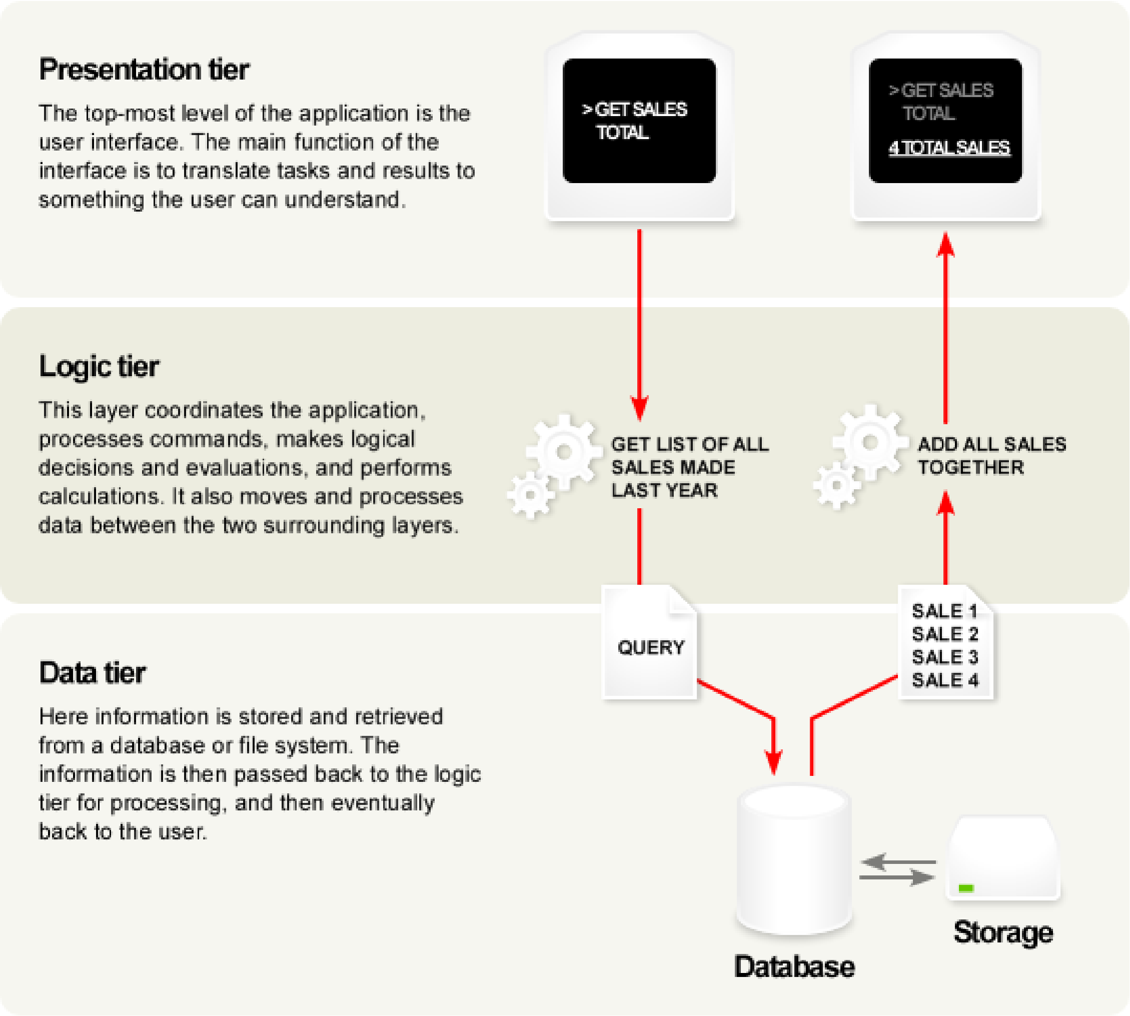
CSC309: Introduction to Web Programming

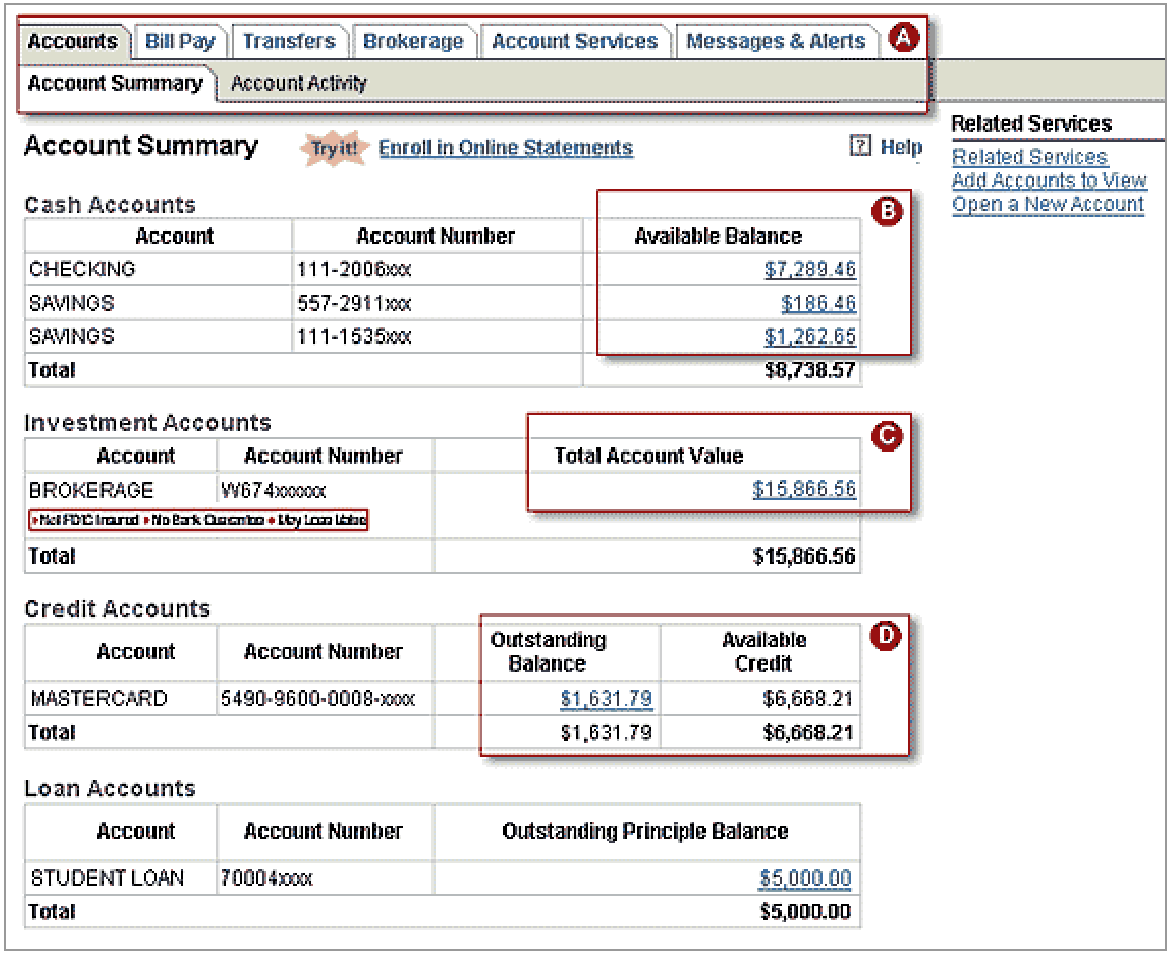
Lecture 6

Wael Aboulsaadat

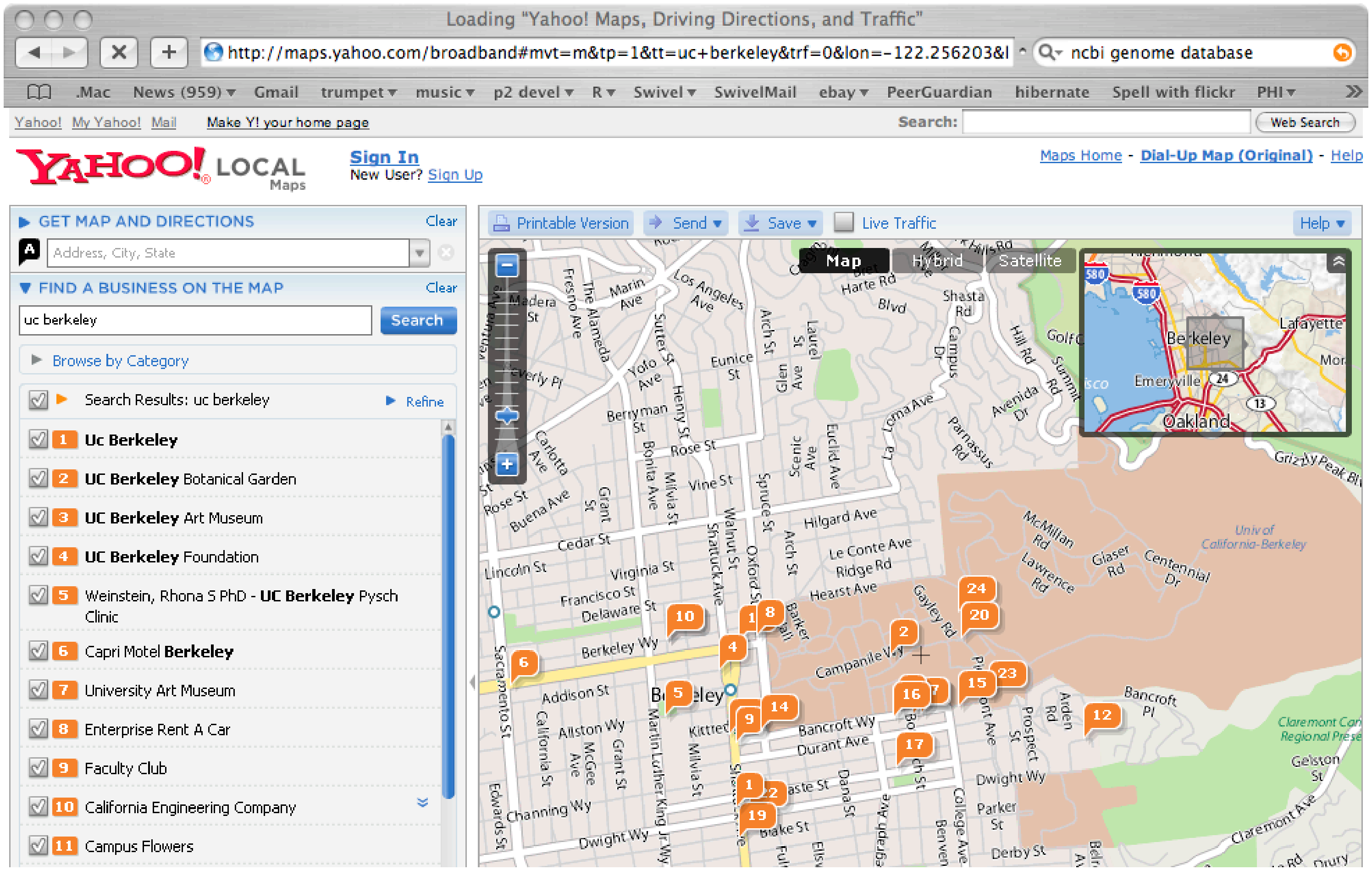
3-Tier Architecture



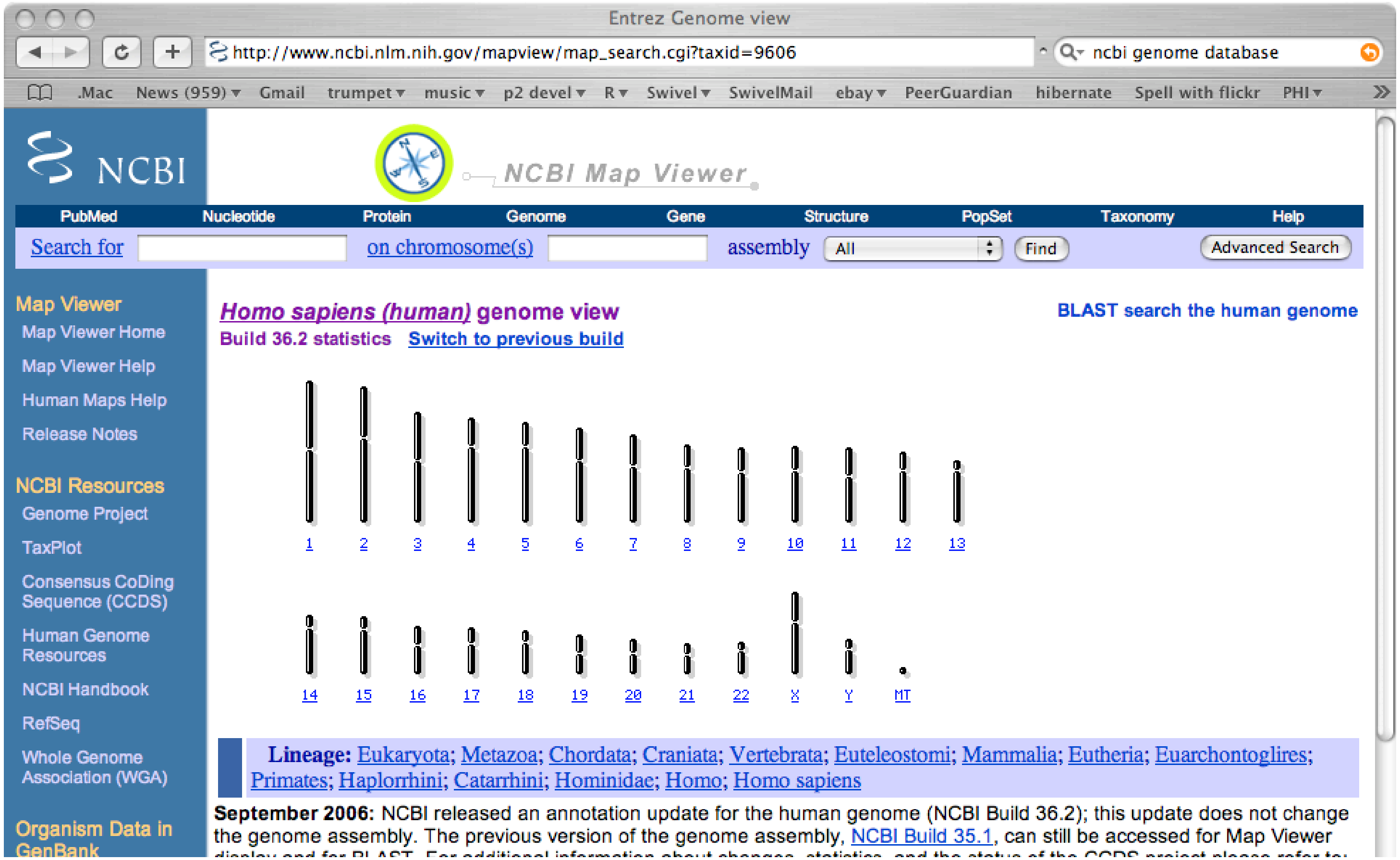
What: Database Systems Today



What: Database Systems Today



What: Database Systems Today



Database Management System (DBMS)

- A collection of programs that enable:
  - Defining (describing the structure),
  - Populating by data (Constructing),
  - Manipulating (querying, updating),
  - Preserving consistency,
  - Protecting from misuse,
  - Recovering from failure, and
  - Concurrent using

  of a database.

Steps in Database Design

1. Requirements Analysis
2. Conceptual Design
3. Logical Design
4. Schema Refinement
5. Physical Design - indexes, disk layout
6. Security Design - who accesses what, and how

Steps in Database Design: conceptual design

A. Define ER Model

B. Translate ER Model to Relational Model

Entity Relation Model (ER)

- Entities
- Attributes
- Relations
- Roles

ER: entities

- A 'thing' is called an Entity
- An entity can be an actual physical object or a

  conceptual object
- And that's it!

ER: how to model entities?

- An entity is an object that is distinguishable from other objects
  - E.g. a specific person, a course module, an event

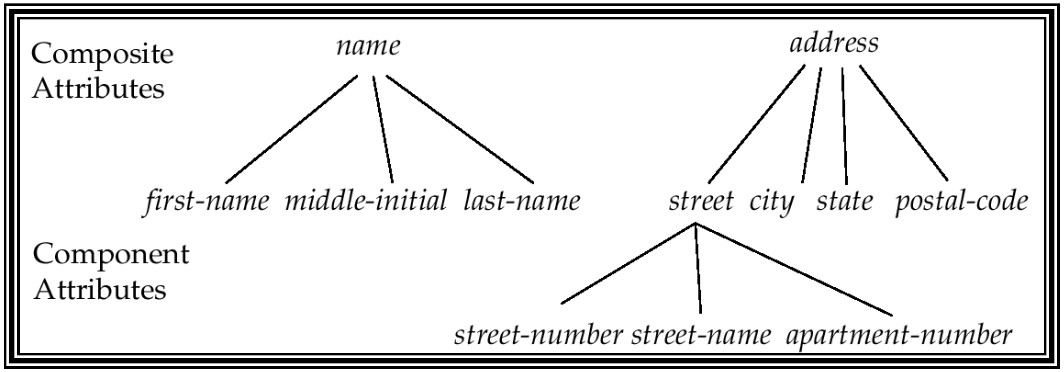
- Note:
  - The fact that two people have the same name does not mean that they are indeed the same entity. They could just share the same attribute value

ER: attributes

- An entity is represented by a set of attributes, that is descriptive properties possessed by all members of an entity set.
  - Example:
    - *customer = (customer-id, customer-name, customer-street, customer-city)*
    - *loan = (loan-number, amount)*
- *Domain* – the set of permitted values for each attribute

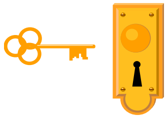
ER: attributes types

- Attribute types:
  - *Simple* and *composite* attributes (e.g., address).



  - *Single-valued* and *multi-valued* attributes
    - E.g. multi-valued attribute: *phone-numbers*
  - *Derived* attributes
    - Can be computed from other attributes
    - E.g. *age*, given the date of birth

ER: a special attribute – key



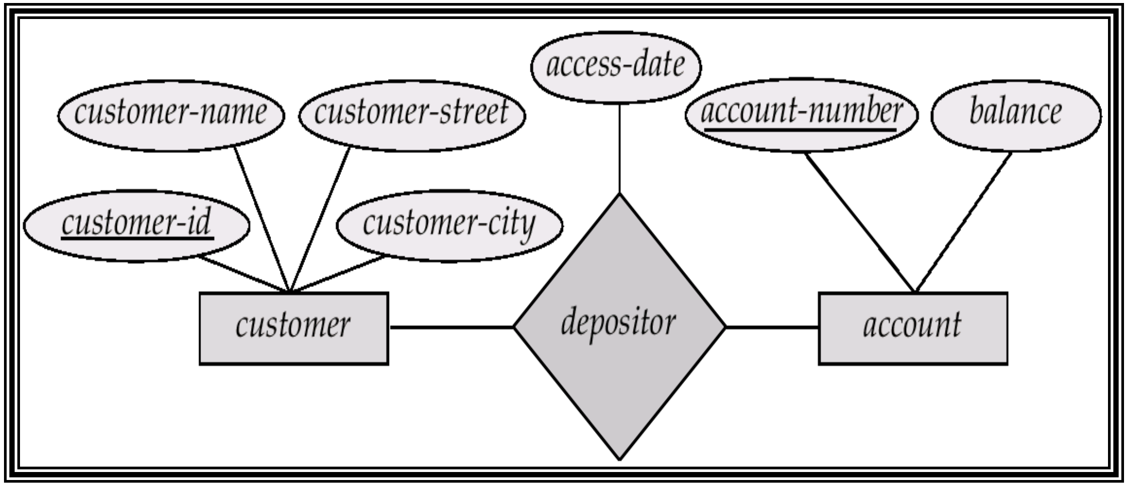
- How to distinguish between entities?
- A *key* of an entity is a set of one or more attributes whose values uniquely determine each entity.
- A Key can be *simple* (a single attribute) or *composite* (more than one field)

ER: relations

- Association among two or more entities.
  - E.g., John *works* in Pharmacy department.
- A relation can have it’s own attributes as well…

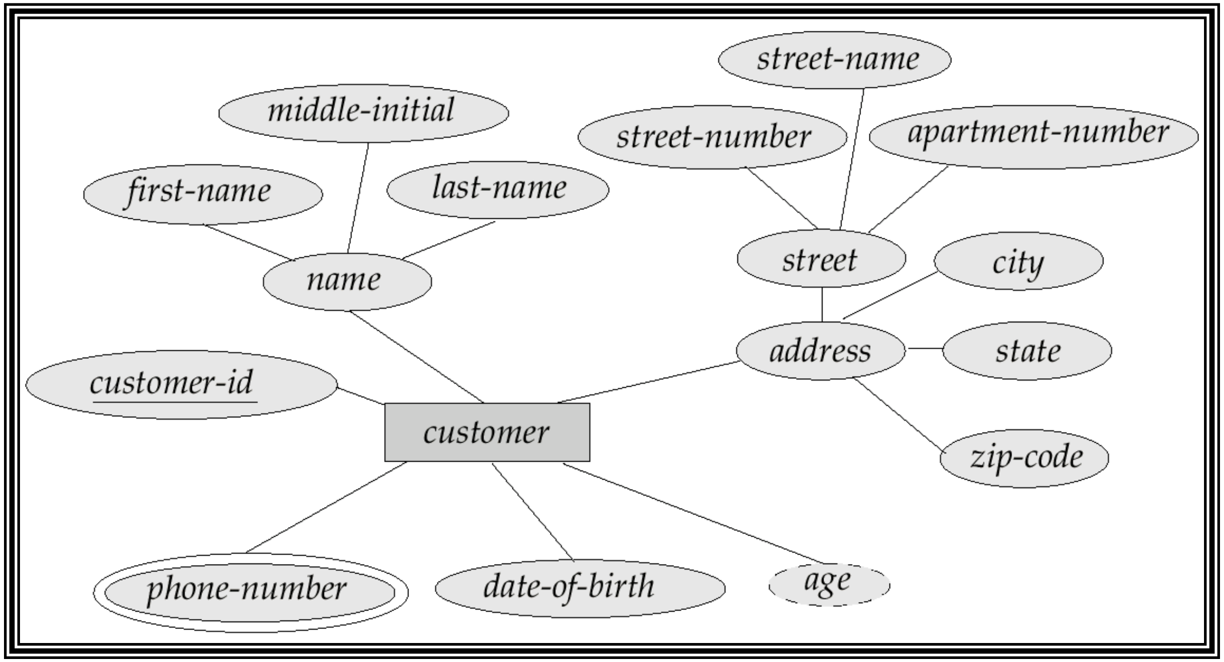
ER: visual notation

- Rectangles represent entity sets.
- Diamonds represent relationship sets.
- Lines link attributes to entity sets and entity sets to relationship sets.
- Ellipses represent attributes
- Underline for keys



ER: visual notation - cont'd

- Ellipses represent attributes
  - Double ellipses represent multi-valued attributes.
  - Dashed ellipses denote derived attributes.

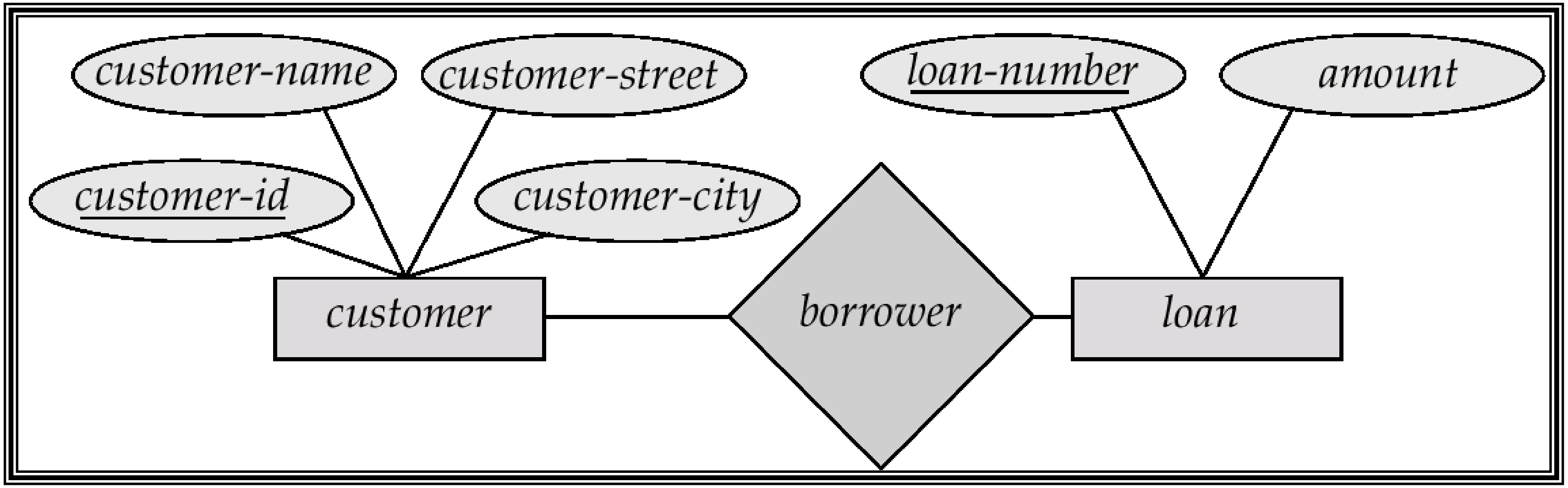


ER: cardinality constraints

- We express cardinality constraints by drawing either a directed line (→), signifying “one,” or
- an undirected line (—), signifying “many,” between the relationship and the entity.

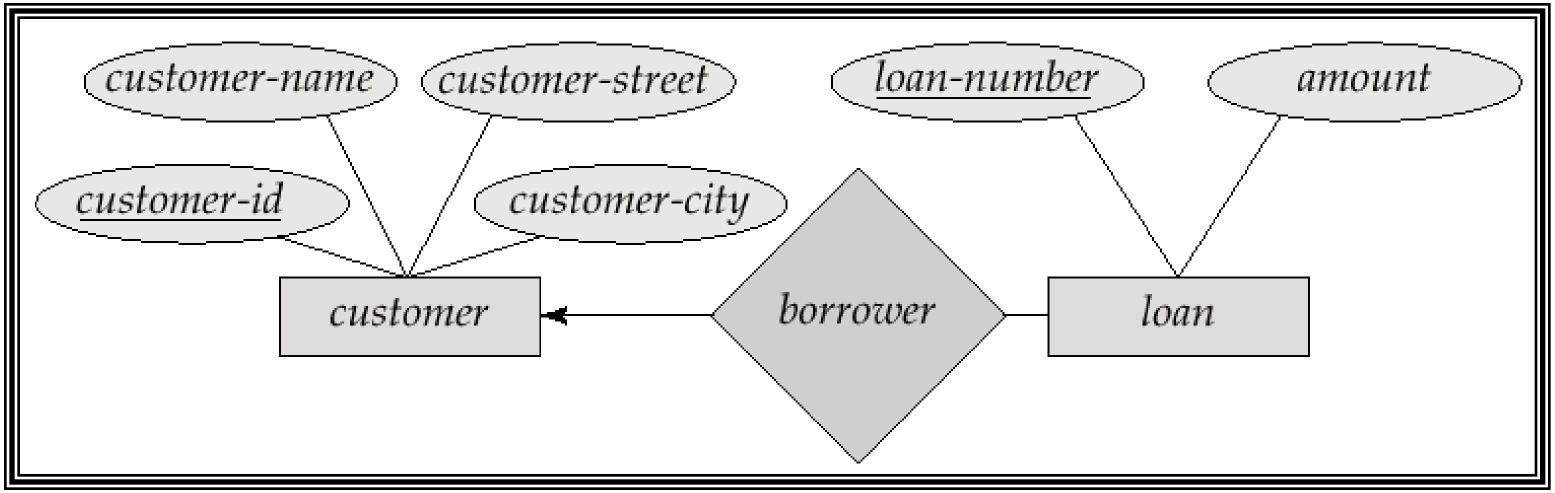
ER: cardinality constraints

- Many-to-many relationship
  - A customer is associated with several (possibly 0) loans via borrower
  - A loan is associated with several (possibly 0) customers via borrower



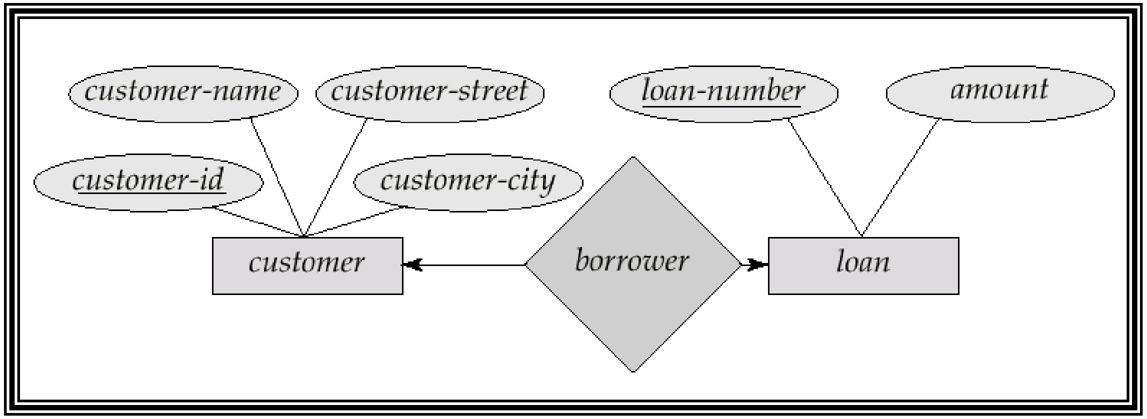
ER: cardinality constraints

- One-to-many relationship
  - a loan is associated with at most one customer via *borrower*, a customer is associated with several (including 0) loans via *borrower*



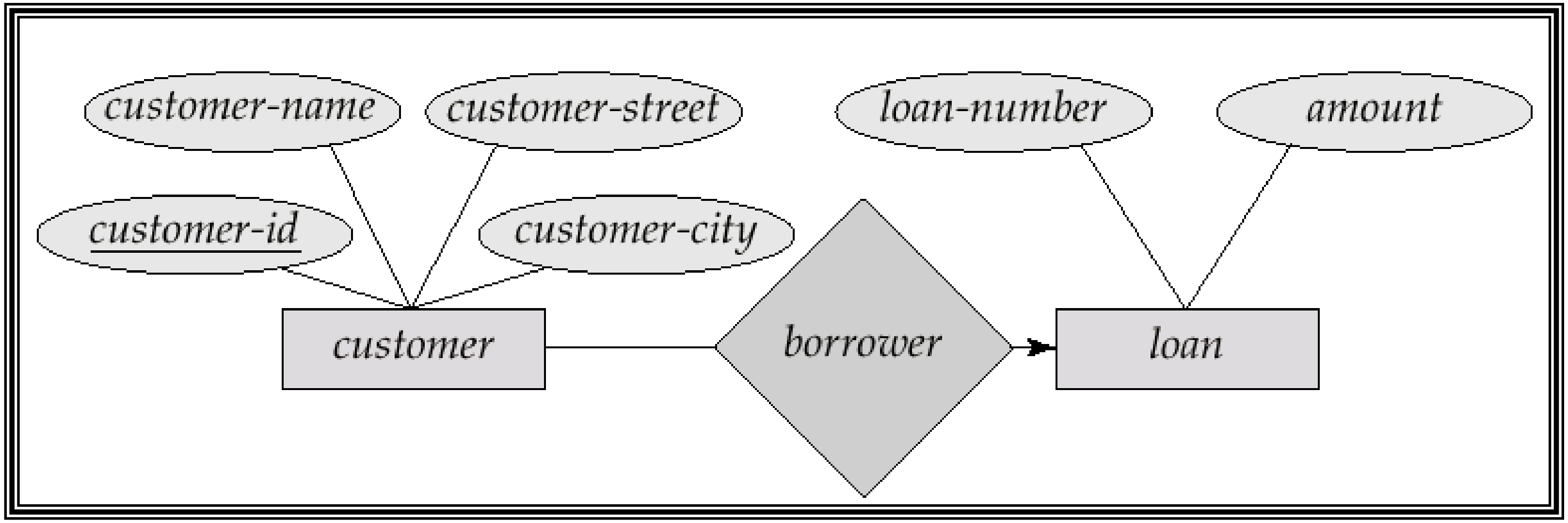
ER: cardinality constraints

- One-to-one relationship:
  - A customer is associated with at most one loan via the relationship *borrower*
  - A loan is associated with at most one customer via *borrower*



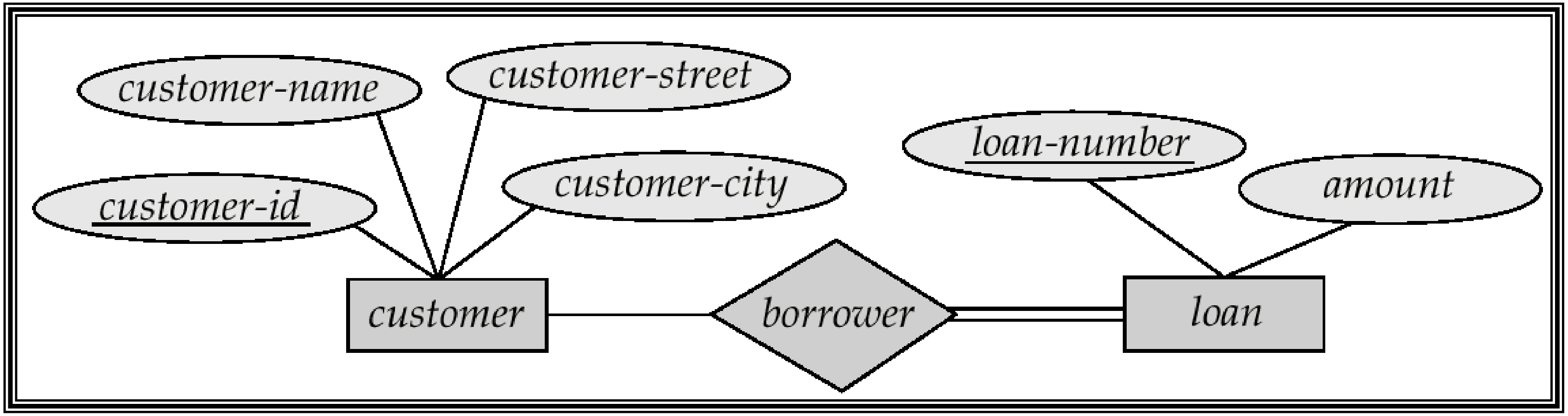
ER: cardinality constraints

- Many-to-one relationship
  - a loan is associated with several (including 0) customers via *borrower*, a customer is associated with at most one loan via *borrower*



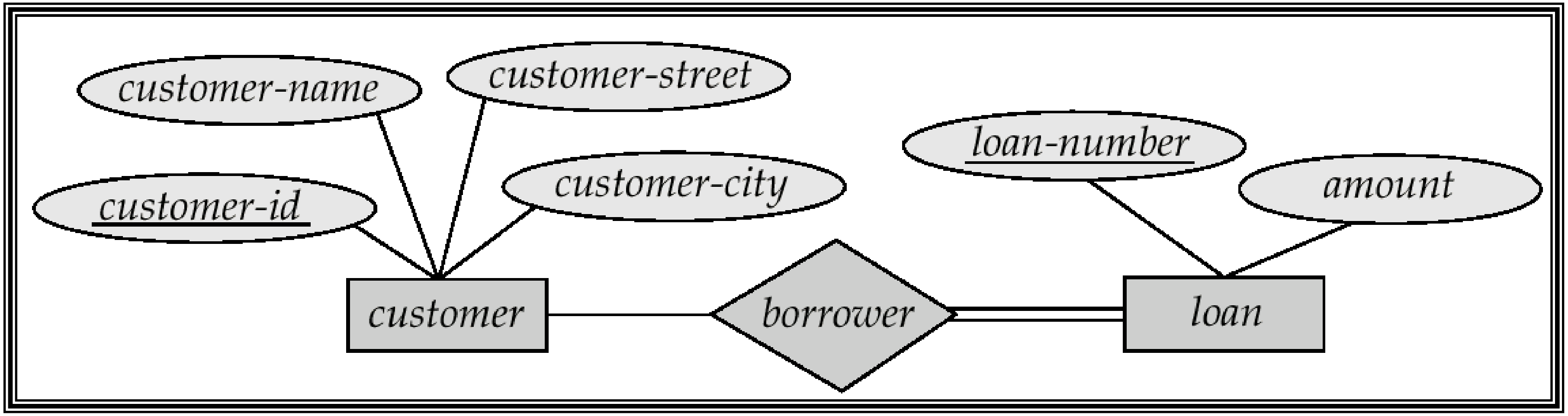
ER: Participation of an Entity Set in a Relationship Set

- Total participation (indicated by double line):
  - every entity in the entity set participates in at least one relationship in the relationship set
  - E.g. participation of *loan* in *borrower* is total
    - every loan must have a customer associated to it via borrower

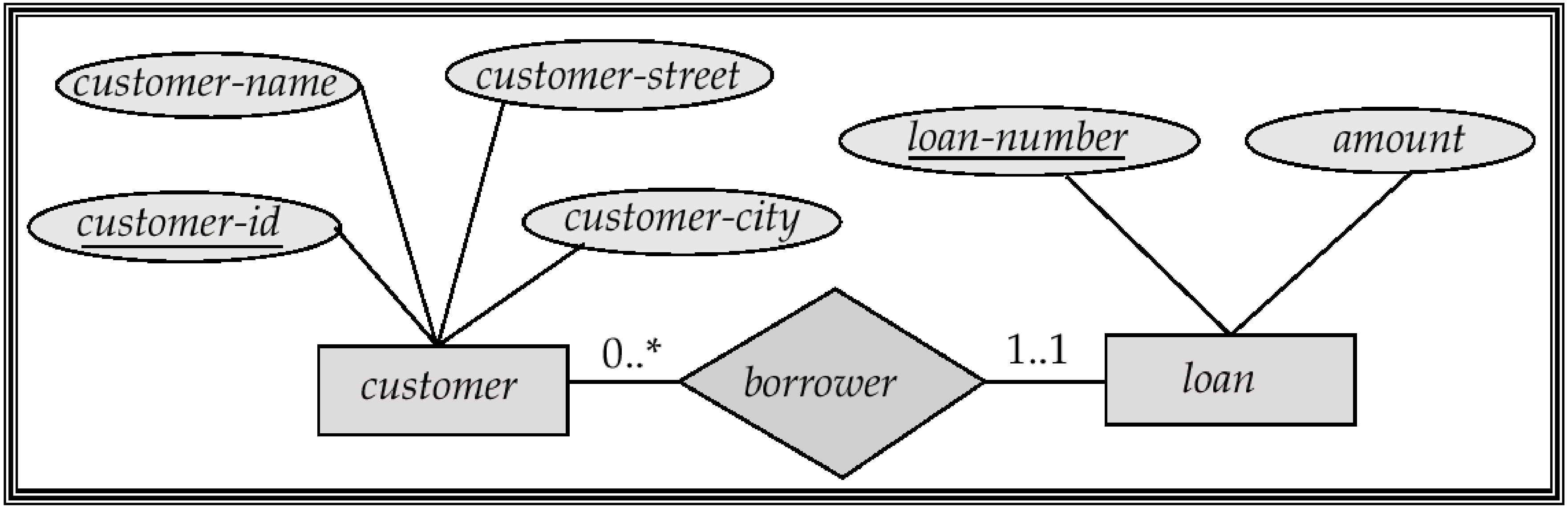


ER: Participation of an Entity Set in a Relationship Set

- Partial participation:
  - some entities may not participate in any relationship in the relationship set
  - E.g. participation of *customer* in *borrower* is partial

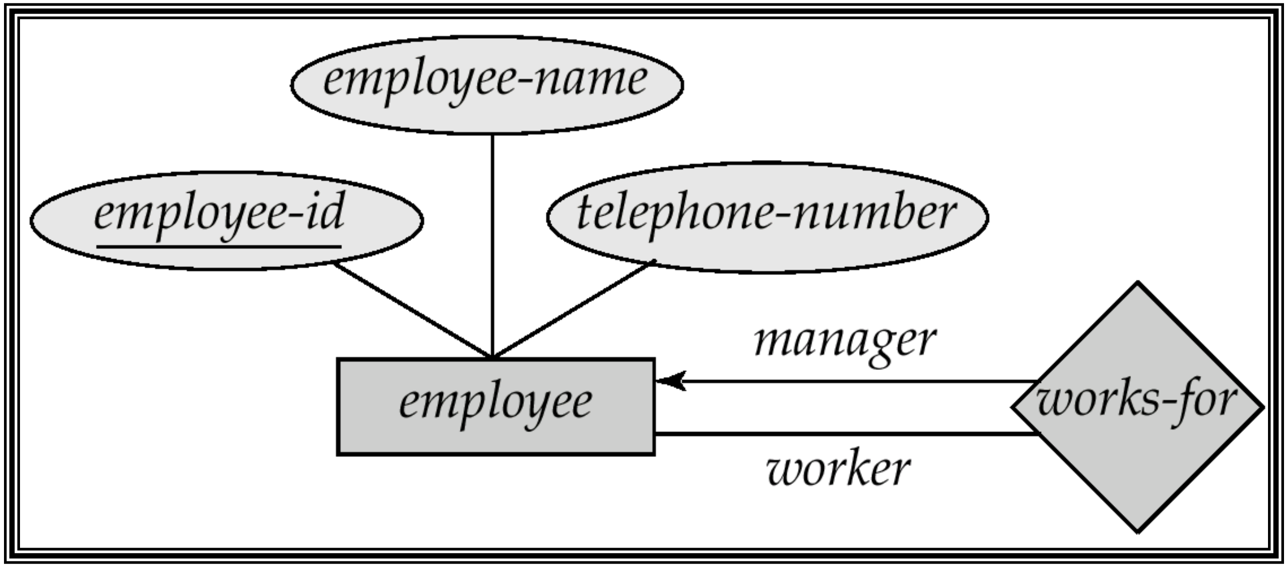


ER: alternative notation for cardinality limits



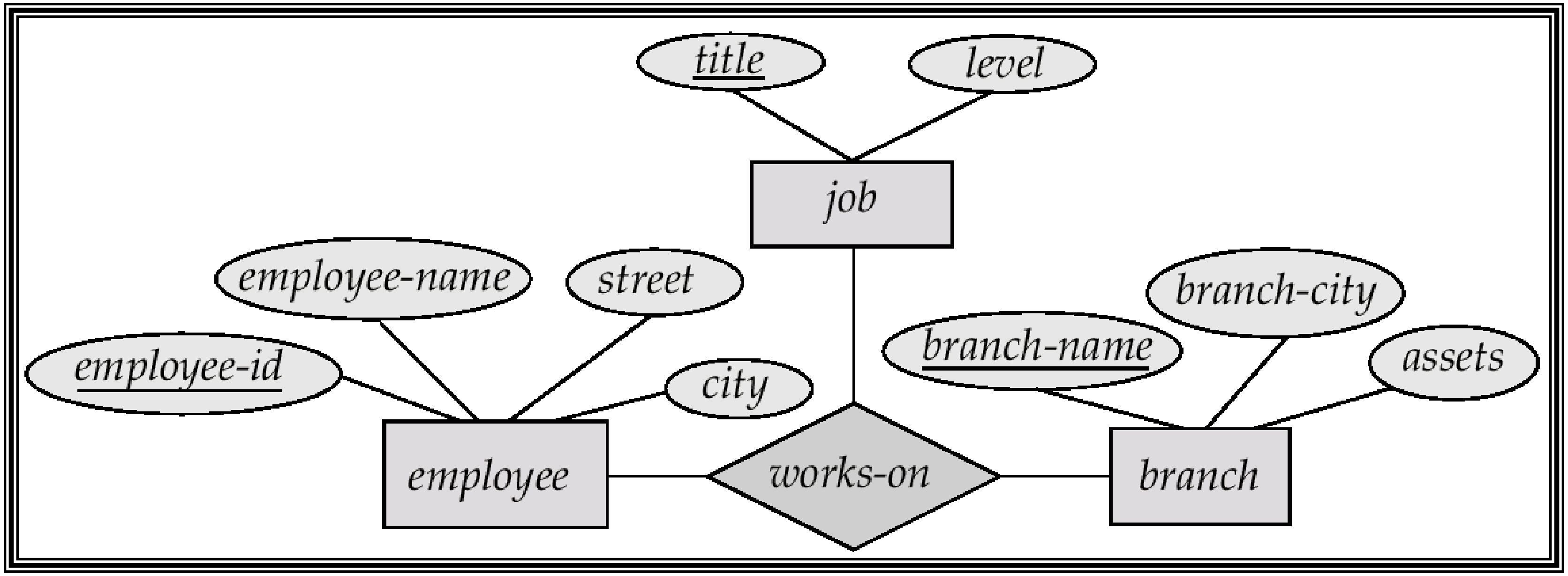
ER: roles

- Entity sets of a relationship need not be distinct
  - The labels "manager" and "worker" are called roles; they specify how employee entities interact via the works-for relationship set.
  - Roles are indicated in E-R diagrams by labeling the lines that connect diamonds to rectangles.
  - Role labels are optional, and are used to clarify semantics of the relationship



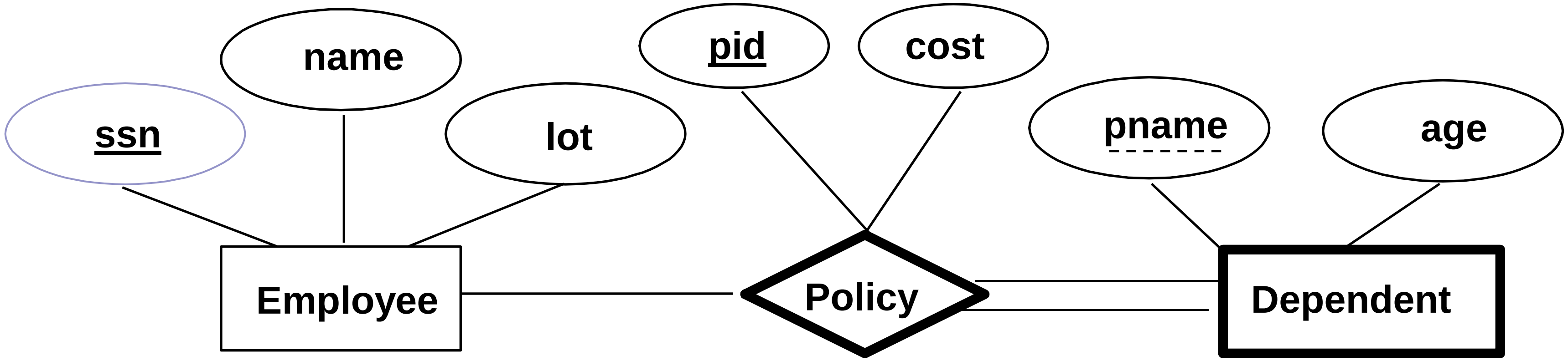
E-R: ternary Relationship

- Suppose employees of a bank may have jobs (responsibilities) at multiple branches, with different jobs at different branches. Then there is a ternary relationship set between entity sets *employee, job and branch*

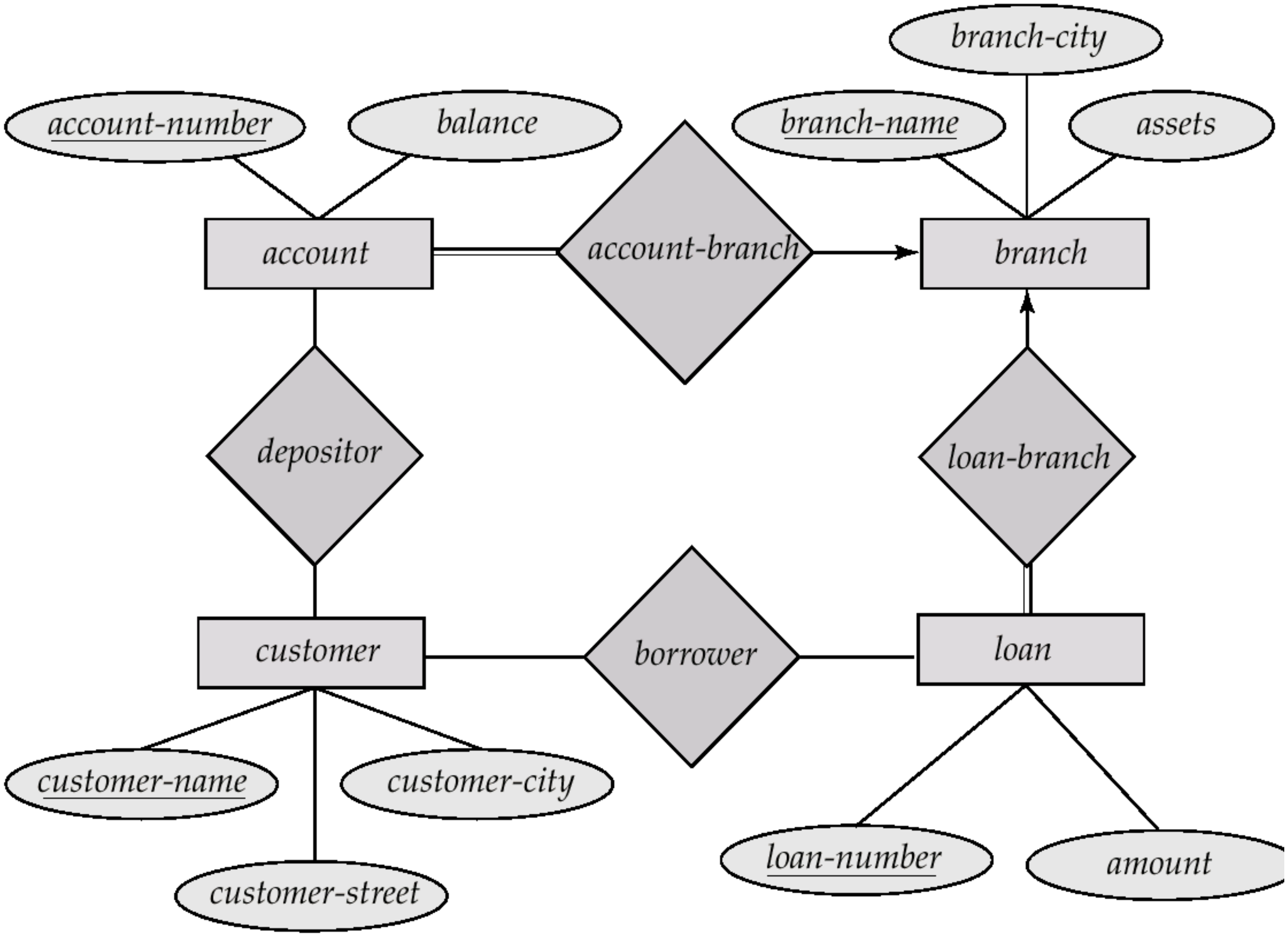


ER: weak entities

- A *weak entity* can be identified uniquely only by considering the primary key of another (*owner*) entity.
  - Owner entity set and weak entity set must participate in a one-to-many relationship set (one owner, many weak entities).
  - Weak entity set must have total participation in this *identifying* relationship set.
  - Weak entities have only a “partial key” (dashed underline)



Example


branch-city
account-number
balance
branch-name
assets
account
account-branch
branch
depositor
loan-branch
customer
borrower
loan
customer-name
customer-city
customer-street
loan-number
amount

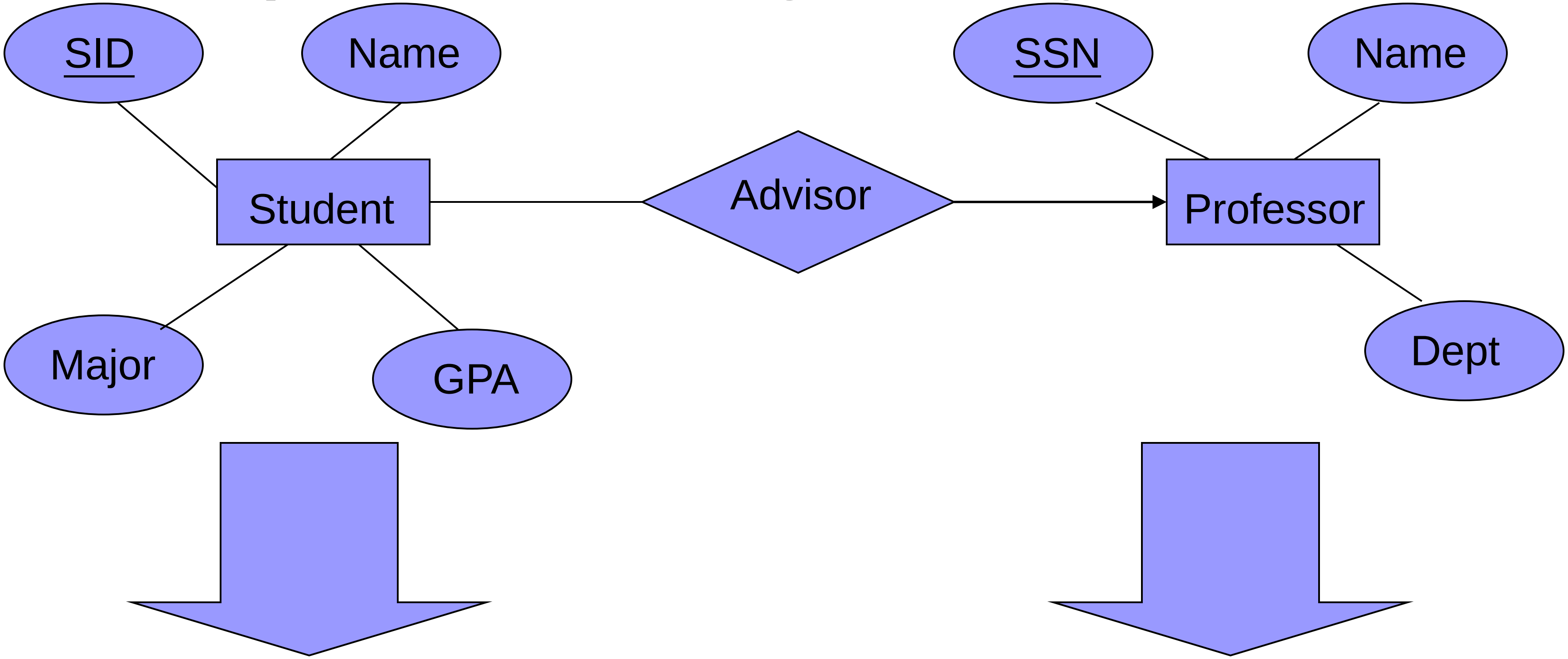
From ER Model to Relational Model

So… how do we convert an ER diagram into a table?? Simple!!

Basic Ideas:

- Build a table for each entity set
- Build a table for each relationship set if necessary (more on this later)
- Make a column in the table for each attribute in the entity set
- Underline Key

Example – Strong Entity Set



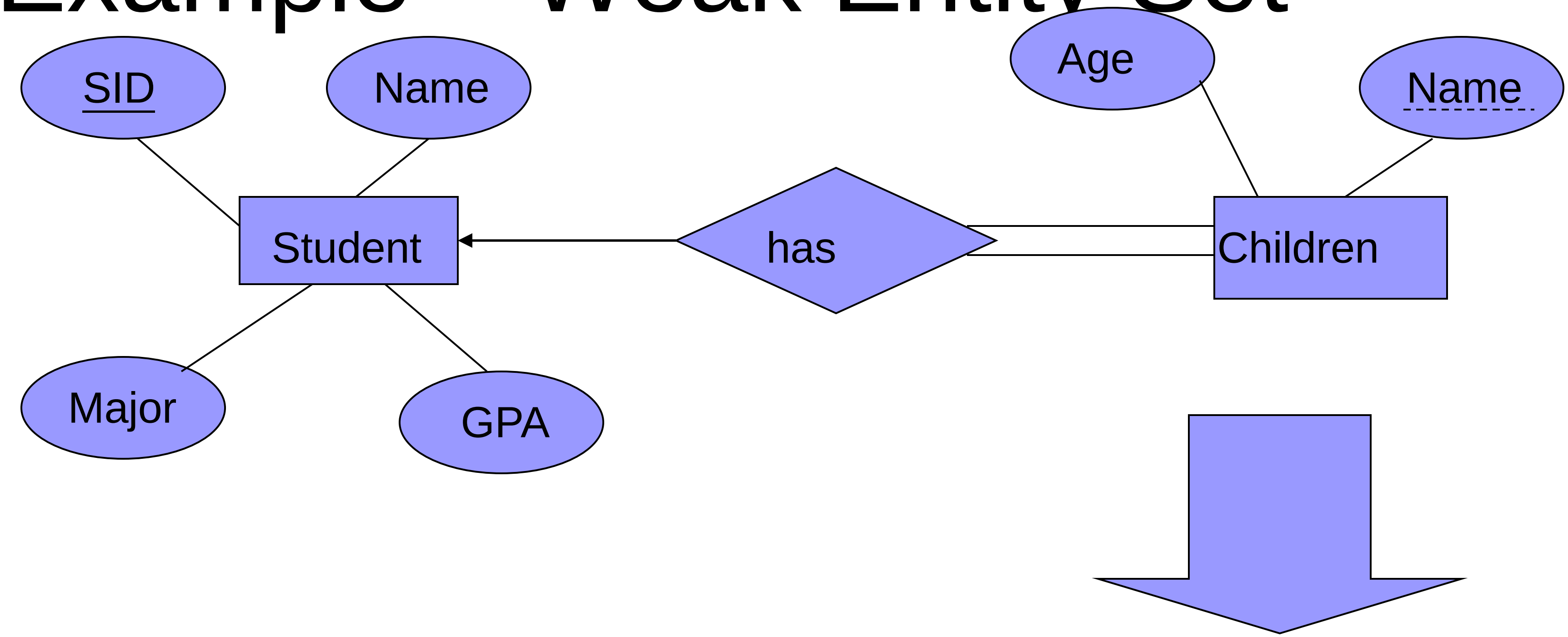
SID	Name	Major	GPA
1234	John	CS	2.8
5678	Mary	EE	3.6

SSN	Name	Dept
9999	Smith	Math
8888	Lee	CS

Representation of Weak Entity Set

- Weak Entity Set Cannot exists alone
- To build a table/schema for weak entity set
  - Construct a table with one column for each attribute in the weak entity set
  - Remember to include discriminator
  - Augment one extra column on the right side of the table, put in there the primary key of the Strong Entity Set (the entity set that the weak entity set is depending on)
  - Primary Key of the weak entity set = Discriminator + foreign key

Example – Weak Entity Set



Age	Name	SID
10	Bart	1234
8	Lisa	5678

* key of *Children* is *Parent_SID* + *Name*

Representation of Relationship Set

--This is a little more complicated--

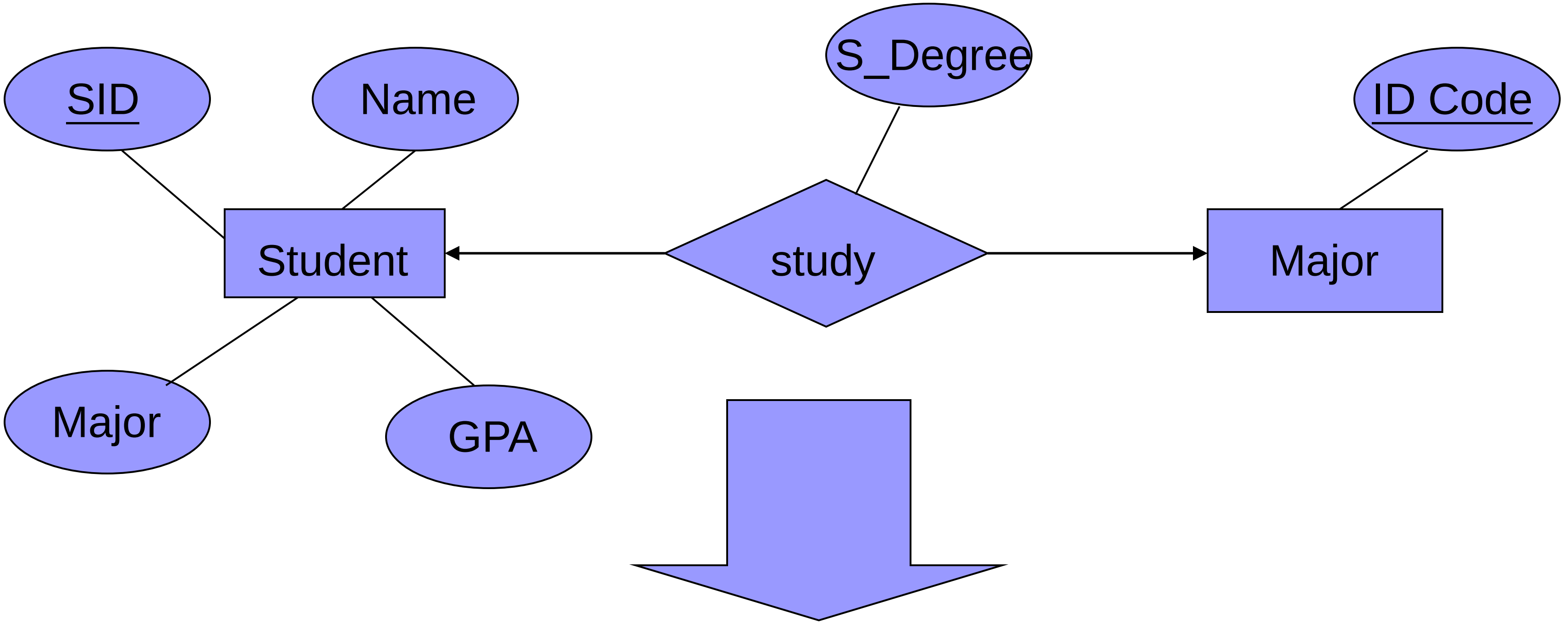
- Unary/Binary Relationship set
  - Depends on the cardinality and participation of the relationship
  - Two possible approaches
- N-ary (multiple) Relationship set
  - Primary Key Issue
- Identifying Relationship
  - No relational model representation necessary

Representing Relationship Set

Unary/Binary Relationship

- For one-to-one relationship w/out total participation
  - Build a table with two columns, one column for each participating entity set's primary key. Add successive columns, one for each descriptive attributes of the relationship set (if any).
- For one-to-one relationship with one entity set having total participation
  - Augment one extra column on the right side of the table of the entity set with total participation, put in there the primary key of the entity set without complete participation as per to the relationship.

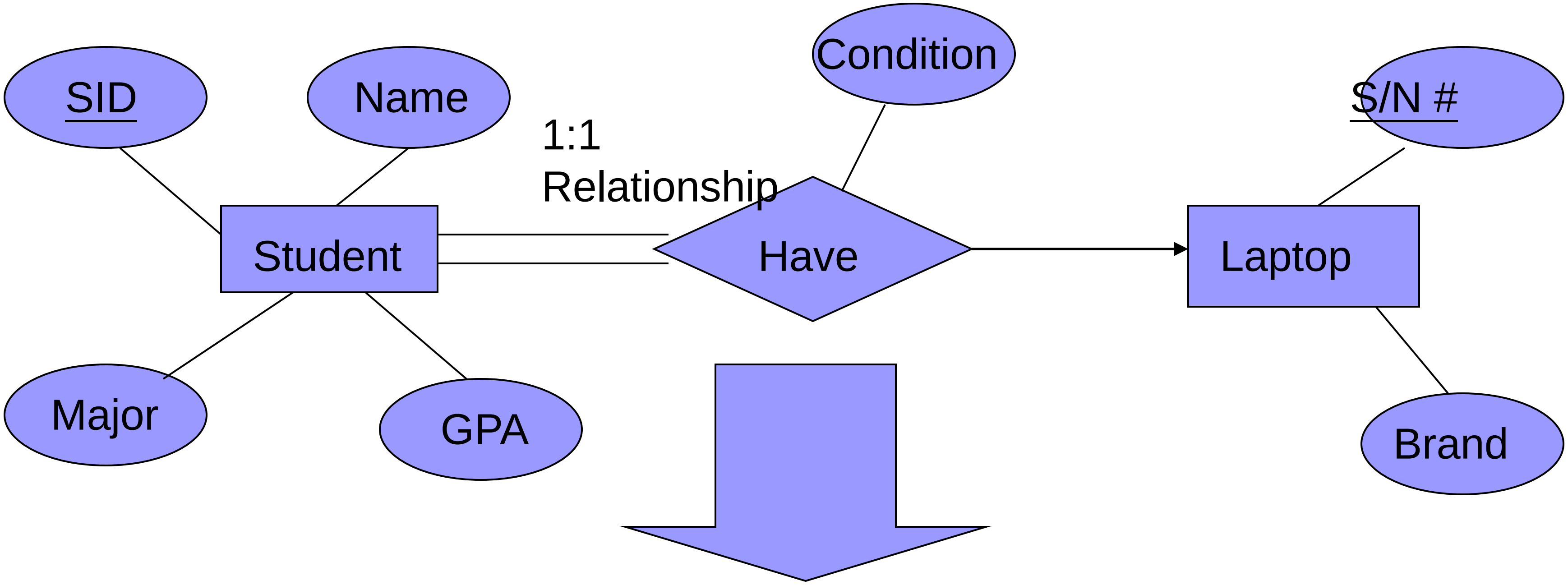
Example – One-to-One Relationship Set



<u>SID</u>	<u>ID Code</u>	S_Degree
9999	07	1234
8888	05	5678

* key can be either *SID* or Maj_*ID_Co*

Example – One-to-One Relationship Set



SID	Name	Major	GPA	LP_S/N	Hav_Cond
9999	Bart	Economy	-4.0	123-456	Own
8888	Lisa	Physics	4.0	567-890	Loan

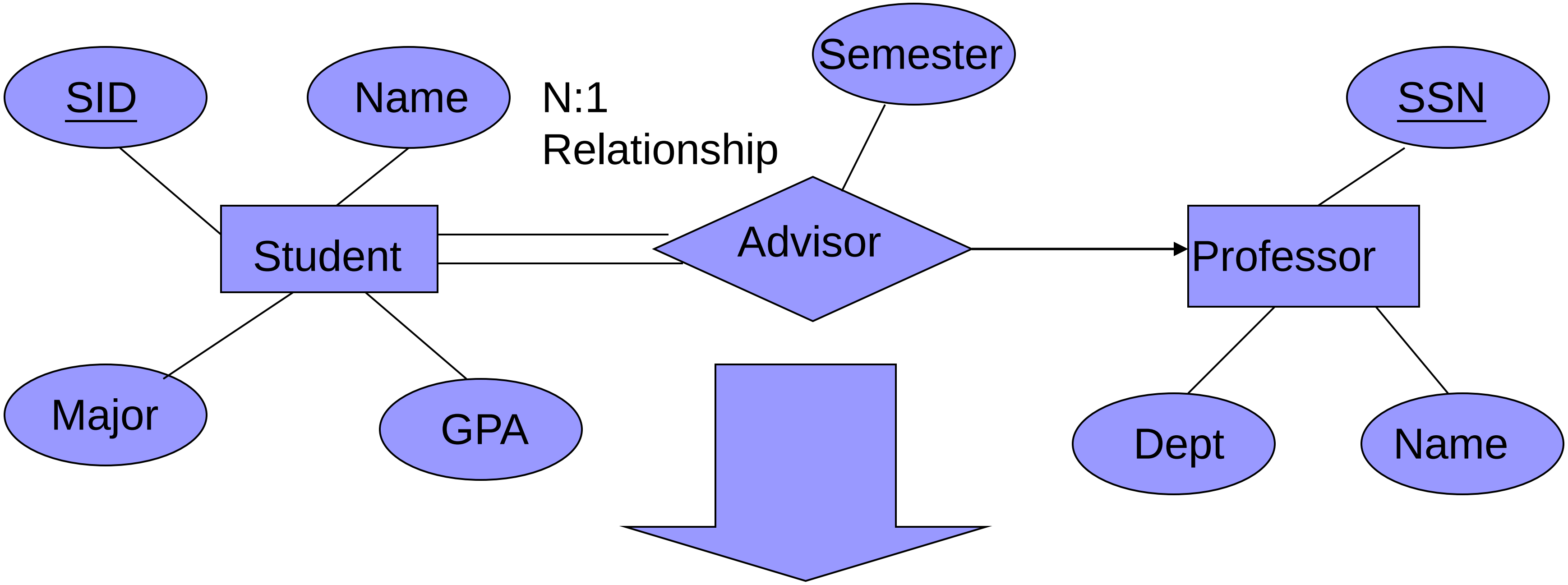
* key can be either *SID* or *LP_S/N*

Representing Relationship Set Unary/Binary Relationship

- For one-to-many relationship w/out total participation
  - Same thing as one-to-one

- For one-to-many/many-to-one relationship with one entity set having total participation on "many" side
  - Augment one extra column on the right side of the table of the entity set on the "many" side, put in there the primary key of the entity set on the "one" side as per to the relationship.

Example – Many-to-One Relationship Set



SID	Name	Major	GPA	SSN	Semester
9999	Bart	Economy	-4.0	123-456	Fall 2006
8888	Lisa	Physics	4.0	567-890	Fall 2005

* Primary key of this table is *SID*

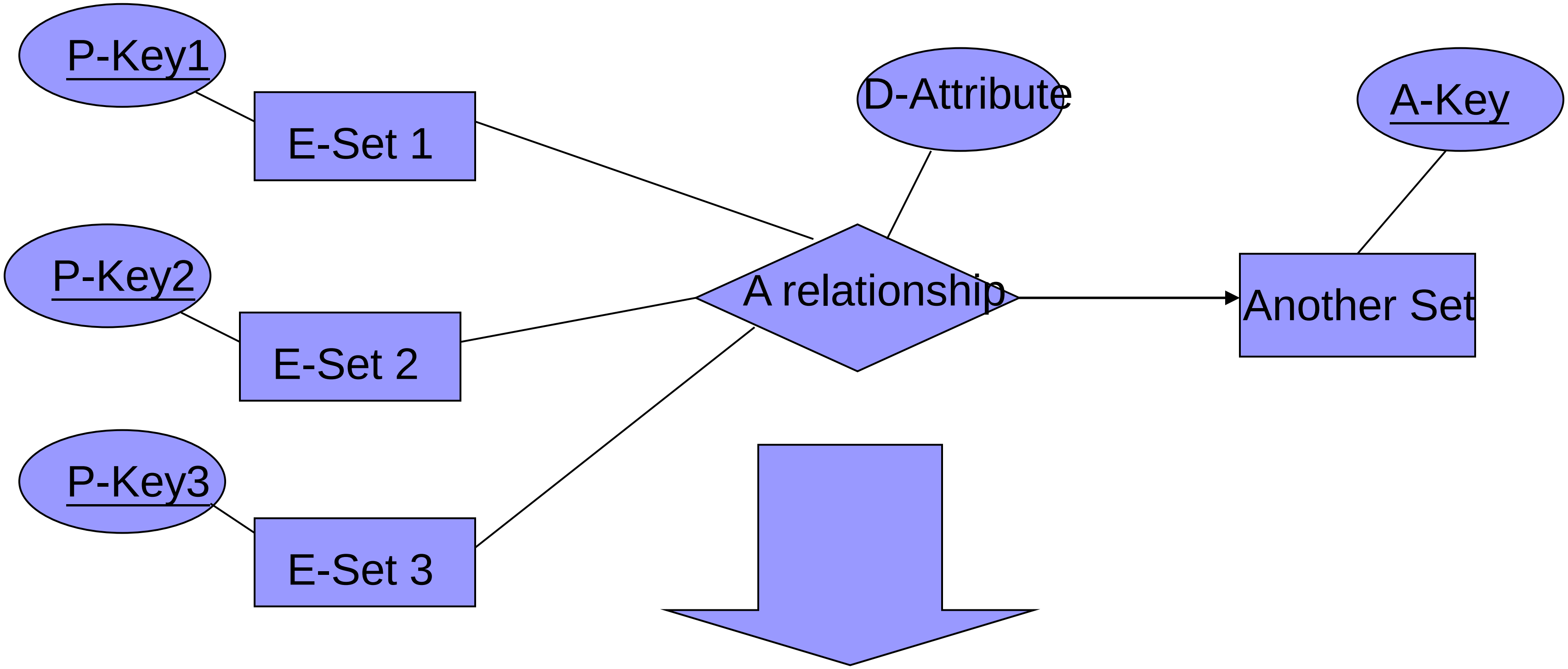
Representing Relationship Set Unary/Binary Relationship

- For many-to-many relationship
  - Same thing as one-to-one relationship without total participation.
  - Primary key of this new schema is the union of the foreign keys of both entity sets.
  - No augmentation approach possible…

Representing Relationship Set N-ary Relationship

- Intuitively Simple
  - Build a new table with as many columns as there are attributes for the union of the primary keys of all participating entity sets.
  - Augment additional columns for descriptive attributes of the relationship set (if necessary)
  - The primary key of this table is the union of all primary keys of entity sets that are on “many” side
  - That is it, we are done.

Example – N-ary Relationship Set

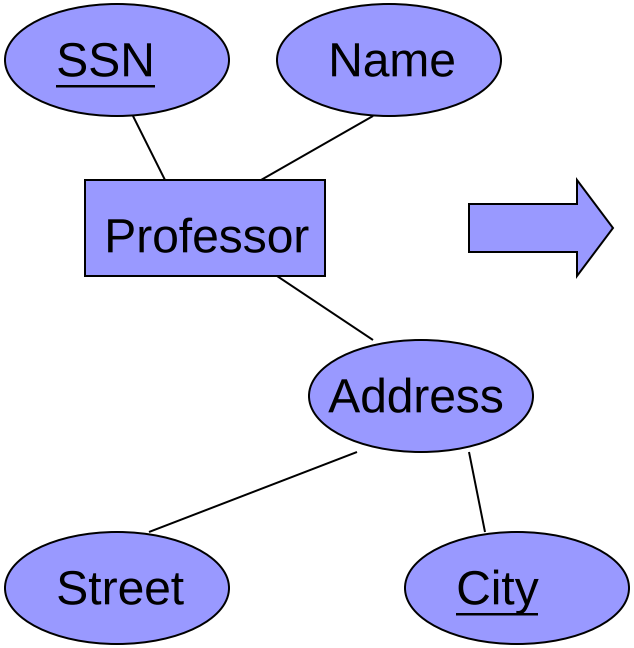


P-Key1	P-Key2	P-Key3	A-Key	D-Attribute
9999	8888	7777	6666	Yes
1234	5678	9012	3456	No

* key of this table is *P-Key1* + *P-Key2* + *P-Key3*

Representing Composite Attribute

- One column for each component attribute
- NO column for the composite attribute itself

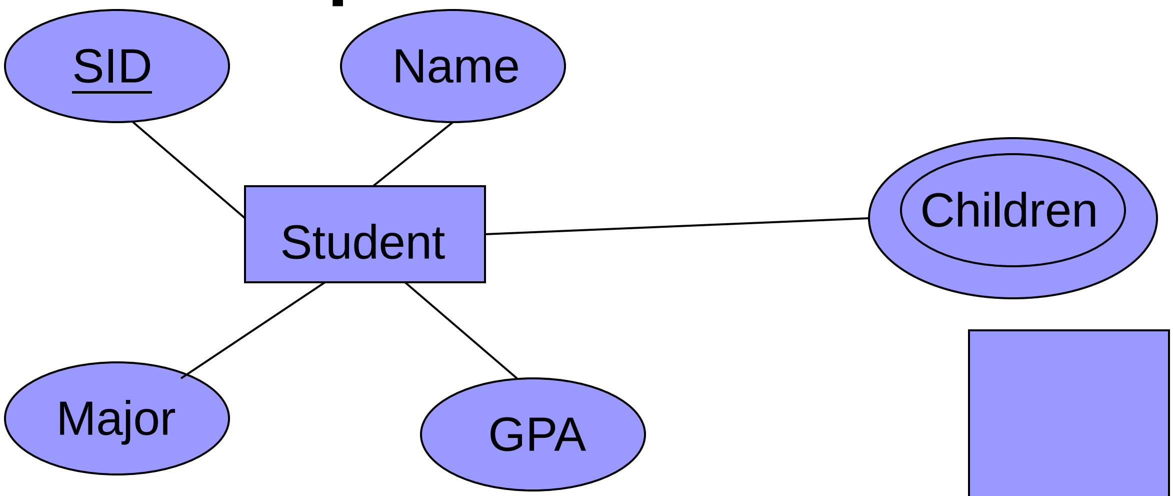


SSN	Name	Street	City
9999	Dr. Smith	50 1st St.	Fake City
8888	Dr. Lee	1 B St.	San Jose

Representing Multivalue Attribute

- For each multivalue attribute in an entity set/relationship set
  - Build a new relation schema with two columns
  - One column for the primary keys of the entity set/relationship set that has the multivalue attribute
  - Another column for the multivalue attributes. Each cell of this column holds only one value. So each value is represented as an unique tuple
  - Primary key for this schema is the union of all attributes

Example – Multivalue attribute



The key for this table is Student_SID + Children, the union of all attributes

SID	Name	Major	GPA
1234	John	CS	2.8
5678	Homer	EE	3.6

Stud_SID	Children
1234	Johnson
1234	Mary
5678	Bart
5678	Lisa
5678	Maggie

SQL: Structure Query Language

Data Definition Language (DDL)	Data Manipulation Language (DML)

Data Definition Language (DDL)

Allows the specification of not only a set of relations but also information about each relation, including:

- The schema for each relation.
- The domain of values associated with each attribute.
- Integrity constraints (what's valid….)
- The set of indices (keys..) to be maintained for each relations.
- Security and authorization information for each relation.
- The physical storage structure of each relation on disk.

Domains

- Domains specify allowable values for attributes.
- Two categories:
  - Elementary (predefined by the standard);
  - User-defined.

Elementary Domains — Character

- Character
  - Single characters or strings;
  - Strings may be of variable length;
  - A Character set different from the default one can be used (e.g., Latin, Greek, Cyrillic, etc.)
  - Syntax:

    character [varying] [(*Length*)] [character set *CharSetName*]
  - It is possible to use **char** and **varchar**, for **character** and **character varying** respectively

More Elementary Domains

- Bit
  - Single Boolean values or strings of Boolean values (may be variable in length);
  - Syntax:

    bit [**varying**] [(*Length*)]

- Exact numeric domains
  - Exact values, integer or with a fractional part
  - Four alternatives: numeric(6,3)

    numeric [(*Precision* [, *Scale*])]
    decimal [(*Precision* [, *Scale*])]
    integer
    smallint

    decimal digits (Scale)
    # of significant digits (Precision)

Approximate Numeric Domains

- Approximate numeric domains
  - Approximate real values
  - Based on a floating point representation

float [(*Precision*)]

double precision

Temporal Instant Domains

- Temporal instants

  date has fields **year,month,day**

  time [(*Precision*)] [**with time zone**]

  has fields **hour,minute,second**

  timestamp [(*Precision*)] [**with time zone**]

- Temporal intervals

  interval *FirstUnitOfTime* [**to** *LastUnitOfTime*]

  - Units of time are divided into two groups:
    - (i) year, month,
    - (ii) day, hour, minute, second
  - For example, **year(5) to month** allows intervals up to **99999yrs + 11mo**

User-Defined Domains

- Comparable to definitions of variable types in programming languages.
- A domain is characterized by name, elementary domain, default value, set of constraints
- Syntax:

create domain *DomainName*

as *ElementaryDomain* [*DefaultValue*] [*Constraints*]

- Example:

```
create domain Mark as smallint default null
```

Default Domain Values

- Define the value that the attribute must assume when a value is not specified during row insertion.
- Syntax:

  `default` < ***GenericValue*** | **`user`** | **`null`** >

- ***GenericValue*** represents a value compatible with the domain, in the form of a constant or an expression.
- **`user`** is the login name of the user who assigns a value to this attribute.

Summary: domain types in SQL

- **char(n).** Fixed length character string, with user-specified length *n.*
- **varchar(n).** Variable length character strings, with user-specified maximum length *n.*
- **int.** Integer (a finite subset of the integers that is machine-dependent).
- **smallint.** Small integer (a machine-dependent subset of the integer domain type).
- **numeric(p,d).** Fixed point number, with user-specified precision of *p* digits, with *n* digits to the right of decimal point.
- **real, double precision.** Floating point and double-precision floating point numbers, with machine-dependent precision.
- **float(n).** Floating point number, with user-specified precision of at least *n* digits.
- Null values are allowed in all the domain types. Declaring an attribute to be **not null** prohibits null values for that attribute.
- **create domain** construct in SQL-92 creates user-defined domain types

Summary: domain types in SQL (cont.)

- **date.** Dates, containing a (4 digit) year, month and date
  - E.g. **date** ‘2001-7-27’
- **time.** Time of day, in hours, minutes and seconds.
  - E.g. **time** ’09:00:30’ **time** ’09:00:30.75’
- **timestamp**: date plus time of day
  - E.g. **timestamp** ‘2001-7-27 09:00:30.75’
- **Interval**: period of time
  - E.g. Interval ‘1’ day
  - Subtracting a date/time/timestamp value from another gives an interval value
  - Interval values can be added to date/time/timestamp values

Schema Definition

- A schema is a collection of objects: domains, tables, indexes, assertions, views, privileges
- A schema has a name and an owner (who determines authorization privileges)
- Syntax:

  create schema [*SchemaName*]

  [[**authorization**] *Authorization*] { *SchemaElementDefinition* }

Table Definition

- An SQL table consists of an ordered set of attributes, and a (possibly empty) set of constraints
- Statement **`create table`** defines a relation schema, creating an empty instance.
- Syntax:

  `create table` *TableName*
  (*AttributeName Domain* [*DefaultValue*] [*Constraints*]
  {, *AttributeName Domain* [*DefaultValue*] [*Constraints*] }
  [*OtherConstraints*])

Example Database

EMPLOYEE

FirstName	Surname	Dept	Office	Salary	City
Mary	Brown	Administration	10	45	London
Charles	White	Production	20	36	Toulouse
Gus	Green	Administration	20	40	Oxford
Jackson	Neri	Distribution	16	45	Dover
Charles	Brown	Planning	14	80	London
Laurence	Chen	Planning	7	73	Worthing
Pauline	Bradshaw	Administration	75	40	Brighton
Alice	Jackson	Production	20	46	Toulouse

DEPARTMENT

DeptName	Address	City
Administration	Bond Street	London
Production	Rue Victor Hugo	Toulouse
Distribution	Pond Road	Brighton
Planning	Bond Street	London
Research	Sunset Street	San José

Example of `create table`

```
Employee:
    RegNo      is 6 characters
    FirstName  is 20 characters
    Surname    is 20 characters
    Salary     is 9 numeric
    City       is 15 characters
```

EMPLOYEE

FirstName	Surname	Dept	RegNo	Salar	City
Mary	Brown	Administration	10	45	London
Charles	White	Production	20	36	Toulouse
Gus	Green	Administration	20	40	Oxford
Jackson	Neri	Distribution	16	45	Dover
Charles	Brown	Planning	14	80	London
Laurence	Chen	Planning	7	73	Worthing
Pauline	Bradshaw	Administration	75	40	Brighton
Alice	Jackson	Production	20	46	Toulouse

Example of `create table`

```
create table Employee
  (
     RegNo     character(6),
     FirstName character(20),
     Surname   character(20),
     Salary    numeric(9),
     City      character(15)
  );
```

EMPLOYEE

FirstName	Surname	Dept	RegNo	Salar	City
Mary	Brown	Administration	10	45	London
Charles	White	Production	20	36	Toulouse
Gus	Green	Administration	20	40	Oxford
Jackson	Neri	Distribution	16	45	Dover
Charles	Brown	Planning	14	80	London
Laurence	Chen	Planning	7	73	Worthing
Pauline	Bradshaw	Administration	75	40	Brighton
Alice	Jackson	Production	20	46	Toulouse

Intra-Relational Constraints

- Constraints are conditions that must be verified by every database instance
- Intra-relational constraints involve a single relation
  - **`not null`** (on single attributes)
  - **`unique`**: permits the definition of keys; syntax:
    - for single attributes: **`unique`**, after the domain
    - for multiple: **`unique (`** ***Attribute*** **{,** ***Attribute*** **}** **`)`**
  - **`primary key`**: defines the primary key (once for each table; implies **not null**); syntax like **`unique`**
  - `check`: described later

Example of Intra-Relational Constraints

- Each pair of **FirstName** and **Surname** uniquely identifies each element

```
FirstName char(20) not null,
Surname char(20) not null,
unique(FirstName,Surname)
```

Example

```
create table Employee
(
    RegNo char(6),
    FirstName char(20) not null,
    Surname char(20) not null,
    Dept char(15),
    Salary numeric(9) default 0,
    City char(15),
    primary key(RegNo),
    foreign key(Dept) references Department(DeptName),
    unique(FirstName,Surname)
);
```

Inter-Relational Constraints

Constraints may involve several relations:

- **check**: checks whether an assertion is true;
- **references** and **foreign key** permit the definition of referential integrity constraints;
  - Syntax for single attributes
    references after the domain
  - Syntax for multiple attributes
    foreign key (***Attribute*** **{,** ***Attribute*** **})**
    references ...
- It is possible to associate reaction policies to violations of referential integrity constraints.

Reaction Policies

Violations arise from

- (a) **updates** on referred attribute or
- (b) row **deletions**.

Reactions operate on internal table, after changes to an external table.

- Reactions are:
  - **`cascade`**: propagate the change;
  - **`set null`**: nullify the referring attribute;
  - **`set default`**: assign default value to the referring attribute;
  - **`no action`**: forbid the change on external table.
- Reactions may depend on the event; syntax:

```
on < delete |update >
      <cascade|set null|set default|no action>
```

Note

- “Correct” policy is a design decision
- E.g., what does it mean if a creditcard goes away? What if a creditcard account changes its number?

Example

```
create table Employee
(
    RegNo char(6),
    FirstName char(20) not null,
    Surname char(20) not null,
    Dept char(15),
    Salary numeric(9) default 0,
    City char(15),
    primary key(RegNo),
    foreign key(Dept)
        references Department(DeptName)
        on delete set null
        on update cascade,
    unique(FirstName,Surname)
);
```

Database Management System (DBMS)

- A collection of programs that enable:

  Defining (describing the structure),

  → Populating by data (Constructing),

  - Manipulating (querying, updating),
  - Preserving consistency,
  - Protecting from misuse,
  - Recovering from failure, and
  - Concurrent using

  of a database.

Modification of the Database – Insertion

- Add a new tuple to *account table*

  insert into *account*
  values (‘A-9732’, ‘Perryridge’,1200);

- *or equivalently*
  insert into *account (branch-name, balance, account-number)*
  values (‘Perryridge’, 1200, ‘A-9732’)

- Add a new tuple to *account* with *balance* set to null

  insert into *account*
  values (‘A-777’,‘Perryridge’, *null*)

Modification of the Database – Updates

- Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.

  Write two **update** statements:

```
update account
set balance = balance * 1.06
where balance > 10000;

update account
set balance = balance * 1.05
where balance ≤ 10000;
```

Drop and Alter Table – cont'd

- Examples:
  - ```
    alter table Department
        add column NoOfOffices numeric(4);
    ```
  - ```
    drop table Department cascade;
    ```

Database Management System (DBMS)

- A collection of programs that enable:

  Defining (describing the structure),

  Populating by data (Constructing),

  → Manipulating (querying, updating),

  - Preserving consistency,
  - Protecting from misuse,
  - Recovering from failure, and
  - Concurrent using

  of a database.

Cross Product

sid	bid	day
22	101	10/10/96
58	103	11/12/96

R1

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

S1

S1 X R1 =

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	11/12/96
58	rusty	10	35.0	22	101	10/10/96
58	rusty	10	35.0	58	103	11/12/96

Natural Join

sid	bid	day
22	101	10/10/96
58	103	11/12/96

R1

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

S1

S1 $\bowtie$ R1 =

sid	sname	rating	age	bid	day
22	dustin	7	45.0	101	10/10/96
58	rusty	10	35.0	103	11/12/96

Outer Join

- Relation *loan*

loan-number	branch-name	amount
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

- Relation borrower

customer-name	loan-number
Jones	L-170
Smith	L-230
Hayes	L-155

Outer Join – Example

- **Inner Join:** *loan* ⋈ *Borrower*

loan-number	branch-name	amount	customer-name
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

- **Left Outer Join:** loan ⟕ Borrower

loan-number	branch-name	amount	customer-name
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	null

Outer Join

- **Right Outer Join :** *loan ⟖ borrower*

loan-number	branch-name	amount	customer-name
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	null	null	Hayes

- **Full Outer Join**

  loan ⟗ borrower

loan-number	branch-name	amount	customer-name
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	null
L-155	null	null	Hayes

SQL Query

- The generic query:

select $T_1.Attr_{11}$, …, $T_h.Attr_{hm}$
from $Table_1\ T_1$, …, $Table_n\ T_n$
where *Condition*

Example Database

EMPLOYEE

FirstName	Surname	Dept	Office	Salary	City
Mary	Brown	Administration	10	45	London
Charles	White	Production	20	36	Toulouse
Gus	Green	Administration	20	40	Oxford
Jackson	Neri	Distribution	16	45	Dover
Charles	Brown	Planning	14	80	London
Laurence	Chen	Planning	7	73	Worthing
Pauline	Bradshaw	Administration	75	40	Brighton
Alice	Jackson	Production	20	46	Toulouse

DEPARTMENT

DeptName	Address	City
Administration	Bond Street	London
Production	Rue Victor Hugo	Toulouse
Distribution	Pond Road	Brighton
Planning	Bond Street	London
Research	Sunset Street	San José

* in the Target List

FirstName	Surname	Dept	RegNo	Salar	City
Mary	Brown	Administration	10	45	London
Charles	White	Production	20	36	Toulouse
Gus	Green	Administration	20	40	Oxford
Jackson	Neri	Distribution	16	45	Dover
Charles	Brown	Planning	14	80	London
Laurence	Chen	Planning	7	73	Worthing
Pauline	Bradshaw	Administration	75	40	Brighton
Alice	Jackson	Production	20	46	Toulouse

- "Find all the information relating to employees named Brown":

```
select *
from Employee
where Surname = ‘Brown’;
```

- Result:

FirstName	Surname	Dept	Office	Salary	City
Mary	Brown	Administration	10	45	London
Charles	Brown	Planning	14	80	London

Predicate Conjunction

FirstName	Surname	Dept	RegNo	Salar	City
Mary	Brown	Administration	10	45	London
Charles	White	Production	20	36	Toulouse
Gus	Green	Administration	20	40	Oxford
Jackson	Neri	Distribution	16	45	Dover
Charles	Brown	Planning	14	80	London
Laurence	Chen	Planning	7	73	Worthing
Pauline	Bradshaw	Administration	75	40	Brighton
Alice	Jackson	Production	20	46	Toulouse

- "Find the first names and surnames of employees who work in office number 20 of the Administration department":

```
select FirstName, Surname
from Employee
where Office = ‘20’ and
      Dept = ‘Administration’
```

- Result:

FirstName	Surname
Gus	Green

Predicate Disjunction

FirstName	Surname	Dept	RegNo	Salar	City
Mary	Brown	Administration	10	45	London
Charles	White	Production	20	36	Toulouse
Gus	Green	Administration	20	40	Oxford
Jackson	Neri	Distribution	16	45	Dover
Charles	Brown	Planning	14	80	London
Laurence	Chen	Planning	7	73	Worthing
Pauline	Bradshaw	Administration	75	40	Brighton
Alice	Jackson	Production	20	46	Toulouse

- "Find the first names and surnames of employees who work in either the Administration or the Production department":

```
select FirstName, Surname
from Employee
where Dept = ‘Administration’ or
      Dept = ‘Production’
```

- Result:

FirstName	Surname
Mary	Brown
Charles	White
Gus	Green
Pauline	Bradshaw
Alice	Jackson

Complex Logical Expressions

FirstName	Surname	Dept	RegNo	Salar	City
Mary	Brown	Administration	10	45	London
Charles	White	Production	20	36	Toulouse
Gus	Green	Administration	20	40	Oxford
Jackson	Neri	Distribution	16	45	Dover
Charles	Brown	Planning	14	80	London
Laurence	Chen	Planning	7	73	Worthing
Pauline	Bradshaw	Administration	75	40	Brighton
Alice	Jackson	Production	20	46	Toulouse

- "Find the first names of employees named Brown who work in the Administration department or the Production department":

```
select FirstName
from Employee
where Surname = ‘Brown’ and
      (Dept = ‘Administration’ or
       Dept = ‘Production’)
```

- Result:

FirstName
Mary

Another Example: Drivers and Cars

DRIVER

FirstName	Surname	DriverID
Mary	Brown	VR 2030020Y
Charles	White	PZ 1012436B
Marco	Neri	AP 4544442R

AUTOMOBILE

CarRegNo	Make	Model	DriverID
ABC 123	BMW	323	VR 2030020Y
DEF 456	BMW	Z3	VR 2030020Y
GHI 789	Lancia	Delta	PZ 1012436B
BBB 421	BMW	316	MI 2020030U

Left Join

- "Find all drivers and their cars, if any":

```
select FirstName,Surname,
  Driver.DriverID,CarRegNo,Make,Model
from Driver left join Automobile on
  (Driver.DriverID = Automobile.DriverID)
```

- Result:

FirstName	Surname	DriverID	CarRegNo	Make	Model
Mary	Brown	VR 2030020Y	ABC 123	BMW	323
Mary	Brown	VR 2030020Y	DEF 456	BMW	Z3
Charles	White	PZ 1012436B	GHI 789	Lancia	Delta
Marco	Neri	AP 4544442R	NULL	NULL	NULL

Full Join

- "Find all possible drivers and their cars":

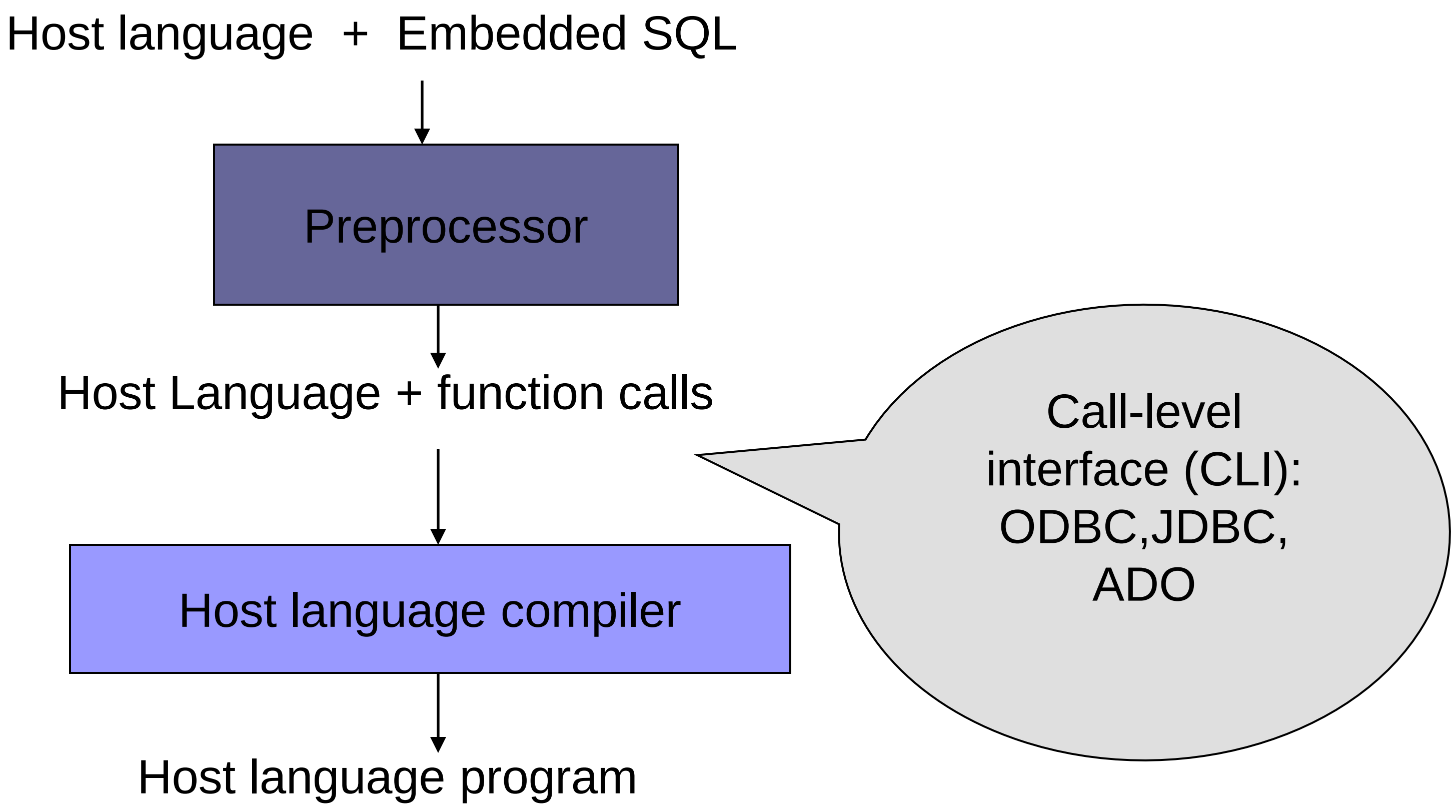
```
select FirstName,Surname,Driver.DriverID
     CarRegNo, Make, Model
from Driver full join Automobile on
  (Driver.DriverID = Automobile.DriverID)
```

- Result:

FirstName	Surname	DriverID	CarRegNo	Make	Model
Mary	Brown	VR 2030020Y	ABC 123	BMW	323
Mary	Brown	VR 2030020Y	DEF 456	BMW	Z3
Charles	White	PZ 1012436B	GHI 789	Lancia	Delta
Marco	Neri	AP 4544442R	NULL	NULL	NULL
NULL	NULL	NULL	BBB 421	BMW	316

JDBC

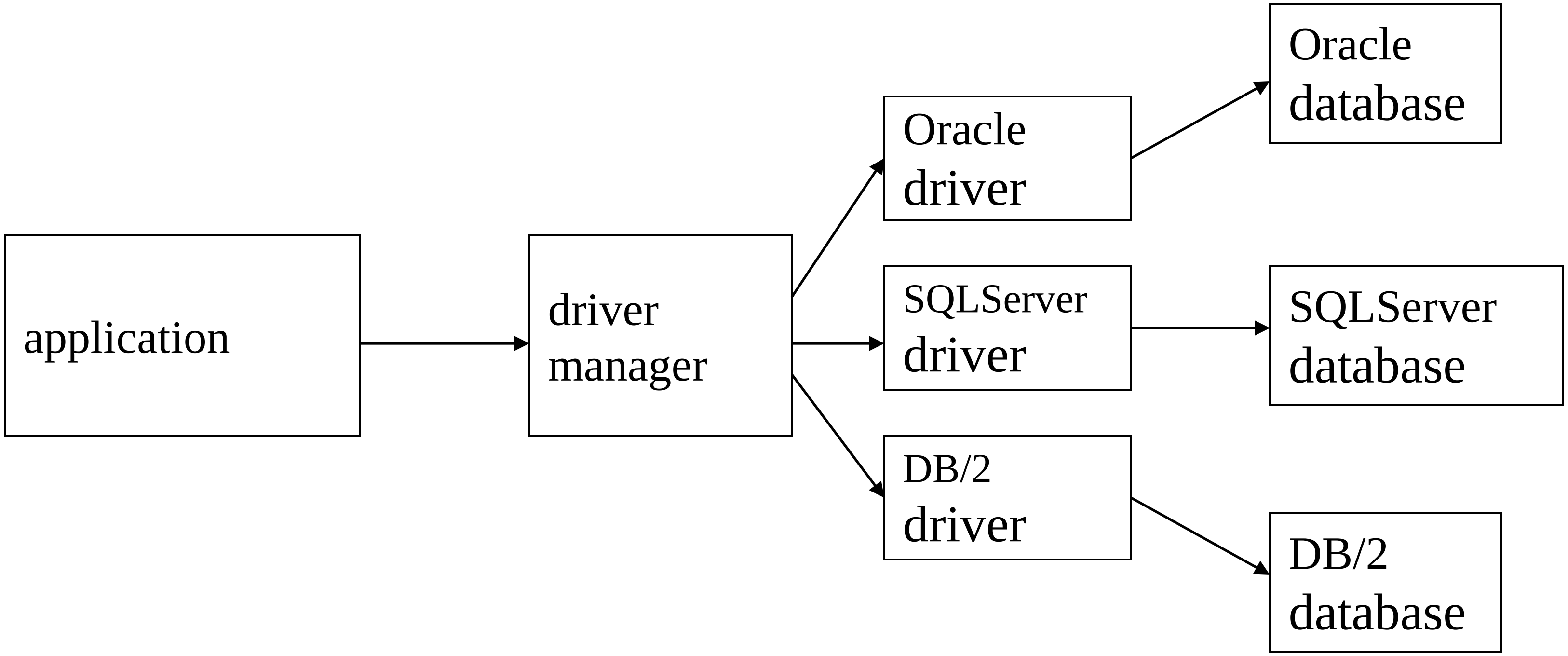
Programs with Embedded SQL



JDBC

- Call-level interface (CLI) for executing SQL from a Java program
- SQL statement is constructed at run time as the value of a Java variable (as in dynamic SQL)
- JDBC passes SQL statements to the underlying DBMS. Can be interfaced to any DBMS that has a JDBC driver
- Part of SQL:2003

JDBC Run-Time Architecture



Steps to execute queries using JDBC

1. Register Oracle Driver

   ```
   DriverManager.registerDriver(new oracle.jdbc.OracleDriver())
   ```

2. Establish connection to DB server

   ```
   Connection con =
   DriverManager.getConnection(<url>,<username>,<password>);
   ```

   <url> identifies which Oracle Driver to use, connect to which database, on which port and what is the service name.

3. Create Statement

   ```
   Statement sta = con.createStatement();
   ```

Steps to execute queries using JDBC (contd..)

4. Execute Query

```
ResultSet query = sta.executeQuery(<Query>);
```

5. Display/Process Result

```
while(query.next()) {
//process data from tuples.
}
```

6. Close connection

```
query.close();
sta.close();
con.close();
```

Executing a Query

import java.sql.*; -- *import all classes in package* java.sql

Class.forName (*driver name*); // *static method of class* Class
// *loads specified driver*

Connection con = DriverManager.getConnection(*Url*, *Id*, *Passwd*);

- *Static method of class* DriverManager; *attempts to connect to DBMS*
- *If successful, creates a connection object*, con, *for managing the connection*

Statement stat = con.createStatement ();

- *Creates a statement object* stat
- *Statements have* executeQuery() *method*

Executing a Query (cont'd)

```
String query = "SELECT   T.StudId  FROM  Transcript T" +
                         "WHERE   T.CrsCode = 'cse305' " +
                         "AND  T.Semester = 'S2000' ";
ResultSet  res = stat.executeQuery (query);
```

- *Creates a result set object*, res.
- *Prepares and executes the query.*
- *Stores the result set produced by execution in* res *(analogous to opening a cursor).*
- *The query string can be constructed at run time (as above).*
- *The input parameters are plugged into the query when the string is formed (as above)*

Preparing and Executing a Query

```
String query = "SELECT  T.StudId  FROM  Transcript T" +
                    "WHERE  T.CrsCode = ?  AND  T.Semester = ?";
```

placeholders

```
PreparedStatement  ps = con.prepareStatement ( query );
```

- *Prepares the statement*
- *Creates a prepared statement object*, ps, *containing the prepared statement*
- **Placeholders** (**?**) *mark positions of* **in** *parameters; special API is provided to plug the actual values in positions indicated by the* **?***'s*

Preparing and Executing a Query (cont'd)

```
String  crs_code, semester;
………
ps.setString(1, crs_code);     // set value of first in parameter
ps.setString(2, semester);     // set value of second in parameter

ResultSet  res = ps.executeQuery ( );
```

- *Creates a result set object*, res
- *Executes the query*
- *Stores the result set produced by execution in* res

```
while  ( res.next ( ) )  {                  // advance the cursor
    j = res.getInt ("StudId");              // fetch output int-value
    …process output value…
}
```

Result Sets and Cursors

- Three types of result sets in JDBC:
  - *Forward-only*: not scrollable
  - *Scroll-insensitive*: scrollable; changes made to underlying tables after the creation of the result set are not visible through that result set
  - *Scroll-sensitive*: scrollable; updates and deletes made to tuples in the underlying tables after the creation of the result set are visible through the set

Result Set

```
Statement stat = con.createStatement (
    ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_UPDATABLE );
```

- Any result set type can be declared *read-only* or *updatable* – CONCUR_UPDATABLE (assuming SQL query satisfies the conditions for updatable views)
- *Updatable*: Current row of an updatable result set can be changed or deleted, or a new row can be inserted. Any such change causes changes to the underlying database table

```
res.updateString ("Name", "John" );  // change the attribute "Name" of
                                     // current row in the row buffer.
res.updateRow ( );   // install changes to the current row buffer
                     // in the underlying database table
```

Handling Exceptions

```
try {
    ...Java/JDBC code...
}  catch ( SQLException  ex ) {
    …exception handling code...
}
```

- try/catch is the basic structure within which an SQL statement should be embedded
- If an exception is thrown, an exception object, *ex*, is created and the catch clause is executed
- The exception object has methods to print an error message, return SQLSTATE, etc.

JMS

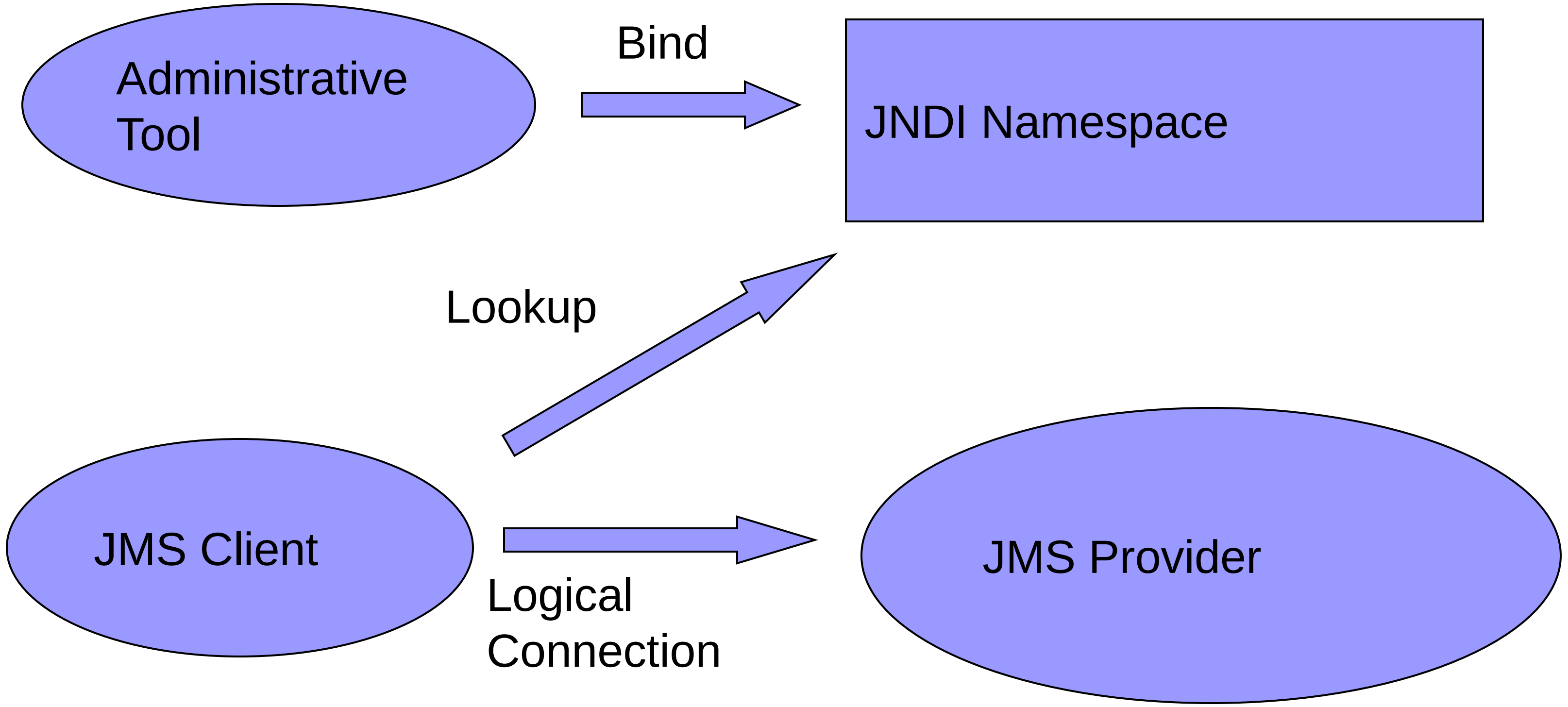
What is JMS?

- A **specification** that describes a common way for Java programs to create, send, receive and read distributed enterprise messages
- *loosely coupled* communication
- *Asynchronous* messaging
- *Reliable* delivery
  - A message is guaranteed to be delivered once and only once.
- Outside the specification
  - Security services
  - Management services

A JMS Application

- JMS Clients
  - Java programs that send/receive messages
- Messages
- Administered Objects
  - preconfigured JMS objects created by an admin for the use of clients
  - ConnectionFactory, Destination (queue or topic)
- JMS Provider
  - messaging system that implements JMS and administrative functionality

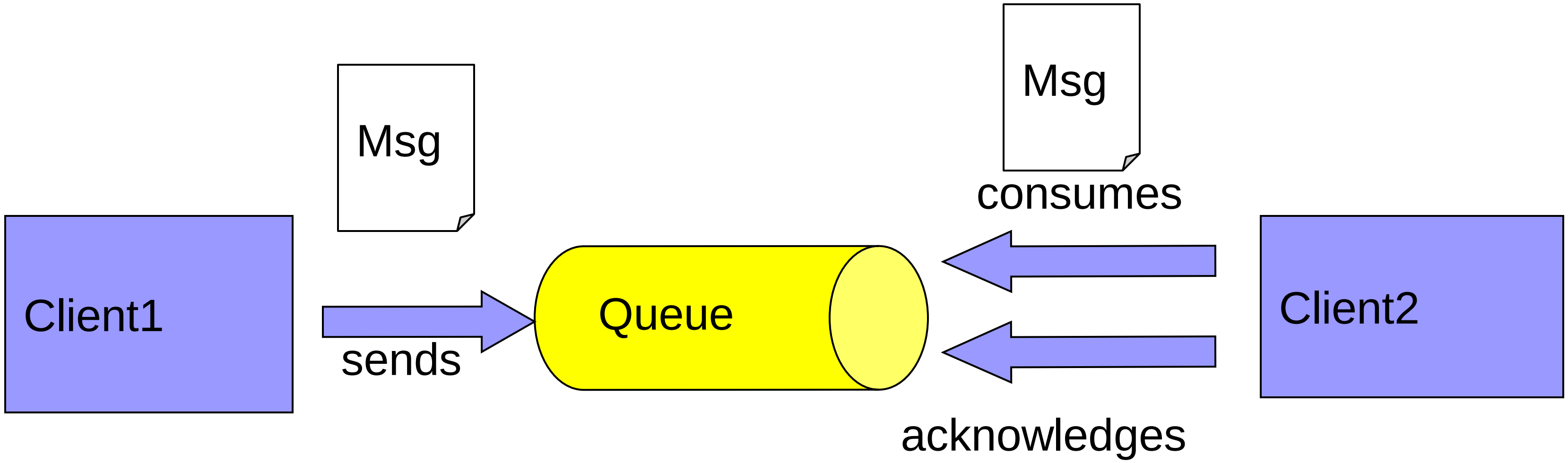
JMS Administration



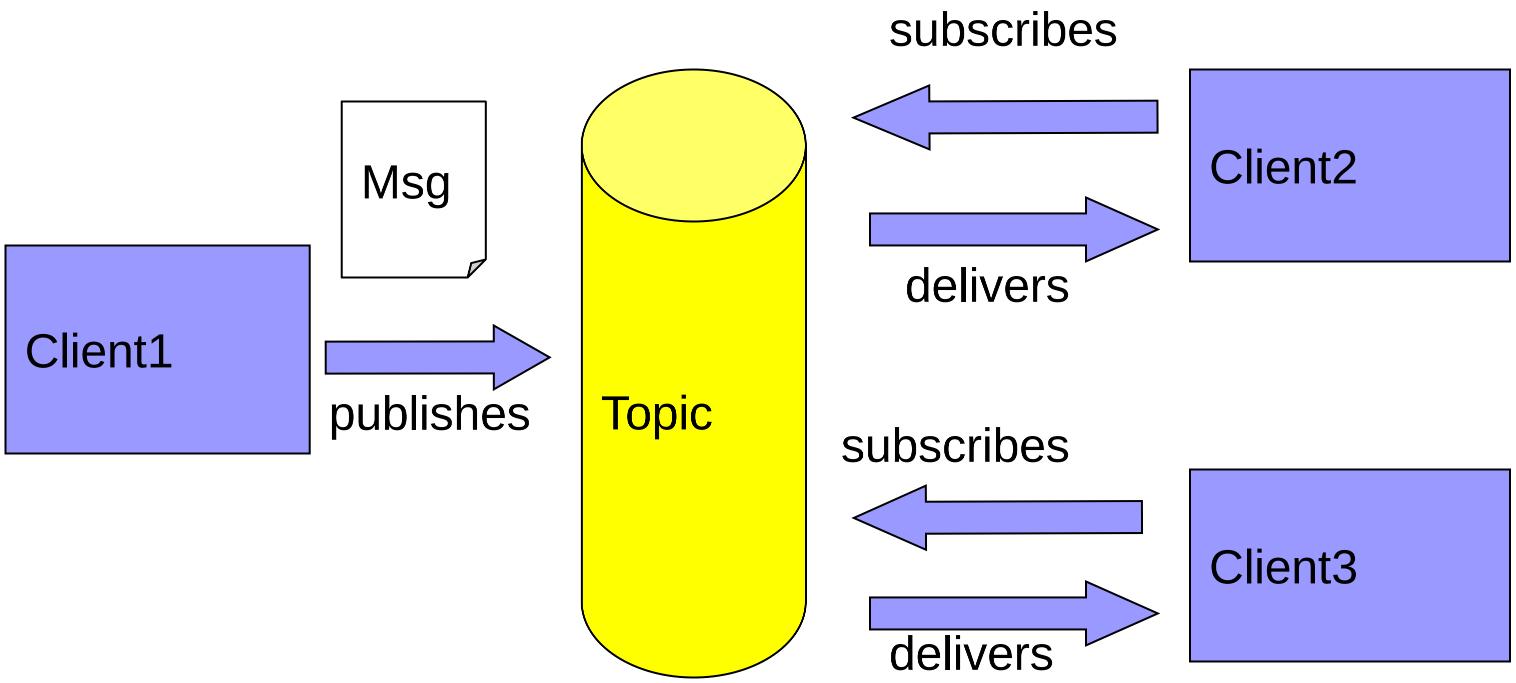
JMS Messaging Domains

- Point-to-Point (PTP)
  - built around the concept of message queues
  - each message has only one consumer
- Publish-Subscribe systems
  - uses a “topic” to send and receive messages
  - each message has multiple consumers

Point-to-Point Messaging



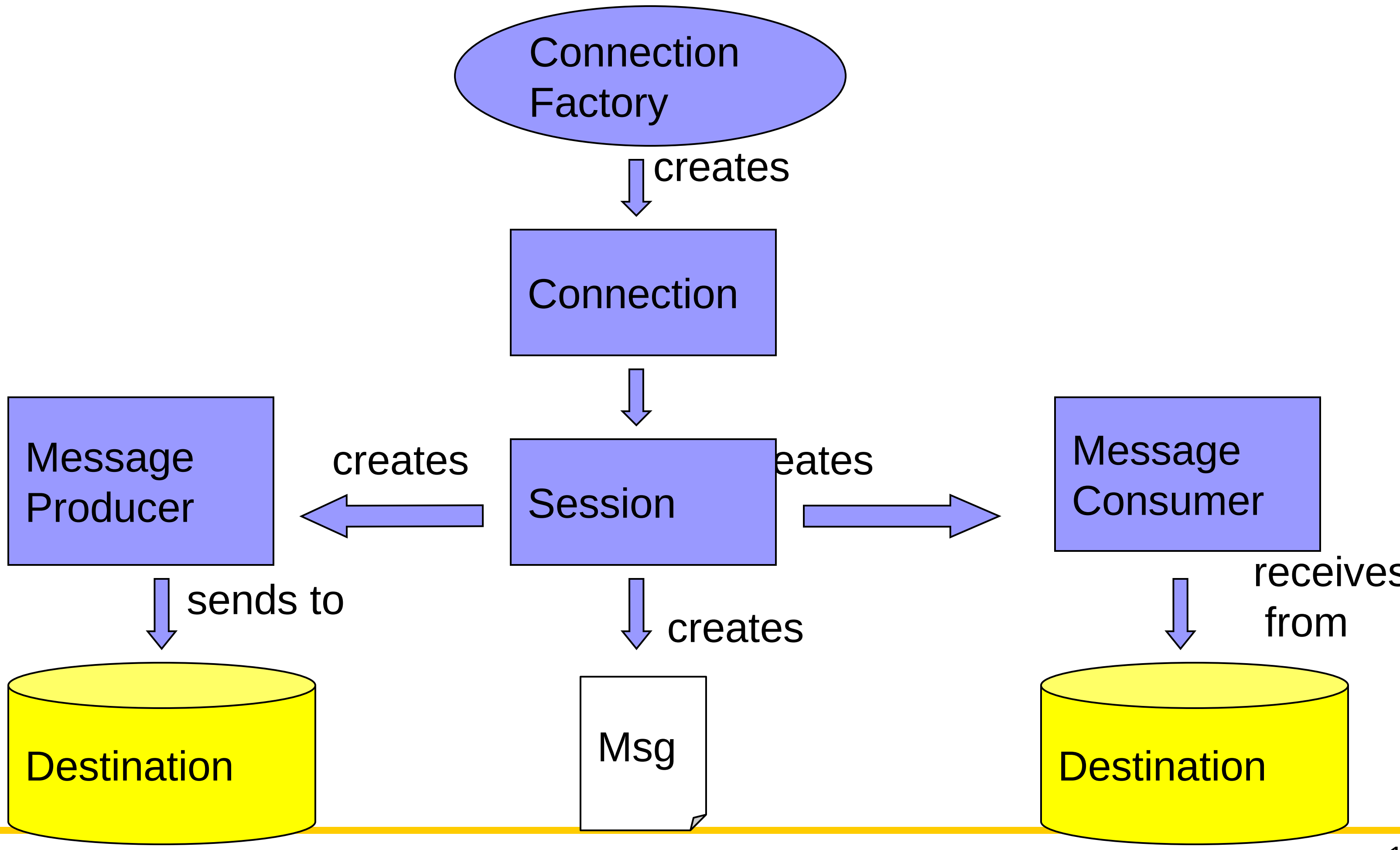
Publish/Subscribe Messaging



Message Consumptions

- Synchronously
  - A subscriber or a receiver explicitly fetches the message from the destination by calling the receive method.
  - The receive method can *block* until a message arrives or can time out if a message does not arrive within a specified time limit.
- Asynchronously
  - A client can register a *message listener* with a consumer.
  - Whenever a message arrives at the destination, the JMS provider delivers the message by calling the listener's onMessage() method.

JMS API Programming Model



JMS Client Example

- Setting up a connection and creating a session

```
InitialContext jndiContext=new InitialContext();
//look up for the connection factory
ConnectionFactory cf=jndiContext.lookup(connectionfactoryname);
//create a connection
Connection connection=cf.createConnection();
//create a session
Session session=connection.createSession(false,Session.AUTO_ACKNOWLEDGE);
//create a destination object
Destination dest1=(Queue) jndiContext.lookup(“/jms/myQueue”); //for PointToPoint
Destination dest2=(Topic)jndiContext.lookup(“/jms/myTopic”); //for publish-subscribe
```

Producer Sample

- Setup connection and create a session
- Creating producer

```
MessageProducer producer=session.createProducer(dest1);
```

- Send a message

```
Message m=session.createTextMessage();
m.setText("just another message");
producer.send(m);
```

- Closing the connection

```
connection.close();
```

Consumer Sample (Synchronous)

- Setup connection and create a session
- Creating consumer

```
MessageConsumer consumer=session.createConsumer(dest1);
```

- Start receiving messages

```
connection.start();
Message m=consumer.receive();
```

Consumer Sample (Asynchronous)

- Setup the connection, create a session
- Create consumer
- Registering the listener
  - MessageListener listener=new myListener();
  - consumer.setMessageListener(listener);
- myListener should have onMessage()

```
public void onMessage(Message msg){
    //read the massage and do computation
}
```

Listener Example

```
public void onMessage(Message message) {
  TextMessage msg = null;
  try {
    if (message instanceof TextMessage) {
      msg = (TextMessage) message;
      System.out.println("Reading message: " + msg.getText());
    } else {
      System.out.println("Message of wrong type: " +
        message.getClass().getName());
    }
  } catch (JMSException e) {
    System.out.println("JMSException in onMessage(): " + e.toString());
  } catch (Throwable t) {
    System.out.println("Exception in onMessage():" + t.getMessage());
  }
}
```

JMS Messages

- Message Header
  - used for identifying and routing messages
  - contains vendor-specified values, but could also contain application-specific data
  - typically name/value pairs
- Message Properties (optional)
- Message Body(optional)
  - contains the data
  - five different message body types in the JMS specification

JMS Message Types

Message Type	Contains	Some Methods
TextMessage	String	getText,setText
MapMessage	set of name/value pairs	setString,setDouble,setLong,getDouble,getString
BytesMessage	stream of uninterpreted bytes	writeBytes,readBytes
StreamMessage	stream of primitive values	writeString,writeDouble, writeLong,readString
ObjectMessage	serialize object	setObject,getObject