null)  
begin atomic  
    update student  
    set tot_cred = tot_cred +  
        (select credits  
         from course  
         where course.course_id = nrow.course_id)  
    where student.id = nrow.id;  
end;
```



Statement Level Triggers

- Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a transaction
 - Use **for each statement** instead of **for each row**
 - Use **referencing old table** or **referencing new table** to refer to temporary tables (called *transition tables*) containing the affected rows
 - Can be more efficient when dealing with SQL statements that update a large number of rows



When Not To Use Triggers

- Triggers were used earlier for tasks such as
 - Maintaining summary data (e.g., total salary of each department)
 - Replicating databases by recording changes to special relations (called **change** or **delta** relations) and having a separate process that applies the changes over to a replica
- There are better ways of doing these now:
 - Databases today provide built in materialized view facilities to maintain summary data
 - Databases provide built-in support for replication
- Encapsulation facilities can be used instead of triggers in many cases
 - Define methods to update fields
 - Carry out actions as part of the update methods instead of through a trigger



When Not To Use Triggers (Cont.)

- Risk of unintended execution of triggers, for example, when
 - Loading data from a backup copy
 - Replicating updates at a remote site
 - Trigger execution can be disabled before such actions.
- Other risks with triggers:
 - Error leading to failure of critical transactions that set off the trigger
 - Cascading execution



Module Summary

- Introduced the use of SQL from a programming language
- Familiarized with functions and procedures in SQL
- Understood the triggers



PPD

Instructor and TAs

Name	Mail	Mobile
Partha Pratim Das, Instructor	ppd@cse.iitkgp.ernet.in	9830030880
Srijoni Majumdar, TA	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, TA	himadribhuyan@gmail.com	9438911655
Gurunath Reddy M	mgurunathreddy@gmail.com	9434137638

Slides used in this presentation are borrowed from <http://db-book.com/> with kind permission of the authors.

Edited and new slides are marked with “PPD”.



Database Management Systems

Module 12: Formal Relational Query Languages

Partha Pratim Das

*Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur*

ppd@cse.iitkgp.ernet.in

**Srijoni Majumdar
Himadri B G S Bhuyan
Gurunath Reddy M**



Database System Concepts, 6th Ed.

©Silberschatz, Korth and Sudarshan
www.db-book.com



Module Recap

- Accessing SQL From a Programming Language
- Functions and Procedural Constructs
- Triggers



Module Objectives

- To understand formal query language through relational algebra



Module Outline

- Relational Algebra
- Tuple Relational Calculus (Overview only)
- Domain Relational Calculus (Overview only)
- Equivalence of Algebra and Calculus



Formal Relational Query Language

- Relational Algebra
 - Procedural and Algebra based
- Tuple Relational Calculus
 - Non-Procedural and Predicate Calculus based
- Domain Relational Calculus
 - Non-Procedural and Predicate Calculus based



PPD

- **Relational Algebra**
- Tuple Relational Calculus
- Domain Relational Calculus
- Equivalence of Algebra and Calculus

RELATIONAL ALGEBRA



Relational Algebra

- Created by Edgar F Codd at IBM in 1970
- Procedural language
- Six basic operators
 - select: σ
 - project: Π
 - union: \cup
 - set difference: $-$
 - Cartesian product: \times
 - rename: ρ
- The operators take one or two relations as inputs and produce a new relation as a result



Select Operation

PPD

- Notation: $\sigma_p(r)$
- p is called the **selection predicate**
- Defined as:

$$\sigma_p(r) = \{t \mid t \in r \text{ and } p(t)\}$$

where p is a formula in propositional calculus consisting of **terms** connected by : \wedge (**and**), \vee (**or**), \neg (**not**)

Each **term** is one of:

<attribute> op <attribute> or <constant>

where op is one of: $=, \neq, >, \geq, <, \leq$

- Example of selection:

$$\sigma_{dept_name="Physics"}(instructor)$$

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

A	B	C	D
α	α	1	7
β	β	23	10

$$\sigma_{A=B \wedge D > 5}(r)$$



Project Operation

- Notation:

$$\Pi_{A_1, A_2, \dots, A_k}(r)$$

where A_1, A_2 are attribute names and r is a relation name

- The result is defined as the relation of k columns obtained by erasing the columns that are not listed
- Duplicate rows removed from result, since relations are sets
- Example: To eliminate the *dept_name* attribute of *instructor*

$$\Pi_{ID, name, salary}(instructor)$$

A	B	C
α	10	1
α	20	1
β	30	1
β	40	2

A	C
α	1
α	1
β	1
β	2

=

A	C
α	1
β	1
β	2

$$\Pi_{A,C}(r)$$



Union Operation

- Notation: $r \cup s$

- Defined as:

$$r \cup s = \{t \mid t \in r \text{ or } t \in s\}$$

- For $r \cup s$ to be valid.

- r, s must have the **same arity** (same number of attributes)
- The attribute domains must be **compatible** (example: 2nd column of r deals with the same type of values as does the 2nd column of s)

- Example: to find all courses taught in the Fall 2009 semester, or in the Spring 2010 semester, or in both

$$\Pi_{course_id}(\sigma_{semester="Fall" \wedge year=2009}(section)) \cup$$

$$\Pi_{course_id}(\sigma_{semester="Spring" \wedge year=2010}(section))$$

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

A	B
α	1
α	2
β	1
β	3

$r \cup s$



Set Difference Operation

- Notation $r - s$
- Defined as:

$$r - s = \{t \mid t \in r \text{ and } t \notin s\}$$
- Set differences must be taken between **compatible** relations
 - r and s must have the **same** arity
 - attribute domains of r and s must be compatible
- Example: to find all courses taught in the Fall 2009 semester, but not in the Spring 2010 semester

$$\Pi_{\text{course_id}} (\sigma_{\text{semester}=\text{"Fall"} \wedge \text{year}=2009}(\text{section})) - \Pi_{\text{course_id}} (\sigma_{\text{semester}=\text{"Spring"} \wedge \text{year}=2010}(\text{section}))$$

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

A	B
α	1
β	1

$r - s$



Set-Intersection Operation

- Notation: $r \cap s$
- Defined as:

$$r \cap s = \{ t \mid t \in r \text{ and } t \in s \}$$
- Assume:
 - r, s have the *same arity*
 - attributes of r and s are compatible
- Note: $r \cap s = r - (r - s)$

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

A	B
α	2

$r \cap s$



Cartesian-Product Operation

- Notation $r \times s$
- Defined as:

$$r \times s = \{t \mid t \in r \text{ and } q \in s\}$$

A	B
α	1
β	2

r

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

s

- Assume that attributes of $r(R)$ and $s(S)$ are disjoint. (That is, $R \cap S = \emptyset$)
- If attributes of $r(R)$ and $s(S)$ are not disjoint, then renaming must be used

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

$r \times s$



Rename Operation

- Allows us to name, and therefore to refer to, the results of relational-algebra expressions.
- Allows us to refer to a relation by more than one name.
- Example:

$$\rho_X(E)$$

returns the expression E under the name X

- If a relational-algebra expression E has arity n , then

$$\rho_{X(A_1, A_2, \dots, A_n)}(E)$$

returns the result of expression E under the name X , and with the attributes renamed to A_1, A_2, \dots, A_n .



Division Operation

- The division operation is applied to two relations
- $R(Z) \div S(X)$, where X subset Z . Let $Y = Z - X$ (and hence $Z = X \cup Y$); that is, let Y be the set of attributes of R that are not attributes of S
- The result of DIVISION is a relation $T(Y)$ that includes a tuple t if tuples t_R appear in R with $t_R[Y] = t$, and with
 - $t_R[X] = t_s$ for every tuple t_s in S .
- For a tuple t to appear in the result T of the DIVISION, the values in t must appear in R in combination with every tuple in S
- Division is a derived operation and can be expressed in terms of other operations



Division Operation – Example

- Relations r, s :

A	B
α	1
α	2
α	3
β	1
γ	1
δ	1
δ	3
δ	4
ϵ	6
ϵ	1
β	2

r

B
1
2

s

- $r \div s$:

A
α
β

e.g.

A is customer name

B is branch-name

1 and 2 here show two specific branch-names

(Find customers who have an account in all branches of the bank)

Source: db.fcnngroup.nl/silberslides/Division%20-%20Slides%20-%20relational%20algebra.pptx



Another Division Example

- Relations r , s :

A	B	C	D	E
α	a	α	a	1
α	a	γ	a	1
α	a	γ	b	1
β	a	γ	a	1
β	a	γ	b	3
γ	a	γ	a	1
γ	a	γ	b	1
γ	a	β	b	1

r

D	E
a	1
b	1

s

- $r \div s$:

A	B	C
α	a	γ
γ	a	γ

e.g.

Students who have taken both "a" and "b" courses, with instructor "1"

(Find students who have taken all courses given by instructor 1)

Source: db.fcnngroup.nl/silberslides/Division%20-%20Slides%20-%20relational%20algebra.pptx



Formal Definition

- A basic expression in the relational algebra consists of either one of the following:
 - A relation in the database
 - A constant relation
- Let E_1 and E_2 be relational-algebra expressions; the following are all relational-algebra expressions:
 - $E_1 \cup E_2$
 - $E_1 - E_2$
 - $E_1 \times E_2$
 - $\sigma_P(E_1)$, P is a predicate on attributes in E_1
 - $\Pi_S(E_1)$, S is a list consisting of some of the attributes in E_1
 - $\rho_x(E_1)$, x is the new name for the result of E_1



PPD

- Relational Algebra
- **Tuple Relational Calculus**
- Domain Relational Calculus
- Equivalence of Algebra and Calculus

TUPLE RELATIONAL CALCULUS



Tuple Relational Calculus

- A nonprocedural query language, where each query is of the form
$$\{t \mid P(t)\}$$
- It is the set of all tuples t such that predicate P is true for t
- t is a *tuple variable*, $t[A]$ denotes the value of tuple t on attribute A
- $t \in r$ denotes that tuple t is in relation r
- P is a *formula* similar to that of the predicate calculus



Predicate Calculus Formula

1. Set of attributes and constants
2. Set of comparison operators: (e.g., $<$, \leq , $=$, \neq , $>$, \geq)
3. Set of connectives: and (\wedge), or (\vee), not (\neg)
4. Implication (\Rightarrow): $x \Rightarrow y$, if x is true, then y is true

$$x \Rightarrow y \equiv \neg x \vee y$$

5. Set of quantifiers:

- ▶ $\exists t \in r (Q(t)) \equiv$ "there exists" a tuple t in relation r such that predicate $Q(t)$ is true
- ▶ $\forall t \in r (Q(t)) \equiv Q$ is true "for all" tuples t in relation r



Safety of Expressions

- It is possible to write tuple calculus expressions that generate infinite relations
- For example, $\{ t \mid \neg t \in r \}$ results in an infinite relation if the domain of any attribute of relation r is infinite
- To guard against the problem, we restrict the set of allowable expressions to safe expressions
- An expression $\{ t \mid P(t) \}$ in the tuple relational calculus is *safe* if every component of t appears in one of the relations, tuples, or constants that appear in P
 - NOTE: this is more than just a syntax condition
 - E.g. $\{ t \mid t[A] = 5 \vee \mathbf{true} \}$ is not safe --- it defines an infinite set with attribute values that do not appear in any relation or tuples or constants in P



PPD

- Relational Algebra
- Tuple Relational Calculus
- **Domain Relational Calculus**
- Equivalence of Algebra and Calculus

DOMAIN RELATIONAL CALCULUS



Domain Relational Calculus

- A nonprocedural query language equivalent in power to the tuple relational calculus
- Each query is an expression of the form:

$$\{ \langle x_1, x_2, \dots, x_n \rangle \mid P(x_1, x_2, \dots, x_n) \}$$

- x_1, x_2, \dots, x_n represent domain variables
- P represents a formula similar to that of the predicate calculus

- Relational Algebra
- Tuple Relational Calculus
- Domain Relational Calculus
- **Equivalence of Algebra and Calculus**

EQUIVALENCE OF ALGEBRA AND CALCULUS



Equivalence of RA, TRC and DRC

Select Operation

$R = (A, B)$

Relational Algebra: $\sigma_{B=17}(r)$

Tuple Calculus: $\{t \mid t \in r \wedge B = 17\}$

Domain Calculus: $\{ \langle a, b \rangle \mid \langle a, b \rangle \in r \wedge b = 17 \}$

Source: http://www.cs.sfu.ca/CourseCentral/354/louie/Equiv_Notations.pdf



Equivalence of RA, TRC and DRC

Project Operation

$R = (A, B)$

Relational Algebra: $\Pi_A(r)$

Tuple Calculus: $\{t \mid \exists p \in r (t[A] = p[A])\}$

Domain Calculus: $\{ \langle a \rangle \mid \exists b (\langle a, b \rangle \in r) \}$

Source: http://www.cs.sfu.ca/CourseCentral/354/louie/Equiv_Notations.pdf



Equivalence of RA, TRC and DRC

Combining Operations

$R = (A, B)$

Relational Algebra: $\Pi_A(\sigma_{B=17}(r))$

Tuple Calculus: $\{t \mid \exists p \in r (t[A] = p[A] \wedge p[B] = 17)\}$

Domain Calculus: $\{ \langle a \rangle \mid \exists b (\langle a, b \rangle \in r \wedge b = 17) \}$

Source: http://www.cs.sfu.ca/CourseCentral/354/louie/Equiv_Notations.pdf



Equivalence of RA, TRC and DRC

Union

$R = (A, B, C) \quad S = (A, B, C)$

Relational Algebra: $r \cup s$

Tuple Calculus: $\{t \mid t \in r \vee t \in s\}$

Domain Calculus: $\{ \langle a, b, c \rangle \mid \langle a, b, c \rangle \in r \vee \langle a, b, c \rangle \in s \}$

Source: http://www.cs.sfu.ca/CourseCentral/354/louie/Equiv_Notations.pdf



Equivalence of RA, TRC and DRC

Set Difference

$R = (A, B, C) \quad S = (A, B, C)$

Relational Algebra: $r - s$

Tuple Calculus: $\{t \mid t \in r \wedge t \notin s\}$

Domain Calculus: $\{ \langle a, b, c \rangle \mid \langle a, b, c \rangle \in r \wedge \langle a, b, c \rangle \notin s \}$

Source: http://www.cs.sfu.ca/CourseCentral/354/louie/Equiv_Notations.pdf



Equivalence of RA, TRC and DRC

Intersection

$R = (A, B, C) \quad S = (A, B, C)$

Relational Algebra: $r \cap s$

Tuple Calculus: $\{t \mid t \in r \wedge t \in s\}$

Domain Calculus: $\{ \langle a, b, c \rangle \mid \langle a, b, c \rangle \in r \wedge \langle a, b, c \rangle \in s \}$

Source: http://www.cs.sfu.ca/CourseCentral/354/louie/Equiv_Notations.pdf



Equivalence of RA, TRC and DRC

Cartesian/Cross Product

$R = (A, B) \quad S = (C, D)$

Relational Algebra: $r \times s$

Tuple Calculus: $\{t \mid \exists p \in r \exists q \in s (t[A] = p[A] \wedge t[B] = p[B] \wedge t[C] = q[C] \wedge t[D] = q[D])\}$

Domain Calculus: $\{ \langle a, b, c, d \rangle \mid \langle a, b \rangle \in r \wedge \langle c, d \rangle \in s \}$

Source: http://www.cs.sfu.ca/CourseCentral/354/louie/Equiv_Notations.pdf



Equivalence of RA, TRC and DRC

Natural Join

$R = (A, B, C, D) \quad S = (B, D, E)$

Relational Algebra: $r \bowtie s$

$$\Pi_{r.A, r.B, r.C, r.D, s.E}(\sigma_{r.B=s.B \wedge r.D=s.D} (r \times s))$$

Tuple Calculus: $\{t \mid \exists p \in r \exists q \in s (t[A] = p[A] \wedge t[B] = p[B] \wedge t[C] = p[C] \wedge t[D] = p[D] \wedge t[E] = q[E] \wedge p[B] = q[B] \wedge p[D] = q[D])\}$

Domain Calculus: $\{ \langle a, b, c, d, e \rangle \mid \langle a, b, c, d \rangle \in r \wedge \langle b, d, e \rangle \in s \}$

Source: http://www.cs.sfu.ca/CourseCentral/354/louie/Equiv_Notations.pdf



Equivalence of RA, TRC and DRC

Division

$R = (A, B) \quad S = (B)$

Relational Algebra: $r \div s$

Tuple Calculus: $\{t \mid \exists p \in r \forall q \in s (p[B] = q[B] \Rightarrow t[A] = p[A])\}$

Domain Calculus: $\{\langle a \rangle \mid \langle a \rangle \in r \wedge \forall \langle b \rangle (\langle b \rangle \in s \Rightarrow \langle a, b \rangle \in r)\}$

Source: http://www.cs.sfu.ca/CourseCentral/354/louie/Equiv_Notations.pdf



Module Summary

- Discussed relational algebra with examples
- Introduced tuple relational and domain relational calculus
- Illustrated equivalence of algebra and calculus



PPD

Instructor and TAs

Name	Mail	Mobile
Partha Pratim Das, Instructor	ppd@cse.iitkgp.ernet.in	9830030880
Srijoni Majumdar, TA	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, TA	himadribhuyan@gmail.com	9438911655
Gurunath Reddy M	mgurunathreddy@gmail.com	9434137638

Slides used in this presentation are borrowed from <http://db-book.com/> with kind permission of the authors.

Edited and new slides are marked with “PPD”.



Database Management Systems

Module 13: Entity-Relationship Model/1

Partha Pratim Das

*Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur*

ppd@cse.iitkgp.ernet.in

**Srijoni Majumdar
Himadri B G S Bhuyan
Gurunath Reddy M**



Database System Concepts, 6th Ed.

©Silberschatz, Korth and Sudarshan
www.db-book.com



Module Recap

- Relational Algebra
- Tuple Relational Calculus (Overview only)
- Domain Relational Calculus (Overview only)
- Equivalence of Algebra and Calculus



Module Objectives

- To understand the Design Process for Database Systems
- To study the E-R Model for real world representation



Module Outline

- Design Process
- E-R Model
 - Entity and Entity Set
 - Relationship
 - Cardinality
 - Attributes
 - Weak Entity Sets



PPD

- **Design Process**
- E-R Model

DESIGN PROCESS



Design Phases

- The initial phase of database design is to characterize fully the data needs of the prospective database users
- Next, the designer chooses a data model and, by applying the concepts of the chosen data model, translates these requirements into a conceptual schema of the database
- A fully developed conceptual schema also indicates the functional requirements of the enterprise. In a “specification of functional requirements”, users describe the kinds of operations (or transactions) that will be performed on the data



Design Phases (Cont.)

The process of moving from an abstract data model to the implementation of the database proceeds in two final design phases.

- Logical Design – Deciding on the database schema. Database design requires that we find a “good” collection of relation schemas.
 - Business decision – What attributes should we record in the database?
 - Computer Science decision – What relation schemas should we have and how should the attributes be distributed among the various relation schemas?
- Physical Design – Deciding on the physical layout of the database



Design Approaches

- Entity Relationship Model (covered in this chapter)
 - Models an enterprise as a collection of *entities* and *relationships*
 - Entity: a “thing” or “object” in the enterprise that is distinguishable from other objects
 - Described by a set of *attributes*
 - Relationship: an association among several entities
 - Represented diagrammatically by an *entity-relationship diagram*:
- Normalization Theory (Chapter 8)
 - Formalize what designs are bad, and test for them



E-R MODEL

PPD

- Design Process
- **E-R Model**



ER model – Database Modeling

- The ER data model was developed to facilitate database design by allowing specification of an **enterprise schema** that represents the overall logical structure of a database
- The ER model is very useful in mapping the meanings and interactions of real-world enterprises onto a conceptual schema. Because of this usefulness, many database-design tools draw on concepts from the ER model
- The ER data model employs three basic concepts:
 - entity sets
 - relationship sets
 - attributes
- The ER model also has an associated diagrammatic representation, the ER diagram, which can express the overall logical structure of a database graphically



Entity Sets

- An **entity** is an object that exists and is distinguishable from other objects.
 - Example: specific person, company, event, plant
- An **entity set** is a set of entities of the same type that share the same properties.
 - Example: set of all persons, companies, trees, holidays
- An entity is represented by a set of attributes; i.e., descriptive properties possessed by all members of an entity set.
 - Example:
 $instructor = (ID, name, street, city, salary)$
 $course = (course_id, title, credits)$
- A subset of the attributes form a **primary key** of the entity set; i.e., uniquely identifying each member of the set.



Entity Sets – *instructor* and *student*

instructor_ID instructor_name

76766	Crick
45565	Katz
10101	Srinivasan
98345	Kim
76543	Singh
22222	Einstein

instructor

student-ID student_name

98988	Tanaka
12345	Shankar
00128	Zhang
76543	Brown
76653	Aoi
23121	Chavez
44553	Peltier

student



Relationship Sets

- A **relationship** is an association among several entities

Example:

44553 (Peltier)	<u>advisor</u>	22222 (<u>Einstein</u>)
<i>student</i> entity	relationship set	<i>instructor</i> entity

- A **relationship set** is a mathematical relation among $n \geq 2$ entities, each taken from entity sets

$$\{(e_1, e_2, \dots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \dots, e_n \in E_n\}$$

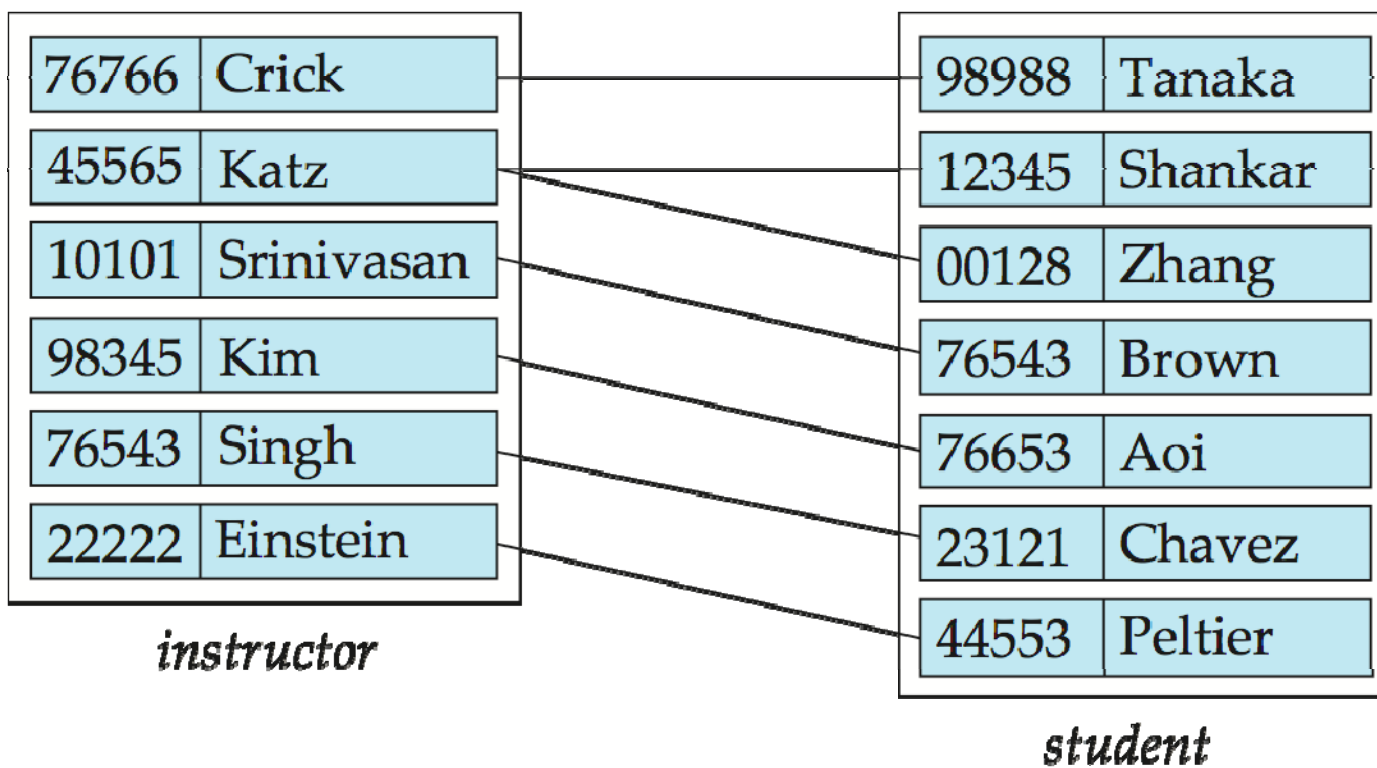
where (e_1, e_2, \dots, e_n) is a relationship

- Example:

$(44553, 22222) \in \text{advisor}$



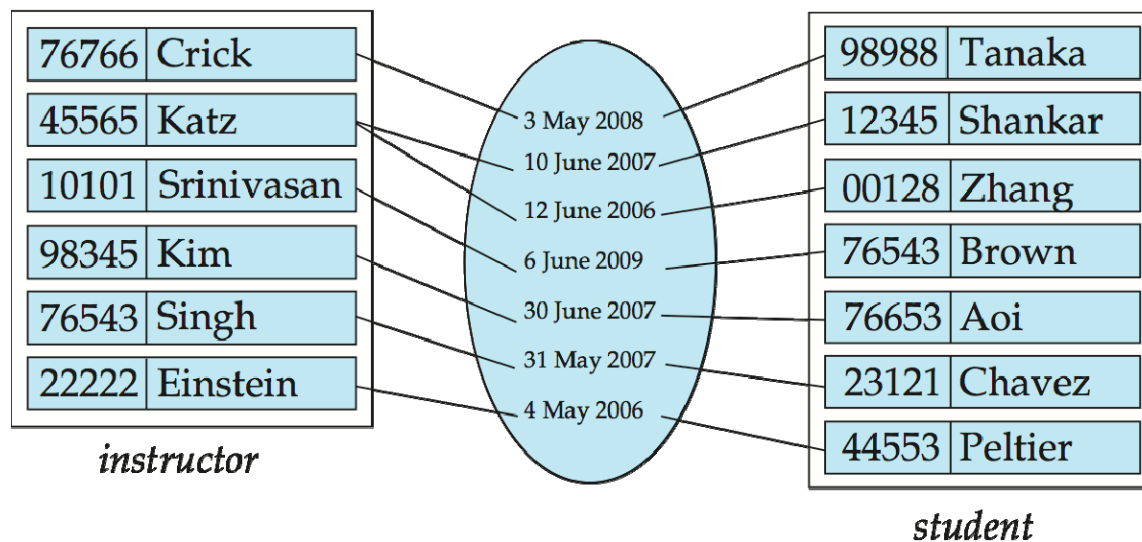
Relationship Set *advisor*





Relationship Sets (Cont.)

- An attribute can also be associated with a relationship set.
- For instance, the *advisor* relationship set between entity sets *instructor* and *student* may have the attribute *date* which tracks when the student started being associated with the advisor





Degree of a Relationship Set

- Binary relationship
 - involve two entity sets (or degree two).
 - most relationship sets in a database system are binary.
- Relationships between more than two entity sets are rare. Most relationships are binary. (More on this later.)
 - Example: *students* work on research *projects* under the guidance of an *instructor*.
 - relationship *proj_guide* is a ternary relationship between *instructor*, *student*, and *project*

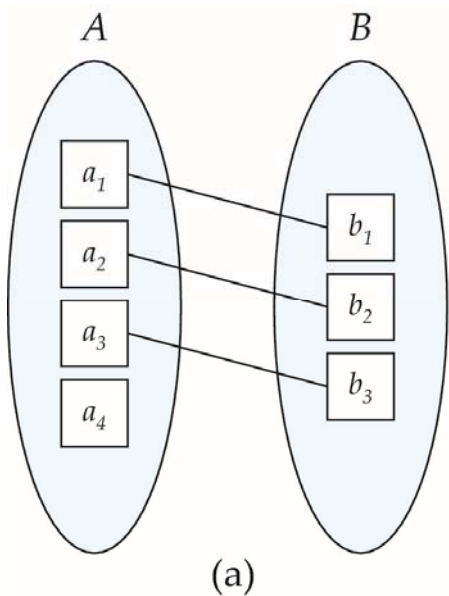


Mapping Cardinality Constraints

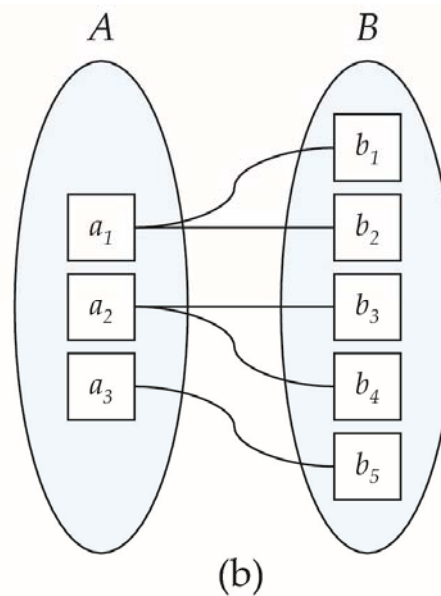
- Express the number of entities to which another entity can be associated via a relationship set.
- Most useful in describing binary relationship sets.
- For a binary relationship set the mapping cardinality must be one of the following types:
 - One to one
 - One to many
 - Many to one
 - Many to many



Mapping Cardinalities



One to one

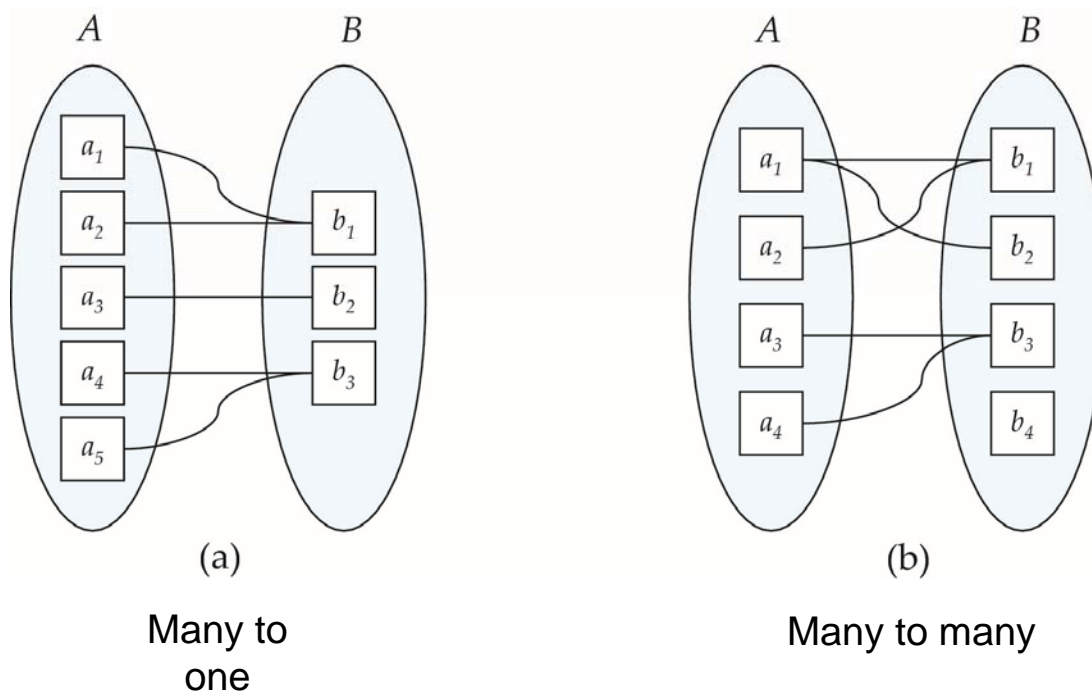


One to many

Note: Some elements in A and B may not be mapped to any elements in the other set



Mapping Cardinalities



Note: Some elements in A and B may not be mapped to any elements in the other set



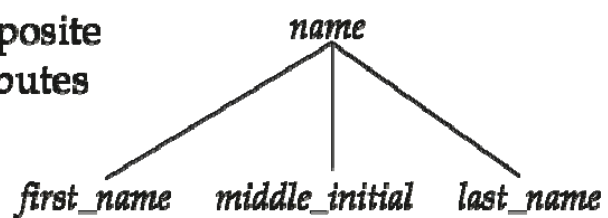
Complex Attributes

- Attribute types:
 - **Simple** and **composite** attributes.
 - **Single-valued** and **multivalued** attributes
 - Example: multivalued attribute: *phone_numbers*
 - **Derived** attributes
 - Can be computed from other attributes
 - Example: age, given date_of_birth
- **Domain** – the set of permitted values for each attribute

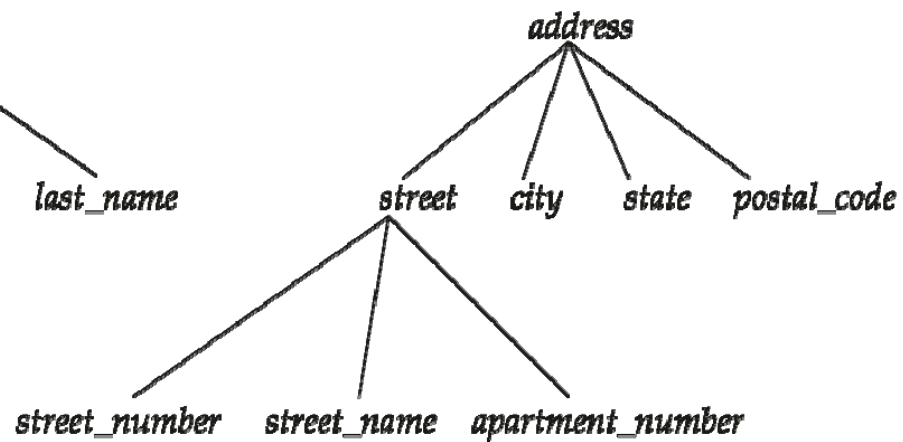


Composite Attributes

composite
attributes



component
attributes





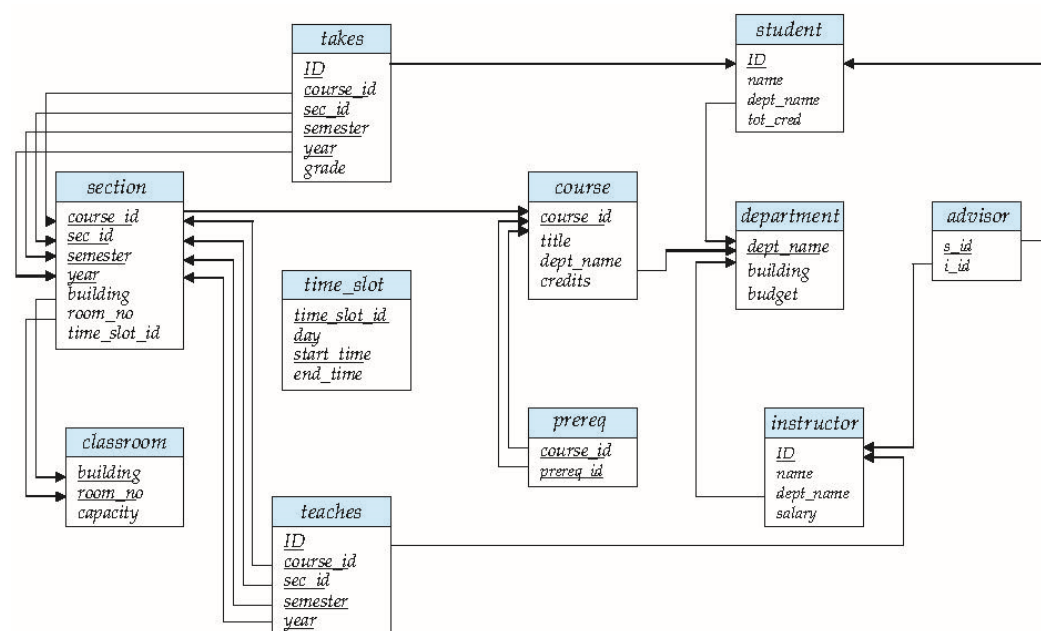
Redundant Attributes

- Suppose we have entity sets:
 - *instructor*, with attributes: *ID*, *name*, *dept_name*, *salary*
 - *department*, with attributes: *dept_name*, *building*, *budget*
- We model the fact that each instructor has an associated department using a relationship set *inst_dept*
- The attribute *dept_name* appears in both entity sets. Since it is the primary key for the entity set *department*, it replicates information present in the relationship and is therefore redundant in the entity set *instructor* and needs to be removed
- BUT: when converting back to tables, in some cases the attribute gets reintroduced, as we will see later



Weak Entity Sets

- Consider a *section* entity, which is uniquely identified by a *course_id*, *semester*, *year*, and *sec_id*.
- Clearly, section entities are related to course entities. Suppose we create a relationship set *sec_course* between entity sets *section* and *course*.
- Note that the information in *sec_course* is redundant, since *section* already has an attribute *course_id*, which identifies the course with which the section is related.
- One option to deal with this redundancy is to get rid of the relationship *sec_course*; however, by doing so the relationship between *section* and *course* becomes implicit in an attribute, which is not desirable.





Weak Entity Sets (Cont.)

- An alternative way to deal with this redundancy is to not store the attribute *course_id* in the *section* entity and to only store the remaining attributes *section_id*, *year*, and *semester*. However, the entity set *section* then does not have enough attributes to identify a particular *section* entity uniquely; although each *section* entity is distinct, sections for different courses may share the same *section_id*, *year*, and *semester*.
- To deal with this problem, we treat the relationship *sec_course* as a special relationship that provides extra information, in this case, the *course_id*, required to identify *section* entities uniquely.
- The notion of **weak entity set** formalizes the above intuition. A weak entity set is one whose existence is dependent on another entity, called its **identifying entity**; instead of associating a primary key with a weak entity, we use the identifying entity, along with extra attributes called **discriminator** to uniquely identify a weak entity. An entity set that is not a weak entity set is termed a **strong entity set**.



Weak Entity Sets (Cont.)

- Every weak entity must be associated with an identifying entity; that is, the weak entity set is said to be **existence dependent** on the identifying entity set. The identifying entity set is said to **own** the weak entity set that it identifies. The relationship associating the weak entity set with the identifying entity set is called the **identifying relationship**.
- Note that the relational schema we eventually create from the entity set *section* does have the attribute *course_id*, for reasons that will become clear later, even though we have dropped the attribute *course_id* from the entity set *section*.



Module Summary

- Introduced the Design Process for Database Systems
- Elucidated the E-R Model for real world representation with entities, entity sets, relationships, etc.



PPD

Instructor and TAs

Name	Mail	Mobile
Partha Pratim Das, Instructor	ppd@cse.iitkgp.ernet.in	9830030880
Srijoni Majumdar, TA	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, TA	himadribhuyan@gmail.com	9438911655
Gurunath Reddy M	mgurunathreddy@gmail.com	9434137638

Slides used in this presentation are borrowed from <http://db-book.com/> with kind permission of the authors.

Edited and new slides are marked with “PPD”.



Database Management Systems

Module 14: Entity-Relationship Model/2

Partha Pratim Das

*Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur*

ppd@cse.iitkgp.ernet.in

**Srijoni Majumdar
Himadri B G S Bhuyan
Gurunath Reddy M**



Database System Concepts, 6th Ed.

©Silberschatz, Korth and Sudarshan
www.db-book.com



Module Recap

- Design Process
- E-R Model
 - Entity and Entity Set
 - Relationship
 - Cardinality
 - Attributes
 - Weak Entity Sets



Module Objectives

- To illustrate E-R Diagram notation for E-R Models
- To explore translation of E-R Models to Relational Schemas



Module Outline

- E-R Diagram
- E-R Model to Relational Schema



E-R DIAGRAM

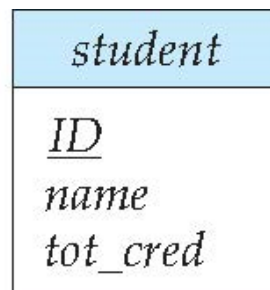
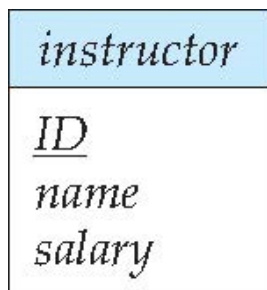
PPD

- E-R Diagram
- E-R Model to Relational Schema



Entity Sets

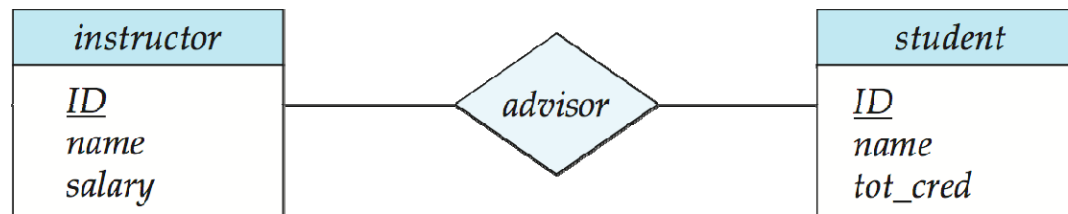
- Entities can be represented graphically as follows:
 - Rectangles represent entity sets.
 - Attributes listed inside entity rectangle
 - Underline indicates primary key attributes





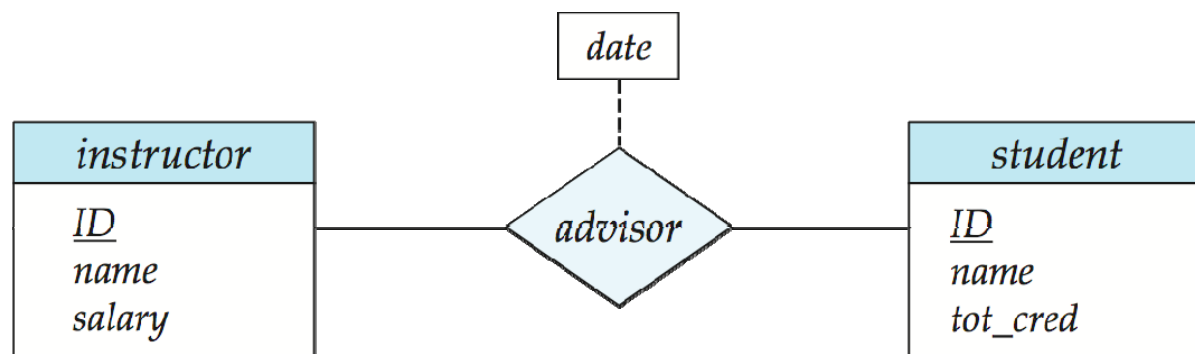
Relationship Sets

- Diamonds represent relationship sets.





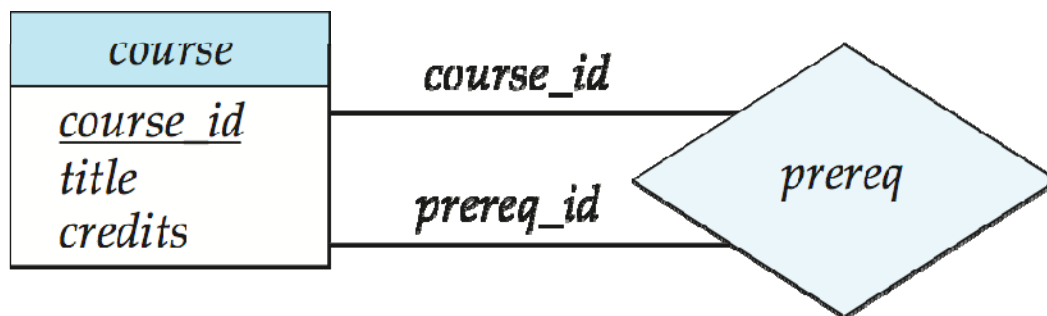
Relationship Sets with Attributes





Roles

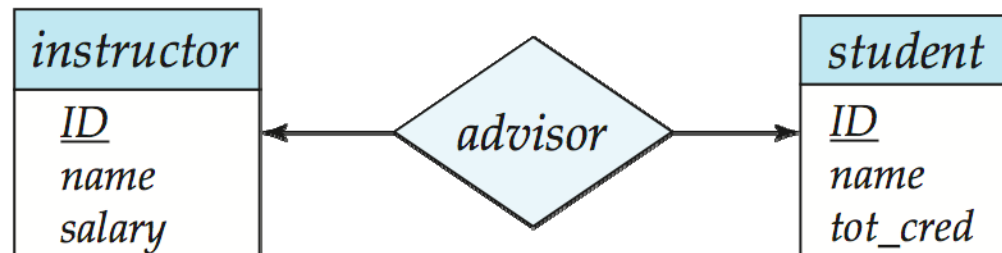
- Entity sets of a relationship need not be distinct
 - Each occurrence of an entity set plays a “role” in the relationship
- The labels “*course_id*” and “*prereq_id*” are called **roles**.





Cardinality Constraints

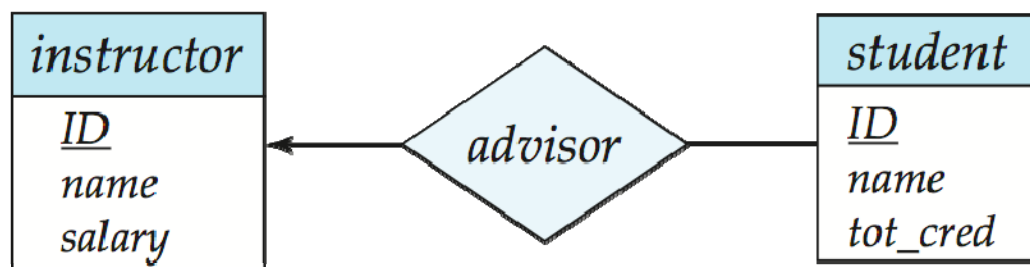
- We express cardinality constraints by drawing either a directed line (\rightarrow), signifying “one,” or an undirected line (—), signifying “many,” between the relationship set and the entity set.
- One-to-one relationship between an *instructor* and a *student* :
 - A student is associated with at most one *instructor* via the relationship *advisor*
 - A *student* is associated with at most one *department* via *stud_dept*





One-to-Many Relationship

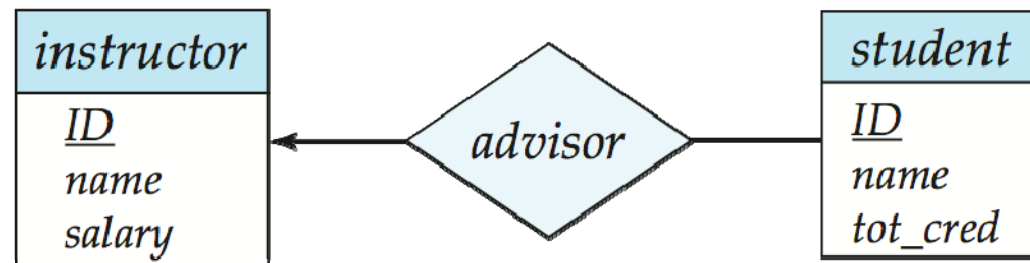
- one-to-many relationship between an *instructor* and a *student*
 - an instructor is associated with several (including 0) students via *advisor*
 - a student is associated with at most one instructor via *advisor*,





Many-to-One Relationships

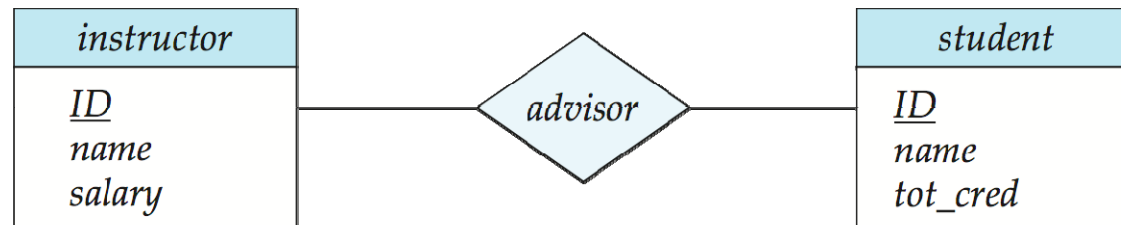
- many-to-one relationship between a *student* and a *instructor*,
 - an instructor is associated with at most one student via *advisor*,
 - and a student is associated with several (including 0) instructors via *advisor*





Many-to-Many Relationship

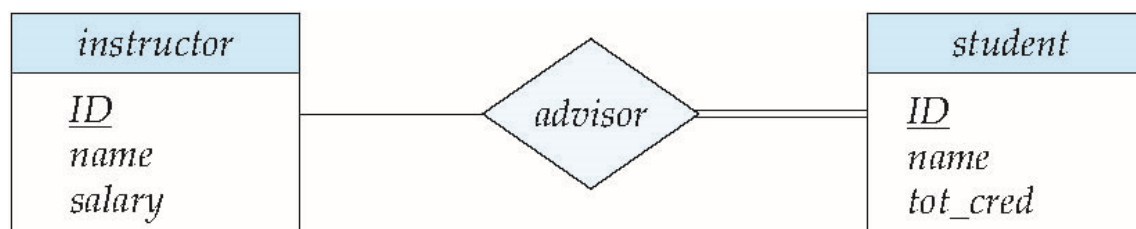
- An instructor is associated with several (possibly 0) students via *advisor*
- A student is associated with several (possibly 0) instructors via *advisor*





Total and Partial Participation

- Total participation (indicated by double line): every entity in the entity set participates in at least one relationship in the relationship set

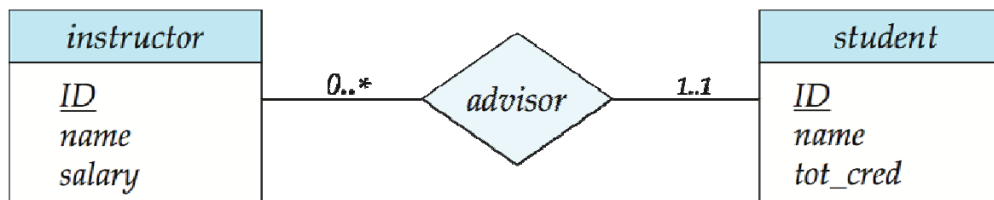


- participation of *student* in *advisor* relation is total
 - every *student* must have an associated instructor
- Partial participation: some entities may not participate in any relationship in the relationship set
 - Example: participation of *instructor* in *advisor* is partial



Notation for Expressing More Complex Constraints

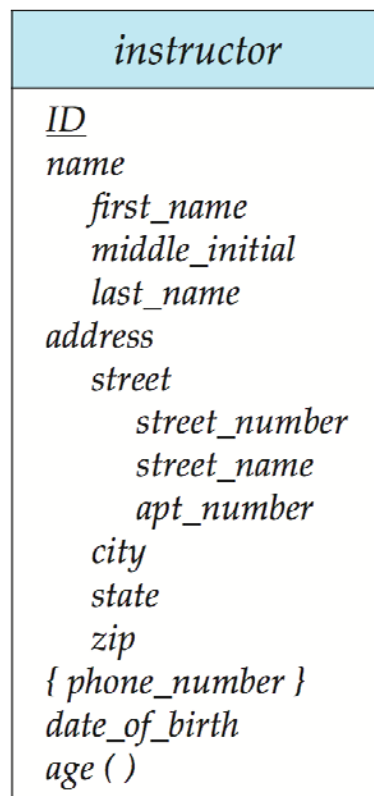
- A line may have an associated minimum and maximum cardinality, shown in the form $l..h$, where l is the minimum and h the maximum cardinality
 - A minimum value of 1 indicates total participation.
 - A maximum value of 1 indicates that the entity participates in at most one relationship
 - A maximum value of * indicates no limit.



Instructor can advise 0 or more students. A student must have 1 advisor; cannot have multiple advisors



Notation to Express Entity with Complex Attributes





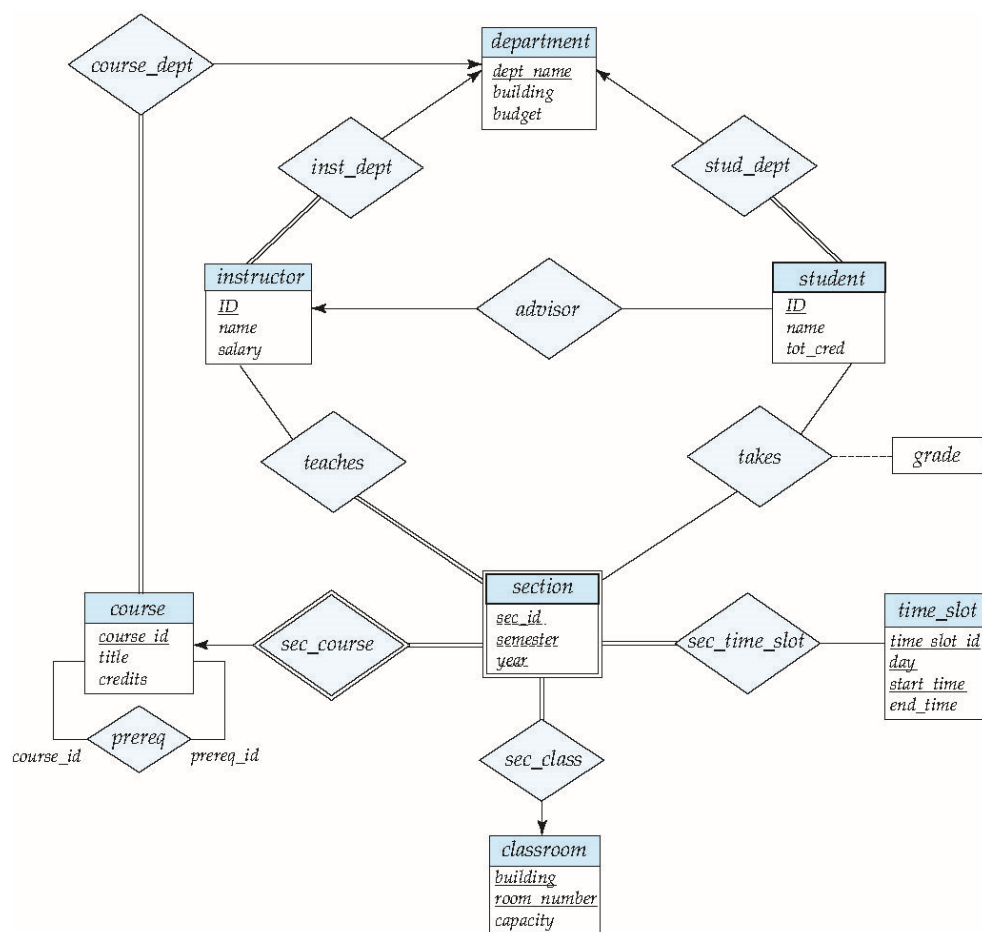
Expressing Weak Entity Sets

- In E-R diagrams, a weak entity set is depicted via a double rectangle
- We underline the discriminator of a weak entity set with a dashed line
- The relationship set connecting the weak entity set to the identifying strong entity set is depicted by a double diamond
- Primary key for *section* – (*course_id*, *sec_id*, *semester*, *year*)





E-R Diagram for a University Enterprise





PPD

- E-R Diagram
- E-R Model to Relational Schema

E-R MODEL TO RELATIONAL SCHEMA



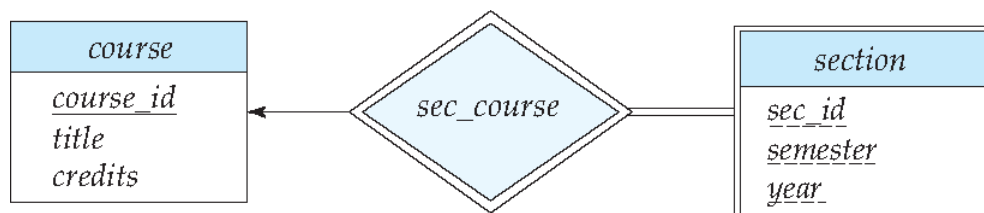
Reduction to Relation Schemas

- Entity sets and relationship sets can be expressed uniformly as *relation schemas* that represent the contents of the database
- A database which conforms to an E-R diagram can be represented by a collection of schemas
- For each entity set and relationship set there is a unique schema that is assigned the name of the corresponding entity set or relationship set
- Each schema has a number of columns (generally corresponding to attributes), which have unique names



Representing Entity Sets

- A strong entity set reduces to a schema with the same attributes
student(ID, name, tot_cred)
- A weak entity set becomes a table that includes a column for the primary key of the identifying strong entity set
section (course_id, sec_id, sem, year)

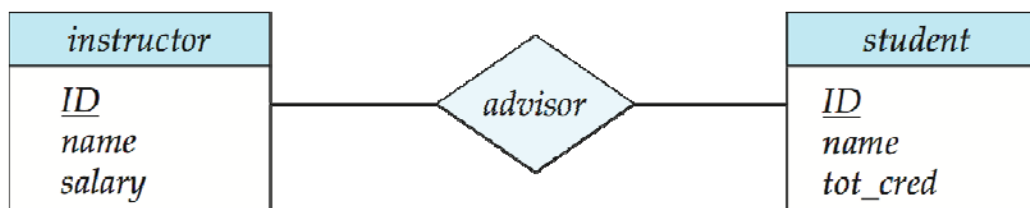




Representing Relationship Sets

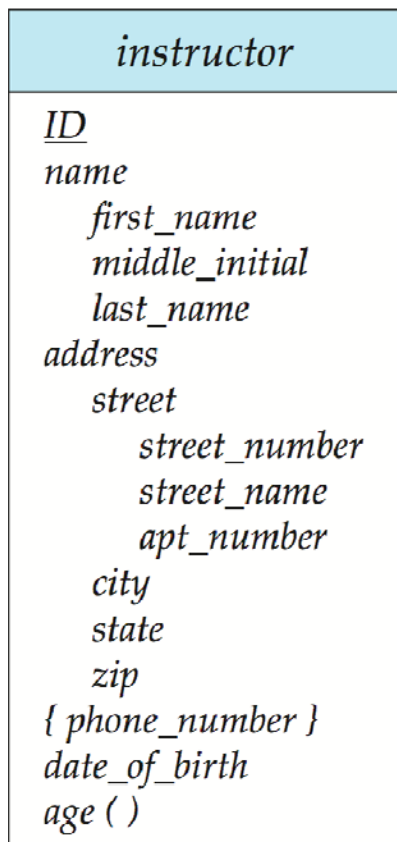
- A many-to-many relationship set is represented as a schema with attributes for the primary keys of the two participating entity sets, and any descriptive attributes of the relationship set.
- Example: schema for relationship set *advisor*

$advisor = (\underline{s_id}, \underline{i_id})$





Representation of Entity Sets with Composite Attributes



- Composite attributes are flattened out by creating a separate attribute for each component attribute
 - Example: given entity set *instructor* with composite attribute *name* with component attributes *first_name* and *last_name* the schema corresponding to the entity set has two attributes *name_first_name* and *name_last_name*
 - Prefix omitted if there is no ambiguity (*name_first_name* could be *first_name*)
- Ignoring multivalued attributes, extended instructor schema is
 - *instructor*(*ID*,
 first_name, *middle_initial*, *last_name*,
 street_number, *street_name*,
 apt_number, *city*, *state*, *zip_code*,
 date_of_birth)



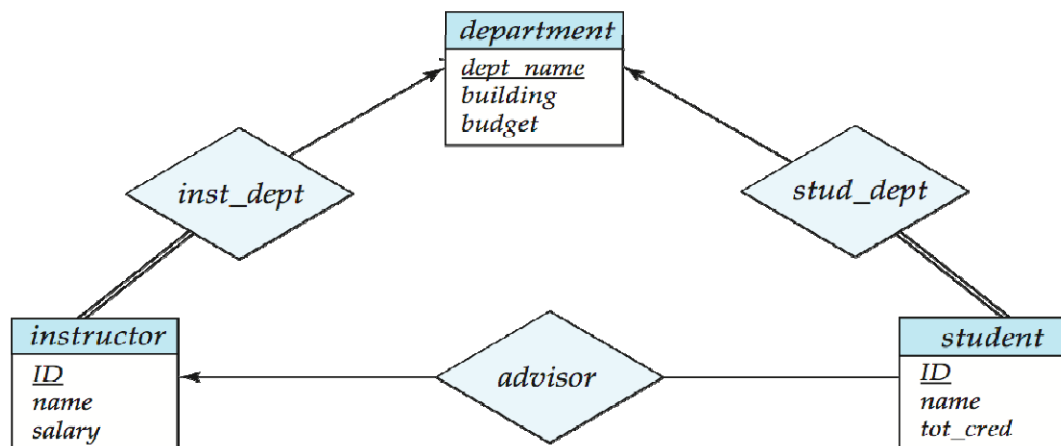
Representation of Entity Sets with Multivalued Attributes

- A multivalued attribute M of an entity E is represented by a separate schema EM
- Schema EM has attributes corresponding to the primary key of E and an attribute corresponding to multivalued attribute M
- Example: Multivalued attribute *phone_number* of *instructor* is represented by a schema:
 $inst_phone = (\underline{ID}, \underline{phone_number})$
- Each value of the multivalued attribute maps to a separate tuple of the relation on schema EM
 - For example, an *instructor* entity with primary key 22222 and phone numbers 456-7890 and 123-4567 maps to two tuples:
(22222, 456-7890) and (22222, 123-4567)



Redundancy of Schemas

- Many-to-one and one-to-many relationship sets that are total on the many-side can be represented by adding an extra attribute to the “many” side, containing the primary key of the “one” side
- Example: Instead of creating a schema for relationship set *inst_dept*, add an attribute *dept_name* to the schema arising from entity set *instructor*





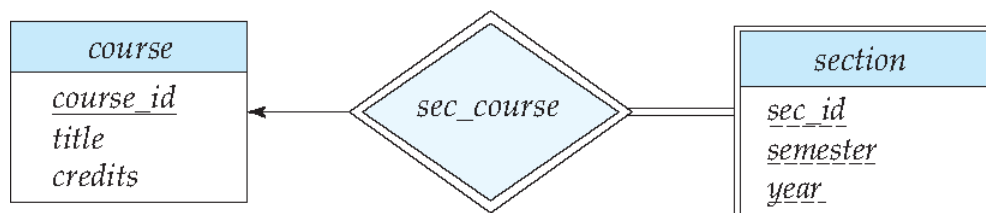
Redundancy of Schemas (Cont.)

- For one-to-one relationship sets, either side can be chosen to act as the “many” side
 - That is, an extra attribute can be added to either of the tables corresponding to the two entity sets
- If participation is *partial* on the “many” side, replacing a schema by an extra attribute in the schema corresponding to the “many” side could result in null values



Redundancy of Schemas (Cont.)

- The schema corresponding to a relationship set linking a weak entity set to its identifying strong entity set is redundant.
- Example: The *section* schema already contains the attributes that would appear in the *sec_course* schema





Module Summary

- Illustrated E-R Diagram notation for E-R Models
- Discussed translation of E-R Models to Relational Schemas



PPD

Instructor and TAs

Name	Mail	Mobile
Partha Pratim Das, Instructor	ppd@cse.iitkgp.ernet.in	9830030880
Srijoni Majumdar, TA	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, TA	himadribhuyan@gmail.com	9438911655
Gurunath Reddy M	mgurunathreddy@gmail.com	9434137638

Slides used in this presentation are borrowed from <http://db-book.com/> with kind permission of the authors.

Edited and new slides are marked with “PPD”.



Database Management Systems

Module 15: Entity-Relationship Model/3

Partha Pratim Das

*Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur*

ppd@cse.iitkgp.ernet.in

**Srijoni Majumdar
Himadri B G S Bhuyan
Gurunath Reddy M**



Database System Concepts, 6th Ed.

©Silberschatz, Korth and Sudarshan
www.db-book.com



Module Recap

- E-R Diagram
- E-R Model to Relational Schema



Module Objectives

- To understand extended features of E-R Model
- To discuss various design issues



Module Outline

- Extended E-R Features
- Design Issues



PPD

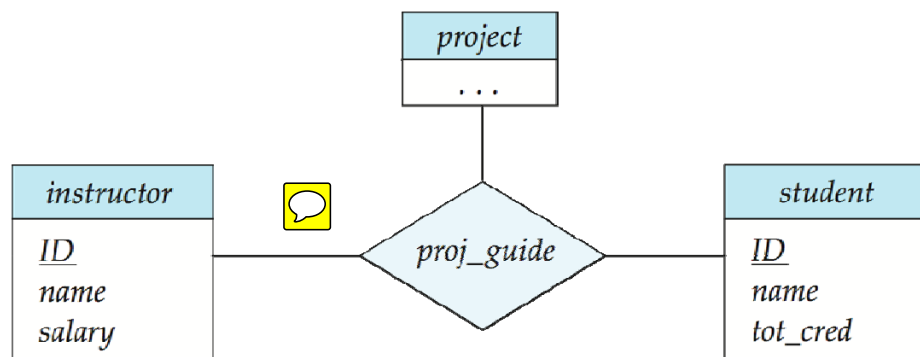
- Extended E-R Features
- Design Issues

EXTENDED E-R FEATURES



Non-binary Relationship Sets

- Most relationship sets are binary
- There are occasions when it is more convenient to represent relationships as non-binary.
- E-R Diagram with a Ternary Relationship





Cardinality Constraints on Ternary Relationship

- We allow **at most one arrow out** of a ternary (or greater degree) relationship to indicate a cardinality constraint
- For example, an arrow from *proj_guide* to *instructor* indicates each student has at most one guide for a project
- If there is more than one arrow, there are two ways of defining the meaning.
 - For example, a ternary relationship R between A , B and C with arrows to B and C could mean
 1. Each A entity is associated with a unique entity from B and C or
 2. Each pair of entities from (A, B) is associated with a unique C entity, and each pair (A, C) is associated with a unique B
 - Each alternative has been used in different formalisms
 - To avoid confusion we outlaw more than one arrow



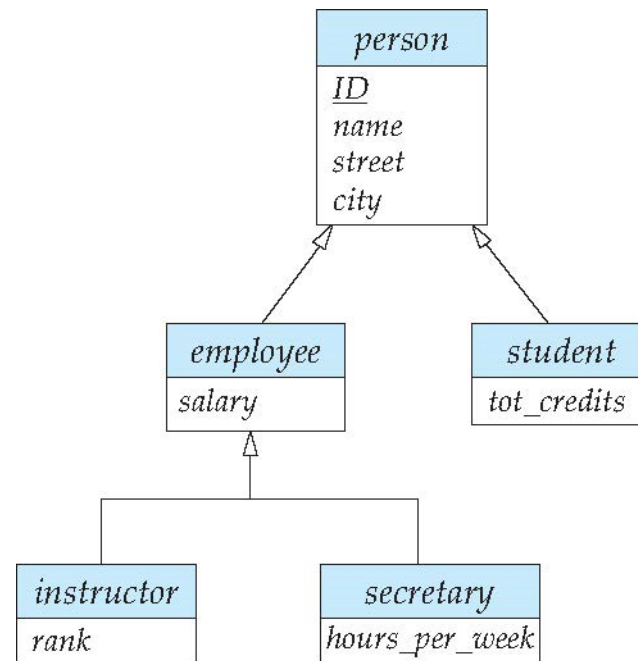
Specialization

- Top-down design process; we designate sub-groupings within an entity set that are distinctive from other entities in the set
- These sub-groupings become lower-level entity sets that have attributes or participate in relationships that do not apply to the higher-level entity set
- Depicted by a *triangle* component labeled ISA (e.g., *instructor* “is a” *person*)
- **Attribute inheritance** – a lower-level entity set inherits all the attributes and relationship participation of the higher-level entity set to which it is linked



Specialization Example

- **Overlapping** – *employee* and *student*
- **Disjoint** – *instructor* and *secretary*
- Total and partial





Representing Specialization via Schemas

- Method 1:
 - Form a schema for the higher-level entity
 - Form a schema for each lower-level entity set, include primary key of higher-level entity set and local attributes

schema	attributes
person	ID, name, street, city
student	ID, tot_cred
employee	ID, salary

- Drawback: Getting information about, an *employee* requires **accessing two relations**, the one corresponding to the low-level schema and the one corresponding to the high-level schema



Representing Specialization as Schemas (Cont.)

- Method 2:
 - Form a schema for each entity set with all local and inherited attributes

schema	attributes
person	ID, name, street, city
student	ID, name, street, city, tot_cred
employee	ID, name, street, city, salary

- Drawback: *name*, *street* and *city* may be stored redundantly for people who are both students and employees



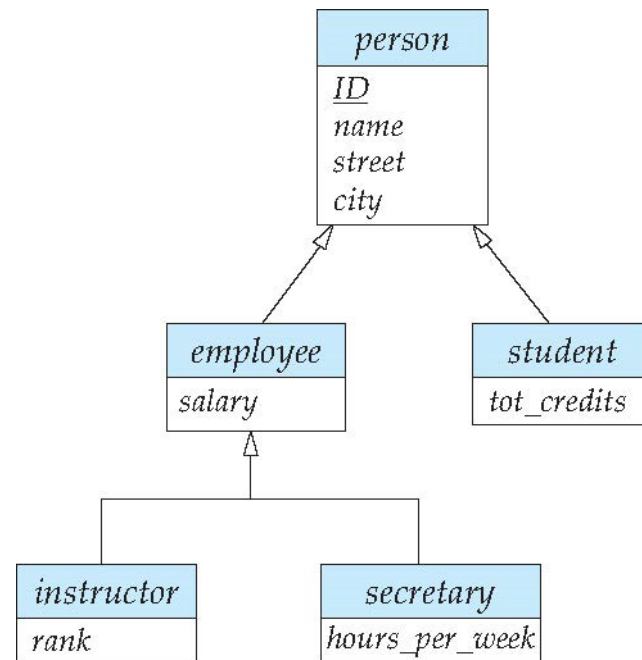
Generalization

- **A bottom-up design process** – combine a number of entity sets that share the same features into a higher-level entity set.
- Specialization and generalization are **simple inversions of each other**; they are represented in an E-R diagram in the same way
- The terms specialization and generalization are used interchangeably



Design Constraints on a Specialization/Generalization

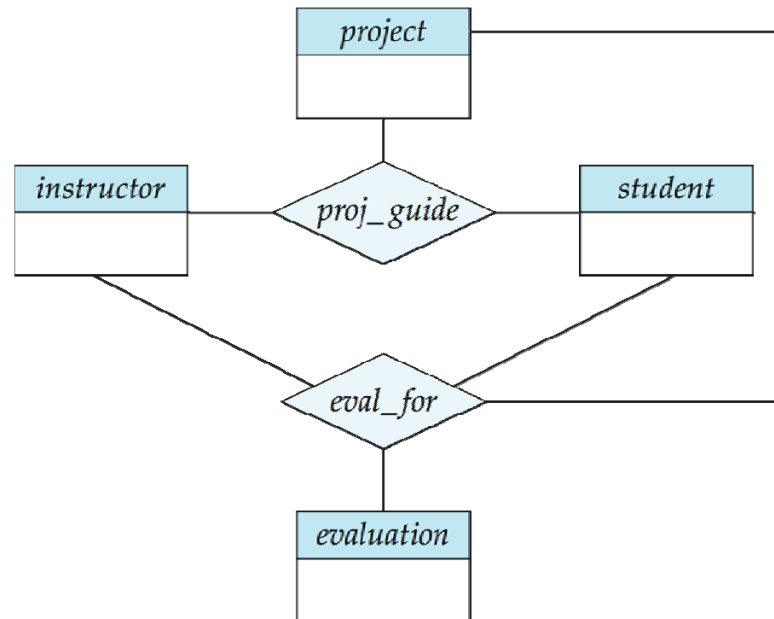
- **Completeness constraint** -- specifies whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets within a generalization
 - **total**: an entity must belong to one of the lower-level entity sets
 - **partial**: an entity need not belong to one of the lower-level entity sets
- Partial generalization is the default. We can specify total generalization in an ER diagram by adding the keyword **total** in the diagram and drawing a dashed line from the keyword to the corresponding hollow arrow-head to which it applies (for a total generalization), or to the set of hollow arrow-heads to which it applies (for an overlapping generalization).
- The *student* generalization is total: All student entities must be either graduate or undergraduate. Because the higher-level entity set arrived at through generalization is generally composed of only those entities in the lower-level entity sets, the completeness constraint for a generalized higher-level entity set is usually total





Aggregation

- Consider the ternary relationship *proj_guide*, which we saw earlier
- Suppose we want to record evaluations of a student by a guide on a project





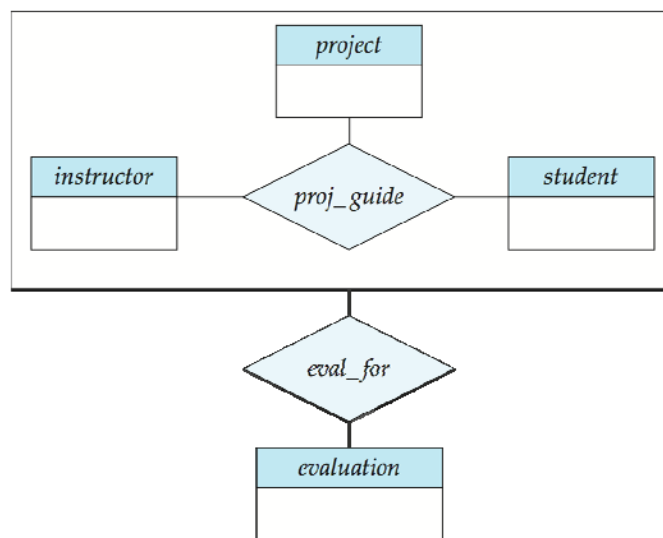
Aggregation (Cont.)

- Relationship sets *eval_for* and *proj_guide* represent overlapping information
 - Every *eval_for* relationship corresponds to a *proj_guide* relationship
 - However, some *proj_guide* relationships may not correspond to any *eval_for* relationships
 - So we cannot discard the *proj_guide* relationship
- Eliminate this redundancy via *aggregation*
 - Treat relationship as an abstract entity
 - Allows relationships between relationships
 - Abstraction of relationship into new entity



Aggregation (Cont.)

- Eliminate this redundancy via *aggregation* without introducing redundancy, the following diagram represents:
 - A student is guided by a particular instructor on a particular project
 - A student, instructor, project combination may have an associated evaluation





Representing Aggregation via Schemas

- To represent aggregation, create a schema containing
 - Primary key of the aggregated relationship,
 - The primary key of the associated entity set
 - Any descriptive attributes
- In our example:
 - The schema *eval_for* is:
eval_for (*s_ID*, *project_id*, *i_ID*, *evaluation_id*)
 - The schema *proj_guide* is redundant



PPD

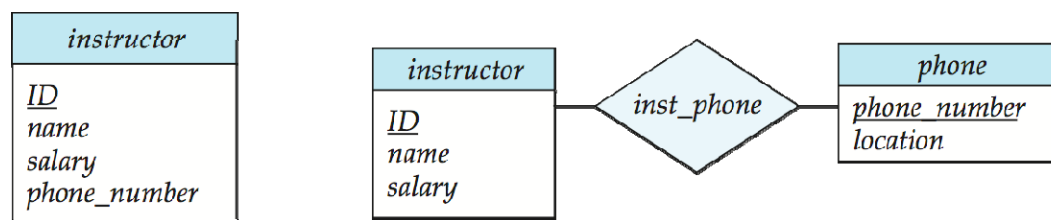
- Extended E-R Features
- Design Issues**

DESIGN ISSUES



Entities vs. Attributes

- Use of entity sets vs. attributes



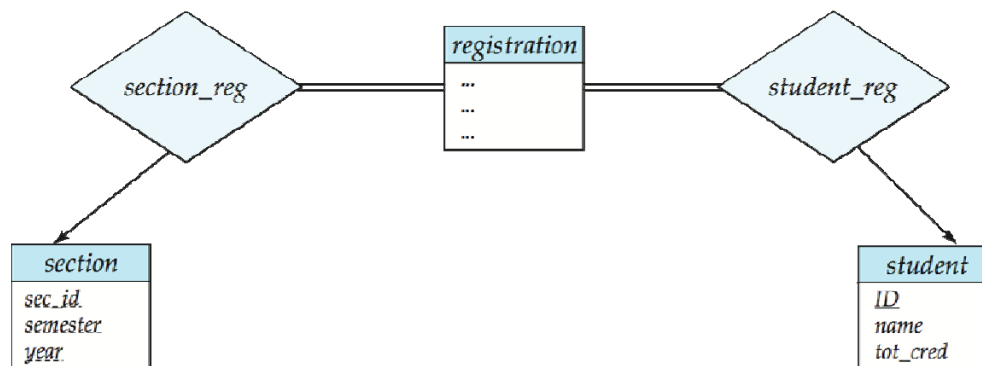
- Use of phone as an entity allows extra information about phone numbers (plus multiple phone numbers)



Entities vs. Relationship sets

- **Use of entity sets vs. relationship sets**

Possible guideline is to designate a relationship set to describe an action that occurs between entities



- **Placement of relationship attributes**

For example, attribute date as attribute of advisor or as attribute of student



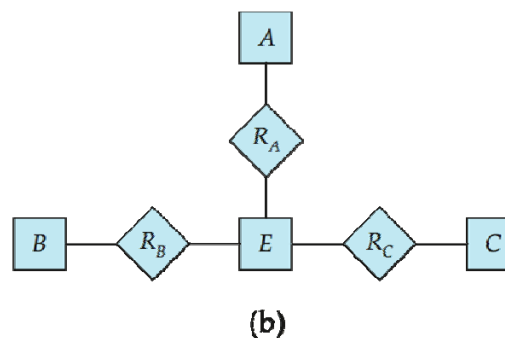
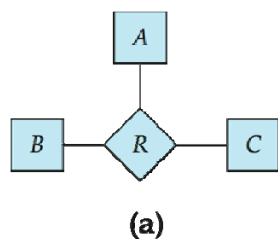
Binary Vs. Non-Binary Relationships

- Although it is possible to replace any non-binary (n -ary, for $n > 2$) relationship set by a number of distinct binary relationship sets, a n -ary relationship set shows more clearly that several entities participate in a single relationship
- Some relationships that appear to be non-binary may be better represented using binary relationships
 - For example, a ternary relationship *parents*, relating a child to his/her father and mother, is best replaced by two binary relationships, *father* and *mother*
 - Using two binary relationships allows partial information (e.g., only mother being known)
 - But there are some relationships that are naturally non-binary
 - Example: *proj_guide*



Converting Non-Binary Relationships to Binary Form

- In general, any non-binary relationship can be represented using binary relationships by creating an artificial entity set.
 - Replace R between entity sets A , B and C by an entity set E , and three relationship sets:
 1. R_A , relating E and A
 2. R_B , relating E and B
 3. R_C , relating E and C
 - Create an identifying attribute for E and add any attributes of R to E
 - For each relationship (a_i, b_i, c_i) in R , create
 1. a new entity e_i in the entity set E
 2. add (e_i, a_i) to R_A
 3. add (e_i, b_i) to R_B
 4. add (e_i, c_i) to R_C





Converting Non-Binary Relationships (Cont.)

- Also need to translate constraints
 - Translating all constraints may not be possible
 - There may be instances in the translated schema that cannot correspond to any instance of R
 - Exercise: *add constraints to the relationships R_A , R_B and R_C to ensure that a newly created entity corresponds to exactly one entity in each of entity sets A , B and C*
 - We can avoid creating an identifying attribute by making E a weak entity set (described shortly) identified by the three relationship sets

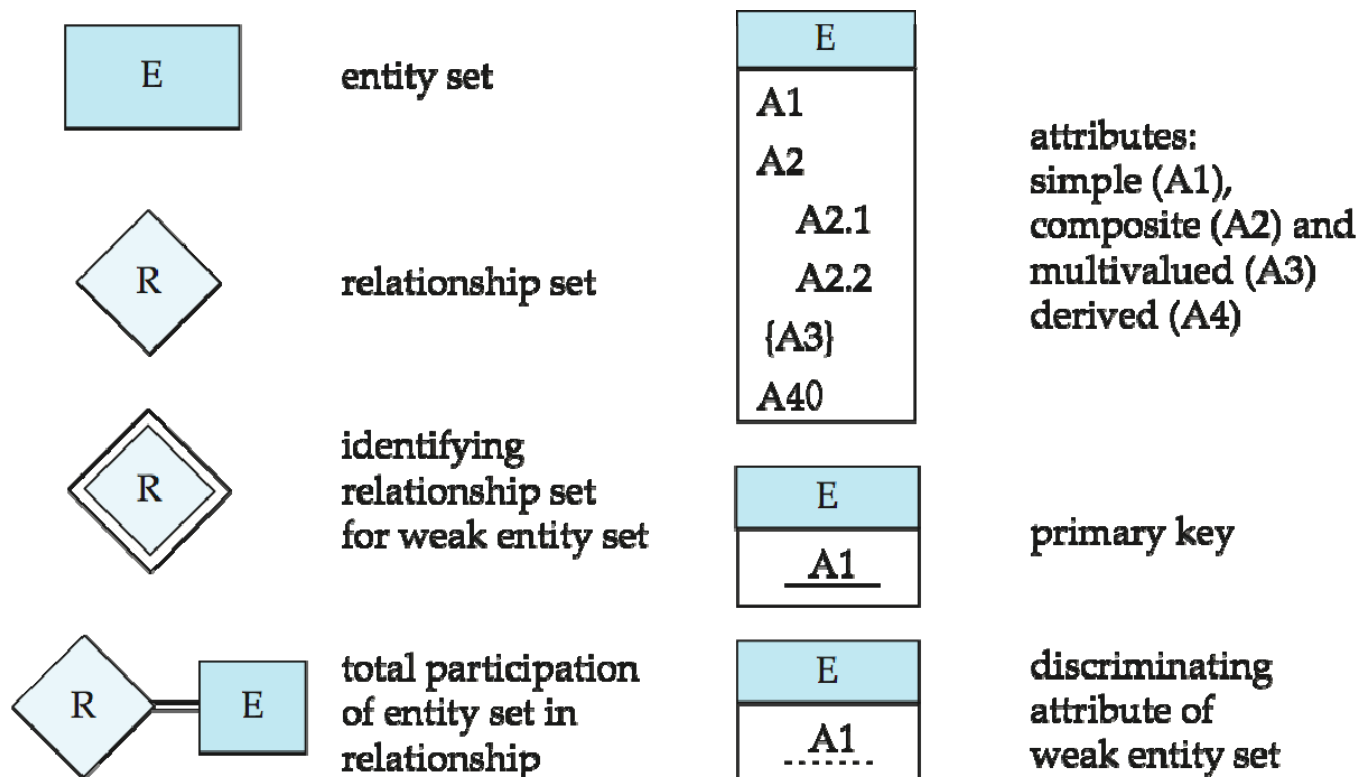


E-R Design Decisions

- The use of an attribute or entity set to represent an object
- Whether a real-world concept is best expressed by an entity set or a relationship set
- The use of a ternary relationship versus a pair of binary relationships
- The use of a strong or weak entity set
- The use of specialization/generalization – contributes to modularity in the design
- The use of aggregation – can treat the aggregate entity set as a single unit without concern for the details of its internal structure

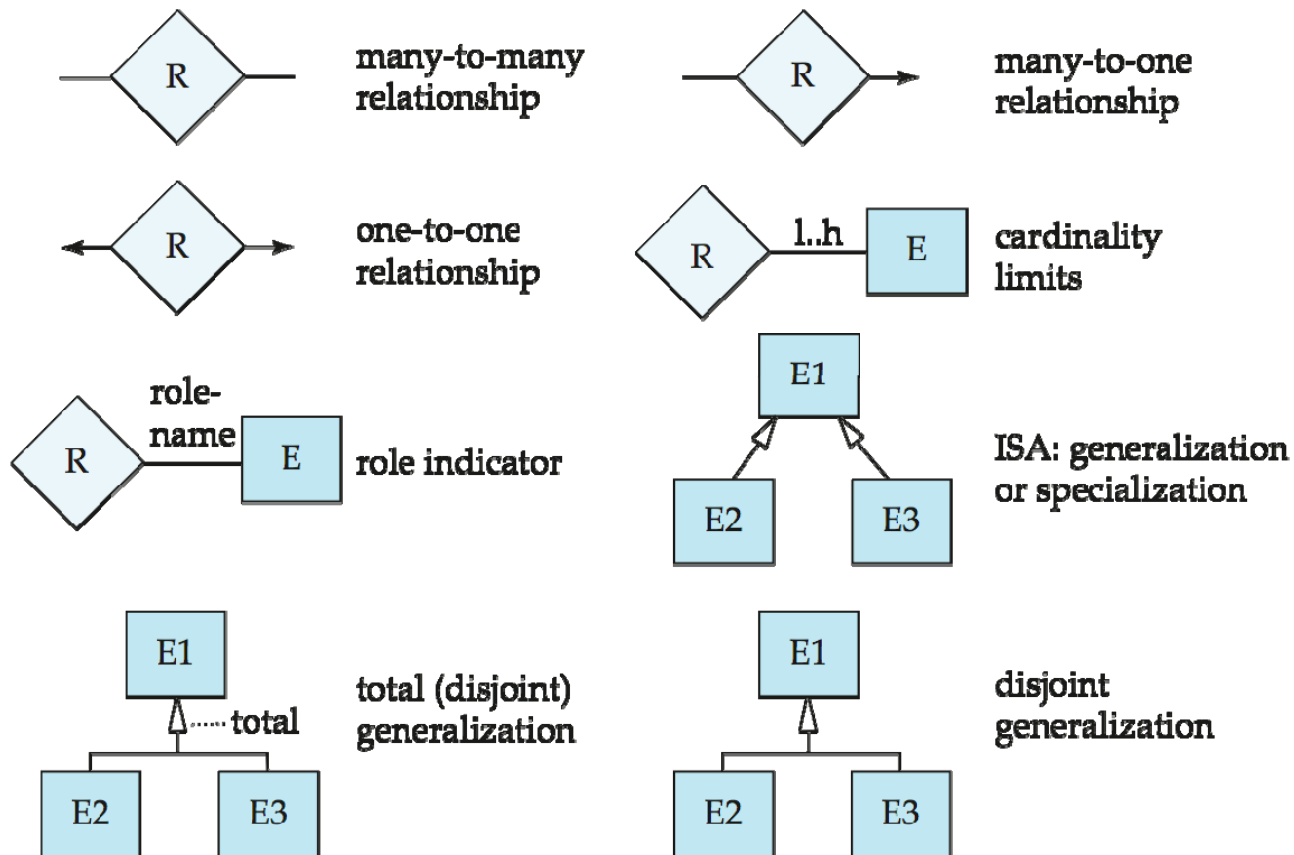


Summary of Symbols Used in E-R Notation





Symbols Used in E-R Notation (Cont.)

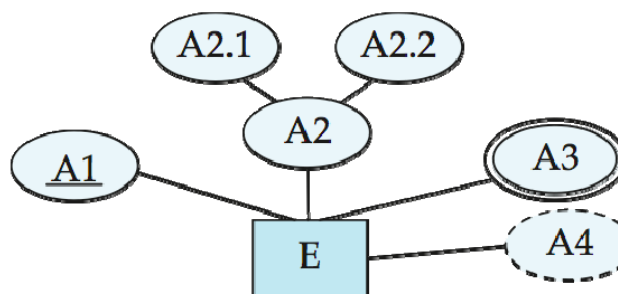




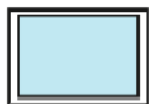
Alternative ER Notations

- Chen, IDE1FX, ...

entity set E with
simple attribute A1,
composite attribute A2,
multivalued attribute A3,
derived attribute A4,
and primary key A1



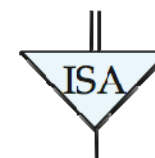
weak entity set



generalization



total
generalization



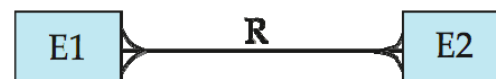
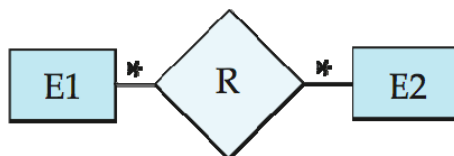


Alternative ER Notations

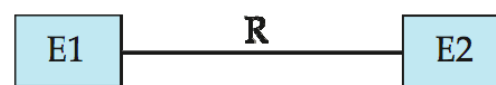
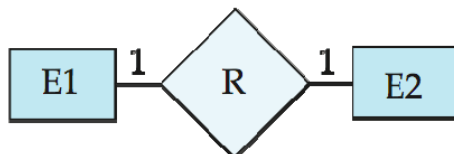
Chen

IDE1FX (Crows feet notation)

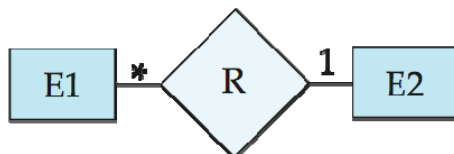
many-to-many
relationship



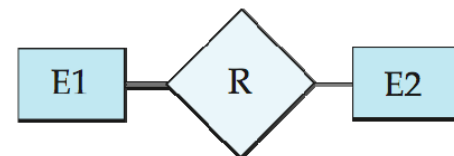
one-to-one
relationship



many-to-one
relationship



participation
in R: total (E1)
and partial (E2)





Module Summary

- Discussed the extended features of E-R Model
- Deliberated on various design issues



PPD

Instructor and TAs

Name	Mail	Mobile
Partha Pratim Das, Instructor	ppd@cse.iitkgp.ernet.in	9830030880
Srijoni Majumdar, TA	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, TA	himadribhuyan@gmail.com	9438911655
Gurunath Reddy M	mgurunathreddy@gmail.com	9434137638

Slides used in this presentation are borrowed from <http://db-book.com/> with kind permission of the authors.

Edited and new slides are marked with “PPD”.