# *Database Management Systems*

## *Chapter 1*

Instructor: Raghu Ramakrishnan
raghu@cs.wisc.edu



## *What Is a DBMS?*

- A very large, integrated collection of data.
- Models real-world *enterprise.*
  - Entities (e.g., students, courses)
  - Relationships (e.g., Madonna is taking CS564)
- A *Database Management System (DBMS)* is a software package designed to store and manage databases.



## *Files vs. DBMS*

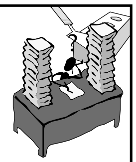
- Application must stage large datasets between main memory and secondary storage (e.g., buffering, page-oriented access, 32-bit addressing, etc.)
- Special code for different queries
- Must protect data from inconsistency due to multiple concurrent users
- Crash recovery
- Security and access control

## *Why Use a DBMS?*

- Data independence and efficient access.
- Reduced application development time.
- Data integrity and security.
- Uniform data administration.
- Concurrent access, recovery from crashes.



## *Why Study Databases??*



- Shift from *<u>computation</u>* to *<u>information</u>*
  - at the "low end": scramble to webspace (a mess!)
  - at the "high end": scientific applications
- Datasets increasing in diversity and volume.
  - Digital libraries, interactive video, Human Genome project, EOS project
  - ... need for DBMS exploding
- DBMS encompasses most of CS
  - OS, languages, theory, "A"I, multimedia, logic

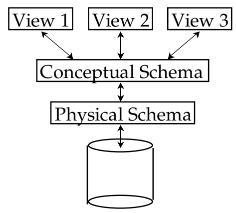


## *Data Models*

- A *<u>data model</u>* is a collection of concepts for describing data.
- A *<u>schema</u>* is a description of a particular collection of data, using the a given data model.
- The *<u>relational model of data</u>* is the most widely used model today.
  - Main concept: *<u>relation</u>*, basically a table with rows and columns.
  - Every relation has a *<u>schema</u>*, which describes the columns, or fields.

## Levels of Abstraction

- Many *views*, single *conceptual (logical) schema* and *physical schema*.
  - Views describe how users see the data.
  - Conceptual schema defines logical structure
  - Physical schema describes the files and indexes used.




* *Schemas are defined using DDL; data is modified/queried using DML.*



## Example: University Database

- Conceptual schema:
  - *Students(sid: string, name: string, login: string, age: integer, gpa:real)*
  - *Courses(cid: string, cname:string, credits:integer)*
  - *Enrolled(sid:string, cid:string, grade:string)*
- Physical schema:
  - Relations stored as unordered files.
  - Index on first column of Students.
- External Schema (View):
  - *Course_info(cid:string,enrollment:integer)*



## Data Independence *

- Applications insulated from how data is structured and stored.
- *Logical data independence*: Protection from changes in *logical* structure of data.
- *Physical data independence*: Protection from changes in *physical* structure of data.

* *One of the most important benefits of using a DBMS!*

## *Concurrency Control*

- Concurrent execution of user programs is essential for good DBMS performance.
  - Because disk accesses are frequent, and relatively slow, it is important to keep the cpu humming by working on several user programs concurrently.
- Interleaving actions of different user programs can lead to inconsistency: e.g., check is cleared while account balance is being computed.
- DBMS ensures such problems don't arise: users can pretend they are using a single-user system.



## *Transaction: An Execution of a DB Program*

- Key concept is ***transaction***, which is an *atomic* sequence of database actions (reads/writes).
- Each transaction, executed completely, must leave the DB in a ***consistent state*** if DB is consistent when the transaction begins.
  - Users can specify some simple ***integrity constraints*** on the data, and the DBMS will enforce these constraints.
  - Beyond this, the DBMS does not really understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).
  - Thus, ensuring that a transaction (run alone) preserves consistency is ultimately the user's responsibility!



## *Scheduling Concurrent Transactions*

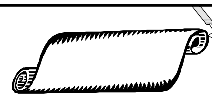
- DBMS ensures that execution of {T1, ... , Tn} is equivalent to some ***serial*** execution T1' ... Tn'.
  - Before reading/writing an object, a transaction requests a lock on the object, and waits till the DBMS gives it the lock. All locks are released at the end of the transaction. (Strict 2PL locking protocol.)
  - Idea: If an action of Ti (say, writing X) affects Tj (which perhaps reads X), one of them, say Ti, will obtain the lock on X first and Tj is forced to wait until Ti completes; this effectively orders the transactions.
  - What if Tj already has a lock on Y and Ti later requests a lock on Y? (Deadlock!) Ti or Tj is aborted and restarted!

## *Ensuring Atomicity*

- DBMS ensures *atomicity* (all-or-nothing property) even if system crashes in the middle of a Xact.
- Idea: Keep a *log* (history) of all actions carried out by the DBMS while executing a set of Xacts:
  - Before a change is made to the database, the corresponding log entry is forced to a safe location. (*WAL protocol*; OS support for this is often inadequate.)
  - After a crash, the effects of partially executed transactions are *undone* using the log. (Thanks to WAL, if log entry wasn't saved before the crash, corresponding change was not applied to database!)



## *The Log*

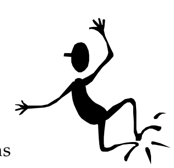


- The following actions are recorded in the log:
  - *Ti writes an object*: the old value and the new value.
    - Log record must go to disk *before* the changed page!
  - *Ti commits/aborts*: a log record indicating this action.
- Log records chained together by Xact id, so it's easy to undo a specific Xact (e.g., to resolve a deadlock).
- Log is often *duplexed* and *archived* on "stable" storage.
- All log related activities (and in fact, all CC related activities such as lock/unlock, dealing with deadlocks etc.) are handled transparently by the DBMS.



## *Databases make these folks happy ...*

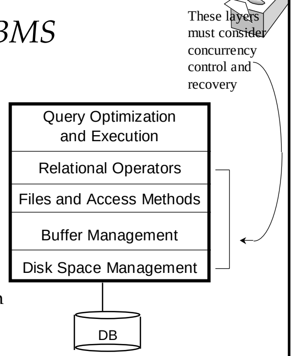
- End users and DBMS vendors
- DB application programmers
  - E.g. smart webmasters
- *Database administrator (DBA)*
  - Designs logical /physical schemas
  - Handles security and authorization
  - Data availability, crash recovery
  - Database tuning as needs evolve



*Must understand how a DBMS works!*

## *Structure of a DBMS*

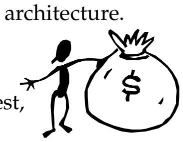
- A typical DBMS has a layered architecture.
- The figure does not show the concurrency control and recovery components.
- This is one of several possible architectures; each system has its own variations.






## *Summary*

- DBMS used to maintain, query large datasets.
- Benefits include recovery from system crashes, concurrent access, quick application development, data integrity and security.
- Levels of abstraction give data independence.
- A DBMS typically has a layered architecture.
- DBAs hold responsible jobs and are well-paid!
- DBMS R&D is one of the broadest, most exciting areas in CS.



# *The Entity-Relationship Model*

Chapter 2

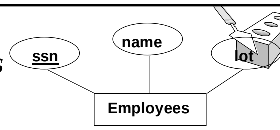


## *Overview of Database Design*

- <u>*Conceptual design*</u>: *(ER Model is used at this stage.)*
  - What are the *entities* and *relationships* in the enterprise?
  - What information about these entities and relationships should we store in the database?
  - What are the *integrity constraints* or *business rules* that hold?
  - A database `schema' in the ER Model can be represented pictorially (*ER diagrams*).
  - Can map an ER diagram into a relational schema.

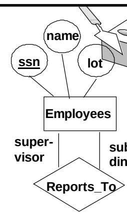


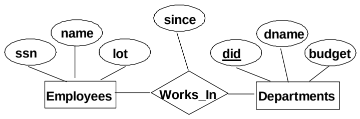
## *ER Model Basics*




- <u>*Entity*</u>: Real-world object distinguishable from other objects. An entity is described (in DB) using a set of <u>*attributes*</u>.
- <u>*Entity Set*</u>: A collection of similar entities. E.g., all employees.
  - All entities in an entity set have the same set of attributes. (Until we consider ISA hierarchies, anyway!)
  - Each entity set has a *key*.
  - Each attribute has a *domain*.

## *ER Model Basics (Contd.)*

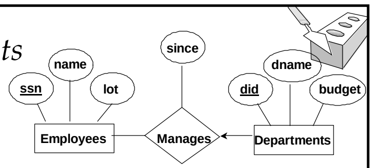


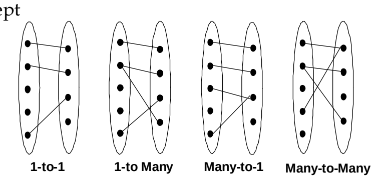

- *Relationship*: Association among two or more entities. E.g., Attishoo works in Pharmacy department.
- *Relationship Set*: Collection of similar relationships.
  - An n-ary relationship set R relates n entity sets E1 ... En; each relationship in R involves entities e1 E1, ..., en En
    - Same entity set could participate in different relationship sets, or in different "roles" in same set.



## *Key Constraints*



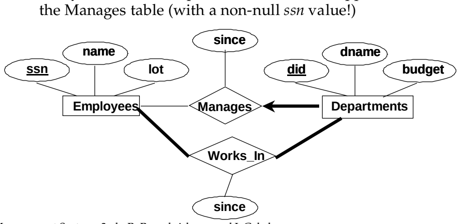

- Consider Works_In: An employee can work in many departments; a dept can have many employees.
- In contrast, each dept has at most one manager, according to the *key constraint* on Manages.





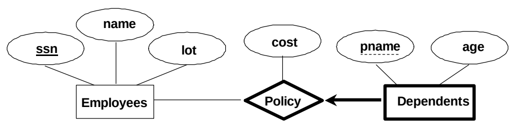

## *Participation Constraints*

- Does every department have a manager?
  - If so, this is a *participation constraint*: the participation of Departments in Manages is said to be *total* (vs. *partial*).
    - Every *did* value in Departments table must appear in a row of the Manages table (with a non-null *ssn* value!)



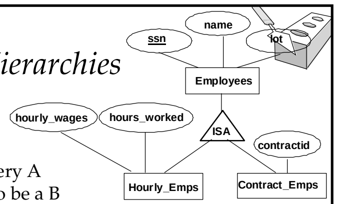
## *Weak Entities*

- 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.






## *ISA (`is a') Hierarchies*

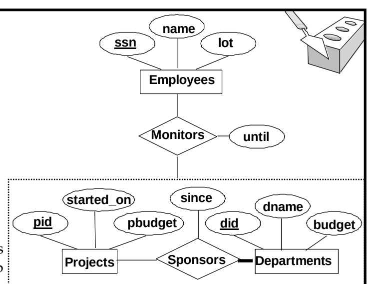



- As in C++, or other PLs, attributes are inherited.
- If we declare A **ISA** B, every A entity is also considered to be a B entity.
- *Overlap constraints*: Can Joe be an Hourly_Emps as well as a Contract_Emps entity? (*Allowed/disallowed*)
- *Covering constraints*: Does every Employees entity also have to be an Hourly_Emps or a Contract_Emps entity? *(Yes/no)*
- Reasons for using ISA:
  - To add descriptive attributes specific to a subclass.
  - To identify entitities that participate in a relationship.



## *Aggregation*

- Used when we have to model a relationship involving (entitity sets and) a *relationship set*.
  - *Aggregation* allows us to treat a relationship set as an entity set for purposes of participation in (other) relationships.




* *Aggregation vs. ternary relationship*:
- Monitors is a distinct relationship, with a descriptive attribute.
- Also, can say that each sponsorship is monitored by at most one employee.

## *Conceptual Design Using the ER Model*

- <u>Design choices:</u>
  - Should a concept be modeled as an entity or an attribute?
  - Should a concept be modeled as an entity or a relationship?
  - Identifying relationships: Binary or ternary? Aggregation?
- Constraints in the ER Model:
  - A lot of data semantics can (and should) be captured.
  - But some constraints cannot be captured in ER diagrams.



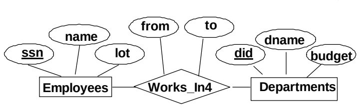
## *Entity vs. Attribute*

- Should *address* be an attribute of Employees or an entity (connected to Employees by a relationship)?
- Depends upon the use we want to make of address information, and the semantics of the data:
  - If we have several addresses per employee, *address* must be an entity (since attributes cannot be set-valued).
  - If the structure (city, street, etc.) is important, e.g., we want to retrieve employees in a given city, *address* must be modeled as an entity (since attribute values are atomic).

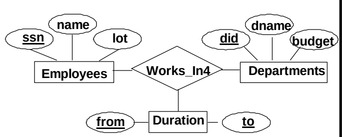


## *Entity vs. Attribute (Contd.)*

- Works_In4 does not allow an employee to work in a department for two or more periods.

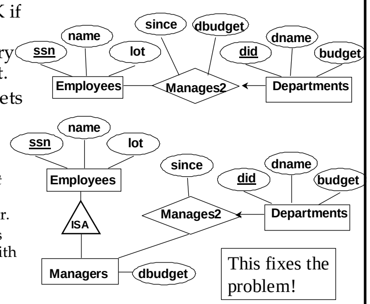



- Similar to the problem of wanting to record several addresses for an employee: We want to record *several values of the descriptive attributes for each instance of this relationship.* Accomplished by introducing new entity set, Duration.



## *Entity vs. Relationship*

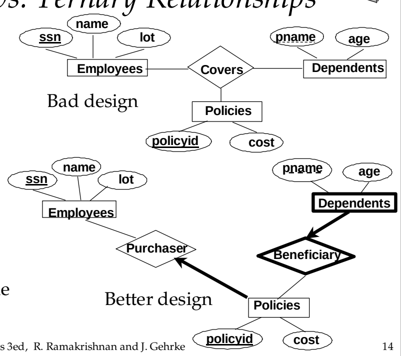
- First ER diagram OK if a manager gets a separate discretionary budget for each dept.
- What if a manager gets a discretionary budget that covers *all* managed depts?
  - Redundancy: *dbudget* stored for each dept managed by manager.
  - Misleading: Suggests *dbudget* associated with department-mgr combination.






## *Binary vs. Ternary Relationships*

- If each policy is owned by just 1 employee, and each dependent is tied to the covering policy, first diagram is inaccurate.
- What are the additional constraints in the 2nd diagram?






## *Binary vs. Ternary Relationships (Contd.)*

- Previous example illustrated a case when two binary relationships were better than one ternary relationship.
- An example in the other direction: a ternary relation Contracts relates entity sets Parts, Departments and Suppliers, and has descriptive attribute *qty*. No combination of binary relationships is an adequate substitute:
  - S “can-supply” P, D “needs” P, and D “deals-with” S does not imply that D has agreed to buy P from S.
  - How do we record *qty*?

## *Summary of Conceptual Design*

- *Conceptual design* follows *requirements analysis*,
  - Yields a high-level description of data to be stored
- ER model popular for conceptual design
  - Constructs are expressive, close to the way people think about their applications.
- Basic constructs: *entities*, *relationships*, and *attributes* (of entities and relationships).
- Some additional constructs: *weak entities*, *ISA hierarchies*, and *aggregation*.
- Note: There are many variations on ER model.



## *Summary of ER (Contd.)*

- Several kinds of integrity constraints can be expressed in the ER model: *key constraints*, *participation constraints*, and *overlap/covering constraints* for ISA hierarchies. Some *foreign key constraints* are also implicit in the definition of a relationship set.
  - Some constraints (notably, *functional dependencies*) cannot be expressed in the ER model.
  - Constraints play an important role in determining the best database design for an enterprise.



## *Summary of ER (Contd.)*

- ER design is *subjective*. There are often many ways to model a given scenario! Analyzing alternatives can be tricky, especially for a large enterprise. Common choices include:
  - Entity vs. attribute, entity vs. relationship, binary or n-ary relationship, whether or not to use ISA hierarchies, and whether or not to use aggregation.
- Ensuring good database design: resulting relational schema should be analyzed and refined further. FD information and normalization techniques are especially useful.

# The Relational Model

Chapter 3



## Why Study the Relational Model?

- Most widely used model.
  - Vendors: IBM, Informix, Microsoft, Oracle, Sybase, etc.
- "Legacy systems" in older models
  - E.G., IBM's IMS
- Recent competitor: object-oriented model
  - ObjectStore, Versant, Ontos
  - A synthesis emerging: *object-relational model*
    - Informix Universal Server, UniSQL, O2, Oracle, DB2



## Relational Database: Definitions

- *Relational database:* a set of *relations*
- *Relation:* made up of 2 parts:
  - *Instance* : a *table*, with rows and columns. #Rows = *cardinality*, #fields = *degree / arity.*
  - *Schema* : specifies name of relation, plus name and type of each column.
    - E.G. Students(*sid*: string, *name*: string, *login*: string, *age*: integer, *gpa*: real).
- Can think of a relation as a *set* of rows or *tuples* (i.e., all rows are distinct).

## *Example Instance of Students Relation*

| sid | name | login | age | gpa |
|---|---|---|---|---|
| 53666 | Jones | jones@cs | 18 | 3.4 |
| 53688 | Smith | smith@eecs | 18 | 3.2 |
| 53650 | Smith | smith@math | 19 | 3.8 |

- Cardinality = 3, degree = 5, all rows distinct
- Do all columns in a relation instance have to be distinct?



## *Relational Query Languages*

- A major strength of the relational model: supports simple, powerful *querying* of data.
- Queries can be written intuitively, and the DBMS is responsible for efficient evaluation.
  - The key: precise semantics for relational queries.
  - Allows the optimizer to extensively re-order operations, and still ensure that the answer does not change.



## *The SQL Query Language*

- Developed by IBM (system R) in the 1970s
- Need for a standard since it is used by many vendors
- Standards:
  - SQL-86
  - SQL-89 (minor revision)
  - SQL-92 (major revision)
  - SQL-99 (major extensions, current standard)

## *The SQL Query Language*

- To find all 18 year old students, we can write:

```
SELECT *
FROM Students S
WHERE S.age=18
```

| sid | name | login | age | gpa |
|---|---|---|---|---|
| 53666 | Jones | jones@cs | 18 | 3.4 |
| 53688 | Smith | smith@ee | 18 | 3.2 |

- To find just names and logins, replace the first line:

```
SELECT S.name, S.login
```



## *Querying Multiple Relations*

- What does the following query compute?

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

Given the following instance of Enrolled (is this possible if the DBMS ensures referential integrity?):

| sid | cid | grade |
|---|---|---|
| 53831 | Carnatic101 | C |
| 53831 | Reggae203 | B |
| 53650 | Topology112 | A |
| 53666 | History105 | B |

we get:

| S.name | E.cid |
|---|---|
| Smith | Topology112 |



## *Creating Relations in SQL*

- Creates the Students relation. Observe that the type (domain) of each field is specified, and enforced by the DBMS whenever tuples are added or modified.
- As another example, the Enrolled table holds information about courses that students take.

```
CREATE TABLE Students
      (sid: CHAR(20),
       name: CHAR(20),
       login: CHAR(10),
       age: INTEGER,
       gpa: REAL)

CREATE TABLE Enrolled
      (sid: CHAR(20),
       cid: CHAR(20),
       grade: CHAR(2))
```

## *Destroying and Altering Relations*

DROP TABLE Students

- Destroys the relation Students. The schema information *and* the tuples are deleted.

ALTER TABLE Students
ADD COLUMN firstYear: integer

- The schema of Students is altered by adding a new field; every tuple in the current instance is extended with a *null* value in the new field.



## *Adding and Deleting Tuples*

- Can insert a single tuple using:

INSERT INTO Students (sid, name, login, age, gpa)
VALUES (53688, 'Smith', 'smith@ee', 18, 3.2)

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

DELETE
FROM Students S
WHERE S.name = 'Smith'

* *Powerful variants of these commands are available; more later!*



## *Integrity Constraints (ICs)*

- IC: condition that must be true for *any* instance of the database; e.g., *domain constraints.*
  - ICs are specified when schema is defined.
  - ICs are checked when relations are modified.
- A *legal* instance of a relation is one that satisfies all specified ICs.
  - DBMS should not allow illegal instances.
- If the DBMS checks ICs, stored data is more faithful to real-world meaning.
  - Avoids data entry errors, too!

## *Primary Key Constraints*

- A set of fields is a *key* for a relation if :
  1. No two distinct tuples can have same values in all key fields, and
  2. This is not true for any subset of the key.
  - Part 2 false? A *superkey*.
  - If there's >1 key for a relation, one of the keys is chosen (by DBA) to be the *primary key*.
- E.g., *sid* is a key for Students. (What about *name*?) The set {*sid*, *gpa*} is a superkey.



## *Primary and Candidate Keys in SQL*

- Possibly many *candidate keys* (specified using UNIQUE), one of which is chosen as the *primary key*.
- "For a given student and course, there is a single grade." vs. "Students can take only one course, and receive a single grade for that course; further, no two students in a course receive the same grade."
- Used carelessly, an IC can prevent the storage of database instances that arise in practice!

```
CREATE TABLE Enrolled
  (sid CHAR(20)
   cid  CHAR(20),
   grade CHAR(2),
   PRIMARY KEY  (sid,cid) )
```

```
CREATE TABLE Enrolled
  (sid CHAR(20)
   cid  CHAR(20),
   grade CHAR(2),
   PRIMARY KEY  (sid),
   UNIQUE (cid, grade) )
```



## *Foreign Keys, Referential Integrity*

- *Foreign key* : Set of fields in one relation that is used to `refer' to a tuple in another relation. (Must correspond to primary key of the second relation.) Like a `logical pointer'.
- E.g. *sid* is a foreign key referring to Students:
  - Enrolled(*sid*: string, *cid*: string, *grade*: string)
  - If all foreign key constraints are enforced, *referential integrity* is achieved, i.e., no dangling references.
  - Can you name a data model w/o referential integrity?
    - Links in HTML!

## Foreign Keys in SQL

- Only students listed in the Students relation should be allowed to enroll for courses.

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

Enrolled

| sid | cid | grade |
|---|---|---|
| 53666 | Carnatic101 | C |
| 53666 | Reggae203 | B |
| 53650 | Topology112 | A |
| 53666 | History105 | B |

Students

| sid | name | login | age | gpa |
|---|---|---|---|---|
| 53666 | Jones | jones@cs | 18 | 3.4 |
| 53688 | Smith | smith@eecs | 18 | 3.2 |
| 53650 | Smith | smith@math | 19 | 3.8 |



## Enforcing Referential Integrity

- Consider Students and Enrolled; *sid* in Enrolled is a foreign key that references Students.
- What should be done if an Enrolled tuple with a non-existent student id is inserted? (*Reject it!*)
- What should be done if a Students tuple is deleted?
  - Also delete all Enrolled tuples that refer to it.
  - Disallow deletion of a Students tuple that is referred to.
  - Set sid in Enrolled tuples that refer to it to a *default sid*.
  - (In SQL, also: Set sid in Enrolled tuples that refer to it to a special value *null*, denoting `*unknown*' or `*inapplicable*'.)
- Similar if primary key of Students tuple is updated.



## Referential Integrity in SQL

- SQL/92 and SQL:1999 support all 4 options on deletes and updates.
  - Default is NO ACTION (*delete/update is rejected*)
  - CASCADE (also delete all tuples that refer to deleted tuple)
  - SET NULL / SET DEFAULT (sets foreign key value of referencing tuple)

```
CREATE TABLE Enrolled
  (sid CHAR(20),
   cid CHAR(20),
   grade CHAR(2),
   PRIMARY KEY (sid,cid),
   FOREIGN KEY (sid)
     REFERENCES Students
         ON DELETE CASCADE
         ON UPDATE SET DEFAULT )
```

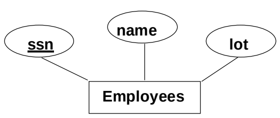
## *Where do ICs Come From?*

- ICs are based upon the semantics of the real-world enterprise that is being described in the database relations.
- We can check a database instance to see if an IC is violated, but we can NEVER infer that an IC is true by looking at an instance.
  - An IC is a statement about *all possible* instances!
  - From example, we know *name* is not a key, but the assertion that *sid* is a key is given to us.
- Key and foreign key ICs are the most common; more general ICs supported too.



## *Logical DB Design: ER to Relational*

- Entity sets to tables:




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



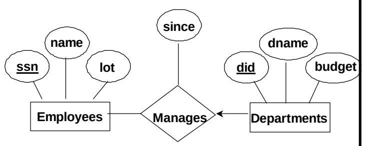
## *Relationship Sets to Tables*

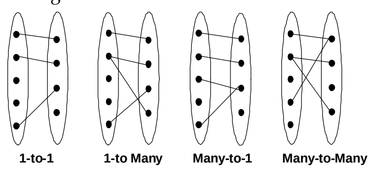
- In translating a relationship set to a relation, attributes of the relation must include:
  - Keys for each participating entity set (as foreign keys).
    - This set of attributes forms a *superkey* for the relation.
  - All descriptive attributes.

```
CREATE TABLE Works_In(
  ssn CHAR(1),
  did INTEGER,
  since DATE,
  PRIMARY KEY (ssn, did),
  FOREIGN KEY (ssn)
     REFERENCES Employees,
  FOREIGN KEY (did)
     REFERENCES Departments)
```

# *Review: Key Constraints*

- Each dept has at most one manager, according to the *key constraint* on Manages.




*Translation to relational model?*



# *Translating ER Diagrams with Key Constraints*

- Map relationship to a table:
  - Note that did is the key now!
  - Separate tables for Employees and Departments.
- Since each department has a unique manager, we could instead combine Manages and Departments.

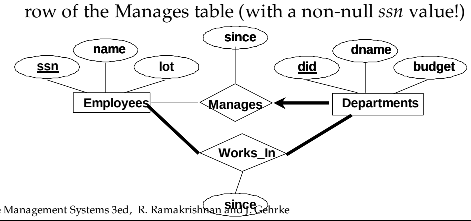
```
CREATE TABLE  Manages(
  ssn  CHAR(11),
  did  INTEGER,
  since  DATE,
  PRIMARY KEY  (did),
  FOREIGN KEY (ssn) REFERENCES Employees,
  FOREIGN KEY (did) REFERENCES Departments)
```

```
CREATE TABLE  Dept_Mgr(
  did  INTEGER,
  dname  CHAR(20),
  budget  REAL,
  ssn  CHAR(11),
  since  DATE,
  PRIMARY KEY  (did),
  FOREIGN KEY (ssn) REFERENCES Employees)
```



# *Review: Participation Constraints*

- Does every department have a manager?
  - If so, this is a *participation constraint*: the participation of Departments in Manages is said to be *total* (vs. *partial*).
    - Every *did* value in Departments table must appear in a row of the Manages table (with a non-null *ssn* value!)



## *Participation Constraints in SQL*

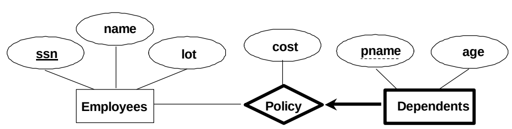
- We can capture participation constraints involving one entity set in a binary relationship, but little else (without resorting to CHECK constraints).

```
CREATE TABLE Dept_Mgr(
  did INTEGER,
  dname CHAR(20),
  budget REAL,
  ssn CHAR(11) NOT NULL,
  since DATE,
  PRIMARY KEY (did),
  FOREIGN KEY (ssn) REFERENCES Employees,
    ON DELETE NO ACTION)
```



## *Review: 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 (1 owner, many weak entities).
  - Weak entity set must have total participation in this *identifying* relationship set.



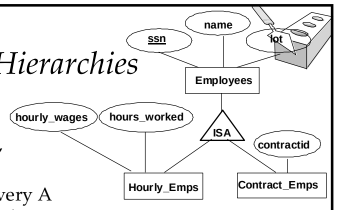



## *Translating Weak Entity Sets*

- Weak entity set and identifying relationship set are translated into a single table.
  - When the owner entity is deleted, all owned weak entities must also be deleted.

```
CREATE TABLE Dep_Policy (
  pname CHAR(20),
  age INTEGER,
  cost REAL,
  ssn CHAR(11) NOT NULL,
  PRIMARY KEY (pname, ssn),
  FOREIGN KEY (ssn) REFERENCES Employees,
    ON DELETE CASCADE)
```

## *Review: ISA Hierarchies*




- As in C++, or other PLs, attributes are inherited.
- If we declare A **ISA** B, every A entity is also considered to be a B entity.
- *Overlap constraints*: Can Joe be an Hourly_Emps as well as a Contract_Emps entity? (*Allowed/disallowed*)
- *Covering constraints*: Does every Employees entity also have to be an Hourly_Emps or a Contract_Emps entity? *(Yes/no)*



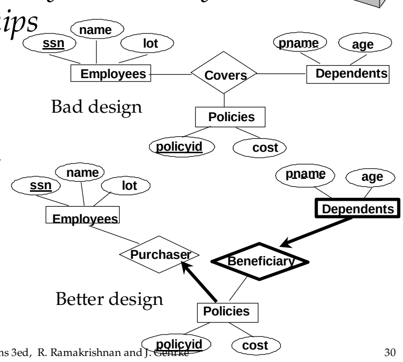
## *Translating ISA Hierarchies to Relations*

- ***General approach:***
  - 3 relations: Employees, Hourly_Emps and Contract_Emps.
    - *Hourly_Emps*: Every employee is recorded in Employees. For hourly emps, extra info recorded in Hourly_Emps (*hourly_wages*, *hours_worked*, *ssn*); must delete Hourly_Emps tuple if referenced Employees tuple is deleted).
    - Queries involving all employees easy, those involving just Hourly_Emps require a join to get some attributes.
- Alternative: Just Hourly_Emps and Contract_Emps.
  - *Hourly_Emps*: *ssn, name, lot, hourly_wages, hours_worked.*
  - Each employee must be in one of these two subclasses.



## *Review: Binary vs. Ternary Relationships*

- What are the additional constraints in the 2nd diagram?



## Binary vs. Ternary Relationships (Contd.)

- The key constraints allow us to combine Purchaser with Policies and Beneficiary with Dependents.
- Participation constraints lead to NOT NULL constraints.
- What if Policies is a weak entity set?

```
CREATE TABLE Policies (
  policyid INTEGER,
  cost REAL,
  ssn CHAR(11) NOT NULL,
  PRIMARY KEY (policyid).
  FOREIGN KEY (ssn) REFERENCES Employees,
    ON DELETE CASCADE)
```

```
CREATE TABLE Dependents (
  pname CHAR(20),
  age INTEGER,
  policyid INTEGER,
  PRIMARY KEY (pname, policyid).
  FOREIGN KEY (policyid) REFERENCES Policies,
    ON DELETE CASCADE)
```



## Views

- A *view* is just a relation, but we store a *definition*, rather than a set of tuples.

```
CREATE VIEW YoungActiveStudents (name, grade)
    AS SELECT S.name, E.grade
    FROM Students S, Enrolled E
    WHERE S.sid = E.sid and S.age<21
```

- Views can be dropped using the DROP VIEW command.
  - How to handle DROP TABLE if there's a view on the table?
    - DROP TABLE command has options to let the user specify this.



## Views and Security

- Views can be used to present necessary information (or a summary), while hiding details in underlying relation(s).
  - Given YoungStudents, but not Students or Enrolled, we can find students s who have are enrolled, but not the *cid's* of the courses they are enrolled in.

## *Relational Model: Summary*

- A tabular representation of data.
- Simple and intuitive, currently the most widely used.
- Integrity constraints can be specified by the DBA, based on application semantics. DBMS checks for violations.
  - Two important ICs: primary and foreign keys
  - In addition, we *always* have domain constraints.
- Powerful and natural query languages exist.
- Rules to translate ER to relational model

## Relational Algebra

Chapter 4, Part A



## Relational Query Languages

- *Query languages:* Allow manipulation and retrieval of data from a database.
- Relational model supports simple, powerful QLs:
  - Strong formal foundation based on logic.
  - Allows for much optimization.
- Query Languages **!=** programming languages!
  - QLs not expected to be "Turing complete".
  - QLs not intended to be used for complex calculations.
  - QLs support easy, efficient access to large data sets.



## Formal Relational Query Languages

- Two mathematical Query Languages form the basis for "real" languages (e.g. SQL), and for implementation:
  - *Relational Algebra*: More operational, very useful for representing execution plans.
  - *Relational Calculus*: Lets users describe what they want, rather than how to compute it. (Non-operational, *declarative*.)

## *Preliminaries*

- A query is applied to *relation instances*, and the result of a query is also a relation instance.
  - *Schemas* of input relations for a query are fixed (but query will run regardless of instance!)
  - The schema for the *result* of a given query is also fixed! Determined by definition of query language constructs.
- Positional vs. named-field notation:
  - Positional notation easier for formal definitions, named-field notation more readable.
  - Both used in SQL



## *Example Instances*

- "Sailors" and "Reserves" relations for our examples.
- We'll use positional or named field notation, assume that names of fields in query results are `inherited' from names of fields in query input relations.

***R1***

| <u>sid</u> | <u>bid</u> | <u>day</u> |
|---|---|---|
| 22 | 101 | 10/10/96 |
| 58 | 103 | 11/12/96 |

***S1***

| <u>sid</u> | sname | rating | age |
|---|---|---|---|
| 22 | dustin | 7 | 45.0 |
| 31 | lubber | 8 | 55.5 |
| 58 | rusty | 10 | 35.0 |

***S2***

| <u>sid</u> | sname | rating | age |
|---|---|---|---|
| 28 | yuppy | 9 | 35.0 |
| 31 | lubber | 8 | 55.5 |
| 44 | guppy | 5 | 35.0 |
| 58 | rusty | 10 | 35.0 |



## *Relational Algebra*

- Basic operations:
  - <u>*Selection*</u> ( $\sigma$ ) Selects a subset of rows from relation.
  - <u>*Projection*</u> ( $\pi$ ) Deletes unwanted columns from relation.
  - <u>*Cross-product*</u> ( $\times$ ) Allows us to combine two relations.
  - <u>*Set-difference*</u> ( $-$ ) Tuples in reln. 1, but not in reln. 2.
  - <u>*Union*</u> ( $\cup$ ) Tuples in reln. 1 and in reln. 2.
- Additional operations:
  - Intersection, <u>*join*</u>, division, renaming: Not essential, but (very!) useful.
- Since each operation returns a relation, operations can be *composed*! (Algebra is "closed".)

## *Projection*

- Deletes attributes that are not in *projection list*.
- *Schema* of result contains exactly the fields in the projection list, with the same names that they had in the (only) input relation.
- Projection operator has to eliminate *duplicates*! (Why??)
  - Note: real systems typically don't do duplicate elimination unless the user explicitly asks for it. (Why not?)

| sname | rating |
|---|---|
| yuppy | 9 |
| lubber | 8 |
| guppy | 5 |
| rusty | 10 |

$$\pi_{sname,rating}(S2)$$

| age |
|---|
| 35.0 |
| 55.5 |

$$\pi_{age}(S2)$$



## *Selection*

- Selects rows that satisfy *selection condition*.
- No duplicates in result! (Why?)
- *Schema* of result identical to schema of (only) input relation.
- *Result* relation can be the *input* for another relational algebra operation! (*Operator composition*.)

| sid | sname | rating | age |
|---|---|---|---|
| 28 | yuppy | 9 | 35.0 |
| 58 | rusty | 10 | 35.0 |

$$\sigma_{rating>8}(S2)$$

| sname | rating |
|---|---|
| yuppy | 9 |
| rusty | 10 |

$$\pi_{sname,rating}(\sigma_{rating>8}(S2))$$



## *Union, Intersection, Set-Difference*

- All of these operations take two input relations, which must be <u>*union-compatible*</u>:
  - Same number of fields.
  - `Corresponding' fields have the same type.
- What is the *schema* of result?

| sid | sname | rating | age |
|---|---|---|---|
| 22 | dustin | 7 | 45.0 |
| 31 | lubber | 8 | 55.5 |
| 58 | rusty | 10 | 35.0 |
| 44 | guppy | 5 | 35.0 |
| 28 | yuppy | 9 | 35.0 |

$$S1 \cup S2$$

| sid | sname | rating | age |
|---|---|---|---|
| 22 | dustin | 7 | 45.0 |

$$S1 - S2$$

| sid | sname | rating | age |
|---|---|---|---|
| 31 | lubber | 8 | 55.5 |
| 58 | rusty | 10 | 35.0 |

$$S1 \cap S2$$

## Cross-Product

- Each row of S1 is paired with each row of R1.
- *Result schema* has one field per field of S1 and R1, with field names `inherited' if possible.
  - *Conflict*: Both S1 and R1 have a field called *sid*.

| (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 |

  - Renaming operator: $\rho\,(C(1 \rightarrow sid1, 5 \rightarrow sid2),\, S1 \times R1)$



## Joins

- Condition Join: $R \bowtie_c S = \sigma_c (R \times S)$

| (sid) | sname | rating | age | (sid) | bid | day |
|---|---|---|---|---|---|---|
| 22 | dustin | 7 | 45.0 | 58 | 103 | 11/12/96 |
| 31 | lubber | 8 | 55.5 | 58 | 103 | 11/12/96 |

$$S1 \bowtie_{S1.sid < R1.sid} R1$$

- *Result schema* same as that of cross-product.
- Fewer tuples than cross-product, might be able to compute more efficiently
- Sometimes called a *theta-join*.



## Joins

- Equi-Join: A special case of condition join where the condition *c* contains only ***equalities.***

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

$$S1 \bowtie_{sid} R1$$

- *Result schema* similar to cross-product, but only one copy of fields for which equality is specified.
- Natural Join: Equijoin on *all* common fields.

## *Division*

- Not supported as a primitive operator, but useful for expressing queries like:
  *Find sailors who have reserved **all** boats.*
- Let $A$ have 2 fields, $x$ and $y$; $B$ have only field $y$:
  - $A/B = \{\langle x \rangle \mid \exists \langle x, y \rangle \in A \;\forall \langle y \rangle \in B\}$
  - i.e., ***A/B* contains all *x* tuples (sailors) such that for *every y* tuple (boat) in *B*, there is an *xy* tuple in *A*.**
  - *Or*: If the set of $y$ values (boats) associated with an $x$ value (sailor) in $A$ contains all $y$ values in $B$, the $x$ value is in $A/B$.
- In general, $x$ and $y$ can be any lists of fields; $y$ is the list of fields in $B$, and $x \cup y$ is the list of fields of $A$.



## *Examples of Division A/B*

| sno | pno |
|---|---|
| s1 | p1 |
| s1 | p2 |
| s1 | p3 |
| s1 | p4 |
| s2 | p1 |
| s2 | p2 |
| s3 | p2 |
| s4 | p2 |
| s4 | p4 |

*A*

| pno |
|---|
| p2 |

*B1*

| pno |
|---|
| p2 |
| p4 |

*B2*

| pno |
|---|
| p1 |
| p2 |
| p4 |

*B3*

| sno |
|---|
| s1 |
| s2 |
| s3 |
| s4 |

*A/B1*

| sno |
|---|
| s1 |
| s4 |

*A/B2*

| sno |
|---|
| s1 |

*A/B3*



## *Expressing A/B Using Basic Operators*

- Division is not essential op; just a useful shorthand.
  - (Also true of joins, but joins are so common that systems implement joins specially.)
- *Idea*: For $A/B$, compute all $x$ values that are not \`disqualified' by some $y$ value in $B$.
  - $x$ value is *disqualified* if by attaching $y$ value from $B$, we obtain an $xy$ tuple that is not in $A$.

Disqualified $x$ values: $\pi_x((\pi_x(A) \times B) - A)$

*A/B*: $\pi_x(A)$ $-$ all disqualified tuples

## *Find names of sailors who've reserved boat #103*

- Solution 1: $\pi_{sname}((\sigma_{bid=103} \text{Reserves}) \bowtie Sailors)$
- Solution 2: $\rho\ (Temp1, \sigma_{bid=103} \text{Reserves})$

  $\rho\ (Temp2, Temp1 \bowtie Sailors)$

  $\pi_{sname}\ (Temp2)$
- Solution 3: $\pi_{sname}(\sigma_{bid=103}(\text{Reserves} \bowtie Sailors))$



## *Find names of sailors who've reserved a red boat*

- Information about boat color only available in Boats; so need an extra join:

$\pi_{sname}((\sigma_{color='red'} Boats) \bowtie \text{Reserves} \bowtie Sailors)$

- A more efficient solution:

$\pi_{sname}(\pi_{sid}((\pi_{bid}\sigma_{color='red'} Boats) \bowtie \text{Res}) \bowtie Sailors)$

*A query optimizer can find this, given the first solution!*



## *Find sailors who've reserved a red or a green boat*

- Can identify all red or green boats, then find sailors who've reserved one of these boats:

$\rho\ (Tempboats, (\sigma_{color='red' \vee color='green'} Boats))$

$\pi_{sname}(Tempboats \bowtie \text{Reserves} \bowtie Sailors)$

- Can also define Tempboats using union! (How?)
- What happens if $\vee$ is replaced by $\wedge$ in this query?

## Find sailors who've reserved a red and a green boat

- Previous approach won't work! Must identify sailors who've reserved red boats, sailors who've reserved green boats, then find the intersection (note that *sid* is a key for Sailors):

$$\rho\ (Tempred,\ \pi_{sid}((\sigma_{color='red'}\ Boats) \bowtie \text{Reserves}))$$

$$\rho\ (Tempgreen,\ \pi_{sid}((\sigma_{color='green'}\ Boats) \bowtie \text{Reserves}))$$

$$\pi_{sname}((Tempred \cap Tempgreen) \bowtie Sailors)$$



## Find the names of sailors who've reserved all boats

- Uses division; schemas of the input relations to / must be carefully chosen:

$$\rho\ (Tempsids,\ (\pi_{sid,bid}\text{Reserves})\ /\ (\pi_{bid}\ Boats))$$

$$\pi_{sname}\ (Tempsids \bowtie Sailors)$$

- To find sailors who've reserved all 'Interlake' boats:

$$.....\ \ /\ \pi_{bid}(\sigma_{bname='Interlake'}\ Boats)$$



## Summary

- The relational model has rigorously defined query languages that are simple and powerful.
- Relational algebra is more operational; useful as internal representation for query evaluation plans.
- Several ways of expressing a given query; a query optimizer should choose the most efficient version.

## *SQL: Queries, Programming, Triggers*

Chapter 5



## *Example Instances*

***R1***

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

- We will use these instances of the Sailors and Reserves relations in our examples.
- If the key for the Reserves relation contained only the attributes *sid* and *bid*, how would the semantics differ?

***S1***

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

***S2***

| sid | sname | rating | age |
|---|---|---|---|
| 28 | yuppy | 9 | 35.0 |
| 31 | lubber | 8 | 55.5 |
| 44 | guppy | 5 | 35.0 |
| 58 | rusty | 10 | 35.0 |



## *Basic SQL Query*

```
SELECT    [DISTINCT]  target-list
FROM      relation-list
WHERE     qualification
```

- *relation-list* A list of relation names (possibly with a *range-variable* after each name).
- *target-list* A list of attributes of relations in *relation-list*
- *qualification* Comparisons (Attr *op* const or Attr1 *op* Attr2, where *op* is one of $<, >, =, \leq, \geq, \neq$ ) combined using AND, OR and NOT.
- DISTINCT is an optional keyword indicating that the answer should not contain duplicates. Default is that duplicates are *not* eliminated!

## *Conceptual Evaluation Strategy*

- Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:
  - Compute the cross-product of *relation-list*.
  - Discard resulting tuples if they fail *qualifications*.
  - Delete attributes that are not in *target-list*.
  - If DISTINCT is specified, eliminate duplicate rows.
- This strategy is probably the least efficient way to compute a query! An optimizer will find more efficient strategies to compute *the same answers*.



## *Example of Conceptual Evaluation*

```
SELECT S.sname
FROM   Sailors S, Reserves R
WHERE  S.sid=R.sid AND R.bid=103
```

| (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 |



## *A Note on Range Variables*

- Really needed only if the same relation appears twice in the FROM clause. The previous query can also be written as:

```
SELECT S.sname
FROM   Sailors S, Reserves R
WHERE  S.sid=R.sid AND bid=103
```

OR

```
SELECT sname
FROM   Sailors, Reserves
WHERE  Sailors.sid=Reserves.sid
       AND bid=103
```

*It is good style, however, to use range variables always!*

*Find sailors who've reserved at least one boat*

```
SELECT S.sid
FROM Sailors S, Reserves R
WHERE S.sid=R.sid
```

- Would adding DISTINCT to this query make a difference?
- What is the effect of replacing *S.sid* by *S.sname* in the SELECT clause? Would adding DISTINCT to this variant of the query make a difference?



## *Expressions and Strings*

```
SELECT S.age, age1=S.age-5, 2*S.age AS age2
FROM Sailors S
WHERE S.sname LIKE 'B_%B'
```

- Illustrates use of arithmetic expressions and string pattern matching: *Find triples (of ages of sailors and two fields defined by expressions) for sailors whose names begin and end with B and contain at least three characters.*
- AS and = are two ways to name fields in result.
- LIKE is used for string matching. `_' stands for any one character and `%' stands for 0 or more arbitrary characters.



*Find sid's of sailors who've reserved a red <u>or</u> a green boat*

- UNION: Can be used to compute the union of any two *union-compatible* sets of tuples (which are themselves the result of SQL queries).
- If we replace OR by AND in the first version, what do we get?
- Also available: EXCEPT (What do we get if we replace UNION by EXCEPT?)

```
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid
  AND (B.color='red' OR B.color='green')
```

```
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid
        AND B.color='red'
UNION
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid
        AND B.color='green'
```

*Find sid's of sailors who've reserved a red <u>and</u> a green boat*

- INTERSECT: Can be used to compute the intersection of any two *union-compatible* sets of tuples.
- Included in the SQL/92 standard, but some systems don't support it.
- Contrast symmetry of the UNION and INTERSECT queries with how much the other versions differ.

```
SELECT S.sid
FROM Sailors S, Boats B1, Reserves R1,
       Boats B2, Reserves R2
WHERE S.sid=R1.sid AND R1.bid=B1.bid
  AND S.sid=R2.sid AND R2.bid=B2.bid
  AND (B1.color='red' AND B2.color='green')
```

Key field!

```
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid
       AND B.color='red'
INTERSECT
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid
       AND B.color='green'
```



# Nested Queries

*Find names of sailors who've reserved boat #103:*

```
SELECT S.sname
FROM Sailors S
WHERE S.sid IN (SELECT R.sid
                FROM Reserves R
                WHERE R.bid=103)
```

- A very powerful feature of SQL: a WHERE clause can itself contain an SQL query! (Actually, so can FROM and HAVING clauses.)
- To find sailors who've *not* reserved #103, use NOT IN.
- To understand semantics of nested queries, think of a <u>*nested loops*</u> evaluation: *For each Sailors tuple, check the qualification by computing the subquery.*



# Nested Queries with Correlation

*Find names of sailors who've reserved boat #103:*

```
SELECT S.sname
FROM Sailors S
WHERE EXISTS (SELECT *
              FROM Reserves R
              WHERE R.bid=103 AND S.sid=R.sid)
```

- EXISTS is another set comparison operator, like IN.
- If UNIQUE is used, and * is replaced by *R.bid*, finds sailors with at most one reservation for boat #103. (UNIQUE checks for duplicate tuples; * denotes all attributes. Why do we have to replace * by *R.bid*?)
- Illustrates why, in general, subquery must be re-computed for each Sailors tuple.

## More on Set-Comparison Operators

- We've already seen IN, EXISTS and UNIQUE. Can also use NOT IN, NOT EXISTS and NOT UNIQUE.
- Also available: *op* ANY, *op* ALL, *op* IN $>,<,=,\geq,\leq,\neq$
- Find sailors whose rating is greater than that of some sailor called Horatio:

```
SELECT  *
FROM  Sailors S
WHERE  S.rating > ANY  (SELECT  S2.rating
                        FROM  Sailors S2
                        WHERE S2.sname='Horatio')
```



## Rewriting INTERSECT Queries Using IN

*Find sid's of sailors who've reserved both a red and a green boat:*

```
SELECT  S.sid
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid AND B.color='red'
       AND S.sid IN  (SELECT  S2.sid
                      FROM  Sailors S2, Boats B2, Reserves R2
                      WHERE  S2.sid=R2.sid AND R2.bid=B2.bid
                             AND  B2.color='green')
```

- Similarly, EXCEPT queries re-written using NOT IN.
- To find *names* (not *sid*'s) of Sailors who've reserved both red and green boats, just replace *S.sid* by *S.sname* in SELECT clause. (What about INTERSECT query?)



## Division in SQL

Find sailors who've reserved all boats.

(1)
```
SELECT  S.sname
FROM  Sailors S
WHERE  NOT EXISTS
        ((SELECT  B.bid
          FROM  Boats B)
         EXCEPT
         (SELECT  R.bid
          FROM  Reserves R
          WHERE  R.sid=S.sid))
```

- Let's do it the hard way, without EXCEPT:

(2)
```
SELECT  S.sname
FROM  Sailors S
WHERE  NOT EXISTS  (SELECT  B.bid
                    FROM  Boats B
                    WHERE  NOT EXISTS  (SELECT  R.bid
                                        FROM  Reserves R
                                        WHERE  R.bid=B.bid
                                               AND R.sid=S.sid))
```

*Sailors S such that ...*

*there is no boat B without ...*

*a Reserves tuple showing S reserved B*

## *Aggregate Operators*

COUNT (*)
COUNT ( [DISTINCT] A)
SUM ( [DISTINCT] A)
AVG ( [DISTINCT] A)
MAX (A)
MIN (A)

*single column*

- Significant extension of relational algebra.

```
SELECT COUNT (*)
FROM Sailors S
```

```
SELECT AVG (S.age)
FROM Sailors S
WHERE S.rating=10
```

```
SELECT S.sname
FROM Sailors S
WHERE S.rating= (SELECT MAX(S2.rating)
                 FROM Sailors S2)
```

```
SELECT COUNT (DISTINCT S.rating)
FROM Sailors S
WHERE S.sname='Bob'
```

```
SELECT AVG ( DISTINCT S.age)
FROM Sailors S
WHERE S.rating=10
```



## *Find name and age of the oldest sailor(s)*

- The first query is illegal! (We'll look into the reason a bit later, when we discuss GROUP BY.)
- The third query is equivalent to the second query, and is allowed in the SQL/92 standard, but is not supported in some systems.

```
SELECT S.sname, MAX (S.age)
FROM Sailors S
```

```
SELECT S.sname, S.age
FROM Sailors S
WHERE S.age =
        (SELECT MAX (S2.age)
         FROM Sailors S2)
```

```
SELECT S.sname, S.age
FROM Sailors S
WHERE (SELECT MAX (S2.age)
       FROM Sailors S2)
       = S.age
```



## *GROUP BY and HAVING*

- So far, we've applied aggregate operators to all (qualifying) tuples. Sometimes, we want to apply them to each of several *groups* of tuples.
- Consider: *Find the age of the youngest sailor for each rating level.*
  - In general, we don't know how many rating levels exist, and what the rating values for these levels are!
  - Suppose we know that rating values go from 1 to 10; we can write 10 queries that look like this (!):

For $i$ = 1, 2, ... , 10:

```
SELECT MIN (S.age)
FROM Sailors S
WHERE S.rating = i
```

## Queries With GROUP BY and HAVING

```
SELECT    [DISTINCT] target-list
FROM      relation-list
WHERE     qualification
GROUP BY  grouping-list
HAVING    group-qualification
```

- The *target-list* contains (i) attribute names (ii) terms with aggregate operations (e.g., MIN (*S.age*)).
  - The attribute list (i) must be a subset of *grouping-list*. Intuitively, each answer tuple corresponds to a *group*, and these attributes must have a single value per group. (A *group* is a set of tuples that have the same value for all attributes in *grouping-list*.)



## Conceptual Evaluation

- The cross-product of *relation-list* is computed, tuples that fail *qualification* are discarded, \`*unnecessary*' fields are deleted, and the remaining tuples are partitioned into groups by the value of attributes in *grouping-list*.
- The *group-qualification* is then applied to eliminate some groups. Expressions in *group-qualification* must have a *single value per group*!
  - In effect, an attribute in *group-qualification* that is not an argument of an aggregate op also appears in *grouping-list*. (SQL does not exploit primary key semantics here!)
- One answer tuple is generated per qualifying group.



## Find the age of the youngest sailor with age ≥ 18, for each rating with at least 2 such sailors

```
SELECT S.rating, MIN (S.age)
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating
HAVING COUNT (*) > 1
```

| sid | sname | rating | age |
|---|---|---|---|
| 22 | dustin | 7 | 45.0 |
| 31 | lubber | 8 | 55.5 |
| 71 | zorba | 10 | 16.0 |
| 64 | horatio | 7 | 35.0 |
| 29 | brutus | 1 | 33.0 |
| 58 | rusty | 10 | 35.0 |

- Only S.rating and S.age are mentioned in the SELECT, GROUP BY or HAVING clauses; other attributes \`*unnecessary*'.
- 2nd column of result is unnamed. (Use AS to name it.)

| rating | age |
|---|---|
| 1 | 33.0 |
| 7 | 45.0 |
| 7 | 35.0 |
| 8 | 55.5 |
| 10 | 35.0 |

| rating | |
|---|---|
| 7 | 35.0 |

*Answer relation*

## *For each red boat, find the number of reservations for this boat*

```
SELECT  B.bid,  COUNT (*) AS scount
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid AND B.color='red'
GROUP BY  B.bid
```

- Grouping over a join of three relations.
- What do we get if we remove *B.color='red'* from the WHERE clause and add a HAVING clause with this condition?
- What if we drop Sailors and the condition involving S.sid?



## *Find the age of the youngest sailor with age > 18, for each rating with at least 2 sailors (of any age)*

```
SELECT  S.rating,  MIN (S.age)
FROM  Sailors S
WHERE  S.age > 18
GROUP BY  S.rating
HAVING  1  <  (SELECT  COUNT (*)
               FROM  Sailors S2
               WHERE  S.rating=S2.rating)
```

- Shows HAVING clause can also contain a subquery.
- Compare this with the query where we considered only ratings with 2 sailors over 18!
- What if HAVING clause is replaced by:
  - HAVING COUNT(*) >1



## *Find those ratings for which the average age is the minimum over all ratings*

- Aggregate operations cannot be nested! WRONG:

```
SELECT  S.rating
FROM  Sailors S
WHERE  S.age =  (SELECT  MIN (AVG (S2.age))  FROM Sailors S2)
```

- Correct solution (in SQL/92):

```
SELECT  Temp.rating, Temp.avgage
FROM  (SELECT  S.rating, AVG (S.age) AS avgage
        FROM  Sailors S
        GROUP BY  S.rating) AS Temp
WHERE  Temp.avgage = (SELECT  MIN (Temp.avgage)
                      FROM  Temp)
```

## Null Values

- Field values in a tuple are sometimes *unknown* (e.g., a rating has not been assigned) or *inapplicable* (e.g., no spouse's name).
  - SQL provides a special value *null* for such situations.
- The presence of *null* complicates many issues. E.g.:
  - Special operators needed to check if value is/is not *null*.
  - Is *rating>8* true or false when *rating* is equal to *null*? What about AND, OR and NOT connectives?
  - We need a 3-valued logic (true, false and *unknown*).
  - Meaning of constructs must be defined carefully. (e.g., WHERE clause eliminates rows that don't evaluate to true.)
  - New operators (in particular, *outer joins*) possible/needed.



## Integrity Constraints (Review)

- An IC describes conditions that every *legal instance* of a relation must satisfy.
  - Inserts/deletes/updates that violate IC's are disallowed.
  - Can be used to ensure application semantics (e.g., *sid* is a key), or prevent inconsistencies (e.g., *sname* has to be a string, *age* must be < 200)
- *Types of IC's*: Domain constraints, primary key constraints, foreign key constraints, general constraints.
  - *Domain constraints*: Field values must be of right type. Always enforced.



## General Constraints

- Useful when more general ICs than keys are involved.
- Can use queries to express constraint.
- Constraints can be named.

```
CREATE TABLE Sailors
    ( sid INTEGER,
      sname CHAR(10),
      rating INTEGER,
      age REAL,
      PRIMARY KEY (sid),
      CHECK ( rating >= 1
          AND rating <= 10 )

CREATE TABLE Reserves
    ( sname CHAR(10),
      bid INTEGER,
      day DATE,
      PRIMARY KEY (bid,day),
      CONSTRAINT noInterlakeRes
      CHECK (`Interlake' <>
                ( SELECT B.bname
                  FROM Boats B
                  WHERE B.bid=bid)))
```

## Constraints Over Multiple Relations

- Awkward and wrong!
- If Sailors is empty, the number of Boats tuples can be anything!
- ASSERTION is the right solution; not associated with either table.

```
CREATE TABLE Sailors
  ( sid INTEGER,
    sname CHAR(10),
    rating INTEGER,
    age REAL,
    PRIMARY KEY (sid),
    CHECK
    ( (SELECT COUNT (S.sid) FROM Sailors S)
    + (SELECT COUNT (B.bid) FROM Boats B) < 100
```

*Number of boats plus number of sailors is < 100*

```
CREATE ASSERTION smallClub
CHECK
( (SELECT COUNT (S.sid) FROM Sailors S)
+ (SELECT COUNT (B.bid) FROM Boats B) < 100
```



## Triggers

- Trigger: procedure that starts automatically if specified changes occur to the DBMS
- Three parts:
  - Event (activates the trigger)
  - Condition (tests whether the triggers should run)
  - Action (what happens if the trigger runs)



## Triggers: Example (SQL:1999)

```
CREATE TRIGGER youngSailorUpdate
  AFTER INSERT ON SAILORS
REFERENCING NEW TABLE NewSailors
FOR EACH STATEMENT
  INSERT
      INTO YoungSailors(sid, name, age, rating)
      SELECT sid, name, age, rating
      FROM NewSailors N
      WHERE N.age <= 18
```

## *Summary*

- SQL was an important factor in the early acceptance of the relational model; more natural than earlier, procedural query languages.
- Relationally complete; in fact, significantly more expressive power than relational algebra.
- Even queries that can be expressed in RA can often be expressed more naturally in SQL.
- Many alternative ways to write a query; optimizer should look for most efficient evaluation plan.
  - In practice, users need to be aware of how queries are optimized and evaluated for best results.



## *Summary (Contd.)*

- NULL for unknown field values brings many complications
- SQL allows specification of rich integrity constraints
- Triggers respond to changes in the database

# Database Application Development

Chapter 6



## Overview

Concepts covered in this lecture:
- SQL in application code
- Embedded SQL
- Cursors
- Dynamic SQL
- JDBC
- SQLJ
- Stored procedures



## SQL in Application Code

- SQL commands can be called from within a host language (e.g., C++ or Java) program.
  - SQL statements can refer to host variables (including special variables used to return status).
  - Must include a statement to *connect* to the right database.
- Two main integration approaches:
  - Embed SQL in the host language (Embedded SQL, SQLJ)
  - Create special API to call SQL commands (JDBC)

## SQL in Application Code (Contd.)

Impedance mismatch:

- SQL relations are (multi-) sets of records, with no *a priori* bound on the number of records. No such data structure exist traditionally in procedural programming languages such as C++. (Though now: STL)
  - SQL supports a mechanism called a *cursor* to handle this.



## Embedded SQL

- Approach: Embed SQL in the host language.
  - A preprocessor converts the SQL statements into special API calls.
  - Then a regular compiler is used to compile the code.
- Language constructs:
  - Connecting to a database:
    EXEC SQL CONNECT
  - Declaring variables:
    EXEC SQL BEGIN (END) DECLARE SECTION
  - Statements:
    EXEC SQL Statement;



## Embedded SQL: Variables

```
EXEC SQL BEGIN DECLARE SECTION
char c_sname[20];
long c_sid;
short c_rating;
float c_age;
EXEC SQL END DECLARE SECTION
```

- Two special "error" variables:
  - SQLCODE (long, is negative if an error has occurred)
  - SQLSTATE (char[6], predefined codes for common errors)

## *Cursors*

- Can declare a cursor on a relation or query statement (which generates a relation).
- Can *open* a cursor, and repeatedly *fetch* a tuple then *move* the cursor, until all tuples have been retrieved.
  - Can use a special clause, called ORDER BY, in queries that are accessed through a cursor, to control the order in which tuples are returned.
    - Fields in ORDER BY clause must also appear in SELECT clause.
  - The ORDER BY clause, which orders answer tuples, is *only* allowed in the context of a cursor.
- Can also modify/delete tuple pointed to by a cursor.



## *Cursor that gets names of sailors who've reserved a red boat, in alphabetical order*

```
EXEC SQL DECLARE sinfo CURSOR FOR
     SELECT  S.sname
     FROM  Sailors S, Boats B, Reserves R
     WHERE  S.sid=R.sid AND R.bid=B.bid AND B.color='red'
     ORDER BY  S.sname
```

- Note that it is illegal to replace *S.sname* by, say, *S.sid* in the ORDER BY clause! (Why?)
- Can we add *S.sid* to the SELECT clause and replace *S.sname* by *S.sid* in the ORDER BY clause?



## *Embedding SQL in C: An Example*

```
char SQLSTATE[6];
EXEC SQL BEGIN DECLARE SECTION
char c_sname[20]; short c_minrating; float c_age;
EXEC SQL END DECLARE SECTION
c_minrating = random();
EXEC SQL DECLARE sinfo CURSOR FOR
   SELECT S.sname, S.age      FROM Sailors S
   WHERE S.rating > :c_minrating
   ORDER BY S.sname;
do {
   EXEC SQL FETCH sinfo INTO :c_sname, :c_age;
   printf("%s is %d years old\n", c_sname, c_age);
} while (SQLSTATE != '02000');
EXEC SQL CLOSE sinfo;
```

## Dynamic SQL

- SQL query strings are now always known at compile time (e.g., spreadsheet, graphical DBMS frontend): Allow construction of SQL statements on-the-fly
- Example:

```
char c_sqlstring[]=
   {"DELETE FROM Sailors WHERE raiting>5"};
EXEC SQL PREPARE readytogo FROM :c_sqlstring;
EXEC SQL EXECUTE readytogo;
```



## Database APIs: Alternative to embedding

Rather than modify compiler, add library with database calls (API)

- Special standardized interface: procedures/objects
- Pass SQL strings from language, presents result sets in a language-friendly way
- Sun's *JDBC:* Java API
- Supposedly DBMS-neutral
  - a "driver" traps the calls and translates them into DBMS-specific code
  - database can be across a network



## JDBC: Architecture

- Four architectural components:
  - Application (initiates and terminates connections, submits SQL statements)
  - Driver manager (load JDBC driver)
  - Driver (connects to data source, transmits requests and returns/translates results and error codes)
  - Data source (processes SQL statements)

## *JDBC Architecture (Contd.)*

Four types of drivers:

Bridge:

- Translates SQL commands into non-native API. Example: JDBC-ODBC bridge. Code for ODBC and JDBC driver needs to be available on each client.

Direct translation to native API, non-Java driver:

- Translates SQL commands to native API of data source. Need OS-specific binary on each client.

Network bridge:

- Send commands over the network to a middleware server that talks to the data source. Needs only small JDBC driver at each client.

Direction translation to native API via Java driver:

- Converts JDBC calls directly to network protocol used by DBMS. Needs DBMS-specific Java driver at each client.



## *JDBC Classes and Interfaces*

Steps to submit a database query:

- Load the JDBC driver
- Connect to the data source
- Execute SQL statements



## *JDBC Driver Management*

- All drivers are managed by the DriverManager class
- Loading a JDBC driver:
  - In the Java code: Class.forName("oracle/jdbc.driver.Oracledriver");
  - When starting the Java application: -Djdbc.drivers=oracle/jdbc.driver

## *Connections in JDBC*

We interact with a data source through sessions. Each connection identifies a logical session.

- JDBC URL:
  jdbc:<subprotocol>:<otherParameters>

Example:

```
String url="jdbc:oracle:www.bookstore.com:3083";
Connection con;
try{
    con = DriverManager.getConnection(url,usedId,password);
} catch SQLException excpt { …}
```



## *Connection Class Interface*

- public int getTransactionIsolation() and
  void setTransactionIsolation(int level)
  Sets isolation level for the current connection.
- public boolean getReadOnly() and
  void setReadOnly(boolean b)
  Specifies whether transactions in this connection are read-only
- public boolean getAutoCommit() and
  void setAutoCommit(boolean b)
  If autocommit is set, then each SQL statement is considered its own transaction. Otherwise, a transaction is committed using commit(), or aborted using rollback().
- public boolean isClosed()
  Checks whether connection is still open.



## *Executing SQL Statements*

- Three different ways of executing SQL statements:
  - Statement (both static and dynamic SQL statements)
  - PreparedStatement (semi-static SQL statements)
  - CallableStatment (stored procedures)
- PreparedStatement class:
  Precompiled, parametrized SQL statements:
  - Structure is fixed
  - Values of parameters are determined at run-time

## Executing SQL Statements (Contd.)

```
String sql="INSERT INTO Sailors VALUES(?,?,?,?)";
PreparedStatment pstmt=con.prepareStatement(sql);
pstmt.clearParameters();
pstmt.setInt(1,sid);
pstmt.setString(2,sname);
pstmt.setInt(3, rating);
pstmt.setFloat(4,age);

// we know that no rows are returned, thus we use
   executeUpdate()
int numRows = pstmt.executeUpdate();
```



## ResultSets

- PreparedStatement.executeUpdate only returns the number of affected records
- PreparedStatement.executeQuery returns data, encapsulated in a ResultSet object (a cursor)

```
ResultSet rs=pstmt.executeQuery(sql);
// rs is now a cursor
While (rs.next()) {
  // process the data
}
```



## ResultSets (Contd.)

A ResultSet is a very powerful cursor:

- previous(): moves one row back
- absolute(int num): moves to the row with the specified number
- relative (int num): moves forward or backward
- first() and last()

## *Matching Java and SQL Data Types*

| SQL Type | Java class | ResultSet get method |
|---|---|---|
| BIT | Boolean | getBoolean() |
| CHAR | String | getString() |
| VARCHAR | String | getString() |
| DOUBLE | Double | getDouble() |
| FLOAT | Double | getDouble() |
| INTEGER | Integer | getInt() |
| REAL | Double | getFloat() |
| DATE | java.sql.Date | getDate() |
| TIME | java.sql.Time | getTime() |
| TIMESTAMP | java.sql.TimeStamp | getTimestamp() |



## *JDBC: Exceptions and Warnings*

- Most of java.sql can throw and SQLException if an error occurs.
- SQLWarning is a subclass of EQLException; not as severe (they are not thrown and their existence has to be explicitly tested)



## *Warning and Exceptions (Contd.)*

```
try {
   stmt=con.createStatement();
   warning=con.getWarnings();
   while(warning != null) {
      // handle SQLWarnings;
      warning = warning.getNextWarning():
   }
   con.clearWarnings();
   stmt.executeUpdate(queryString);
   warning = con.getWarnings();
   …
} //end try
catch( SQLException SQLe) {
   // handle the exception
}
```

## *Examining Database Metadata*

DatabaseMetaData object gives information about the database system and the catalog.

```
DatabaseMetaData md = con.getMetaData();
// print information about the driver:
System.out.println(
   "Name:" + md.getDriverName() +
   "version: " + md.getDriverVersion());
```



## *Database Metadata (Contd.)*

```
DatabaseMetaData md=con.getMetaData();
ResultSet trs=md.getTables(null,null,null,null);
String tableName;
While(trs.next()) {
   tableName = trs.getString("TABLE_NAME");
   System.out.println("Table: " + tableName);
   //print all attributes
   ResultSet crs = md.getColumns(null,null,tableName, null);
   while (crs.next()) {
      System.out.println(crs.getString("COLUMN_NAME" + ", ");
   }
}
```



## *A (Semi-)Complete Example*

```
Connection con = // connect
  DriverManager.getConnection(url, "login", "pass");
Statement stmt = con.createStatement(); // set up stmt
String query = "SELECT name, rating FROM Sailors";
ResultSet rs = stmt.executeQuery(query);
try { // handle exceptions
   // loop through result tuples
   while (rs.next()) {
      String s = rs.getString("name");
      Int n = rs.getFloat("rating");
      System.out.println(s + "   " + n);
   }
} catch(SQLException ex) {
   System.out.println(ex.getMessage ()
      + ex.getSQLState () + ex.getErrorCode ());
}
```

## SQLJ

Complements JDBC with a (semi-)static query model: Compiler can perform syntax checks, strong type checks, consistency of the query with the schema

- All arguments always bound to the same variable:
```
#sql = {
   SELECT name, rating INTO :name, :rating
   FROM Books WHERE sid = :sid;
```
- Compare to JDBC:
```
sid=rs.getInt(1);
if (sid==1) {sname=rs.getString(2);}
else { sname2=rs.getString(2);}
```

- SQLJ (part of the SQL standard) versus embedded SQL (vendor-specific)



## SQLJ Code

```
Int sid; String name; Int rating;
// named iterator
#sql iterator Sailors(Int sid, String name, Int rating);
Sailors sailors;
// assume that the application sets rating
#sailors = {
   SELECT sid, sname INTO :sid, :name
    FROM Sailors WHERE rating = :rating
};
// retrieve results
while (sailors.next()) {
   System.out.println(sailors.sid + " " + sailors.sname));
}
sailors.close();
```



## SQLJ Iterators

Two types of iterators ("cursors"):

- Named iterator
  - Need both variable type and name, and then allows retrieval of columns by name.
  - See example on previous slide.
- Positional iterator
  - Need only variable type, and then uses FETCH .. INTO construct:
```
#sql iterator Sailors(Int, String, Int);
Sailors sailors;
#sailors = …
while (true) {
   #sql {FETCH :sailors INTO :sid, :name} ;
   if (sailors.endFetch()) { break; }
   // process the sailor
}
```

## *Stored Procedures*

- What is a stored procedure:
  - Program executed through a single SQL statement
  - Executed in the process space of the server
- Advantages:
  - Can encapsulate application logic while staying "close" to the data
  - Reuse of application logic by different users
  - Avoid tuple-at-a-time return of records through cursors



## *Stored Procedures: Examples*

```
CREATE PROCEDURE ShowNumReservations
   SELECT S.sid, S.sname, COUNT(*)
   FROM Sailors S, Reserves R
   WHERE S.sid = R.sid
   GROUP BY S.sid, S.sname
```

Stored procedures can have parameters:

- Three different modes: IN, OUT, INOUT

```
CREATE PROCEDURE IncreaseRating(
   IN sailor_sid INTEGER, IN increase INTEGER)
UPDATE Sailors
   SET rating = rating + increase
   WHERE sid = sailor_sid
```



## *Stored Procedures: Examples (Contd.)*

Stored procedure do not have to be written in SQL:

```
CREATE PROCEDURE TopSailors(
   IN num INTEGER)
LANGUAGE JAVA
EXTERNAL NAME "file:///c:/storedProcs/rank.jar"
```

## *Calling Stored Procedures*

```
EXEC SQL BEGIN DECLARE SECTION
Int sid;
Int rating;
EXEC SQL END DECLARE SECTION

// now increase the rating of this sailor
EXEC CALL IncreaseRating(:sid,:rating);
```



## *Calling Stored Procedures (Contd.)*

JDBC:

```
CallableStatement cstmt=
    con.prepareCall("{call
    ShowSailors});
ResultSet rs =
    cstmt.executeQuery();
while (rs.next()) {
  …
}
```

SQLJ:

```
#sql iterator
    ShowSailors(…);
ShowSailors showsailors;
#sql showsailors={CALL
    ShowSailors};
while (showsailors.next()) {
  …
}
```



## *SQL/PSM*

Most DBMSs allow users to write stored procedures in a simple, general-purpose language (close to SQL) à SQL/PSM standard is a representative

**Declare a stored procedure:**

```
CREATE PROCEDURE name(p1, p2, …, pn)
    local variable declarations
    procedure code;
```

**Declare a function:**

```
CREATE FUNCTION name (p1, …, pn) RETURNS
    sqlDataType
    local variable declarations
    function code;
```

## Main SQL/PSM Constructs

```
CREATE FUNCTION rate Sailor
      (IN sailorId INTEGER)
      RETURNS INTEGER
DECLARE rating INTEGER
DECLARE numRes INTEGER
SET numRes = (SELECT COUNT(*)
                    FROM Reserves R
                    WHERE R.sid = sailorId)
IF (numRes > 10) THEN rating =1;
ELSE rating = 0;
END IF;
RETURN rating;
```



## Main SQL/PSM Constructs (Contd.)

- Local variables (DECLARE)
- RETURN values for FUNCTION
- Assign variables with SET
- Branches and loops:
  - IF (condition) THEN statements;
    ELSEIF (condition) statements;
    … ELSE statements; END IF;
  - LOOP statements; END LOOP
- Queries can be parts of expressions
- Can use cursors naturally without "EXEC SQL"



## Summary

- Embedded SQL allows execution of parametrized static queries within a host language
- Dynamic SQL allows execution of completely ad-hoc queries within a host language
- Cursor mechanism allows retrieval of one record at a time and bridges impedance mismatch between host language and SQL
- APIs such as JDBC introduce a layer of abstraction between application and DBMS

## *Summary (Contd.)*

- SQLJ: Static model, queries checked a compile-time.
- Stored procedures execute application logic directly at the server
- SQL/PSM standard for writing stored procedures

# Internet Applications

Chapter 7



## Lecture Overview

- Internet Concepts
- Web data formats
  - HTML, XML, DTDs
- Introduction to three-tier architectures
- The presentation layer
  - HTML forms; HTTP Get and POST, URL encoding; Javascript; Stylesheets. XSLT
- The middle tier
  - CGI, application servers, Servlets, JavaServerPages, passing arguments, maintaining state (cookies)



## Uniform Resource Identifiers

- Uniform naming schema to identify *resources* on the Internet
- A resource can be anything:
  - Index.html
  - mysong.mp3
  - picture.jpg
- Example URIs:

  http://www.cs.wisc.edu/~dbbook/index.html
  mailto:webmaster@bookstore.com

## Structure of URIs

http://www.cs.wisc.edu/~dbbook/index.html

- URI has three parts:
  - Naming schema (http)
  - Name of the host computer (www.cs.wisc.edu)
  - Name of the resource (~dbbook/index.html)
- URLs are a subset of URIs



## Hypertext Transfer Protocol

- What is a communication protocol?
  - Set of standards that defines the structure of messages
  - Examples: TCP, IP, HTTP
- What happens if you click on www.cs.wisc.edu/~dbbook/index.html?
- Client (web browser) sends HTTP request to server
- Server receives request and replies
- Client receives reply; makes new requests



## HTTP (Contd.)

Client to Server:

GET ~/index.html HTTP/1.1
User-agent: Mozilla/4.0
Accept: text/html, image/gif, image/jpeg

Server replies:

HTTP/1.1 200 OK
Date: Mon, 04 Mar 2002 12:00:00 GMT
Server: Apache/1.3.0 (Linux)
Last-Modified: Mon, 01 Mar 2002 09:23:24 GMT
Content-Length: 1024
Content-Type: text/html
<HTML> <HEAD></HEAD>
<BODY>
<h1>Barns and Nobble Internet Bookstore</h1>
Our inventory:
<h3>Science</h3>
<b>The Character of Physical Law</b>
...

## HTTP Protocol Structure

HTTP Requests

- Request line: GET ~/index.html HTTP/1.1
  - GET: Http method field (possible values are GET and POST, more later)
  - ~/index.html: URI field
  - HTTP/1.1: HTTP version field
- Type of client: User-agent: Mozilla/4.0
- What types of files will the client accept:
  Accept: text/html, image/gif, image/jpeg



## HTTP Protocol Structure (Contd.)

HTTP Responses

- Status line: HTTP/1.1 200 OK
  - HTTP version: HTTP/1.1
  - Status code: 200
  - Server message: OK
  - Common status code/server message combinations:
    - 200 OK: Request succeeded
    - 400 Bad Request: Request could not be fulfilled by the server
    - 404 Not Found: Requested object does not exist on the server
    - 505 HTTP Version not Supported
- Date when the object was created:
  Last-Modified: Mon, 01 Mar 2002 09:23:24 GMT
- Number of bytes being sent: Content-Length: 1024
- What type is the object being sent: Content-Type: text/html
- Other information such as the server type, server time, etc.



## Some Remarks About HTTP

- HTTP is stateless
  - No "sessions"
  - Every message is completely self-contained
  - No previous interaction is "remembered" by the protocol
  - Tradeoff between ease of implementation and ease of application development: Other functionality has to be built on top
- Implications for applications:
  - Any state information (shopping carts, user login-information) need to be encoded in every HTTP request and response!
  - Popular methods on how to maintain state:
    - Cookies (later this lecture)
    - Dynamically generate unique URL's at the server level (later this lecture)

# Web Data Formats

- HTML
  - The presentation language for the Internet
- Xml
  - A self-describing, hierarchal data model
- DTD
  - Standardizing schemas for Xml
- XSLT (not covered in the book)



# HTML: An Example

```
<HTML>
  <HEAD></HEAD>
  <BODY>
  <h1>Barns and Nobble Internet
    Bookstore</h1>
  Our inventory:

  <h3>Science</h3>
  <b>The Character of Physical
    Law</b>
  <UL>
     <LI>Author: Richard
    Feynman</LI>
    <LI>Published 1980</LI>
    <LI>Hardcover</LI>
  </UL>
  <h3>Fiction</h3>
  <b>Waiting for the Mahatma</b>
  <UL>
    <LI>Author: R.K. Narayan</LI>
    <LI>Published 1981</LI>
  </UL>
  <b>The English Teacher</b>
  <UL>
    <LI>Author:  R.K. Narayan</LI>
    <LI>Published 1980</LI>
    <LI>Paperback</LI>
  </UL>

  </BODY>
</HTML>
```



# HTML: A Short Introduction

- HTML is a markup language
- Commands are tags:
  - Start tag and end tag
  - Examples:
    - <HTML> … </HTML>
    - <UL> … </UL>
- Many editors automatically generate HTML directly from your document (e.g., Microsoft Word has an “Save as html” facility)

## *HTML: Sample Commands*

- <HTML>:
- <UL>: unordered list
- <LI>: list entry
- <h1>: largest heading
- <h2>: second-level heading, <h3>, <h4> analogous
- <B>Title</B>: Bold



## *XML: An Example*

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<BOOKLIST>
  <BOOK genre="Science" format="Hardcover">
    <AUTHOR>
      <FIRSTNAME>Richard</FIRSTNAME><LASTNAME>Feynman</LASTNAME>
    </AUTHOR>
    <TITLE>The Character of Physical Law</TITLE>
    <PUBLISHED>1980</PUBLISHED>
  </BOOK>
  <BOOK genre="Fiction">
    <AUTHOR>
      <FIRSTNAME>R.K.</FIRSTNAME><LASTNAME>Narayan</LASTNAME>
    </AUTHOR>
    <TITLE>Waiting for the Mahatma</TITLE>
    <PUBLISHED>1981</PUBLISHED>
  </BOOK>
  <BOOK genre="Fiction">
    <AUTHOR>
      <FIRSTNAME>R.K.</FIRSTNAME><LASTNAME>Narayan</LASTNAME>
    </AUTHOR>
    <TITLE>The English Teacher</TITLE>
    <PUBLISHED>1980</PUBLISHED>
  </BOOK>
</BOOKLIST>
```



## *XML – The Extensible Markup Language*

- Language
  - A way of communicating information
- Markup
  - Notes or meta-data that describe your data or language
- Extensible
  - Limitless ability to define new languages or data sets

## XML – What's The Point?

- You can include your data and a description of what the data represents
  - This is useful for defining your own language or protocol
- Example: Chemical Markup Language

```
<molecule>
        <weight>234.5</weight>
        <Spectra>…</Spectra>
        <Figures>…</Figures>
</molecule>
```

- XML design goals:
  - XML should be compatible with SGML
  - It should be easy to write XML processors
  - The design should be formal and precise



## XML – Structure

- XML: Confluence of SGML and HTML
- Xml looks like HTML
- Xml is a hierarchy of user-defined tags called elements with attributes and data
- Data is described by elements, elements are described by attributes

```
<BOOK genre="Science" format="Hardcover">…</BOOK>
```

attribute

open tag
element name

attribute value

data

closing tag



## XML – Elements

```
<BOOK genre="Science" format="Hardcover">…</BOOK>
```

attribute

open tag
element name

attribute value

data

closing tag

- Xml is case and space sensitive
- Element opening and closing tag names must be identical
- Opening tags: "<" + element name + ">"
- Closing tags: "</" + element name + ">"
- Empty Elements have no data and no closing tag:
  - They begin with a "<" and end with a "/>"

```
<BOOK/>
```

# XML – Attributes

<BOOK genre="Science" format="Hardcover">…</BOOK>

open tag element name · attribute · attribute value · data · closing tag

- Attributes provide additional information for element tags.
- There can be zero or more attributes in every element; each one has the the form:

  *attribute_name*=‘*attribute_value*’
  - There is no space between the name and the “=‘”
  - Attribute values must be surrounded by “ or ‘ characters
- Multiple attributes are separated by white space (one or more spaces or tabs).



# XML – Data and Comments

<BOOK genre="Science" format="Hardcover">…</BOOK>

open tag element name · attribute · attribute value · closing tag · data

- Xml data is any information between an opening and closing tag
- Xml data must not contain the ‘<‘ or ‘>’ characters

- Comments:
  <!- comment ->



# XML – Nesting & Hierarchy

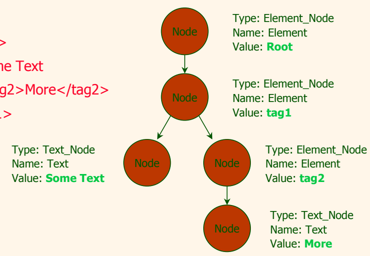
- Xml tags can be nested in a tree hierarchy
- Xml documents can have only one root tag
- Between an opening and closing tag you can insert:
  1. Data
  2. More Elements
  3. A combination of data and elements

```
<root>
  <tag1>
    Some Text
    <tag2>More</tag2>
  </tag1>
</root>
```

## *Xml – Storage*

- Storage is done just like an n-ary tree (DOM)

```
<root>
   <tag1>
      Some Text
      <tag2>More</tag2>
   </tag1>
</root>
```






## *DTD – Document Type Definition*

- A DTD is a schema for Xml data
- Xml protocols and languages can be standardized with DTD files
- A DTD says what elements and attributes are required or optional
  - Defines the formal structure of the language



## *DTD – An Example*

```
<?xml version='1.0'?>
<!ELEMENT Basket (Cherry+, (Apple | Orange)*) >
   <!ELEMENT Cherry EMPTY>
      <!ATTLIST Cherry flavor CDATA #REQUIRED>
   <!ELEMENT Apple EMPTY>
      <!ATTLIST Apple color CDATA #REQUIRED>
   <!ELEMENT Orange EMPTY>
      <!ATTLIST Orange location 'Florida'>
--------------------------------------------------------------------------------
```

```
<Basket>
   <Cherry flavor='good'/>
   <Apple color='red'/>
   <Apple color='green'/>
</Basket>
```

```
<Basket>
   <Apple/>
   <Cherry flavor='good'/>
   <Orange/>
</Basket>
```

## DTD - !ELEMENT

<!ELEMENT Basket (Cherry+, (Apple | Orange)*) >

Name Children

- !ELEMENT declares an element name, and what children elements it should have
- Content types:
  - Other elements
  - #PCDATA (parsed character data)
  - EMPTY (no content)
  - ANY (no checking inside this structure)
  - A regular expression



## DTD - !ELEMENT (Contd.)

- A regular expression has the following structure:
  - $exp_1$, $exp_2$, $exp_3$, …, $exp_k$: A list of regular expressions
  - exp*: An optional expression with zero or more occurrences
  - exp+: An optional expression with one or more occurrences
  - $exp_1$ | $exp_2$ | … | $exp_k$: A disjunction of expressions



## DTD - !ATTLIST

<!ATTLIST Cherry flavor CDATA #REQUIRED>

Element Attribute Type Flag

<!ATTLIST Orange location CDATA #REQUIRED
color 'orange'>

- !ATTLIST defines a list of attributes for an element
- Attributes can be of different types, can be required or not required, and they can have default values.

## DTD – Well-Formed and Valid

```
<?xml version='1.0'?>
<!ELEMENT Basket (Cherry+)>
   <!ELEMENT Cherry EMPTY>
        <!ATTLIST Cherry flavor CDATA #REQUIRED>
------------------------------------------------------------------------------
```

Not Well-Formed

```
<basket>
   <Cherry flavor=good>
</Basket>
```

Well-Formed but Invalid

```
<Job>
   <Location>Home</Location>
</Job>
```

Well-Formed and Valid

```
<Basket>
   <Cherry flavor='good'/>
</Basket>
```



## XML and DTDs

- More and more standardized DTDs will be developed
  - MathML
  - Chemical Markup Language
- Allows light-weight exchange of data with the same semantics
- Sophisticated query languages for XML are available:
  - Xquery
  - XPath



## Lecture Overview

- Internet Concepts
- Web data formats
  - HTML, XML, DTDs
- Introduction to three-tier architectures
- The presentation layer
  - HTML forms; HTTP Get and POST, URL encoding; Javascript; Stylesheets. XSLT
- The middle tier
  - CGI, application servers, Servlets, JavaServerPages, passing arguments, maintaining state (cookies)

## *Components of Data-Intensive Systems*

Three separate types of functionality:

- Data management
- Application logic
- Presentation

- The system architecture determines whether these three components reside on a single system ("tier) or are distributed across several tiers



## *Single-Tier Architectures*

All functionality combined into a single tier, usually on a mainframe

- User access through dumb terminals

Advantages:

- Easy maintenance and administration

Disadvantages:

- Today, users expect graphical user interfaces.
- Centralized computation of all of them is too much for a central system

- GRAPHIC



## *Client-Server Architectures*

Work division: Thin client

- Client implements only the graphical user interface
- Server implements business logic and data management

- Work division: Thick client
  - Client implements both the graphical user interface and the business logic
  - Server implements data management

- GRAPHIC

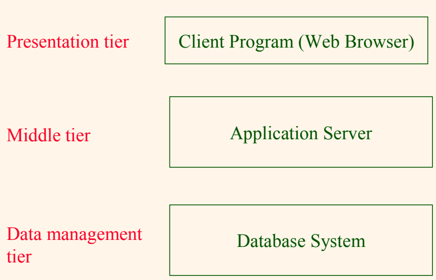
## Client-Server Architectures (Contd.)

Disadvantages of thick clients

- No central place to update the business logic
- Security issues: Server needs to trust clients
  - Access control and authentication needs to be managed at the server
  - Clients need to leave server database in consistent state
  - One possibility: Encapsulate all database access into stored procedures
- Does not scale to more than several 100s of clients
  - Large data transfer between server and client
  - More than one server creates a problem: x clients, y servers: x*y connections



## The Three-Tier Architecture






## The Three Layers

Presentation tier

- Primary interface to the user
- Needs to adapt to different display devices (PC, PDA, cell phone, voice access?)

Middle tier

- Implements business logic (implements complex actions, maintains state between different steps of a workflow)
- Accesses different data management systems

Data management tier

- One or more standard database management systems

## Example 1: Airline reservations

- Build a system for making airline reservations
- What is done in the different tiers?
- Database System
  - Airline info, available seats, customer info, etc.
- Application Server
  - Logic to make reservations, cancel reservations, add new airlines, etc.
- Client Program
  - Log in different users, display forms and human-readable output



## Example 2: Course Enrollment

- Build a system using which students can enroll in courses
- Database System
  - Student info, course info, instructor info, course availability, pre-requisites, etc.
- Application Server
  - Logic to add a course, drop a course, create a new course, etc.
- Client Program
  - Log in different users (students, staff, faculty), display forms and human-readable output



## Technologies

Client Program
*(Web Browser)*

*HTML*
*Javascript*
*XSLT*

Application Server
*(Tomcat, Apache)*

*JSP*
*Servlets*
*Cookies*
*CGI*

Database System
*(DB2)*

*XML*
*Stored Procedures*

## Advantages of the Three-Tier Architecture

- Heterogeneous systems
  - Tiers can be independently maintained, modified, and replaced
- Thin clients
  - Only presentation layer at clients (web browsers)
- Integrated data access
  - Several database systems can be handled transparently at the middle tier
  - Central management of connections
- Scalability
  - Replication at middle tier permits scalability of business logic
- Software development
  - Code for business logic is centralized
  - Interaction between tiers through well-defined APIs: Can reuse standard components at each tier



## Lecture Overview

- Internet Concepts
- Web data formats
  - HTML, XML, DTDs
- Introduction to three-tier architectures
- The presentation layer
  - HTML forms; HTTP Get and POST, URL encoding; Javascript; Stylesheets. XSLT
- The middle tier
  - CGI, application servers, Servlets, JavaServerPages, passing arguments, maintaining state (cookies)



## Overview of the Presentation Tier

- Recall: Functionality of the presentation tier
  - Primary interface to the user
  - Needs to adapt to different display devices (PC, PDA, cell phone, voice access?)
  - Simple functionality, such as field validity checking
- We will cover:
  - HTML Forms: How to pass data to the middle tier
  - JavaScript: Simple functionality at the presentation tier
  - Style sheets: Separating data from formatting

## HTML Forms

- Common way to communicate data from client to middle tier
- General format of a form:
  - <FORM ACTION="page.jsp" METHOD="GET"
            NAME="LoginForm">
    …
    </FORM>
- Components of an HTML FORM tag:
  - ACTION: Specifies URI that handles the content
  - METHOD: Specifies HTTP GET or POST method
  - NAME: Name of the form; can be used in client-side scripts to refer to the form



## Inside HTML Forms

- INPUT tag
  - Attributes:
    - TYPE: text (text input field), password (text input field where input is, reset (resets all input fields)
    - NAME: symbolic name, used to identify field value at the middle tier
    - VALUE: default value
  - Example: <INPUT TYPE="text" Name="title">
- Example form:

```
<form method="POST" action="TableOfContents.jsp">
  <input type="text" name="userid">
  <input type="password" name="password">
  <input type="submit" value="Login" name="submit">
  <input type="reset" value="Clear">
</form>
```



## Passing Arguments

Two methods: GET and POST

- GET
  - Form contents go into the submitted URI
  - Structure:
    action?name1=value1&name2=value2&name3=value3
    - Action: name of the URI specified in the form
    - (name,value)-pairs come from INPUT fields in the form; empty fields have empty values ("name=")
  - Example from previous password form:
    TableOfContents.jsp?userid=john&password=johnpw
  - Note that the page named action needs to be a program, script, or page that will process the user input

## HTTP GET: Encoding Form Fields

- Form fields can contain general ASCII characters that cannot appear in an URI
- A special encoding convention converts such field values into “URI-compatible” characters:
  - Convert all “special” characters to %xyz, were xyz is the ASCII code of the character. Special characters include &, =, +, %, etc.
  - Convert all spaces to the “+” character
  - Glue (name,value)-pairs from the form INPUT tags together with “&” to form the URI



## HTML Forms: A Complete Example

```
<form method="POST" action="TableOfContents.jsp">
    <table align = "center" border="0" width="300">
    <tr>
          <td>Userid</td>
          <td><input type="text" name="userid" size="20"></td>
    </tr>
    <tr>
          <td>Password</td>
          <td><input type="password" name="password" size="20"></td>
    </tr>
    <tr>
          <td align = "center"><input type="submit" value="Login“
                    name="submit"></td>
    </tr>
    </table>
</form>
```



## JavaScript

- Goal: Add functionality to the presentation tier.
- Sample applications:
  - Detect browser type and load browser-specific page
  - Form validation: Validate form input fields
  - Browser control: Open new windows, close existing windows (example: pop-up ads)
- Usually embedded directly inside the HTML with the <SCRIPT> … </SCRIPT> tag.
- <SCRIPT> tag has several attributes:
  - LANGUAGE: specifies language of the script (such as javascript)
  - SRC: external file with script code
  - Example:
    <SCRIPT LANGUAGE=“JavaScript” SRC=“validate.js>
    </SCRIPT>

## JavaScript (Contd.)

- If <SCRIPT> tag does not have a SRC attribute, then the JavaScript is directly in the HTML file.
- Example:

```
<SCRIPT LANGUAGE="JavaScript">
<!-- alert("Welcome to our bookstore")
//-->
</SCRIPT>
```

- Two different commenting styles
  - <!-- comment for HTML, since the following JavaScript code should be ignored by the HTML processor
  - // comment for JavaScript in order to end the HTML comment



## JavaScript (Contd.)

- JavaScript is a complete scripting language
  - Variables
  - Assignments (=, +=, …)
  - Comparison operators (<,>,…), boolean operators (&&, | |, !)
  - Statements
    - if (condition) {statements;} else {statements;}
    - for loops, do-while loops, and while-loops
  - Functions with return values
    - Create functions using the function keyword
    - f(arg1, …, argk) {statements;}



## JavaScript: A Complete Example

HTML Form:

```
<form method="POST"
  action="TableOfContents.jsp">
  <input type="text"
    name="userid">
  <input type="password"
    name="password">
  <input type="submit"
    value="Login"
    name="submit">
  <input type="reset"
    value="Clear">
</form>
```

Associated JavaScript:

```
<script language="javascript">
function testLoginEmpty()
{
 loginForm = document.LoginForm
 if ((loginForm.userid.value == "") ||
   (loginForm.password.value == ""))
 {
  alert('Please enter values for userid and
   password.');
  return false;
 }
 else return true;
}
</script>
```

## Stylesheets

- Idea: Separate display from contents, and adapt display to different presentation formats
- Two aspects:
  - Document transformations to decide what parts of the document to display in what order
  - Document rending to decide how each part of the document is displayed
- Why use stylesheets?
  - Reuse of the same document for different displays
  - Tailor display to user's preferences
  - Reuse of the same document in different contexts
- Two stylesheet languages
  - Cascading style sheets (CSS): For HTML documents
  - Extensible stylesheet language (XSL): For XML documents



## CSS: Cascading Style Sheets

- Defines how to display HTML documents
- Many HTML documents can refer to the same CSS
  - Can change format of a website by changing a single style sheet
  - Example:
    <LINK REL="style sheet" TYPE="text/css" HREF="books.css"/>

Each line consists of three parts:

selector {property: value}

- Selector: Tag whose format is defined
- Property: Tag's attribute whose value is set
- Value: value of the attribute



## CSS: Cascading Style Sheets

Example style sheet:

```
body {background-color: yellow}
h1 {font-size: 36pt}
h3 {color: blue}
p {margin-left: 50px; color: red}
```

The first line has the same effect as:

<body background-color="yellow>

## *XSL*

- Language for expressing style sheets
  - More at: http://www.w3.org/Style/XSL/
- Three components
  - XSLT: XSL Transformation language
    - Can transform one document to another
    - More at http://www.w3.org/TR/xslt
  - XPath: XML Path Language
    - Selects parts of an XML document
    - More at http://www.w3.org/TR/xpath
  - XSL Formatting Objects
    - Formats the output of an XSL transformation
    - More at http://www.w3.org/TR/xsl/



## *Lecture Overview*

- Internet Concepts
- Web data formats
  - HTML, XML, DTDs
- Introduction to three-tier architectures
- The presentation layer
  - HTML forms; HTTP Get and POST, URL encoding; Javascript; Stylesheets. XSLT
- The middle tier
  - CGI, application servers, Servlets, JavaServerPages, passing arguments, maintaining state (cookies)



## *Overview of the Middle Tier*

- Recall: Functionality of the middle tier
  - Encodes business logic
  - Connects to database system(s)
  - Accepts form input from the presentation tier
  - Generates output for the presentation tier
- We will cover
  - CGI: Protocol for passing arguments to programs running at the middle tier
  - Application servers: Runtime environment at the middle tier
  - Servlets: Java programs at the middle tier
  - JavaServerPages: Java scripts at the middle tier
  - Maintaining state: How to maintain state at the middle tier. Main focus: Cookies.

## CGI: Common Gateway Interface

- Goal: Transmit arguments from HTML forms to application programs running at the middle tier
- Details of the actual CGI protocol unimportant à libraries implement high-level interfaces
- Disadvantages:
  - The application program is invoked in a new process at every invocation (remedy: FastCGI)
  - No resource sharing between application programs (e.g., database connections)
  - Remedy: Application servers



## CGI: Example

- HTML form:

```
<form action="findbooks.cgi" method=POST>
Type an author name:
<input type="text" name="authorName">
<input type="submit" value="Send it">
<input type="reset" value="Clear form">
</form>
```

- Perl code:

```
use CGI;
$dataIn=new CGI;
$dataIn->header();
$authorName=$dataIn->param('authorName');
print("<HTML><TITLE>Argument passing test</TITLE>");
print("The author name is " + $authorName);
print("</HTML>");
exit;
```

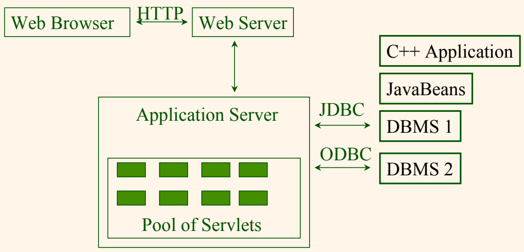


## Application Servers

- Idea: Avoid the overhead of CGI
  - Main pool of threads of processes
  - Manage connections
  - Enable access to heterogeneous data sources
  - Other functionality such as APIs for session management

## *Application Server: Process Structure*

Process Structure






## *Servlets*

- Java Servlets: Java code that runs on the middle tier
  - Platform independent
  - Complete Java API available, including JDBC

Example:

```
import java.io.*;
import java.servlet.*;
import java.servlet.http.*;

public class ServetTemplate extends HttpServlet {
    public void doGet(HTTPServletRequest request,
                                  HTTPServletResponse response)
    throws SerletExpection, IOException {
          PrintWriter out=response.getWriter();
          out.println("Hello World");
    }
}
```



## *Servlets (Contd.)*

- Life of a servlet?
  - Webserver forwards request to servlet container
  - Container creates servlet instance (calls init() method; deallocation time: calls destroy() method)
  - Container calls service() method
    - service() calls doGet() for HTTP GET or doPost() for HTTP POST
    - Usually, don't override service(), but override doGet() and doPost()

## *Servlets: A Complete Example*

```
public class ReadUserName extends HttpServlet {
    public void doGet(        HttpServletRequest request,
                              HttpSevletResponse response)
            throws ServletException, IOException {
            reponse.setContentType(“text/html”);
            PrintWriter out=response.getWriter();
            out.println(“<HTML><BODY>\n <UL> \n” +
                    “<LI>” + request.getParameter(“userid”) + “\n” +
                    “<LI>” + request.getParameter(“password”) + “\n” +
                    “<UL>\n<BODY></HTML>”);
    }
    public void doPost(       HttpServletRequest request,
                              HttpSevletResponse response)
            throws ServletException, IOException {
            doGet(request,response);
    }
}
```



## *Java Server Pages*

- Servlets
  - Generate HTML by writing it to the “PrintWriter” object
  - Code first, webpage second
- JavaServerPages
  - Written in HTML, Servlet-like code embedded in the HTML
  - Webpage first, code second
  - They are usually compiled into a Servlet



## *JavaServerPages: Example*

```
<html>
<head><title>Welcome to B&N</title></head>
<body>
    <h1>Welcome back!</h1>
    <% String name=“NewUser”;
            if (request.getParameter(“username”) != null) {
                    name=request.getParameter(“username”);
            }
    %>
    You are logged on as user <%=name%>
    <p>
</body>
</html>
```

## *Maintaining State*

HTTP is stateless.

- Advantages
  - Easy to use: don't need anything
  - Great for static-information applications
  - Requires no extra memory space
- Disadvantages
  - No record of previous requests means
    - No shopping baskets
    - No user logins
    - No custom or dynamic content
    - Security is more difficult to implement

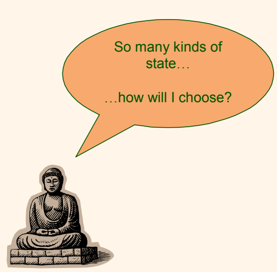


## *Application State*

- Server-side state
  - Information is stored in a database, or in the application layer's local memory
- Client-side state
  - Information is stored on the client's computer in the form of a cookie
- Hidden state
  - Information is hidden within dynamically created web pages



## *Application State*



## *Server-Side State*

- Many types of Server side state:
- 1. Store information in a database
  - Data will be safe in the database
  - BUT: requires a database access to query or update the information
- 2. Use application layer's local memory
  - Can map the user's IP address to some state
  - BUT: this information is volatile and takes up lots of server main memory

5 million IPs = 20 MB



## *Server-Side State*

- Should use Server-side state maintenance for information that needs to persist
  - Old customer orders
  - "Click trails" of a user's movement through a site
  - Permanent choices a user makes



## *Client-side State: Cookies*

- Storing text on the client which will be passed to the application with every HTTP request.
  - Can be disabled by the client.
  - Are wrongfully perceived as "dangerous", and therefore will scare away potential site visitors if asked to enable cookies[1]
- Are a collection of (Name, Value) pairs

[1] http://www.webdevelopersjournal.com/columns/stateful.html

## *Client State: Cookies*

- Advantages
  - Easy to use in Java Servlets / JSP
  - Provide a simple way to persist non-essential data on the client even when the browser has closed
- Disadvantages
  - Limit of 4 kilobytes of information
  - Users can (and often will) disable them
- Should use cookies to store interactive state
  - The current user's login information
  - The current shopping basket
  - Any non-permanent choices the user has made



## *Creating A Cookie*

```
Cookie myCookie =
  new Cookie(“username", “jeffd");
response.addCookie(userCookie);
```

- You can create a cookie at any time



## *Accessing A Cookie*

```
Cookie[] cookies = request.getCookies();
String theUser;
for(int i=0; i<cookies.length; i++) {
  Cookie cookie = cookies[i];
  if(cookie.getName().equals(“username”)) theUser =
  cookie.getValue();
}
// at this point theUser == “username”
```

- Cookies need to be accessed BEFORE you set your response header:

```
response.setContentType("text/html");
PrintWriter out = response.getWriter();
```

## Cookie Features

- Cookies can have
  - A duration (expire right away or persist even after the browser has closed)
  - Filters for which domains/directory paths the cookie is sent to
- See the Java Servlet API and Servlet Tutorials for more information



## Hidden State

- Often users will disable cookies
- You can "hide" data in two places:
  - Hidden fields within a form
  - Using the path information
- Requires no "storage" of information because the state information is passed inside of each web page



## Hidden State: Hidden Fields

- Declare hidden fields within a form:
  - <input type='hidden' name='user' value='username'/>
- Users will not see this information (unless they view the HTML source)
- If used prolifically, it's a killer for performance since EVERY page must be contained within a form.

## *Hidden State: Path Information*

- Path information is stored in the URL request:

  `http://server.com/index.htm?user=jeffd`
- Can separate 'fields' with an & character:

  `index.htm?user=jeffd&preference=pepsi`
- There are mechanisms to parse this field in Java. Check out the `javax.servlet.http.HttpUtils parserQueryString()` method.



## *Multiple state methods*

- Typically all methods of state maintenance are used:
  - User logs in and this information is stored in a cookie
  - User issues a query which is stored in the path information
  - User places an item in a shopping basket cookie
  - User purchases items and credit-card information is stored/retrieved from a database
  - User leaves a click-stream which is kept in a log on the web server (which can later be analyzed)



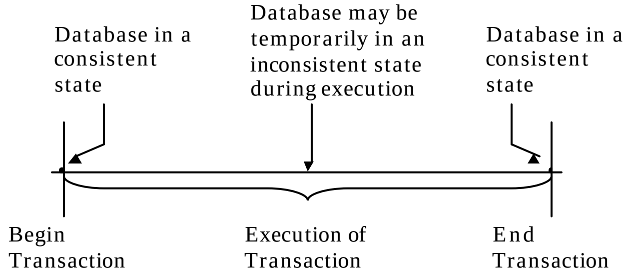
## *Summary*

We covered:

- Internet Concepts (URIs, HTTP)
- Web data formats
  - HTML, XML, DTDs
- Three-tier architectures
- The presentation layer
  - HTML forms; HTTP Get and POST, URL encoding; Javascript; Stylesheets. XSLT
- The middle tier
  - CGI, application servers, Servlets, JavaServerPages, passing arguments, maintaining state (cookies)

# Transaction

- A transaction is a collection of actions that make consistent transformations of system states while preserving system consistency.
  - concurrency transparency
  - failure transparency






# Transaction Example – A Simple SQL Query

```
…
main() {
  …
  EXEC SQL UPDATE Project
     SET Budget = Budget * 1.1
     WHERE Pname = `CAD/CAM';
  EXEC SQL COMMIT RELEASE;
  return(0);
  …}
```

# Example Database

Consider an airline reservation example with the relations:

FLIGHT(FNO, DATE, SRC, DEST, STSOLD, CAP)
CUST(CNAME, ADDR, BAL)
FC(FNO, DATE, CNAME,SPECIAL)



# Example Reservation Transaction

```
…
main {
…
  EXEC SQL BEGIN DECLARE SECTION;
     char flight_no[6], customer_name[20];
     char day;
  EXEC SQL END DECLARE SECTION;
  scanf(flight_no, day, customer_name);
  EXEC SQL  UPDATE FLIGHT
     SET    STSOLD = STSOLD + 1
     WHERE  FNO = :flight_no AND DATE = :day;
  EXEC SQL  INSERT
     INTO   FC(FNO, DATE, CNAME, SPECIAL);
     VALUES(:flight_no,:day,:customer_name, null);
  printf("Reservation completed");
  EXEC SQL COMMIT RELEASE;
  return(0);}
```



# Termination of Transactions

```
…
main {
…
  EXEC SQL BEGIN DECLARE SECTION;
     char flight_no[6], customer_name[20];
     char day; int temp1, temp2;
  EXEC SQL END DECLARE SECTION;
  scanf(flight_no, day, customer_name);
  EXEC SQL    SELECT STSOLD,CAP INTO :temp1,:temp2
      FROM    FLIGHT
      WHERE   FNO = :flight_no AND DATE = :day;
  if temp1 = temp2 then {
      printf(“no free seats”);
      EXEC SQL ROLLBACK RELEASE;
      return(-1);}
  else {
      EXEC SQL UPDATE FLIGHT
        SET    STSOLD = STSOLD + 1
        WHERE  FNO = :flight_no AND DATE = :day;
      EXEC SQL INSERT
        INTO   FC(FNO, DATE, CNAME, SPECIAL);
        VALUES (:flight_no, :day, :customer_name, null);
      EXEC SQL COMMIT RELEASE;
      printf(“Reservation completed”);
      return(0);}
}
```



# Characterization

- Read set (RS)
  - The set of data items that are read by a transaction
- Write set (WS)
  - The set of data items whose values are changed by this transaction
- Base set (BS)
  - RS ∪ WS

# Formalization

Let

- $o_{ij}(x)$ be some operation $o_j$ of transaction $T_i$ operating on data item $x$, where $o_j \in$ {read,write} and $o_j$ is atomic
- $OS_i = \cup_j \, o_{ij}$
- $N_i \in$ {abort,commit}

Transaction $T_i$ is a partial order $T_i = \{\Sigma_i, <_i\}$ where

❶ $\Sigma_i = OS_i \cup \{N_i\}$

❷ For any two operations $o_{ij}, o_{ik} \in OS_i$, if $o_{ij} = R(x)$ and $o_{ik}=W(x)$ for any data item $x$, then either $o_{ij} <_i o_{ik}$ or $o_{ik} <_i o_{ij}$

❸ $\forall o_{ij} \in OS_i, o_{ij} <_i N_i$



# Example

Consider a transaction $T$:

Read($x$)

Read($y$)

$x \leftarrow x + y$

Write($x$)

Commit

Then

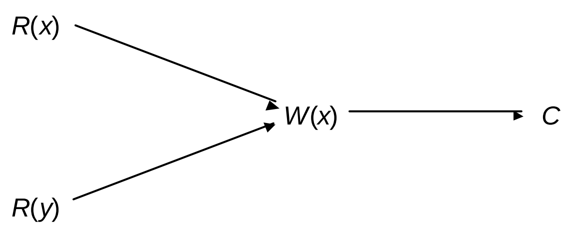
$\Sigma = \{R(x), R(y), W(x), C\}$

$< = \{(R(x), W(x)), (R(y), W(x)), (W(x), C), (R(x), C), (R(y), C)\}$

# DAG Representation

Assume

$< \; = \{(R(x),W(x)), (R(y),W(x)), (R(x), C), (R(y), C), (W(x), C)\}$






# Properties of Transactions

**ATOMICITY**

- all or nothing

**CONSISTENCY**

- no violation of integrity constraints

**ISOLATION**

- concurrent changes invisible $\Rightarrow$ serializable

**DURABILITY**

- committed updates persist

# Atomicity

- Either all or none of the transaction's operations are performed.
- Atomicity requires that if a transaction is interrupted by a failure, its partial results must be undone.
- The activity of preserving the transaction's atomicity in presence of transaction aborts due to input errors, system overloads, or deadlocks is called transaction recovery.
- The activity of ensuring atomicity in the presence of system crashes is called crash recovery.



# Consistency

- Internal consistency
  - A transaction which executes *alone* against a *consistent* database leaves it in a consistent state.
  - Transactions do not violate database integrity constraints.
- Transactions are correct programs

# Isolation

- Serializability
  - If several transactions are executed concurrently, the results must be the same as if they were executed serially in some order.
- Incomplete results
  - An incomplete transaction cannot reveal its results to other transactions before its commitment.
  - Necessary to avoid cascading aborts.



# Isolation Example

- Consider the following two transactions:

| | | | |
|---|---|---|---|
| $T_1$: | Read($x$) | $T_2$: | Read($x$) |
| | $x \leftarrow x+1$ | | $x \leftarrow x+1$ |
| | Write($x$) | | Write($x$) |
| | Commit | | Commit |

- Possible execution sequences:

| | | | |
|---|---|---|---|
| $T_1$: | Read($x$) | $T_1$: | Read($x$) |
| $T_1$: | $x \leftarrow x+1$ | $T_1$: | $x \leftarrow x+1$ |
| $T_1$: | Write($x$) | $T_2$: | Read($x$) |
| $T_1$: | Commit | $T_1$: | Write($x$) |
| $T_2$: | Read($x$) | $T_2$: | $x \leftarrow x+1$ |
| $T_2$: | $x \leftarrow x+1$ | $T_2$: | Write($x$) |
| $T_2$: | Write($x$) | $T_1$: | Commit |
| $T_2$: | Commit | $T_2$: | Commit |

# Consistency Degrees (due to Jim Gray)

- Degree 0
  - Transaction *T* does not overwrite dirty data of other transactions
  - Dirty data refers to data values that have been updated by a transaction prior to its commitment
- Degree 1
  - *T* does not overwrite dirty data of other transactions
  - *T* does not commit any writes before EOT



# Consistency Degrees (cont'd) (due to Jim Gray)

- Degree 2
  - *T* does not overwrite dirty data of other transactions
  - *T* does not commit any writes before EOT
  - *T* does not read dirty data from other transactions
- Degree 3
  - *T* does not overwrite dirty data of other transactions
  - *T* does not commit any writes before EOT
  - *T* does not read dirty data from other transactions
  - Other transactions do not dirty any data read by *T* before *T* completes.

# SQL-92 Isolation Levels

Phenomena:

- Dirty read
  - $T_1$ modifies $x$ which is then read by $T_2$ before $T_1$ terminates; $T_1$ aborts $\Rightarrow$ $T_2$ has read value which never exists in the database.
- Non-repeatable (fuzzy) read
  - $T_1$ reads $x$; $T_2$ then modifies or deletes x and commits. $T_1$ tries to read $x$ again but reads a different value or can't find it.
- Phantom
  - $T_1$ searches the database according to a predicate while $T_2$ inserts new tuples that satisfy the predicate.



# SQL-92 Isolation Levels (cont'd)

- Read Uncommitted
  - For transactions operating at this level, all three phenomena are possible.
- Read Committed
  - Fuzzy reads and phantoms are possible, but dirty reads are not.
- Repeatable Read
  - Only phantoms possible.
- Anomaly Serializable
  - None of the phenomena are possible.

## Durability

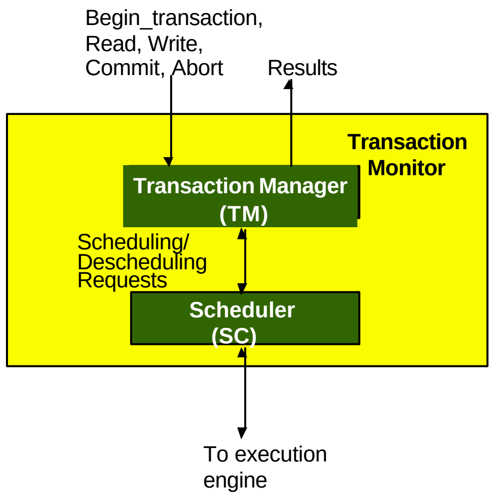
- Once a transaction commits, the system must guarantee that the results of its operations will never be lost, in spite of subsequent failures.
- Database recovery



## Transactions Provide…

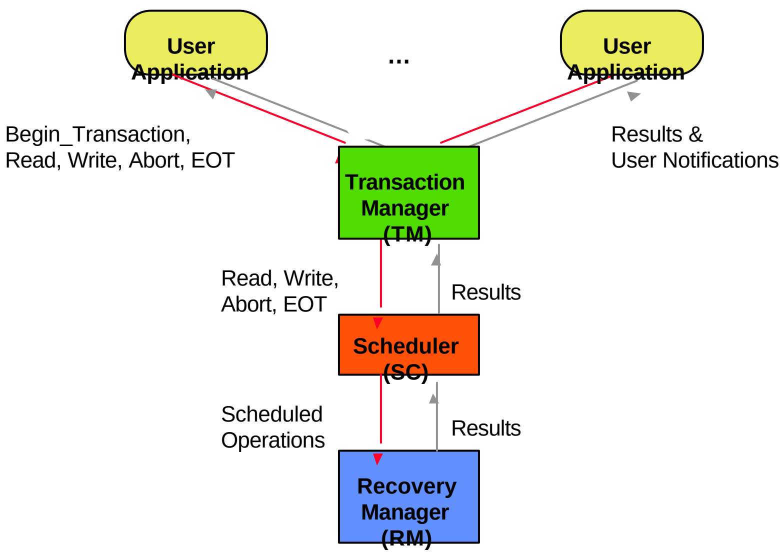
- *Atomic* and *reliable* execution in the presence of failures
- *Correct* execution in the presence of multiple user accesses
- Correct management of *replicas* (if they support it)

# Architecture






# Transaction Execution



# MySQL and Java

Ömer Erdem Demir

January 25, 2006

## 1 Requirements

You will need:

1. `java` and `javac`
2. MySQL installed. Directions for installing MySQL on CSIF machines can be found at http://csifdocs.cs.ucdavis.edu/tiki-index.php?page=CSIF+MySQL+4.x+Install
3. MySQL JDBC driver. You can download it from http://dev.mysql.com/downloads/connector/j/3.1.html
   Extract `mysql-connector-java-`*`3.1.12`*`-bin.jar` (or the latest version you have) from the Connector/J archive to your home directory, you will not need the other files.

## 2 Setting up the tutorial database

In this section we will create a new database, a new user, and a very simple table. MySQL has a two level directory like hierarchy for keeping databases and tables. At the root there is MySQL; under root you can only create "databases." Database is almost like a directory, you can create "tables" under a database. Follow the steps listed below.

1. Start the mysql server (follow the CSIF MySQL tutorial).
2. Check if mysql server is running.

   ```
   $ mysqladmin -u root -p status
   Uptime:  434 Threads:  1 Questions:  86 Slow queries:  0 ...
   ```

3. Start the mysql client. We will use the command line client to create a new database, a new user and a table in the new database.

(a)
```
$ mysql -u root -p
Welcome to the MySQL monitor.  Commands end with ; or \g.
⋮
mysql>
```

(b) Create a new database named `ecs160tutorial`.

```
mysql> CREATE DATABASE ecs160tutorial;
Query OK, 1 row affected (0.06 sec)
```

(c) Create a user with all privileges on this database. The user name will be `tutorialuser` and the password will be `123456`. Although this is NOT good practice, it will suffice.

```
mysql> GRANT ALL ON ecs160tutorial.* TO tutorialuser@'%'
    -> IDENTIFIED BY  '123456';
mysql> GRANT ALL ON ecs160tutorial.* TO tutorialuser@'localhost'
    -> IDENTIFIED BY  '123456';
```

(d) Quit mysql client. We'll reconnect as `tutorialuser` to the `ecs160tutorial` database to setup a table.

```
mysql> quit
Bye
$ mysql -u tutorialuser -p ecs160tutorial
Enter password:
Welcome...
⋮
mysql>
```

(e) Now we will create a simple table with two columns, name and last name.

```
mysql> CREATE TABLE simple_table (name CHAR(128), last_name CHAR(128));
Query OK, 0 rows affected (0.01 sec)
```

`show tables` command will list all the tables created in this database.

```
mysql> show tables;
+--------------------------+
| Tables_in_ecs160tutorial |
+--------------------------+
| simple_table             |
+--------------------------+
1 row in set (0.00 sec)
```

So far we set up a new database for this tutorial, created a user and a very simple table. I think it is a good idea to create a new database and user for your project as we did in this tutorial. In the next section, I'll describe how to connect to the `ecs160tutorial` database from a `Java` program and execute simple queries.

## 3 Connecting to a MySQL database from a Java program using the Connector/J JDBC driver

I assume that you downloaded and installed `Connector/J`. If you haven't done so, read section 1 for the requirements.

You can connect to the MySQL database in two steps. Those steps are detailed below.

1. First load the driver.

```
Class driver_class=null;
try {
   driver_class = Class.forName("com.mysql.jdbc.Driver");
} catch (ClassNotFoundException e) {
   e.printStackTrace();
}
System.out.println("Found driver " + driver_class);
```

   We don't need to register the driver, once it is loaded it will be used for connection requests to mysql databases.

2. Next step is to connect to the MySQL server and the `ecs160tutorial` database. Recall that the user name is `tutorialuser` and the password is `123456`.

```
Connection connection=null;
try {
        connection = DriverManager.getConnection
         ("jdbc:mysql://localhost:3306/ecs160tutorial","tutorialuser","123456");
} catch (SQLException e) {
        e.printStackTrace();
}

try {
        System.out.println
         ("Established connection to "+ connection.getMetaData().getURL());
} catch (SQLException e1) {
        e1.printStackTrace();
}
```

   You must have noticed that `DriverManager.getConnection` takes three arguments. The first argument is the URL of the server; URLs always start with `jdbc:mysql://` and followed by the server address and the database name. Therefore, if you are running the MySQL server on a different machine you should replace `localhost` with the correct machine address, either name or IP address. Moreover, you'll need to replace 3306 with the number of the port your MySQL server is listening on. Next component of the URL is the database name. The second argument is the user name and the last one is the password.

Next, we will switch back to the mysql client to populate simple_table.

1. Connect to the database using the mysql client.

```
$ mysql -u tutorialuser -p ecs160tutorial
Enter password:
Welcome...
⋮
mysql>
```

2. Now we will insert two items into our simple_table.

```
mysql> INSERT INTO simple_table VALUES ("omer","demir");
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO simple_table VALUES ("kivilcim","dogerlioglu-demir");
Query OK, 1 row affected (0.00 sec)
```

3. Run the following query, the output you see should be similar to the output given below.

```
mysql> SELECT * from simple_table;
+----------+-------------------+
| name     | last_name         |
+----------+-------------------+
| omer     | demir             |
| kivilcim | dogerlioglu-demir |
+----------+-------------------+
2 rows in set (0.00 sec)
```

Now, we will execute the same SELECT query from our Java program.

1. We will use the connection to create an empty statement.

```
statement = connection.createStatement();
```

2. Execute the SELECT query.

```
statement.execute("SELECT * FROM  simple_table");
```

3. Get the result set of the query.

```
ResultSet resset = statement.getResultSet();
```

See http://java.sun.com/j2se/1.4.2/docs/api/java/sql/ResultSet.html for the API documentation.

4. We are ready to print the result of the query. The result set returned by the statement initially points before the first row, thus you must call `next` to advance to the first row. See the code snippet below.

```
System.out.println("Row Name Last_Name");
while(resset.next())
{
        System.out.print(resset.getRow());
        System.out.print(" " + resset.getString("name"));
        System.out.println(" " + resset.getString("last_name"));
}
resset.close();
```

A row of the result set is made up of columns. We know the column names and the types of the columns of `simple_table`; they are `name` and `last_name` and both are type string. Therefore, we will use `getString` (remember column type) method with the column names.

The output should be similar to the one below.

```
Row Name Last_Name
1 omer demir
2 kivilcim dogerlioglu-demir
```

# 4 Summary

In this tutorial I explained, using MySQL, how to create a database, a user, and a simple table. I also explained how to connect to a MySQL database from a Java program and execute queries. The Java program I used as the example can be found in the appendix. You can use `javac` to compile the program. Don't forget to change the host address and the port number. To run it, you will need to pass `-classpath` option:

```
java -classpath /home/<user_name>/mysql-connector-java-3.1.12-bin.jar:$CLASSPATH:. Main
```

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class Main {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		Class driver_class=null;
		try {
			driver_class = Class.forName("com.mysql.jdbc.Driver");
		} catch (ClassNotFoundException e) {
			e.printStackTrace();
			return;
		}
		System.out.println("Found driver " + driver_class);

		Connection connection=null;
		try {
			connection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/ecs160tutorial","tutorialuser","12
3456");
		} catch (SQLException e) {
			e.printStackTrace();
			return;
		}

		try {
			System.out.println("Established connection to "+
connection.getMetaData().getURL());
		} catch (SQLException e1) {
			e1.printStackTrace();
		}

		Statement statement=null;
		try {
			statement = connection.createStatement();
			statement.execute("SELECT * FROM  simple_table");
			ResultSet resset = statement.getResultSet();
			System.out.println("Row Name Last_Name");
			while(resset.next())
			{
				System.out.print(resset.getRow());
				System.out.print(" " + resset.getString("name"));
				System.out.println(" " + resset.getString("last_name"));
			}
			resset.close();
		} catch (SQLException e) {
			e.printStackTrace();
		}
		finally{
			if (statement != null)
			{
				try {
					statement.close();
				} catch (SQLException e) {
					e.printStackTrace();
```

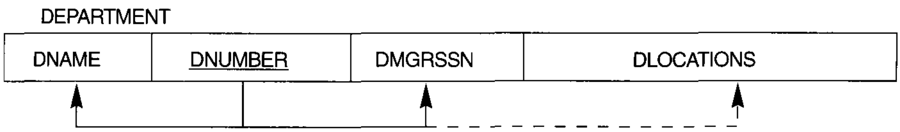
```
                                }
                        }
                        if (connection != null)
                        {
                                try {
                                        connection.close();
                                } catch (SQLException e) {
                                        e.printStackTrace();
                                }
                        }

                }


        }
}
```

## 10.3.4 First Normal Form

**First normal form (1NF)** is now considered to be part of the formal definition of a relation in the basic (flat) relational model;[12] historically, it was defined to disallow multivalued attributes, composite attributes, and their combinations. It states that the domain of an attribute must include only *atomic* (simple, indivisible) *values* and that the value of any attribute in a tuple must be a *single value* from the domain of that attribute. Hence, 1NF disallows having a set of values, a tuple of values, or a combination of both as an attribute value for a *single tuple*. In other words, 1NF disallows "relations within relations" or "relations as attribute values within tuples." The only attribute values permitted by 1NF are single **atomic** (or **indivisible**) **values.**

Consider the DEPARTMENT relation schema shown in Figure 10.1, whose primary key is DNUMBER, and suppose that we extend it by including the DLOCATIONS attribute as shown in Figure 10.8a. We assume that each department can have *a number of* locations. The DEPARTMENT schema and an example relation state are shown in Figure 10.8. As we can see,

(a)




(b) DEPARTMENT

| DNAME | DNUMBER | DMGRSSN | DLOCATIONS |
|---|---|---|---|
| Research | 5 | 333445555 | {Bellaire, Sugarland, Houston} |
| Administration | 4 | 987654321 | {Stafford} |
| Headquarters | 1 | 888665555 | {Houston} |

(c) DEPARTMENT

| DNAME | DNUMBER | DMGRSSN | DLOCATION |
|---|---|---|---|
| Research | 5 | 333445555 | Bellaire |
| Research | 5 | 333445555 | Sugarland |
| Research | 5 | 333445555 | Houston |
| Administration | 4 | 987654321 | Stafford |
| Headquarters | 1 | 888665555 | Houston |

**FIGURE 10.8** Normalization into 1NF. (a) A relation schema that is not in 1NF. (b) Example state of relation DEPARTMENT. (c) 1NF version of same relation with redundancy.

12. This condition is removed in the *nested relational model* and in *object-relational systems* (ORDBMSs), both of which allow *unnormalized relations* (see Chapter 22).

this is not in 1NF because DLOCATIONS is not an atomic attribute, as illustrated by the first tuple in Figure 10.8b. There are two ways we can look at the DLOCATIONS attribute:

- The domain of DLOCATIONS contains atomic values, but some tuples can have a set of these values. In this case, DLOCATIONS *is not* functionally dependent on the primary key DNUMBER.
- The domain of DLOCATIONS contains sets of values and hence is nonatomic. In this case, DNUMBER $\rightarrow$ DLOCATIONS, because each set is considered a single member of the attribute domain.[13]

In either case, the DEPARTMENT relation of Figure 10.8 is not in 1NF; in fact, it does not even qualify as a relation according to our definition of relation in Section 5.1. There are three main techniques to achieve first normal form for such a relation:

1. Remove the attribute DLOCATIONS that violates 1NF and place it in a separate relation DEPT_LOCATIONS along with the primary key DNUMBER of DEPARTMENT. The primary key of this relation is the combination {DNUMBER, DLOCATION}, as shown in Figure 10.2. A distinct tuple in DEPT_LOCATIONS exists for *each location* of a department. This decomposes the non-1NF relation into two 1NF relations.
2. Expand the key so that there will be a separate tuple in the original DEPARTMENT relation for each location of a DEPARTMENT, as shown in Figure 10.8c. In this case, the primary key becomes the combination {DNUMBER, DLOCATION}. This solution has the disadvantage of introducing *redundancy* in the relation.
3. If a *maximum number of values* is known for the attribute—for example, if it is known that *at most three locations* can exist for a department—replace the DLOCATIONS attribute by three atomic attributes: DLOCATION1, DLOCATION2, and DLOCATION3. This solution has the disadvantage of introducing *null values* if most departments have fewer than three locations. It further introduces a spurious semantics about the ordering among the location values that is not originally intended. Querying on this attribute becomes more difficult; for example, consider how you would write the query: "List the departments that have "Bellaire" as one of their locations" in this design.

Of the three solutions above, the first is generally considered best because it does not suffer from redundancy and it is completely general, having no limit placed on a maximum number of values. In fact, if we choose the second solution, it will be decomposed further during subsequent normalization steps into the first solution.

First normal form also disallows multivalued attributes that are themselves composite. These are called **nested relations** because each tuple can have a relation *within it*. Figure 10.9 shows how the EMP_PROJ relation could appear if nesting is allowed. Each tuple represents an employee entity, and a relation PROJS(PNUMBER, HOURS) *within each*

13. In this case we can consider the domain of DLOCATIONS to be the **power set** of the set of single locations; that is, the domain is made up of all possible subsets of the set of single locations.

(a) **EMP_PROJ**

| SSN | ENAME | PROJS | |
|---|---|---|---|
| | | PNUMBER | HOURS |

(b) **EMP_PROJ**

| SSN | ENAME | PNUMBER | HOURS |
|---|---|---|---|
| 123456789 | Smith,John B. | 1 | 32.5 |
| | | 2 | 7.5 |
| 666884444 | Narayan,Ramesh K. | 3 | 40.0 |
| 453453453 | English,Joyce A. | 1 | 20.0 |
| | | 2 | 20.0 |
| 333445555 | Wong,Franklin T. | 2 | 10.0 |
| | | 3 | 10.0 |
| | | 10 | 10.0 |
| | | 20 | 10.0 |
| 999887777 | Zelaya,Alicia J. | 30 | 30.0 |
| | | 10 | 10.0 |
| 987987987 | Jabbar,Ahmad V. | 10 | 35.0 |
| | | 30 | 5.0 |
| 987654321 | Wallace,Jennifer S. | 30 | 20.0 |
| | | 20 | 15.0 |
| 888665555 | Borg,James E. | 20 | null |

(c) **EMP_PROJ1**

| SSN | ENAME |
|---|---|

**EMP_PROJ2**

| SSN | PNUMBER | HOURS |
|---|---|---|

**FIGURE 10.9** Normalizing nested relations into 1NF. (a) Schema of the EMP_PROJ relation with a "nested relation" attribute PROJS. (b) Example extension of the EMP_PROJ relation showing nested relations within each tuple. (c) Decomposition of EMP_PROJ into relations EMP_PROJ1 and EMP_PROJ2 by propagating the primary key.

tuple represents the employee's projects and the hours per week that employee works on each project. The schema of this EMP_PROJ relation can be represented as follows:

EMP_PROJ(SSN, ENAME, {PROJS(PNUMBER, HOURS)})

The set braces { } identify the attribute PROJS as multivalued, and we list the component attributes that form PROJS between parentheses ( ). Interestingly, recent trends for supporting complex objects (see Chapter 20) and XML data (see Chapter 26) using the relational model attempt to allow and formalize nested relations within relational database systems, which were disallowed early on by 1NF.

Notice that SSN is the primary key of the EMP_PROJ relation in Figures 10.9a and b, while PNUMBER is the **partial** key of the nested relation; that is, within each tuple, the nested relation must have unique values of PNUMBER. To normalize this into 1NF, we remove the nested relation attributes into a new relation and *propagate the primary key* into it; the primary key of the new relation will combine the partial key with the primary key of the original relation. Decomposition and primary key propagation yield the schemas EMP_PROJ1 and EMP_PROJ2 shown in Figure 10.9c.

This procedure can be applied recursively to a relation with multiple-level nesting to **unnest** the relation into a set of 1NF relations. This is useful in converting an unnormalized relation schema with many levels of nesting into 1NF relations. The existence of more than one multivalued attribute in one relation must be handled carefully. As an example, consider the following non-1NF relation:

PERSON (SS#, {CAR_LIC#}, {PHONE#})

This relation represents the fact that a person has multiple cars and multiple phones. If a strategy like the second option above is followed, it results in an all-key relation:

PERSON_IN_1NF (SS#, CAR_LIC#, PHONE#)

To avoid introducing any extraneous relationship between CAR_LIC# and PHONE#, all possible combinations of values are represented for every SS#, giving rise to redundancy. This leads to the problems handled by multivalued dependencies and 4NF, which we discuss in Chapter 11. The right way to deal with the two multivalued attributes in PERSON above is to decompose it into two separate relations, using strategy 1 discussed above: P1(SS#, CAR_LIC#) and P2( SS#, PHONE#).

## 10.3.5 Second Normal Form

**Second normal form** (2NF) is based on the concept of *full functional dependency*. A functional dependency $X \rightarrow Y$ is a **full functional dependency** if removal of any attribute $A$ from $X$ means that the dependency does not hold any more; that is, for any attribute $A \in X$, $(X - \{A\})$ does *not* functionally determine $Y$. A functional dependency $X \rightarrow Y$ is a **partial dependency** if some attribute $A \in X$ can be removed from $X$ and the dependency still holds; that is, for some $A \in X$, $(X - \{A\}) \rightarrow Y$. In Figure 10.3b, {SSN, PNUMBER} → HOURS is a full dependency (neither SSN → HOURS nor PNUMBER → HOURS holds). However, the dependency {SSN, PNUMBER} → ENAME is partial because SSN → ENAME holds.

**Definition.** A relation schema $R$ is in **2NF** if every nonprime attribute $A$ in $R$ is *fully functionally dependent* on the primary key of $R$.

The test for 2NF involves testing for functional dependencies whose left-hand side attributes are part of the primary key. If the primary key contains a single attribute, the test need not be applied at all. The EMP_PROJ relation in Figure 10.3b is in 1NF but is not in 2NF. The nonprime attribute ENAME violates 2NF because of FD2, as do the nonprime attributes PNAME and PLOCATION because of FD3. The functional dependencies FD2 and FD3 make ENAME, PNAME, and PLOCATION partially dependent on the primary key {SSN, PNUMBER} of EMP_PROJ, thus violating the 2NF test.

If a relation schema is not in 2NF, it can be "second normalized" or "2NF normalized" into a number of 2NF relations in which nonprime attributes are associated only with the part of the primary key on which they are fully functionally dependent. The functional dependencies FD1, FD2, and FD3 in Figure 10.3b hence lead to the decomposition of EMP_PROJ into the three relation schemas EP1, EP2, and EP3 shown in Figure 10.10a, each of which is in 2NF.

## 10.3.6 Third Normal Form

**Third normal form** (3NF) is based on the concept of *transitive dependency*. A functional dependency $X \rightarrow Y$ in a relation schema $R$ is a **transitive dependency** if there is a set of

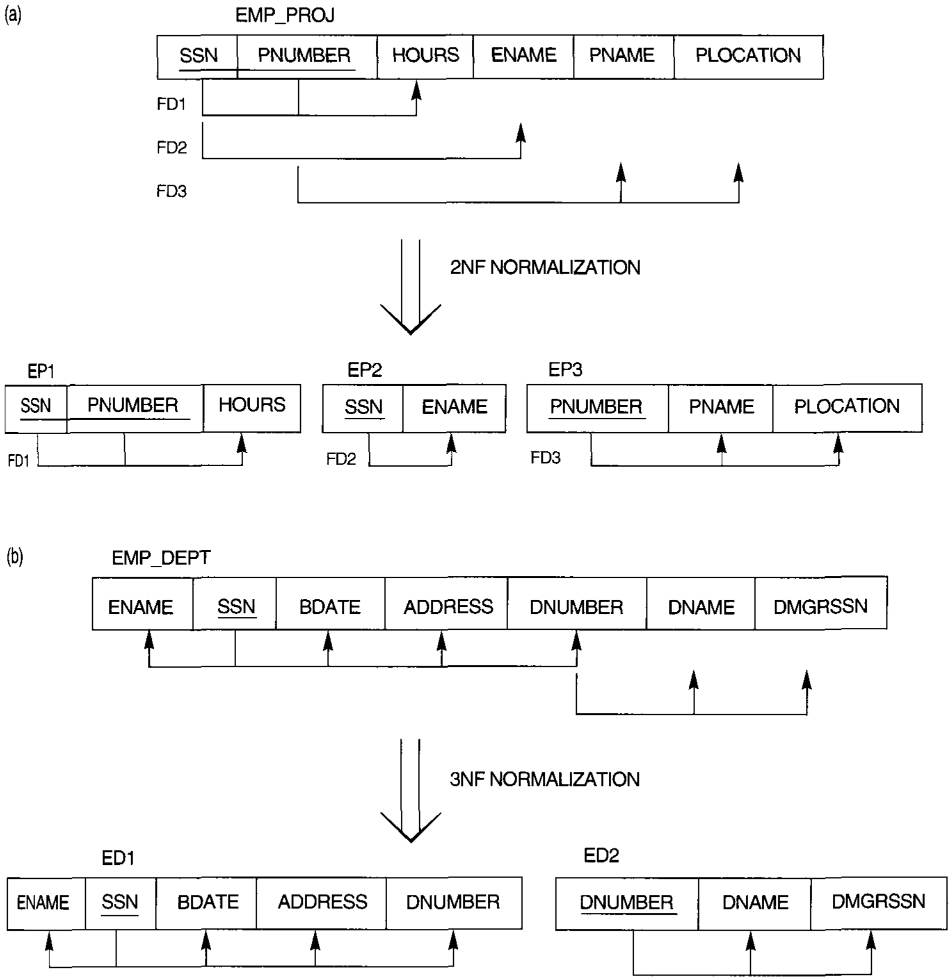



**FIGURE 10.10** Normalizing into 2NF and 3NF. (a) Normalizing EMP_PROJ into 2NF relations. (b) Normalizing EMP_DEPT into 3NF relations.

attributes $Z$ that is neither a candidate key nor a subset of any key of $R$,[14] and both $X \rightarrow Z$ and $Z \rightarrow Y$ hold. The dependency SSN $\rightarrow$ DMGRSSN is transitive through DNUMBER in EMP_DEPT of Figure 10.3a because both the dependencies SSN $\rightarrow$ DNUMBER and DNUMBER $\rightarrow$ DMGRSSN hold *and* DNUMBER is neither a key itself nor a subset of the key of EMP_DEPT. Intuitively, we can see that the dependency of DMGRSSN on DNUMBER is undesirable in EMP_DEPT since DNUMBER is not a key of EMP_DEPT.

**Definition.** According to Codd's original definition, a relation schema $R$ is in **3NF** if it satisfies 2NF *and* no nonprime attribute of $R$ is transitively dependent on the primary key.

The relation schema EMP_DEPT in Figure 10.3a is in 2NF, since no partial dependencies on a key exist. However, EMP_DEPT is not in 3NF because of the transitive dependency of DMGRSSN (and also DNAME) on SSN via DNUMBER. We can normalize EMP_DEPT by decomposing it into the two 3NF relation schemas ED1 and ED2 shown in Figure 10.10b. Intuitively, we see that ED1 and ED2 represent independent entity facts about employees and departments. A NATURAL JOIN operation on ED1 and ED2 will recover the original relation EMP_DEPT without generating spurious tuples.

Intuitively, we can see that any functional dependency in which the left-hand side is part (proper subset) of the primary key, or any functional dependency in which the left-hand side is a nonkey attribute is a "problematic" FD. 2NF and 3NF normalization remove these problem FDs by decomposing the original relation into new relations. In terms of the normalization process, it is not necessary to remove the partial dependencies before the transitive dependencies, but historically, 3NF has been defined with the assumption that a relation is tested for 2NF first before it is tested for 3NF. Table 10.1 informally summarizes the three normal forms based on primary keys, the tests used in each case, and the corresponding "remedy" or normalization performed to achieve the normal form.

## 10.4 General Definitions of Second and Third Normal Forms

In general, we want to design our relation schemas so that they have neither partial nor transitive dependencies, because these types of dependencies cause the update anomalies discussed in Section 10.1.2. The steps for normalization into 3NF relations that we have discussed so far disallow partial and transitive dependencies on the *primary key*. These definitions, however, do not take other candidate keys of a relation, if any, into account. In this section we give the more general definitions of 2NF and 3NF that take *all* candidate keys of a relation into account. Notice that this does not affect the definition of 1NF, since it is independent of keys and functional dependencies. As a general definition of **prime attribute,** an attribute that is part of *any candidate key* will be considered as prime.

14. This is the general definition of transitive dependency. Because we are concerned only with primary keys in this section, we allow transitive dependencies where $X$ is the primary key but $Z$ may be (a subset of) a candidate key.

**TABLE 10.1 SUMMARY OF NORMAL FORMS BASED ON PRIMARY KEYS AND CORRESPONDING NORMALIZATION**

| NORMAL FORM | TEST | REMEDY (NORMALIZATION) |
|---|---|---|
| First (1NF) | Relation should have no nonatomic attributes or nested relations. | Form new relations for each nonatomic attribute or nested relation. |
| Second (2NF) | For relations where primary key contains multiple attributes, no nonkey attribute should be functionally dependent on a part of the primary key. | Decompose and set up a new relation for each partial key with its dependent attribute(s). Make sure to keep a relation with the original primary key and any attributes that are fully functionally dependent on it. |
| Third (3NF) | Relation should not have a nonkey attribute functionally determined by another nonkey attribute (or by a set of nonkey attributes.) That is, there should be no transitive dependency of a nonkey attribute on the primary key. | Decompose and set up a relation that includes the nonkey attribute(s) that functionally determine(s) other nonkey attribute(s). |

Partial and full functional dependencies and transitive dependencies will now be considered *with respect to all candidate keys* of a relation.

## 10.4.1 General Definition of Second Normal Form

**Definition.** A relation schema $R$ is in **second normal form** (2NF) if every nonprime attribute $A$ in $R$ is not partially dependent on *any* key of $R$.[15]

The test for 2NF involves testing for functional dependencies whose left-hand side attributes are *part of* the primary key. If the primary key contains a single attribute, the test need not be applied at all. Consider the relation schema LOTS shown in Figure 10.11a, which describes parcels of land for sale in various counties of a state. Suppose that there are two candidate keys: PROPERTY_ID# and {COUNTY_NAME, LOT#}; that is, lot numbers are unique only within each county, but PROPERTY_ID numbers are unique across counties for the entire state.

Based on the two candidate keys PROPERTY_ID# and {COUNTY_NAME, LOT#}, we know that the functional dependencies FD1 and FD2 of Figure 10.11a hold. We choose PROPERTY_ID# as the primary key, so it is underlined in Figure 10.11a, but no special consideration will

15. This definition can be restated as follows: A relation schema $R$ is in 2NF if every nonprime attribute $A$ in $R$ is fully functionally dependent on *every* key of $R$.

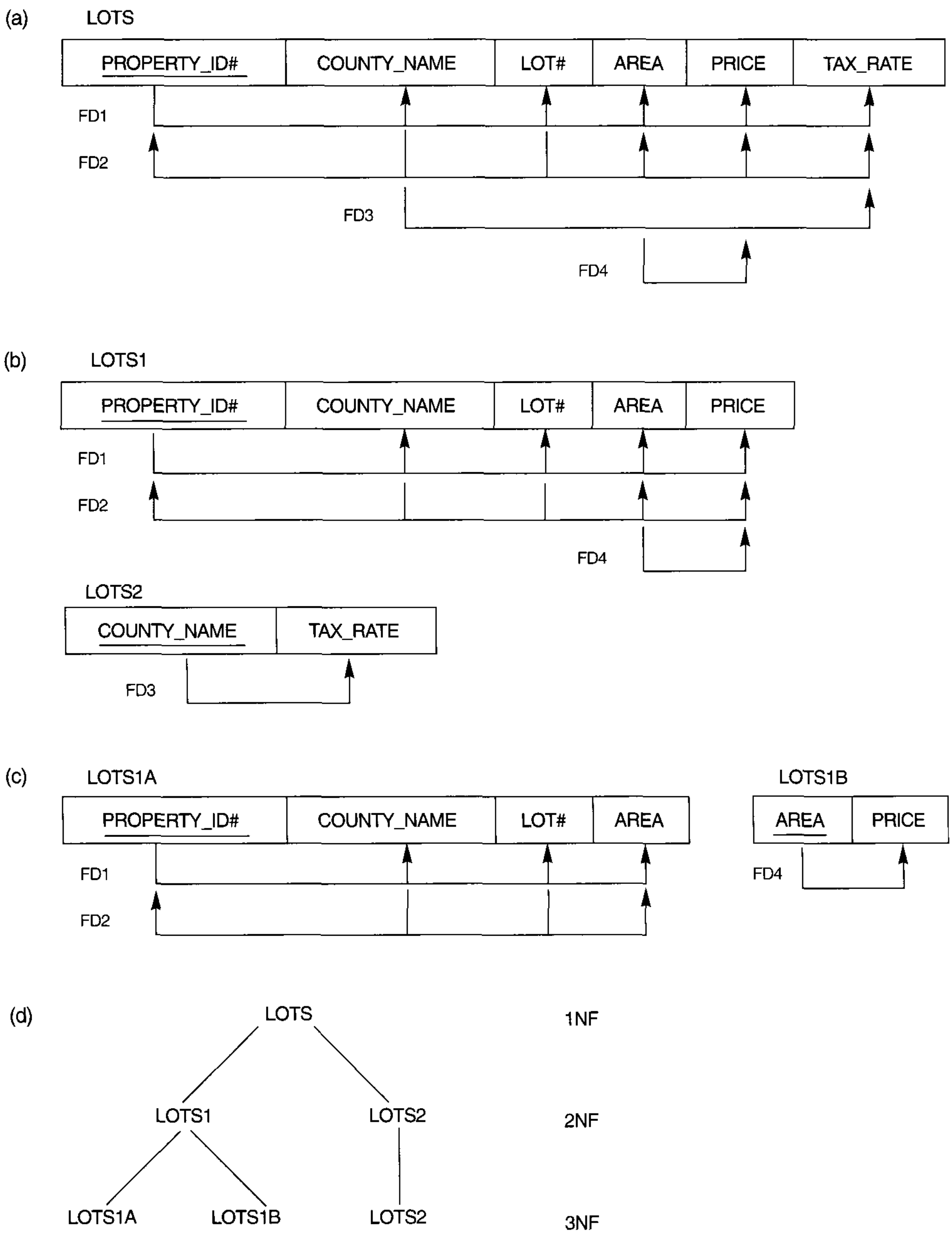



**FIGURE 10.11** Normalization into 2NF and 3NF. (a) The LOTS relation with its functional dependencies FD1 through FD4. (b) Decomposing into the 2NF relations LOTS1 and LOTS2. (c) Decomposing LOTS1 into the 3NF relations LOTS1A and LOTS1B. (d) Summary of the progressive normalization of LOTS.

be given to this key over the other candidate key. Suppose that the following two additional functional dependencies hold in LOTS:

FD3: COUNTY_NAME → TAX_RATE

FD4: AREA → PRICE

In words, the dependency FD3 says that the tax rate is fixed for a given county (does not vary lot by lot within the same county), while FD4 says that the price of a lot is determined by its area regardless of which county it is in. (Assume that this is the price of the lot for tax purposes.)

The LOTS relation schema violates the general definition of 2NF because TAX_RATE is partially dependent on the candidate key {COUNTY_NAME, LOT#}, due to FD3. To normalize LOTS into 2NF, we decompose it into the two relations LOTS1 and LOTS2, shown in Figure 10.11b. We construct LOTS1 by removing the attribute TAX_RATE that violates 2NF from LOTS and placing it with COUNTY_NAME (the left-hand side of FD3 that causes the partial dependency) into another relation LOTS2. Both LOTS1 and LOTS2 are in 2NF. Notice that FD4 does not violate 2NF and is carried over to LOTS1.

### 10.4.2 General Definition of Third Normal Form

**Definition.** A relation schema $R$ is in **third normal form (3NF)** if, whenever a *nontrivial* functional dependency $X \rightarrow A$ holds in $R$, either (a) $X$ is a superkey of $R$, or (b) $A$ is a prime attribute of $R$.

According to this definition, LOTS2 (Figure 10.11b) is in 3NF. However, FD4 in LOTS1 violates 3NF because AREA is not a superkey and PRICE is not a prime attribute in LOTS1. To normalize LOTS1 into 3NF, we decompose it into the relation schemas LOTS1A and LOTS1B shown in Figure 10.11c. We construct LOTS1A by removing the attribute PRICE that violates 3NF from LOTS1 and placing it with AREA (the left-hand side of FD4 that causes the transitive dependency) into another relation LOTS1B. Both LOTS1A and LOTS1B are in 3NF.

Two points are worth noting about this example and the general definition of 3NF:

- LOTS1 violates 3NF because PRICE is transitively dependent on each of the candidate keys of LOTS1 via the nonprime attribute AREA.
- This general definition can be applied *directly* to test whether a relation schema is in 3NF; it does *not* have to go through 2NF first. If we apply the above 3NF definition to LOTS with the dependencies FD1 through FD4, we find that *both* FD3 and FD4 violate 3NF. We could hence decompose LOTS into LOTS1A, LOTS1B, and LOTS2 directly. Hence the transitive and partial dependencies that violate 3NF can be removed *in any order*.

### 10.4.3 Interpreting the General Definition of Third Normal Form

A relation schema $R$ violates the general definition of 3NF if a functional dependency $X \rightarrow A$ holds in $R$ that violates *both* conditions (a) and (b) of 3NF. Violating (b) means that

$A$ is a nonprime attribute. Violating (a) means that $X$ is not a superset of any key of $R$; hence, $X$ could be nonprime or it could be a proper subset of a key of $R$. If $X$ is nonprime, we typically have a transitive dependency that violates 3NF, whereas if $X$ is a proper subset of a key of $R$, we have a partial dependency that violates 3NF (and also 2NF). Hence, we can state a **general alternative definition of 3NF** as follows: A relation schema $R$ is in 3NF if every nonprime attribute of $R$ meets both of the following conditions:

- It is fully functionally dependent on every key of $R$.
- It is nontransitively dependent on every key of $R$.

## 10.5 Boyce-Codd Normal Form

**Boyce-Codd normal form (BCNF)** was proposed as a simpler form of 3NF, but it was found to be stricter than 3NF. That is, every relation in BCNF is also in 3NF; however, a relation in 3NF is *not necessarily* in BCNF. Intuitively, we can see the need for a stronger normal form than 3NF by going back to the LOTS relation schema of Figure 10.11a with its four functional dependencies FD1 through FD4. Suppose that we have thousands of lots in the relation but the lots are from only two counties: Dekalb and Fulton. Suppose also that lot sizes in Dekalb County are only 0.5, 0.6, 0.7, 0.8, 0.9, and 1.0 acres, whereas lot sizes in Fulton County are restricted to 1.1, 1.2, . . . , 1.9, and 2.0 acres. In such a situation we would have the additional functional dependency FD5: AREA $\rightarrow$ COUNTY_NAME. If we add this to the other dependencies, the relation schema LOTS1A still is in 3NF because COUNTY_NAME is a prime attribute.

The area of a lot that determines the county, as specified by FD5, can be represented by 16 tuples in a separate relation $R$(AREA, COUNTY_NAME), since there are only 16 possible AREA values. This representation reduces the redundancy of repeating the same information in the thousands of LOTS1A tuples. BCNF is a *stronger normal form* that would disallow LOTS1A and suggest the need for decomposing it.

**Definition.** A relation schema $R$ is in **BCNF** if whenever a *nontrivial* functional dependency $X \rightarrow A$ holds in $R$, then $X$ is a superkey of $R$.

The formal definition of BCNF differs slightly from the definition of 3NF. The only difference between the definitions of BCNF and 3NF is that condition (b) of 3NF, which allows $A$ to be prime, is absent from BCNF. In our example, FD5 violates BCNF in LOTS1A because AREA is not a superkey of LOTS1A. Note that FD5 satisfies 3NF in LOTS1A because COUNTY_NAME is a prime attribute (condition b), but this condition does not exist in the definition of BCNF. We can decompose LOTS1A into two BCNF relations LOTS1AX and LOTS1AY, shown in Figure 10.12a. This decomposition loses the functional dependency FD2 because its attributes no longer coexist in the same relation after decomposition.

In practice, most relation schemas that are in 3NF are also in BCNF. Only if $X \rightarrow A$ holds in a relation schema $R$ with $X$ not being a superkey *and* $A$ being a prime attribute will $R$ be in 3NF but not in BCNF. The relation schema $R$ shown in Figure 10.12b illustrates the general case of such a relation. Ideally, relational database design should strive to achieve BCNF or 3NF for every relation schema. Achieving the normalization

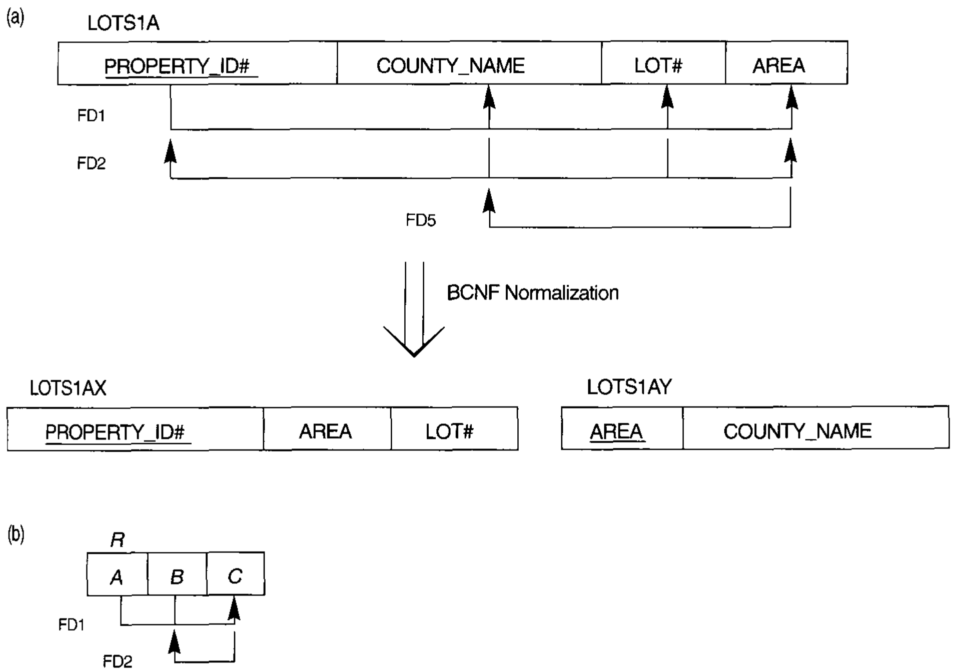



**FIGURE 10.12** Boyce-Codd normal form. (a) BCNF normalization of LOTS1A with the functional dependency FD2 being lost in the decomposition. (b) A schematic relation with FDs; it is in 3NF, but not in BCNF.

status of just 1NF or 2NF is not considered adequate, since they were developed historically as stepping stones to 3NF and BCNF.

As another example, consider Figure 10.13, which shows a relation TEACH with the following dependencies:

FD1: { STUDENT, COURSE} → INSTRUCTOR

FD2:[16] INSTRUCTOR → COURSE

Note that {STUDENT, COURSE} is a candidate key for this relation and that the dependencies shown follow the pattern in Figure 10.12b, with STUDENT as *A*, COURSE as *B*, and INSTRUCTOR as *C*. Hence this relation is in 3NF but not BCNF. Decomposition of this relation schema into two schemas is not straightforward because it may be decomposed into one of the three following possible pairs:

1. {STUDENT, INSTRUCTOR} and {STUDENT, COURSE}.
2. {COURSE, INSTRUCTOR } and {COURSE, STUDENT}.
3. {INSTRUCTOR, COURSE } and {INSTRUCTOR, STUDENT}.

16. This dependency means that "each instructor teaches one course" is a constraint for this application.

TEACH

| STUDENT | COURSE | INSTRUCTOR |
|---|---|---|
| Narayan | Database | Mark |
| Smith | Database | Navathe |
| Smith | Operating Systems | Ammar |
| Smith | Theory | Schulman |
| Wallace | Database | Mark |
| Wallace | Operating Systems | Ahamad |
| Wong | Database | Omiecinski |
| Zelaya | Database | Navathe |

**FIGURE 10.13** A relation TEACH that is in 3NF but not BCNF.

All three decompositions "lose" the functional dependency FD1. The *desirable decomposition* of those just shown is 3, because it will not generate spurious tuples after a join.

A test to determine whether a decomposition is nonadditive (lossless) is discussed in Section 11.1.4 under Property LJ1. In general, a relation not in BCNF should be decomposed so as to meet this property, while possibly forgoing the preservation of all functional dependencies in the decomposed relations, as is the case in this example. Algorithm 11.3 does that and could be used above to give decomposition 3 for TEACH.

# 10.6 SUMMARY

In this chapter we first discussed several pitfalls in relational database design using intuitive arguments. We identified informally some of the measures for indicating whether a relation schema is "good" or "bad," and provided informal guidelines for a good design. We then presented some formal concepts that allow us to do relational design in a top-down fashion by analyzing relations individually. We defined this process of design by analysis and decomposition by introducing the process of normalization.

We discussed the problems of update anomalies that occur when redundancies are present in relations. Informal measures of good relation schemas include simple and clear attribute semantics and few nulls in the extensions (states) of relations. A good decomposition should also avoid the problem of generation of spurious tuples as a result of the join operation.

We defined the concept of functional dependency and discussed some of its properties. Functional dependencies specify semantic constraints among the attributes of a relation schema. We showed how from a given set of functional dependencies, additional dependencies can be inferred using a set of inference rules. We defined the concepts of closure and cover related to functional dependencies. We then defined minimal cover of a set of dependencies, and provided an algorithm to compute a minimal cover. We also showed how to check whether two sets of functional dependencies are equivalent.

We then described the normalization process for achieving good designs by testing relations for undesirable types of "problematic" functional dependencies. We provided a treatment of successive normalization based on a predefined primary key in each relation, then relaxed this requirement and provided more general definitions of second normal form (2NF) and third normal form (3NF) that take all candidate keys of a relation into account. We presented examples to illustrate how by using the general definition of 3NF a given relation may be analyzed and decomposed to eventually yield a set of relations in 3NF.

Finally, we presented Boyce-Codd normal form (BCNF) and discussed how it is a stronger form of 3NF. We also illustrated how the decomposition of a non-BCNF relation must be done by considering the nonadditive decomposition requirement.

Chapter 11 presents synthesis as well as decomposition algorithms for relational database design based on functional dependencies. Related to decomposition, we discuss the concepts of *lossless (nonadditive) join* and *dependency preservation*, which are enforced by some of these algorithms. Other topics in Chapter 11 include multivalued dependencies, join dependencies, and fourth and fifth normal forms, which take these dependencies into account.

## Review Questions

10.1. Discuss attribute semantics as an informal measure of goodness for a relation schema.

10.2. Discuss insertion, deletion, and modification anomalies. Why are they considered bad? Illustrate with examples.

10.3. Why should nulls in a relation be avoided as far as possible? Discuss the problem of spurious tuples and how we may prevent it.

10.4. State the informal guidelines for relation schema design that we discussed. Illustrate how violation of these guidelines may be harmful.

10.5. What is a functional dependency? What are the possible sources of the information that defines the functional dependencies that hold among the attributes of a relation schema?

10.6. Why can we not infer a functional dependency automatically from a particular relation state?

10.7. What role do Armstrong's inference rules—the three inference rules IR1 through IR3—play in the development of the theory of relational design?

10.8. What is meant by the completeness and soundness of Armstrong's inference rules?

10.9. What is meant by the closure of a set of functional dependencies? Illustrate with an example.

10.10. When are two sets of functional dependencies equivalent? How can we determine their equivalence?

10.11. What is a minimal set of functional dependencies? Does every set of dependencies have a minimal equivalent set? Is it always unique?

10.12. What does the term *unnormalized relation* refer to? How did the normal forms develop historically from first normal form up to Boyce-Codd normal form?

10.13. Define first, second, and third normal forms when only primary keys are considered. How do the general definitions of 2NF and 3NF, which consider all keys of a relation, differ from those that consider only primary keys?

10.14. What undesirable dependencies are avoided when a relation is in 2NF?

10.15. What undesirable dependencies are avoided when a relation is in 3NF?

10.16. Define Boyce-Codd normal form. How does it differ from 3NF? Why is it considered a stronger form of 3NF?

## Exercises

10.17. Suppose that we have the following requirements for a university database that is used to keep track of students' transcripts:

a. The university keeps track of each student's name (SNAME), student number (SNUM), social security number (SSN), current address (SCADDR) and phone (SCPHONE), permanent address (SPADDR) and phone (SPPHONE), birth date (BDATE), sex (SEX), class (CLASS) (freshman, sophomore, . . . , graduate), major department (MAJORCODE), minor department (MINORCODE) (if any), and degree program (PROG) (B.A., B.S., . . . , PH.D.). Both SSSN and student number have unique values for each student.

b. Each department is described by a name (DNAME), department code (DCODE), office number (DOFFICE), office phone (DPHONE), and college (DCOLLEGE). Both name and code have unique values for each department.

c. Each course has a course name (CNAME), description (CDESC), course number (CNUM), number of semester hours (CREDIT), level (LEVEL), and offering department (CDEPT). The course number is unique for each course.

d. Each section has an instructor (INAME), semester (SEMESTER), year (YEAR), course (SECCOURSE), and section number (SECNUM). The section number distinguishes different sections of the same course that are taught during the same semester/year; its values are 1, 2, 3, . . . , up to the total number of sections taught during each semester.

e. A grade record refers to a student (SSN), a particular section, and a grade (GRADE).

Design a relational database schema for this database application. First show all the functional dependencies that should hold among the attributes. Then design relation schemas for the database that are each in 3NF or BCNF. Specify the key attributes of each relation. Note any unspecified requirements, and make appropriate assumptions to render the specification complete.

10.18. Prove or disprove the following inference rules for functional dependencies. A proof can be made either by a proof argument or by using inference rules IR1 through IR3. A disproof should be performed by demonstrating a relation instance that satisfies the conditions and functional dependencies in the left-hand side of the inference rule but does not satisfy the dependencies in the right-hand side.

a. $\{W \rightarrow Y, X \rightarrow Z\} \models \{WX \rightarrow Y\}$

b. $\{X \rightarrow Y\}$ and $Y \supseteq Z \models \{X \rightarrow Z\}$

c. $\{X \to Y, X \to W, WY \to Z\} \models \{X \to Z\}$
d. $\{XY \to Z, Y \to W\} \models \{XW \to Z\}$
e. $\{X \to Z, Y \to Z\} \models \{X \to Y\}$
f. $\{X \to Y, XY \to Z\} \models \{X \to Z\}$
g. $\{X \to Y, Z \to W\} \models \{XZ \to YW\}$
h. $\{XY \to Z, Z \to X\} \models \{Z \to Y\}$
i. $\{X \to Y, Y \to Z\} \models \{X \to YZ\}$
j. $\{XY \to Z, Z \to W\} \models \{X \to W\}$

10.19. Consider the following two sets of functional dependencies: $F = \{A \to C, AC \to D, E \to AD, E \to H\}$ and $G = \{A \to CD, E \to AH\}$. Check whether they are equivalent.

10.20. Consider the relation schema EMP_DEPT in Figure 10.3a and the following set G of functional dependencies on EMP_DEPT: G = {SSN → {ENAME, BDATE, ADDRESS, DNUMBER}, DNUMBER → {DNAME, DMGRSSN}}. Calculate the closures $\{\text{SSN}\}^+$ and $\{\text{DNUMBER}\}^+$ with respect to G.

10.21. Is the set of functional dependencies G in Exercise 10.20 minimal? If not, try to find a minimal set of functional dependencies that is equivalent to G. Prove that your set is equivalent to G.

10.22. What update anomalies occur in the EMP_PROJ and EMP_DEPT relations of Figures 10.3 and 10.4?

10.23. In what normal form is the LOTS relation schema in Figure 10.11a with respect to the restrictive interpretations of normal form that take *only the primary key* into account? Would it be in the same normal form if the general definitions of normal form were used?

10.24. Prove that any relation schema with two attributes is in BCNF.

10.25. Why do spurious tuples occur in the result of joining the EMP_PROJ1 and EMP_ LOCS relations of Figure 10.5 (result shown in Figure 10.6)?

10.26. Consider the universal relation $R = \{A, B, C, D, E, F, G, H, I, J\}$ and the set of functional dependencies $F = \{\{A, B\} \to \{C\}, \{A\} \to \{D, E\}, \{B\} \to \{F\}, \{F\} \to \{G, H\}, \{D\} \to \{I, J\}\}$. What is the key for $R$? Decompose $R$ into 2NF and then 3NF relations.

10.27. Repeat Exercise 10.26 for the following different set of functional dependencies $G = \{\{A, B\} \to \{C\}, \{B, D\} \to \{E, F\}, \{A, D\} \to \{G, H\}, \{A\} \to \{I\}, \{H\} \to \{J\}\}$.

10.28. Consider the following relation:

| A | B | C | TUPLE# |
|---|---|---|---|
| 10 | b1 | c1 | #1 |
| 10 | b2 | c2 | #2 |
| 11 | b4 | c1 | #3 |
| 12 | b3 | c4 | #4 |
| 13 | b1 | c1 | #5 |
| 14 | b3 | c4 | #6 |

a. Given the previous extension (state), which of the following dependencies *may hold* in the above relation? If the dependency cannot hold, explain why *by specifying the tuples that cause the violation*.

   i. $A \rightarrow B$, ii. $B \rightarrow C$, iii. $C \rightarrow B$, iv. $B \rightarrow A$, v. $C \rightarrow A$

b. Does the above relation have a *potential* candidate key? If it does, what is it? If it does not, why not?

10.29. Consider a relation $R(A, B, C, D, E)$ with the following dependencies:

$AB \rightarrow C$, $CD \rightarrow E$, $DE \rightarrow B$

Is $AB$ a candidate key of this relation? If not, is $ABD$? Explain your answer.

10.30. Consider the relation $R$, which has attributes that hold schedules of courses and sections at a university; $R$ = {CourseNo, SecNo, OfferingDept, Credit–Hours, CourseLevel, InstructorSSN, Semester, Year, Days_Hours, RoomNo, NoOfStudents}. Suppose that the following functional dependencies hold on $R$:

{CourseNo} → {OfferingDept, CreditHours, CourseLevel}

{CourseNo, SecNo, Semester, Year} → {Days_Hours, RoomNo, NoOfStudents, InstructorSSN}

{RoomNo, Days_Hours, Semester, Year} → {Instructorssn, CourseNo, SecNo}

Try to determine which sets of attributes form keys of $R$. How would you normalize this relation?

10.31. Consider the following relations for an order-processing application database at ABC, Inc.

ORDER (<u>O#</u>, Odate, Cust#, Total_amount)

ORDER-ITEM(<u>O#</u>, <u>I#</u>, Qty_ordered, Total_price, Discount%)

Assume that each item has a different discount. The TOTAL_PRICE refers to one item, ODATE is the date on which the order was placed, and the TOTAL_AMOUNT is the amount of the order. If we apply a natural join on the relations ORDER-ITEM and ORDER in this database, what does the resulting relation schema look like? What will be its key? Show the FDs in this resulting relation. Is it in 2NF? Is it in 3NF? Why or why not? (State assumptions, if you make any.)

10.32. Consider the following relation:

CAR_SALE(Car#, Date_sold, Salesman#, Commission%, Discount_amt)

Assume that a car may be sold by multiple salesmen, and hence {CAR#, SALESMAN#} is the primary key. Additional dependencies are

Date_sold → Discount_amt

and

Salesman# → Commission%

Based on the given primary key, is this relation in 1NF, 2NF, or 3NF? Why or why not? How would you successively normalize it completely?

10.33. Consider the following relation for published books:

BOOK (Book_title, Authorname, Book_type, Listprice, Author_affil, Publisher)

Author_affil refers to the affiliation of author. Suppose the following dependencies exist:

Book_title → Publisher, Book_type

Book_type → Listprice

Authorname → Author-affil

a. What normal form is the relation in? Explain your answer.
b. Apply normalization until you cannot decompose the relations further. State the reasons behind each decomposition.

## Selected Bibliography

Functional dependencies were originally introduced by Codd (1970). The original definitions of first, second, and third normal form were also defined in Codd (1972a), where a discussion on update anomalies can be found. Boyce-Codd normal form was defined in Codd (1974). The alternative definition of third normal form is given in Ullman (1988), as is the definition of BCNF that we give here. Ullman (1988), Maier (1983), and Atzeni and De Antonellis (1993) contain many of the theorems and proofs concerning functional dependencies.

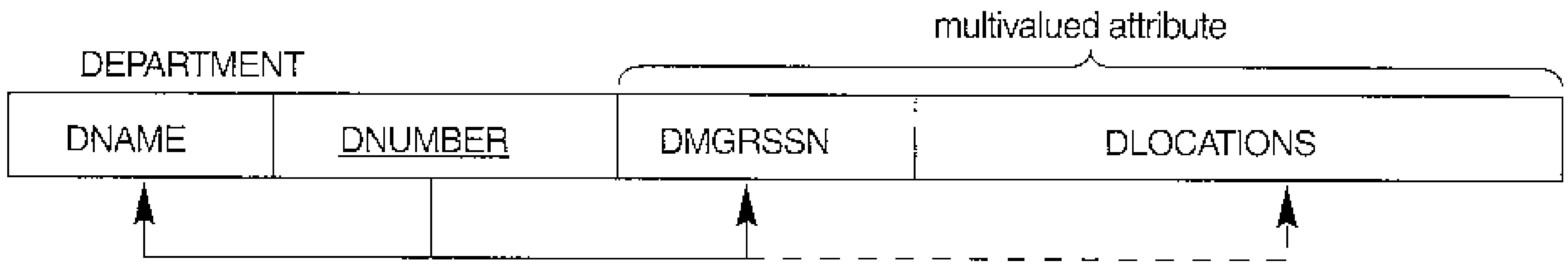
Armstrong (1974) shows the soundness and completeness of the inference rules IR1 through IR3. Additional references to relational design theory are given in Chapter 11.

### 10.3.4 First Normal Form

**First normal form** (1NF) is now considered to be part of the formal definition of a relation in the basic (flat) relational model;[12] historically, it was defined to disallow multivalued attributes, composite attributes, and their combinations. It states that the domain of an attribute must include only *atomic* (simple, indivisible) *values* and that the value of any attribute in a tuple must be a *single value* from the domain of that attribute. Hence, 1NF disallows having a set of values, a tuple of values, or a combination of both as an attribute value for a *single tuple*. In other words, 1NF disallows "relations within relations" or "relations as attribute values within tuples." The only attribute values permitted by 1NF are single **atomic** (or **indivisible**) **values.**

Consider the DEPARTMENT relation schema shown in Figure 10.1, whose primary key is DNUMBER, and suppose that we extend it by including the DLOCATIONS attribute as shown in Figure 10.8a. We assume that each department can have *a number of* locations. The DEPARTMENT schema and an example relation state are shown in Figure 10.8. As we can see,

(a)




(b) DEPARTMENT

| DNAME | DNUMBER | DMGRSSN | DLOCATIONS |
|---|---|---|---|
| Research | 5 | 333445555 | {Bellaire, Sugarland, Houston} |
| Administration | 4 | 987654321 | {Stafford} |
| Headquarters | 1 | 888665555 | {Houston} |

(c) DEPARTMENT_1NF

| DNAME | DNUMBER | DMGRSSN | DLOCATION |
|---|---|---|---|
| Research | 5 | 333445555 | Bellaire |
| Research | 5 | 333445555 | Sugarland |
| Research | 5 | 333445555 | Houston |
| Administration | 4 | 987654321 | Stafford |
| Headquarters | 1 | 888665555 | Houston |

**FIGURE 10.8** Normalization into 1NF. (a) A relation schema that is not in 1NF. (b) Example state of relation DEPARTMENT. (c) 1NF version of same relation with redundancy.

12. This condition is removed in the *nested relational model* and in *object-relational systems* (ORDBMSs), both of which allow *unnormalized relations* (see Chapter 22).

this is not in 1NF because DLOCATIONS is not an atomic attribute, as illustrated by the first tuple in Figure 10.8b. There are two ways we can look at the DLOCATIONS attribute:

- The domain of DLOCATIONS contains atomic values, but some tuples can have a set of these values. In this case, DLOCATIONS *is not* functionally dependent on the primary key DNUMBER.
- The domain of DLOCATIONS contains sets of values and hence is nonatomic. In this case, DNUMBER → DLOCATIONS, because each set is considered a single member of the attribute domain.[13]

In either case, the DEPARTMENT relation of Figure 10.8 is not in 1NF; in fact, it does not even qualify as a relation according to our definition of relation in Section 5.1. There are three main techniques to achieve first normal form for such a relation:

1. Remove the attribute DLOCATIONS that violates 1NF and place it in a separate relation DEPT_LOCATIONS along with the primary key DNUMBER of DEPARTMENT. The primary key of this relation is the combination {DNUMBER, DLOCATION}, as shown in Figure 10.2. A distinct tuple in DEPT_LOCATIONS exists for *each location* of a department. This decomposes the non-1NF relation into two 1NF relations.
2. Expand the key so that there will be a separate tuple in the original DEPARTMENT relation for each location of a DEPARTMENT, as shown in Figure 10.8c. In this case, the primary key becomes the combination {DNUMBER, DLOCATION}. This solution has the disadvantage of introducing *redundancy* in the relation.
3. If a *maximum number of values* is known for the attribute—for example, if it is known that *at most three locations* can exist for a department—replace the DLOCATIONS attribute by three atomic attributes: DLOCATION1, DLOCATION2, and DLOCATION3. This solution has the disadvantage of introducing *null values* if most departments have fewer than three locations. It further introduces a spurious semantics about the ordering among the location values that is not originally intended. Querying on this attribute becomes more difficult; for example, consider how you would write the query: "List the departments that have "Bellaire" as one of their locations" in this design.

Of the three solutions above, the first is generally considered best because it does not suffer from redundancy and it is completely general, having no limit placed on a maximum number of values. In fact, if we choose the second solution, it will be decomposed further during subsequent normalization steps into the first solution.

First normal form also disallows multivalued attributes that are themselves composite. These are called **nested relations** because each tuple can have a relation *within it*. Figure 10.9 shows how the EMP_PROJ relation could appear if nesting is allowed. Each tuple represents an employee entity, and a relation PROJS(PNUMBER, HOURS) *within each*

13. In this case we can consider the domain of DLOCATIONS to be the **power set** of the set of single locations; that is, the domain is made up of all possible subsets of the set of single locations.

(a) **EMP_PROJ**

| SSN | ENAME | PROJS | |
|---|---|---|---|
| | | PNUMBER | HOURS |

(b) **EMP_PROJ**

| SSN | ENAME | PNUMBER | HOURS |
|---|---|---|---|
| 123456789 | Smith,John B. | 1 | 32.5 |
| | | 2 | 7.5 |
| 666884444 | Narayan,Ramesh K. | 3 | 40.0 |
| 453453453 | English,Joyce A. | 1 | 20.0 |
| | | 2 | 20.0 |
| 333445555 | Wong,Franklin T. | 2 | 10.0 |
| | | 3 | 10.0 |
| | | 10 | 10.0 |
| | | 20 | 10.0 |
| 999887777 | Zelaya,Alicia J. | 30 | 30.0 |
| | | 10 | 10.0 |
| 987987987 | Jabbar,Ahmad V. | 10 | 35.0 |
| | | 30 | 5.0 |
| 987654321 | Wallace,Jennifer S. | 30 | 20.0 |
| | | 20 | 15.0 |
| 888665555 | Borg,James E. | 20 | null |

(c) **EMP_PROJ1**

| SSN | ENAME |
|---|---|

**EMP_PROJ2**

| SSN | PNUMBER | HOURS |
|---|---|---|

**FIGURE 10.9** Normalizing nested relations into 1NF. (a) Schema of the EMP_PROJ relation with a "nested relation" attribute PROJS. (b) Example extension of the EMP_PROJ relation showing nested relations within each tuple. (c) Decomposition of EMP_PROJ into relations EMP_PROJ1 and EMP_PROJ2 by propagating the primary key.

*tuple* represents the employee's projects and the hours per week that employee works on each project. The schema of this EMP_PROJ relation can be represented as follows:

EMP_PROJ(SSN, ENAME, {PROJS(PNUMBER, HOURS)})

The set braces { } identify the attribute PROJS as multivalued, and we list the component attributes that form PROJS between parentheses ( ). Interestingly, recent trends for supporting complex objects (see Chapter 20) and XML data (see Chapter 26) using the relational model attempt to allow and formalize nested relations within relational database systems, which were disallowed early on by 1NF.

Notice that SSN is the primary key of the EMP_PROJ relation in Figures 10.9a and b, while PNUMBER is the **partial** key of the nested relation; that is, within each tuple, the nested relation must have unique values of PNUMBER. To normalize this into 1NF, we remove the nested relation attributes into a new relation and *propagate the primary key* into it; the primary key of the new relation will combine the partial key with the primary key of the original relation. Decomposition and primary key propagation yield the schemas EMP_PROJ1 and EMP_PROJ2 shown in Figure 10.9c.

This procedure can be applied recursively to a relation with multiple-level nesting to **unnest** the relation into a set of 1NF relations. This is useful in converting an unnormalized relation schema with many levels of nesting into 1NF relations. The existence of more than one multivalued attribute in one relation must be handled carefully. As an example, consider the following non-1NF relation:

PERSON (SS#, {CAR_LIC#}, {PHONE#})

This relation represents the fact that a person has multiple cars and multiple phones. If a strategy like the second option above is followed, it results in an all-key relation:

PERSON_IN_1NF (SS#, CAR_LIC#, PHONE#)

To avoid introducing any extraneous relationship between CAR_LIC# and PHONE#, all possible combinations of values are represented for every SS#, giving rise to redundancy. This leads to the problems handled by multivalued dependencies and 4NF, which we discuss in Chapter 11. The right way to deal with the two multivalued attributes in PERSON above is to decompose it into two separate relations, using strategy 1 discussed above: P1(SS#, CAR_LIC#) and P2( SS#, PHONE#).

## 10.3.5 Second Normal Form

**Second normal form** (2NF) is based on the concept of *full functional dependency*. A functional dependency $X \rightarrow Y$ is a **full functional dependency** if removal of any attribute $A$ from $X$ means that the dependency does not hold any more; that is, for any attribute $A \in X$, $(X - \{A\})$ does *not* functionally determine $Y$. A functional dependency $X \rightarrow Y$ is a **partial dependency** if some attribute $A \in X$ can be removed from $X$ and the dependency still holds; that is, for some $A \in X$, $(X - \{A\}) \rightarrow Y$. In Figure 10.3b, {SSN, PNUMBER} → HOURS is a full dependency (neither SSN → HOURS nor PNUMBER → HOURS holds). However, the dependency {SSN, PNUMBER} → ENAME is partial because SSN → ENAME holds.

**Definition.** A relation schema $R$ is in **2NF** if every nonprime attribute $A$ in $R$ is *fully functionally dependent* on the primary key of $R$.

The test for 2NF involves testing for functional dependencies whose left-hand side attributes are part of the primary key. If the primary key contains a single attribute, the test need not be applied at all. The EMP_PROJ relation in Figure 10.3b is in 1NF but is not in 2NF. The nonprime attribute ENAME violates 2NF because of FD2, as do the nonprime attributes PNAME and PLOCATION because of FD3. The functional dependencies FD2 and FD3 make ENAME, PNAME, and PLOCATION partially dependent on the primary key {SSN, PNUMBER} of EMP_PROJ, thus violating the 2NF test.

If a relation schema is not in 2NF, it can be "second normalized" or "2NF normalized" into a number of 2NF relations in which nonprime attributes are associated only with the part of the primary key on which they are fully functionally dependent. The functional dependencies FD1, FD2, and FD3 in Figure 10.3b hence lead to the decomposition of EMP_PROJ into the three relation schemas EP1, EP2, and EP3 shown in Figure 10.10a, each of which is in 2NF.

## 10.3.6 Third Normal Form

**Third normal form** (3NF) is based on the concept of *transitive dependency*. A functional dependency $X \rightarrow Y$ in a relation schema $R$ is a **transitive dependency** if there is a set of

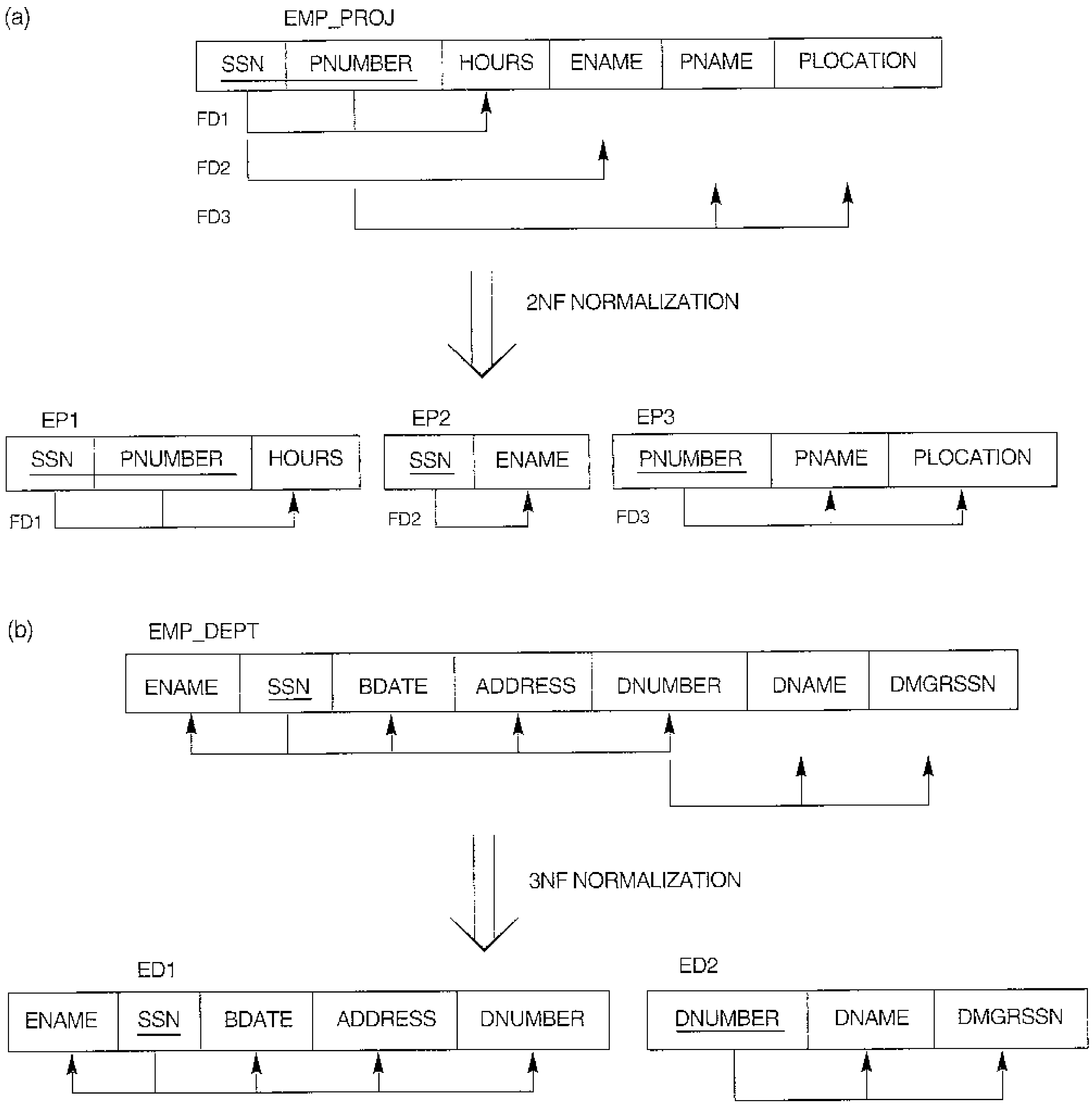



**FIGURE 10.10** Normalizing into 2NF and 3NF. (a) Normalizing EMP_PROJ into 2NF relations. (b) Normalizing EMP_DEPT into 3NF relations.

attributes $Z$ that is neither a candidate key nor a subset of any key of $R$,[14] and both $X \rightarrow Z$ and $Z \rightarrow Y$ hold. The dependency SSN $\rightarrow$ DMGRSSN is transitive through DNUMBER in EMP_DEPT of Figure 10.3a because both the dependencies SSN $\rightarrow$ DNUMBER and DNUMBER $\rightarrow$ DMGRSSN hold *and* DNUMBER is neither a key itself nor a subset of the key of EMP_DEPT. Intuitively, we can see that the dependency of DMGRSSN on DNUMBER is undesirable in EMP_DEPT since DNUMBER is not a key of EMP_DEPT.

**Definition.** According to Codd's original definition, a relation schema $R$ is in 3NF if it satisfies 2NF *and* no nonprime attribute of $R$ is transitively dependent on the primary key.

The relation schema EMP_DEPT in Figure 10.3a is in 2NF, since no partial dependencies on a key exist. However, EMP_DEPT is not in 3NF because of the transitive dependency of DMGRSSN (and also DNAME) on SSN via DNUMBER. We can normalize EMP_DEPT by decomposing it into the two 3NF relation schemas ED1 and ED2 shown in Figure 10.10b. Intuitively, we see that ED1 and ED2 represent independent entity facts about employees and departments. A NATURAL JOIN operation on ED1 and ED2 will recover the original relation EMP_DEPT without generating spurious tuples.

Intuitively, we can see that any functional dependency in which the left-hand side is part (proper subset) of the primary key, or any functional dependency in which the left-hand side is a nonkey attribute is a "problematic" FD. 2NF and 3NF normalization remove these problem FDs by decomposing the original relation into new relations. In terms of the normalization process, it is not necessary to remove the partial dependencies before the transitive dependencies, but historically, 3NF has been defined with the assumption that a relation is tested for 2NF first before it is tested for 3NF. Table 10.1 informally summarizes the three normal forms based on primary keys, the tests used in each case, and the corresponding "remedy" or normalization performed to achieve the normal form.

# 10.4 General Definitions of Second and Third Normal Forms

In general, we want to design our relation schemas so that they have neither partial nor transitive dependencies, because these types of dependencies cause the update anomalies discussed in Section 10.1.2. The steps for normalization into 3NF relations that we have discussed so far disallow partial and transitive dependencies on the *primary key*. These definitions, however, do not take other candidate keys of a relation, if any, into account. In this section we give the more general definitions of 2NF and 3NF that take *all* candidate keys of a relation into account. Notice that this does not affect the definition of 1NF, since it is independent of keys and functional dependencies. As a general definition of **prime attribute,** an attribute that is part of *any candidate key* will be considered as prime.

14. This is the general definition of transitive dependency. Because we are concerned only with primary keys in this section, we allow transitive dependencies where $X$ is the primary key but $Z$ may be (a subset of) a candidate key.

**TABLE 10.1 SUMMARY OF NORMAL FORMS BASED ON PRIMARY KEYS AND CORRESPONDING NORMALIZATION**

| NORMAL FORM | TEST | REMEDY (NORMALIZATION) |
|---|---|---|
| First (1NF) | Relation should have no nonatomic attributes or nested relations. | Form new relations for each nonatomic attribute or nested relation. |
| Second (2NF) | For relations where primary key contains multiple attributes, no nonkey attribute should be functionally dependent on a part of the primary key. | Decompose and set up a new relation for each partial key with its dependent attribute(s). Make sure to keep a relation with the original primary key and any attributes that are fully functionally dependent on it. |
| Third (3NF) | Relation should not have a nonkey attribute functionally determined by another nonkey attribute (or by a set of nonkey attributes.) That is, there should be no transitive dependency of a nonkey attribute on the primary key. | Decompose and set up a relation that includes the nonkey attribute(s) that functionally determine(s) other nonkey attribute(s). |

Partial and full functional dependencies and transitive dependencies will now be considered *with respect to all candidate keys* of a relation.

## 10.4.1 General Definition of Second Normal Form

**Definition.** A relation schema $R$ is in **second normal form** (2NF) if every nonprime attribute $A$ in $R$ is not partially dependent on *any* key of $R$.[15]

The test for 2NF involves testing for functional dependencies whose left-hand side attributes are *part of* the primary key. If the primary key contains a single attribute, the test need not be applied at all. Consider the relation schema LOTS shown in Figure 10.11a, which describes parcels of land for sale in various counties of a state. Suppose that there are two candidate keys: PROPERTY_ID# and {COUNTY_NAME, LOT#}; that is, lot numbers are unique only within each county, but PROPERTY_ID numbers are unique across counties for the entire state.

Based on the two candidate keys PROPERTY_ID# and {COUNTY_NAME, LOT#}, we know that the functional dependencies FD1 and FD2 of Figure 10.11a hold. We choose PROPERTY_ID# as the primary key, so it is underlined in Figure 10.11a, but no special consideration will

15. This definition can be restated as follows: A relation schema $R$ is in 2NF if every nonprime attribute $A$ in $R$ is fully functionally dependent on *every* key of $R$.

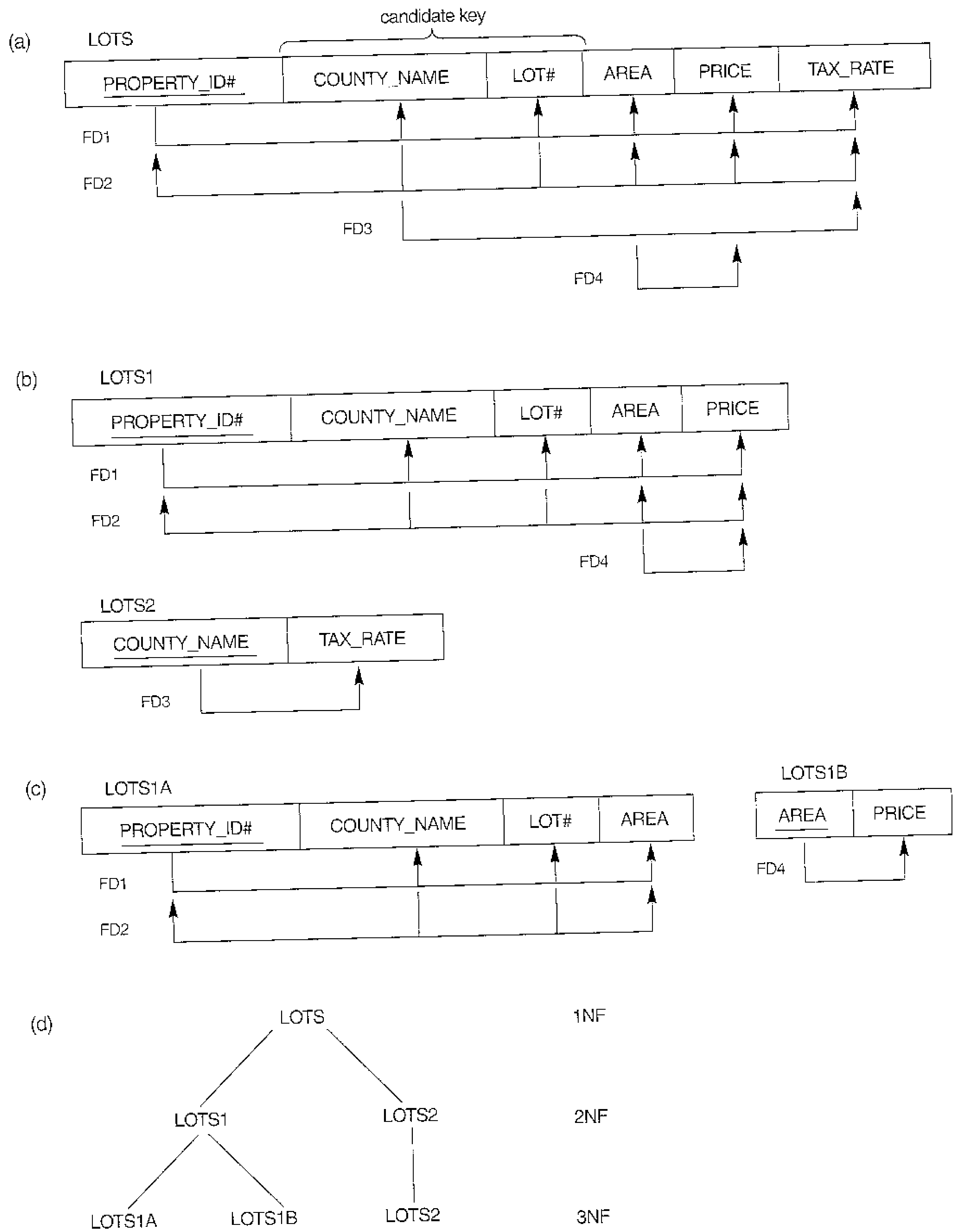



**FIGURE 10.11** Normalization into 2NF and 3NF. (a) The LOTS relation with its functional dependencies FD1 through FD4. (b) Decomposing into the 2NF relations LOTS1 and LOTS2. (c) Decomposing LOTS1 into the 3NF relations LOTS1A and LOTS1B. (d) Summary of the progressive normalization of LOTS.

be given to this key over the other candidate key. Suppose that the following two additional functional dependencies hold in LOTS:

FD3: COUNTY_NAME → TAX_RATE

FD4: AREA → PRICE

In words, the dependency FD3 says that the tax rate is fixed for a given county (does not vary lot by lot within the same county), while FD4 says that the price of a lot is determined by its area regardless of which county it is in. (Assume that this is the price of the lot for tax purposes.)

The LOTS relation schema violates the general definition of 2NF because TAX_RATE is partially dependent on the candidate key {COUNTY_NAME, LOT#}, due to FD3. To normalize LOTS into 2NF, we decompose it into the two relations LOTS1 and LOTS2, shown in Figure 10.11b. We construct LOTS1 by removing the attribute TAX_RATE that violates 2NF from LOTS and placing it with COUNTY_NAME (the left-hand side of FD3 that causes the partial dependency) into another relation LOTS2. Both LOTS1 and LOTS2 are in 2NF. Notice that FD4 does not violate 2NF and is carried over to LOTS1.

### 10.4.2 General Definition of Third Normal Form

**Definition.** A relation schema $R$ is in **third normal form** (3NF) if, whenever a *nontrivial* functional dependency $X \rightarrow A$ holds in $R$, either (a) $X$ is a superkey of $R$, or (b) $A$ is a prime attribute of $R$.

According to this definition, LOTS2 (Figure 10.11b) is in 3NF. However, FD4 in LOTS1 violates 3NF because AREA is not a superkey and PRICE is not a prime attribute in LOTS1. To normalize LOTS1 into 3NF, we decompose it into the relation schemas LOTS1A and LOTS1B shown in Figure 10.11c. We construct LOTS1A by removing the attribute PRICE that violates 3NF from LOTS1 and placing it with AREA (the left-hand side of FD4 that causes the transitive dependency) into another relation LOTS1B. Both LOTS1A and LOTS1B are in 3NF.

Two points are worth noting about this example and the general definition of 3NF:

- LOTS1 violates 3NF because PRICE is transitively dependent on each of the candidate keys of LOTS1 via the nonprime attribute AREA.
- This general definition can be applied *directly* to test whether a relation schema is in 3NF; it does *not* have to go through 2NF first. If we apply the above 3NF definition to LOTS with the dependencies FD1 through FD4, we find that *both* FD3 and FD4 violate 3NF. We could hence decompose LOTS into LOTS1A, LOTS1B, and LOTS2 directly. Hence the transitive and partial dependencies that violate 3NF can be removed *in any order*.

### 10.4.3 Interpreting the General Definition of Third Normal Form

A relation schema $R$ violates the general definition of 3NF if a functional dependency $X \rightarrow A$ holds in $R$ that violates *both* conditions (a) and (b) of 3NF. Violating (b) means that

$A$ is a nonprime attribute. Violating (a) means that $X$ is not a superset of any key of $R$; hence, $X$ could be nonprime or it could be a proper subset of a key of $R$. If $X$ is nonprime, we typically have a transitive dependency that violates 3NF, whereas if $X$ is a proper subset of a key of $R$, we have a partial dependency that violates 3NF (and also 2NF). Hence, we can state a **general alternative definition of 3NF** as follows: A relation schema $R$ is in 3NF if every nonprime attribute of $R$ meets both of the following conditions:

- It is fully functionally dependent on every key of $R$.
- It is nontransitively dependent on every key of $R$.

## 10.5 BOYCE-CODD NORMAL FORM

**Boyce-Codd normal form** (BCNF) was proposed as a simpler form of 3NF, but it was found to be stricter than 3NF. That is, every relation in BCNF is also in 3NF; however, a relation in 3NF is *not necessarily* in BCNF. Intuitively, we can see the need for a stronger normal form than 3NF by going back to the LOTS relation schema of Figure 10.11a with its four functional dependencies FD1 through FD4. Suppose that we have thousands of lots in the relation but the lots are from only two counties: Dekalb and Fulton. Suppose also that lot sizes in Dekalb County are only 0.5, 0.6, 0.7, 0.8, 0.9, and 1.0 acres, whereas lot sizes in Fulton County are restricted to 1.1, 1.2, . . . , 1.9, and 2.0 acres. In such a situation we would have the additional functional dependency FD5: AREA $\rightarrow$ COUNTY_NAME. If we add this to the other dependencies, the relation schema LOTS1A still is in 3NF because COUNTY_NAME is a prime attribute.

The area of a lot that determines the county, as specified by FD5, can be represented by 16 tuples in a separate relation $R$(AREA, COUNTY_NAME), since there are only 16 possible AREA values. This representation reduces the redundancy of repeating the same information in the thousands of LOTS1A tuples. BCNF is a *stronger normal form* that would disallow LOTS1A and suggest the need for decomposing it.

**Definition.** A relation schema $R$ is in **BCNF** if whenever a *nontrivial* functional dependency $X \rightarrow A$ holds in $R$, then $X$ is a superkey of $R$.

The formal definition of BCNF differs slightly from the definition of 3NF. The only difference between the definitions of BCNF and 3NF is that condition (b) of 3NF, which allows $A$ to be prime, is absent from BCNF. In our example, FD5 violates BCNF in LOTS1A because AREA is not a superkey of LOTS1A. Note that FD5 satisfies 3NF in LOTS1A because COUNTY_NAME is a prime attribute (condition b), but this condition does not exist in the definition of BCNF. We can decompose LOTS1A into two BCNF relations LOTS1AX and LOTS1AY, shown in Figure 10.12a. This decomposition loses the functional dependency FD2 because its attributes no longer coexist in the same relation after decomposition.

In practice, most relation schemas that are in 3NF are also in BCNF. Only if $X \rightarrow A$ holds in a relation schema $R$ with $X$ not being a superkey *and* $A$ being a prime attribute will $R$ be in 3NF but not in BCNF. The relation schema $R$ shown in Figure 10.12b illustrates the general case of such a relation. Ideally, relational database design should strive to achieve BCNF or 3NF for every relation schema. Achieving the normalization

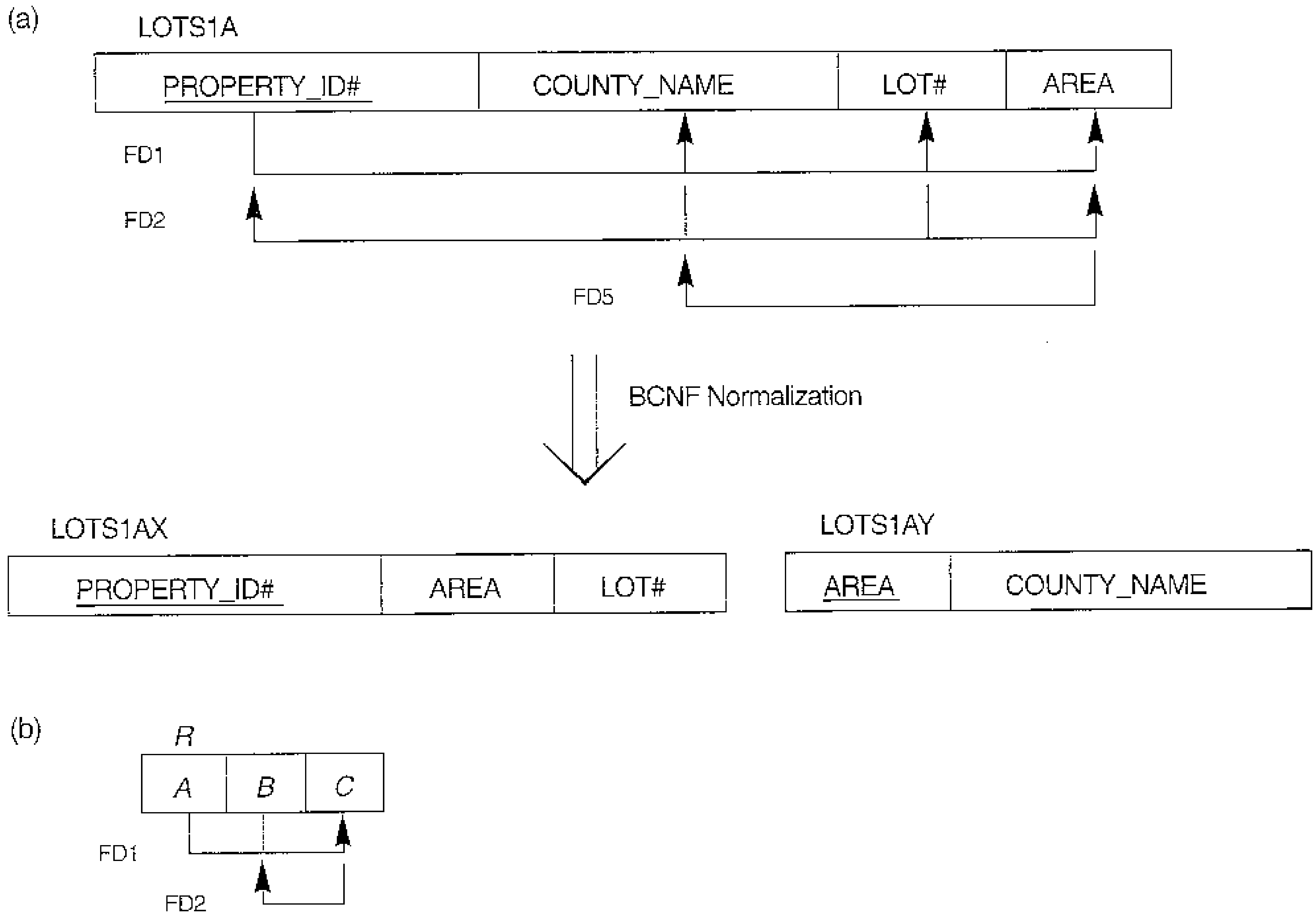



**FIGURE 10.12** Boyce-Codd normal form. (a) BCNF normalization of LOTS1A with the functional dependency FD2 being lost in the decomposition. (b) A schematic relation with FDs; it is in 3NF, but not in BCNF.

status of just 1NF or 2NF is not considered adequate, since they were developed historically as stepping stones to 3NF and BCNF.

As another example, consider Figure 10.13, which shows a relation TEACH with the following dependencies:

FD1: { STUDENT, COURSE} → INSTRUCTOR

FD2:[16] INSTRUCTOR → COURSE

Note that {STUDENT, COURSE} is a candidate key for this relation and that the dependencies shown follow the pattern in Figure 10.12b, with STUDENT as A, COURSE as B, and INSTRUCTOR as C. Hence this relation is in 3NF but not BCNF. Decomposition of this relation schema into two schemas is not straightforward because it may be decomposed into one of the three following possible pairs:

1. {STUDENT, INSTRUCTOR} and {STUDENT, COURSE}.
2. {COURSE, INSTRUCTOR } and {COURSE, STUDENT}.
3. {INSTRUCTOR, COURSE } and {INSTRUCTOR, STUDENT}.

16. This dependency means that "each instructor teaches one course" is a constraint for this application.

TEACH

| STUDENT | COURSE | INSTRUCTOR |
|---|---|---|
| Narayan | Database | Mark |
| Smith | Database | Navathe |
| Smith | Operating Systems | Ammar |
| Smith | Theory | Schulman |
| Wallace | Database | Mark |
| Wallace | Operating Systems | Ahamad |
| Wong | Database | Omiecinski |
| Zelaya | Database | Navathe |

**FIGURE 10.13** A relation TEACH that is in 3NF but not BCNF.

All three decompositions "lose" the functional dependency FD1. The *desirable decomposition* of those just shown is 3, because it will not generate spurious tuples after a join.

A test to determine whether a decomposition is nonadditive (lossless) is discussed in Section 11.1.4 under Property LJ1. In general, a relation not in BCNF should be decomposed so as to meet this property, while possibly forgoing the preservation of all functional dependencies in the decomposed relations, as is the case in this example. Algorithm 11.3 does that and could be used above to give decomposition 3 for TEACH.

# 10.6 SUMMARY

In this chapter we first discussed several pitfalls in relational database design using intuitive arguments. We identified informally some of the measures for indicating whether a relation schema is "good" or "bad," and provided informal guidelines for a good design. We then presented some formal concepts that allow us to do relational design in a top-down fashion by analyzing relations individually. We defined this process of design by analysis and decomposition by introducing the process of normalization.

We discussed the problems of update anomalies that occur when redundancies are present in relations. Informal measures of good relation schemas include simple and clear attribute semantics and few nulls in the extensions (states) of relations. A good decomposition should also avoid the problem of generation of spurious tuples as a result of the join operation.

We defined the concept of functional dependency and discussed some of its properties. Functional dependencies specify semantic constraints among the attributes of a relation schema. We showed how from a given set of functional dependencies, additional dependencies can be inferred using a set of inference rules. We defined the concepts of closure and cover related to functional dependencies. We then defined

# Normalization…Motivation

- We saw how we can derive logical DB design (initial DB schema) from ER diagram.
- But how can we measure our work to be good…or better than other?
- How can we be sure that this schema is better than other one
- We need a formal way for doing so..!!!
- IC can be used to refine the conceptual schema produced

# Motivation…2

- We will concentrate on special class of IC which is **Functional Dependencies**

- We will start with an overview of the problems that normalization should address:

# Redundancy

- Redundancy of storage is the root

- **<u>Problem 1: Redundant Storage:</u>** information is stored repeatedly

- **<u>Problem 2: Update Anomalies:</u>** if one copy of one repeated data is updated, all other copies should be updated as well

- **Problem 3: Insertion Anomalies** it will be impossible to insert some data without inserting other, unrelated information as well

- **Problem 3:Delete Anomalies** it may be not possible to delete some information without deleting other, unrelated one.

**EMP_DEPT**

| ENAME | SSN | BDATE | ADDRESS | DNUMBER | DNAME | DMGRSSN |
|---|---|---|---|---|---|---|
| Smith,John B. | 123456789 | 1965-01-09 | 731 Fondren,Houston,TX | 5 | Research | 333445555 |
| Wong,Franklin T. | 333445555 | 1955-12-08 | 638 Voss,Houston,TX | 5 | Research | 333445555 |
| Zelaya, Alicia J. | 999887777 | 1968-07-19 | 3321 Castle,Spring,TX | 4 | Administration | 987654321 |
| Wallace,Jennifer S. | 987654321 | 1941-06-20 | 291 Berry,Bellaire,TX | 4 | Administration | 987654321 |
| Narayan,Ramesh K. | 666884444 | 1962-09-15 | 975 FireOak,Humble,TX | 5 | Research | 333445555 |
| English,Joyce A. | 453453453 | 1972-07-31 | 5631 Rice,Houston,TX | 5 | Research | 333445555 |
| Jabbar,Ahmad V. | 987987987 | 1969-03-29 | 980 Dallas,Houston,TX | 4 | Administration | 987654321 |
| Borg,James E. | 888665555 | 1937-11-10 | 450 Stone,Houston,TX | 1 | Headquarters | 888665555 |

# Decomposition

- In general, redundancy arises when a relational schema forces an association between attributes that is not natural.
- Functional Dependencies can be used
- Main idea is to decompose the relation into smaller relations.

# Problems Related to Decomposition

- Do we need to decompose the relation?
  - *Several normal forms have been found*
- What problems are associated with decomposition?

# Functional Dependencies

- A **functional dependency** (FD) is a kind of IC that generalizes the concept of a *key*

- Let $R$ be a relation schema and let $X$ and $Y$ be nonempty sets of attributes in $R$. We

say that an instance $r$ of $R$ satisfies the FD $X \rightarrow Y$ if the following holds for every pair of tuples $t1$ and $t2$ in $r$:

<u>If $t1{:}X = t2{:}X$, then $t1{:}Y = t2{:}Y$</u> .

# Example

- In the following relation: FD AB→C holds
- But if we add $\{a1; b1; c2; d1\}$ it wont hold

| $A$ | $B$ | $C$ | $D$ |
|---|---|---|---|
| a1 | b1 | c1 | d1 |
| a1 | b1 | c1 | d2 |
| a1 | b2 | c2 | d1 |
| a2 | b1 | c3 | d1 |

- A primary key constraint is a special case of an FD.
- Note,however, that the denition of an FD does not require that the set *X* be minimal;

- The additional minimality condition must be met for *X* to be a key

# SuperKey

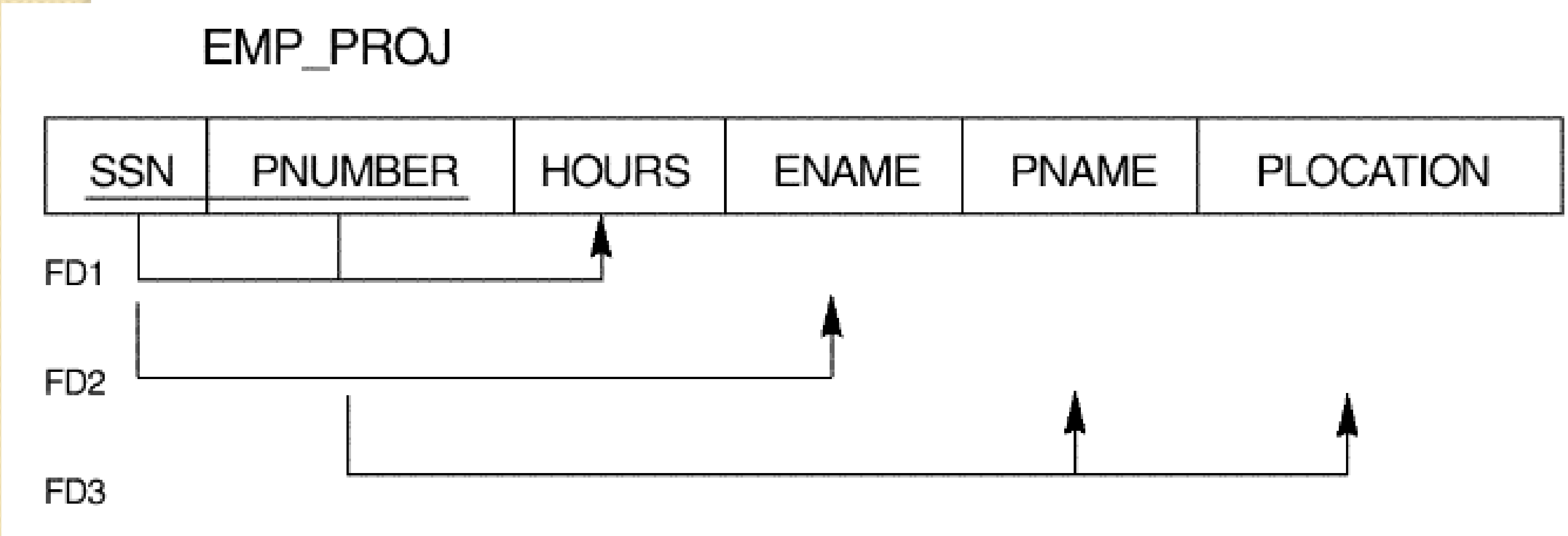
- If $X \rightarrow Y$ holds, where
- $Y$ is the set of all attributes, and there is some subset $V$ of $X$ such that $V \rightarrow Y$ holds,
- then $X$ is a *superkey*;

# Another Example

- Consider the Hourly Emps relation again. The constraint that attribute *ssn* is a key
- can be expressed as an FD:
- {*ssn*} → {*ssn; name; lot; rating; hourly wages; hours worked*}

- FD as *S* → *SNLRWH*, for simplicity
- Also R→W

- **legal relation states**) of *R*, obey the functional dependency constraints.
- Ex:



- Consider the relation schema EMP_PROJ in from the semantics of the attributes, we know that the following functional dependencies should hold:
- SSN → ENAME
- PNUMBER → {PNAME, PLOCATION}
- {SSN, PNUMBER} → HOURS

| S# | CITY | P# | QTY |
|---|---|---|---|
| S1 | London | P1 | 100 |
| S1 | London | P2 | 100 |
| S2 | Paris | P1 | 200 |
| S2 | Paris | P2 | 200 |
| S3 | Paris | P2 | 300 |
| S4 | London | P2 | 400 |
| S4 | London | P4 | 400 |
| S4 | London | P5 | 400 |

- S# → CITY
- S# → QTY
- QTY → S#
- {S#, P#} → QTY

# Inference Rules for Functional Dependencies

- We denote by *F* the set of functional dependencies that are specified on relation schema *R*

- We usually specify the FDs that are semantically obvious
- But there are other FDs that can be detucted

- The set of all such dependencies is called the **closure** of *F* and is denoted by F*
- To determine a systematic way to infer dependencies, we must discover a set of **inference rules** that can be used to infer new dependencies from a given set of dependencies.

- Reflexivity: If $X \supseteq Y$, then $X \rightarrow Y$.
- Y is a subset of X

- Augmentation: if $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z

- Transitivity : if $X \rightarrow Y$ and $Y \rightarrow Z$ then $X \rightarrow Z$
- Union: If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$
- Decomposition: if $X \rightarrow YZ$ then $X \rightarrow Y$ and $X \rightarrow Z$

1. The contract id $C$ is a key: $C \rightarrow CSJDPQV$.

2. A project purchases a given part using a single contract: $JP \rightarrow C$.

3. A department purchases at most one part from a supplier: $SD \rightarrow P$.

Several additional FDs hold in the closure of the set of given FDs:

From $JP \rightarrow C$, $C \rightarrow CSJDPQV$ and transitivity, we infer $JP \rightarrow CSJDPQV$.

From $SD \rightarrow P$ and augmentation, we infer $SDJ \rightarrow JP$.

From $SDJ \rightarrow JP$, $JP \rightarrow CSJDPQV$ and transitivity, we infer $SDJ \rightarrow CSJDPQV$. (Incidentally, while it may appear tempting to do so, we *cannot* conclude $SD \rightarrow CSDPQV$, canceling $J$ on both sides. FD inference is not like arithmetic multiplication!)

# Normalization

- Having studied functional dependencies and some of their properties, we are now ready to use them as information about the semantics of the relation schemas

- We assume that:
  -- a set of functional dependencies is given for each relation,
  --and that each relation has a designated primary key;
  --this information combined with the tests (conditions) for normal forms drives the normalization process

- takes a relation schema through a series of tests to "certify" whether it satisfies a certain **normal form**
- We have 3 normal forms
- All these normal forms are based on the functional dependencies among the attributes of a relation
- Unsatisfactory relation schemas that do not meet certain conditions—the **normal form tests**—<u>are decomposed into smaller relation schemas that meet the tests and hence possess the desirable properties</u>

# First Normal Form

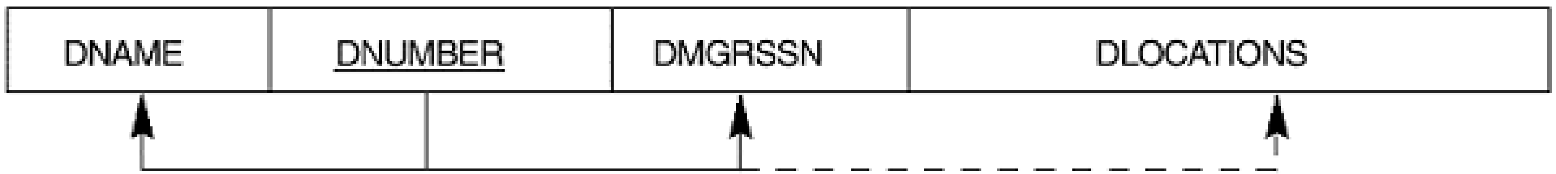
- historically, it was defined to disallow multivalued attributes, composite attributes, and their combinations

- It states that the domain of an attribute must include only *atomic* (simple, indivisible) *values* and that the value of any attribute in a tuple must be a *single value* from the domain of that attribute.

# Consider the DEPARTMENT relation schema shown

(a)

DEPARTMENT




(b)

DEPARTMENT

| DNAME | DNUMBER | DMGRSSN | DLOCATIONS |
|---|---|---|---|
| Research | 5 | 333445555 | {Bellaire, Sugarland, Houston} |
| Administration | 4 | 987654321 | {Stafford} |
| Headquarters | 1 | 888665555 | {Houston} |

(c)

DEPARTMENT

| DNAME | DNUMBER | DMGRSSN | DLOCATION |
|---|---|---|---|
| Research | 5 | 333445555 | Bellaire |
| Research | 5 | 333445555 | Sugarland |
| Research | 5 | 333445555 | Houston |
| Administration | 4 | 987654321 | Stafford |
| Headquarters | 1 | 888665555 | Houston |

- As we can see, this is not in 1NF because DLOCATIONS is not an atomic attribute
- DLOCATIONS *is not* functionally dependent on DNUMBER.

- here are three main techniques to achieve first normal form for such a relation

1. Remove the attribute DLOCATIONS that violates 1NF and place it in a separate relation DEPT_LOCATIONS along with the primary key DNUMBER of DEPARTMENT

   --The primary key of this relation is the combination {DNUMBER, DLOCATION}

2. Expand the key , In this case, the primary key becomes the combination {DNUMBER, DLOCATION}. This solution has the disadvantage of introducing *redundancy* in the relation. As in the prevous diagram (c)

3. If a *maximum number of values* is known for the attribute—for example, if it is known that *at most three locations* can exist for a department—replace the DLOCATIONS attribute by three atomic attributes: DLOCATION1, DLOCATION2, and DLOCATION3. This solution has the disadvantage of introducing *null values* if most departments have fewer than three locations.

# 1 NF

- The first normal form also disallows multivalued attributes that are themselves composite.
- These are called **nested relations** because each tuple can have a relation *within it.*
- Take a look at the following diagram:

(a) **EMP_PROJ**

| SSN | ENAME | PROJS | |
|---|---|---|---|
| | | PNUMBER | HOURS |

(b) **EMP_PROJ**

| SSN | ENAME | PNUMBER | HOURS |
|---|---|---|---|
| 123456789 | Smith,John B. | 1 | 32.5 |
| | | 2 | 7.5 |
| 666884444 | Narayan,Ramesh K. | 3 | 40.0 |
| 453453453 | English,Joyce A. | 1 | 20.0 |
| | | 2 | 20.0 |
| 333445555 | Wong,Franklin T. | 2 | 10.0 |
| | | 3 | 10.0 |
| | | 10 | 10.0 |
| | | 20 | 10.0 |
| 999887777 | Zelaya,Alicia J. | 30 | 30.0 |
| | | 10 | 10.0 |
| 987987987 | Jabbar,Ahmad V. | 10 | 35.0 |
| | | 30 | 5.0 |
| 987654321 | Wallace,Jennifer S. | 30 | 20.0 |
| | | 20 | 15.0 |
| 888665555 | Borg,James E. | 20 | null |

- To normalize this into 1NF, we remove the nested relation attributes into a new relation and *propagate the primary key* into it;

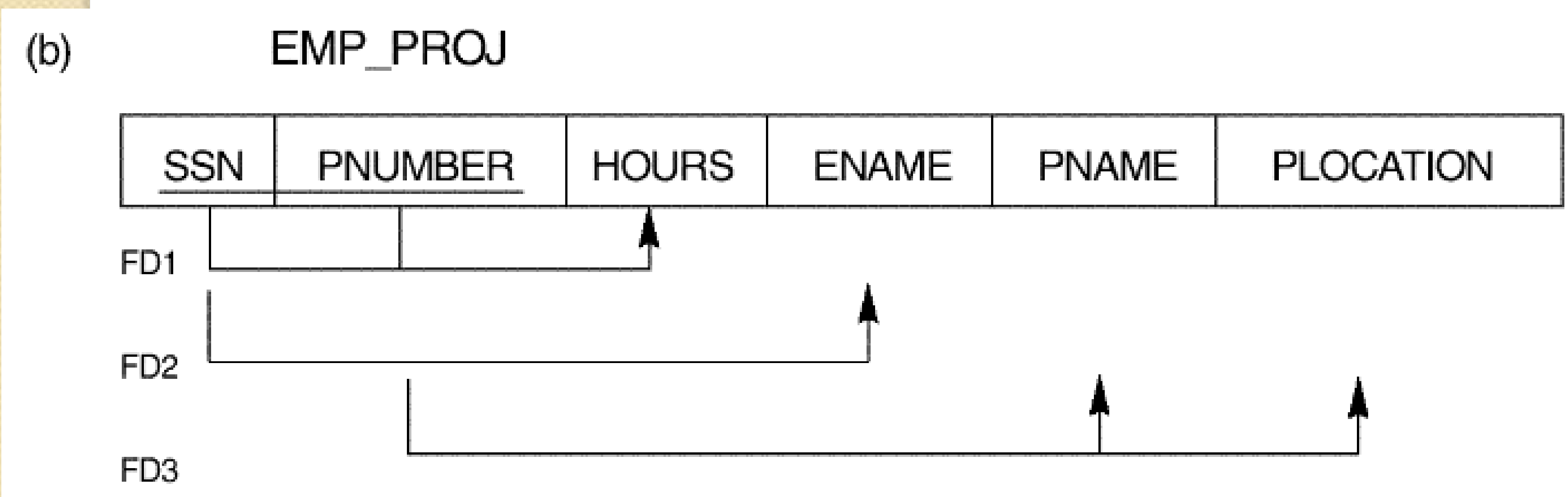
- This procedure can be applied recursively to a relation with multiple-level nesting to **unnest** the relation into a set of 1NF relations

# 2 NF

- **Second normal form (2NF)** is based on the concept of *full functional dependency.*

- A functional dependency $X \rightarrow Y$ is a **full functional dependency** if removal of any attribute $A$ from $X$ means that the dependency does not hold any more;

- A functional dependency $X \rightarrow Y$ is a **partial dependency** if some attribute $A$   $X$ can be removed from $X$ and the dependency still holds



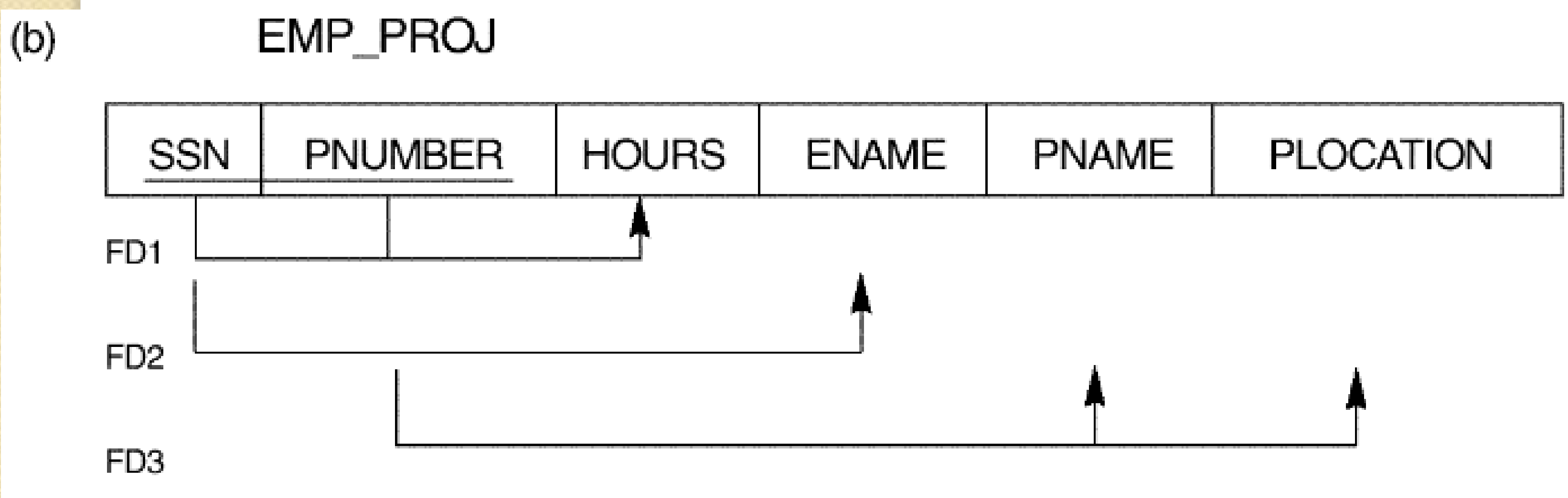
- {SSN, PNUMBER}→ HOURS is a full dependency (neither SSN → HOURS nor PNUMBER → HOURS holds).

- However, the dependency {SSN, PNUMBER} → ENAME is partial because SSN → ENAME holds.

- The test for 2NF involves testing for functional dependencies whose left-hand side attributes are part of the primary key.
- If the primary key contains a single attribute, the test need not be applied at all.
- A relation schema *R* is in **2NF** if every nonprime attribute *A* in *R* is *fully functionally dependent* on the primary key of *R*

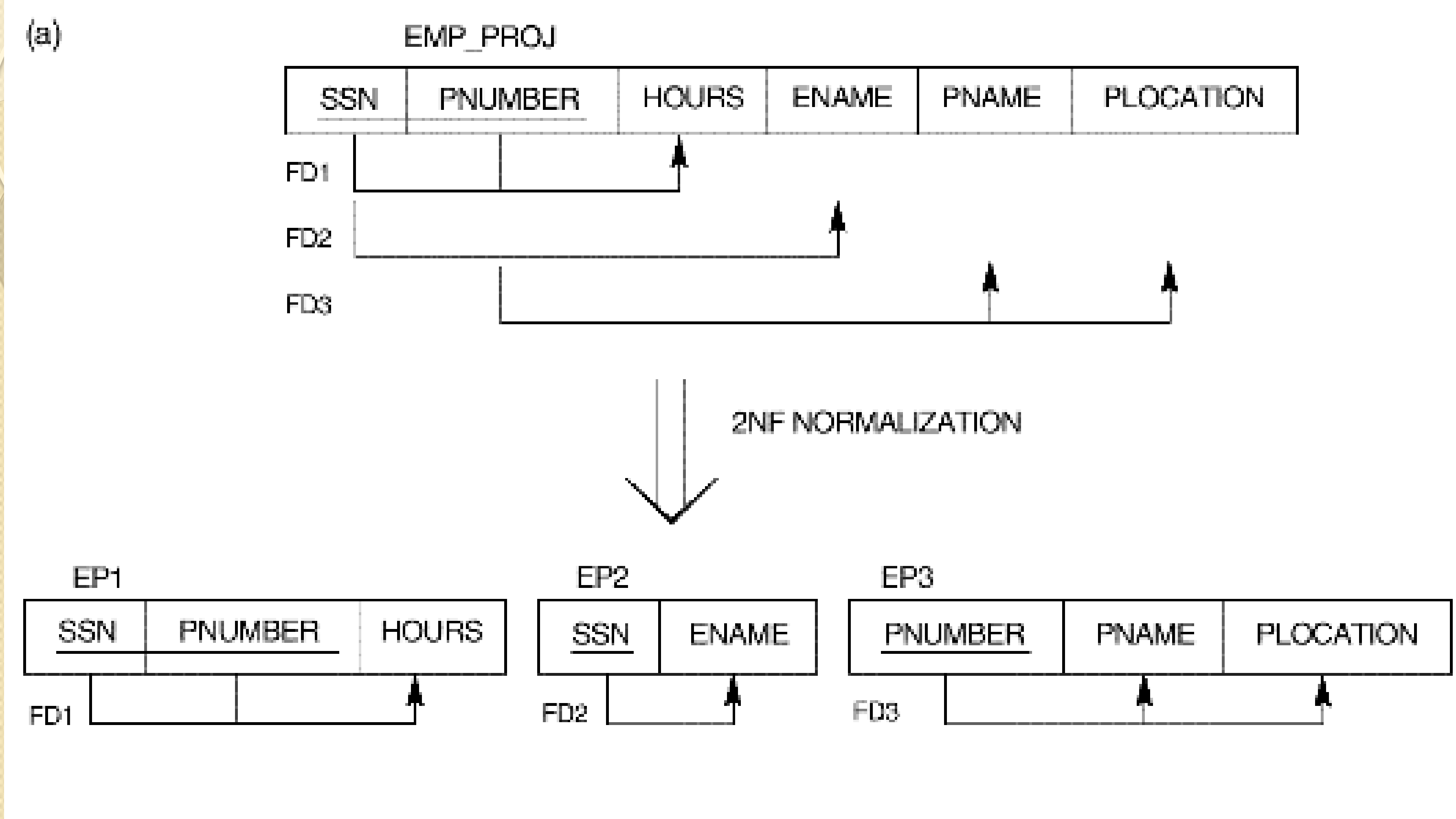
•The EMP_PROJ relation is in 1NF but is not in 2NF.

•The nonprime attribute ENAME violates 2NF . because of FD2, as do the nonprime . attributes PNAME and PLOCATION because of FD3



- If a relation schema is not in 2NF, it can be "second normalized" or "2NF normalized" into a number of 2NF relations in which nonprime attributes are associated only with the part of the primary key on which they are fully functionally dependent.

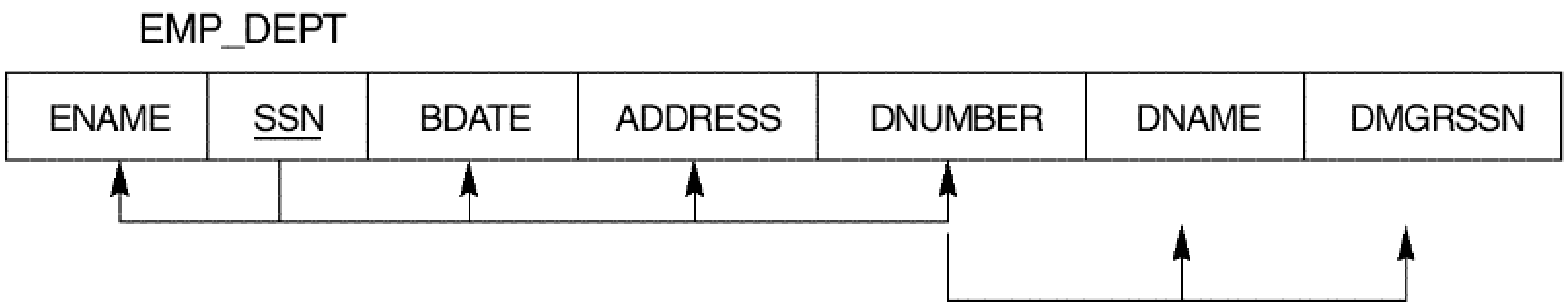
- The functional dependencies FD1, FD2, and FD3 in hence lead to the decomposition of EMP_PROJ into the three relation schemas EP1, EP2, and EP3 shown in


(a)
EMP_PROJ
SSN
PNUMBER
HOURS
ENAME
PNAME
PLOCATION
FD1
FD2
FD3
2NF NORMALIZATION
EP1
SSN
PNUMBER
HOURS
FD1
EP2
SSN
ENAME
FD2
EP3
PNUMBER
PNAME
PLOCATION
FD3

# 3 NF

- **Third normal form (3NF)** is based on the concept of *transitive dependency*

- A functional dependency $X \rightarrow Y$ in a relation schema $R$ is a **transitive dependency** if there is a set of attributes $Z$ that is neither a candidate key nor a subset of any key of $R$, and both $X \rightarrow Z$ and $Z \rightarrow Y$ hold.

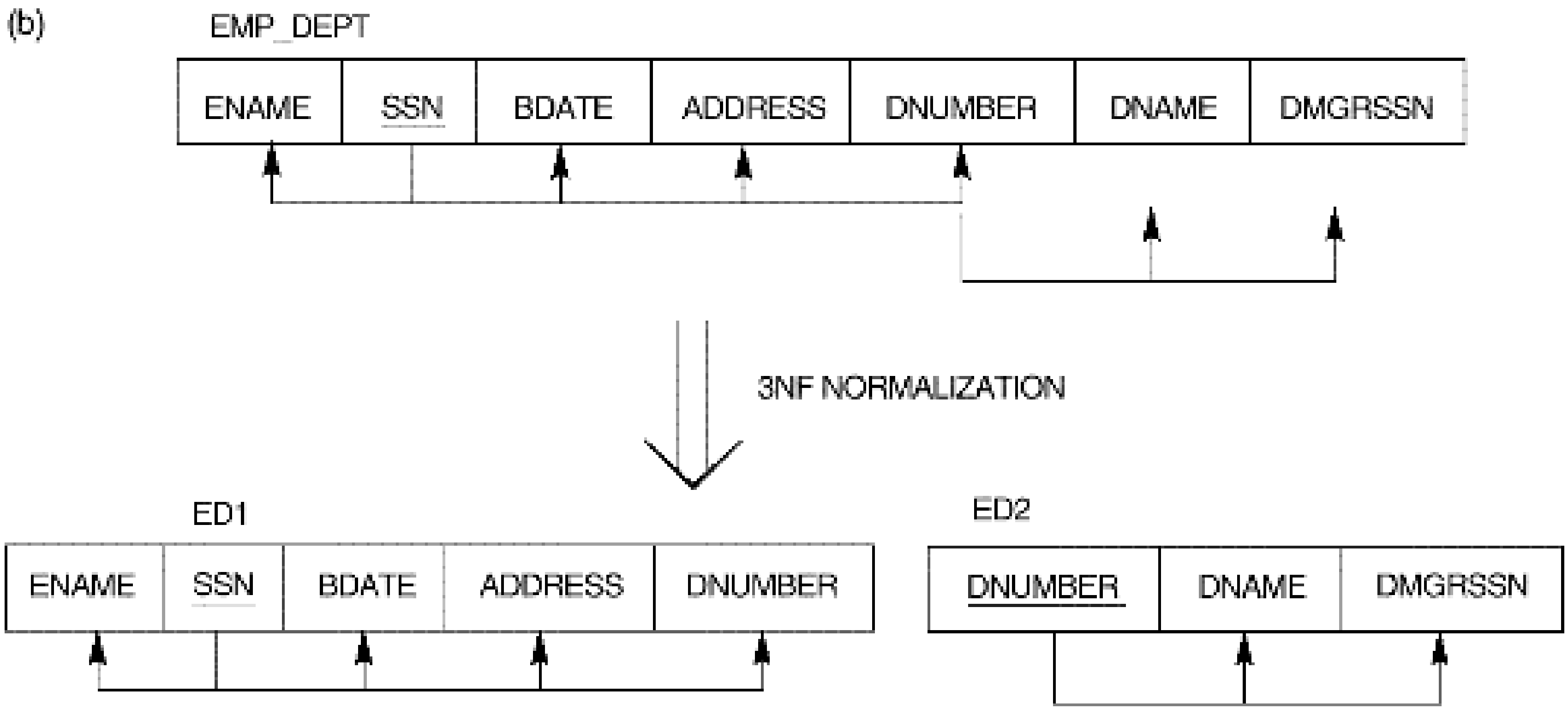

EMP_DEPT
ENAME
SSN
BDATE
ADDRESS
DNUMBER
DNAME
DMGRSSN

- The dependency SSN → DMGRSSN is transitive through DNUMBER in EMP_DEPT
- because both the dependencies SSN → DNUMBER and DNUMBER → DMGRSSN hold *and* DNUMBER is neither a key itself nor a subset of the key of EMP_DEPT.
- <u>we can see that the dependency of DMGRSSN on DNUMBER is undesirable in EMP_DEPT since DNUMBER is not a key of EMP_DEPT.</u>

- According to Codd's original definition, a relation schema *R* is in **3NF** if it satisfies 2NF *and* no nonprime attribute of *R* is transitively dependent on the primary key.

- The relation schema EMP_DEPT in is in 2NF, since no partial dependencies on a key exist.
- However, EMP_DEPT is not in 3NF because of the transitive dependency of DMGRSSN (and also DNAME) on SSN via DNUMBER.

- We can normalize EMP_DEPT by decomposing it into the two 3NF relation schemas ED1 and ED2



(b)
EMP_DEPT
ENAME
SSN
BDATE
ADDRESS
DNUMBER
DNAME
DMGRSSN
3NF NORMALIZATION
ED1
ENAME
SSN
BDATE
ADDRESS
DNUMBER
ED2
DNUMBER
DNAME
DMGRSSN

# General definition of normal forms

- The steps for normalization into 3NF relations that we discussed so far disallow partial and transitive dependencies on the *primary key*.

- These definitions, however, do not take other candidate keys of a relation, if any, into account.

- In this section we give the more general definitions of 2NF and 3NF that take *all* candidate keys of a relation into account

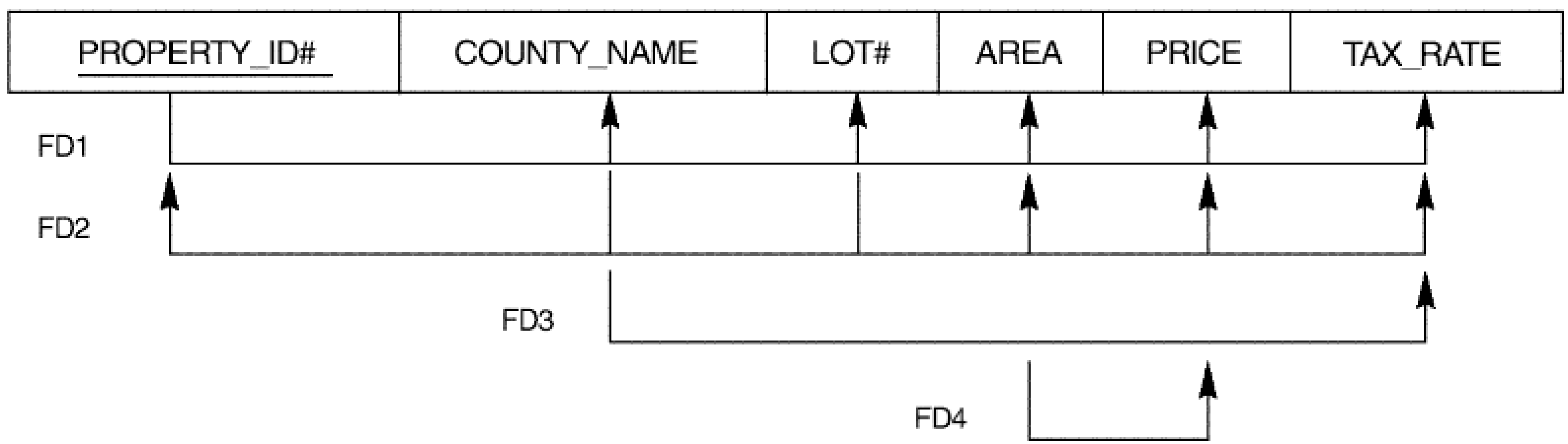
- As a general definition of **prime attribute,** an attribute that is part of *any candidate key* will be considered as prime.

- Partial and full functional dependencies and transitive dependencies will now be *with respect to all candidate keys* of a relation.

- A relation schema *R* is in **second normal form (2NF)** if every nonprime attribute *A* in *R* is not partially dependent on ***any key*** of R

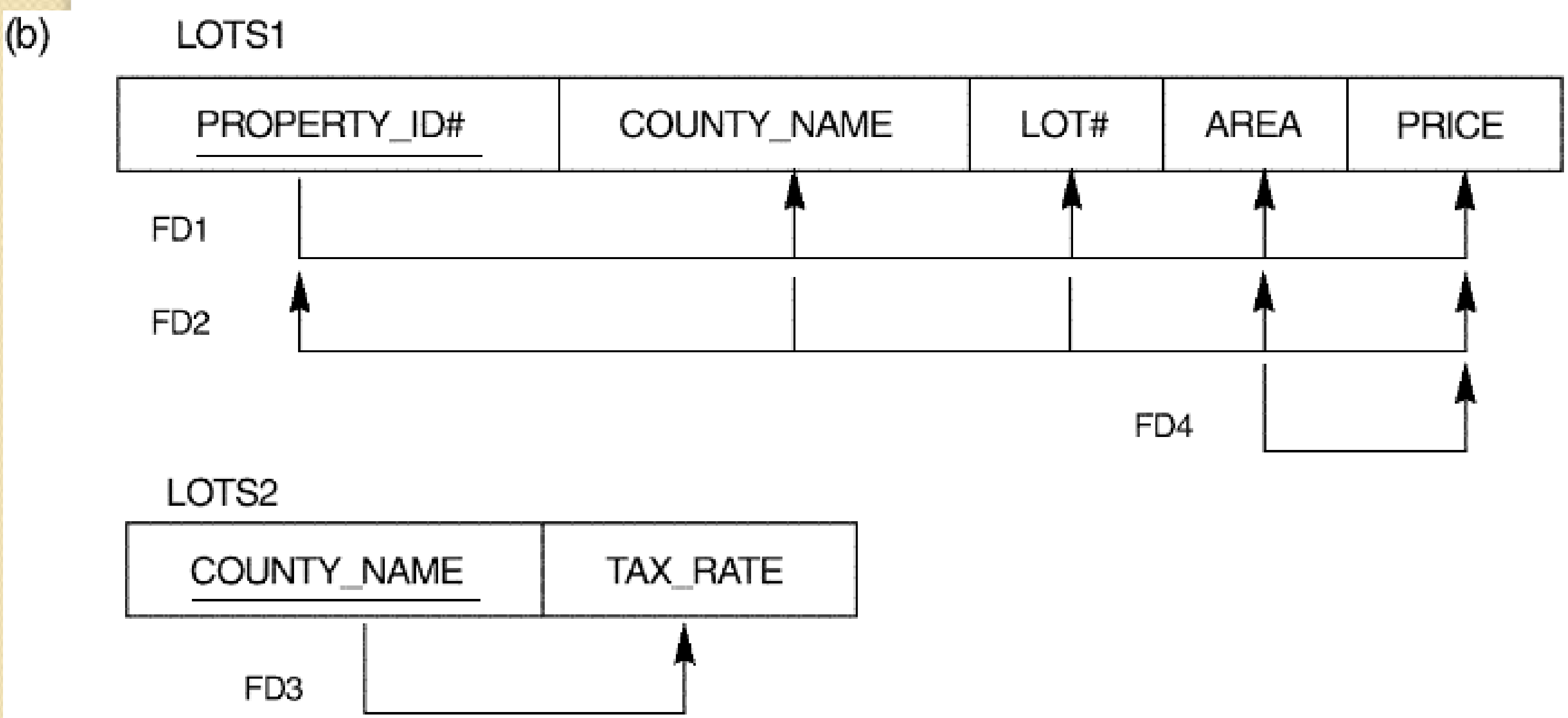
- Consider the following relation

(a) LOTS


PROPERTY_ID#
COUNTY_NAME
LOT#
AREA
PRICE
TAX_RATE
FD1
FD2
FD3
FD4

- The LOTS relation schema violates the general definition of 2NF
- because TAX_RATE is partially dependent on the candidate key {COUNTY_NAME, LOT#}, due to FD3

- To normalize LOTS into 2NF, we decompose it into the two relations LOTS1 and LOTS2,



# General Definition of Third Normal Form

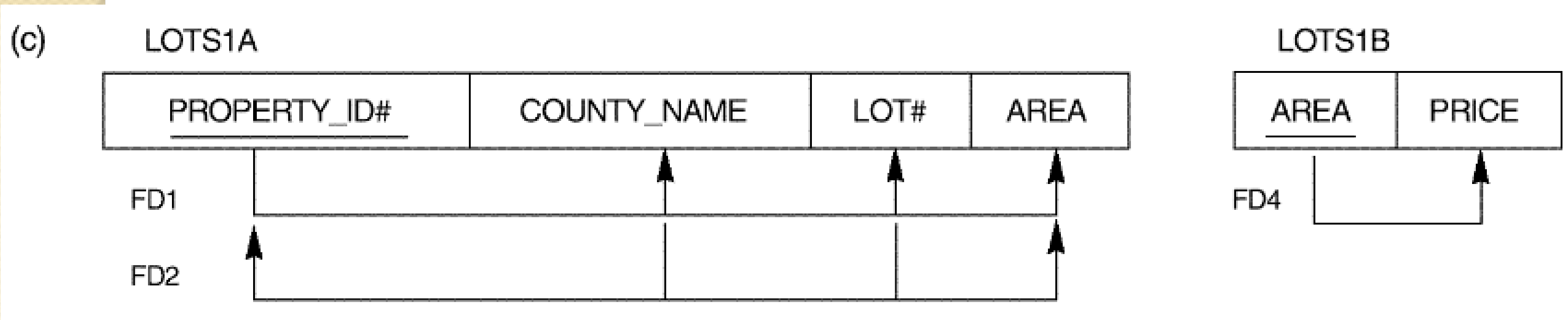
- A relation schema *R* is in **third normal form (3NF)** if, whenever a *nontrivial* functional dependency $X \rightarrow A$ holds in *R*, **<u>either</u>**

- (a) *X* is a (candidate key)of *R*, or

- (b) *A* is a prime attribute of *R*.

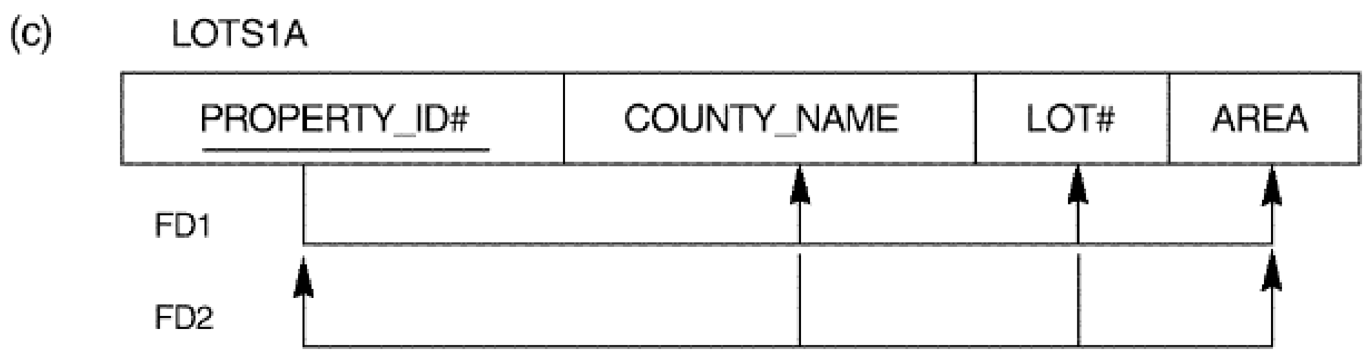
- LOTS2 is in 3NF. However, FD4 in LOTS1 violates 3NF because AREA is not a candidate key and PRICE is not a prime attribute in LOTS1

- To normalize LOTS1 into 3NF, we decompose it into the relation schemas LOTS1A and LOTS1B


(c)
LOTS1A
PROPERTY_ID#
COUNTY_NAME
LOT#
AREA
FD1
FD2
LOTS1B
AREA
PRICE
FD4

- **Boyce-Codd normal form (BCNF)** was proposed as a simpler form of 3NF,
- but it was found to be stricter than 3NF,
- because every relation in BCNF is also in 3NF; however, a relation in 3NF is *not necessarily* in BCNF

- Lets go back to this schema




And let us add this FD AREA → County_Name

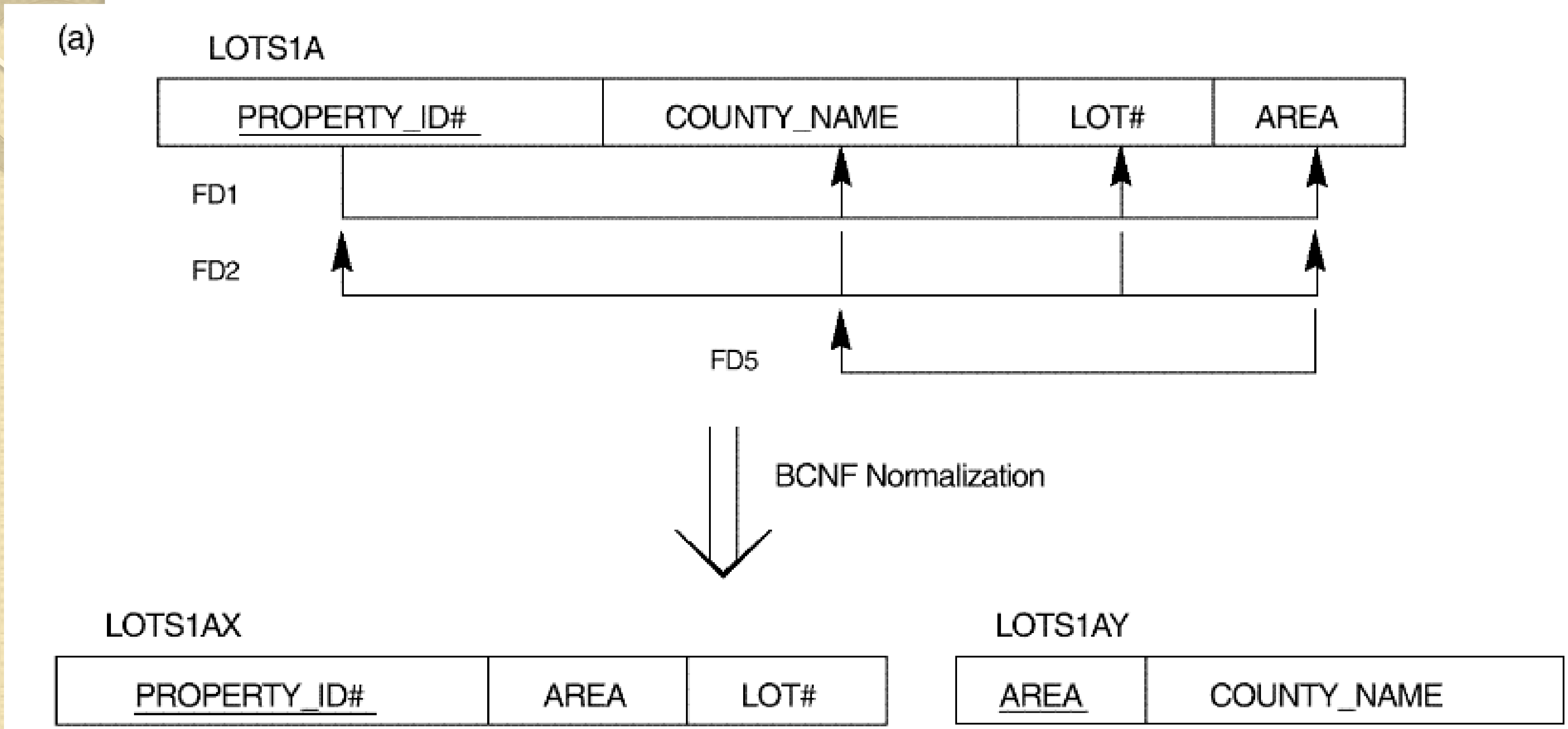
- the relation schema LOTS1A still is in 3NF because COUNTY_NAME is a prime attribute.

- Definition: **A relation schema *R* is in BCNF if whenever a *nontrivial* functional dependency *X* → *A* holds in *R*, then *X* is a superkey (candidate key)of *R*.**

- In our example, AREA→ County_name violates BCNF in LOTS1A because AREA is not a superkey of LOTS1A

- Note that FD5 satisfies 3NF in LOTS1A because COUNTY_NAME is a prime attribute

- We can decompose LOTS1A into two BCNF relations LOTS1AX and LOTS1AY,



# Examples on Armstrong Axioms

- Prove Union:
- $X \rightarrow Y, X \rightarrow Z$
- $X \rightarrow YZ$
- Prove Decomposition
- $X \rightarrow YZ, X \rightarrow Y, X \rightarrow Z$
- ABCDEFGHIJ
- $AB \rightarrow E$
- $AG \rightarrow J$
- $BE \rightarrow I$
- $E \rightarrow G$
- $GI \rightarrow H$
- Prove $AB \rightarrow GH$

# 6

# QUERY-BY-EXAMPLE (QBE)

Example is always more efficacious than precept.

—Samuel Johnson

## 6.1 INTRODUCTION

Query-by-Example (QBE) is another language for querying (and, like SQL, for creating and modifying) relational data. It is different from SQL, and from most other database query languages, in having a graphical user interface that allows users to write queries by creating *example tables* on the screen. A user needs minimal information to get started and the whole language contains relatively few concepts. QBE is especially suited for queries that are not too complex and can be expressed in terms of a few tables.

QBE, like SQL, was developed at IBM and QBE is an IBM trademark, but a number of other companies sell QBE-like interfaces, including Paradox. Some systems, such as Microsoft Access, offer partial support for form-based queries and reflect the influence of QBE. Often a QBE-like interface is offered in addition to SQL, with QBE serving as a more intuitive user-interface for simpler queries and the full power of SQL available for more complex queries. An appreciation of the features of QBE offers insight into the more general, and widely used, paradigm of tabular query interfaces for relational databases.

This presentation is based on IBM's Query Management Facility (QMF) and the QBE version that it supports (Version 2, Release 4). This chapter explains how a tabular interface can provide the expressive power of relational calculus (and more) in a user-friendly form. The reader should concentrate on the connection between QBE and domain relational calculus (DRC), and the role of various important constructs (e.g., the conditions box), rather than on QBE-specific details. We note that every QBE query can be expressed in SQL; in fact, QMF supports a command called `CONVERT` that generates an SQL query from a QBE query.

We will present a number of example queries using the following schema:

Sailors(<u>*sid:* `integer`</u>, *sname:* `string`, *rating:* `integer`, *age:* `real`)

Boats(*bid:* `integer`, *bname:* `string`, *color:* `string`)
Reserves(*sid:* `integer`, *bid:* `integer`, *day:* `dates`)

The key fields are underlined, and the domain of each field is listed after the field name.

We introduce QBE queries in Section 6.2 and consider queries over multiple relations in Section 6.3. We consider queries with set-difference in Section 6.4 and queries with aggregation in Section 6.5. We discuss how to specify complex constraints in Section 6.6. We show how additional computed fields can be included in the answer in Section 6.7. We discuss update operations in QBE in Section 6.8. Finally, we consider relational completeness of QBE and illustrate some of the subtleties of QBE queries with negation in Section 6.9.

## 6.2 BASIC QBE QUERIES

A user writes queries by creating *example tables*. QBE uses *domain variables*, as in the DRC, to create example tables. The domain of a variable is determined by the column in which it appears, and variable symbols are prefixed with underscore (_) to distinguish them from constants. Constants, including strings, appear unquoted, in contrast to SQL. The fields that should appear in the answer are specified by using the command `P.`, which stands for *print*. The fields containing this command are analogous to the *target-list* in the `SELECT` clause of an SQL query.

We introduce QBE through example queries involving just one relation. To print the names and ages of all sailors, we would create the following example table:

| *Sailors* | *sid* | *sname* | *rating* | *age* |
|---|---|---|---|---|
| | | P._N | | P._A |

A variable that appears only once can be omitted; QBE supplies a unique new name internally. Thus the previous query could also be written by omitting the variables _N and _A, leaving just `P.` in the *sname* and *age* columns. The query corresponds to the following DRC query, obtained from the QBE query by introducing existentially quantified domain variables for each field.

$$\{\langle N, A\rangle \mid \exists I, T(\langle I, N, T, A\rangle \in Sailors)\}$$

A large class of QBE queries can be translated to DRC in a direct manner. (Of course, queries containing features such as aggregate operators cannot be expressed in DRC.) We will present DRC versions of several QBE queries. Although we will not define the translation from QBE to DRC formally, the idea should be clear from the examples;

intuitively, there is a term in the DRC query for each row in the QBE query, and the terms are connected using $\wedge$.[1]

A convenient shorthand notation is that if we want to print all fields in some relation, we can place `P.` under the name of the relation. This notation is like the `SELECT *` convention in SQL. It is equivalent to placing a `P.` in every field:

| *Sailors* | *sid* | *sname* | *rating* | *age* |
|---|---|---|---|---|
| P. | | | | |

Selections are expressed by placing a constant in some field:

| *Sailors* | *sid* | *sname* | *rating* | *age* |
|---|---|---|---|---|
| P. | | | 10 | |

Placing a constant, say 10, in a column is the same as placing the condition *=10*. This query is very similar in form to the equivalent DRC query

$$\{\langle I, N, 10, A\rangle \mid \langle I, N, 10, A\rangle \in \mathit{Sailors}\}$$

We can use other comparison operations ($<, >, <=, >=, \neg$) as well. For example, we could say $< 10$ to retrieve sailors with a rating less than 10 or say $\neg 10$ to retrieve sailors whose rating is not equal to 10. The expression $\neg 10$ in an attribute column is the same as $\neq 10$. As we will see shortly, $\neg$ under the relation name denotes (a limited form of) $\neg\exists$ in the relational calculus sense.

### 6.2.1 Other Features: Duplicates, Ordering Answers

We can explicitly specify whether duplicate tuples in the answer are to be eliminated (or not) by putting `UNQ.` (respectively `ALL.`) under the relation name.

We can order the presentation of the answers through the use of the `.AO` (for *ascending order*) and `.DO` commands in conjunction with `P.` An optional integer argument allows us to sort on more than one field. For example, we can display the names, ages, and ratings of all sailors in ascending order by age, and for each age, in ascending order by rating as follows:

| *Sailors* | *sid* | *sname* | *rating* | *age* |
|---|---|---|---|---|
| | | P. | P.AO(2) | P.AO(1) |

[1] The semantics of QBE is unclear when there are several rows containing `P.` or if there are rows that are not linked via shared variables to the row containing `P.` We will discuss such queries in Section 6.6.1.

## 6.3 QUERIES OVER MULTIPLE RELATIONS

To find sailors with a reservation, we have to combine information from the Sailors and the Reserves relations. In particular we have to select tuples from the two relations with the same value in the join column *sid*. We do this by placing the same variable in the *sid* columns of the two example relations.

| *Sailors* | *sid* | *sname* | *rating* | *age* |
|---|---|---|---|---|
| | _Id | P._S | | |

| *Reserves* | *sid* | *bid* | *day* |
|---|---|---|---|
| | _Id | | |

To find sailors who have reserved a boat for 8/24/96 and who are older than 25, we could write:[2]

| *Sailors* | *sid* | *sname* | *rating* | *age* |
|---|---|---|---|---|
| | _Id | P._S | | $> 25$ |

| *Reserves* | *sid* | *bid* | *day* |
|---|---|---|---|
| | _Id | | '8/24/96' |

Extending this example, we could try to find the colors of Interlake boats reserved by sailors who have reserved a boat for 8/24/96 and who are older than 25:

| *Sailors* | *sid* | *sname* | *rating* | *age* |
|---|---|---|---|---|
| | _Id | | | $> 25$ |

| *Reserves* | *sid* | *bid* | *day* |
|---|---|---|---|
| | _Id | _B | '8/24/96' |

| *Boats* | *bid* | *bname* | *color* |
|---|---|---|---|
| | _B | Interlake | P. |

As another example, the following query prints the names and ages of sailors who have reserved some boat that is also reserved by the sailor with id 22:

| *Sailors* | *sid* | *sname* | *rating* | *age* |
|---|---|---|---|---|
| | _Id | P._N | | |

| *Reserves* | *sid* | *bid* | *day* |
|---|---|---|---|
| | _Id | _B | |
| | 22 | _B | |

Each of the queries in this section can be expressed in DRC. For example, the previous query can be written as follows:

$$\{\langle N\rangle \mid \exists Id, T, A, B, D1, D2(\langle Id, N, T, A\rangle \in Sailors$$
$$\wedge \langle Id, B, D1\rangle \in Reserves \wedge \langle 22, B, D2\rangle \in Reserves)\}$$

[2] Incidentally, note that we have quoted the date value. In general, constants are not quoted in QBE. The exceptions to this rule include date values and string values with embedded blanks or special characters.

Notice how the only free variable ($N$) is handled and how $Id$ and $B$ are repeated, as in the QBE query.

## 6.4 NEGATION IN THE RELATION-NAME COLUMN

We can print the names of sailors who do *not* have a reservation by using the ¬ command in the relation name column:

| *Sailors* | *sid* | *sname* | *rating* | *age* |
|---|---|---|---|---|
| | _Id | `P.`_S | | |

| *Reserves* | *sid* | *bid* | *day* |
|---|---|---|---|
| ¬ | _Id | | |

This query can be read as follows: "Print the *sname* field of Sailors tuples such that there is *no* tuple in Reserves with the same value in the *sid* field." Note the importance of *sid* being a key for Sailors. In the relational model, keys are the only available means for *unique identification* (of sailors, in this case). (Consider how the meaning of this query would change if the Reserves schema contained *sname*—which is not a key!—rather than *sid*, and we used a common variable in this column to effect the join.)

All variables in a negative row (i.e., a row that is preceded by ¬) must also appear in positive rows (i.e., rows not preceded by ¬). Intuitively, variables in positive rows can be instantiated in many ways, based on the tuples in the input instances of the relations, and each negative row involves a simple check to see if the corresponding relation contains a tuple with certain given field values.

The use of ¬ in the relation-name column gives us a limited form of the set-difference operator of relational algebra. For example, we can easily modify the previous query to find sailors who are not (both) younger than 30 and rated higher than 4:

| *Sailors* | *sid* | *sname* | *rating* | *age* |
|---|---|---|---|---|
| | _Id | `P.`_S | | |

| *Sailors* | *sid* | *sname* | *rating* | *age* |
|---|---|---|---|---|
| ¬ | _Id | | $> 4$ | $< 30$ |

This mechanism is not as general as set-difference, because there is no way to control the order in which occurrences of ¬ are considered if a query contains more than one occurrence of ¬. To capture full set-difference, views can be used. (The issue of QBE's relational completeness, and in particular the ordering problem, is discussed further in Section 6.9.)

## 6.5 AGGREGATES

Like SQL, QBE supports the aggregate operations `AVG.`, `COUNT.`, `MAX.`, `MIN.`, and `SUM.` By default, these aggregate operators do *not* eliminate duplicates, with the exception of `COUNT.`, which does eliminate duplicates. To eliminate duplicate values, the variants `AVG.UNQ.` and `SUM.UNQ.` must be used. (Of course, this is irrelevant for `MIN.` and `MAX.`) Curiously, there is no variant of `COUNT.` that does *not* eliminate duplicates.

Consider the instance of Sailors shown in Figure 6.1. On this instance the following

| *sid* | *sname* | *rating* | *age* |
|---|---|---|---|
| 22 | dustin | 7 | 45.0 |
| 58 | rusty | 10 | 35.0 |
| 44 | horatio | 7 | 35.0 |

**Figure 6.1** An Instance of Sailors

query prints the value 38.3:

| *Sailors* | *sid* | *sname* | *rating* | *age* | |
|---|---|---|---|---|---|
| | | | | _A | `P.AVG._A` |

Thus, the value 35.0 is counted twice in computing the average. To count each age only once, we could specify `P.AVG.UNQ.` instead, and we would get 40.0.

QBE supports *grouping*, as in SQL, through the use of the `G.` command. To print average ages by rating, we could use:

| *Sailors* | *sid* | *sname* | *rating* | *age* | |
|---|---|---|---|---|---|
| | | | `G.P.` | _A | `P.AVG._A` |

To print the answers in sorted order by rating, we could use `G.P.AO` or `G.P.DO.` instead. When an aggregate operation is used in conjunction with `P.`, or there is a use of the `G.` operator, every column to be printed must specify either an aggregate operation or the `G.` operator. (Note that SQL has a similar restriction.) If `G.` appears in more than one column, the result is similar to placing each of these column names in the `GROUP BY` clause of an SQL query. If we place `G.` in the *sname* and *rating* columns, all tuples in each group have the same *sname* value and also the same *rating* value.

We consider some more examples using aggregate operations after introducing the conditions box feature.

## 6.6 THE CONDITIONS BOX

Simple conditions can be expressed directly in columns of the example tables. For more complex conditions QBE provides a feature called a **conditions box**.

Conditions boxes are used to do the following:

- *Express a condition involving two or more columns*, such as *_R/_A > 0.2*.
- *Express a condition involving an aggregate operation on a group*, for example, `AVG.`*_A > 30*. Notice that this use of a conditions box is similar to the `HAVING` clause in SQL. The following query prints those ratings for which the average age is more than 30:

| *Sailors* | *sid* | *sname* | *rating* | *age* |
|---|---|---|---|---|
| | | | `G.P.` | _A |

| *Conditions* |
|---|
| `AVG.`_A > 30 |

As another example, the following query prints the *sid*s of sailors who have reserved all boats for which there is some reservation:

| *Sailors* | *sid* | *sname* | *rating* | *age* |
|---|---|---|---|---|
| | `P.G.`_Id | | | |

| *Reserves* | *sid* | *bid* | *day* |
|---|---|---|---|
| | _Id | _B1 | |
| | | _B2 | |

| *Conditions* |
|---|
| `COUNT.`_B1 = `COUNT.`_B2 |

For each _Id value (notice the `G.` operator), we count all _B1 values to get the number of (distinct) *bid* values reserved by sailor _Id. We compare this count against the count of all _B2 values, which is simply the total number of (distinct) *bid* values in the Reserves relation (i.e., the number of boats with reservations). If these counts are equal, the sailor has reserved all boats for which there is some reservation. Incidentally, the following query, intended to print the names of such sailors, is incorrect:

| *Sailors* | *sid* | *sname* | *rating* | *age* |
|---|---|---|---|---|
| | `P.G.`_Id | `P.` | | |

| *Reserves* | *sid* | *bid* | *day* |
|---|---|---|---|
| | _Id | _B1 | |
| | | _B2 | |

| *Conditions* |
|---|
| `COUNT.`_B1 = `COUNT.`_B2 |

The problem is that in conjunction with `G.`, only columns with either `G.` or an aggregate operation can be printed. This limitation is a direct consequence of the SQL definition of `GROUPBY`, which we discussed in Section 5.5.1; QBE is typically implemented by translating queries into SQL. If `P.G.` replaces `P.` in the *sname* column, the query is legal, and we then group by both *sid* and *sname*, which results in the same groups as before because *sid* is a key for Sailors.

- *Express conditions involving the* `AND` *and* `OR` *operators.* We can print the names of sailors who are younger than 20 *or* older than 30 as follows:

| *Sailors* | *sid* | *sname* | *rating* | *age* | *Conditions* |
|---|---|---|---|---|---|
| | | `P.` | | _A | _A < 20 `OR` 30 < _A |

We can print the names of sailors who are both younger than 20 *and* older than 30 by simply replacing the condition with *_A < 20* `AND` *30 < _A*; of course, the set of such sailors is always empty! We can print the names of sailors who are either older than 20 *or* have a rating equal to 8 by using the condition *20 < _A* `OR` *_R = 8*, and placing the variable _R in the *rating* column of the example table.

### 6.6.1 And/Or Queries

It is instructive to consider how queries involving `AND` and `OR` can be expressed in QBE without using a conditions box. We can print the names of sailors who are younger than 30 *or* older than 20 by simply creating two example rows:

| *Sailors* | *sid* | *sname* | *rating* | *age* |
|---|---|---|---|---|
| | | `P.` | | < 30 |
| | | `P.` | | > 20 |

To translate a QBE query with several rows containing `P.`, we create subformulas for each row with a `P.` and connect the subformulas through $\vee$. If a row containing `P.` is linked to other rows through shared variables (which is not the case in this example), the subformula contains a term for each linked row, all connected using $\wedge$. Notice how the answer variable $N$, which must be a free variable, is handled:

$$\{\langle N\rangle \mid \exists I1, N1, T1, A1, I2, N2, T2, A2($$
$$\langle I1, N1, T1, A1\rangle \in Sailors(A1 < 30 \wedge N = N1)$$
$$\vee\langle I2, N2, T2, A2\rangle \in Sailors(A2 > 20 \wedge N = N2))\}$$

To print the names of sailors who are both younger than 30 *and* older than 20, we use the same variable in the key fields of both rows:

| *Sailors* | *sid* | *sname* | *rating* | *age* |
|---|---|---|---|---|
| | _Id | `P.` | | $< 30$ |
| | _Id | | | $> 20$ |

The DRC formula for this query contains a term for each linked row, and these terms are connected using $\wedge$:

$$\{\langle N\rangle \mid \exists I1, N1, T1, A1, N2, T2, A2$$
$$(\langle I1, N1, T1, A1\rangle \in Sailors(A1 < 30 \wedge N = N1)$$
$$\wedge \langle I1, N2, T2, A2\rangle \in Sailors(A2 > 20 \wedge N = N2))\}$$

Compare this DRC query with the DRC version of the previous query to see how closely they are related (and how closely QBE follows DRC).

## 6.7 UNNAMED COLUMNS

If we want to display some information in addition to fields retrieved from a relation, we can create *unnamed columns* for display.[3] As an example—admittedly, a silly one!—we could print the name of each sailor along with the ratio *rating/age* as follows:

| *Sailors* | *sid* | *sname* | *rating* | *age* | |
|---|---|---|---|---|---|
| | | `P.` | _R | _A | `P.`_R / _A |

All our examples thus far have included `P.` commands in exactly one table. This is a QBE restriction. If we want to display fields from more than one table, we have to use unnamed columns. To print the names of sailors along with the dates on which they have a boat reserved, we could use the following:

| *Sailors* | *sid* | *sname* | *rating* | *age* | |
|---|---|---|---|---|---|
| | _Id | `P.` | | | `P.`_D |

| *Reserves* | *sid* | *bid* | *day* |
|---|---|---|---|
| | _Id | | _D |

Note that unnamed columns should not be used for expressing *conditions* such as *_D >8/9/96*; a conditions box should be used instead.

## 6.8 UPDATES

Insertion, deletion, and modification of a tuple are specified through the commands `I.`, `D.`, and `U.`, respectively. We can insert a new tuple into the Sailors relation as follows:

[3]A QBE facility includes simple commands for drawing empty example tables, adding fields, and so on. We do not discuss these features but assume that they are available.

| Sailors | sid | sname | rating | age |
|---|---|---|---|---|
| `I.` | 74 | Janice | 7 | 41 |

We can insert several tuples, computed essentially through a query, into the Sailors relation as follows:

| Sailors | sid | sname | rating | age |
|---|---|---|---|---|
| `I.` | _Id | _N | | _A |

| Students | sid | name | login | age |
|---|---|---|---|---|
| | _Id | _N | | _A |

| Conditions |
|---|
| _A > 18 `OR` _N `LIKE` 'C%' |

We insert one tuple for each student older than 18 or with a name that begins with C. (QBE's `LIKE` operator is similar to the SQL version.) The *rating* field of every inserted tuple contains a *null* value. The following query is very similar to the previous query, but differs in a subtle way:

| Sailors | sid | sname | rating | age |
|---|---|---|---|---|
| `I.` | _Id1 | _N1 | | _A1 |
| `I.` | _Id2 | _N2 | | _A2 |

| Students | sid | name | login | age |
|---|---|---|---|---|
| | _Id1 | _N1 | | _A1 > 18 |
| | _Id2 | _N2 `LIKE` 'C%' | | _A2 |

The difference is that a student older than 18 with a name that begins with 'C' is now inserted *twice* into Sailors. (The second insertion will be rejected by the integrity constraint enforcement mechanism because *sid* is a key for Sailors. However, if this integrity constraint is not declared, we would find two copies of such a student in the Sailors relation.)

We can delete all tuples with *rating* $> 5$ from the Sailors relation as follows:

| Sailors | sid | sname | rating | age |
|---|---|---|---|---|
| `D.` | | | > 5 | |

We can delete all reservations for sailors with *rating* $< 4$ by using:

| *Sailors* | *sid* | *sname* | *rating* | *age* |
|---|---|---|---|---|
| | _Id | | < 4 | |

| *Reserves* | *sid* | *bid* | *day* |
|---|---|---|---|
| `D.` | _Id | | |

We can update the age of the sailor with *sid* 74 to be 42 years by using:

| *Sailors* | *sid* | *sname* | *rating* | *age* |
|---|---|---|---|---|
| | 74 | | | `U.`42 |

The fact that *sid* is the key is significant here; we cannot update the key field, but we can use it to identify the tuple to be modified (in other fields). We can also change the age of sailor 74 from 41 to 42 by incrementing the age value:

| *Sailors* | *sid* | *sname* | *rating* | *age* |
|---|---|---|---|---|
| | 74 | | | `U.`_A+1 |

### 6.8.1 Restrictions on Update Commands

There are some restrictions on the use of the `I.`, `D.`, and `U.` commands. First, we cannot mix these operators in a single example table (or combine them with `P.`). Second, we cannot specify `I.`, `D.`, or `U.` in an example table that contains `G.` Third, we cannot insert, update, or modify tuples based on values in fields of other tuples in the same table. Thus, the following update is incorrect:

| *Sailors* | *sid* | *sname* | *rating* | *age* |
|---|---|---|---|---|
| | | john | | `U.`_A+1 |
| | | joe | | _A |

This update seeks to change John's age based on Joe's age. Since *sname* is not a key, the meaning of such a query is ambiguous—should we update *every* John's age, and if so, based on *which* Joe's age? QBE avoids such anomalies using a rather broad restriction. For example, if *sname* were a key, this would be a reasonable request, even though it is disallowed.

## 6.9 DIVISION AND RELATIONAL COMPLETENESS *

In Section 6.6 we saw how division can be expressed in QBE using `COUNT.` It is instructive to consider how division can be expressed in QBE without the use of aggregate operators. If we don't use aggregate operators, we cannot express division in QBE without using the update commands to create a temporary relation or view. However, taking the update commands into account, QBE is relationally complete, even without the aggregate operators. Although we will not prove these claims, the example that we discuss below should bring out the underlying intuition.

We use the following query in our discussion of division:

*Find sailors who have reserved all boats.*

In Chapter 4 we saw that this query can be expressed in DRC as:

$$\{\langle I,N,T,A\rangle \mid \langle I,N,T,A\rangle \in Sailors\ \wedge \forall\langle B,BN,C\rangle \in Boats$$
$$(\exists\langle Ir,Br,D\rangle \in Reserves(I = Ir \wedge Br = B))\}$$

The $\forall$ quantifier is not available in QBE, so let us rewrite the above without $\forall$:

$$\{\langle I,N,T,A\rangle \mid \langle I,N,T,A\rangle \in Sailors \wedge \neg\exists\langle B,BN,C\rangle \in Boats$$
$$(\neg\exists\langle Ir,Br,D\rangle \in Reserves(I = Ir \wedge Br = B))\}$$

This calculus query can be read as follows: "Find Sailors tuples (with *sid* I) for which there is no Boats tuple (with *bid* B) such that no Reserves tuple indicates that sailor I has reserved boat B." We might try to write this query in QBE as follows:

| *Sailors* | *sid* | *sname* | *rating* | *age* |
|---|---|---|---|---|
| | _Id | P._S | | |

| *Boats* | *bid* | *bname* | *color* |
|---|---|---|---|
| ¬ | _B | | |

| *Reserves* | *sid* | *bid* | *day* |
|---|---|---|---|
| ¬ | _Id | _B | |

This query is illegal because the variable _$B$ does not appear in any positive row. Going beyond this technical objection, this QBE query is ambiguous with respect to the *ordering* of the two uses of ¬. It could denote either the calculus query that we want to express or the following calculus query, which is not what we want:

$$\{\langle I,N,T,A\rangle \mid \langle I,N,T,A\rangle \in Sailors \wedge \neg\exists\langle Ir,Br,D\rangle \in Reserves$$
$$(\neg\exists\langle B,BN,C\rangle \in Boats(I = Ir \wedge Br = B))\}$$

There is no mechanism in QBE to control the order in which the ¬ operations in a query are applied. (Incidentally, the above query finds all Sailors who have made reservations only for boats that exist in the Boats relation.)

One way to achieve such control is to break the query into several parts by using temporary relations or views. As we saw in Chapter 4, we can accomplish division in two logical steps: first, identify *disqualified* candidates, and then remove this set from the set of all candidates. In the query at hand, we have to first identify the set of *sid*s (called, say, BadSids) of sailors who have not reserved some boat (i.e., for each such sailor, we can find a boat not reserved by that sailor), and then we have to remove BadSids from the set of *sid*s of all sailors. This process will identify the set of sailors who've reserved all boats. The view BadSids can be defined as follows:

| *Sailors* | *sid* | *sname* | *rating* | *age* |
|---|---|---|---|---|
| | _Id | | | |

| *Reserves* | *sid* | *bid* | *day* |
|---|---|---|---|
| ¬ | _Id | _B | |

| *Boats* | *bid* | *bname* | *color* |
|---|---|---|---|
| | _B | | |

| *BadSids* | *sid* |
|---|---|
| I. | _Id |

Given the view BadSids, it is a simple matter to find sailors whose *sid*s are not in this view.

The ideas in this example can be extended to show that QBE is *relationally complete*.

## 6.10 POINTS TO REVIEW

- QBE is a user-friendly query language with a graphical interface. The interface depicts each relation in tabular form. **(Section 6.1)**
- Queries are posed by placing constants and variables into individual columns and thereby creating an example tuple of the query result. Simple conventions are used to express selections, projections, sorting, and duplicate elimination. **(Section 6.2)**
- Joins are accomplished in QBE by using the same variable in multiple locations. **(Section 6.3)**
- QBE provides a limited form of set difference through the use of ¬ in the relation-name column. **(Section 6.4)**
- Aggregation (`AVG.`, `COUNT.`, `MAX.`, `MIN.`, and `SUM.`) and grouping (`G.`) can be expressed by adding prefixes. **(Section 6.5)**
- The condition box provides a place for more complex query conditions, although queries involving `AND` or `OR` can be expressed without using the condition box. **(Section 6.6)**
- New, unnamed fields can be created to display information beyond fields retrieved from a relation. **(Section 6.7)**

- QBE provides support for insertion, deletion and updates of tuples. **(Section 6.8)**
- Using a temporary relation, division can be expressed in QBE without using aggregation. QBE is relationally complete, taking into account its querying and view creation features. **(Section 6.9)**

## EXERCISES

**Exercise 6.1** Consider the following relational schema. An employee can work in more than one department.

Emp(*eid:* `integer`, *ename:* `string`, *salary:* `real`)
Works(*eid:* `integer`, *did:* `integer`)
Dept(*did:* `integer`, *dname:* `string`, *managerid:* `integer`, *floornum:* `integer`)

Write the following queries in QBE. Be sure to underline your variables to distinguish them from your constants.

1. Print the names of all employees who work on the 10th floor and make less than $50,000.
2. Print the names of all managers who manage three or more departments on the same floor.
3. Print the names of all managers who manage 10 or more departments on the same floor.
4. Give every employee who works in the toy department a 10 percent raise.
5. Print the names of the departments that employee Santa works in.
6. Print the names and salaries of employees who work in both the toy department and the candy department.
7. Print the names of employees who earn a salary that is either less than $10,000 or more than $100,000.
8. Print all of the attributes for employees who work in some department that employee Santa also works in.
9. Fire Santa.
10. Print the names of employees who make more than $20,000 and work in either the video department or the toy department.
11. Print the names of all employees who work on the floor(s) where Jane Dodecahedron works.
12. Print the name of each employee who earns more than the manager of the department that he or she works in.
13. Print the name of each department that has a manager whose last name is Psmith and who is neither the highest-paid nor the lowest-paid employee in the department.

**Exercise 6.2** Write the following queries in QBE, based on this schema:

Suppliers(*sid:* `integer`, *sname:* `string`, *city:* `string`)
Parts(*pid:* `integer`, *pname:* `string`, *color:* `string`)
Orders(*sid:* `integer`, *pid:* `integer`, *quantity:* `integer`)

1. For each supplier from whom all of the following things have been ordered in quantities of at least 150, print the name and city of the supplier: a blue gear, a red crankshaft, and a yellow bumper.
2. Print the names of the purple parts that have been ordered from suppliers located in Madison, Milwaukee, or Waukesha.
3. Print the names and cities of suppliers who have an order for more than 150 units of a yellow or purple part.
4. Print the *pid*s of parts that have been ordered from a supplier named American but have also been ordered from some supplier with a different name in a quantity that is greater than the American order by at least 100 units.
5. Print the names of the suppliers located in Madison. Could there be any duplicates in the answer?
6. Print all available information about suppliers that supply green parts.
7. For each order of a red part, print the quantity and the name of the part.
8. Print the names of the parts that come in both blue and green. (Assume that no two distinct parts can have the same name and color.)
9. Print (in ascending order alphabetically) the names of parts supplied both by a Madison supplier and by a Berkeley supplier.
10. Print the names of parts supplied by a Madison supplier, but not supplied by any Berkeley supplier. Could there be any duplicates in the answer?
11. Print the total number of orders.
12. Print the largest quantity per order for each *sid* such that the minimum quantity per order for that supplier is greater than 100.
13. Print the average quantity per order of red parts.
14. Can you write this query in QBE? If so, how?
    *Print the* sid*s of suppliers from whom every part has been ordered.*

**Exercise 6.3** Answer the following questions:

1. Describe the various uses for unnamed columns in QBE.
2. Describe the various uses for a conditions box in QBE.
3. What is unusual about the treatment of duplicates in QBE?
4. Is QBE based upon relational algebra, tuple relational calculus, or domain relational calculus? Explain briefly.
5. Is QBE relationally complete? Explain briefly.
6. What restrictions does QBE place on update commands?

## PROJECT-BASED EXERCISES

**Exercise 6.4** Minibase's version of QBE, called MiniQBE, tries to preserve the spirit of QBE but cheats occasionally. Try the queries shown in this chapter and in the exercises, and identify the ways in which MiniQBE differs from QBE. For each QBE query you try in MiniQBE, examine the SQL query that it is translated into by MiniQBE.

## BIBLIOGRAPHIC NOTES

The QBE project was led by Moshe Zloof [702] and resulted in the first visual database query language, whose influence is seen today in products such as Borland's Paradox and, to a lesser extent, Microsoft's Access. QBE was also one of the first relational query languages to support the computation of transitive closure, through a special operator, anticipating much subsequent research into extensions of relational query languages to support recursive queries. A successor called Office-by-Example [701] sought to extend the QBE visual interaction paradigm to applications such as electronic mail integrated with database access. Klug presented a version of QBE that dealt with aggregate queries in [377].