

MODULE-3

Relational model concepts

- The relational model represents the database as a collection of *relations*.
- Each relation resembles a table of values or, to some extent, a *flat file of records*. It is called a **flat file** because each record has a simple linear or *flat structure*.
- When a relation is thought of as a **table of values**, each row in the table represents a collection of related data values.

- A row represents a fact that typically corresponds to a real-world entity or relationship.
- The table name and column names are used to help to interpret the meaning of the values in each row.

STUDENT

Name	Student_number	Class	Major
Smith	17	1	CS
Brown	8	2	CS

COURSE

Course_name	Course_number	Credit_hours	Department
Intro to Computer Science	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Database	CS3380	3	CS

SECTION

Section_identifier	Course_number	Semester	Year	Instructor
85	MATH2410	Fall	07	King
92	CS1310	Fall	07	Anderson
102	CS3320	Spring	08	Knuth
112	MATH2410	Fall	08	Chang
119	CS1310	Fall	08	Anderson
135	CS3380	Fall	08	Stone

GRADE REPORT

Student_number	Section_identifier	Grade
17	112	B
17	119	C
8	85	A
8	92	A
8	102	B
8	135	A

PREREQUISITE

Course_number	Prerequisite_number
CS3380	CS3320
CS3380	MATH2410
CS3320	CS1310

- For example, the first table of Figure is called STUDENT because each row represents facts about a particular student entity.
- The column names—Name, Student_number, Class, and Major—specify how to interpret the data values in each row, based on the column each value is in.
- All values in a column are of the same data type.
- In the formal relational model terminology, **a row is called a tuple**,
- **a column header** is called an **attribute**, and the **table** is called **a relation**.
- *The data type describing the types of values that can appear in each column is represented by a **domain** of possible values. Domain is given name, data type, and format*

- **Domains, Attributes, Tuples, and Relations**
- A domain ***D*** is a set of atomic values.
- *By atomic we mean that each value in the domain is indivisible as far as the formal relational model is concerned.*
- Domain is given a name, data type, and format.
- Some examples of domains follow:
- ■ Usa_phone_numbers. The set of ten-digit phone numbers valid in the United States.
- ■ Local_phone_numbers. The set of seven-digit phone numbers valid within a particular area code in the United States. The use of local phone numbers is quickly becoming obsolete, being replaced by standard ten-digit numbers.

- `Social_security_numbers`. The set of valid nine-digit Social Security numbers.(This is a unique identifier assigned to each person in the United States for employment, tax, and benefits purposes.)
- ■ `Names`: The set of character strings that represent names of persons.
- ■ `Grade_point_averages`. Possible values of computed grade point averages;each must be a real (floating-point) number between 0 and 4.
- ■ `Employee_ages`. Possible ages of employees in a company; each must be an integer value between 15 and 80.
- ■ `Academic_department_names`. The set of academic department names in a university, such as Computer Science, Economics, and Physics.
- ■ `Academic_department_codes`. The set of academic department codes, such as 'CS', 'ECON', and 'PHYS'

- A relation schema R , denoted by $R(A_1, A_2, \dots, A_n)$, is made up of a relation name R and a list of attributes, A_1, A_2, \dots, A_n .
- Each attribute A_i is the name of a role played by some domain D in the relation schema R .
- D is called the domain of A_i and is denoted by $\text{dom}(A_i)$.
- A relation schema is used to describe a relation; R is called the name of this relation.
- The degree of a relation is the number of attributes n of its relation schema.
- Ex: $\text{STUDENT}(\text{Name}, \text{Ssn}, \text{Home_phone}, \text{Address}, \text{Office_phone}, \text{Age}, \text{Gpa})$
- $\text{STUDENT}(\text{Name: string}, \text{Ssn: string}, \text{Home_phone: string}, \text{Address: string}, \text{Office_phone: string}, \text{Age: integer}, \text{Gpa: real})$

- A relation (or relation state) r of the relation schema $R(A_1, A_2, \dots, A_n)$, also denoted by $r(R)$, is a set of n -tuples $r = \{t_1, t_2, \dots, t_m\}$.
- Each n -tuple t is an ordered list of n values $t = \langle v_1, v_2, \dots, v_n \rangle$, where each value v_i , $1 \leq i \leq n$, is an element of $\text{dom}(A_i)$ or is a special NULL value.
- The i th value in tuple t , which corresponds to the attribute A_i , is referred to as $t[A_i]$ or $t.A_i$ (or $t[i]$ if we use the positional notation).
- The terms relation intension for the schema R and relation extension for a relation state $r(R)$ are also commonly used.
- NULL values represent attributes whose values are unknown or do not exist.

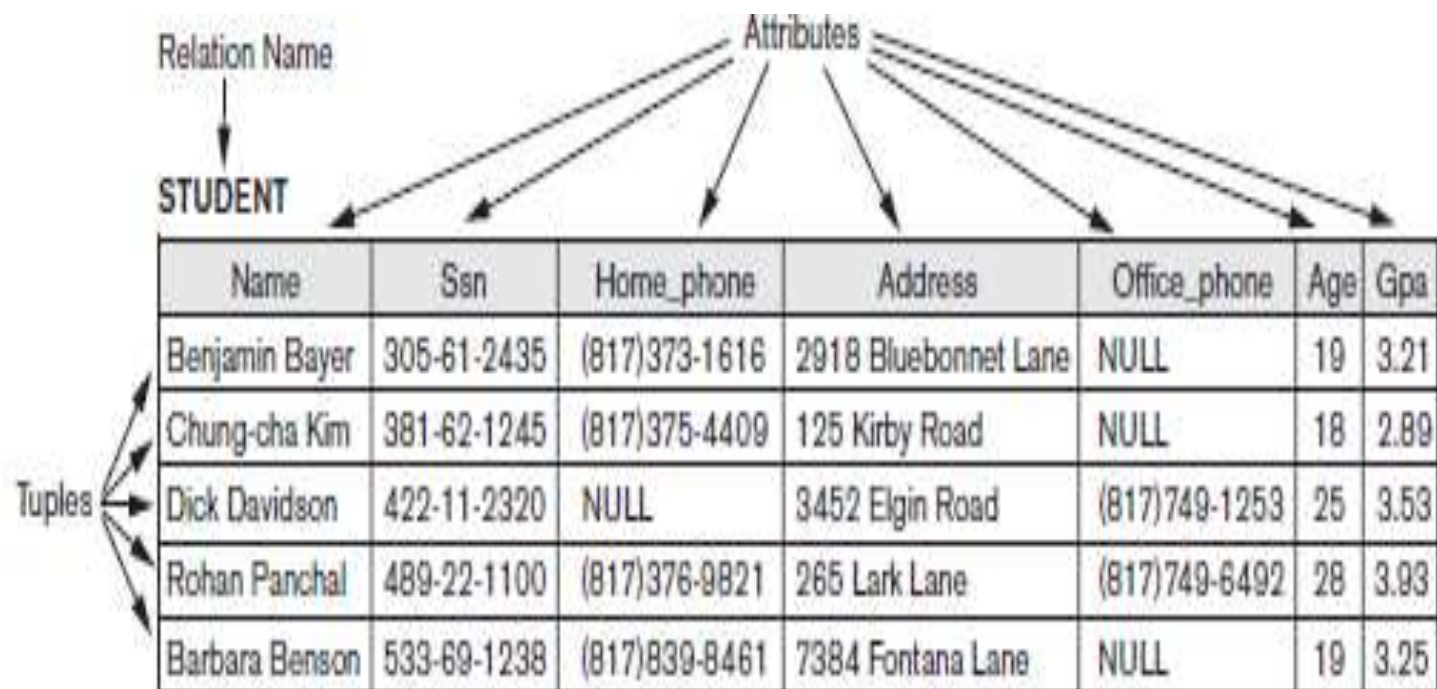


Figure 3.1

The attributes and tuples of a relation **STUDENT**.

Characteristics of Relations

- 1. Ordering of Tuples in a Relation.** A relation is defined as a *set of tuples*. Mathematically, elements of a set have *no order among them; hence, tuples in a relation* do not have any particular order. when we display a relation as a table, the rows are displayed in a certain order. There is *no preference for one ordering over another*.

Figure 3.2

The relation STUDENT from Figure 3.1 with a different order of tuples.

STUDENT

Name	Ssn	Home_phone	Address	Office_phone	Age	Gpa
Dick Davidson	422-11-2320	NULL	3452 Elgin Road	(817)749-1253	25	3.53
Barbara Benson	533-69-1238	(817)839-8461	7384 Fontana Lane	NULL	19	3.25
Rohan Panchal	489-22-1100	(817)376-9821	265 Lark Lane	(817)749-6492	28	3.93
Chung-cha Kim	381-62-1245	(817)375-4409	125 Kirby Road	NULL	18	2.89
Benjamin Bayer	305-61-2435	(817)373-1616	2918 Bluebonnet Lane	NULL	19	3.21

2. Values and NULLs in the Tuples: Each value in a tuple is an **atomic value**; that is, it is not divisible into components within the framework of the basic relational model.

- Much of the theory behind the relational model was developed with this assumption in mind, which is called the **first normal form assumption**.
- In general, we can have several meanings for NULL values, such as *value unknown*, *value exists but is not available*, or *attribute does not apply to this tuple (also known as value undefined)*.
- a comparison of two NULL values leads to ambiguities

3. Interpretation (Meaning) of a Relation: Each tuple in the relation can then be interpreted as a fact or a particular instance of the assertion.

For example,

- the first tuple in Figure 3.1 asserts the fact that there is a STUDENT whose Name is Benjamin Bayer, Ssn is 305-61-2435, Age is 19, and so on

RELATIONAL MODEL CONSTRAINTS

- There are generally many restrictions or constraints on the actual values in a database state.
- Constraints on databases can generally be divided into three main categories:
 1. Constraints that are inherent in the data model. We call these inherent model-based constraints or implicit constraints.(For example, the constraint that a relation cannot have duplicate tuples is an inherent constraint)
 2. Constraints that can be directly expressed in schemas of the data model, typically by specifying them in the DDL.We call these schema-based constraints or explicit constraints.

3. Constraints that cannot be directly expressed in the schemas of the data model, and hence must be expressed and enforced by the application programs. We call these application-based or semantic constraints or business rules.
- The schema-based constraints include **domain constraints, key constraints, constraints on NULLs, entity integrity constraints, and referential integrity constraints.**
 - **Domain Constraints:** Domain constraints specify that within each tuple, the value of each attribute A must be an atomic value from the domain $\text{dom}(A)$.

- The data types associated with domains typically include standard numeric data types for integers (such as short integer, integer, and long integer) and real numbers (float and double precision float). Characters, Booleans, fixed-length strings, and variable-length
- strings are also available, as are date, time, timestamp, and money, or other special data types.
- **Key Constraints and Constraints on NULL Values:**all tuples in a relation must also be distinct. This means that no two tuples can have the same combination of values for *all their* attributes.

- Suppose that we denote one such subset of attributes by SK; then for any two *distinct tuples* $t1$ and $t2$ in a relation state r of R ,
- we have the constraint that: $t1[SK] \neq t2[SK]$
- A superkey can have redundant attributes, however, so a more useful concept is that of a *key*, which has no redundancy.
- Hence, a key satisfies two properties:
- **1.** Two distinct tuples in any state of the relation cannot have identical values for (all) the attributes in the key. This first property also applies to a superkey.
- **2.** It is a minimal superkey—that is, a superkey from which we cannot remove any attributes and still have the uniqueness constraint in condition 1 hold.
- This property is not required by a superkey.

CAR

<u>License_number</u>	Engine_serial_number	Make	Model	Year
Texas ABC-739	A69352	Ford	Mustang	02
Florida TVP-347	B43696	Oldsmobile	Cutlass	05
New York MPO-22	X83554	Oldsmobile	Delta	01
California 432-TFY	C43742	Mercedes	190-D	99
California RSK-629	Y82935	Toyota	Camry	04
Texas RSK-629	U028365	Jaguar	XJS	04

Figure 3.4

The CAR relation, with two candidate keys: License_number and Engine_serial_number.

- A relation schema may have more than one key. In this case, each of the keys is called a **candidate key**.
- Another constraint on attributes specifies whether NULL values are or are not permitted.
- For example, if every STUDENT tuple must have a valid, non-NULL value for the Name attribute, then Name of STUDENT is constrained to be NOT NULL.
- **Relational Databases and Relational Database Schemas:** A relational database schema S is a set of relation schemas $S = \{R_1, R_2, \dots, R_m\}$ and a set of integrity constraints IC.

- A relational database state DB of S is a set of relation states $DB = \{r_1, r_2, \dots, r_m\}$ such that each r_i is a state of R_i and such that the r_i relation states satisfy the integrity constraints specified in IC .
- A database state that does not obey all the integrity constraints is called an **invalid state**, and a state that satisfies all the constraints in the defined set of integrity constraints IC is called a **valid state**.

EMPLOYEE

Fname	Minit	Lname	<u>Ssn</u>	Bdate	Address	Sex	Salary	Super_ssn	Dno
-------	-------	-------	------------	-------	---------	-----	--------	-----------	-----

DEPARTMENT

Dname	<u>Dnumber</u>	Mgr_ssn	Mgr_start_date
-------	----------------	---------	----------------

DEPT_LOCATIONS

<u>Dnumber</u>	<u>Dlocation</u>
----------------	------------------

PROJECT

Pname	<u>Pnumber</u>	Plocation	Dnum
-------	----------------	-----------	------

WORKS_ON

<u>Essn</u>	<u>Pno</u>	Hours
-------------	------------	-------

DEPENDENT

<u>Essn</u>	<u>Dependent_name</u>	Sex	Bdate	Relationship
-------------	-----------------------	-----	-------	--------------

Figure 3.5

Schema diagram for the COMPANY relational database schema.

Figure 3.6

One possible database state for the COMPANY relational database schema.

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	NULL

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship
333445555	Alice	F	1986-04-05	Daughter
333445555	Theodore	M	1983-10-25	Son
333445555	Joy	F	1958-05-03	Spouse
987654321	Abner	M	1942-02-28	Spouse
123456789	Michael	M	1988-01-04	Son
123456789	Alice	F	1988-12-30	Daughter
123456789	Elizabeth	F	1967-05-05	Spouse

- **Entity Integrity, Referential Integrity, and Foreign Keys:**
- The **entity integrity constraint states** that no primary key value can be NULL. This is because the primary key value is used to identify individual tuples in a relation.
- Key constraints and entity integrity constraints are specified on individual relations.
- The **referential integrity constraint** is specified between two relations and is used to maintain the consistency among tuples in the two relations.

- To define referential integrity more formally, first we define the concept of a *foreign key*.
- *The conditions for a foreign key, given below, specify a referential integrity constraint between the two relation schemas $R1$ and $R2$.*
- *A set of attributes FK in relation schema $R1$ is a **foreign key of $R1$ that references relation $R2$ if it satisfies the following rules:***
 - 1. The attributes in FK have the same domain(s) as the primary key attributes PK of $R2$; the attributes FK are said to *reference or refer to the relation $R2$.***
 - 2. A value of FK in a tuple $t1$ of the current state $r1(R1)$ either occurs as a value of PK for some tuple $t2$ in the current state $r2(R2)$ or is $NULL$. In the former case, we have $t1[FK] = t2[PK]$, and we say that the tuple $t1$ *references or refers to the tuple $t2$.***

EMPLOYEE

Fname	Minit	Lname	<u>Ssn</u>	Bdate	Address	Sex	Salary	Super_ssn	Dno
-------	-------	-------	------------	-------	---------	-----	--------	-----------	-----

DEPARTMENT

Dname	<u>Dnumber</u>	Mgr_ssn	Mgr_start_date
-------	----------------	---------	----------------

DEPT_LOCATIONS

<u>Dnumber</u>	<u>Dlocation</u>
----------------	------------------

PROJECT

Pname	<u>Pnumber</u>	Plocation	Dnum
-------	----------------	-----------	------

WORKS_ON

<u>Essn</u>	<u>Pno</u>	Hours
-------------	------------	-------

DEPENDENT

<u>Essn</u>	<u>Dependent_name</u>	Sex	Bdate	Relationship
-------------	-----------------------	-----	-------	--------------

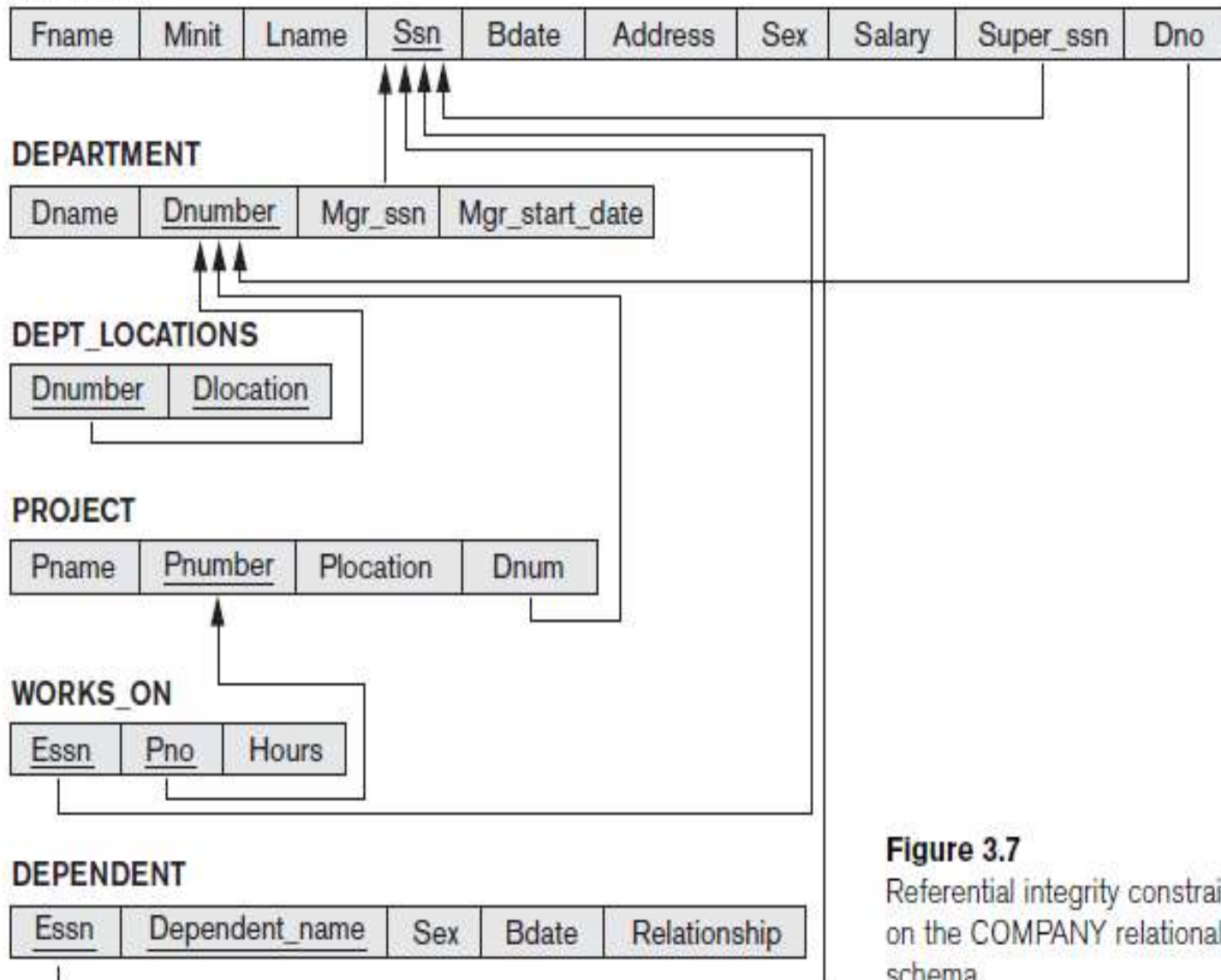


Figure 3.7

Referential integrity constraints displayed on the COMPANY relational database schema.

SQL(STRUCTURED QUERY LANGUAGE)

- higher-level *declarative language* interface, so the user only specifies what the result is to be, leaving the actual optimization and decisions on how to execute the query to the DBMS
- it is both a DDL *and a DML*
- Non procedural language(what data to retrieve or modify without telling it how to do its job)
- SQL does not provide any flow of control constructs,do-loops,if -then-else stataments
- SQL provide fixed set of datatypes

- DDL-allow you to create,alter,drop schema objects.grant and revoke user privileges and roles.ex:create
- DML-Manipulate and query data in the data base.SELECT,INSERT,DELETE,UPDATE
- SELECT-Used to retrieve rows of data from table.default-retrieve all rows.
- INSERT-insert new rows to table
- Delete-delete row from table
- UPDATE-change existing data values

The CREATE TABLE Command in SQL

- The **CREATE TABLE** command is used to **specify a new relation by giving it a name** and specifying its attributes and initial constraints.
- The attributes are specified first, and each attribute is given a name, a data type to specify its domain of values, and any attribute constraints, such as NOT NULL.

- SYNTAX:
- CREATE TABLE TABLENAME(Attribute1 datatype1 [NOT NULL],..... AttributeN datatype);

EX: CREATE TABLE EMPLOYEE(NAME VARCHAR(15) NOT NULL,ROLLNO INT PRIMARY KEY);

- The relations declared through CREATE TABLE statements are called **base tables** (or base relations); this means that the relation and its tuples are actually created and stored as a file by the DBMS.
- In SQL, the attributes in a base table are considered to be *ordered in the sequence in which they are specified in the CREATE TABLE statement*.
- However, rows (tuples) are not considered to be ordered within a relation.

```

CREATE TABLE EMPLOYEE
( Fname          VARCHAR(15)          NOT NULL,
  Minit          CHAR,
  Lname          VARCHAR(15)          NOT NULL,
  Ssn            CHAR(9)              NOT NULL,
  Bdate          DATE,
  Address        VARCHAR(30),
  Sex            CHAR,
  Salary         DECIMAL(10,2),
  Super_ssn      CHAR(9),
  Dno            INT                  NOT NULL,
  PRIMARY KEY (Ssn),
  FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(Ssn),
  FOREIGN KEY (Dno) REFERENCES DEPARTMENT(Dnumber) );

CREATE TABLE DEPARTMENT
( Dname          VARCHAR(15)          NOT NULL,
  Dnumber        INT                  NOT NULL,
  Mgr_ssn        CHAR(9)              NOT NULL,
  Mgr_start_date DATE,
  PRIMARY KEY (Dnumber),
  UNIQUE (Dname),
  FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn) );

CREATE TABLE DEPT_LOCATIONS
( Dnumber        INT                  NOT NULL,
  Dlocation      VARCHAR(15)          NOT NULL,
  PRIMARY KEY (Dnumber, Dlocation),
  FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber) );

CREATE TABLE PROJECT
( Pname          VARCHAR(15)          NOT NULL,
  Pnumber        INT                  NOT NULL,
  Plocation      VARCHAR(15),
  Dnum           INT                  NOT NULL,
  PRIMARY KEY (Pnumber),
  UNIQUE (Pname),
  FOREIGN KEY (Dnum) REFERENCES DEPARTMENT(Dnumber) );

CREATE TABLE WORKS_ON
( Essn           CHAR(9)              NOT NULL,
  Pno            INT                  NOT NULL,
  Hours          DECIMAL(3,1)         NOT NULL,
  PRIMARY KEY (Essn, Pno),
  FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn),
  FOREIGN KEY (Pno) REFERENCES PROJECT(Pnumber) );

CREATE TABLE DEPENDENT
( Essn           CHAR(9)              NOT NULL,
  Dependent_name VARCHAR(15)          NOT NULL,
  Sex            CHAR,
  Bdate          DATE,
  Relationship    VARCHAR(8),
  PRIMARY KEY (Essn, Dependent_name),
  FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn) );

```

Figure 4.1
SQL CREATE TABLE
data definition state-
ments for defining the
COMPANY schema
from Figure 3.7.

ALTER TABLE AND DROP

- Change the structure of a table not its content
- Add new column
- Increase or decrease the width of an existing column
- Specify default value
- SYNTAX:
- ALTER TABLE table_name ADD(columnspecification|table_constraint,.....);
- ALTER TABLE table_name
MODIFY(columnspecification|table_constraint,.....);
- ALTER TABLE table_name DROP PRIMARY KEY

- ALTER TABLE table_name ADD(PRIMARY KEY(ROLLNO));
- EX:ALTER TABLE emp MODIFY(fname varchar(60));
- ALTER TABLE table_name DROP CONSTRAINT Constraint_name;

DATA TYPES

- The basic data types available for attributes include numeric, character string, bit string, Boolean, date, and time.
- **1.Numeric data types:** include integer numbers of various sizes (INTEGER or INT, and SMALLINT) and floating-point (real) numbers of various precision (FLOAT or REAL, and DOUBLE PRECISION).
- Formatted numbers can be declared by using DECIMAL(i,j)—or DEC(i,j) or NUMERIC(i,j)—where i, the, is the total number of decimal digits and j, the scale, is the number of digits after the decimal point.
- The default for scale is zero, and the default for precision is implementation-defined.

2.Character-string:

- fixed length— $\text{CHAR}(n)$ or $\text{CHARACTER}(n)$, where n is the number of characters
- varying length— $\text{VARCHAR}(n)$ or $\text{CHAR VARYING}(n)$ or $\text{CHARACTER VARYING}(n)$, where n is the maximum number of characters

3. Bit-string data types: are either of fixed length n — $\text{BIT}(n)$

- **Varying** length— $\text{BIT VARYING}(n)$, where n is the maximum number of bits.
- The default for n , the length of a character string or bit string, is 1.

4. A **Boolean data type**: has the traditional values of TRUE or FALSE. In SQL, because of the presence of NULL values, a three-valued logic is used UNKNOWN.
5. The **DATE data type** has ten positions, and its components are YEAR, MONTH, and DAY in the form YYYY-MM-DD.
- The TIME data type has at least eight positions, with the components HOUR, MINUTE, and SECOND in the form HH:MM:SS.

Basic Retrieval Queries in SQL

- 1. The **SELECT-FROM-WHERE** Structure of Basic SQL Queries:
- **SELECT** <attribute list>
- **FROM** <table list>
- **WHERE** <condition>;

where

- ■ <attribute list> is a list of attribute names whose values are to be retrieved by the query.
- ■ <table list> is a list of the relation names required to process the query.
- ■ <condition> is a conditional (Boolean) expression that identifies the tuples to be retrieved by the query.

Simple SQL Queries

- Basic SQL queries correspond to using the SELECT, PROJECT, and JOIN operations of the relational algebra
- All subsequent examples use the COMPANY database
- Example of a simple query on *one* relation
- Query 0: Retrieve the birthdate and address of the employee whose name is 'John B. Smith'.

```
Q0: SELECT  BDATE, ADDRESS  
      FROM    EMPLOYEE  
      WHERE   FNAME='John' AND MINIT='B'  
      AND     LNAME='Smith'
```

- Similar to a SELECT-PROJECT pair of relational algebra operations; the SELECT-clause specifies the *projection attributes* and the WHERE-clause specifies the *selection condition*
- However, the result of the query *may contain* duplicate tuples

Simple SQL Queries (cont.)

- Query 1: Retrieve the name and address of all employees who work for the 'Research' department.

**Q1: SELECT FNAME, LNAME, ADDRESS
FROM EMPLOYEE, DEPARTMENT
WHERE DNAME='Research' AND DNUMBER=DNO**

- Similar to a SELECT-PROJECT-JOIN sequence of relational algebra operations
- (DNAME='Research') is a *selection condition* (corresponds to a SELECT operation in relational algebra)
- (DNUMBER=DNO) is a *join condition* (corresponds to a JOIN operation in relational algebra)

Simple SQL Queries (cont.)

- Query 2: For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birthdate.

**Q2: SELECT PNUMBER, DNUM, LNAME, BDATE, ADDRESS
FROM PROJECT, DEPARTMENT, EMPLOYEE
WHERE DNUM=DNUMBER AND MGRSSN=SSN AND
PLOCATION='Stafford'**

- In Q2, there are *two* join conditions
- The join condition DNUM=DNUMBER relates a project to its controlling department
- The join condition MGRSSN=SSN relates the controlling department to the employee who manages that department

Aliases, * and DISTINCT, Empty WHERE-clause

- In SQL, we can use the same name for two (or more) attributes as long as the attributes are in *different relations*

A query that refers to two or more attributes with the same name must *qualify* the attribute name with the relation name by *prefixing* the relation name to the attribute name

Example:

- EMPLOYEE.LNAME, DEPARTMENT.DNAME

ALIASES

- Some queries need to refer to the same relation twice
- In this case, *aliases* are given to the relation name
- Query 8: For each employee, retrieve the employee's name, and the name of his or her immediate supervisor.

**Q8: SELECT E.FNAME, E.LNAME, S.FNAME, S.LNAME
 FROM EMPLOYEE E S
 WHERE E.SUPERSSN=S.SSN**

- In Q8, the alternate relation names E and S are called *aliases* or *tuple variables* for the EMPLOYEE relation
- We can think of E and S as two *different copies* of EMPLOYEE; E represents employees in role of *supervisees* and S represents employees in role of *supervisors*

ALIASES (cont.)

- Aliasing can also be used in any SQL query for convenience
Can also use the AS keyword to specify aliases

Q8: SELECT E.FNAME, E.LNAME, S.FNAME, S.LNAME

FROM EMPLOYEE AS E, EMPLOYEE AS S

WHERE E.SUPERSSN=S.SSN

UNSPECIFIED WHERE-clause

- A *missing WHERE-clause* indicates no condition; hence, *all tuples* of the relations in the FROM-clause are selected
- This is equivalent to the condition WHERE TRUE
- Query 9: Retrieve the SSN values for all employees.

**Q9:SELECT SSN
 FROM EMPLOYEE**

- If more than one relation is specified in the FROM-clause *and* there is no join condition, then the *CARTESIAN PRODUCT* of tuples is selected

UNSPECIFIED WHERE-clause (cont.)

- Example:

**Q10: SELECT SSN, DNAME
 FROM EMPLOYEE, DEPARTMENT**

- It is extremely important not to overlook specifying any selection and join conditions in the WHERE-clause; otherwise, incorrect and very large relations may result

USE OF *

- To retrieve all the attribute values of the selected tuples, a * is used, which stands for *all the attributes*

Examples:

Q1C: **SELECT ***
 FROM EMPLOYEE
 WHERE DNO=5

Q1D: **SELECT ***
 FROM EMPLOYEE, DEPARTMENT
 WHERE DNAME='Research' AND
 DNO=DNUMBER

USE OF DISTINCT

- SQL does not treat a relation as a set; *duplicate tuples can appear*
- To eliminate duplicate tuples in a query result, the keyword **DISTINCT** is used
- For example, the result of Q11 may have duplicate SALARY values whereas Q11A does not have any duplicate values

Q11: **SELECT SALARY**
 FROM EMPLOYEE

Q11A: **SELECT DISTINCT SALARY**
 FROM EMPLOYEE

Figure 4.4

Results of additional SQL queries when applied to the COMPANY database state shown in Figure 3.6.

(a) Q11. (b) Q11A.

(c) Q16. (d) Q18.

(a)

Salary
30000
40000
25000
43000
38000
25000
25000
55000

(b)

Salary
30000
40000
25000
43000
38000
55000

SET OPERATIONS

- SQL has directly incorporated some set operations
- There is a union operation (**UNION**), and in *some versions* of SQL there are set difference (**MINUS**) and intersection (**INTERSECT**) operations
- The resulting relations of these set operations are sets of tuples; *duplicate tuples are eliminated from the result*
- The set operations apply only to *union compatible relations* ; the two relations must have the same attributes and the attributes must appear in the same order

SET OPERATIONS (cont.)

- Query 4: Make a list of all project numbers for projects that involve an employee whose last name is 'Smith' as a worker or as a manager of the department that controls the project.

**Q4: (SELECT PNAME
FROM PROJECT, DEPARTMENT, EMPLOYEE
WHERE DNUM=DNUMBER AND MGRSSN=SSN AND
LNAME='Smith')
UNION (SELECT PNAME
FROM PROJECT, WORKS_ON, EMPLOYEE
WHERE PNUMBER=PNO AND ESSN=SSN AND
LNAME='Smith')**

NESTING OF QUERIES

- A complete SELECT query, called a *nested query* , can be specified within the WHERE-clause of another query, called the *outer query*
- Many of the previous queries can be specified in an alternative form using nesting
- Query 1: Retrieve the name and address of all employees who work for the 'Research' department.

```
Q1:  SELECT  FNAME, LNAME, ADDRESS  
      FROM      EMPLOYEE  
      WHERE DNO IN (SELECT DNUMBER  
                      FROM      DEPARTMENT  
                      WHERE DNAME='Research' )
```

NESTING OF QUERIES (cont.)

- The nested query selects the number of the 'Research' department
- The outer query select an EMPLOYEE tuple if its DNO value is in the result of either nested query
- The comparison operator **IN** compares a value *v* with a set (or multi-set) of values *V*, and evaluates to **TRUE** if *v* is one of the elements in *V*
- In general, we can have several levels of nested queries
- A reference to an *unqualified attribute* refers to the relation declared in the *innermost nested query*
- In this example, the nested query is *not correlated* with the outer query

CORRELATED NESTED QUERIES

- If a condition in the WHERE-clause of a *nested query* references an attribute of a relation declared in the *outer query* , the two queries are said to be *correlated*
- The result of a correlated nested query is *different for each tuple (or combination of tuples) of the relation(s) the outer query*
- Query 12: Retrieve the name of each employee who has a dependent with the same first name as the employee.

```
Q12: SELECT  E.FNAME, E.LNAME
        FROM      EMPLOYEE AS E
        WHERE  E.SSN IN (SELECT ESSN
                        FROM  DEPENDENT
                        WHERE  ESSN=E.SSN AND
                                E.FNAME=DEPENDENT_NAME)
```

CORRELATED NESTED QUERIES (cont.)

- In Q12, the nested query has a different result *for each tuple* in the outer query
- A query written with nested SELECT... FROM... WHERE... blocks and using the = or IN comparison operators can ***always*** be expressed as a single block query. For example, Q12 may be written as in Q12A

Q12A: **SELECT E.FNAME, E.LNAME**
 FROM EMPLOYEE E, DEPENDENT D
 WHERE E.SSN=D.ESSN AND
 E.FNAME=D.DEPENDENT_NAME

- The original SQL as specified for SYSTEM R also had a **CONTAINS** comparison operator, which is used in conjunction with nested correlated queries
- This operator was dropped from the language, possibly because of the difficulty in implementing it efficiently

CORRELATED NESTED QUERIES (cont.)

- Most implementations of SQL *do not* have this operator
- The CONTAINS operator compares two *sets of values* , and returns TRUE if one set contains all values in the other set (reminiscent of the *division* operation of algebra).
 - Query 3: Retrieve the name of each employee who works on *all* the projects controlled by department number 5.

```
Q3:  SELECT  FNAME, LNAME
      FROM    EMPLOYEE
      WHERE   ( (SELECT  PNO
                  FROM    WORKS_ON
                  WHERE   SSN=ESSN)
                CONTAINS
                (SELECT  PNUMBER
                  FROM    PROJECT
                  WHERE   DNUM=5) )
```


CORRELATED NESTED QUERIES (cont.)

- In Q3, the second nested query, which is not correlated with the outer query, retrieves the project numbers of all projects controlled by department 5
- The first nested query, which is correlated, retrieves the project numbers on which the employee works, which is different *for each employee tuple* because of the correlation

THE EXISTS FUNCTION

- EXISTS is used to check whether the result of a correlated nested query is empty (contains no tuples) or not
- We can formulate Query 12 in an alternative form that uses EXISTS as Q12B below

THE EXISTS FUNCTION (cont.)

- Query 12: Retrieve the name of each employee who has a dependent with the same first name as the employee.

```
Q12B:      SELECT      FNAME, LNAME  
            FROM EMPLOYEE  
            WHERE      EXISTS      (SELECT      *  
                                     FROM          DEPENDENT  
                                     WHERE          SSN=ESSN AND  
                                     FNAME=DEPENDENT_NAME)
```

THE EXISTS FUNCTION (cont.)

- Query 6: Retrieve the names of employees who have no dependents.

Q6:

```
SELECT      FNAME, LNAME
FROM EMPLOYEE
WHERE       NOT EXISTS (SELECT *
                        FROM DEPENDENT
                        WHERE SSN=ESSN)
```

- In Q6, the correlated nested query retrieves all DEPENDENT tuples related to an EMPLOYEE tuple. If *none exist*, the EMPLOYEE tuple is selected
- EXISTS is necessary for the expressive power of SQL

EXPLICIT SETS

- It is also possible to use an **explicit (enumerated) set of values** in the WHERE-clause rather than a nested query
- Query 13: Retrieve the social security numbers of all employees who work on project number 1, 2, or 3.

```
Q13:      SELECT      DISTINCT ESSN
           FROM WORKS_ON
           WHERE        PNO IN (1, 2, 3)
```

NULLS IN SQL QUERIES

- SQL allows queries that check if a value is NULL (missing or undefined or not applicable)
- SQL uses **IS** or **IS NOT** to compare NULLs because it considers each NULL value distinct from other NULL values, so equality comparison is not appropriate .
- Query 14: Retrieve the names of all employees who do not have supervisors.

**Q14: SELECT FNAME, LNAME
 FROM EMPLOYEE
 WHERE SUPERSSN IS NULL**

Note: If a join condition is specified, tuples with NULL values for the join attributes are not included in the result

BETWEEN

- Query 14. Retrieve all employees in department 5 whose salary is between \$30,000 and \$40,000.
- Q14: **SELECT * FROM EMPLOYEE WHERE (Salary BETWEEN 30000 AND 40000) AND Dno = 5; OR**
- **SELECT * FROM EMPLOYEE WHERE((Salary >= 30000) AND (Salary <= 40000)) AND Dno = 5;**

AGGREGATE FUNCTIONS

- Include **COUNT**, **SUM**, **MAX**, **MIN**, and **AVG**
- Query 15: Find the maximum salary, the minimum salary, and the average salary among all employees.

Q15: **SELECT** **MAX(SALARY),**
 MIN(SALARY), AVG(SALARY)
 FROM EMPLOYEE

- Some SQL implementations *may not allow more than one function* in the SELECT-clause

AGGREGATE FUNCTIONS (cont.)

- Query 16: Find the maximum salary, the minimum salary, and the average salary among employees who work for the 'Research' department.

```
Q16: SELECT      MAX(SALARY), MIN(SALARY),  
                  AVG(SALARY)  
FROM EMPLOYEE, DEPARTMENT  
WHERE            DNO=DNUMBER AND  
                  DNAME='Research'
```

AGGREGATE FUNCTIONS (cont.)

- Queries 17 and 18: Retrieve the total number of employees in the company (Q17), and the number of employees in the 'Research' department (Q18).

Q17: **SELECT COUNT (*)**
 FROM EMPLOYEE

Q18: **SELECT COUNT (*)**
 FROM EMPLOYEE,
 DEPARTMENT
 WHERE DNO=DNUMBER AND
 DNAME='Research'

ORDER BY

- SQL allows the user to order the tuples in the result of a query by the values of one or more of the attributes that appear in the query result, by using the **ORDER BY** clause.
- **SELECT <attribute list> FROM <table list> [WHERE <condition>] [ORDER BY <attribute list>];**
- **SELECT Fname,Lname FROM EMPLOYEE WHERE Dno=5 ORDER BY Fname;**
- The default order is in ascending order of values. We can specify the keyword **DESC** if we want to see the result in a descending order of values.
- The keyword **ASC** can be used to specify ascending order explicitly.

ORDER BY

- The **ORDER BY** clause is used to sort the tuples in a query result based on the values of some attribute(s)
- Query 28: Retrieve a list of employees and the projects each works in, ordered by the employee's department, and within each department ordered alphabetically by employee last name.

```
Q28: SELECT  DNAME, LNAME, FNAME, PNAME
           FROM      DEPARTMENT, EMPLOYEE,
                   WORKS_ON, PROJECT
           WHERE  DNUMBER=DNO AND SSN=ESSN           AND
           PNO=PNUMBER
           ORDER BY      DNAME, LNAME
```

GROUPING

- In many cases, we want to apply the aggregate functions *to subgroups of tuples in a relation*
- Each subgroup of tuples consists of the set of tuples that have *the same value* for the *grouping attribute(s)*
- The function is applied to each subgroup independently
- SQL has a **GROUP BY**-clause for specifying the grouping attributes, which *must also appear in the SELECT-clause*

GROUPING (cont.)

- Query 20: For each department, retrieve the department number, the number of employees in the department, and their average salary.

**Q20: SELECT DNO, COUNT (*), AVG (SALARY)
FROM EMPLOYEE
GROUP BY DNO**

- In Q20, the EMPLOYEE tuples are divided into groups--each group having the same value for the grouping attribute DNO
- The COUNT and AVG functions are applied to each such group of tuples separately
- The SELECT-clause includes only the grouping attribute and the functions to be applied on each group of tuples
- A join condition can be used in conjunction with grouping

GROUPING (cont.)

- Query 21: For each project, retrieve the project number, project name, and the number of employees who work on that project.

Q21: **SELECT PNUMBER, PNAME, COUNT (*)**
 FROM PROJECT, WORKS_ON
 WHERE PNUMBER=PNO
 GROUP BY PNUMBER, PNAME

- In this case, the grouping and functions are applied *after* the joining of the two relations

THE HAVING-CLAUSE

- Sometimes we want to retrieve the values of these functions for only those *groups that satisfy certain conditions*
- The HAVING-clause is used for specifying a selection condition on groups (rather than on individual tuples)

THE HAVING-CLAUSE (cont.)

- Query 22: For each project *on which more than two employees work* , retrieve the project number, project name, and the number of employees who work on that project.

Q22:

```
SELECT      PNUMBER, PNAME, COUNT (*)
FROM PROJECT, WORKS_ON
WHERE       PNUMBER=PNO
GROUP BY    PNUMBER, PNAME
HAVING      COUNT (*) > 2
```

SUBSTRING COMPARISON

- The **LIKE** comparison operator is used to compare partial strings
- Two reserved characters are used: '%' (or '*' in some implementations) replaces an arbitrary number of characters, and '_' replaces a single arbitrary character

SUBSTRING COMPARISON (cont.)

- Query 25: Retrieve all employees whose address is in Houston, Texas. Here, the value of the ADDRESS attribute must contain the substring 'Houston,TX'.

Q25:	SELECT	FNAME, LNAME
	FROM	EMPLOYEE
	WHERE	ADDRESS LIKE
		'%Houston,TX%'

SUBSTRING COMPARISON (cont.)

- Query 26: Retrieve all employees who were born during the 1950s. Here, '5' must be the 8th character of the string (according to our format for date), so the BDATE value is '_____5_', with each underscore as a place holder for a single arbitrary character.

Q26: **SELECT FNAME, LNAME**
 FROM EMPLOYEE
 WHERE BDATE LIKE '_____5_'

- The LIKE operator allows us to get around the fact that each value is considered atomic and indivisible; hence, in SQL, character string attribute values are not atomic

ARITHMETIC OPERATIONS

- The standard arithmetic operators '+', '-', '*', and '/' (for addition, subtraction, multiplication, and division, respectively) can be applied to numeric values in an SQL query result
- Query 27: Show the effect of giving all employees who work on the 'ProductX' project a 10% raise.

```
Q27: SELECT  FNAME, LNAME, 1.1*SALARY  
            FROM    EMPLOYEE, WORKS_ON, PROJECT  
            WHERE   SSN=ESSN AND PNO=PNUMBER AND  
                   PNAME='ProductX'
```

Summary of SQL Queries

- A query in SQL can consist of up to six clauses, but only the first two, SELECT and FROM, are mandatory. The clauses are specified in the following order:

SELECT <attribute list>
FROM <table list>
[WHERE <condition>
[GROUP BY <grouping attribute(s)>
[HAVING <group condition>
[ORDER BY <attribute list>

Summary of SQL Queries (cont.)

- The SELECT-clause lists the attributes or functions to be retrieved
- The FROM-clause specifies all relations (or aliases) needed in the query but not those needed in nested queries
- The WHERE-clause specifies the conditions for selection and join of tuples from the relations specified in the FROM-clause
- GROUP BY specifies grouping attributes
- HAVING specifies a condition for selection of groups
- ORDER BY specifies an order for displaying the result of a query
- A query is evaluated by first applying the WHERE-clause, then GROUP BY and HAVING, and finally the SELECT-clause

Specifying Updates in SQL

- There are three SQL commands to modify the database; INSERT, DELETE, and UPDATE

INSERT

- In its simplest form, it is used to add one or more tuples to a relation
- Attribute values should be listed in the same order as the attributes were specified in the CREATE TABLE command

INSERT (cont.)

- Example:

**U1: INSERT INTO EMPLOYEE
VALUES ('Richard','K','Marini', '653298653', '30-DEC-52',
'98 Oak Forest,Katy,TX', 'M', 37000,'987654321', 4)**

- An alternate form of INSERT specifies explicitly the attribute names that correspond to the values in the new tuple
- Attributes with NULL values can be left out
- Example: Insert a tuple for a new EMPLOYEE for whom we only know the FNAME, LNAME, and SSN attributes.

**U1A: INSERT INTO EMPLOYEE (FNAME, LNAME, SSN)
VALUES ('Richard', 'Marini', '653298653')**

INSERT (cont.)

- Important Note: Only the constraints specified in the DDL commands are automatically enforced by the DBMS when updates are applied to the database
- Another variation of INSERT allows insertion of *multiple tuples* resulting from a query into a relation

INSERT (cont.)

- Example: Suppose we want to create a temporary table that has the name, number of employees, and total salaries for each department. A table DEPTS_INFO is created by U3A, and is loaded with the summary information retrieved from the database by the query in U3B.

U3A: **CREATE TABLE DEPTS_INFO**
 (DEPT_NAME VARCHAR(10),
 NO_OF_EMPS INTEGER,
 TOTAL_SAL INTEGER);

U3B: **INSERT INTO DEPTS_INFO (DEPT_NAME,**
 NO_OF_EMPS, TOTAL_SAL)
 SELECT DNAME, COUNT (*), SUM (SALARY)
 FROM DEPARTMENT, EMPLOYEE
 WHERE DNUMBER=DNO
 GROUP BY DNAME ;

INSERT (cont.)

- Note: The DEPTS_INFO table may not be up-to-date if we change the tuples in either the DEPARTMENT or the EMPLOYEE relations *after* issuing U3B. We have to create a view (see later) to keep such a table up to date.

DELETE

- Removes tuples from a relation
- Includes a WHERE-clause to select the tuples to be deleted
- Tuples are deleted from only *one table* at a time (unless CASCADE is specified on a referential integrity constraint)
- A missing WHERE-clause specifies that *all tuples* in the relation are to be deleted; the table then becomes an empty table
- The number of tuples deleted depends on the number of tuples in the relation that satisfy the WHERE-clause
- Referential integrity should be enforced

DELETE (cont.)

- Examples:

U4A:	DELETE FROM WHERE	EMPLOYEE LNAME='Brown'
U4B:	DELETE FROM WHERE	EMPLOYEE SSN='123456789'
U4C:	DELETE FROM WHERE (SELECT FROM DEPARTMENT WHERE	EMPLOYEE DNO IN DNUMBER DNAME='Research')
U4D:	DELETE FROM	EMPLOYEE

UPDATE

- Used to modify attribute values of one or more selected tuples
- A WHERE-clause selects the tuples to be modified
- An additional SET-clause specifies the attributes to be modified and their new values
- Each command modifies tuples *in the same relation*
- Referential integrity should be enforced

UPDATE (cont.)

- Example: Change the location and controlling department number of project number 10 to 'Bellaire' and 5, respectively.

**U5: UPDATE
SET
WHERE**

**PROJECT
PLOCATION = 'Bellaire', DNUM = 5
PNUMBER=10**

UPDATE (cont.)

- Example: Give all employees in the 'Research' department a 10% raise in salary.

U6: UPDATE EMPLOYEE

```
SET          SALARY = SALARY *1.1  
WHERE DNO IN (SELECT DNUMBER  
                FROM DEPARTMENT  
                WHERE      DNAME='Research')
```

- In this request, the modified SALARY value depends on the original SALARY value in each tuple
- The reference to the SALARY attribute on the right of = refers to the old SALARY value before modification
- The reference to the SALARY attribute on the left of = refers to the new SALARY value after modification

- Another feature allows the use of arithmetic in queries. The standard arithmetic operators for addition (+), subtraction (−), multiplication (*), and division (/) can be applied to numeric values or attributes with numeric domains
- **Query 13.** Show the resulting salaries if every employee working on the 'ProductX' project is given a 10 percent raise.
- **Q13: SELECT E.Fname, E.Lname, 1.1 * E.Salary AS Increased_sal FROM EMPLOYEE AS E, WORKS_ON AS W, PROJECT AS P WHERE E.Ssn=W.Essn AND W.Pno=P.Pnumber AND P.Pname='ProductX';**

Joined Relations Feature in SQL2

- Can specify a "joined relation" in the FROM-clause
- Looks like any other relation but is the result of a join
- Allows the user to specify different types of joins (regular "theta" JOIN, NATURAL JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, CROSS JOIN, etc)

JOIN

- Write `SELECT` statements to access data from more than one table using equality and nonequality joins
- View data that generally does not meet a join condition by using outer joins
- Join a table to itself by using a self join

- The concept of a joined table also allows the user to specify different types of join, such as NATURAL JOIN and various types of OUTER JOIN.
- In a NATURAL JOIN on two relations *R and S*, *no join condition is specified*;
- *an implicit EQUIJOIN condition for each pair of attributes with the same name from R and S is created. Each such pair of attributes is included only once in the resulting relation*

Cartesian Products

- A Cartesian product is formed when:
 - A join condition is omitted
 - A join condition is invalid
 - All rows in the first table are joined to all rows in the second table
- To avoid a Cartesian product, always include a valid join condition in a `WHERE` clause.

Generating a Cartesian Product

EMPLOYEES (20 rows)

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
100	King	90
101	Kochhar	90
...		
202	Fay	20
205	Higgins	110
206	Gietz	110

20 rows selected.

DEPARTMENTS (8 rows)

DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
10	Administration	1700
20	Marketing	1800
50	Shipping	1500
60	IT	1400
80	Sales	2500
90	Executive	1700
110	Accounting	1700
190	Contracting	1700

8 rows selected.

**Cartesian
product:** →
20x8=160 rows

EMPLOYEE_ID	DEPARTMENT_ID	LOCATION_ID
100	90	1700
101	90	1700
102	90	1700
103	60	1700
104	60	1700
107	60	1700
...		

160 rows selected.

Joining Tables

Use a join to query data from more than one table.

```
SELECT    table1.column, table2.column
FROM      table1, table2
WHERE     table1.column1 = table2.column2;
```

- Write the join condition in the `WHERE` clause.
- Prefix the column name with the table name when the same column name appears in more than one table.

What is an Equijoin?

EMPLOYEES

EMPLOYEE_ID	DEPARTMENT_ID
200	10
201	20
202	20
124	50
141	50
142	50
143	50
144	50
103	60
104	60
107	60
149	80
174	80
176	80

...

DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME
10	Administration
20	Marketing
20	Marketing
50	Shipping
50	Shipping
50	Shipping
50	Shipping
50	Shipping
50	Shipping
60	IT
60	IT
60	IT
80	Sales
80	Sales
80	Sales

...



Foreign key



Primary key

Retrieving Records with Equijoins

```
SELECT employees.employee_id, employees.last_name,  
       employees.department_id, departments.department_id,  
       departments.location_id  
FROM   employees, departments  
WHERE  employees.department_id = departments.department_id;
```

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID	LOCATION_ID
200	Whalen	10	10	1700
201	Hartstein	20	20	1800
202	Fay	20	20	1800
124	Mourgos	50	50	1500
141	Rajs	50	50	1500
142	Davies	50	50	1500
143	Matos	50	50	1500
144	Vargas	50	50	1500

19 rows selected.

Joining More than Two Tables

EMPLOYEES

LAST_NAME	DEPARTMENT_ID
King	90
Kochhar	90
De Haan	90
Hunold	60
Ernst	60
Lorentz	60
Mourgos	50
Rajs	50
Davies	50
Matos	50
Vargas	50
Zlotkey	80
Abel	80
Taylor	80

20 rows selected.

DEPARTMENTS

DEPARTMENT_ID	LOCATION_ID
10	1700
20	1800
50	1500
60	1400
80	2500
90	1700
110	1700
190	1700

8 rows selected.

LOCATIONS

LOCATION_ID	CITY
1400	Southlake
1500	South San Francisco
1700	Seattle
1800	Toronto
2500	Oxford

- To join n tables together, you need a minimum of $n-1$ join conditions. For example, to join three tables, a minimum of two joins is required.

Non-EquiJoins

EMPLOYEES

LAST_NAME	SALARY
King	24000
Kochhar	17000
De Haan	17000
Hunold	9000
Ernst	6000
Lorentz	4200
Mourgos	5800
Rajs	3500
Davies	3100
Matos	2600
Vargas	2500
Zlotkey	10500
Abel	11000
Taylor	8600

20 rows selected.

JOB_GRADES

GRA	LOWEST_SAL	HIGHEST_SAL
A	1000	2999
B	3000	5999
C	6000	9999
D	10000	14999
E	15000	24999
F	25000	40000



Salary in the EMPLOYEES table must be between lowest salary and highest salary in the JOB_GRADES table.

Retrieving Records with Non-Equi Joins

```
SELECT e.last_name, e.salary, j.grade_level
FROM   employees e, job_grades j
WHERE  e.salary
      BETWEEN j.lowest_sal AND j.highest_sal;
```

LAST_NAME	SALARY	GRA
Matos	2600	A
Vargas	2500	A
Lorentz	4200	B
Mourgos	5800	B
Rajs	3500	B
Davies	3100	B
Whalen	4400	B
Hunold	9000	C
Ernst	6000	C

20 rows selected.

Outer Joins

DEPARTMENTS

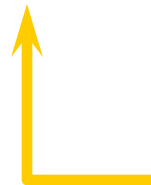
DEPARTMENT_NAME	DEPARTMENT_ID
Administration	10
Marketing	20
Shipping	50
IT	60
Sales	80
Executive	90
Accounting	110
Contracting	190

8 rows selected.

EMPLOYEES

DEPARTMENT_ID	LAST_NAME
90	King
90	Kochhar
90	De Haan
60	Hunold
60	Ernst
60	Lorentz
50	Mourgos
50	Rajs
50	Davies
50	Matos
50	Vargas
80	Zlotkey
...	

20 rows selected.



There are no employees in department 190.

Outer Joins Syntax

- You use an outer join to also see rows that do not meet the join condition.
- The Outer join operator is the plus sign (+).

```
SELECT table1.column, table2.column  
FROM   table1, table2  
WHERE  table1.column += table2.column;
```

```
SELECT table1.column, table2.column  
FROM   table1, table2  
WHERE  table1.column +=+ table2.column;
```


RIGHT Outer Joins

```
SELECT e.last_name, e.department_id, d.department_name
FROM   employees e, departments d
WHERE  e.department_id =+ d.department_id ;
```

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Hartstein	20	Marketing
Fay	20	Marketing
Mourgos	50	Shipping
Rajs	50	Shipping
Davies	50	Shipping
Matos	50	Shipping
...		
Gietz	110	Accounting
		Contracting

20 rows selected.

LEFT Outer Joins

```
SELECT e.last_name, e.department_id, d.department_name
FROM   employees e, departments d
WHERE  e.department_id <= d.department_id ;
```

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Fay	20	Marketing
Hartstein	20	Marketing

De Haan	90	Executive
Kochhar	90	Executive
King	90	Executive
Gietz	110	Accounting
Higgins	110	Accounting
Grant		

20 rows selected.

LEFT OUTER JOIN

```
SELECT e.last_name, e.department_id, d.department_name
FROM   employees e
LEFT OUTER JOIN departments d
ON     (e.department_id = d.department_id) ;
```

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Fay	20	Marketing
Hartstein	20	Marketing
...		
De Haan	90	Executive
Kochhar	90	Executive
King	90	Executive
Gietz	110	Accounting
Higgins	110	Accounting
Grant		

20 rows selected.

RIGHT OUTER JOIN

```
SELECT e.last_name, e.department_id, d.department_name
FROM   employees e
RIGHT OUTER JOIN departments d
ON      (e.department_id = d.department_id) ;
```

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
King	90	Executive
Kochhar	90	Executive
...		
Whalen	10	Administration
Hartstein	20	Marketing
Fay	20	Marketing
Higgins	110	Accounting
Gietz	110	Accounting
		Contracting

20 rows selected.

FULL OUTER JOIN

```
SELECT e.last_name, e.department_id, d.department_name
FROM   employees e
FULL OUTER JOIN departments d
ON     (e.department_id = d.department_id) ;
```

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Fay	20	Marketing
...		
De Haan	90	Executive
Kochhar	90	Executive
King	90	Executive
Gietz	110	Accounting
Higgins	110	Accounting
Grant		
		Contracting

21 rows selected.

Self Joins

EMPLOYEES (WORKER)

EMPLOYEE_ID	LAST_NAME	MANAGER_ID
100	King	
101	Kochhar	100
102	De Haan	100
103	Hunold	102
104	Ernst	103
107	Lorentz	103
124	Mourgos	100

...

EMPLOYEES (MANAGER)

EMPLOYEE_ID	LAST_NAME
100	King
101	Kochhar
102	De Haan
103	Hunold
104	Ernst
107	Lorentz
124	Mourgos

...



MANAGER_ID in the WORKER table is equal to
EMPLOYEE_ID in the MANAGER table.

Joining a Table to Itself

```
SELECT worker.last_name || ' works for '
       || manager.last_name
FROM   employees worker, employees manager
WHERE  worker.manager_id = manager.employee_id ;
```

WORKER.LAST_NAME 'WORKSFOR' MANAGER.LAST_NAME
Kochhar works for King
De Haan works for King
Mourgos works for King
Zlotkey works for King
Hartstein works for King
Whalen works for Kochhar
Higgins works for Kochhar
Hunold works for De Haan
Ernst works for Hunold

19 rows selected.

Creating Cross Joins

- The `CROSS JOIN` clause produces the cross-product of two tables.
- This is the same as a Cartesian product between the two tables.

```
SELECT last_name, department_name  
FROM   employees  
CROSS JOIN departments ;
```

LAST_NAME	DEPARTMENT_NAME
King	Administration
Kochhar	Administration
De Haan	Administration
Hunold	Administration
...	

160 rows selected.

Additional Conditions

```
SELECT e.employee_id, e.last_name, e.department_id,  
       d.department_id, d.location_id  
FROM   employees e JOIN departments d  
ON     (e.department_id = d.department_id)  
AND    e.manager_id = 149 ;
```

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID	LOCATION_ID
174	Abel	80	80	2500
176	Taylor	80	80	2500

Formal Relational Query Languages

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

Relational Algebra

- The basic set of operations for the relational model is known as the relational algebra. These operations enable a user to specify basic retrieval requests.
- Procedural language.
- The result of a retrieval is a new relation, which may have been formed from one or more relations. The **algebra operations** thus produce new relations, which can be further manipulated using operations of the same algebra.
- A sequence of relational algebra operations forms a **relational algebra expression**, whose result will also be a relation that represents the result of a database query (or retrieval request).

Relational Algebra

- Basic operations:
 - Selection (σ) Selects a subset of rows from relation.
 - Projection (π) Deletes unwanted columns from relation.
 - Cross-product (\times) Allows us to combine two relations.
 - Set-difference ($-$) Tuples in reln. 1, but not in reln. 2.
 - Union (\cup) Tuples in reln. 1 and in reln. 2.
- Additional operations:
 - Intersection, join, division, renaming
- Since each operation returns a relation, **operations can be composed**

Unary Relational Operations

- **SELECT Operation**

SELECT operation is used to select a *subset* of the tuples from a relation that satisfy a **selection condition**. It is a filter that keeps only those tuples that satisfy a qualifying condition – those satisfying the condition are selected while others are discarded.

Example: To select the EMPLOYEE tuples whose department number is four or those whose salary is greater than \$30,000 the following notation is used:

$\sigma_{DNO = 4} (EMPLOYEE)$
 $\sigma_{SALARY > 30,000} (EMPLOYEE)$

In general, the select operation is denoted by $\sigma_{\langle \text{selection condition} \rangle}(R)$ where the symbol σ (sigma) is used to denote the select operator, and the selection condition is a Boolean expression specified on the attributes of relation R

Selection(σ)

- Selects rows that satisfy *selection condition*.
- In general, the SELECT operation is denoted by
- $\sigma_{\langle \text{selection condition} \rangle}(R)$

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

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

sname	rating
yuppy	9
rusty	10

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

Unary Relational Operations

SELECT Operation Properties

- The SELECT operation $\sigma_{\langle \text{selection condition} \rangle}(R)$ produces a relation S that has the same schema as R
- A cascaded SELECT operation may be replaced by a single selection with a conjunction of all the conditions; i.e.
$$= \sigma_{\langle \text{condition1} \rangle \text{ AND } \langle \text{condition2} \rangle \text{ AND } \langle \text{condition3} \rangle} (R)$$

Unary Relational Operations (cont.)

Figure 7.8 Results of SELECT and PROJECT operations.

- (a) $\sigma_{(DNO=4 \text{ AND } SALARY>25000) \text{ OR } (DNO=5 \text{ AND } SALARY>30000)}(EMPLOYEE)$.
 (b) $\pi_{LNAME, FNAME, SALARY}(EMPLOYEE)$. (c) $\pi_{SEX, SALARY}(EMPLOYEE)$

(a)

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
Franklin	T	Wong	333445555	1955-12-08	638 Voss,Houston,TX	M	40000	888665555	5
Jennifer		Wallace	987654321	1941-06-20	291 Berry,Bellaire,TX	F	43000	888665555	4
Ramesh		Narayan	666884444	1962-09-15	975 FireOak,Humble,TX	M	38000	333445555	5

(b)

LNAME	FNAME	SALARY
Smith	John	30000
Wong	Franklin	40000
Zelaya	Alicia	25000
Wallace	Jennifer	43000
Narayan	Ramesh	38000
English	Joyce	25000
Jabbar	Ahmad	25000
Borg	James	55000

(c)

SEX	SALARY
M	30000
M	40000
F	25000
F	43000
M	38000
M	25000
M	55000

Unary Relational Operations (cont.)

- **PROJECT Operation(π)**

This operation selects certain *columns* from the table and discards the other columns. The PROJECT creates a vertical partitioning – one with the needed columns (attributes) containing results of the operation and other containing the discarded Columns.

Example: To list each employee's first and last name and salary, the following is used:

$$\pi_{\text{LNAME, FNAME, SALARY}}(\text{EMPLOYEE})$$

The general form of the project operation is $\pi_{\langle \text{attribute list} \rangle}(\text{R})$ where π (π) is the symbol used to represent the project operation and $\langle \text{attribute list} \rangle$ is the desired list of attributes from the attributes of relation R.

The project operation *removes any duplicate tuples*, so the result of the project operation is a set of tuples and hence a valid relation.

Unary Relational Operations (cont.)

Figure 7.8 Results of SELECT and PROJECT operations.

(a) $\sigma_{(DNO=4 \text{ AND } SALARY>25000) \text{ OR } (DNO=5 \text{ AND } SALARY>30000)}(EMPLOYEE)$.

(b) $\pi_{LNAME, FNAME, SALARY}(EMPLOYEE)$. (c) $\pi_{SEX, SALARY}(EMPLOYEE)$

(a)

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
Franklin	T	Wong	333445555	1955-12-08	638 Voss,Houston,TX	M	40000	888665555	5
Jennifer		Wallace	987654321	1941-06-20	291 Berry,Bellaire,TX	F	43000	888665555	4
Ramesh		Narayan	666884444	1962-09-15	975 FireOak,Humble,TX	M	38000	333445555	5

(b)

LNAME	FNAME	SALARY
Smith	John	30000
Wong	Franklin	40000
Zelaya	Alicia	25000
Wallace	Jennifer	43000
Narayan	Ramesh	38000
English	Joyce	25000
Jabbar	Ahmad	25000
Borg	James	55000

(c)

SEX	SALARY
M	30000
M	40000
F	25000
F	43000
M	38000
M	25000
M	55000

RENAME(ρ)

- $\rho S(B1, B2, \dots, Bn)(R)$ or $\rho S(R)$ or $\rho(B1, B2, \dots, Bn)(R)$
- where the symbol ρ (rho) is used to denote the RENAME operator, *S is the new relation*
- name, and *B1, B2, ..., Bn are the new attribute names.*
- *The first expression* renames both the relation and its attributes, the second renames the relation only,
- and the third renames the attributes only. If the attributes of *R are (A1, A2, ..., An) in*
- that order, then each *Ai is renamed as Bi*

Relational Algebra Operations From Set Theory

- **UNION Operation**

The result of this operation, denoted by $R \cup S$, is a relation that includes all tuples that are either in R or in S or in both R and S. Duplicate tuples are eliminated.

Example: To retrieve the social security numbers of all employees who either work in department 5 or directly supervise an employee who works in department 5, we can use the union operation as follows:

DEP5_EMPS $\leftarrow \sigma_{\text{DNO}=5}(\text{EMPLOYEE})$

RESULT1 $\leftarrow \pi_{\text{SSN}}(\text{DEP5_EMPS})$

RESULT2(SSN) $\leftarrow \pi_{\text{SUPERSSN}}(\text{DEP5_EMPS})$

RESULT $\leftarrow \text{RESULT1} \cup \text{RESULT2}$

The union operation produces the tuples that are in either RESULT1 or RESULT2 or both. The two operands must be “type compatible”.

Unary Relational Operations (cont.)

Figure 7.9 Results of relational algebra expressions.

(a) $\pi_{\text{LNAME, FNAME, SALARY}} (\sigma_{\text{DNO}=5}(\text{EMPLOYEE}))$. (b) The same expression using intermediate relations and renaming of attributes.

(a)

FNAME	LNAME	SALARY
John	Smith	30000
Franklin	Wong	40000
Ramesh	Narayan	38000
Joyce	English	25000

(b)

TEMP	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
	John	B	Smith	123456789	1985-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
	Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
	Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
	Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5

	FIRSTNAME	LASTNAME	SALARY
	John	Smith	30000
	Franklin	Wong	40000
	Ramesh	Narayan	38000
	Joyce	English	25000

Relational Algebra Operations From Set Theory

- **Type Compatibility**

- The operand relations $R_1(A_1, A_2, \dots, A_n)$ and $R_2(B_1, B_2, \dots, B_n)$ must have the same number of attributes, and the domains of corresponding attributes must be compatible; that is, $\text{dom}(A_i) = \text{dom}(B_i)$ for $i=1, 2, \dots, n$.
- The resulting relation for $R_1 \cup R_2$, $R_1 \cap R_2$, or $R_1 - R_2$ has the same attribute names as the *first* operand relation R_1 (by convention).

Relational Algebra Operations From Set Theory

(cont.) – use Fig. 6.4

Figure 7.11 Illustrating the set operations union, intersection, and difference. (a) Two union compatible relations. (b) $\text{STUDENT} \cup \text{INSTRUCTOR}$. (c) $\text{STUDENT} \cap \text{INSTRUCTOR}$. (d) $\text{STUDENT} - \text{INSTRUCTOR}$. (e) $\text{INSTRUCTOR} - \text{STUDENT}$.

(a)

STUDENT	FN	LN
	Susan	Yao
	Ramesh	Shah
	Johnny	Kohler
	Barbara	Jones
	Amy	Ford
	Jimmy	Wang
	Ernest	Gilbert

INSTRUCTOR	FNAME	LNAME
	John	Smith
	Ricardo	Browne
	Susan	Yao
	Francis	Johnson
	Ramesh	Shah

(b)

FN	LN
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert
John	Smith
Ricardo	Browne
Francis	Johnson

(c)

FN	LN
Susan	Yao
Ramesh	Shah

(d)

FN	LN
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

(e)

FNAME	LNAME
John	Smith
Ricardo	Browne
Francis	Johnson

Relational Algebra Operations From Set Theory (cont.)

- **INTERSECTION OPERATION**

The result of this operation, denoted by $R \cap S$, is a relation that includes all tuples that are in both R and S. The two operands must be "type compatible"

Example: The result of the intersection operation (figure below) includes only those who are both students and instructors.

FN	LN
Susan	Yao
Ramesh	Shah

(d)

FN	LN
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

STUDENT \cap INSTRUCTOR

Relational Algebra Operations From Set Theory (cont.)

- **Set Difference (or MINUS) Operation**

The result of this operation, denoted by $R - S$, is a relation that includes all tuples that are in R but not in S . The two operands must be "type compatible".

Example: The figure shows the names of students who are not instructors, and the names of instructors who are not students.

STUDENT	FN	LN
	Susan	Yao
	Ramesh	Shah
	Johnny	Kohler
	Barbara	Jones
	Amy	Ford
	Jimmy	Wang
	Ernest	Gilbert

FN	LN
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

STUDENT-INSTRUCTOR

FNAME	LNAME
John	Smith
Ricardo	Browne
Francis	Johnson

INSTRUCTOR-STUDENT

Relational Algebra Operations From Set Theory (cont.)

- Notice that both union and intersection are *commutative operations*; that is

$$\mathbf{R \cup S = S \cup R, \text{ and } R \cap S = S \cap R}$$

- Both union and intersection can be treated as n-ary operations applicable to any number of relations as both are *associative operations*; that is

$$\mathbf{R \cup (S \cup T) = (R \cup S) \cup T, \text{ and } (R \cap S) \cap T = R \cap (S \cap T)}$$

- The minus operation is *not commutative*; that is, in general

$$\mathbf{R - S \neq S - R}$$

Relational Algebra Operations From Set Theory (cont.)

- **CARTESIAN (or cross product) Operation**

- This operation is used to combine tuples from two relations in a combinatorial fashion. In general, the result of $R(A_1, A_2, \dots, A_n) \times S(B_1, B_2, \dots, B_m)$ is a relation Q with degree $n + m$ attributes $Q(A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m)$, in that order. The resulting relation Q has one tuple for each combination of tuples—one from R and one from S .
- Hence, if R has n_R tuples (denoted as $|R| = n_R$), and S has n_S tuples, the $|R \times S|$ will have $n_R * n_S$ tuples.
- The two operands do NOT have to be "type compatible"

Example:

FEMALE_EMPS $\leftarrow \sigma_{SEX='F'}(EMPLOYEE)$

EMPNAMES $\leftarrow \pi_{FNAME, LNAME, SSN}(FEMALE_EMPS)$

EMP_DEPENDENTS $\leftarrow EMPNAMES \times DEPENDENT$

Relational Algebra Operations From Set Theory (cont.)

Figure 7.12 An illustration of the CARTESIAN PRODUCT operation.

FEMALE_EMPS	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
	Alicia	J	Zelaysa	999887777	1966-07-19	3321 Castle Spring, TX	F	25000	987654321	4
	Jennifer	S	Wallace	987654321	1944-06-20	291 Beery Bellare, TX	F	43000	888905555	4
	Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5

EMP_NAMES	FNAME	LNAME	SSN
	Alicia	Zelaysa	999887777
	Jennifer	Wallace	987654321
	Joyce	English	453453453


EMP_DEPENDENTS	FNAME	LNAME	SSN	ESSN	DEPENDENT_NAME	SEX	BDATE	***
	Alicia	Zelaysa	999887777	333445555	Alice	F	1966-04-05	***
	Alicia	Zelaysa	999887777	333445555	Theodore	M	1963-10-25	***
	Alicia	Zelaysa	999887777	333445555	Joy	F	1958-05-03	***
	Alicia	Zelaysa	999887777	987654321	Abner	M	1942-02-28	***
	Alicia	Zelaysa	999887777	123456789	Michael	M	1988-01-04	***
	Alicia	Zelaysa	999887777	123456789	Alice	F	1968-12-30	***
	Alicia	Zelaysa	999887777	123456789	Elizabeth	F	1967-05-05	***
	Jennifer	Wallace	987654321	333445555	Alice	F	1966-04-05	***
	Jennifer	Wallace	987654321	333445555	Theodore	M	1963-10-25	***
	Jennifer	Wallace	987654321	333445555	Joy	F	1958-05-03	***
	Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28	***
	Jennifer	Wallace	987654321	123456789	Michael	M	1988-01-04	***
	Jennifer	Wallace	987654321	123456789	Alice	F	1968-12-30	***
	Jennifer	Wallace	987654321	123456789	Elizabeth	F	1967-05-05	***
	Joyce	English	453453453	333445555	Alice	F	1966-04-05	***
	Joyce	English	453453453	333445555	Theodore	M	1963-10-25	***
	Joyce	English	453453453	333445555	Joy	F	1958-05-03	***
	Joyce	English	453453453	987654321	Abner	M	1942-02-28	***
	Joyce	English	453453453	123456789	Michael	M	1988-01-04	***
	Joyce	English	453453453	123456789	Alice	F	1968-12-30	***
	Joyce	English	453453453	123456789	Elizabeth	F	1967-05-05	***

ACTUAL_DEPENDENTS	FNAME	LNAME	SSN	ESSN	DEPENDENT_NAME	SEX	BDATE
	Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28

RESULT	FNAME	LNAME	DEPENDENT_NAME
	Jennifer	Wallace	Abner

Binary Relational Operations

- **JOIN Operation**

- The sequence of cartesian product followed by select is used quite commonly to identify and select related tuples from two relations, a special operation, called **JOIN**. It is denoted by a 
- This operation is very important for any relational database with more than a single relation, because it allows us to process relationships among relations.
- The general form of a join operation on two relations $R(A_1, A_2, \dots, A_n)$ and $S(B_1, B_2, \dots, B_m)$ is:

$$R \bowtie_{\langle \text{join condition} \rangle} S$$

where R and S can be any relations that result from general *relational algebra expressions*.

Joins

- Condition Join or theta:

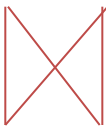
θ (theta) is one of the comparison operators $\{=, <, \leq, >, \geq, \neq\}$.

$$R \bowtie_c S = \sigma_c (R \times S)$$

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

- *Result schema* same as that of cross-product.
- Fewer tuples than cross-product. Filters tuples not satisfying the join condition.
- Sometimes called a *theta-join*.


Binary Relational Operations (cont.)

Example: Suppose that we want to retrieve the name of the manager of each department. To get the manager's name, we need to combine each DEPARTMENT tuple with the EMPLOYEE tuple whose SSN value matches the MGRSSN value in the department tuple. We do this by using the join  operation.

DEPT_MGR \leftarrow **DEPARTMENT**  **EMPLOYEE**
MGRSSN=SSN

RESULT $\leftarrow \pi_{Dname, Lname, Fname}(\text{DEPT_MGR})$

DEPT_MGR	DNAME	DNUMBER	MGRSSN	...	FNAME	MINIT	LNAME	SSN	...
	Research	5	333445555	...	Franklin	T	Wong	333445555	...
	Administration	4	987654321	...	Jennifer	S	Wallace	987654321	...
	Headquarters	1	888665555	...	James	E	Borg	888665555	...

FIGURE 6.6 Result of the JOIN operation **DEPT_MGR** \leftarrow **DEPARTMENT**  **EMPLOYEE**, MGRSSN=SSN.

Binary Relational Operations (cont.)

- **EQUIJOIN Operation(=)**

The most common use of join involves join conditions with equality comparisons only. Such a join, where the only comparison operator used is =, is called an EQUIJOIN. In the result of an EQUIJOIN we always have one or more pairs of attributes (whose names need not be identical) that have *identical values* in every tuple.

The JOIN seen in the previous example was EQUIJOIN.

- **NATURAL JOIN Operation(*)**

Because one of each pair of attributes with identical values is superfluous, a new operation called natural join was created to get rid of the second attribute in an EQUIJOIN condition.

The standard definition of natural join requires that the two join attributes, or each pair of corresponding join attributes, have the **same name** in both relations. If this is not the case, a renaming operation is applied first.

- ex: combine each PROJECT tuple with the DEPARTMENT tuple that controls the project.

$$\text{PROJ_DEPT} \leftarrow \text{PROJECT} *_{\rho(\text{Dname}, \text{Dnum}, \text{Mgr_ssn}, \text{Mgr_start_date})}(\text{DEPARTMENT})$$

or

- $$\text{DEPT} \leftarrow \rho(\text{Dname}, \text{Dnum}, \text{Mgr_ssn}, \text{Mgr_start_date})(\text{DEPARTMENT})$$

$$\text{PROJ_DEPT} \leftarrow \text{PROJECT} * \text{DEPT}$$

Binary Relational Operations (cont.)

(a)

PROJ_DEPT

Pname	<u>Pnumber</u>	Plocation	Dnum	Dname	Mgr_ssn	Mgr_start_date
ProductX	1	Bellaire	5	Research	333445555	1988-05-22
ProductY	2	Sugarland	5	Research	333445555	1988-05-22
ProductZ	3	Houston	5	Research	333445555	1988-05-22
Computerization	10	Stafford	4	Administration	987654321	1995-01-01
Reorganization	20	Houston	1	Headquarters	888665555	1981-06-19
Newbenefits	30	Stafford	4	Administration	987654321	1995-01-01

(b)

DEPT_LOCS

Dname	Dnumber	Mgr_ssn	Mgr_start_date	Location
Headquarters	1	888665555	1981-06-19	Houston
Administration	4	987654321	1995-01-01	Stafford
Research	5	333445555	1988-05-22	Bellaire
Research	5	333445555	1988-05-22	Sugarland
Research	5	333445555	1988-05-22	Houston

Figure 6.7

Results of two NATURAL JOIN operations. (a) PROJ_DEPT \leftarrow PROJECT * DEPT.
 (b) DEPT_LOCS \leftarrow DEPARTMENT * DEPT_LOCATIONS.

Complete Set of Relational Operations

- The set of operations including **select** σ , **project** π , **union** \cup , **set difference** $-$, and **cartesian product** \times is called a complete set because any other relational algebra expression can be expressed by a combination of these five operations. ✕

- For example:

$$\mathbf{R} \cap \mathbf{S} = (\mathbf{R} \cup \mathbf{S}) - ((\mathbf{R} - \mathbf{S}) \cup (\mathbf{S} - \mathbf{R}))$$

$$\mathbf{R} \bowtie_{\langle \text{join condition} \rangle} \mathbf{S} = \sigma_{\langle \text{join condition} \rangle} (\mathbf{R} \times \mathbf{S})$$

Binary Relational Operations (cont.)

- **DIVISION Operation**

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

Binary Relational Operations (cont.)

SSN_PNOS	ESSN	PNO
	123456789	1
	123456789	2
	666884444	3
	453453453	1
	453453453	2
	333445555	2
	333445555	3
	333445555	10
	333445555	20
	999887777	30
	999887777	10
	987987987	10
	987987987	30
	987654321	30
	987654321	20
	888665555	20

R	A	B
	a1	b1
	a2	b1
	a3	b1
	a4	b1
	a1	b2
	a3	b2
	a2	b3
	a3	b3
	a4	b3
	a1	b4
	a2	b4
	a3	b4

Table 6.1 Operations of Relational Algebra

OPERATION	PURPOSE	NOTATION
SELECT	Selects all tuples that satisfy the selection condition from a relation R .	$\sigma_{\langle \text{selection condition} \rangle}(R)$
PROJECT	Produces a new relation with only some of the attributes of R , and removes duplicate tuples.	$\pi_{\langle \text{attribute list} \rangle}(R)$
THETA JOIN	Produces all combinations of tuples from R_1 and R_2 that satisfy the join condition.	$R_1 \bowtie_{\langle \text{join condition} \rangle} R_2$
EQUIJOIN	Produces all the combinations of tuples from R_1 and R_2 that satisfy a join condition with only equality comparisons.	$R_1 \bowtie_{\langle \text{join condition} \rangle} R_2$, OR $R_1 \bowtie_{(\langle \text{join attributes 1} \rangle, \langle \text{join attributes 2} \rangle)} R_2$
NATURAL JOIN	Same as EQUIJOIN except that the join attributes of R_2 are not included in the resulting relation; if the join attributes have the same names, they do not have to be specified at all.	$R_1 *_{\langle \text{join condition} \rangle} R_2$, OR $R_1 *_{(\langle \text{join attributes 1} \rangle, \langle \text{join attributes 2} \rangle)} R_2$ OR $R_1 * R_2$
UNION	Produces a relation that includes all the tuples in R_1 or R_2 or both R_1 and R_2 ; R_1 and R_2 must be union compatible.	$R_1 \cup R_2$
INTERSECTION	Produces a relation that includes all the tuples in both R_1 and R_2 ; R_1 and R_2 must be union compatible.	$R_1 \cap R_2$
DIFFERENCE	Produces a relation that includes all the tuples in R_1 that are not in R_2 ; R_1 and R_2 must be union compatible.	$R_1 - R_2$
CARTESIAN PRODUCT	Produces a relation that has the attributes of R_1 and R_2 and includes as tuples all possible combinations of tuples from R_1 and R_2 .	$R_1 \times R_2$
DIVISION	Produces a relation $R(X)$ that includes all tuples $t[X]$ in $R_1(Z)$ that appear in R_1 in combination with every tuple from $R_2(Y)$, where $Z = X \cup Y$.	$R_1(Z) \div R_2(Y)$

Additional Relational Operations

- **Aggregate Functions and Grouping**
 - A type of request that cannot be expressed in the basic relational algebra is to specify mathematical **aggregate functions** on collections of values from the database.
 - Examples of such functions include retrieving the average or total salary of all employees or the total number of employee tuples. These functions are used in simple statistical queries that summarize information from the database tuples.
 - Common functions applied to collections of numeric values include SUM, AVERAGE, MAXIMUM, and MINIMUM. The COUNT function is used for counting tuples or values.

- $\langle \text{grouping attributes} \rangle \mathfrak{F} \langle \text{function list} \rangle (R)$
AGGREGATE FUNCTION operation, using the symbol \mathfrak{F} (pronounced *script F*)
- where $\langle \text{grouping attributes} \rangle$ is a list of attributes of the relation specified in R ,
- $\langle \text{function list} \rangle$ is a list of ($\langle \text{function} \rangle \langle \text{attribute} \rangle$) pairs.
- In each such pair, $\langle \text{function} \rangle$ is one of the allowed functions—such as SUM, AVERAGE, MAXIMUM, MINIMUM, COUNT—and $\langle \text{attribute} \rangle$ is an attribute of the relation specified by R .

Figure 6.10

The aggregate function operation.

- a. $\rho_{R(Dno, No_of_employees, Average_sal)}(Dno \int COUNT Ssn, AVERAGE Salary (EMPLOYEE)).$
 b. $Dno \int COUNT Ssn, AVERAGE Salary (EMPLOYEE).$
 c. $\int COUNT Ssn, AVERAGE Salary (EMPLOYEE).$

R

(a)

Dno	No_of_employees	Average_sal
5	4	33250
4	3	31000
1	1	55000

(b)

Dno	Count_ssn	Average_salary
5	4	33250
4	3	31000
1	1	55000

(c)

Count_ssn	Average_salary
8	35125

Additional Relational Operations (cont.)

Use of the Functional operator \mathcal{F}

$\mathcal{F}_{\text{MAX Salary}}$ (**Employee**) retrieves the maximum salary value from the Employee relation

$\mathcal{F}_{\text{MIN Salary}}$ (**Employee**) retrieves the minimum Salary value from the Employee relation

$\mathcal{F}_{\text{SUM Salary}}$ (**Employee**) retrieves the sum of the Salary from the Employee relation

$\text{DNO } \mathcal{F}_{\text{COUNT SSN, AVERAGE Salary}}$ (**Employee**) groups employees by DNO (department number) and computes the count of employees and average salary per department.[Note: count just counts the number of rows, without removing duplicates]

Additional Relational Operations (cont.)

- **The OUTER JOIN Operation**

- In NATURAL JOIN tuples without a *matching* (or *related*) tuple are eliminated from the join result. Tuples with null in the join attributes are also eliminated. This amounts to loss of information.
- A set of operations, called outer joins, can be used when we want to keep all the tuples in R, or all those in S, or all those in both relations in the result of the join, regardless of whether or not they have matching tuples in the other relation.
- The left outer join operation keeps every tuple in the *first* or *left* relation R in $R \bowtie\!\!\!\! \bowtie S$; if no matching tuple is found in S, then the attributes of S in the join result are filled or “padded” with null values.
- A similar operation, right outer join, keeps every tuple in the *second* or right relation S in the result of $R \bowtie\!\!\!\! \bowtie S$.
- A third operation, full outer join, denoted by $\overline{\bowtie\!\!\!\! \bowtie}$ keeps all tuples in both the left and the right relations when no matching tuples are found, padding them with null values as needed.

Additional Relational Operations (cont.)

RESULT	FNAME	MINIT	LNAME	DNAME
	John	B	Smith	null
	Franklin	T	Wong	Research
	Alicia	J	Zelaya	null
	Jennifer	S	Wallace	Administration
	Ramesh	K	Narayan	null
	Joyce	A	English	null
	Ahmad	V	Jabbar	null
	James	E	Borg	Headquarters

Examples of Queries in Relational Algebra

- **Q1: Retrieve the name and address of all employees who work for the 'Research' department.**

$\text{RESEARCH_DEPT} \leftarrow \sigma_{\text{DNAME}='Research'} (\text{DEPARTMENT})$

$\text{RESEARCH_EMPS} \leftarrow (\text{RESEARCH_DEPT} \bowtie_{\text{DNUMBER}=\text{DNO}} \text{EMPLOYEE})$

$\text{RESULT} \leftarrow \pi_{\text{FNAME}, \text{LNAME}, \text{ADDRESS}} (\text{RESEARCH_EMPS})$

Relational Calculus

- A **relational calculus** expression creates a new relation, which is specified in terms of variables that range over rows of the stored database relations (in **tuple calculus**) or over columns of the stored relations (in **domain calculus**).
- In a calculus expression, there is *no order of operations* to specify how to retrieve the query result—a calculus expression specifies only what information the result should contain. This is the main distinguishing feature between relational algebra and relational calculus.
- Relational calculus is considered to be a **nonprocedural** language. This differs from relational algebra, where we must write a *sequence of operations* to specify a retrieval request; hence relational algebra can be considered as a **procedural** way of stating a query.

Tuple Relational Calculus

- The tuple relational calculus is based on specifying a number of **tuple variables**. Each tuple variable usually *ranges over* a particular database relation, meaning that the variable may take as its value any individual tuple from that relation.
- A simple tuple relational calculus query is of the form
 $\{t \mid \text{COND}(t)\}$
where t is a tuple variable and $\text{COND}(t)$ is a conditional expression involving t . The result of such a query is the set of all tuples t that satisfy $\text{COND}(t)$.

Example: To find the first and last names of all employees whose salary is above \$50,000, we can write the following tuple calculus expression:

$\{t.\text{FNAME}, t.\text{LNAME} \mid \text{EMPLOYEE}(t) \text{ AND } t.\text{SALARY} > 50000\}$

The condition $\text{EMPLOYEE}(t)$ specifies that the **range relation** of tuple variable t is EMPLOYEE . The first and last name ($\text{PROJECTION } \pi_{\text{FNAME}, \text{LNAME}}$) of each EMPLOYEE tuple t that satisfies the condition $t.\text{SALARY} > 50000$ ($\text{SELECTION } \sigma_{\text{SALARY} > 50000}$) will be retrieved.

The Existential and Universal Quantifiers

- Two special symbols called **quantifiers** can appear in formulas; these are the **universal quantifier** (\forall) and the **existential quantifier** (\exists).
- Informally, a tuple variable t is bound if it is quantified, meaning that it appears in an $(\forall t)$ or $(\exists t)$ clause; otherwise, it is **free**.
- If F is a formula, then so is $(\exists t)(F)$, where t is a tuple variable. The formula $(\exists t)(F)$ is true if the formula F evaluates to true for *some* (at least one) tuple assigned to free occurrences of t in F ; otherwise $(\exists t)(F)$ is **false**.
- If F is a formula, then so is $(\forall t)(F)$, where t is a tuple variable. The formula $(\forall t)(F)$ is true if the formula F evaluates to true for *every tuple* (in the universe) assigned to free occurrences of t in F ; otherwise $(\forall t)(F)$ is **false**. It is called the universal or “for all” quantifier because every tuple in “the universe of” tuples must make F true to make the quantified formula true.

Example Query Using Existential Quantifier

- Retrieve the name and address of all employees who work for the 'Research' department.

Query :

$\{t.FNAME, t.LNAME, t.ADDRESS \mid \text{EMPLOYEE}(t) \text{ and } (\exists d) (\text{DEPARTMENT}(d) \text{ and } d.DNAME='Research' \text{ and } d.DNUMBER=t.DNO) \}$

- The *only free tuple variables* in a relational calculus expression should be those that appear to the left of the bar (\mid). In above query, t is the only free variable; it is then *bound successively* to each tuple. If a tuple *satisfies the conditions* specified in the query, the attributes FNAME, LNAME, and ADDRESS are retrieved for each such tuple.
- The conditions EMPLOYEE (t) and DEPARTMENT(d) specify the range relations for t and d. The condition d.DNAME = 'Research' is a selection condition and corresponds to a SELECT operation in the relational algebra, whereas the condition d.DNUMBER = t.DNO is a JOIN condition.

Example Query Using Universal Quantifier

- Find the names of employees who work on *all* the projects controlled by department number 5.

Query :

**{e.LNAME, e.FNAME | EMPLOYEE(e) and ((\forall x)(not(PROJECT(x)) or not(x.DNUM=5)
OR (\exists w)(WORKS_ON(w) and w.ESSN=e.SSN and x.PNUMBER=w.PNO))) }**

- Exclude from the universal quantification all tuples that we are not interested in by making the condition true *for all such tuples*. The first tuples to exclude (by making them evaluate automatically to true) are those that are not in the relation R of interest.
- In query above, using the expression not(PROJECT(x)) inside the universally quantified formula evaluates to true all tuples x that are not in the PROJECT relation. Then we exclude the tuples we are not interested in from R itself. The expression not(x.DNUM=5) evaluates to true all tuples x that are in the project relation but are not controlled by department 5.
- Finally, we specify a condition that must hold on all the remaining tuples in R.
(\exists w)(WORKS_ON(w) and w.ESSN=e.SSN and x.PNUMBER=w.PNO)

Languages Based on Tuple Relational Calculus

- The language **SQL** is based on tuple calculus. It uses the basic
SELECT <list of attributes>
FROM <list of relations>
WHERE <conditions>
block structure to express the queries in tuple calculus where the SELECT clause mentions the attributes being projected, the FROM clause mentions the relations needed in the query, and the WHERE clause mentions the selection as well as the join conditions.
SQL syntax is expanded further to accommodate other operations. (See Chapter 8).
- Another language which is based on tuple calculus is **QUEL** which actually uses the range variables as in tuple calculus.
Its syntax includes:
RANGE OF <variable name> IS <relation name>
Then it uses
RETRIEVE <list of attributes from range variables>
WHERE <conditions>
This language was proposed in the relational DBMS INGRES.

The Domain Relational Calculus

- Another variation of relational calculus called the domain relational calculus, or simply, **domain calculus** is equivalent to tuple calculus and to relational algebra.
- The language called QBE (Query-By-Example) that is related to domain calculus was developed almost concurrently to SQL at IBM Research, Yorktown Heights, New York. Domain calculus was thought of as a way to explain what QBE does.
- Domain calculus differs from tuple calculus in the *type of variables* used in formulas: rather than having variables range over tuples, the variables range over single values from domains of attributes. To form a relation of degree n for a query result, we must have n of these **domain variables**—one for each attribute.
- An expression of the domain calculus is of the form
 $\{x_1, x_2, \dots, x_n \mid \text{COND}(x_1, x_2, \dots, x_n, x_{n+1}, x_{n+2}, \dots, x_{n+m})\}$
where $x_1, x_2, \dots, x_n, x_{n+1}, x_{n+2}, \dots, x_{n+m}$ are domain variables that range over domains (of attributes) and COND is a **condition** or **formula** of the domain relational calculus.

2.1 Functional Dependencies (1)

- Functional dependencies (FDs) are used to specify *formal measures* of the "goodness" of relational designs
- FDs and keys are used to define **normal forms** for relations
- FDs are **constraints** that are derived from the *meaning* and *interrelationships* of the data attributes
- A set of attributes *X functionally determines* a set of attributes *Y* if the value of *X* determines a unique value for *Y*

Functional Dependencies (2)

- $X \rightarrow Y$ holds if whenever two tuples have the same value for X , they *must have* the same value for Y
- For any two tuples $t1$ and $t2$ in any relation instance $r(R)$: *If* $t1[X]=t2[X]$, *then* $t1[Y]=t2[Y]$
- $X \rightarrow Y$ in R specifies a *constraint* on all relation instances $r(R)$
- Written as $X \rightarrow Y$; can be displayed graphically on a relation schema as in Figures. (denoted by the arrow:).
- FDs are derived from the real-world constraints on the attributes

Examples of FD constraints (1)

- social security number determines employee name
SSN \rightarrow ENAME
- project number determines project name and location
PNUMBER \rightarrow {PNAME, PLOCATION}
- employee ssn and project number determines the hours per week that the employee works on the project
{SSN, PNUMBER} \rightarrow HOURS

Examples of FD constraints (2)

- An FD is a property of the attributes in the schema R
- The constraint must hold on *every relation instance* $r(R)$
- If K is a key of R , then K functionally determines all attributes in R (since we never have two distinct tuples with $t1[K]=t2[K]$)

2.2 Inference Rules for FDs (1)

- Given a set of FDs F , we can *infer* additional FDs that hold whenever the FDs in F hold
- $(\text{Dept_no} \rightarrow \text{Mgr_ssn})$
- $(\text{Mgr_ssn} \rightarrow \text{Mgr_phone})$, then these two dependencies together imply that
- $\text{Dept_no} \rightarrow \text{Mgr_phone}$.
- This is an inferred FD and need *not* be explicitly stated in addition to the two given FDs
- 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

Armstrong's inference rules:

IR1. (**Reflexive**) If $X \supseteq Y$, then $X \rightarrow Y$

IR2. (**Augmentation**) If $X \rightarrow Y$, then $XZ \rightarrow YZ$

(Notation: XZ stands for $X \cup Z$)

IR3. (**Transitive**) If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$

- IR1, IR2, IR3 form a *sound* and *complete* set of inference rules
- Because IR1 generates dependencies that are always true, such dependencies are called *trivial*.

Formally, a functional dependency $X \rightarrow Y$ is trivial if $X \supseteq Y$; otherwise, it is nontrivial

Inference Rules for FDs (2)

Some **additional inference rules** that are useful:

(Decomposition) If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$

(Union) If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$

(Pseudotransitivity) If $X \rightarrow Y$ and $WY \rightarrow Z$, then $WX \rightarrow Z$

- The last three inference rules, as well as any other inference rules, can be deduced from IR1, IR2, and IR3 (completeness property)

Inference Rules for FDs (3)

- **Closure** of a set F of FDs is the set F^+ of all FDs that can be inferred from F
- $F = \{Ssn \rightarrow \{Ename, Bdate, Address, Dnumber\}, Dnumber \rightarrow \{Dname, Dmgr_ssn\}\}$
- Some of the additional functional dependencies that we can *infer from F are the following*:
 - $Ssn \rightarrow \{Dname, Dmgr_ssn\}$
 - $Ssn \rightarrow Ssn$
 - $Dnumber \rightarrow Dname$

- **Closure** of a set of attributes X with respect to F is the set X^+ of all attributes that are functionally determined by X
- X^+ can be calculated by repeatedly applying IR1, IR2, IR3 using the FDs in F
- $F = \{Ssn \rightarrow Ename,$
- $Pnumber \rightarrow \{Pname, Plocation\},$
- $\{Ssn, Pnumber\} \rightarrow Hours\}$
- Closure:-
- $\{Ssn\}^+ = \{Ssn, Ename\}$
- $\{Pnumber\}^+ = \{Pnumber, Pname, Plocation\}$
- $\{Ssn, Pnumber\}^+ = \{Ssn, Pnumber, Ename, Pname, Plocation, Hours\}$

Properties of Relational Decompositions

- Using the functional dependencies, the algorithms decompose the universal relation schema R into a set of relation schemas
- $D = \{R_1, R_2, \dots, R_m\}$ that will become the relational database schema; D is called a **decomposition of R** .
- ***Decomposition*** should be
 - dependency preserving
 - Lossless decomposition

Nonadditive (Lossless) Join Property of a Decomposition

- **Definition.** Formally, a decomposition $D = \{R_1, R_2, \dots, R_m\}$ of R has the lossless(nonadditive) join property with respect to the set of dependencies F on R if, for every relation state r of R that satisfies F , the following holds, where $*$ is the NATURAL JOIN of all the relations in D : $*(\pi_{R_1}(r), \dots, \pi_{R_m}(r)) = r$.

Algorithm

- **Algorithm 16.3. Testing for Nonadditive Join Property**
 - **Input:** A universal relation R , a decomposition $D = \{R_1, R_2, \dots, R_m\}$ of R , and a set F of functional dependencies.
1. Create an initial matrix S with one row i for each relation R_i in D , and one column j for each attribute A_j in R
 2. Set $S(i, j) := b_{ij}$ for all matrix entries.
 3. For each row i representing relation schema R_i {for each column j representing attribute A_j {if (relation R_i includes attribute A_j) then set $S(i, j) := a_j$ };};

4. Repeat the following loop until a *complete loop execution results in no changes to S* {for each functional dependency $X \rightarrow Y$ in F {for all rows in S that have the same symbols in the columns corresponding to attributes in X {make the symbols in each column that correspond to an attribute in Y be the same in all these rows as follows: If any of the rows has an a symbol for the column, set the other rows to that same a symbol in the column. If no a symbol exists for the attribute in any of the rows, choose one of the b symbols that appears in one of the rows for the attribute and set the other rows to that same b symbol in the column } ; } ; } ;
5. If a row is made up entirely of a symbols, then the decomposition has the nonadditive join property; otherwise, it does not.

Dependency Preservation Property of a Decomposition

- **Definition.** Given a set of dependencies F on R , the projection of F on R_i , denoted by $\pi_{R_i}(F)$ where R_i is a subset of R , is the set of dependencies $X \rightarrow Y$ in F^+ such that the attributes in $X \cup Y$ are all contained in R_i . Hence, the projection of F on each relation schema R_i in the decomposition D is the set of functional dependencies in F^+ , the closure of F , such that all their left- and right-hand-side attributes are in R_i . We say that a decomposition $D = \{R_1, R_2, \dots, R_m\}$ of R is dependency-preserving with respect to F if the union of the projections of F on each R_i in D is equivalent to F ; that is, $((\pi_{R_1}(F)) \cup \dots \cup (\pi_{R_m}(F)))^+ = F^+$.

algorithm

- It guarantees only the dependency-preserving property; it does *not guarantee the nonadditive join property*
- The first step of Algorithm to find a minimal cover G for F ;

2.3 Equivalence of Sets of FDs

- Two sets of FDs F and G are **equivalent** if:
 - every FD in F can be inferred from G , *and*
 - every FD in G can be inferred from F
- Hence, F and G are equivalent if $F^+ = G^+$

Definition: F **covers** G if every FD in G can be inferred from F (i.e., if $G^+ \text{ subset-of } F^+$)

- F and G are equivalent if F covers G and G covers F
- There is an algorithm for checking equivalence of sets of FDs

2.4 Minimal Sets of FDs (1)

- A set of FDs is **minimal** if it satisfies the following conditions:
 - (1) Every dependency in F has a single attribute for its RHS.
 - (2) We cannot remove any dependency from F and have a set of dependencies that is equivalent to F .
 - (3) We cannot replace any dependency $X \rightarrow A$ in F with a dependency $Y \rightarrow A$, where Y proper-subset-of X (Y subset-of X) and still have a set of dependencies that is equivalent to F .

Dependency-Preserving Decomposition into 3NF Schemas

- **Algorithm 16.4. Relational Synthesis into 3NF with Dependency Preservation**
- **Input:** A universal relation *R* and a set of functional dependencies *F* on the attributes of *R*.
- **1. Find a minimal cover *G* for *F* (use Algorithm 16.2);**
- **2. For each left-hand-side *X* of a functional dependency that appears in *G*, create** a relation schema in *D* with attributes $\{X \cup \{A_1\} \cup \{A_2\} \dots \cup \{A_k\}\}$, where $X \rightarrow A_1$, $X \rightarrow A_2$, ..., $X \rightarrow A_k$ are the only dependencies in *G* with *X* as the left-hand-side (*X* is the key of this relation);
- **3. Place any remaining attributes (that have not been placed in any relation) in a single relation schema** to ensure the attribute preservation property.

3 Normal Forms Based on Primary Keys

3.1 Normalization of Relations

3.2 Practical Use of Normal Forms

3.3 Definitions of Keys and Attributes
Participating in Keys

3.4 First Normal Form

3.5 Second Normal Form

3.6 Third Normal Form

3.1 Normalization of Relations (1)

- **Normalization:** The process of decomposing unsatisfactory "bad" relations by breaking up their attributes into smaller relations
- **Normal form:** Condition using keys and FDs of a relation to certify whether a relation schema is in a particular normal form

3.2 Practical Use of Normal Forms

- **Normalization** is carried out in practice so that the resulting designs are of high quality and meet the desirable properties
- The practical utility of these normal forms becomes questionable when the constraints on which they are based are **hard to understand** or to **detect**
- The database designers ***need not*** normalize to the highest possible normal form. (usually up to 3NF, BCNF or 4NF)
- **Denormalization:** the process of storing the join of higher normal form relations as a base relation—which is in a lower normal form

3.3 Definitions of Keys and Attributes Participating in Keys (1)

- A **superkey** of a relation schema $R = \{A_1, A_2, \dots, A_n\}$ is a set of attributes S subset-of R with the property that no two tuples t_1 and t_2 in any legal relation state r of R will have $t_1[S] = t_2[S]$
- A **key** K is a superkey with the *additional property* that removal of any attribute from K will cause K not to be a superkey any more.

Definitions of Keys and Attributes Participating in Keys (2)

- If a relation schema has more than one key, each is called a **candidate key**. One of the candidate keys is *arbitrarily* designated to be the **primary key**, and the others are called *secondary keys*.
- A **Prime attribute** must be a member of *some candidate key*
- A **Nonprime attribute** is not a prime attribute—that is, it is not a member of any candidate key.


3.2 First Normal Form

- Disallows composite attributes, multivalued attributes, and **nested relations**; attributes whose values *for an individual tuple* are non-atomic
- Considered to be part of the definition of relation

(a)

DEPARTMENT

Dname	<u>Dnumber</u>	Dmgr_ssn	Dlocations



(b)

DEPARTMENT

Dname	<u>Dnumber</u>	Dmgr_ssn	Dlocations
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

(c)

DEPARTMENT

Dname	<u>Dnumber</u>	Dmgr_ssn	<u>Dlocation</u>
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

Figure 15.9

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

3.3 Second Normal Form (1)

- Uses the concepts of **FDs**, **primary key**

Definitions:

- **Prime attribute** - attribute that is member of the primary key K
- **Full functional dependency** - a FD $Y \rightarrow Z$ where removal of any attribute from Y means the FD does not hold any more

Examples: - {SSN, PNUMBER} \rightarrow HOURS is a full FD since neither SSN \rightarrow HOURS nor PNUMBER \rightarrow HOURS hold

- {SSN, PNUMBER} \rightarrow ENAME is *not* a full FD (it is called a *partial dependency*) since SSN \rightarrow ENAME also holds

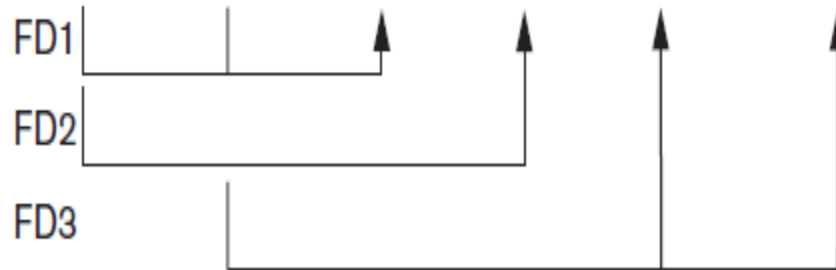
Second Normal Form (2)

- A relation schema R is in **second normal form (2NF)** if every non-prime attribute A in R is fully functionally dependent on the primary key
- R can be decomposed into 2NF relations via the process of 2NF normalization

(a)

EMP_PROJ

<u>Ssn</u>	<u>Pnumber</u>	Hours	Ename	Pname	Plocation
------------	----------------	-------	-------	-------	-----------



2NF Normalization



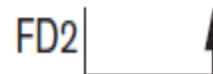
EP1

<u>Ssn</u>	<u>Pnumber</u>	Hours
------------	----------------	-------



EP2

<u>Ssn</u>	Ename
------------	-------



EP3

<u>Pnumber</u>	Pname	Plocation
----------------	-------	-----------



3.4 Third Normal Form (1)

Definition:

- **Transitive functional dependency** - a FD $X \rightarrow Z$ that can be derived from two FDs $X \rightarrow Y$ and $Y \rightarrow Z$

Examples:

- $SSN \rightarrow DMGRSSN$ is a *transitive* FD since $SSN \rightarrow DNUMBER$ and $DNUMBER \rightarrow DMGRSSN$ hold
- $SSN \rightarrow ENAME$ is *non-transitive* since there is no set of attributes X where $SSN \rightarrow X$ and $X \rightarrow ENAME$

Third Normal Form (2)

- A relation schema R is in **third normal form (3NF)** if it is in 2NF *and* no non-prime attribute A in R is transitively dependent on the primary key
- R can be decomposed into 3NF relations via the process of 3NF normalization

NOTE:

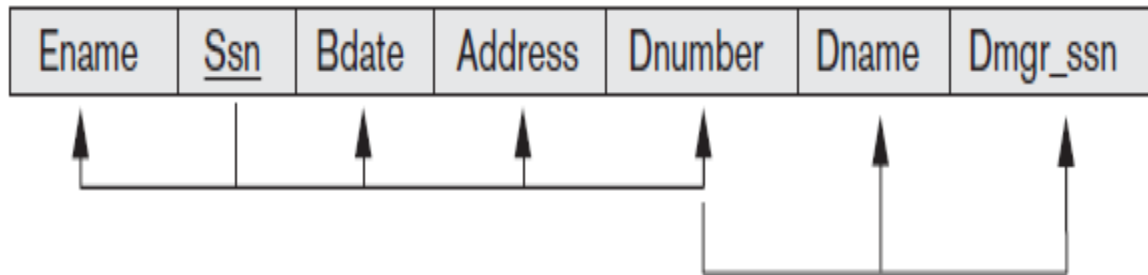
In $X \rightarrow Y$ and $Y \rightarrow Z$, with X as the primary key, we consider this a problem only if Y is not a candidate key. When Y is a candidate key, there is no problem with the transitive dependency .

E.g., Consider EMP (SSN, Emp#, Salary).

Here, $SSN \rightarrow Emp\# \rightarrow Salary$ and Emp# is a candidate key.

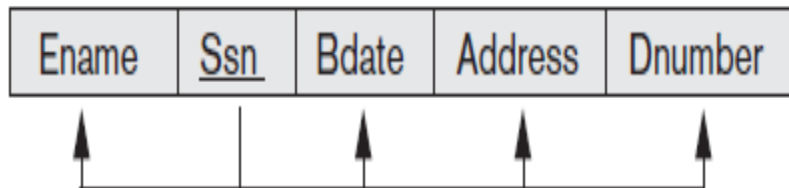
(b)

EMP_DEPT

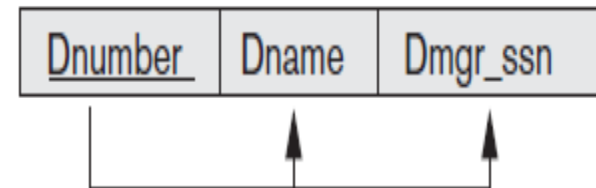


3NF Normalization

ED1



ED2



BCNF (Boyce-Codd Normal Form)

- A relation schema R is in **Boyce-Codd Normal Form (BCNF)** if whenever an FD $X \rightarrow A$ holds in R , then X is a superkey of R
- Each normal form is strictly stronger than the previous one
 - Every 2NF relation is in 1NF
 - Every 3NF relation is in 2NF
 - Every BCNF relation is in 3NF
- There exist relations that are in 3NF but not in BCNF
- The goal is to have each relation in BCNF (or 3NF)

Figure 10.12 Boyce-Codd normal form

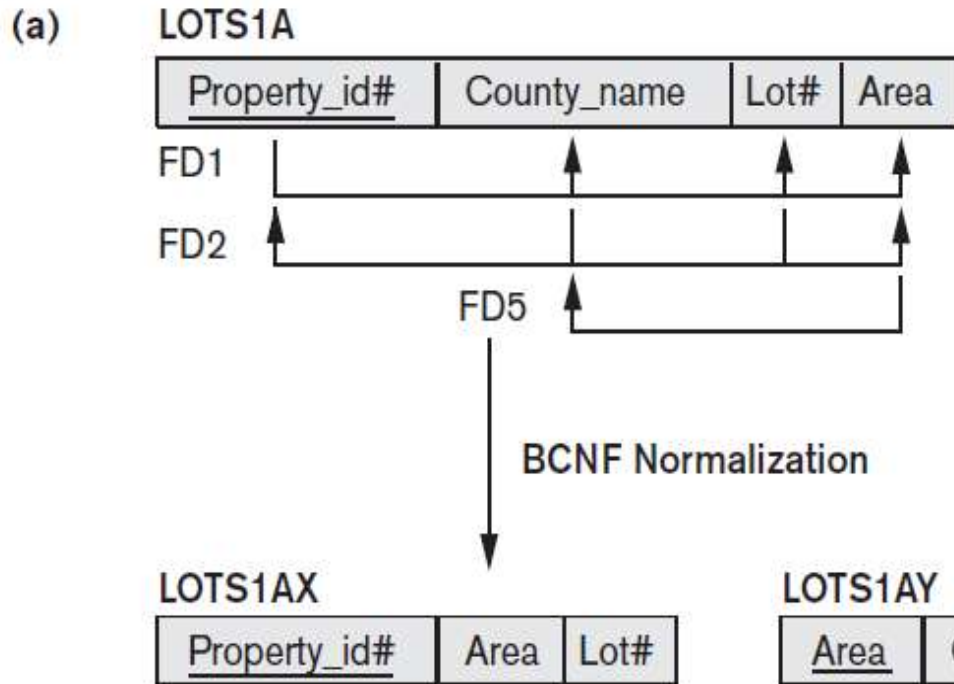


Figure 15.13

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.

Note: The above figure is now called Figure 10.12 in Edition 4

Figure 10.13 a relation TEACH that is in 3NF but not in BCNF

Figure 14.13 A relation TEACH that is in 3NF but not in BCNF.

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

© Addison Wesley Longman, Inc. 2000, Elmasri/Navathe, Fundamentals of Database Systems, Third Edition

Note: The above figure is now called Figure 10.13 in Edition 4

Achieving the BCNF by Decomposition (1)

- Two FDs exist in the relation TEACH:
fd1: { student, course} -> instructor
fd2: instructor -> course
- {student, course} is a candidate key for this relation and that the dependencies shown follow the pattern in Figure 10.12 (b). So this relation is in 3NF but not in BCNF
- 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. (See Algorithm 11.3)

Achieving the BCNF by Decomposition (2)

- Three possible decompositions for relation TEACH
 1. {student, instructor} and {student, course}
 2. {course, instructor} and {course, student}
 3. {instructor, course} and {instructor, student}
- All three decompositions will lose fd1. We have to settle for sacrificing the functional dependency preservation. But we cannot sacrifice the non-additivity property after decomposition.
- Out of the above three, only the 3rd decomposition will not generate spurious tuples after join.(and hence has the non-additivity property).
- A test to determine whether a binary decomposition (decomposition into two relations) is nonadditive (lossless) is discussed in section 11.1.4 under Property LJ1. Verify that the third decomposition above meets the property.

MODULE-4

- **1.Introduction to Transaction Processing:-**

- **1.1 Single-User versus Multiuser Systems**

- A DBMS is **single-user** if at most one user at a time can use the system,
- and it is multiuser if many users can use the system—and hence access the database—concurrently.
- Single-user DBMSs are mostly restricted to personal computer systems; most other DBMSs are multiuser.
- For example, an airline reservations system is used by hundreds of travel agents and reservation clerks concurrently.

- Multiple users can access databases simultaneously
- because of the concept of **multiprogramming, which allows the operating system**
- of the computer to execute multiple programs—or **processes—at the same time. A**
- single central processing unit (CPU) can only execute at most one process at a time.
- However, **multiprogramming operating systems execute some commands from**
- one process, then suspend that process and execute some commands from the next process, and so on.
- A process is resumed at the point where it was suspended whenever
- it gets its turn to use the CPU again. Hence, concurrent execution of processes
- is actually **interleaved**

- If the computer system has multiple hardware processors (CPUs), **parallel processing**
- of multiple processes is possible, as illustrated by processes C and D in Figure 21.1. Most of the theory concerning concurrency control in databases is developed in terms of **interleaved concurrency**

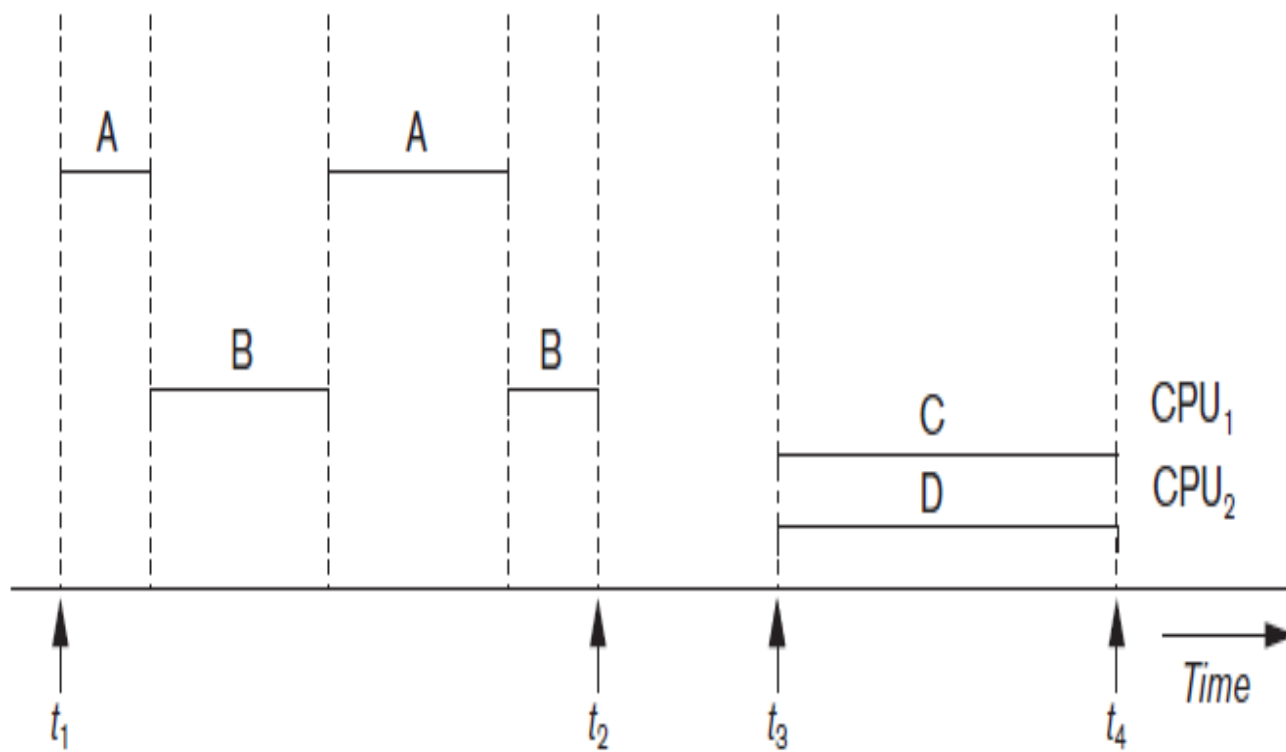


Figure 21.1
Interleaved processing versus parallel processing of concurrent transactions.

Transactions, Read and Write Operations, and DBMS Buffers

- A **transaction** is an executing program that forms a logical unit of database processing.
- A transaction includes one or more database access operations—these can include insertion, deletion, modification, or retrieval operations
- One way of specifying the transaction boundaries is by specifying explicit **begin transaction** and **end transaction** statements in an application program
- In this case, all database access operations between the two are considered as forming one transaction.

- If the database operations in a transaction do not update the database but only retrieve data, the transaction is called a **read-only transaction**; otherwise it is known as a **read-write transaction**.
- Using this simplified database model, the basic database access
- operations that a transaction can include are as follows:
- ■ **read_item(*X*)**. *Reads a database item named X into a program variable. To* simplify our notation, we assume that *the program variable is also named X*.
- ■ **write_item(*X*)**. *Writes the value of program variable X into the database* item named *X*.

- The DBMS will maintain in the **database cache** a **number of data buffers in main memory**. Each **buffer typically**
- holds the contents of one database disk block, which contains some of the database items being processed
- The **read-set** of a transaction is the set of all items that the transaction reads, and the **write-set** is the set of all items that the transaction writes. For example, the read-set of $T1$ in
- Figure 21.2 is $\{X, Y\}$ and its write-set is also $\{X, Y\}$.

Why Concurrency Control Is Needed

- Consider the ex: of airline reservation system
- With following transaction

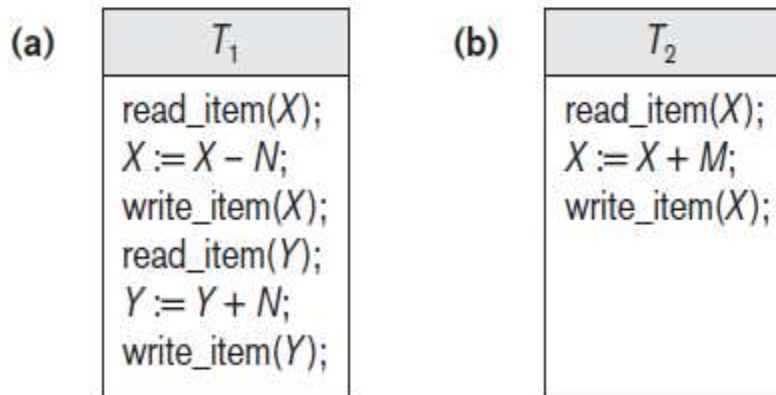


Figure 21.2

Two sample transactions. (a) Transaction T_1 . (b) Transaction T_2 .

- **The Lost Update Problem:** This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database items incorrect.
- Figure 21.3(a); then the final value of item X is *incorrect* because $T2$ reads the value of X before $T1$ changes it in the database, and hence the updated value resulting from $T1$ is lost.

For example, if $X = 80$ at the start (originally there were 80 reservations on the flight), $N = 5$ ($T1$ transfers 5 seat reservations from the flight corresponding to X to the flight corresponding to Y), and $M = 4$ ($T2$ reserves 4 seats on X), the final result should be $X = 79$.

However, in the interleaving of operations shown in Figure 21.3(a), it is $X = 84$ because the update in $T1$ that

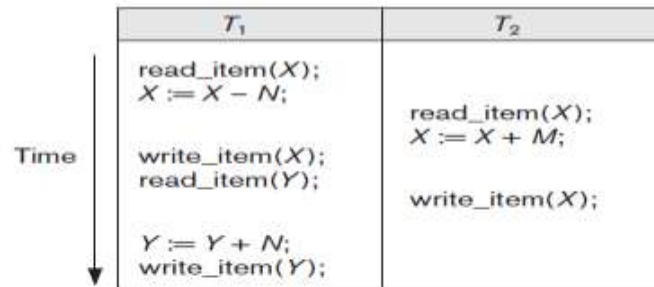
- *removed the five seats from X was lost.*

- **The Temporary Update (or Dirty Read) Problem.**
- **This problem occurs when** one transaction updates a database item and then the transaction fails for some reason
- .Meanwhile, the updated item is accessed (read) by another transaction before it is changed back to its original value.
- Figure 21.3(b) shows an example where *T1 updates item X and then fails before completion, so the system must change X back to its original value.*
- *Before it can do so, however, transaction T2 reads the temporary value of X, which will not be recorded permanently in the database because of the failure of T1.*
- *The value of item X that is read by T2 is called dirty data because it has been created by a transaction that has not completed and committed yet; hence, this problem is also known as the dirty read problem.*

- **The Incorrect Summary Problem.** If one transaction is calculating an aggregate
- summary function on a number of database items while other transactions are
- updating some of these items, the aggregate function may calculate some values
- before they are updated and others after they are updated. For example, suppose
- that a transaction *T3* is calculating the total number of reservations on all the flights;
- meanwhile, transaction *T1* is executing. If the interleaving of operations shown in
- Figure 21.3(c) occurs, the result of *T3* will be off by an amount *N* because *T3* reads
- the value of *X* after *N* seats have been subtracted from it but reads the value of *Y*
- before those *N* seats have been added to it.

- **The Unrepeatable Read Problem.** Another problem that may occur is called
- *unrepeatable read*, where a transaction T reads the same item twice and the item is
- changed by another transaction T between the two reads. Hence, T receives
- *different values for its two reads of the same item. This may occur, for example, if*
- during an airline reservation transaction, a customer inquires about seat availability
- on several flights. When the customer decides on a particular flight, the transaction
- then reads the number of seats on that flight a second time before completing the
- reservation, and it may end up reading a different value for the item.

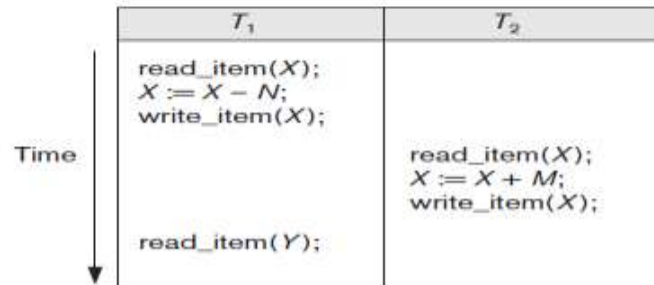
(a)

**Figure 21.3**

Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

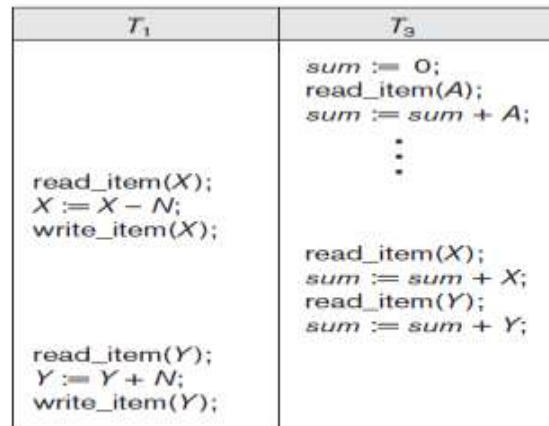
Item X has an incorrect value because its update by T_1 is *lost* (overwritten).

(b)



Transaction T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the *temporary* incorrect value of X .

(c)



T_3 reads X after N is subtracted and reads Y before N is added; a wrong summary is the result (off by N).

Why Recovery Is Needed

- Whenever a transaction is submitted to a DBMS for execution, the system is responsible for making sure that either all the operations in the transaction are completed successfully and their effect is recorded permanently in the database,-committed
- The transaction does not have any effect on the database or any other transactions-aborted
- If a transaction **fails** after executing some of its operations but before executing all of them, the operations already executed must be undone and have no lasting effect.

Database Concurrency Control

1 Purpose of Concurrency Control

- To enforce Isolation (through mutual exclusion) among conflicting transactions.
- To preserve database consistency through consistency preserving execution of transactions.
- To resolve read-write and write-write conflicts.

Example: In concurrent execution environment if T_1 conflicts with T_2 over a data item A , then the existing concurrency control decides if T_1 or T_2 should get the A and if the other transaction is rolled-back or waits.

Database Concurrency Control

Two-Phase Locking Techniques

Locking is an operation which secures (a) permission to Read or (b) permission to Write a data item for a transaction. Example: *Lock(X)*. Data item *X* is locked in behalf of the requesting transaction.

Unlocking is an operation which removes these permissions from the data item. Example: *Unlock(X)*. Data item *X* is made available to all other transactions.

Lock and *Unlock* are atomic operations.

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

Two locks modes (a) shared (read) and (b) exclusive (write).

Shared mode: shared lock (X). More than one transaction can apply share lock on X for reading its value but no write lock can be applied on X by any other transaction.

Exclusive mode: Write lock (X). Only one write lock on X can exist at any time and no shared lock can be applied by any other transaction on X .

Conflict matrix

	Read	Write
Read	Y	N
Write	N	N

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

Lock Manager: Managing locks on data items.

Lock table: Lock manager uses it to store the identity of transaction locking (the data item, lock mode and pointer to the next data item locked). One simple way to implement a lock table is through linked list.

Transaction ID	Data item id	lock mode	Ptr to next data item
T1	X1	Read	Next

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

Database requires that all transactions should be well-formed. A transaction is well-formed if:

- It must lock the data item before it reads or writes to it.
- It must not lock an already locked data items and it must not try to unlock a free data item.

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

The following code performs the lock operation:

```
B: if LOCK (X) = 0 (*item is unlocked*)  
    then LOCK (X)  $\leftarrow$  1 (*lock the item*)  
    else begin  
        wait (until lock (X) = 0) and  
        the lock manager wakes up the transaction);  
    goto B  
end;
```

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

The following code performs the unlock operation:

```
LOCK (X)  $\leftarrow$  0 (*unlock the item*)  
if any transactions are waiting then  
    wake up one of the waiting the transactions;
```

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

The following code performs the read operation:

```
B: if LOCK (X) = "unlocked" then
    begin LOCK (X) ← "read-locked";
        no_of_reads (X) ← 1;
    end
else if LOCK (X) ← "read-locked" then
    no_of_reads (X) ← no_of_reads (X) + 1
else begin wait (until LOCK (X) = "unlocked" and
    the lock manager wakes up the transaction);
    go to B
end;
```

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

The following code performs the write lock operation:

```
B: if LOCK (X) = "unlocked" then
    LOCK (X) ← "write-locked";
else begin
    wait (until LOCK (X) = "unlocked" and
        the lock manager wakes up the transaction);
    go to B
end;
```

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

The following code performs the unlock operation:

```
if LOCK (X) = "write-locked" then
    begin LOCK (X) ← "unlocked";
        wakes up one of the transactions, if any
    end
else if LOCK (X) ← "read-locked" then
    begin
        no_of_reads (X) ← no_of_reads (X) -1
        if no_of_reads (X) = 0 then
            begin
                LOCK (X) = "unlocked";
                wake up one of the transactions, if any
            end
        end
    end;
end;
```

When we use the share/exclusive locking scheme, the system must enforce the following rules:

- 1. A transaction T must issue the operation $read_lock(X)$ or $write_lock(X)$ before any $read_item(X)$ operation is performed in T .
- 2. A transaction T must issue the operation $write_lock(X)$ before any $write_item(X)$ operation is performed in T .
- 3. A transaction T must issue the operation $unlock(X)$ after all $read_item(X)$ and $write_item(X)$ operations are completed in T .
- 4. A transaction T must not issue a $read_lock(X)$ operation if it already holds a read(shared) lock or a write(exclusive) lock on item X .
- 5. A transaction T must not issue a $write_lock(X)$ operation if it already holds a read(shared) lock or a write(exclusive) lock on item X .
- 6. A transaction T must not issue the operation $unlock(X)$ unless it already holds a read (shared) lock or a write(exclusive) lock on item X .
-

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

Lock conversion

Lock upgrade: existing read lock to write lock

if T_i has a read-lock (X) and T_j has no read-lock (X) ($i \neq j$) then
 convert read-lock (X) to write-lock (X)
else
 force T_i to wait until T_j unlocks X

Lock downgrade: existing write lock to read lock

T_i has a write-lock (X) (*no transaction can have any lock on X*)
 convert write-lock (X) to read-lock (X)

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

Two Phases: (a) Locking (Growing) (b) Unlocking (Shrinking).

Locking (Growing) Phase: A transaction applies locks (read or write) on desired data items one at a time.

Unlocking (Shrinking) Phase: A transaction unlocks its locked data items one at a time.

Requirement: For a transaction these two phases must be mutually exclusively, that is, during locking phase unlocking phase must not start and during unlocking phase locking phase must not begin.

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

T1

read_lock (Y);
read_item (Y);
unlock (Y);
write_lock (X);
read_item (X);
X:=X+Y;
write_item (X); write_item (Y);
unlock (X);

T2

read_lock (X);
read_item (X);
unlock (X);
Write_lock (Y);
read_item (Y);
Y:=X+Y;
unlock (Y);

Result

Initial values: X=20; Y=30
Result of serial execution
T₁ followed by T₂
X=50, Y=80.
Result of serial execution
T₂ followed by T₁


X=70, Y=50

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

T1	T2	<u>Result</u>
read_lock (Y); read_item (Y); unlock (Y);	read_lock (X); read_item (X); unlock (X); write_lock (Y); read_item (Y); Y:=X+Y; write_item (Y); unlock (Y);	X=50; Y=50 Nonserializable because it. violated two-phase policy.
write_lock (X); read_item (X); X:=X+Y; write_item (X); unlock (X);		

Time



Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

T'1

read_lock (Y);
read_item (Y);
write_lock (X);
unlock (Y);
read_item (X);
X:=X+Y;
write_item (X); write_item (Y);
unlock (X);

T'2

read_lock (X);
read_item (X);
write_lock (Y);
unlock (X);
read_item (Y);
Y:=X+Y;
unlock (Y);

T_1 and T_2 follow two-phase policy but they are subject to deadlock, which must be dealt with.

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

Two-phase policy generates two locking algorithms (a) Basic and (b) Conservative.

Conservative: Prevents deadlock by locking all desired data items before transaction begins execution.

Basic: Transaction locks data items incrementally. This may cause deadlock which is dealt with.

Strict: A more stricter version of Basic algorithm where unlocking is performed after a transaction terminates (commits or aborts and rolled-back). This is the most commonly used two-phase locking algorithm.

Database Concurrency Control

Dealing with Deadlock and Starvation

Deadlock

T'₁

read_lock (Y);
read_item (Y);

write_lock (X);
(waits for X)

T'₂

read_lock (X);
read_item (Y);

write_lock (Y);
(waits for Y)

T'₁ and T'₂ did follow two-phase policy but they are deadlock

Deadlock (T'₁ and T'₂)

Database Concurrency Control

Dealing with Deadlock and Starvation

Deadlock prevention

A transaction locks all data items it refers to before it begins execution. This way of locking prevents deadlock since a transaction never waits for a data item. The *conservative* two-phase locking uses this approach.

Database Concurrency Control

Dealing with Deadlock and Starvation

Deadlock detection and resolution

In this approach, deadlocks are allowed to happen. The scheduler maintains a *wait-for-graph* for detecting cycle. If a *cycle* exists, then one transaction involved in the cycle is selected (victim) and rolled-back.

A *wait-for-graph* is created using the lock table. As soon as a transaction is blocked, it is added to the graph. When a chain like: T_i waits for T_j waits for T_k waits for T_i or T_j occurs, then this creates a cycle. One of the transaction of the cycle is selected and rolled back.

Wait-for graph

T'₁

read_lock (Y);

read_lock (X);

write_lock (X);
(waits for X)

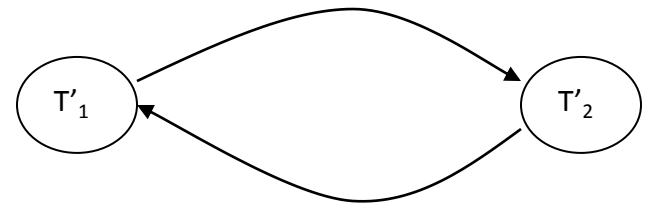
T'₂

read_item (Y);

read_item (Y);

write_lock (Y);
(waits for Y)

a) Partial schedule of T'₁ and T'₂



b) wait-for graph

Database Concurrency Control

Dealing with Deadlock and Starvation

Deadlock avoidance

There are many variations of two-phase locking algorithm. Some avoid deadlock by not letting the cycle to complete. That is as soon as the algorithm discovers that blocking a transaction is likely to create a cycle, it rolls back the transaction. **Wound-Wait** and **Wait-Die** algorithms use timestamps to avoid deadlocks by rolling-back victim.

Database Concurrency Control

Dealing with Deadlock and Starvation

Starvation

Starvation occurs when a particular transaction consistently waits or restarted and never gets a chance to proceed further. In a deadlock resolution it is possible that the same transaction may consistently be selected as victim and rolled-back. This limitation is inherent in all priority based scheduling mechanisms. In *Wound-Wait* scheme a younger transaction may always be wounded (aborted) by a long running older transaction which may create starvation.

Database Concurrency Control

Timestamp based concurrency control algorithm

Timestamp

A monotonically increasing variable (integer) indicating the age of an operation or a transaction. A larger timestamp value indicates a more recent event or operation.

Timestamp-based algorithm uses *timestamp* to serialize the execution of concurrent transactions.

Timestamps

- The algorithm associates with each database item X with two timestamp (TS) values:
 - $Read_TS(X)$: The **read timestamp** of item X ; this is the largest timestamp among all the timestamps of transactions that have successfully read item X .
 - $Write_TS(X)$: The **write timestamp** of item X ; this is the largest timestamp among all the timestamps of transactions that have successfully written item X .

Database Concurrency Control

Timestamp based concurrency control algorithm

Basic Timestamp Ordering

1. Transaction T issues a $write_item(X)$ operation:
 - a. If $read_TS(X) > TS(T)$ or if $write_TS(X) > TS(T)$, then a younger transaction has already read the data item so abort and roll-back T and reject the operation.
 - b. If the condition in part (a) does not exist, then execute $write_item(X)$ of T and set $write_TS(X)$ to $TS(T)$.
2. Transaction T issues a $read_item(X)$ operation:
 - a. If $write_TS(X) > TS(T)$, then a younger transaction has already written to the data item so abort and roll-back T and reject the operation.
 - b. If $write_TS(X) \leq TS(T)$, then execute $read_item(X)$ of T and set $read_TS(X)$ to the larger of $TS(T)$ and the current $read_TS(X)$.

Ex: Three transactions executing under a timestamp-based scheduler

T1	T2	T3	A	B	C
200	150	175	RT = 0 WT = 0	RT = 0 WT = 0	RT = 0 WT = 0
r1(B) w1(B) w1(A)	r2(A) w2(C) Abort	r3(C) w3(A)	RT = 150 WT = 200	RT = 200 WT = 200	RT = 175

Why T2 must be aborted (rolled-back)?

Database Concurrency Control

Timestamp based concurrency control algorithm

Strict Timestamp Ordering

1. Transaction T issues a *write_item(X)* operation:
 - a. If $TS(T) > read_TS(X)$, then delay T until the transaction T' that wrote or read X has terminated (committed or aborted).
2. Transaction T issues a *read_item(X)* operation:
 - a. If $TS(T) > write_TS(X)$, then delay T until the transaction T' that wrote or read X has terminated (committed or aborted).

Database Concurrency Control

Timestamp based concurrency control algorithm

Thomas's Write Rule

1. If $read_TS(X) > TS(T)$ then abort and roll-back T and reject the operation.
2. If $write_TS(X) > TS(T)$, then just ignore the write operation and continue execution. This is because the most recent writes counts in case of two consecutive writes.
3. If the conditions given in 1 and 2 above do not occur, then execute $write_item(X)$ of T and set $write_TS(X)$ to $TS(T)$.

Database Concurrency Control

Multiversion concurrency control techniques

Concept

This approach maintains a number of versions of a data item and allocates the right version to a read operation of a transaction. Thus unlike other mechanisms a read operation in this mechanism is never rejected.

Side effect: Significantly more storage (RAM and disk) is required to maintain multiple versions. To check unlimited growth of versions, a garbage collection is run when some criteria is satisfied.

Database Concurrency Control

Multiversion technique based on timestamp ordering

Assume X_1, X_2, \dots, X_n are the versions of a data item X created by a write operation of transactions. With each X_i a *read_TS* (read timestamp) and a *write_TS* (write timestamp) are associated.

read_TS(X_i): The read timestamp of X_i is the largest of all the timestamps of transactions that have successfully read version X_i .

write_TS(X_i): The write timestamp of X_i that wrote the value of version X_i .

A new version of X_i is created only by a write operation.

Database Concurrency Control

Multiversion technique based on timestamp ordering

To ensure serializability, the following two rules are used.

If transaction T issues $write_item(X)$ and version i of X has the highest $write_TS(X_i)$ of all versions of X that is also less than or equal to $TS(T)$, and $read_TS(X_i) > TS(T)$, then abort and roll-back T ; otherwise create a new version X_i and $read_TS(X) = write_TS(X_j) = TS(T)$.

If transaction T issues $read_item(X)$, find the version i of X that has the highest $write_TS(X_i)$ of all versions of X that is also less than or equal to $TS(T)$, then return the value of X_i to T , and set the value of $read_TS(X_i)$ to the largest of $TS(T)$ and the current $read_TS(X_i)$.

Rule 2 guarantees that a read will never be rejected.

Ex: Execution of transactions using multiversion concurrency control

T1	T2	T3	T4	A ₀	A ₁₅₀	A ₂₀₀
150	200	175	225			
r1(A) w1(A)	r2(A) w2(A)	r3(A)	r4(A)	read	Create Read read	Create read

Note: T3 does not have to abort, because it can read an earlier version of A.

Database Concurrency Control

Multiversion Two-Phase Locking Using Certify Locks

Concept

Allow a transaction T' to read a data item X while it is write locked by a conflicting transaction T .

This is accomplished by maintaining two versions of each data item X where one version must always have been written by some committed transaction. This means a write operation always creates a new version of X .

Database Concurrency Control

Multiversion Two-Phase Locking Using Certify Locks

Steps

1. X is the committed version of a data item.
2. T creates a second version X' after obtaining a write lock on X .
3. Other transactions continue to read X .
4. T is ready to commit so it obtains a ***certify lock*** on X' .
5. The committed version X becomes X' .
6. T releases its ***certify lock*** on X' , which is X now.

Compatibility tables for

	Read	Write
Read	yes	no
Write	no	no

read/write locking scheme

	Read	Write	Certify
Read	yes	no	no
Write	no	no	no
Certify	no	no	no

read/write/certify locking scheme

Database Concurrency Control

Multiversion Two-Phase Locking Using Certify Locks

Note

In multiversion 2PL, read and write operations from conflicting transactions can be processed concurrently. This improves concurrency but it may delay transaction commit because of obtaining certify locks on all its writes. It avoids cascading abort but like strict two-phase locking scheme conflicting transactions may get deadlocked.

Database Concurrency Control

Validation (Optimistic) Concurrency Control Schemes

In this technique only at the time of commit serializability is checked and transactions are aborted in case of non-serializable schedules.

Three phases:

Read phase: A transaction can read values of committed data items. However, updates are applied only to *local copies* (versions) of the data items (in database cache).

Database Concurrency Control

Validation (Optimistic) Concurrency Control Schemes

Validation phase: Serializability is checked before transactions write their updates to the database.

This phase for T_i checks that, for each transaction T_j that is either committed or is in its validation phase, one of the following conditions holds:

1. T_j completes its write phase before T_i starts its read phase.
2. T_i starts its write phase after T_j completes its write phase, and the *read_set* of T_i has no items in common with the *write_set* of T_j
3. Both the *read_set* and *write_set* of T_i have no items in common with the *write_set* of T_j , and T_j completes its read phase.

Database Concurrency Control

Validation (Optimistic) Concurrency Control Schemes

When validating T_i , the first condition is checked first for each transaction T_j , since (1) is the simplest condition to check. If (1) is false then (2) is checked and if (2) is false then (3) is checked. If none of these conditions holds, the validation fails and T_i is aborted.

Write phase: On a successful validation transactions' updates are applied to the database; otherwise, transactions are restarted.

Database Concurrency Control

Granularity of data items and Multiple Granularity Locking

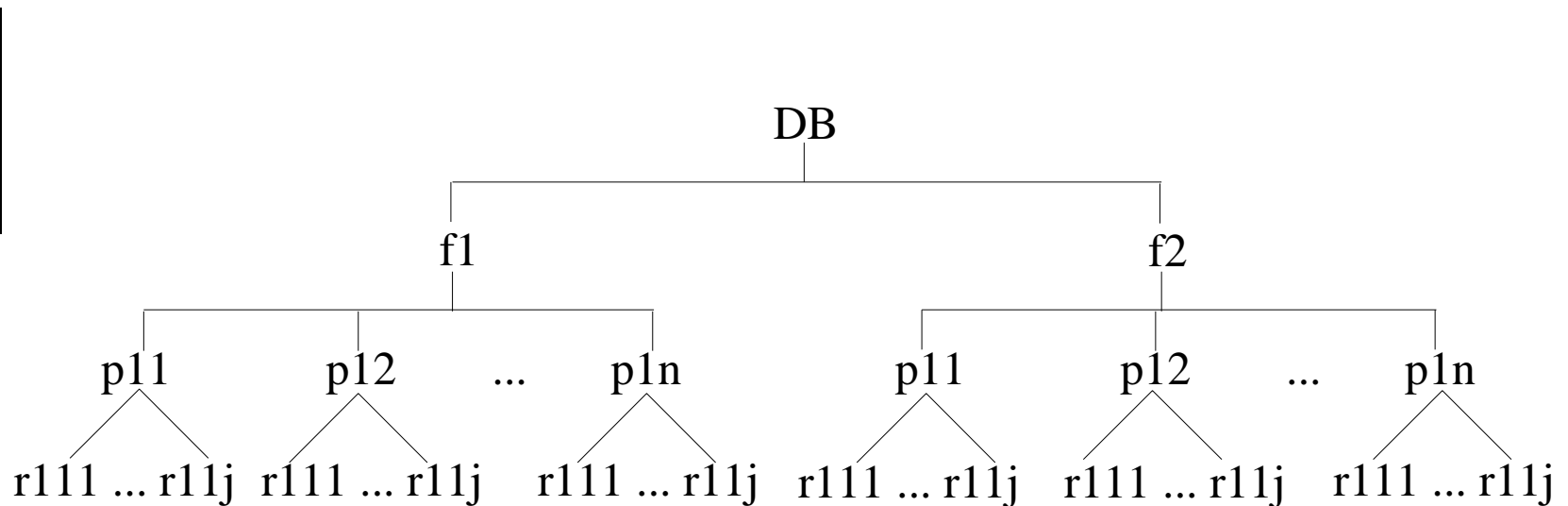
A lockable unit of data defines its *granularity*. Granularity can be coarse (entire database) or it can be fine (a tuple or an attribute of a relation). Data item granularity significantly affects concurrency control performance. Thus, the degree of concurrency is low for coarse granularity and high for fine granularity. Example of data item granularity:

1. A field of a database record (an attribute of a tuple).
2. A database record (a tuple or a relation).
3. A disk block.
4. An entire file.
5. The entire database.

Database Concurrency Control

Granularity of data items and Multiple Granularity Locking

The following diagram illustrates a hierarchy of granularity from coarse (database) to fine (record).



Database Concurrency Control

Granularity of data items and Multiple Granularity Locking

To manage such hierarchy, in addition to read and write, three additional locking modes, called *intention lock modes* are defined:

Intention-shared (IS): indicates that a shared lock(s) will be requested on some descendent nodes(s).

Intention-exclusive (IX): indicates that an exclusive lock(s) will be requested on some descendent nodes(s).

Shared-intention-exclusive (SIX): indicates that the current node is locked in shared mode but an exclusive lock(s) will be requested on some descendent nodes(s).

Database Concurrency Control

Granularity of data items and Multiple Granularity Locking

These locks are applied using the following compatibility matrix:

	IS	IX	S	SIX	X
IS	yes	yes	yes	yes	no
IX	yes	yes	no	no	no
S	yes	no	yes	no	no
SIX	yes	no	no	no	no
X	no	no	no	no	no

Database Concurrency Control

Granularity of data items and Multiple Granularity Locking

The set of rules which must be followed for producing serializable schedule are

1. The lock compatibility must adhered to.
2. The root of the tree must be locked first, in any mode..
3. A node N can be locked by a transaction T in S or IX mode only if the parent node is already locked by T in either IS or IX mode.
4. A node N can be locked by T in X, IX, or SIX mode only if the parent of N is already locked by T in either IX or SIX mode.
5. T can lock a node only if it has not unlocked any node (to enforce 2PL policy).
6. T can unlock a node, N , only if none of the children of N are currently locked by T .

Database Concurrency Control

Granularity of data items and Multiple Granularity Locking

An example of a serializable execution:

T₁
IX(db)
IX(f₁)

T₂

IX(db)

T₃

IS(db)
IS(f₁)
IS(p₁₁)

IX(p₁₁)
X(r₁₁₁)

IX(f₁)
X(p₁₂)

S(r_{11j})

IX(f₂)
IX(p₂₁)
IX(r₂₁₁)
Unlock (r₂₁₁)
Unlock (p₂₁)
Unlock (f₂)

S(f₂)

T₁ want to update r₁₁₁, r₁₁₂

T₂ want to update page p₁₂

T₂ wants to update r_{11j} and f₂

Database Concurrency Control

Granularity of data items and Multiple Granularity Locking

An example of a serializable execution (continued):

T₁

unlock(r₁₁₁)
unlock(p₁₁)
unlock(f₁)
unlock(db)

T₂

unlock(p₁₂)
unlock(f₁)
unlock(db)

T₃

unlock (r_{111j})
unlock (p₁₁)
unlock (f₁)
unlock(f₂)
unlock(db)