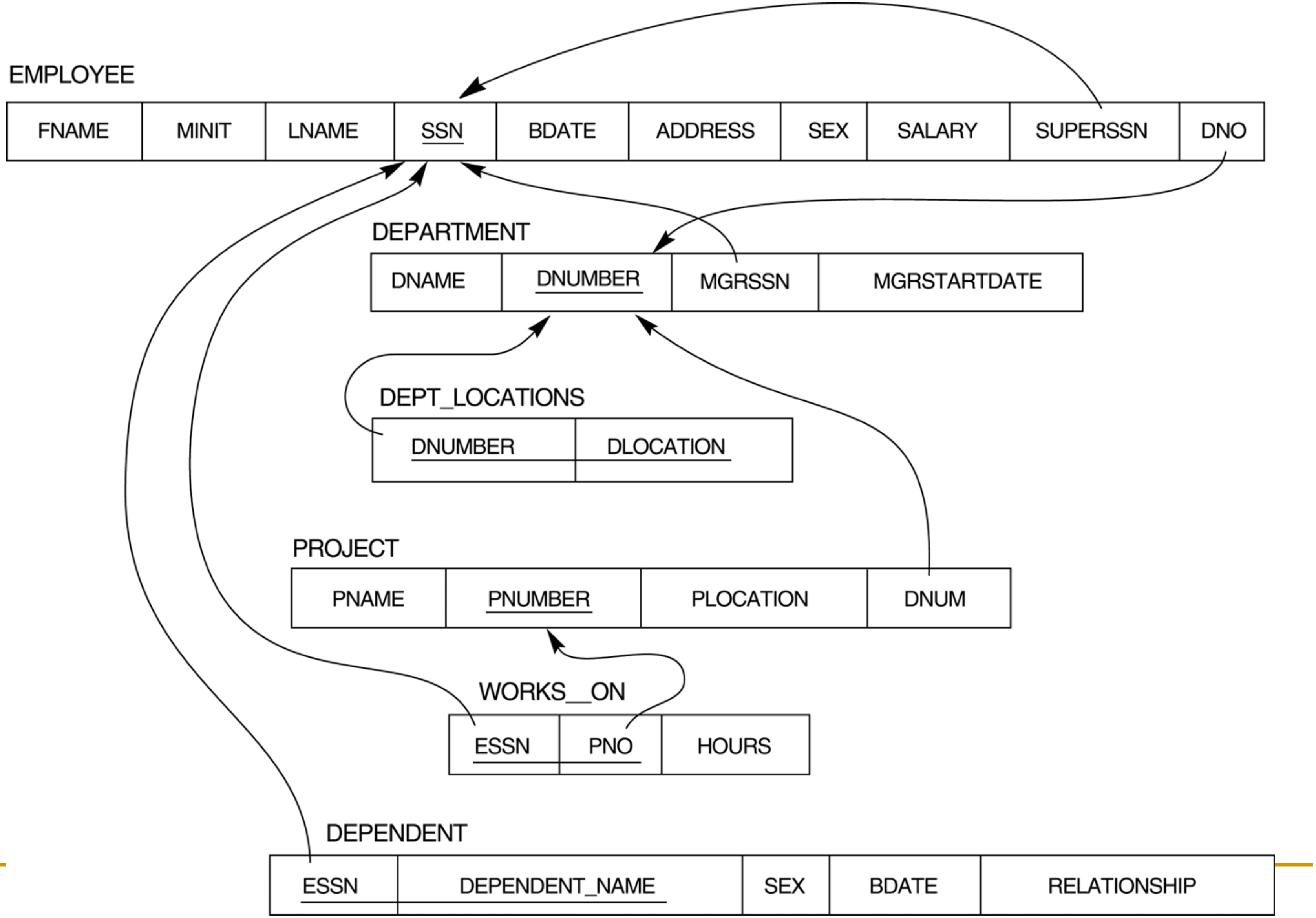
Chapter 6:

SQL (Structured Query Language)

Contents

1	**The COMPANY Database**
2	SQL developments: an overview
3	DDL: Create, Alter, Drop
4	DML: select, insert, update, delete
5	DCL: commit, rollback, grant, revoke
6	Trigger, Store Procedure, Function & Cursor in Oracle

The COMPANY Database



EMPLOYEE

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO

DEPARTMENT

DNAME	DNUMBER	MGRSSN	MGRSTARTDATE

DEPT_LOCATIONS

DNUMBER	DLOCATION

PROJECT

PNAME	PNUMBER	PLOCATION	DNUM

WORKS__ON

ESSN	PNO	HOURS

DEPENDENT

ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP

Contents

1	The COMPANY Database
2	**SQL developments: an overview**
3	DDL: Create, Alter, Drop
4	DML: select, insert, update, delete
5	DCL: commit, rollback, grant, revoke
6	Trigger, Store Procedure, Function & Cursor in Oracle

SQL developments: an overview

- In 1986, ANSI and ISO published an initial standard for SQL: SQL-86 or SQL1
- In 1992, first major revision to ISO standard occurred, referred to as SQL2 or SQL-92
- In 1999, SQL-99 (SQL3) was released with support for object-oriented data management
- In late 2003, SQL-2003 was released
- Now: SQL-2006 was published

SQL developments: an overview

(http://en.wikipedia.org/wiki/SQL)

Year	Name	Alias	Comments
1986	SQL-86	SQL-87	First published by ANSI. Ratified by ISO in 1987
1989	SQL-89		Minor revision
1992	SQL-92	SQL2	Major revision (ISO 9075)
1999	SQL:1999	SQL3	Added regular expression matching, recursive queries, triggers, non-scalar types and some object-oriented features. (The last two are somewhat controversial and not yet widely supported)
2003	SQL:2003		Introduced XML-related features, *window functions*, standardized sequences and columns with auto-generated values (including identity-columns)
2006	SQL:2006		ISO/IEC 9075-14:2006 defines ways in which SQL can be used in conjunction with XML. It defines ways of importing and storing XML data in an SQL database, manipulating it within the database and publishing both XML and conventional SQL-data in XML form. In addition, it provides facilities that permit applications to integrate into their SQL code the use of XQuery, the XML Query Language published by the World Wide Web Consortium (W3C), to concurrently access ordinary SQL-data and XML documents

Basic SQL

- DDL: Data Definition Language
  - Create, Alter, Drop

- DML: Data Manipulation Language
  - Select, Insert, Update, Delete

- DCL: Data Control Language
  - Commit, Rollback, Grant, Revoke

Basic SQL

- **SQL**
  - **Structured Query Language**
  - Statements for data definitions, queries, and updates (both DDL and DML)
  - **Core specification**
  - Plus specialized **extensions**

Contents

1 The COMPANY Database

2 SQL developments: an overview

3 DDL: Create, Alter, Drop

4 DML: select, insert, update, delete

5 DCL: commit, rollback, grant, revoke

6 Trigger, Store Procedure, Function & Cursor in Oracle

DDL: Create, Alter, Drop

CREATE SCHEMA

- **SQL schema**
  - Identified by a **schema name**
  - Includes an **authorization identifier** and **descriptors** for each element
- Schema **elements** include
  - Tables, constraints, views, domains, and other constructs
- **Catalog**
  - Named collection of schemas in an SQL environment

DDL: Create, Alter, Drop

CREATE SCHEMA

- CREATE SCHEMA SchemaName AUTHORIZATION AuthorizationIdentifier;
- To create a relational database schema: started with SQL-92

```
CREATE SCHEMA Company AUTHORIZATION
   JSmith;
```

- Homework: SCHEMA in ORACLE

DDL: Create, Alter, Drop

CREATE TABLE

- CREATE TABLE SchemaName.TableName …

or

- CREATE TABLE TableName …

DDL: Create, Alter, Drop

CREATE TABLE

```
CREATE TABLE TableName
{(colName dataType [NOT NULL] [UNIQUE]
[DEFAULT defaultOption]
[CHECK searchCondition] [,...]}

[PRIMARY KEY (listOfColumns),]
{[UNIQUE (listOfColumns),] […,]}
{[FOREIGN KEY (listOfFKColumns)
  REFERENCES ParentTableName [(listOfCKColumns)],
  [ON UPDATE referentialAction]
  [ON DELETE referentialAction ]] [,…]}
{[CHECK (searchCondition)] [,…] })
```

DDL: Create, Alter, Drop

CREATE TABLE

- **Base tables (base relations)**
  - Relation and its tuples are actually created and stored as a file by the DBMS.
- **Virtual relations**
  - Created through the `CREATE VIEW` statement.
- Some foreign keys may cause errors
  - Specified either via:
    - Circular references
    - Or because they refer to a table that has not yet been created

Attribute Data Types and Domains in SQL

- Basic **data types**
  - **Numeric** data types
    - Integer numbers: `INTEGER`, `INT`, and `SMALLINT`
    - Floating-point (real) numbers: `FLOAT` or `REAL`, and `DOUBLE PRECISION`
  - **Character-string** data types
    - Fixed length: `CHAR(`*n*`)`, `CHARACTER(`*n*`)`
    - Varying length: `VARCHAR(`*n*`)`, `CHAR VARYING(`*n*`)`, `CHARACTER VARYING(`*n*`)`

Attribute Data Types and Domains in SQL

- **Bit-string** data types
  - Fixed length: `BIT(`*n*`)`
  - Varying length: `BIT VARYING(`*n*`)`
  - Ex: B’1001’
- **Boolean** data type
  - Values of `TRUE` or `FALSE` or `NULL`
- **DATE** data type
  - Ten positions
  - Components are `YEAR`, `MONTH`, and `DAY` in the form YYYY-MM-DD

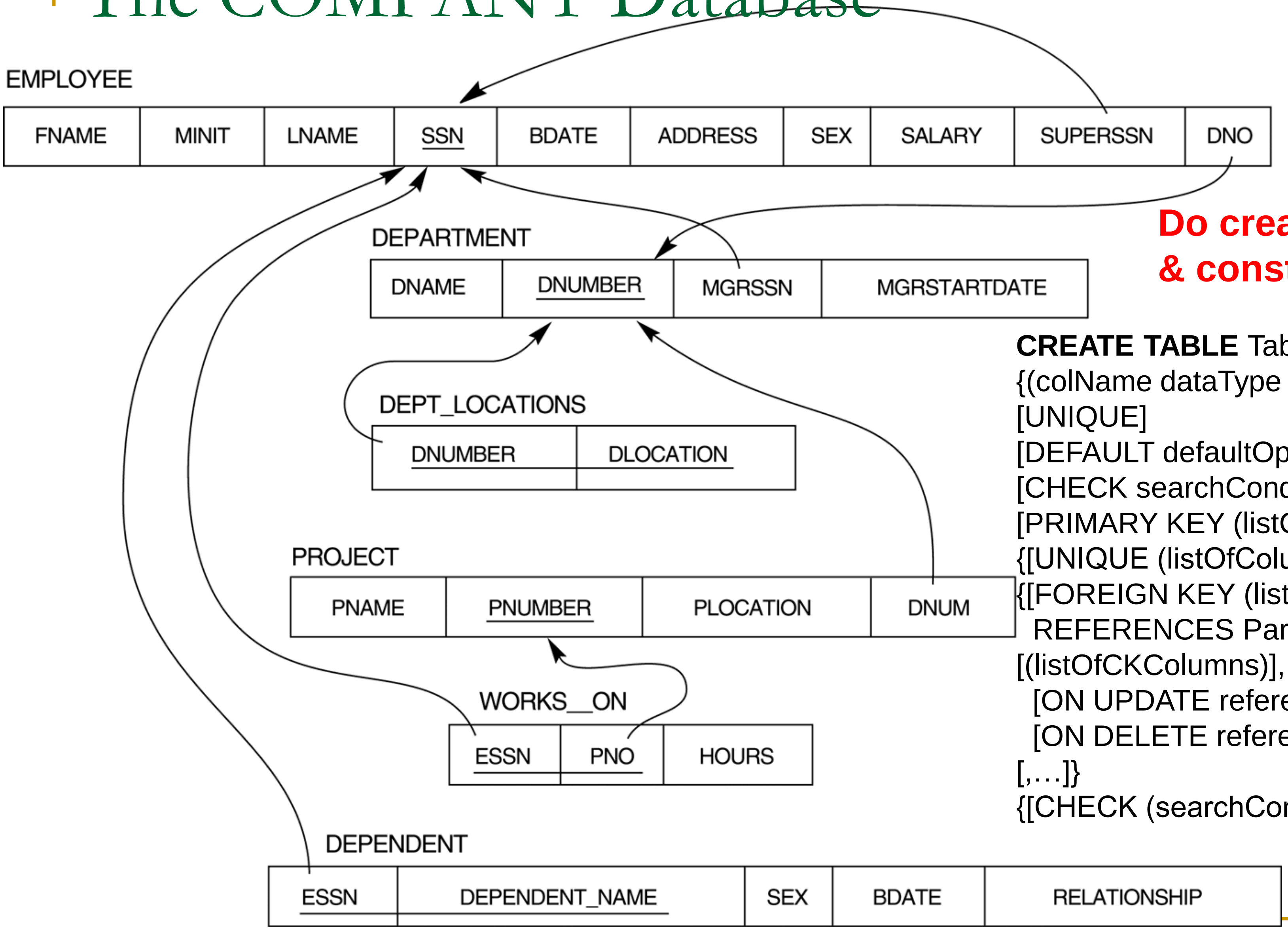
Attribute Data Types and Domains in SQL

- Additional data types
  - **Timestamp** data type (`TIMESTAMP`)
    - Includes the `DATE` and `TIME` fields
    - Plus a minimum of six positions for decimal fractions of seconds
    - Optional `WITH TIME ZONE` qualifier
  - **`INTERVAL`** data type
    - Specifies a relative value that can be used to increment or decrement an absolute value of a date, time, or timestamp

Attribute Data Types and Domains in SQL

- Domain
  - Name used with the attribute specification
  - Makes it easier to change the data type for a domain that is used by numerous attributes
  - Improves schema readability
  - CREATE DOMAIN DomainName AS DataType [CHECK conditions];
  - Example:
    - `CREATE DOMAIN SSN_TYPE AS CHAR(9);`

The COMPANY Database



EMPLOYEE

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO

DEPARTMENT

DNAME	DNUMBER	MGRSSN	MGRSTARTDATE

DEPT_LOCATIONS

DNUMBER	DLOCATION

PROJECT

PNAME	PNUMBER	PLOCATION	DNUM

WORKS__ON

ESSN	PNO	HOURS

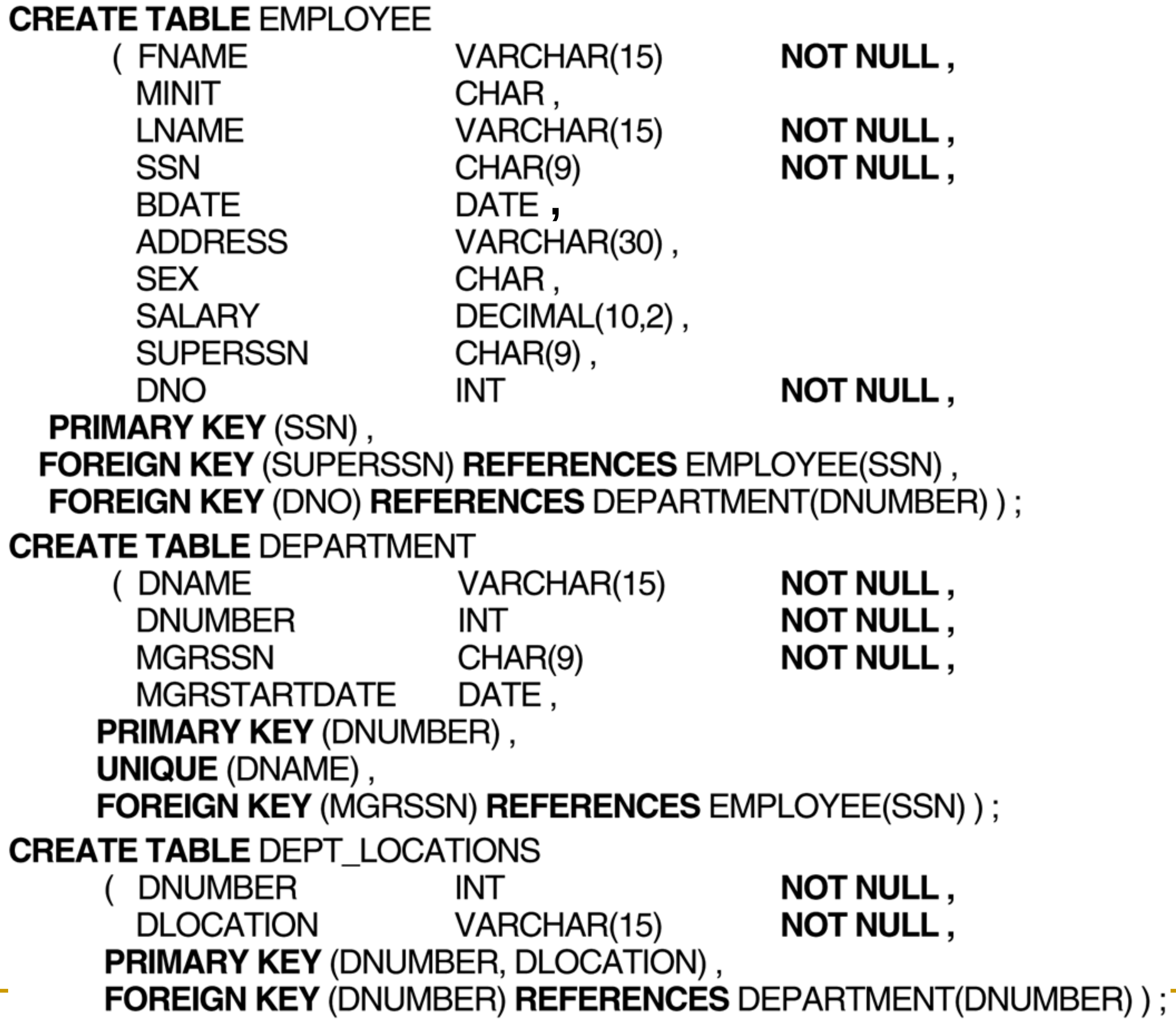
DEPENDENT

ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP

Do create tables & constraints !!

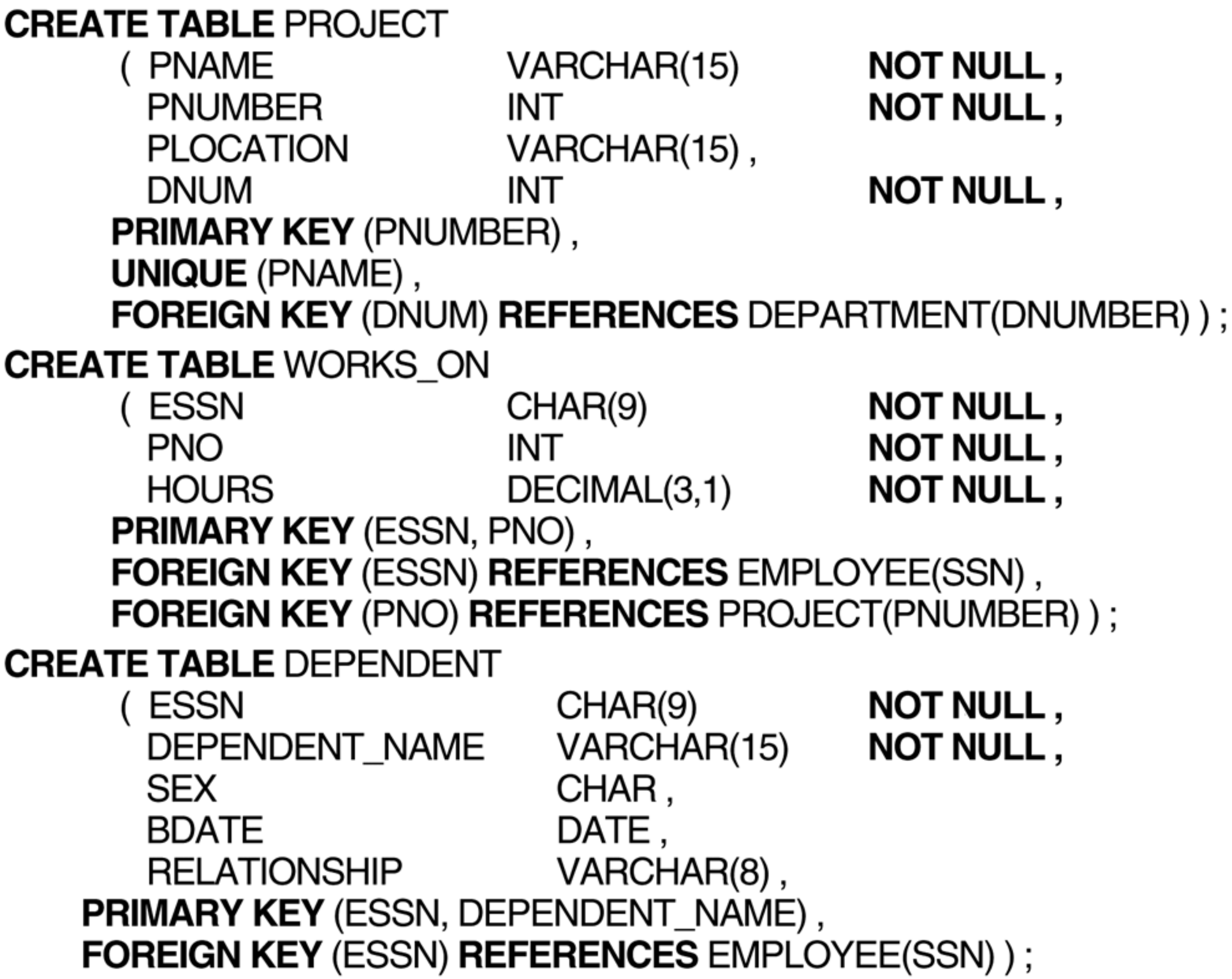
CREATE TABLE TableName
{(colName dataType [NOT NULL]
[UNIQUE]
[DEFAULT defaultOption]
[CHECK searchCondition] [,...]}
[PRIMARY KEY (listOfColumns),]
{[UNIQUE (listOfColumns),] [,…,]}
{[FOREIGN KEY (listOfFKColumns)
REFERENCES ParentTableName
[(listOfCKColumns)],
[ON UPDATE referentialAction]
[ON DELETE referentialAction]]
[,…]}
{[CHECK (searchCondition)] [,…] })

Defining the COMPANY DB schema (1)



```
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) ,
            SUPERSSN           CHAR(9) ,
            DNO                INT                NOT NULL ,
   PRIMARY KEY (SSN) ,
  FOREIGN KEY (SUPERSSN) REFERENCES EMPLOYEE(SSN) ,
   FOREIGN KEY (DNO) REFERENCES DEPARTMENT(DNUMBER) ) ;
CREATE TABLE DEPARTMENT
          ( DNAME              VARCHAR(15)        NOT NULL ,
            DNUMBER            INT                NOT NULL ,
            MGRSSN             CHAR(9)            NOT NULL ,
            MGRSTARTDATE       DATE ,
        PRIMARY KEY (DNUMBER) ,
        UNIQUE (DNAME) ,
        FOREIGN KEY (MGRSSN) 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) ) ;
```

Defining the COMPANY DB schema (2)



```
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) ) ;
```

Specifying Constraints in SQL

- Basic constraints:
  - Key and referential integrity constraints
  - Restrictions on attribute domains and NULLs
  - Constraints on individual tuples within a relation

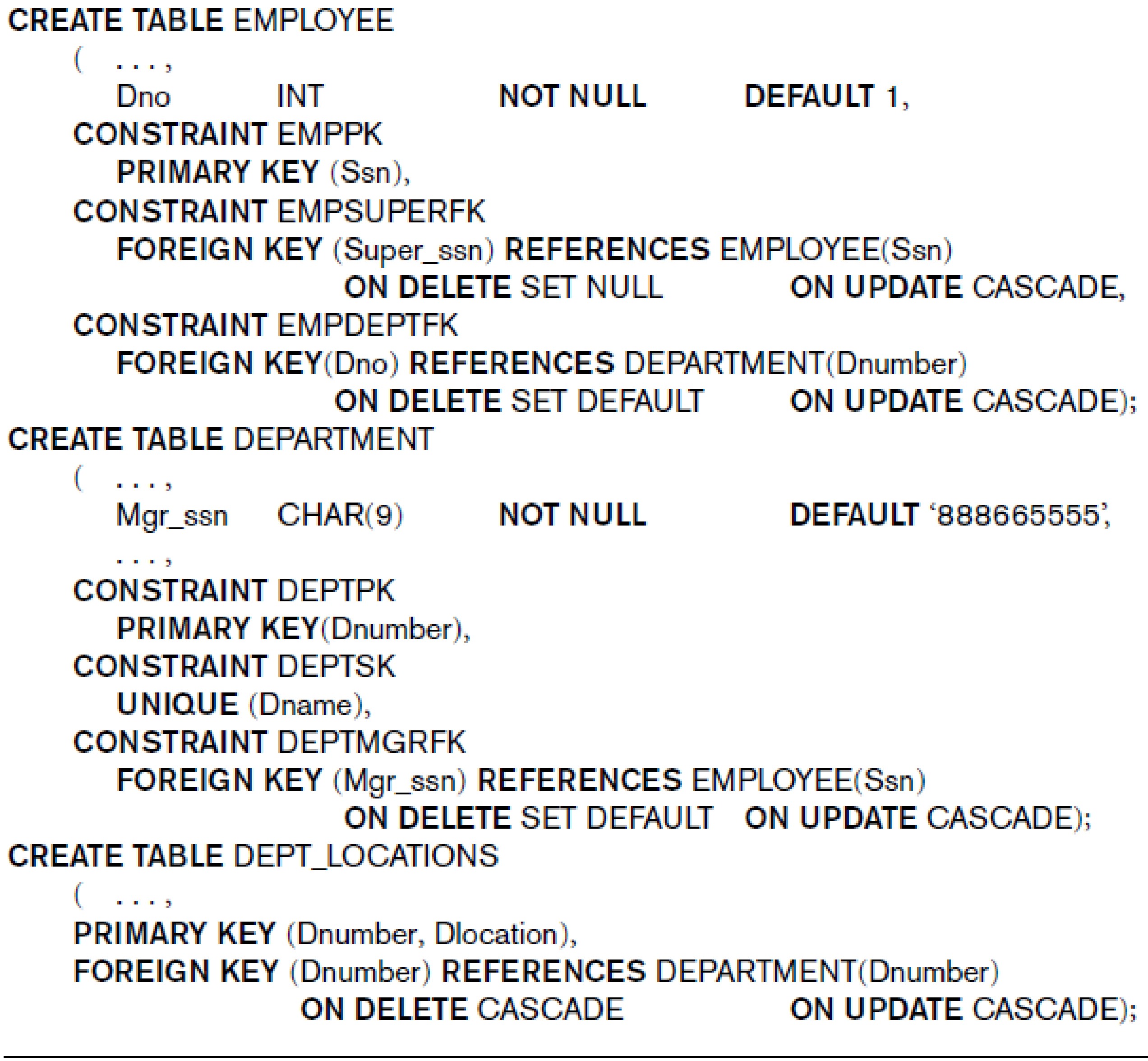
Specifying Attribute Constraints and Attribute Defaults

- NOT NULL
  - NULL is not permitted for a particular attribute
- Default values
  - DEFAULT <value> can be specified for an attribute
  - If no default clause is specified, the default value is NULL for attributes that do not have the NOT NULL constraint
    - If NOT NULL option is specified on attribute A and no value is specified as inserting a tupe r(…A…) ?
- CHECK clause:

DNUMBER INT NOT NULL CHECK (DNUMBER>0 AND DNUMBER<21);

  - CREATE DOMAIN can also be used in conjunction with the CHECK clause:

CREATE DOMAIN D_NUM AS INTEGER CHECK (D_NUM>0 AND D_NUM<21);



```
CREATE TABLE EMPLOYEE
      (   . . . ,
          Dno          INT                  NOT NULL          DEFAULT 1,
      CONSTRAINT EMPPK
          PRIMARY KEY (Ssn),
      CONSTRAINT EMPSUPERFK
          FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(Ssn)
                              ON DELETE SET NULL              ON UPDATE CASCADE,
      CONSTRAINT EMPDEPTFK
          FOREIGN KEY(Dno) REFERENCES DEPARTMENT(Dnumber)
                              ON DELETE SET DEFAULT           ON UPDATE CASCADE);
CREATE TABLE DEPARTMENT
      (   . . . ,
          Mgr_ssn      CHAR(9)              NOT NULL              DEFAULT '888665555',
          . . . ,
      CONSTRAINT DEPTPK
          PRIMARY KEY(Dnumber),
      CONSTRAINT DEPTSK
          UNIQUE (Dname),
      CONSTRAINT DEPTMGRFK
          FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn)
                              ON DELETE SET DEFAULT   ON UPDATE CASCADE);
CREATE TABLE DEPT_LOCATIONS
      (   . . . ,
      PRIMARY KEY (Dnumber, Dlocation),
      FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber)
                           ON DELETE CASCADE                  ON UPDATE CASCADE);
```

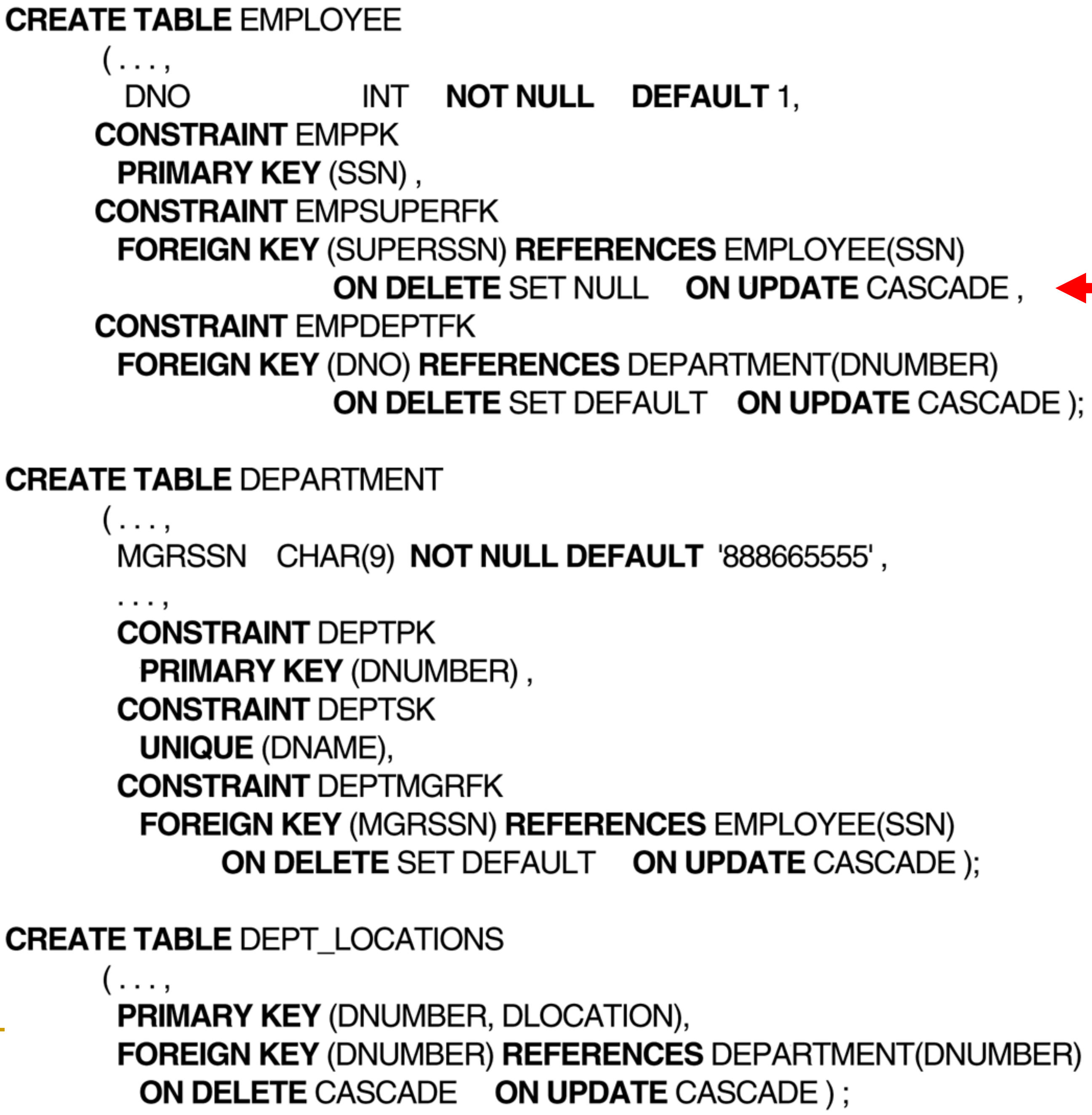
Figure 4.2
Example illustrating how default attribute values and referential integrity triggered actions are specified in SQL.

Specifying Key and Referential Integrity Constraints

- **`PRIMARY KEY`** clause
  - Specifies one or more attributes that make up the primary key of a relation.
  - `Dnumber INT PRIMARY KEY;`
- **`UNIQUE`** clause
  - Specifies alternate (secondary) keys.
  - `Dname VARCHAR(15) UNIQUE;`

Specifying Key and Referential Integrity Constraints (cont'd.)

- **`FOREIGN KEY`** clause
  - Default operation: reject update on violation
  - Attach **referential triggered action** clause
    - Options include `SET NULL`, `CASCADE`, and `SET DEFAULT`
    - An option must be qualified with either `ON DELETE` or `ON UPDATE`



```
CREATE TABLE EMPLOYEE
        ( . . . ,
          DNO               INT    NOT NULL    DEFAULT 1,
        CONSTRAINT EMPPK
          PRIMARY KEY (SSN) ,
        CONSTRAINT EMPSUPERFK
          FOREIGN KEY (SUPERSSN) REFERENCES EMPLOYEE(SSN)
                        ON DELETE SET NULL    ON UPDATE CASCADE ,
        CONSTRAINT EMPDEPTFK
          FOREIGN KEY (DNO) REFERENCES DEPARTMENT(DNUMBER)
                        ON DELETE SET DEFAULT    ON UPDATE CASCADE );

CREATE TABLE DEPARTMENT
        ( . . . ,
          MGRSSN   CHAR(9)  NOT NULL DEFAULT  '888665555' ,
          . . . ,
          CONSTRAINT DEPTPK
            PRIMARY KEY (DNUMBER) ,
          CONSTRAINT DEPTSK
            UNIQUE (DNAME),
          CONSTRAINT DEPTMGRFK
            FOREIGN KEY (MGRSSN) REFERENCES EMPLOYEE(SSN)
                  ON DELETE SET DEFAULT    ON UPDATE CASCADE );

CREATE TABLE DEPT_LOCATIONS
        ( . . . ,
          PRIMARY KEY (DNUMBER, DLOCATION),
          FOREIGN KEY (DNUMBER) REFERENCES DEPARTMENT(DNUMBER)
            ON DELETE CASCADE    ON UPDATE CASCADE ) ;
```

An example

Specifying Constraints in SQL

- Giving names to constraints
  - This is optional.
  - Keyword **`CONSTRAINT`**
  - The name is unique within a particular DB schema.
  - Used to identify a particular constraint in case it must be dropped later and replaced with another one.

Specifying Constraints in SQL

- Specifying constraints on tuples using CHECK
  - Affected on each tuple individually as being inserted or modified (tuple-based constraints)
  - Department create date must be earlier than the manager’s start date:

    CHECK (DEPT_CREATE_DATE < MGRSTARTDATE);
  - More general constraints: CREATE ASSERTION

DDL: Create, Alter, Drop

DROP Command

- Used to drop ***named*** schema elements: tables, domains, constraints, and the schema itself
- Drop behavior options:
  - `CASCADE` and `RESTRICT`

```
DROP SCHEMA Company CASCADE;
```

or

```
DROP SCHEMA Company RESTRICT;
```

DDL: Create, Alter, Drop

DROP Command

- Drop a table:

  DROP TABLE Dependent CASCADE | RESTRICT;

  - RESTRICT option: dropped on if it is not referenced in any constraints or views.
  - CASCADE option: all such constraints and views that reference the table are dropped automatically from the schema along with the table itself.

- Similarly, we can drop constraints & domains.

DDL: Create, Alter, Drop

ALTER Command

- Base tables: adding or dropping a column or constraints, changing a column definition

  ```
  ALTER TABLE Company.Employee ADD Job VARCHAR(15);
  ```

  - Job value for each tuple: default clause or UPDATE command
  - What value does each tuple take wrt. the attribute Job if:

  ```
  ALTER TABLE Company.Employee ADD Job VARCHAR(15) NOT NULL;
  ```

DDL: Create, Alter, Drop

ALTER Command

- Drop a column: similarly to drop a table, CASCADE or RESTRICT option must be specified
  - CASCADE option: all constraints and views referencing the column are dropped along with the column
  - RESTRICT option: successful only if no constraints and views are referencing the column

```
ALTER TABLE Company.Employee DROP Address
   CASCADE;
```

Contents

1	The COMPANY Database
2	SQL developments: an overview
3	DDL: Create, Alter, Drop
4	**DML: select, insert, update, delete**
5	DCL: commit, rollback, grant, revoke
6	Trigger, Store Procedure, Function & Cursor in Oracle

DML: Select, Insert, Update, Delete

SELECT

- SQL has one basic statement for retrieving information from a database: the SELECT statement.
- This is *not the same as* the SELECT operation of the relational algebra.
- Important distinction between SQL and the formal relational model; SQL allows a table (relation) to have two or more tuples that are identical in all their attribute values.
- Hence, an SQL relation (table) is a *multi-set* (sometimes called a bag) of tuples; it *is not* a set of tuples.
- SQL relations can be constrained to be sets by specifying PRIMARY KEY or UNIQUE attributes, or by using the DISTINCT option in a query.

DML: Select, Insert, Update, Delete

SELECT

- *Basic form* of the SQL SELECT statement is called a *mapping* or a *SELECT-FROM-WHERE block*

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

  - <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

DML: Select, Insert, Update, Delete

SELECT

- Logical comparison operators
  - =, <, <=, >, >=, and <>
- **Projection attributes**
  - Attributes whose values are to be retrieved
- **Selection condition**
  - Boolean condition that must be true for any retrieved tuple

DML: Select, Insert, Update, Delete

SELECT

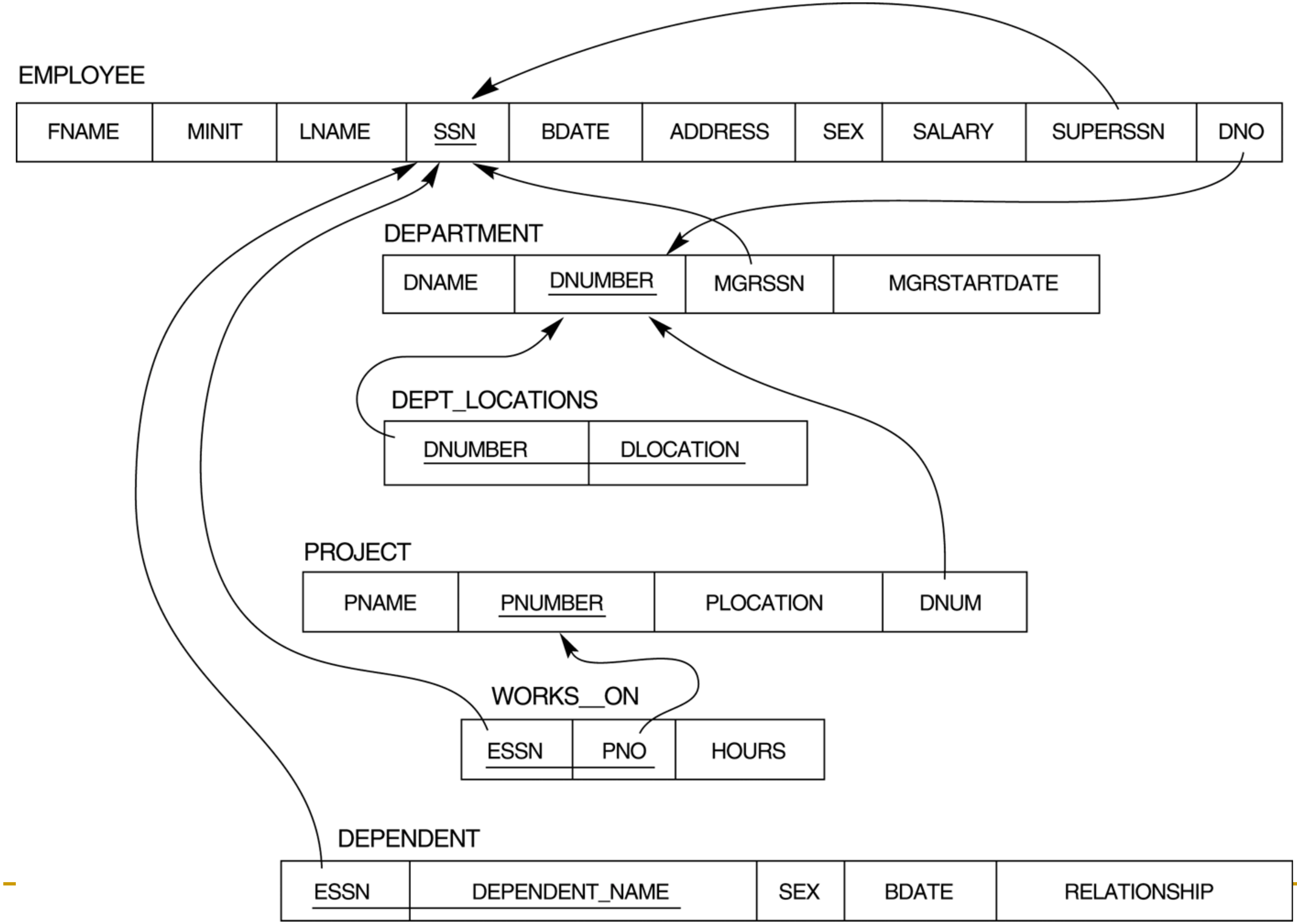
```
SELECT [DISTINCT | ALL]
    {* | [columnExpression [AS newName]] [,...] }
FROM TableName [alias] [, ...]
[WHERE condition]
[GROUP BY columnList]
[HAVING condition]
[ORDER BY columnList]
```

DML: Select, Insert, Update, Delete

SELECT

SELECT	Specifies which columns are to appear in output
FROM	Specifies table(s) to be used
WHERE	Filters rows
GROUP BY	Forms groups of rows with same column value
HAVING	Filters groups subject to some condition
ORDER BY	Specifies the order of the output

The COMPANY Database



EMPLOYEE

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO

DEPARTMENT

DNAME	DNUMBER	MGRSSN	MGRSTARTDATE

DEPT_LOCATIONS

DNUMBER	DLOCATION

PROJECT

PNAME	PNUMBER	PLOCATION	DNUM

WORKS__ON

ESSN	PNO	HOURS

DEPENDENT

ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP

DML: Select, Insert, Update, Delete

SELECT

- Basic SQL queries correspond to using the SELECT, PROJECT, and JOIN operations of the relational algebra
- 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**.

DML: Select, Insert, Update, Delete

SELECT

- 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).

DML: Select, Insert, Update, Delete

SELECT

- 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

DML: Select, Insert, Update, Delete

SELECT

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

- There are *2* 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

Ambiguous Attribute Names

- In SQL, we can use the same name for attributes as long as the attributes are in *different relations*. Query referring to attributes with the same name **must** *qualify* the attribute name with the relation name by *prefixing* the relation name to the attribute name
- Examples:

  DEPARTMENT.DNUMBER, DEPT_LOCATIONS.DNUMBER

Aliases

- Some queries need to refer to the same relation twice: *aliases* are given to the relation name
- Query 3: For each employee, retrieve the employee's name, and the name of his or her immediate supervisor.

```
Q3:  SELECT        E.FNAME, E.LNAME, S.FNAME, S.LNAME
     FROM          EMPLOYEE E, EMPLOYEE S
     WHERE         E.SUPERSSN=S.SSN;
```

  - 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

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

```
Q4: SELECT      E.FNAME, E.LNAME, S.FNAME,
                S.LNAME
      FROM      EMPLOYEE AS E, EMPLOYEE AS S
      WHERE     E.SUPERSSN=S.SSN;
```

- Renaming using aliases:

```
EMPLOYEE AS E(FN, MI, LN, SSN, BD, ADDR, SEX,
   SAL, SSSN, DNO)
```

(in the FROM clause)

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 5: Retrieve the SSN values for all employees.

```
Q5: SELECT      SSN
    FROM        EMPLOYEE;
```

Unspecified WHERE-clause

- 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.
- Example:

```
Q6:  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 ASTERISK (*)

- An asterisk (*) stands for *all the attributes.*
- Examples:

```
Q7: SELECT      *
    FROM        EMPLOYEE
    WHERE       DNO=5;

Q8: 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 in a query result.* To eliminate duplicate tuples, use the keyword **DISTINCT.**
- For example, the result of Q9 may have duplicate SALARY values, but Q9A's

```
Q9:     SELECT   SALARY
        FROM     EMPLOYEE;

Q9A:    SELECT   DISTINCT SALARY
        FROM     EMPLOYEE;
```

Set Operations

- Set union **(UNION)**, set difference (**EXCEPT)** and set 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.***
- UNION ALL, EXCEPT ALL, INTERSECT ALL ??

Set Operations

- Query 10: 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.

```
Q10:(SELECT      DISTINCT PNUMBER
    FROM         PROJECT, DEPARTMENT, EMPLOYEE
    WHERE        DNUM=DNUMBER AND MGRSSN=SSN
                      AND LNAME='Smith')
    UNION
    (SELECT      DISTINCT PNUMBER
    FROM         PROJECT, WORKS_ON, EMPLOYEE
    WHERE        PNUMBER=PNO AND ESSN=SSN AND
                      LNAME='Smith');
```

Substring pattern matching and arithmetic operators

- Two reserved characters: % and _

```
Q11:    SELECT *
        FROM     Employee
        WHERE    Address LIKE ‘%HCMC%’;

Q12:    SELECT *
        FROM   Employee
        WHERE  BDate LIKE ‘_ _8_ _ _ _ _ _ _’;
```

Substring pattern matching and arithmetic operators

- Standard arithmetic operators: +, -, *, /
- Query 13: show the resulting salaries if every employee working on “ProductX” is given 10% raise

```
Q13: SELECT       FNAME, LNAME, 1.1*Salary AS INC_SAL
     FROM         Employee, Works_on, Project
     WHERE        SSN=ESSN AND PNO=PNUMBER AND
                       PNAME=‘ProductX’;
```

NULL & 3-valued logic

AND	True	False	Unknown
True	T	F	U
False	F	F	F
Unknown	U	F	U

OR	True	False	Unknown
True	T	T	T
False	T	F	U
Unknown	T	U	U

NOT	
True	F
False	T
Unknown	U

SELECT * FROM Employee WHERE SuperSSN IS NULL;

SELECT * FROM Employee WHERE SuperSSN IS NOT NULL;

Nested Queries

- Complete **select-from-where** blocks within WHERE clause of another query.
- Comparison operator **IN**
  - Compares value v with a set (or multiset) of values V
  - Evaluates to TRUE if v is one of the elements in V
- Query 14: Retrieve the name and address of all employees who work for the 'Research' department

```
Q14:SELECT        FNAME, LNAME, ADDRESS
      FROM        EMPLOYEE
      WHERE       DNO IN (SELECT  DNUMBER
                          FROM    DEPARTMENT
                          WHERE   DNAME='Research' );
```

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.*
- Query 15: Retrieve the name of each employee who has a dependent with the same first name as the employee.
-

```
Q15: 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

- A query written with nested SELECT... FROM... WHERE... blocks and using IN comparison operator can ***always*** be expressed as a single block query For example, Q15 may be written as in Q15A:

```
Q15A:   SELECT      E.FNAME, E.LNAME
        FROM        EMPLOYEE E, DEPENDENT D
        WHERE       E.SSN=D.ESSN AND
                    E.FNAME=D.DEPENDENT_NAME;
```

Nested Query Exercises

- Query 16: Retrieve the SSNs of all employees who work the same (project, hours) combination on some project that employee John Smith (SSN=123456789) works on (using a nested query)

```
Q16: SELECT        DISTINCT      ESSN
         FROM      Works_on
         WHERE     (PNO, HOURS) IN
                         (SELECT       PNO, HOURS
                         FROM          Works_on
                         WHERE         ESSN=‘123456789’);
```

More Comparison Operators

- Use other comparison operators to compare a single value v
  - = ANY (or = SOME) operator
  - Returns TRUE if the value v is equal to some value in the set V and is hence equivalent to IN
- Other operators that can be combined with ANY (or SOME), ALL: >, >=, <, <=, and <>
- Query 17: Retrieve all employees whose salary is greater than the salary of all employees in dept. 5

```
Q17: SELECT     *
     FROM       Employee
     WHERE      Salary > ALL (SELECT Salary
                              FROM    Employee
                              WHERE   DNO=5);
```

The EXISTS and UNIQUE Functions in SQL

- `EXISTS` function
  - Check whether the result of a correlated nested query is empty or not.
- `EXISTS` and `NOT EXISTS`
  - Typically used in conjunction with a correlated nested query.
- SQL function `UNIQUE(Q)`
  - Returns `TRUE` if there are no duplicate tuples in the result of query Q.

The EXISTS Function

- Query 15: Retrieve the name of each employee who has a dependent with the same first name as the employee.

```
Q15B: SELECT      E.FNAME, E.LNAME
      FROM        EMPLOYEE
      WHERE       EXISTS (SELECT *
                          FROM  DEPENDENT
                          WHERE SSN=ESSN AND
                          FNAME=DEPENDENT_NAME);
```

The EXISTS Function

- Query 18: Retrieve the names of employees who have no dependents

```
Q18:        SELECT       FNAME, LNAME
            FROM         EMPLOYEE
            WHERE        NOT EXISTS  (SELECT  *
                         FROM  DEPENDENT
                         WHERE SSN=ESSN);
```

  - In Q18, 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.

Enumerated Sets

- It is also possible to use an **explicit (enumerated) set of values** in the WHERE-clause rather than a nested query
- Query 19: Retrieve the SSNs of all employees who work on project numbers 1, 2, or 3.

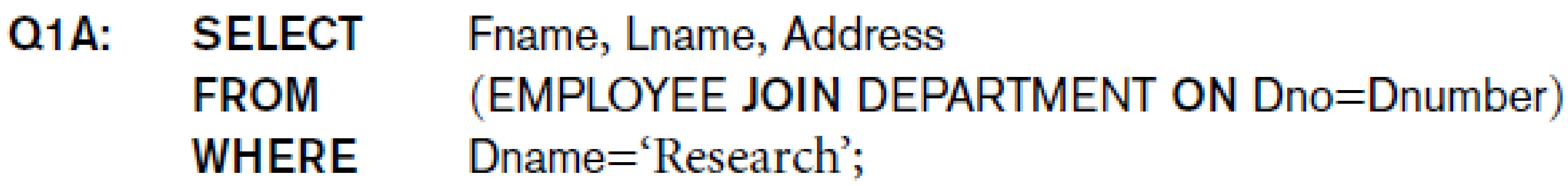
```
Q19:SELECT      DISTINCT ESSN
    FROM        WORKS_ON
    WHERE       PNO IN  (1, 2, 3);
```

Joined Relations Feature in SQL2

- Can specify a "joined relation" in the FROM-clause
- Allows the user to specify different types of joins (EQUIJOIN, NATURAL JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN)

Joined Tables in SQL and Outer Joins

- **Joined table**
  - Permits users to specify a table resulting from a join operation in the FROM clause of a query
- The FROM clause in Q1A
  - Contains a single joined table



```
Q1A:   SELECT   Fname, Lname, Address
       FROM     (EMPLOYEE JOIN DEPARTMENT ON Dno=Dnumber)
       WHERE    Dname='Research';
```

Joined Tables in SQL and Outer Joins

- Specify different types of join
  - NATURAL JOIN
  - Various types of OUTER JOIN
- NATURAL JOIN on two relations R and S
  - No join condition specified
  - Implicit EQUIJOIN condition for each pair of attributes with same name from R and S

Joined Tables in SQL and Outer Joins

- **Inner join**
  - Default type of join in a joined table
  - Tuple is included in the result only if a matching tuple exists in the other relation
- LEFT OUTER JOIN
  - Every tuple in left table must appear in result
  - If no matching tuple
    - Padded with NULL values for attributes of right table

Joined Tables in SQL and Outer Joins

- RIGHT OUTER JOIN
  - Every tuple in right table must appear in result
  - If no matching tuple
    - Padded with NULL values for the attributes of left table
- FULL OUTER JOIN
- Can nest join specifications

Joined Relations Feature in SQL2

- Examples:

```
SELECT    E.FNAME, E.LNAME, S.FNAME, S.LNAME
FROM      EMPLOYEE E, EMPLOYEE S
WHERE     E.SUPERSSN=S.SSN;
```

can be written as:

```
SELECT    E.FNAME, E.LNAME, S.FNAME, S.LNAME
FROM      (EMPLOYEE E LEFT OUTER JOIN
          EMPLOYEE S ON  E.SUPERSSN=S.SSN);
```

- Any differences ??

Joined Relations Feature in SQL2

- Examples:

```
SELECT     FNAME, LNAME, ADDRESS
FROM       EMPLOYEE, DEPARTMENT
WHERE      DNAME='Research' AND DNUMBER=DNO;
```

could be written as:

```
SELECT     FNAME, LNAME, ADDRESS
FROM       (EMPLOYEE JOIN DEPARTMENT ON
                 DNUMBER=DNO)
WHERE      DNAME='Research’;
```

or as:

```
SELECT     FNAME, LNAME, ADDRESS
FROM       (EMPLOYEE NATURAL JOIN (DEPARTMENT
            AS DEPT(DNAME, DNO, MSSN, MSDATE)))
WHERE      DNAME='Research’;
```

Joined Relations Feature in SQL2

- 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 could be written as follows; this illustrates multiple joins in the joined tables

```
SELECT      PNUMBER, DNUMBER, LNAME, BDATE, ADDRESS
FROM        ((PROJECT JOIN DEPARTMENT ON DNUM=
                  DNUMBER) JOIN EMPLOYEE ON
                  MGRSSN=SSN))
WHERE       PLOCATION='Stafford’;
```

Aggregate functions

- **COUNT**, **SUM**, **MAX**, **MIN**, **AVG**

- Query 20: Find the max, min, & average salary among all employees

```
Q20:SELECT      MAX(SALARY), MIN(SALARY),
                AVG(SALARY)
    FROM        EMPLOYEE;
```

Aggregate functions

- Queries 21 and 22: Retrieve the total number of employees in the company (Q17), and the number of employees in the 'Research' department (Q18)

```
Q21:SELECT      COUNT (*)
       FROM     EMPLOYEE;

Q22:SELECT      COUNT (*)
       FROM     EMPLOYEE, DEPARTMENT
       WHERE    DNO=DNUMBER AND
                      DNAME='Research’;
```

- Note: NULL values are discarded wrt. aggregate functions as applied to a particular column

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.*
- If NULLs exist in grouping attribute
    - Separate group created for all tuples with a NULL value in grouping attribute

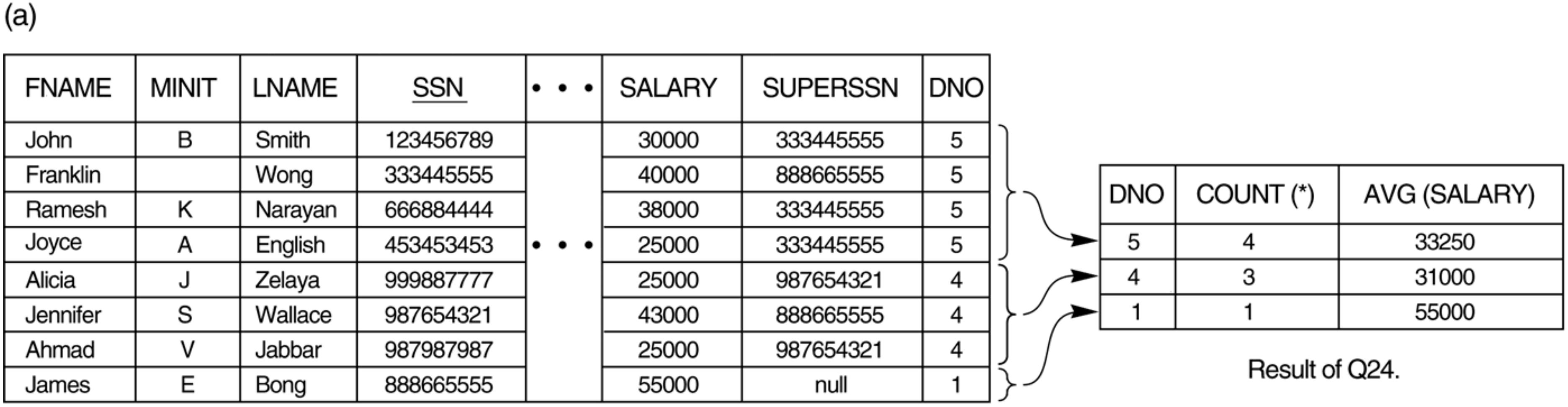
Grouping

- Query 23: For each department, retrieve the department number, the number of employees in the department, and their average salary.

```
Q23: SELECT        DNO, COUNT (*), AVG (SALARY)
     FROM          EMPLOYEE
     GROUP BY      DNO;
```

  - In Q23, 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: Q23 result



(a)

FNAME	MINIT	LNAME	SSN	• • •	SALARY	SUPERSSN	DNO
John	B	Smith	123456789		30000	333445555	5
Franklin		Wong	333445555		40000	888665555	5
Ramesh	K	Narayan	666884444		38000	333445555	5
Joyce	A	English	453453453	• • •	25000	333445555	5
Alicia	J	Zelaya	999887777		25000	987654321	4
Jennifer	S	Wallace	987654321		43000	888665555	4
Ahmad	V	Jabbar	987987987		25000	987654321	4
James	E	Bong	888665555		55000	null	1

Grouping EMPLOYEE tuples by the value of DNO.

DNO	COUNT (*)	AVG (SALARY)
5	4	33250
4	3	31000
1	1	55000

Result of Q24.

Grouping: 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).

Grouping: the having-clause

- Query 24: 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.

```
Q24:    SELECT      PNUMBER, PNAME, COUNT (*)
        FROM        PROJECT, WORKS_ON
        WHERE       PNUMBER=PNO
        GROUP BY    PNUMBER, PNAME
        HAVING      COUNT (*) > 2;
```

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 25: 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.

```
Q25: SELECT         DNAME, LNAME, FNAME, PNAME
       FROM         DEPARTMENT, EMPLOYEE, WORKS_ON,
                    PROJECT
       WHERE        DNUMBER=DNO AND SSN=ESSN AND
                    PNO=PNUMBER
       ORDER BY     DNAME, LNAME [DESC|ASC];
```

SELECT – summarization

```
SELECT [DISTINCT | ALL]
   {* | [columnExpression [AS newName]] [,...] }
FROM TableName [alias] [, ...]
[WHERE condition]
[GROUP BY columnList]    [HAVING condition]
[ORDER BY columnList]
```

DML: Select, Insert, Update, Delete

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 INTO <*table name*> [(<*list of columns*>)] VALUES (<*list of expressions*>);**
- **INSERT INTO <*table name*> [(<*list of columns*>)] SELECT *statement*;**

DML: Select, Insert, Update, Delete

INSERT

- 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.

```
U2:   INSERT INTO EMPLOYEE (FNAME, LNAME, SSN)
         VALUES ('Richard', 'Marini', '653298653');
```

DML: Select, Insert, Update, Delete

INSERT

- 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.

DML: Select, Insert, Update, Delete

INSERT

- 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 U3, and is loaded with the summary information retrieved from the database by the query in U3A

```
U3:CREATE TABLE  DEPTS_INFO
                 (DEPT_NAME       VARCHAR(10),
                  NO_OF_EMPS      INTEGER,
                  TOTAL_SAL       INTEGER);

U3A: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;
```

DML: Select, Insert, Update, Delete

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.
- **DELETE [FROM] <*table name*>**

  [WHERE <*row conditions*>];

DML: Select, Insert, Update, Delete

DELETE

- Examples:

```
U4A:       DELETE FROM          EMPLOYEE
           WHERE                LNAME='Brown’;

U4B:       DELETE FROM          EMPLOYEE
           WHERE                SSN='123456789’;

U4C:       DELETE FROM          EMPLOYEE
           WHERE                DNO  IN
                          (SELECT        DNUMBER
                          FROM           DEPARTMENT
                          WHERE          DNAME='Research');

U4D:       DELETE FROM          EMPLOYEE;
```

DML: Select, Insert, Update, Delete

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 <*table name*> [<*alias*>]**

  SET <*column1*> = {<*expression*>, <*subquery*>}

  [, <*column2*> = {<*expression*>, <*subquery*>} …]

  [WHERE <*row conditions*>];

DML: Select, Insert, Update, Delete

UPDATE

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

```
U5:  UPDATE          PROJECT
     SET             PLOCATION = 'Bellaire', DNUM = 5
     WHERE           PNUMBER=10;
```

DML: Select, Insert, Update, Delete

UPDATE

- 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');
```

Advanced DDL: Assertions & Triggers

- **`CREATE ASSERTION`**
  - Specify additional types of constraints outside scope of built-in relational model constraints.
  - components include: a constraint name, followed by `CHECK`, followed by a condition.
- **`CREATE TRIGGER`**
  - Specify automatic actions that database system will perform when certain events and conditions occur.

Advanced DDL: Assertions & Triggers

- `CREATE ASSERTION`
  - Specify a query that selects any tuples that violate the desired condition.
  - Use only in cases where it is not possible to use `CHECK` on attributes and domains.

Advanced DDL: Assertions & Triggers

- “The salary of an employee must not be greater than the salary of the manager of the department that the employee works for.”

```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK (NOT EXISTS (SELECT *
    FROM EMPLOYEE E, EMPLOYEE M, DEPARTMENT D
    WHERE E.SALARY>M.SALARY AND E.DNO=D.NUMBER
          AND D.MGRSSN=M.SSN));
```

Advanced DDL: Assertions & Triggers

- Triggers: to specify the type of action to be taken as certain events occur & as certain conditions are satisfied.

VIEWs

- A view is a **“virtual” table that is derived from other tables.**
- Allows for limited update operations (since the table may not physically be stored).
- Allows full query operations.
- A convenience for expressing certain operations.

VIEWs

- SQL command: **CREATE VIEW**
  - a view (table) name
  - a possible list of attribute names
  - a query to specify the view contents
- Specify a different WORKS_ON table (view)

```
CREATE VIEW      WORKS_ON_NEW AS
    SELECT       FNAME, LNAME, PNAME, HOURS
    FROM         EMPLOYEE, PROJECT, WORKS_ON
    WHERE        SSN=ESSN AND PNO=PNUMBER;
```

VIEWs

- We can specify SQL queries on a newly create table (view):

```
SELECT FNAME, LNAME FROM WORKS_ON_NEW
WHERE PNAME='Seena';
```

- View always up-to-date
  - Responsibility of the DBMS and not the user
- When no longer needed, a view can be dropped:

```
DROP VIEW WORKS_ON_NEW;
```

View Update and Inline Views

- Update on a view defined on a single table without any aggregate functions
  - Can be mapped to an update on underlying base table.
- View involving joins
  - Often not possible for DBMS to determine which of the updates is intended.
- More details: 5.3.3

View Update and Inline Views

- Clause **`WITH CHECK OPTION`**
  - Must be added at the end of the view definition if a view is to be updated
- **In-line view**
  - Defined in the `FROM` clause of an SQL query

Contents

1	The COMPANY Database
2	SQL developments: an overview
3	DDL: Create, Alter, Drop
4	DML: select, insert, update, delete
5	**DCL: commit, rollback, grant, revoke**
6	Trigger, Store Procedure, Function & Cursor in Oracle

DCL: Commit, Rollback, Grant, Revoke

- Chapter 17: Transaction Processing
- Chapter 23: DB security

Contents

1	The COMPANY Database
2	SQL developments: an overview
3	DDL: Create, Alter, Drop
4	DML: select, insert, update, delete
5	DCL: commit, rollback, grant, revoke
6	**Trigger, Store Procedure, Function & Cursor in Oracle**

Trigger, Store Procedure, Function & Cursor in Oracle

- **Trigger**
- Store Procedure & Function
- Cursor

Trigger Overview

- A trigger is a procedure which is executed implicitly whenever the triggering event happens.
- Executing a trigger is to “fire” the trigger.
- Triggering Events are:
  - DML Commands: INSERT, UPDATE, DELETE
  - DDL Commands : CREATE, ALTER, DROP
  - Database Events: SERVERERROR, LOGON, LOGOFF, STARTUP, SHUTDOWN

Trigger Overview

- Uses for triggers:
    - Automatically generate derived column values.
    - Maintain complex integrity constraints.
    - Enforce complex business rules.
    - Record auditing information about database changes.
    - Invoke a program when database changes.

Simple DML Trigger Syntax

CREATE [OR REPLACE] TRIGGER *schema.trigger_name*

BEFORE | AFTER | INSTEAD OF

DELETE | INSERT | UPDATE [*OF columns list*] [OR …]

ON *schema.table_name*

[*REFERENCING OLD [AS] <old_name>* | *NEW [AS] <new_name>*]

[**FOR EACH ROW**]

[**WHEN** *(condition)*]

BEGIN

  PL/SQL_block | call_procedure_statement;

END *trigger_name* **;**

Types of Triggers

Category	Values	Comments
DML	Insert Update Delete	Type of DML which makes the trigger fire.
Timing	Before After Instead of	When the trigger fires.
Level	Row	Row level triggers fire for each affected row. Identified by keywords FOR EACH ROW
	Statement	Statement level triggers fire once per DML Statement

Trigger Firing Order

1. **Before statement** triggers fire.
2. For Each Row:

    A) **Before row** triggers fire.

    B) Execute the Insert/Update/Delete.

    C) **After row** triggers fire.

3. **After statement** triggers fire.

REFERCING Clause: Old and New Data

- When **row-triggers** fire, there are 2 pseudo-records created called new and old.

```
    new table_name%ROWTYPE;
    old table_name%ROWTYPE;
```

- old and new are of datatype ROWTYPE from the affected table. Use dot notation to reference columns from old and new.
- old is undefined for insert statements.
- new is undefined for delete statements.

REFERCING Clause: Old and New Data

- Instead of a REFERENCING clause, Oracle assumes that new tuples are referred to as “new” and old tuples by “old.”
- Also, for statement-level triggers: “newtable” and “oldtable”.
- In actions, *but not in conditions*, you must prefix “new,” etc., by a colon
    - :new
    - :old

Example: Row Level Trigger

```
CREATE TRIGGER    NoLowerPrices
AFTER UPDATE OF  price  ON Product
FOR EACH ROW
WHEN (old.price > new.price)
BEGIN
   UPDATE  Product
   SET  price = :old.price
   WHERE  p_name = :new.p_name;
END;
```

Bad Things Can Happen

```
CREATE TRIGGER  Bad_trigger
AFTER UPDATE OF price ON Product
FOR EACH ROW
WHEN   (new.price > 50)
BEGIN
   UPDATE  Product
   SET  price = :new.price * 2
   WHERE  p_name = :new.p_name;
END;
```

Trigger, Store Procedure, Function & Cursor in Oracle

- Trigger
- **Store Procedure & Function**
- Cursor

Database Stored Procedures

- **Stored procedures**
  - Program modules stored by the DBMS at the database server
  - Can be functions or procedures
- **Persistent stored modules**
  - Stored persistently by the DBMS

Stored Procedures & Functions

- Useful:
  - When database program is needed by several applications
  - To reduce data transfer and communication cost between client and server in certain situations
  - To enhance modeling power provided by views

Stored Procedures & Functions

- Declaring stored procedures:

```
CREATE [OR REPLACE] PROCEDURE
  procedure_name
[(parameter_name [IN | OUT | IN OUT]
  datatype )]
{IS | AS}
BEGIN
  procedure_body
END procedure_name;
```

Stored Procedures & Functions

- Parameter:
  - **Data type**: one of the SQL data types.
  - **Parameter mode**: `IN`, `OUT`, or `IN OUT`
    - **IN:** you must supply a value for the parameter when calling the procedure.
    - **OUT:** procedure passes a value for this parameter back to its calling environment after execution.
    - **IN OUT:** you must supply a value for the parameter when calling the procedure and that the procedure passes a value back to its calling environment after execution.
    - **Defaults: IN**.

Stored Procedures & Functions

- Example of store procedure:

```
CREATE OR REPLACE PROCEDURE update_salary
    (p_emp_id IN EMPLOYEE.SSN%type,
     p_factor IN NUMBER)
AS
   v_emp_count INTEGER;
BEGIN
   SELECT COUNT(*) INTO v_emp_count
    FROM employee
   WHERE SSN = p_emp_id;
   IF v_emp_count = 1 THEN
          UPDATE employee
          SET salary = salary * p_factor
          WHERE SSN = p_emp_id;
          COMMIT;
   END IF;
END update_salary;
```

Stored Procedures & Functions

- Calling a store procedure:
  - EXECUTE **update_salary (‘123456789’, 1.5);**
  - BEGIN

```
BEGIN
            update_salary (‘123456789’, 1.5);
END;
```

Stored Procedures & Functions

- Declaring function:

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter_name [IN | OUT | IN OUT]
  datatype )]
RETURN datatype
{IS | AS}
BEGIN
  function_body
END function_name;
```

Stored Procedures & Functions

- Example of Function:

```
CREATE OR REPLACE FUNCTION get_salary
  (p_emp_id IN EMPLOYEE.SSN%TYPE)
RETURN NUMBER
AS
  v_sal NUMBER;
BEGIN
  SELECT salary into v_sal
  FROM EMPLOYEE
  WHERE SSN = p_emp_id;
  RETURN v_sal;
END get_salary;
```

Stored Procedures & Functions

- Calling a function:
  - SELECT * FROM EMPLOYEE

    WHERE salary = **get_salary (‘123456789’);**

  - SELECT **get_salary (‘123456789’)** FROM dual;

Trigger, Store Procedure, Function & Cursor in Oracle

- Trigger
- Store Procedure & Function
- **Cursor**

Database Access Using Cursors

- When the result of an SQL query (select statement) consists of more than one row, the simple select into statement can not be used.
- A PL/SQL **cursor** allows the program to fetch and process information from the database into the PL/SQL program, **one row at a time**.

Explicit Cursor

- Explicit cursor: used for processing a query resulting in more than one row.
- Implicit cursor: is automatically defined by PL/SQL for the select into statements, which result in one or fewer rows.
- Syntax of explicit cursor:

```
cursor <cname>  [return-spec]
is <select-statement>;
```

Cursor Example

```
cursor c1 return customers%rowtype is
select * from customers;
```

has return clause

```
cursor c2 is
select pno, pname, price*markdown sale_price
from parts;
```

Use PL/SQL variable markdown

Process cursor

- One a cursor has been declared, it can be processed using the **open**, **fetch**, and **close** statements.

```
open <cname>;
fetch <cname> into <Record-or-VariableList>;
close <cname>;
```

Explicit Cursor Attributes

- Obtain status information about a cursor.

%FOUND	Returns TRUE if the last fetch returned a row, or FALSE if the last fetch failed to return a row.
%NOTFOUND	The logical opposite of %FOUND.
%ROWCOUNT	Before the first fetch, returns 0. When a cursor is opened, %ROWCOUNT is zeroed. Thereafter, returns **the number of rows fetched** so far. The number is incremented if the latest fetch returned a row.
%ISOPEN	If a cursor is open, returns TRUE; otherwise, it returns FALSE.

Explicit Cursor Attributes example

```
IF c1%ISOPEN THEN
   FETCH c1 INTO v_ename, v_sal,
   v_hiredate;
ELSE
      OPEN c1;
END IF;
```

```
LOOP
   FETCH c1 INTO v_ename, v_sal,
   v_hiredate;
   EXIT WHEN c1%ROWCOUNT > 10;
END LOOP;
```

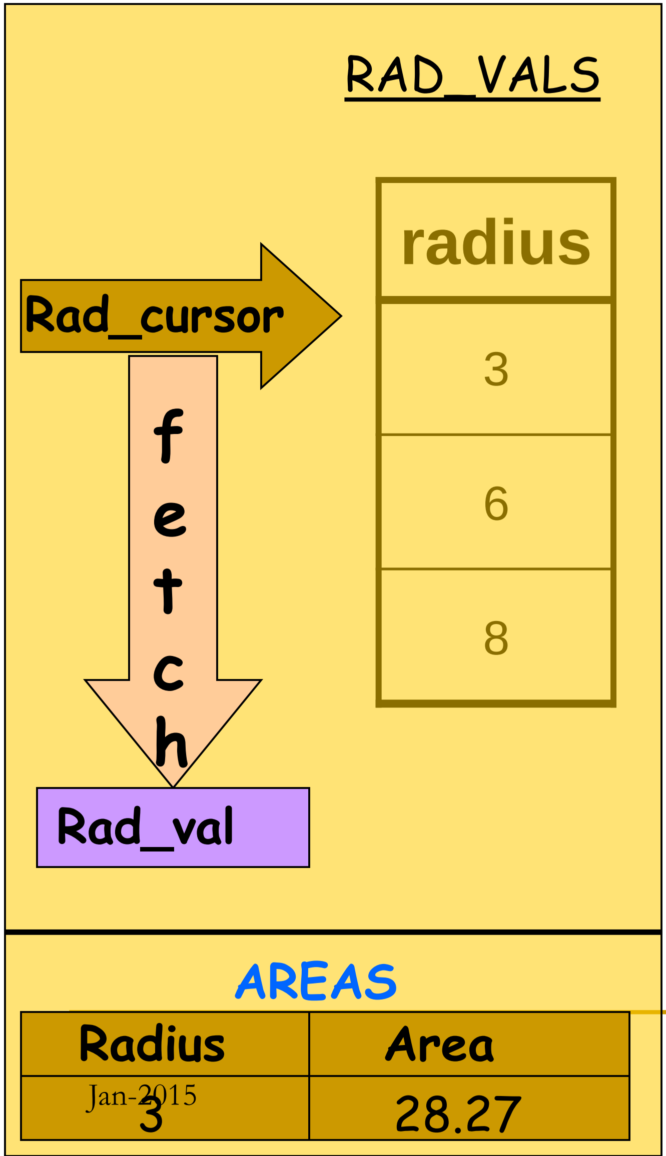
Cursor Example

```
DECLARE
    cursor c is select * from sailors;
     sailorData sailors%ROWTYPE;

BEGIN
    open c;
    fetch c into sailorData;
```

sailorData is a variable that can hold a ROW from the sailors table

Here the first row of sailors is inserted into sailorData

Cursor Example



RAD_VALS

radius
3
6
8

AREAS

Radius	Area
3	28.27

```
DECLARE
  Pi constant NUMBER(8,7) := 3.1415926;
  area NUMBER(14,2);
  cursor rad_cursor is select * from RAD_VALS;
  rad_val rad_cursor%ROWTYPE;

BEGIN
  open rad_cursor;
  fetch rad_cursor into rad_val;
  area:=pi*power(rad_val.radius,2);
  insert into AREAS values (rad_val.radius,
     area);
  close rad_cursor;
END;
/
```

Cursor FOR LOOP statement

- This loop is very useful when all rows of the cursors are to be processed.

```
for <record_index> in <cursor name>
  loop
  <loop-body>;
end loop;
```

- `<record_index>` is a record variable that is implicitly declared by PL/SQL. Its scope is the for loop, and it can not be accessed outside the for loop.

Cursor FOR LOOP statement

- The loop terminates automatically when all rows of the cursor have been fetched.
- There is no need to open, fetch, or close the cursor, and there is no need to declare the record into which the cursor rows are to be fetched.

Cursor FOR LOOP example

```
declare
  cursor c1 is
    select cno, cname, city
    from customers, zipcodes
    where customers.zip = zipcodes.zip;
begin
  for c1_rec in c1 loop
     dbms_output.put_line(‘Row number ’ ||
  c1%rowcount ||     ‘> ‘ || c1_rec.cno || ‘
  ‘ || c1_rec.cname || ‘ ‘ || c1_rec.city);
  end loop
end;
```

c1_rec

No declare for the record into which the cursor rows are to be fetched

Another controlling Cursor Example

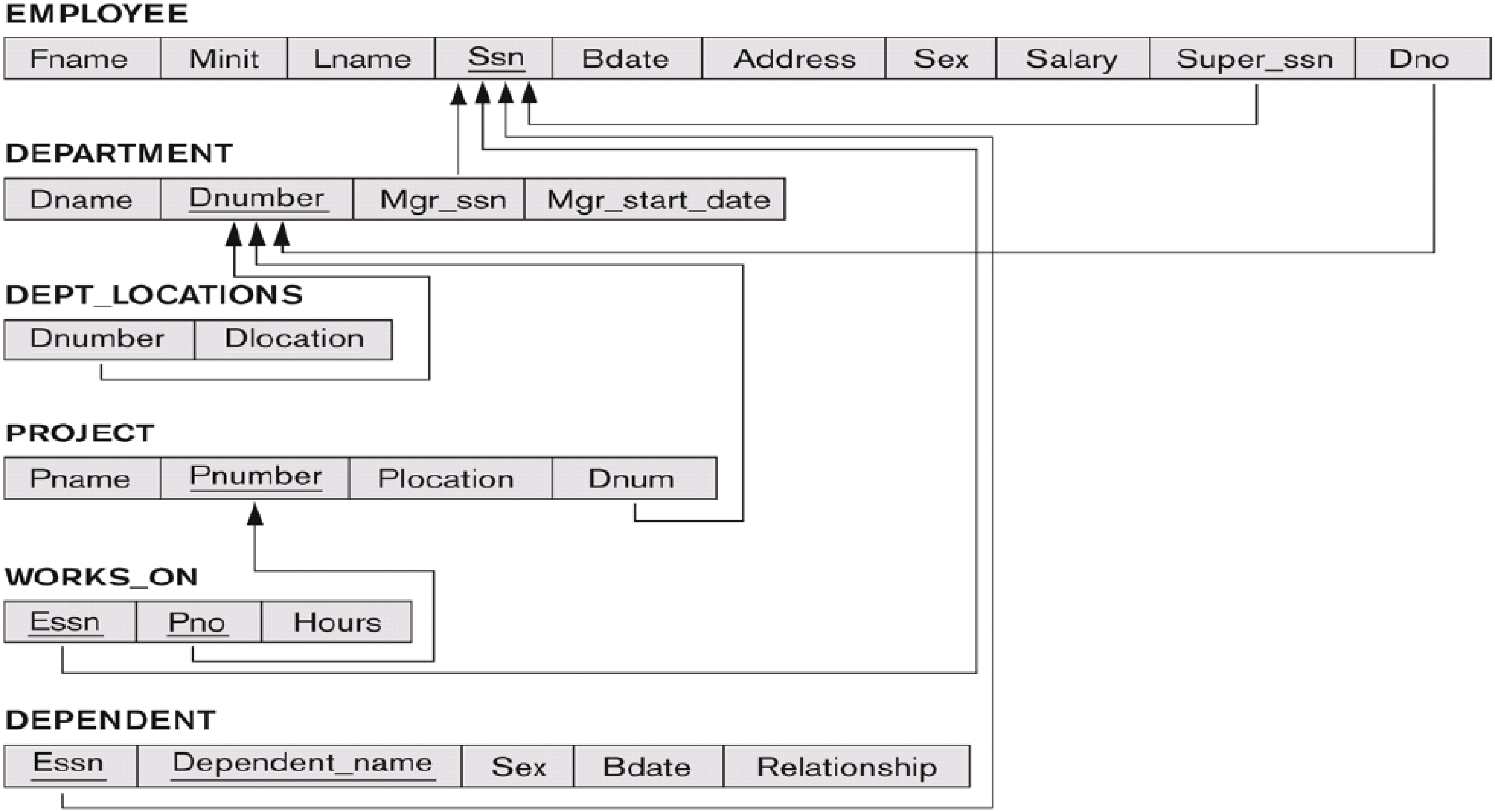
```
OPEN c_1;
LOOP
  -- fetch from cursor variable
  FETCH c_1 INTO a, b, c;
  -- exit when last row is fetched
  EXIT WHEN c_1%NOTFOUND;

  -- process data record

END LOOP;
```

Summary

- SQL developments: an overview
- SQL
  - DDL: Create, Alter, Drop
  - DML: select, insert, update, delete
  - Introduction to advanced DDL (assertions & triggers), views, DCL (commit, rollback, grant, revoke)
- Trigger, Store Procedure, Function & Cursor in Oracle

Q & A

Exercise



EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date

DEPT_LOCATIONS

Dnumber	Dlocation

PROJECT

Pname	Pnumber	Plocation	Dnum

WORKS_ON

Essn	Pno	Hours

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship

1. For each employee, retrieve the employee’s first name and last name and the first and last name of his/her immediate supervisor.
2. Retrieve the names of all employees in the departments which are located in Houston.
3. List the names of all employees who have a dependent with the same first name as themselves.
4. For each project, calculate the total number of employees who work for it, and the total number of hours that these employees work for the project.
5. Retrieve the average salary of all female employees.
6. For each department whose average employee salary is more than $30.000, retrieve the department name and the number of employees work for that department.

7. Write a trigger for ensuring that the employee’s ages must be between18 and 60.
8. Write a trigger to enforce that when an employee has a new project, his or her salary will be increased by 10% * number of hours per week working on that project.
9. Write a store procedure to read an employee’s id and print the names of his/her dependents.
10. Write a function to read a project’s id and return the total number of employees who work for that project.

Review questions

1) How do the relations (tables) in SQL differ from the relations defined formally in Chapter 4? Discuss the other differences in terminology.Why does SQL allow duplicate tuples in a table or in a query result?
2) List the data types that are allowed for SQL attributes.
3) How does SQL allow implementation of the entity integrity and referential integrity constraints described in Chapter 4? What about referential triggered actions?