DBMD UNIT - 3 and 4 Notes

UNIT-III: DATABASE IMPLEMENTATION AND PHYSICAL DATABASE DESIGN

1. Database Creation using SQL [PYQ Q6a]

- **Data Definition Using SQL (SQL/DDL):** (Ch 10.1, p.444)
  - SQL is the standard language. Commercial DBMSs might have variations.
  - A "relation" is formally called a "table" in SQL. "Attributes" are "columns", "tuples" are "rows".
  - SQL tables can have duplicate rows and nulls (unlike formal relations). Column order matters.
  - Major DDL constructs: `CREATE`, `ALTER`, `DROP`.
- **Base Table Specification in SQL/DDL:** (Ch 10.1.1, p.445)
  - `CREATE TABLE` **Statement:** Defines a new base table (stored physically).

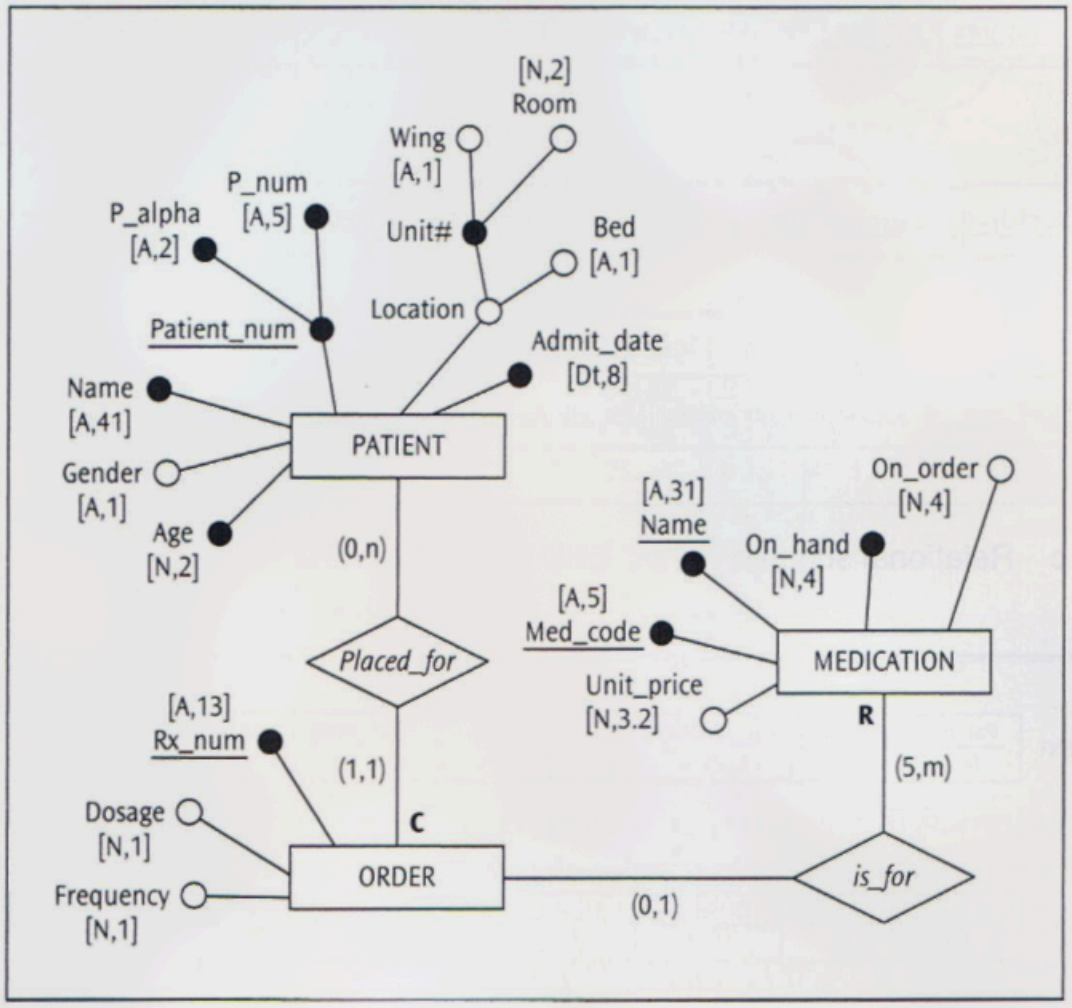


Figure 10.1a ER diagram: An excerpt from a hypothetical medical information system

> Constraint	Gender	IN ('M', 'F')
> Constraint	Age	IN (1 through 90)
> Constraint	Bed	IN ('A', 'B')
> Constraint	Unit_price	< 4.50
> Constraint	(Qty_onhand + Qty_onorder)	IN (1000 through 3000)
> Constraint	Dosage	DEFAULT 2
> Constraint	Dosage	IN (1 through 3)
> Constraint	Frequency	DEFAULT 1
> Constraint	Frequency	IN (1 through 3)

Figure 10.1b Semantic integrity constraints for the Fine-granular Design-Specific ER diagram

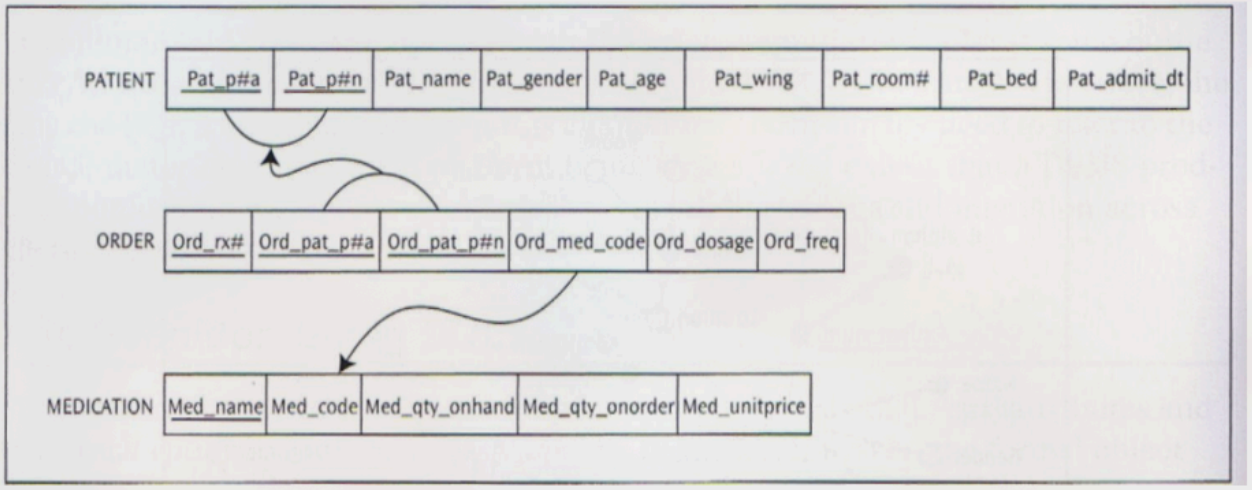


Figure 10.1c Relational schema for the ERD in Figure 10.1a

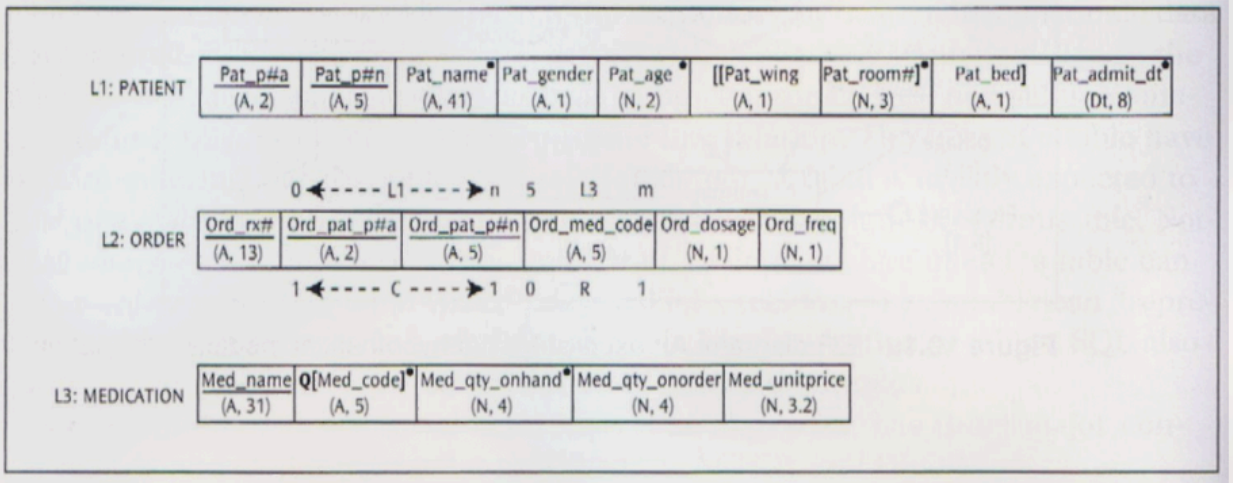
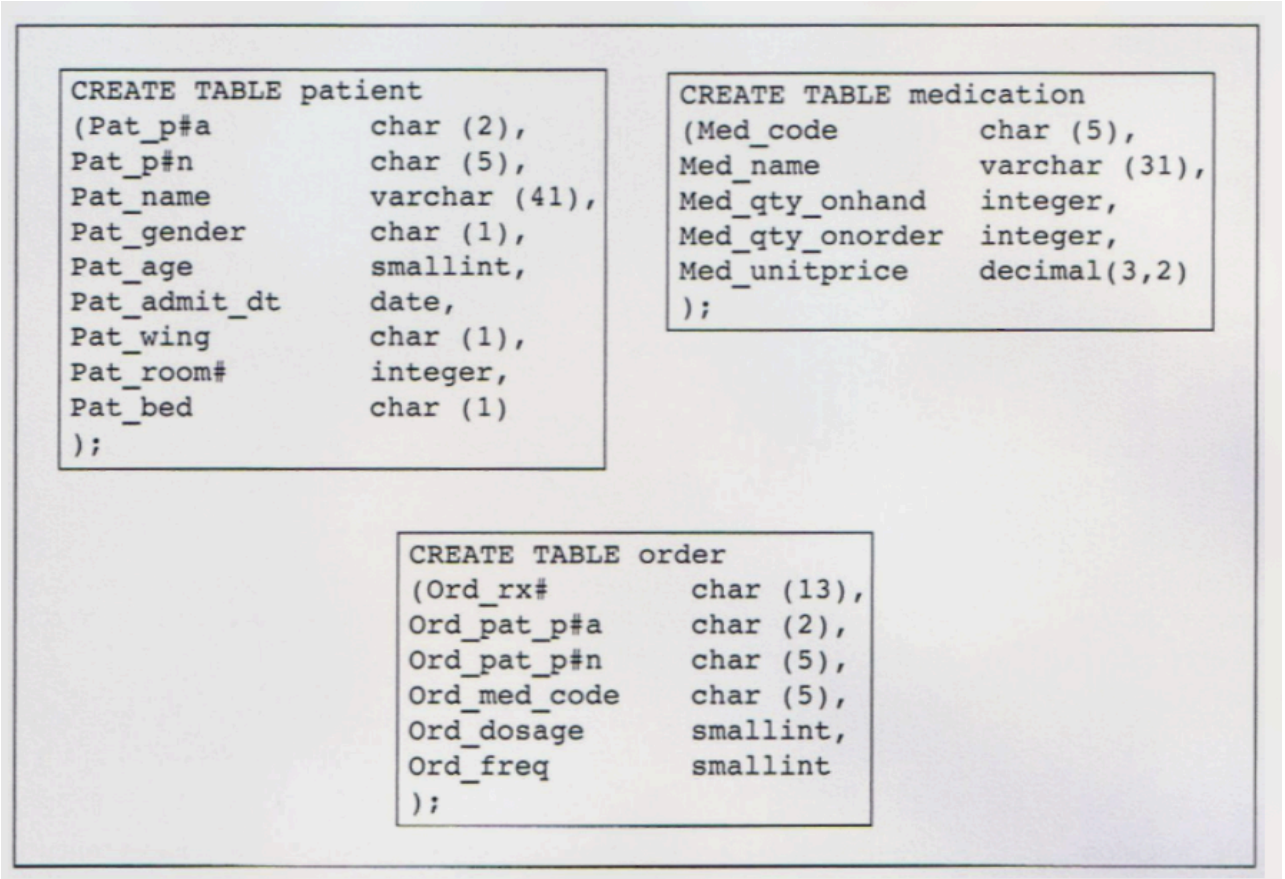


Figure 10.1d An information-preserving logical schema for the ERD in Figure 10.1a

*(Box 1, p.447 for minimal syntax; ERD Fig 10.1a, Semantic Constraints Fig 10.1b, Relational

Schema Fig 10.1c, Info-Preserving Logical Schema Fig 10.1d, p.445-446



```
CREATE TABLE patient
(Pat_p#a          char (2),
Pat_p#n           char (5),
Pat_name          varchar (41),
Pat_gender        char (1),
Pat_age           smallint,
Pat_admit_dt      date,
Pat_wing          char (1),
Pat_room#         integer,
Pat_bed           char (1)
);
```

```
CREATE TABLE medication
(Med_code         char (5),
Med_name          varchar (31),
Med_qty_onhand    integer,
Med_qty_onorder   integer,
Med_unitprice     decimal(3,2)
);
```

```
CREATE TABLE order
(Ord_rx#          char (13),
Ord_pat_p#a       char (2),
Ord_pat_p#n       char (5),
Ord_med_code      char (5),
Ord_dosage        smallint,
Ord_freq          smallint
);
```

Box 1

- **Syntax:** `CREATE TABLE table_name (column_definition_1, column_definition_2, ..., [table_constraint_1, ...]);`
- **Column Definition:** `column_name data_type [DEFAULT default_value] [column_constraint_list]`

- **Data Types:** (Standard SQL-92 types, vendors may add more) *(Table 10.1, p.448-449)*

Table 10.1 Data types supported by SQL-92

Number Data Types	
numeric (p, s) where **p** indicates precision and **s** indicates scale	Exact numeric type—literal representation of number values—*decimal portion exactly the size dictated by the scale*—storage length = (precision + 1) when scale is > 0—most DBMSs have an upper limit on **p** (e.g., 28)
decimal (p, s) where **p** indicates precision and **s** indicates scale	Exact numeric type—literal representation of number values—*decimal portion at least the size of the scale; but expandable to limit set by the specific DBMS*—storage length = (precision + 1) when scale is > 0 —most DBMSs have an upper limit on **p** (e.g., 28)
integer or **integer (p)** where **p** indicates precision	Exact numeric type—binary representation of large whole number values—often precision set by the DBMS vendor (e.g., 2 bytes)
smallint or **smallint (p)** where **p** indicates precision	Exact numeric type—binary representation of small whole number values—often precision set by the DBMS vendor (e.g., 1 byte)
float (p) where **p** indicates precision	Approximate numeric type—represents a given value in an exponential format—precision value represents the minimum size used, up to the maximum set by the DBMS
real	Approximate numeric type—represents a given value in an exponential format—has a default precision value set below that set for double precision data type by the DBMS
double precision	Approximate numeric type—represents a given value in an exponential format—has a default precision value set above that set for real data type by the DBMS
String Data Types	
character (ℓ) or **char (ℓ)** where ℓ indicates length	Fixed length character strings including blanks from the defined language set SQL_TEXT within a database—can be compared to other columns of the same type with different lengths or varchar type with different maximum lengths—most DBMS have an upper limit on ℓ (e.g., 255)
character varying (ℓ) or **char (ℓ) varying** or **varchar (ℓ)** where ℓ indicates the maximum length	Variable length character strings except trailing blanks from the defined language set SQL_TEXT within a database—DBMS records actual length of column values—can be compared to other columns of the same type with different maximum lengths or char type with different lengths—most DBMS have an upper limit on ℓ (e.g., 2000)

Table 10.1 Data types supported by SQL-92 (continued)

String Data Types (continued)	
bit (ℓ) where ℓ indicates length	Fixed length binary digits (0,1)—can be compared to other columns of the same type with different lengths or bit varying type with different maximum lengths
bit varying (ℓ) where ℓ indicates maximum length	Variable length binary digits (0,1)—can be compared to other columns of the same type with different maximum lengths or bit type with different lengths
Date/Time & Interval Data Types	
date	10 characters long—format: yyyy-mm-dd—can be compared to only other date type columns—allowable dates conform to the Gregorian calendar
time (p)	Format: hh:mi:ss—sometimes precision (**p**) specified to indicate fractions of a second—the length of a TIME value is 8 characters, if there are no fractions of a second. Otherwise, the length is 8, plus the precision, plus one for the delimiter: hh:mi:ss.p—if no precision is specified, it is 0 by default—TIME can only be compared to other TIME data type columns
timestamp (p)	Format: yyyy:mm:dd hh:mi:ss.p—a timestamp length is nineteen characters, plus the precision, plus one for the precision delimiter—timestamp can only be compared to other timestamp data type columns
interval (q)	Represents measure of time—there are two types of intervals: year-month (yyyy:mm) which stores the year and month; and day-time (dd hh:mi:ss) which stores the days, hours, minutes, and seconds—the qualifier (**q**) known in some databases as the interval lead precision, dictates whether the interval is year-month or day-time—implementation of the qualifier value varies

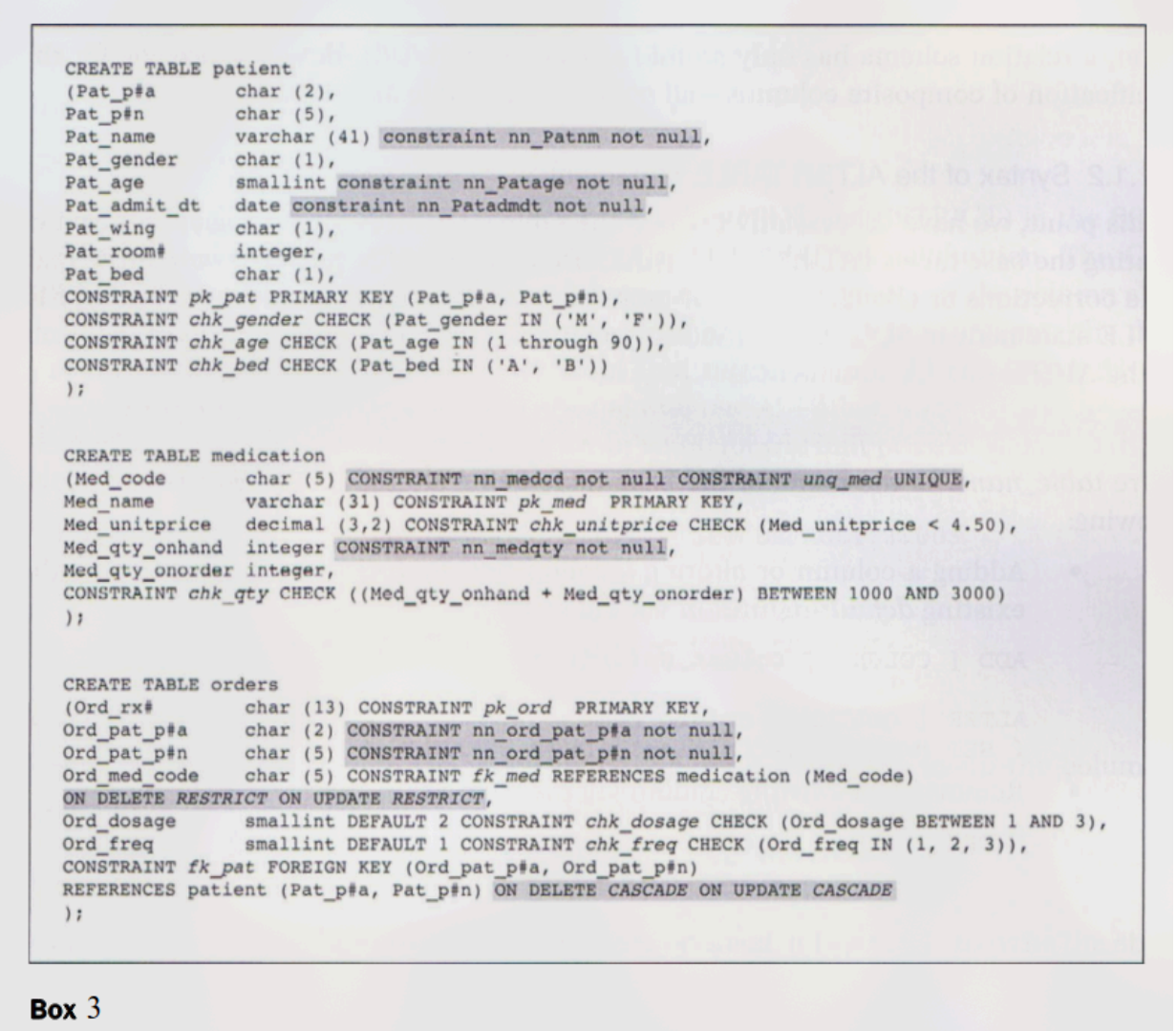
- Numeric: `NUMERIC(p,s)`, `DECIMAL(p,s)`, `INTEGER`, `SMALLINT`, `FLOAT(p)`, `REAL`, `DOUBLE PRECISION`.
- String: `CHARACTER(n)` or `CHAR(n)` (fixed-length), `CHARACTER VARYING(n)` or `VARCHAR(n)` (variable-length).
- Bit String: `BIT(n)`, `BIT VARYING(n)`.
- Date/Time & Interval: `DATE`, `TIME(p)`, `TIMESTAMP(p)`, `INTERVAL`.

- **Constraints (Column-level or Table-level):**
  - `NOT NULL`: Ensures a column cannot have null values.
  - `UNIQUE`: Ensures all values in a column (or set of columns) are unique. Can define alternate keys.
  - `PRIMARY KEY`: Specifies the primary key for the table (implies `NOT NULL` and `UNIQUE`).
  - `FOREIGN KEY ... REFERENCES ...`: Defines a foreign key and the referenced table/columns.
    - **Referential Triggered Actions (Deletion/Update Rules):** [PYQ - Implied in DDL] (p.451, Ch 3.2.3)
      - `ON DELETE {CASCADE | SET NULL | SET DEFAULT | RESTRICT | NO ACTION}`

        - `ON UPDATE {CASCADE | SET NULL | SET DEFAULT | RESTRICT | NO ACTION}`
    - `CHECK (condition)`: Specifies a condition that must be true for every row.
- **Naming Conventions:** Follow consistent naming for tables and columns.
  (Box 2 & 3, p.452, 453 for `CREATE TABLE` with constraints for Patient/Medication/Orders example).

```
CREATE TABLE patient
(Pat_p#a        char (2),
Pat_p#n         char (5),
Pat_name        varchar (41),
Pat_gender      char (1),
Pat_age         smallint,
Pat_admit_dt    date,
Pat_wing        char (1),
Pat_room#       integer,
Pat_bed         char (1),
CONSTRAINT pk_pat PRIMARY KEY (Pat_p#a, Pat_p#n),
CONSTRAINT chk_gender CHECK (Pat_gender IN ('M', 'F')),
CONSTRAINT chk_age CHECK (Pat_age IN (1 through 90)),
CONSTRAINT chk_bed CHECK (Pat_bed IN ('A', 'B'))
);


CREATE TABLE medication
(Med_code         char (5),
Med_name          varchar (31) CONSTRAINT pk_med  PRIMARY KEY,
Med_unitprice     decimal (3,2) CONSTRAINT chk_unitprice CHECK (Med_unitprice < 4.50),
Med_qty_onhand    integer,
Med_qty_onorder   integer,
CONSTRAINT chk_qty CHECK ((Med_qty_onhand + Med_qty_onorder) BETWEEN 1000 AND 3000)
);


CREATE TABLE orders
(Ord_rx#          char (13) CONSTRAINT pk_ord  PRIMARY KEY,
Ord_pat_p#a       char (2),
Ord_pat_p#n       char (5),
Ord_med_code      char (5) CONSTRAINT fk_med FOREIGN KEY REFERENCES medication (med_code),
Ord_dosage        smallint DEFAULT 2 CONSTRAINT chk_dosage CHECK (Ord_dosage BETWEEN 1 AND 3),
Ord_freq          smallint DEFAULT 1 CONSTRAINT chk_freq CHECK (Ord_freq IN (1,2,3)),
CONSTRAINT fk_pat FOREIGN KEY (Ord_pat_p#a, Ord_pat_p#n)
REFERENCES patient (Pat_p#a, Pat_p#n)
);
```

BOX 2



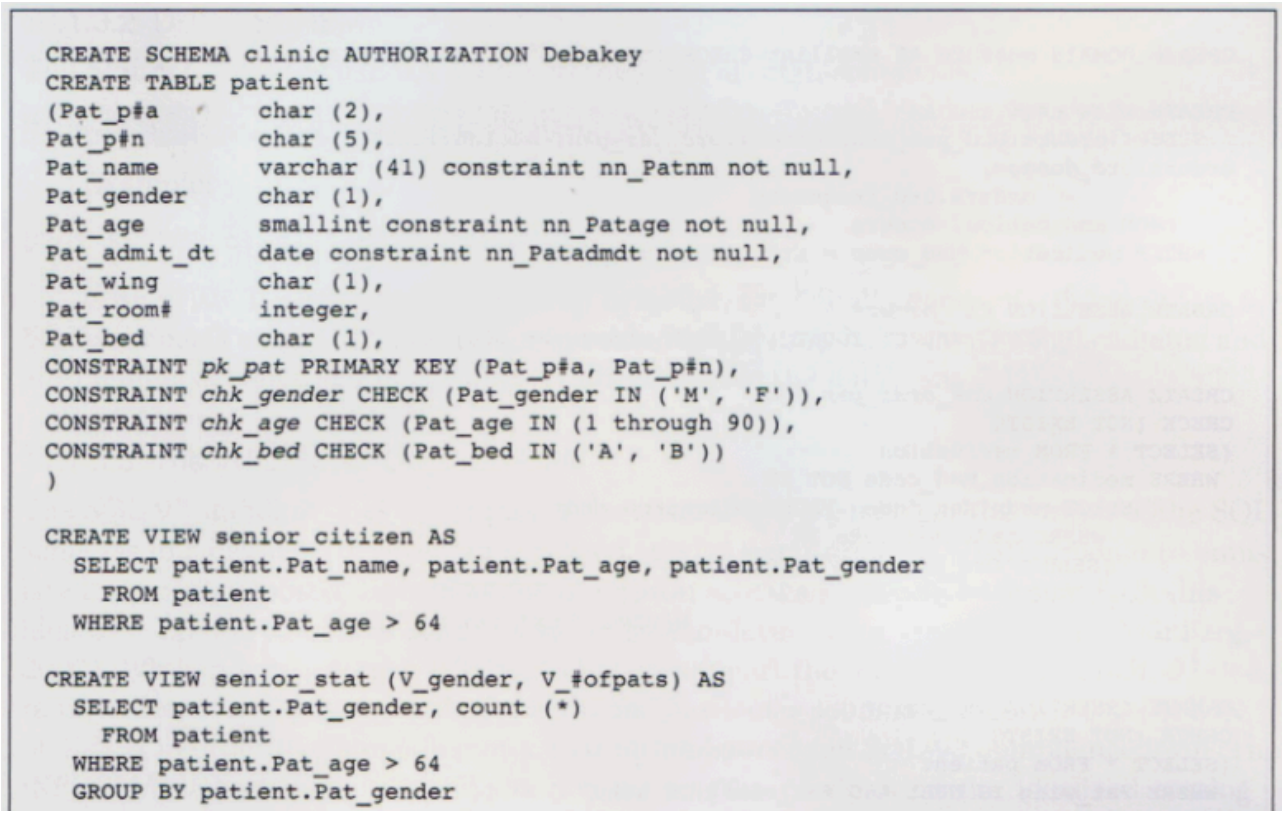
```
CREATE TABLE patient
(Pat_p#a        char (2),
Pat_p#n         char (5),
Pat_name        varchar (41) constraint nn_Patnm not null,
Pat_gender      char (1),
Pat_age         smallint constraint nn_Patage not null,
Pat_admit_dt    date constraint nn_Patadmdt not null,
Pat_wing        char (1),
Pat_room#       integer,
Pat_bed         char (1),
CONSTRAINT pk_pat PRIMARY KEY (Pat_p#a, Pat_p#n),
CONSTRAINT chk_gender CHECK (Pat_gender IN ('M', 'F')),
CONSTRAINT chk_age CHECK (Pat_age IN (1 through 90)),
CONSTRAINT chk_bed CHECK (Pat_bed IN ('A', 'B'))
);


CREATE TABLE medication
(Med_code       char (5) CONSTRAINT nn_medcd not null CONSTRAINT unq_med UNIQUE,
Med_name        varchar (31) CONSTRAINT pk_med  PRIMARY KEY,
Med_unitprice   decimal (3,2) CONSTRAINT chk_unitprice CHECK (Med_unitprice < 4.50),
Med_qty_onhand  integer CONSTRAINT nn_medqty not null,
Med_qty_onorder integer,
CONSTRAINT chk_qty CHECK ((Med_qty_onhand + Med_qty_onorder) BETWEEN 1000 AND 3000)
);


CREATE TABLE orders
(Ord_rx#        char (13) CONSTRAINT pk_ord  PRIMARY KEY,
Ord_pat_p#a     char (2) CONSTRAINT nn_ord_pat_p#a not null,
Ord_pat_p#n     char (5) CONSTRAINT nn_ord_pat_p#n not null,
Ord_med_code    char (5) CONSTRAINT fk_med REFERENCES medication (Med_code)
ON DELETE RESTRICT ON UPDATE RESTRICT,
Ord_dosage      smallint DEFAULT 2 CONSTRAINT chk_dosage CHECK (Ord_dosage BETWEEN 1 AND 3),
Ord_freq        smallint DEFAULT 1 CONSTRAINT chk_freq CHECK (Ord_freq IN (1, 2, 3)),
CONSTRAINT fk_pat FOREIGN KEY (Ord_pat_p#a, Ord_pat_p#n)
REFERENCES patient (Pat_p#a, Pat_p#n) ON DELETE CASCADE ON UPDATE CASCADE
);
```

Box 3

- `ALTER TABLE` **Statement:** Modifies an existing table structure. (Ch 10.1.1.2, p.454)
  - `ADD [COLUMN] column_definition`
  - `DROP [COLUMN] column_name {CASCADE | RESTRICT}`
  - `ALTER [COLUMN] column_name SET DEFAULT ... | DROP DEFAULT`
  - `ADD table_constraint_definition`
  - `DROP CONSTRAINT constraint_name {CASCADE | RESTRICT}`
- `DROP TABLE` **Statement:** Deletes a table definition and all its data. (Ch 10.1.1.4, p.456)
  - `DROP TABLE table_name {CASCADE | RESTRICT}`
  - `CASCADE`: Drops the table and also any views, FK constraints, etc., that reference it.
  - `RESTRICT`: Prevents dropping if other objects reference it.
- **Specification of User-Defined Domains:** (Ch 10.1.2, p.462)
  - `CREATE DOMAIN domain_name AS data_type [DEFAULT default_value] [domain_constraint_definition_list];`
  - Allows defining a named domain with specific data type and constraints, which can then be used in column definitions for modularity.

- `ALTER DOMAIN domain_name {ADD domain_constraint | DROP CONSTRAINT name | SET DEFAULT | DROP DEFAULT};`
- `DROP DOMAIN domain_name {CASCADE | RESTRICT};`
  (Examples 1-7, p.462-463, illustrate creating, altering, and dropping domains like 'measure' and 'valid_states').

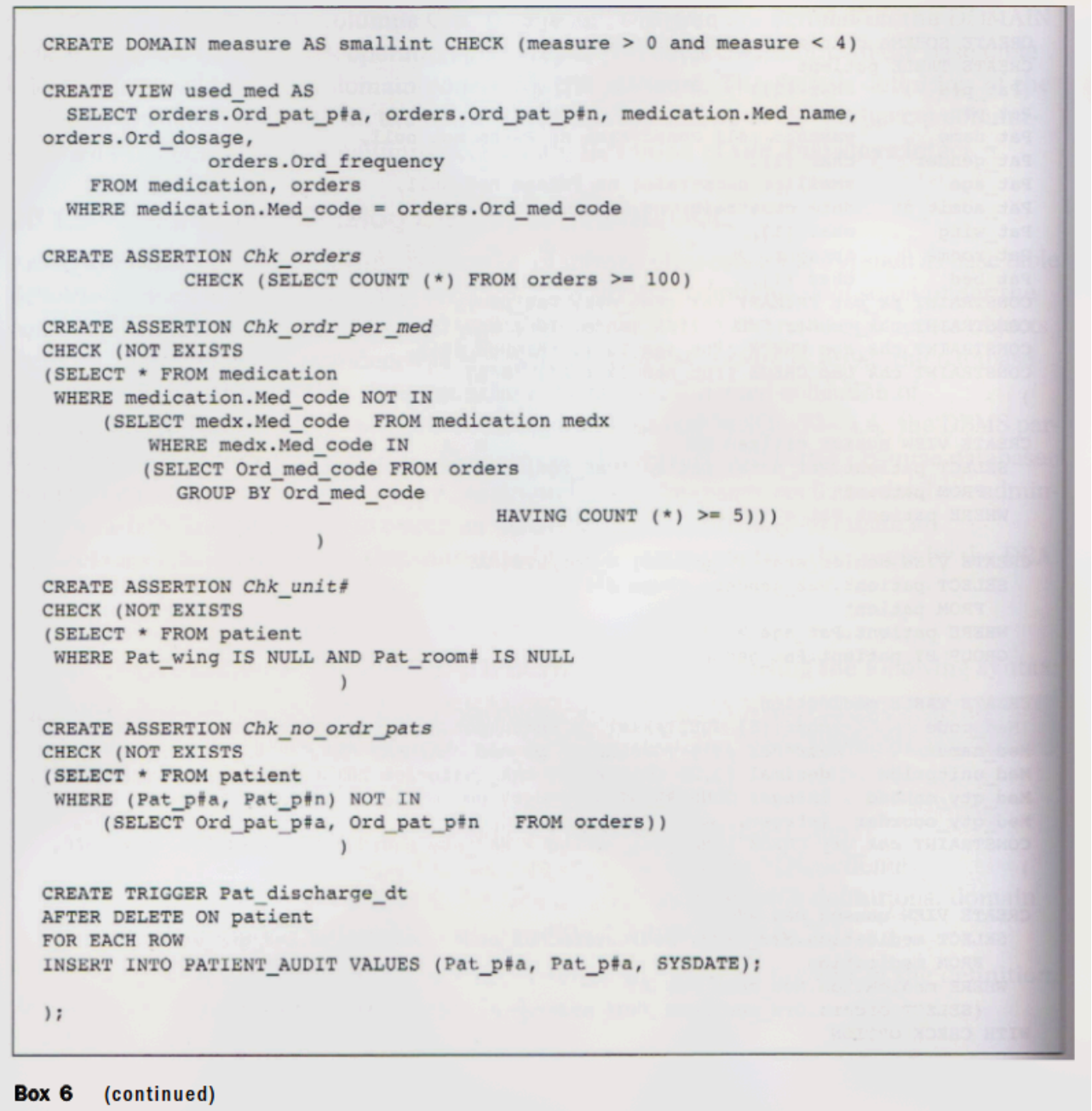
- **Schema and Catalog Concepts in SQL/DDL:** (Ch 10.1.3, p.466)
  - **SQL-Schema:** A named collection of schema elements (tables, views, domains, constraints) under the control of a single user/authorization ID.
  - `CREATE SCHEMA schema_name [AUTHORIZATION user_name] [schema_element_list];`
    (Box 6, p.467-468 for comprehensive example)



```
CREATE SCHEMA clinic AUTHORIZATION Debakey
CREATE TABLE patient
(Pat_p#a      char (2),
Pat_p#n       char (5),
Pat_name      varchar (41) constraint nn_Patnm not null,
Pat_gender    char (1),
Pat_age       smallint constraint nn_Patage not null,
Pat_admit_dt  date constraint nn_Patadmdt not null,
Pat_wing      char (1),
Pat_room#     integer,
Pat_bed       char (1),
CONSTRAINT pk_pat PRIMARY KEY (Pat_p#a, Pat_p#n),
CONSTRAINT chk_gender CHECK (Pat_gender IN ('M', 'F')),
CONSTRAINT chk_age CHECK (Pat_age IN (1 through 90)),
CONSTRAINT chk_bed CHECK (Pat_bed IN ('A', 'B'))
)

CREATE VIEW senior_citizen AS
  SELECT patient.Pat_name, patient.Pat_age, patient.Pat_gender
    FROM patient
  WHERE patient.Pat_age > 64

CREATE VIEW senior_stat (V_gender, V_#ofpats) AS
  SELECT patient.Pat_gender, count (*)
    FROM patient
  WHERE patient.Pat_age > 64
  GROUP BY patient.Pat_gender
```

```
CREATE VIEW senior_stat (V_gender, V_#ofpats) AS
  SELECT patient.Pat_gender, count (*)
    FROM patient
  WHERE patient.Pat_age > 64
  GROUP BY patient.Pat_gender

CREATE TABLE medication
(Med_code         char (5) CONSTRAINT nn_medcd not null CONSTRAINT unq_med UNIQUE,
Med_name          varchar (31) CONSTRAINT pk_med  PRIMARY KEY,
Med_unitprice     decimal (3,2) CONSTRAINT chk_unitprice CHECK (Med_unitprice < 4.50),
Med_qty_onhand    integer CONSTRAINT nn_medqty not null,
Med_qty_onorder   integer,
CONSTRAINT chk_qty CHECK ((Med_qty_onhand + Med_qty_onorder) BETWEEN 1000 AND 3000)
)

CREATE VIEW unused_med AS
  SELECT medication.Med_name, medication.Med_code, medication.Med_qty_onhand
    FROM medication
  WHERE medication.Med_code NOT IN
    (SELECT orders.Ord_med_code FROM orders)
WITH CHECK OPTION

CREATE TABLE orders
 (Ord_rx#      char (13) CONSTRAINT pk_ord  PRIMARY KEY,
Ord_pat_p#a    char (2),
Ord_pat_p#n    char (5),
Ord_med_code   char (5) CONSTRAINT fk_med REFERENCES medication (Med_code)
ON DELETE RESTRICT ON UPDATE RESTRICT,
Ord_dosage        measure,
Ord_freq          measure,
CONSTRAINT fk_pat FOREIGN KEY (Ord_pat_p#a, Ord_pat_p#n)
REFERENCES patient (Pat_p#a, Pat_p#n) ON DELETE CASCADE ON UPDATE CASCADE
)
```

Box 6



```
CREATE DOMAIN measure AS smallint CHECK (measure > 0 and measure < 4)

CREATE VIEW used_med AS
  SELECT orders.Ord_pat_p#a, orders.Ord_pat_p#n, medication.Med_name,
orders.Ord_dosage,
              orders.Ord_frequency
    FROM medication, orders
  WHERE medication.Med_code = orders.Ord_med_code

CREATE ASSERTION Chk_orders
           CHECK (SELECT COUNT (*) FROM orders >= 100)

CREATE ASSERTION Chk_ordr_per_med
CHECK (NOT EXISTS
(SELECT * FROM medication
 WHERE medication.Med_code NOT IN
     (SELECT medx.Med_code  FROM medication medx
         WHERE medx.Med_code IN
        (SELECT Ord_med_code FROM orders
           GROUP BY Ord_med_code
                                    HAVING COUNT (*) >= 5)))
                      )

CREATE ASSERTION Chk_unit#
CHECK (NOT EXISTS
(SELECT * FROM patient
 WHERE Pat_wing IS NULL AND Pat_room# IS NULL
                       )

CREATE ASSERTION Chk_no_ordr_pats
CHECK (NOT EXISTS
(SELECT * FROM patient
 WHERE (Pat_p#a, Pat_p#n) NOT IN
     (SELECT Ord_pat_p#a, Ord_pat_p#n   FROM orders))
                       )

CREATE TRIGGER Pat_discharge_dt
AFTER DELETE ON patient
FOR EACH ROW
INSERT INTO PATIENT_AUDIT VALUES (Pat_p#a, Pat_p#a, SYSDATE);

);
```

Box 6 (continued)

- `DROP SCHEMA schema_name {CASCADE | RESTRICT};`
- **Catalog:** A named collection of SQL-schemas in an SQL environment.
- **INFORMATION_SCHEMA:** A mandatory schema in SQL-92 compliant systems that provides views on the system catalog (metadata).

2. SQL Commands – DML & Views

- **Data Population Using SQL (INSERT):** (Ch 10.2, p.469)
  - **`INSERT INTO table_name [(column_list)] VALUES (value_list);`** (Single-row insert)
    - Values must match column order and data types.
    - If `column_list` is omitted, values must be provided for all columns in their defined order. *(Examples 1 & 2, p.470 for inserting into Patient, Medication, Orders tables).*
  - **`INSERT INTO table_name [(column_list)] subquery;`** (Multi-row insert)
    - Inserts rows selected by the `subquery` into the target table.

        - Column list must match the columns returned by the subquery.
          (Example 3, p.470, inserting from `PATIENT_SUGARLAND` into `PATIENT`).

- **Data Deletion Using SQL (DELETE):** (Ch 10.2.2, p.470)
    - `DELETE FROM table_name [WHERE search_condition];`
    - If `WHERE` clause is omitted, all rows are deleted (table structure remains).
    - Can trigger referential actions (e.g., CASCADE).
      (Example 1, p.471, deleting from `PATIENT` and cascading to `ORDERS`).
- **Data Modification Using SQL (UPDATE):** (Ch 10.2.3, p.472)
    - `UPDATE table_name SET column_name_1 = value_expression_1, ... [WHERE search_condition];`
    - If `WHERE` clause is omitted, all rows are updated.
    - `value_expression` can be a constant, another column, or an expression.
      (Examples 1-3, p.472-473, updating `MEDICATION` unit price).
- **Advanced Data Manipulation using SQL (Ch 11)**
    - **Relational Algebra Concepts (Recap):** Select, Project, Union, Intersection, Difference, Cartesian Product, Join (Equijoin, Natural Join, Theta Join, Outer Joins), Divide, Aggregate Functions. (Ch 11.1, p.493-505) *(Fig 11.2 operators, p.496; Fig 11.6 Venn diagrams, p.502; Fig 11.9 Division, p.505).*

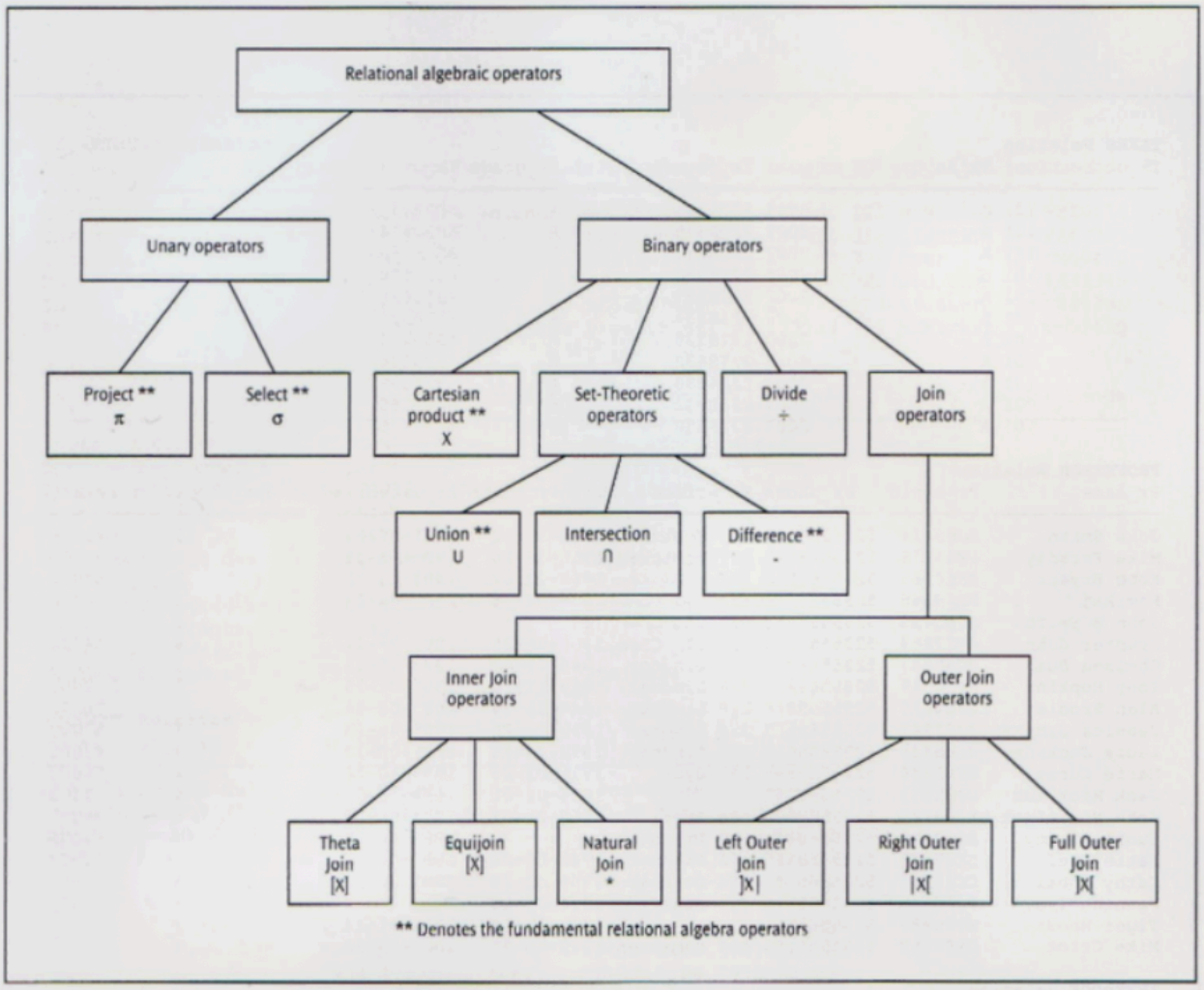


Figure 11.2 Classification of relational algebra operators

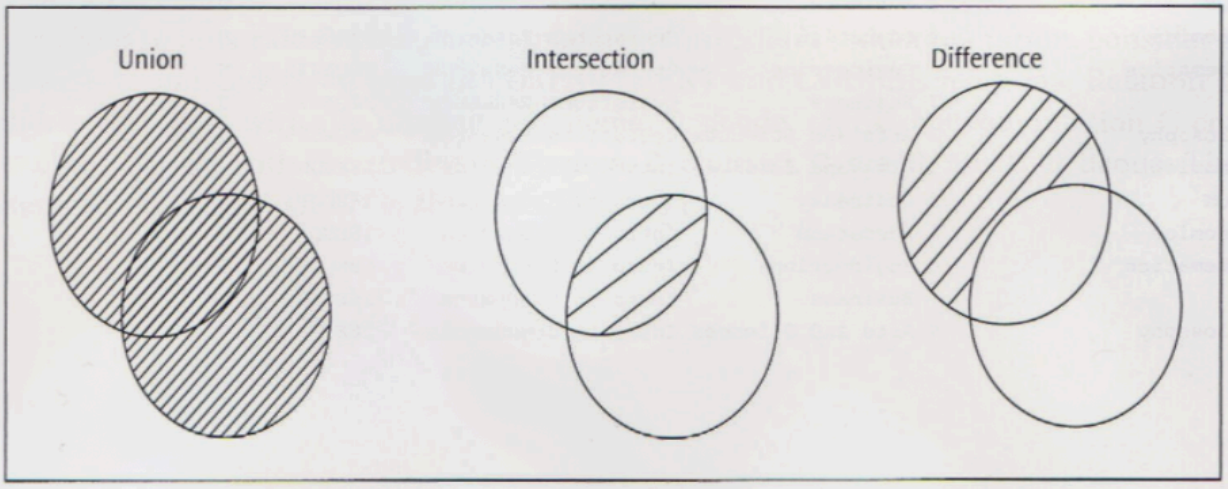


Figure 11.6 The Union, Intersection, and Difference operations

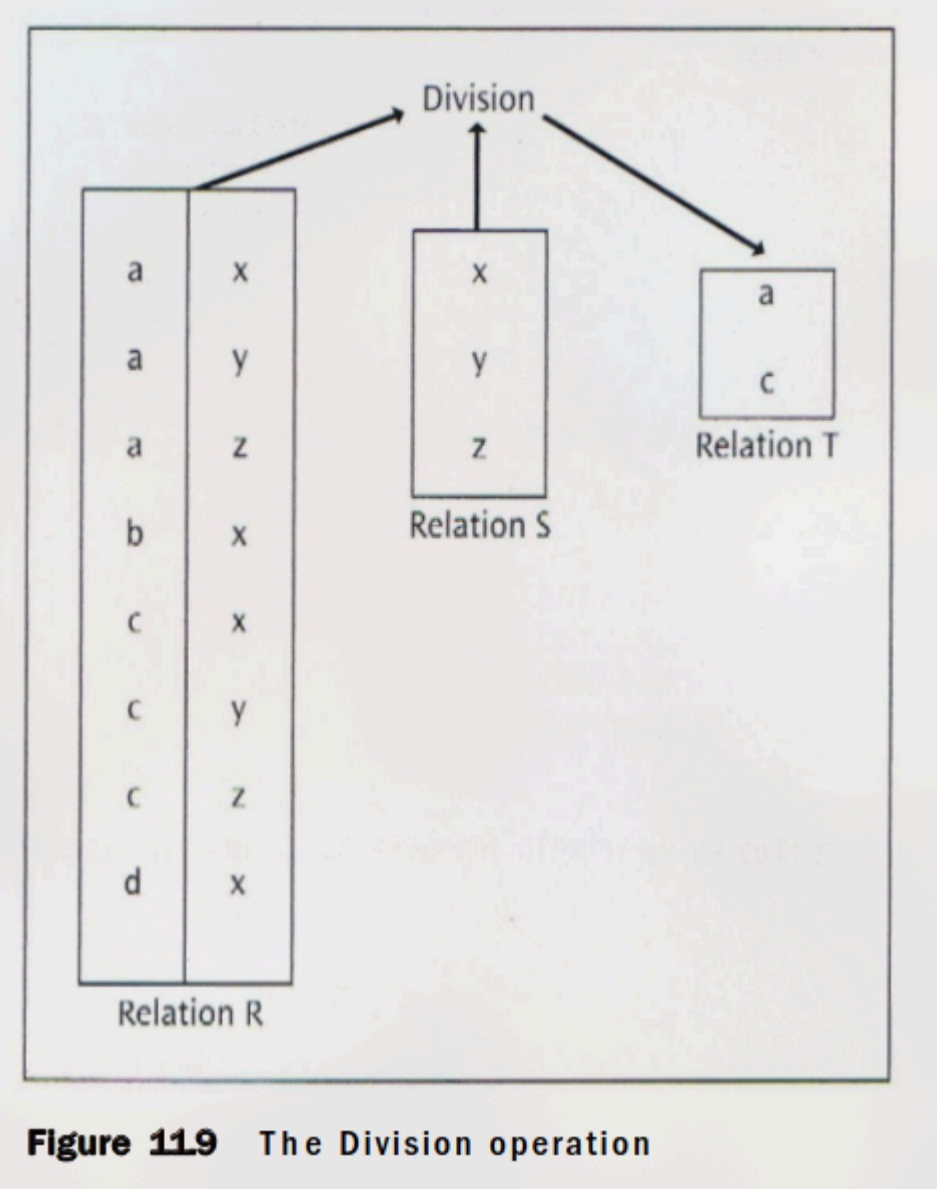


Figure 11.9 The Division operation

- **SQL Queries Based on a Single Table:** (Ch 11.2.1, p.506)
  - `SELECT [DISTINCT | ALL] column_list FROM table_name [WHERE condition] [GROUP BY column_list] [HAVING condition] [ORDER BY column_list [ASC|DESC]];`
  - **Selection (`WHERE`):** Filtering rows. (p.507-509)
  - **Projection (`SELECT column_list`):** Selecting columns. `DISTINCT` removes duplicates. (p.510)
  - **Expressions in `SELECT` and `WHERE`**. (p.511)
  - **`BETWEEN`, `IN`, `NOT IN`, `LIKE` (pattern matching: `%`, `_`, `ESCAPE` character).** (p.512-514)
  - **Handling Null Values (`IS NULL`, `IS NOT NULL`).** Nulls in arithmetic/comparisons yield unknown. Aggregate functions (except `COUNT(*)`) ignore nulls. (p.515-518)
  - **Aggregate Functions (`COUNT`, `SUM`, `AVG`, `MAX`, `MIN`).** (p.518-519)
  - **Grouping (`GROUP BY`) and Filtering Groups (`HAVING`).** `HAVING` applies to groups, `WHERE` to individual rows. (p.519-520)

- **SQL Queries Based on Binary Operators (Joins):** (Ch 11.2.2, p.522)
  - **Cartesian Product (`CROSS JOIN` or comma in `FROM` clause with no join condition).** (p.522-523)
  - **Inner Joins:**
    - `FROM table1 [INNER] JOIN table2 ON join_condition;`
    - `FROM table1 JOIN table2 USING (common_column_list);`
    - `FROM table1 NATURAL JOIN table2;` (joins on all identically named columns)
    - Self Joins (joining a table to itself using aliases). *(p.525)*
    - N-way Joins (joining multiple tables). *(p.526)*
  - **Outer Joins:** Include rows that do not have matching values in the other table.
    - `LEFT [OUTER] JOIN`: Keeps all rows from the left table.
    - `RIGHT [OUTER] JOIN`: Keeps all rows from the right table.
    - `FULL [OUTER] JOIN`: Keeps all rows from both tables.
      (p.528-531 for examples).
  - **Set Theoretic Operators (`UNION`, `INTERSECT`, `MINUS`/`EXCEPT`).** Tables must be union-compatible. `UNION ALL` keeps duplicates. (p.523-524)
- **Subqueries (Nested Queries):** (Ch 11.2.3, p.531) Queries embedded within other queries.
  - **In `WHERE` clause:**
    - Single-row subquery (returns one value): Use with standard comparison operators (=, <, >, etc.).
    - Multiple-row subquery (returns a list of values): Use with `IN`, `NOT IN`, `ANY`, `ALL`.
      - `> ANY` (greater than the minimum), `< ANY` (less than the maximum).
      - `> ALL` (greater than the maximum), `< ALL` (less than the minimum).
      - `= ANY` (equivalent to `IN`).
        (Examples p.532-535).
  - **Correlated Subqueries:** Inner query depends on the outer query for its execution (executed once for each row of outer query). Often uses `EXISTS`, `NOT EXISTS`. *(p.536-539)*
    - `EXISTS`: True if subquery returns at least one row.
  - **In `FROM` clause (Inline Views / Derived Tables).** Subquery result treated as a table. *(p.539)*
  - **In `SELECT` clause (Scalar Subqueries).** Must return a single value. *(p.540)*
  - **In `HAVING` clause.** *(p.540)*
- **Views:** (Ch 6.6 from Unit IV syllabus, Ch 12.1.3 p.542 in textbook)
  - A virtual table based on the result-set of a stored query. Does not store data itself.

- `CREATE VIEW view_name [(column_list)] AS subquery [WITH [CASCADED | LOCAL] CHECK OPTION];`
- **Purpose:** Simplify complex queries, provide security (restrict access to certain rows/columns), present data in a customized way, logical data independence.
- **Updating Views:** Possible only for simple views (usually based on a single table, no aggregates, no `GROUP BY`, no `DISTINCT`). `WITH CHECK OPTION` ensures that DML operations on the view do not cause rows to disappear from the view.
  (Examples 1-4, p.542-543 showing view creation).

3. Database Programming [PYQ Q7a, Q7b]

Database programming refers to the process of writing application programs that interact with a database to store, retrieve, and manipulate data. It involves using a general-purpose programming language (like Java, C#, Python, C++) in conjunction with database access mechanisms (like SQL).

Types/Approaches to Database Programming:

1. **Embedded SQL:**
   - **Description:** SQL statements are directly embedded within the source code of a host programming language. A precompiler processes the source code, converting SQL statements into host language calls to a runtime library.
   - **Example (Conceptual C with Embedded SQL):**

```
EXEC SQL BEGIN DECLARE SECTION;
    char emp_name[50];
    int emp_id_val;
    float emp_salary;
EXEC SQL END DECLARE SECTION;

printf("Enter employee ID: ");
scanf("%d", &emp_id_val);

EXEC SQL SELECT ename, salary INTO :emp_name, :emp_salary
         FROM Employee WHERE empno = :emp_id_val;

if (SQLSTATE == "00000") { // SQLSTATE check for success
    printf("Name: %s, Salary: %.2f\n", emp_name, emp_salary);
} else {
    printf("Employee not found or error.\n");
}
```

   - Used for static SQL where queries are known at compile time.
2. **API-based Database Access (e.g., JDBC, ODBC, ADO.NET):**

- **Description:** Uses an Application Programming Interface (API) – a set of functions, classes, and protocols – provided as a library for a specific programming language. The application makes calls to this API to connect to the database, send SQL statements (often as strings), and process results.
- **Example (Conceptual Java with JDBC):[IMP]**

```
String url = "jdbc:oracle:thin:@localhost:1521:xe";
Connection conn = DriverManager.getConnection(url, "user", "password");
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT ename, salary FROM Employee
WHERE deptno = 10");
while (rs.next()) {
    String name = rs.getString("ename");
    float salary = rs.getFloat("salary");
    System.out.println("Name: " + name + ", Salary: " + salary);
}
rs.close(); stmt.close(); conn.close();
```

- More portable than Embedded SQL, supports dynamic SQL easily. JDBC (Java Database Connectivity) is for Java, ODBC (Open Database Connectivity) is language-independent but common with C/C++.

3. **Stored Procedures and Functions:** (Given below...)
4. **Object-Relational Mappers (ORMs):**
   - **Description:** Frameworks (e.g., Hibernate for Java, SQLAlchemy for Python, Entity Framework for .NET) that map objects in an object-oriented programming language to tables in a relational database. Developers interact with objects, and the ORM handles the SQL generation and database interaction.
   - **Example (Conceptual - interaction, not the ORM code itself):**

```
# Python with an ORM like SQLAlchemy
employee = session.query(Employee).filter_by(id=101).first()
if employee:
    print(f"Name: {employee.name}, Salary: {employee.salary}")
    employee.salary = employee.salary * 1.10 # Give a 10% raise
    session.commit()
```

   - Abstracts away much of the direct SQL, can speed up development but may have a learning curve and performance overhead if not used carefully.
5. **Database-Specific Scripting Languages:**
   - **Description:** Languages provided by the DBMS vendor for database administration and development (e.g., PL/SQL in Oracle, T-SQL in SQL Server, PL/pgSQL in PostgreSQL).

- Used primarily for writing stored procedures, functions, triggers, and administrative scripts directly within the database environment.

- **Embedded SQL & Dynamic SQL:** (Ch 6.1 textbook, p.187-194)
  - **Embedded SQL:** SQL statements embedded directly within a host programming language (e.g., C, Java, COBOL). Requires a precompiler. (Ch 6.1.1)
    - Host variables prefixed with `:`.
    - `EXEC SQL ...` prefix.
    - `SQLCA` (SQL Communication Area) or `SQLSTATE` / `SQLCODE` for error/status checking.
    - `WHENEVER` clause for automatic error handling.
  - **Dynamic SQL:** SQL statements constructed and executed at runtime. Useful when the full SQL statement is not known at compile time.
    - `PREPARE` statement: Parses and compiles an SQL string.
    - `EXECUTE` statement: Executes a prepared statement.
- **Cursors:** (Ch 6.1.2 textbook, p.190-193) [PYQ Q7a]
  - Mechanism to process query results row by row within an application program (bridges set-oriented SQL with row-oriented host languages).
  - **Declaration:** `DECLARE cursor_name [INSENSITIVE] [SCROLL] CURSOR FOR SELECT_statement [ORDER BY ...] [FOR READ ONLY | FOR UPDATE [OF column_list]];`
  - **Operations:**
    - `OPEN cursor_name;`: Executes the query, positions cursor before the first row.
    - `FETCH cursor_name INTO :host_variable_list;`: Retrieves current row, advances cursor.
    - `UPDATE ... WHERE CURRENT OF cursor_name;`: Modifies the current row (if updatable).
    - `DELETE ... WHERE CURRENT OF cursor_name;`: Deletes the current row (if updatable).
    - `CLOSE cursor_name;`: Releases resources.
  - **Properties:** Scrollable, Insensitive (operates on a snapshot), Updatable.
  - **Need for Cursors in Database Programming:** (Ch 6.1.2, p.190)
    SQL is a set-oriented language (it operates on sets of rows), while many general-purpose programming languages (like C, Java, Python) are record-oriented or object-oriented (they process data one record/object at a time). This difference is often called an **impedance mismatch**.
    When an SQL query embedded in a host language program returns multiple rows, the host language typically doesn't have a direct way to handle this entire set of rows at once as a native data structure.
    Cursors bridge this gap. They provide a mechanism for an application program to:

1. Define a set of rows (the result of an SQL query).

2. Iterate through this set, retrieving one row at a time into host language variables for processing.
3. Potentially update or delete the row currently pointed to by the cursor (for updatable cursors).

- **Types of Cursors (Based on Properties and Behavior - often DBMS-specific variations exist):** (Refer to Ch 6.1.2 "Properties of Cursors" p.191-192 for general concepts)
  1. **Read-Only Cursor (Default in many cases):**
     - **Purpose:** Allows only fetching (retrieving) data. No updates or deletions through the cursor are permitted.
     - **Declaration:** Often the default, or explicitly `FOR READ ONLY`.
  2. **Updatable Cursor:**
     - **Purpose:** Allows fetching data and also modifying (`UPDATE ... WHERE CURRENT OF cursor_name`) or deleting (`DELETE ... WHERE CURRENT OF cursor_name`) the row currently pointed to by the cursor.
     - **Declaration:** `FOR UPDATE [OF column_list]`. The `OF column_list` is optional and specifies which columns can be updated through this cursor.
     - **Restrictions:** Typically, the query defining an updatable cursor must be simple (e.g., based on a single table, no aggregates, `GROUP BY`, `DISTINCT`, or complex joins that make row identity ambiguous).
  3. **Scrollable Cursor:**
     - **Purpose:** Allows flexible movement within the result set beyond just fetching the next row.
     - **Declaration:** `SCROLL CURSOR`.
     - **`FETCH` options:** `NEXT` (default), `PRIOR`, `FIRST`, `LAST`, `ABSOLUTE n`, `RELATIVE n`.
     - **Note:** Standard SQL provides for scrollable cursors, but implementation details and the exact fetch options can vary between DBMSs. Scrollable cursors might be read-only by default in some systems if not explicitly updatable.
  4. **Insensitive Cursor (Snapshot Cursor):**
     - **Purpose:** The cursor operates on a temporary copy (snapshot) of the data as it existed when the cursor was opened. Changes made to the underlying base tables by other transactions (or even the same transaction outside the cursor) after the cursor is opened are *not* visible to this cursor. Provides a static view.
     - **Declaration:** `INSENSITIVE CURSOR`.
     - **Behavior:** Useful for consistency if you need to work with a stable set of data, but the data seen by the cursor might become stale.
  5. **Sensitive Cursor:**
     - **Purpose:** The cursor attempts to reflect changes made to the underlying data by other transactions after the cursor was opened. The degree of sensitivity can vary.
     - **Behavior:** More complex to implement by the DBMS. The exact behavior (which changes are visible and when) can be DBMS-dependent. Some changes might make the current row

invalid or cause rows to appear/disappear from the result set.

6. **Keyset-Driven Cursor:** (A type of sensitive cursor, common in some DBMSs)
    - **Purpose:** When opened, the set of keys for the qualifying rows is fixed and stored. The cursor uses these keys to fetch rows.
    - **Behavior:** Changes to non-key values of rows in the keyset are visible. Rows deleted by others will appear as "holes." New rows inserted by others that would qualify are typically not seen.
7. **Forward-Only Cursor (Non-Scrollable):**
    - **Purpose:** The default type. Rows can only be fetched sequentially from the first to the last. No backward movement.
    - **Behavior:** Generally most efficient as it requires fewer resources from the DBMS.
8. **Static Cursor:** Similar to an insensitive cursor; operates on a snapshot.
9. **Dynamic Cursor:** The most "sensitive" type; all committed changes (inserts, updates, deletes) made by others are visible. The order, membership, and values of rows in the result set can change dynamically. Can be resource-intensive.

- **Stored Procedures & Functions:** (Ch 6.5 textbook, p.209-212)
    - Precompiled SQL and procedural statements stored in the database.
    - **Procedures:** `CREATE PROCEDURE proc_name ([param_list]) AS BEGIN ... END;` Can have IN, OUT, INOUT parameters. Called using `CALL` or `EXEC`.
    - **Functions:** `CREATE FUNCTION func_name ([param_list]) RETURNS data_type AS BEGIN ... RETURN value; END;` Must return a value. Can be used in SQL expressions.
    - **Advantages:** Performance (precompiled, reduced network traffic), reusability, security, modularity.
    - **SQL/PSM (Persistent Stored Modules):** Standard for procedural extensions. Includes variables, control structures (IF, LOOP, WHILE), cursors. * **Example (Calling a stored procedure from a host language - conceptual):**

```
// Assume a stored procedure GET_EMP_SALARY(IN emp_id INT, OUT o_salary FLOAT) exists
CallableStatement cstmt = conn.prepareCall("{CALL GET_EMP_SALARY(?, ?)}");
cstmt.setInt(1, 101); // Set IN parameter
cstmt.registerOutParameter(2, Types.FLOAT); // Register OUT parameter
cstmt.execute();
float salary = cstmt.getFloat(2);
System.out.println("Salary: " + salary);
```

    - Reduces network traffic, improves performance, enforces business logic centrally.

- **Exception Handling:** (Part of procedural SQL like PL/SQL, T-SQL, SQL/PSM)

- Mechanism to handle errors or exceptional conditions during execution of stored procedures/blocks.
- `DECLARE exception_name EXCEPTION;`
- `RAISE exception_name;`
- `EXCEPTION WHEN exception_name THEN ...; WHEN OTHERS THEN ...;`

- **Packages (in Oracle PL/SQL):**
  - Schema objects that group logically related PL/SQL types, variables, constants, subprograms, cursors, and exceptions.
  - Consist of a specification (declares public items) and a body (defines public items and private items).
  - Promote modularity, reusability, information hiding, better performance.
- **Triggers:** (Ch 12.1.2 textbook, p.588-595) [PYQ Q7b]
  - Procedural code automatically executed by the DBMS in response to certain DML events (INSERT, UPDATE, DELETE) on a specified table.
  - **ECA Model (Event-Condition-Action):**
    - **Event:** The DML operation that fires the trigger.
    - **Condition:** (Optional) A Boolean expression; trigger fires only if true.
    - **Action:** The PL/SQL block or procedure to be executed.
  - **`CREATE [OR REPLACE] TRIGGER trigger_name {BEFORE | AFTER} {INSERT | DELETE | UPDATE [OF column_list]} ON table_name [FOR EACH ROW] [WHEN (condition)] BEGIN ... END;`**
  - **Row-level trigger (`FOR EACH ROW`):** Fires once for each affected row. Can access `:OLD` and `:NEW` row values.
  - **Statement-level trigger (default):** Fires once per DML statement.
  - **Uses:** Enforcing complex integrity constraints, auditing, maintaining derived data, logging. *(Examples in Ch 12.1.2.2 - 12.1.2.4, p.590-594, like `PATIENT_AFT_DEL_ROW` for auditing, `ORDERS_BEF_INS_ROW` for business rule enforcement).*
    More types of trigger, additional perspectives or specialized:

1. **`INSTEAD OF` Triggers:**
   - **Purpose:** Used primarily with views, especially complex views that are not inherently updatable (e.g., views involving joins, aggregates, `GROUP BY`, `DISTINCT`).
   - **Behavior:** When an DML operation (INSERT, UPDATE, DELETE) is attempted on the view, the `INSTEAD OF` trigger fires *instead* of the DML operation on the view. The trigger's code then defines how to translate this operation into appropriate DML operations on the underlying base tables.
   - **Example:** An `INSTEAD OF INSERT` trigger on a view joining `EMPLOYEE` and `DEPARTMENT` tables might insert data into both base tables correctly.

2. **DDL Triggers:**
    - **Purpose:** Fire in response to DDL events (e.g., `CREATE TABLE`, `ALTER TABLE`, `DROP TABLE`, `CREATE INDEX`).
    - **Behavior:** Used for auditing schema changes, enforcing naming conventions, preventing certain DDL operations, or automating related tasks.
    - **Availability:** Not part of standard SQL, but supported by some DBMSs (e.g., SQL Server, Oracle).
3. **Logon/Logoff Triggers (System-Level Triggers):**
    - **Purpose:** Fire when a user session connects to or disconnects from the database.
    - **Behavior:** Used for auditing user sessions, setting session-specific parameters, or restricting connections based on time or location.
    - **Availability:** DBMS-specific.
4. **Database Event Triggers (Server-Level Triggers):**
    - **Purpose:** Fire in response to database-level events such as startup, shutdown, or specific errors occurring at the server level.
    - **Behavior:** Used for administrative tasks, custom error logging, or initiating recovery procedures.
    - **Availability:** DBMS-specific.
5. **Compound Triggers (Oracle specific):**
    - **Purpose:** Allows defining actions for multiple timing points (BEFORE STATEMENT, BEFORE EACH ROW, AFTER EACH ROW, AFTER STATEMENT) for a single DML event on a table within a single trigger body.
    - **Behavior:** Can simplify logic and share variables across different timing points for the same triggering event.

UNIT-IV: DATABASE TUNING, MAINTENANCE, AND SECURITY

1. Database Tuning and Maintenance

- **Introduction to Database Tuning:** (Ch 20.1, Ch 20.7 textbook, p.650, 669) [PYQ Q9c]
    - Process of optimizing database performance to meet user requirements and service level agreements.
    - Involves adjusting various aspects: physical design, conceptual schema, queries, application code, DBMS parameters, hardware.
    - Iterative process: Monitor -> Identify Bottlenecks -> Diagnose -> Implement Changes -> Measure.
    - **Workload Analysis:** Understanding the types and frequencies of queries and updates is crucial. (Ch 20.1.1, p.651)
        - List of queries and their frequencies.

    - List of updates and their frequencies.
    - Performance goals for each type.
    - Identify accessed relations, attributes, selection/join conditions, and their selectivity.

- **Clustering and Indexing [PYQ Q8a, Q9a]**
    - **Indexing:** Databases can store vast amounts of data. Without indexing, retrieving specific data (e.g., finding an employee with a particular ID, or all products in a certain price range) would require a **full table scan**. This means the DBMS would have to read every single row in the table and check if it matches the query criteria. For large tables, full table scans are extremely slow and inefficient, leading to poor application performance and user experience.

  Indexing provides a shortcut. An index is a separate data structure (usually stored on disk alongside the table data) that:
  1. Is built on one or more columns of a table (the index key).
  2. Stores the index key values and pointers (e.g., rowids, physical addresses) to the actual data rows containing those key values.
  3. Is typically ordered or organized in a way that allows for very fast searching (e.g., using a tree structure like B+-tree or a hash function).

  When a query involves a condition on an indexed column, the DBMS can use the index to quickly locate the relevant rows without scanning the entire table, significantly speeding up data retrieval.

- **Guidelines for Index Selection:** (Ch 20.2, p.653-654)
  (These were listed before but are central to understanding "why" and "how" of indexing, so they are re-emphasized here with a bit more context).
  1. **Guideline 1 (Whether to Index - The Obvious Points):**
     - Index attributes frequently used in `WHERE` clauses for selections and joins. These are the primary candidates for performance improvement.
     - Index attributes used in `ORDER BY` and `GROUP BY` clauses, as indexes can help avoid costly sort operations or facilitate efficient grouping.
     - Do not build an index unless some query (including the query part of DML statements) will benefit from it.
     - Whenever possible, choose indexes that can benefit multiple queries.
  2. **Guideline 2 (Choice of Search Key - For Single Attribute Indexes):**
     - **Exact Match Conditions (e.g., `column = value`):**
       - Both Hash indexes and B+-tree indexes are efficient. Hash indexes are often slightly faster for pure equality searches *if range queries are not also needed on that column*.
     - **Range Conditions (e.g., `column > value`, `column BETWEEN val1 AND val2`):**
       - B+-tree indexes are essential as they store keys in sorted order, allowing efficient retrieval of ranges. Hash indexes are not suitable for range queries.
  3. **Guideline 3 (Multi-Attribute Search Keys - Composite Indexes):**

- Create a composite index on multiple columns if queries frequently have conditions on those columns together (e.g., `WHERE LName = 'Smith' AND FName = 'John'`).
- **Order of columns in a composite index is critical.** An index on `(A, B, C)` can efficiently support queries on `(A)`, `(A,B)`, and `(A,B,C)`. It can also help with queries on `(A,C)` to some extent (by scanning a portion of the index for A), but less so for queries on `(B)` or `(B,C)` or `(C)` alone. Place the most selective attribute (or attribute used in equality first) earlier in the index.

4. **Guideline 4 (Whether to Cluster):**
   - At most one index per table can be a clustered index (where the physical order of data rows matches the index order).
   - Clustered indexes are highly beneficial for:
     - Range queries on the clustered index key.
     - Queries that retrieve many rows in the order of the clustered key.
     - Joins on the primary key if it's clustered, especially if the foreign key table is also accessed in that order.
   - Consider the selectivity of queries and their frequency when choosing which index (if any) to cluster.
5. **Guideline 5 (Hash versus Tree Index - Revisited):**
   - **B+-tree:** Default choice, good for range queries and equality searches. Supports sorted order retrieval.
   - **Hash Index:** Best for equality searches on the exact key. Not good for range queries or partial key searches. Can be very fast for point lookups.
   - Consider if the index is primarily for join operations (index nested loops join) where the indexed relation is the inner relation and the search key includes the join columns.
6. **Guideline 6 (Balancing Cost of Index Maintenance vs. Benefit of Query Speedup):**
   - Indexes speed up `SELECT` queries but slow down DML operations (`INSERT`, `DELETE`, `UPDATE`) because the indexes themselves must also be updated.
   - For frequently updated tables (high write activity), be selective about creating too many indexes.
   - For read-heavy tables (OLAP, data warehouses), more indexes might be justified.
   - Consider dropping indexes if their maintenance overhead outweighs their query performance benefits.

- **Indexing Example:**
  Consider a table `Orders(OrderID (PK), CustomerID, OrderDate, OrderAmount, Status)`.
  - **Scenario 1: Frequent query to find all orders for a specific customer.**
    - `SELECT * FROM Orders WHERE CustomerID = 123;`

        - **Indexing:** Create a B+-tree index on `CustomerID`. `CREATE INDEX idx_orders_custid ON Orders(CustomerID);` This will quickly find all rows for customer 123.
    - **Scenario 2: Frequent query to find orders within a date range.**
        - `SELECT * FROM Orders WHERE OrderDate BETWEEN '2023-01-01' AND '2023-01-31';`
        - **Indexing:** Create a B+-tree index on `OrderDate`. `CREATE INDEX idx_orders_orderdate ON Orders(OrderDate);`
    - **Scenario 3: Frequent query to find orders for a specific customer on a specific date, and the results need to be sorted by OrderAmount.**
        - `SELECT OrderID, OrderAmount FROM Orders WHERE CustomerID = 456 AND OrderDate = '2023-02-15' ORDER BY OrderAmount;`
        - **Indexing:** A composite B+-tree index on `(CustomerID, OrderDate, OrderAmount)` could be very beneficial. `CREATE INDEX idx_orders_cust_date_amt ON Orders(CustomerID, OrderDate, OrderAmount);`
            - The index helps find matching `CustomerID` and `OrderDate` quickly.
            - Because `OrderAmount` is also in the index and is the third component, the data retrieved from the index might already be in the correct order for `OrderAmount` (for matching CustomerID and OrderDate), potentially avoiding a separate sort operation. This could even be an index-only scan if `OrderID` is also part of the index or is the clustered key.
- **Types of Indexing (Index Structures and Properties):**
    1. **Primary Index (often Clustered):**
        - An index whose search key specifies the sequential order of the file (the **ordering key field**).
        - The data file is ordered by this key.
        - There can be at most one primary index for a data file.
        - If the primary index is built on the primary key of the table, it's often called a **clustered index**.
    2. **Clustered Index:**
        - The physical order of the data rows in the table is the same as the order of the index key values.
        - Only one clustered index can exist on a table.
        - Very efficient for range queries on the clustering key and for retrieving rows in the key's order.
        - Example: If `EmployeeID` is the clustered index, employee records are physically stored sorted by `EmployeeID`.
    3. **Secondary Index (Non-Clustered Index):**

- An index whose search key specifies an order different from the sequential order of the file.
- The data rows are *not* physically ordered according to the secondary index key.
- A table can have multiple secondary indexes.
- The index entries point to the actual data rows (e.g., using rowids or by looking up the clustered index key if one exists).
- Slower than clustered indexes for range queries if many rows are retrieved, as it involves more random disk I/O to fetch data rows.

4. **B+-Tree Index:**
    - The most common type of index. A balanced tree structure where all leaf nodes are at the same depth.
    - Leaf nodes contain (key value, pointer to data record/rowid) pairs and are linked sequentially to allow efficient range searching.
    - Supports equality and range queries efficiently.
    - Can be clustered or non-clustered.
5. **Hash Index:**
    - Uses a hash function to compute the address of the bucket/page where the index entry (and possibly data record) for a key value is stored.
    - Very fast for equality searches on the exact hash key.
    - Not suitable for range queries (as hash values are not ordered meaningfully for ranges).
    - Can suffer from collisions (multiple keys hashing to the same bucket), requiring overflow handling.
6. **Unique Index:**
    - An index where the DBMS enforces that no two rows in the table can have the same value for the indexed column(s).
    - Primary key indexes are always unique. Alternate keys are often implemented using unique indexes.
7. **Composite Index (Multi-column Index):**
    - An index created on two or more columns.
    - The order of columns in the index definition is important for query performance.
8. **Covering Index:**
    - A non-clustered index that contains all the columns required to satisfy a query (both in `SELECT` list and `WHERE` clause).
    - Allows the query to be answered entirely from the index (index-only scan), without accessing the base table, which can be very fast.
9. **Bitmap Index:**

- Specialized index type particularly effective for columns with low cardinality (few distinct values, e.g., Gender, Status).
- Uses a bitmap (a sequence of bits) for each distinct value in the indexed column. Each bit corresponds to a row; if the bit is 1, the row has that value.
- Very efficient for complex queries with multiple AND/OR conditions on low-cardinality columns, as bitmaps can be quickly combined using bitwise operations.

10. **Function-Based Index (Expression Index):**
    - An index built on the result of a function or expression applied to one or more columns (e.g., `UPPER(LastName)`, `Salary * 1.1`).
    - Useful when queries frequently use conditions on such expressions.
11. **Spatial Index (e.g., R-tree, Quad-tree):**
    - Used for indexing spatial data types (e.g., points, lines, polygons) to efficiently answer queries like "find all restaurants within 1 mile of this point."
12. **Full-Text Index:**
    - Used for indexing text data (e.g., `VARCHAR(MAX)`, `CLOB`) to support fast searching for words or phrases within the text content.
    - **Index-Only Plans:** (Ch 20.5, p.662) If all columns needed by a query are in an index, the DBMS can answer the query by only accessing the index, not the table. Very efficient.

- **Clustering: [PYQ]** Storing related data physically close together on disk.

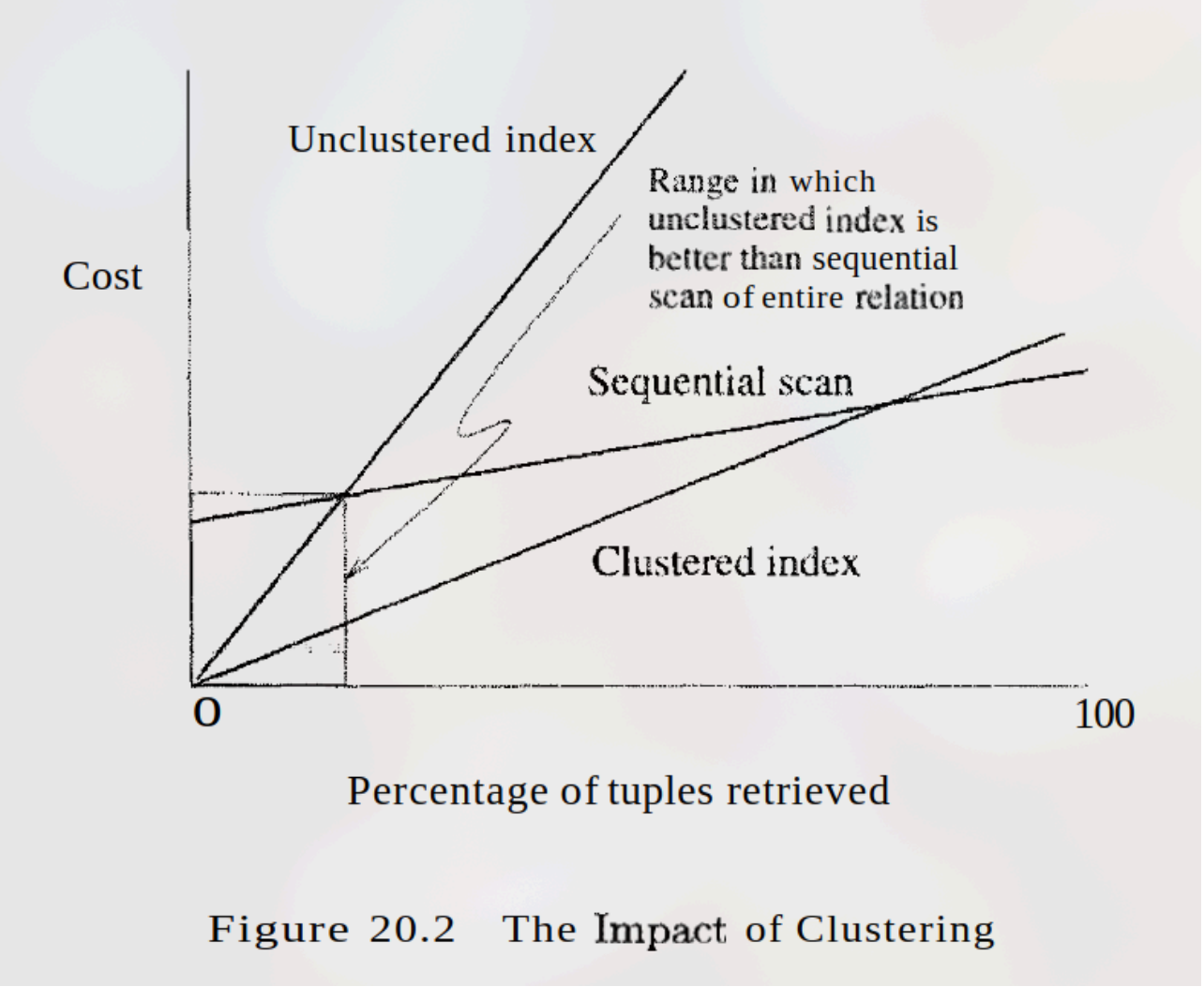


Figure 20.2 The Impact of Clustering

- **Clustering (Database Storage):** Physically storing related data records close together on disk (ideally in the same or adjacent blocks) to minimize Disk I/O when accessing them together.

- **How it Works:** Achieved by organizing data rows based on the values of one or more columns (the clustering key), often via a clustered index.
- **Types of Clustering / Implementations:**
  1. **Clustered Index (Intra-Table Clustering):**
     - **Description:** The physical storage order of data rows in a table matches the logical order of the clustered index key.
     - **Impact:** Only one per table. Excellent for range queries on the clustering key and retrieving data in that key's order.
     - **Choice:** Often the Primary Key or a frequently used range-query column.
  2. **Co-clustering (Inter-Table Clustering / Multi-Table Clustering):**
     - **Description:** Physically interleaving rows from two or more related tables on disk, based on their join key (e.g., PK-FK).
     - **Impact:** Speeds up frequent joins between these tables significantly by reducing I/O needed to fetch related rows.
     - **Example:** Storing `OrderItems` physically near their parent `Order` record.

3. **Hash Clustering:**
    - **Description:** Rows are placed into physical storage buckets based on a hash function applied to a clustering key.
    - **Impact:** Very fast for equality lookups on the hash key. Not suitable for range queries.

- **Benefits of Clustering:**
    - Reduced Disk I/O for accessing related data.
    - Improved query performance, especially for range queries (on clustered index) and joins (with co-clustering).
- **Drawbacks of Clustering:**
    - Higher maintenance overhead for `INSERT`, `DELETE`, `UPDATE` operations, as physical order might need to be maintained (can cause page splits).
    - Only one clustered index per table (for intra-table clustering).
    - Full table scans might be slightly slower if data pages are not as densely packed.
    - **Tools to Assist in Index Selection:** (Ch 20.6, p.663)
        - Database Tuning Advisors / Wizards (e.g., DB2 Index Advisor, SQL Server Index Tuning Wizard).
        - Analyze workload, suggest candidate indexes, estimate benefits ("what-if" analysis).
- **Guidelines for Index Selection (reiteration/summary):**
    - Index primary keys (usually done automatically).
    - Index foreign keys (good for joins).
    - Index attributes frequently used in `WHERE` clauses with high selectivity.
    - Index attributes used in `ORDER BY` and `GROUP BY`.
    - Consider multi-column indexes for queries with multiple conditions.
    - Avoid indexing small tables.
    - Avoid indexing columns that are frequently updated with large values.
    - Avoid indexing columns with very few distinct values (low selectivity).
    - Regularly review and drop unused or ineffective indexes.
- **De-normalization:[PYQ]** (Ch 20.7.2, p.670)
- **Denormalization:** Intentionally adding controlled redundancy to a normalized schema to boost specific query performance by reducing joins. It's the reverse of normalization.
- **Rationale/Use Case:** Highly normalized schemas can lead to slow queries due to many joins. Denormalization pre-joins or duplicates data for critical, frequently run queries that are performance bottlenecks.
- **Trade-off:**
    - **Benefit:** Faster read/query performance for targeted queries.

- **Cost/Drawback:**
  - Increased storage (data redundancy).
  - Risk of update/insert/delete anomalies and data inconsistencies if duplicated data isn't managed carefully.
  - More complex DML operations and application logic to maintain consistency.
- **Guideline:** Apply selectively *after* normalization and *only if* indexing/query tuning isn't sufficient for critical query performance.

- **Types of Denormalization / Techniques:**
  1. **Pre-joining Tables:** Adding attributes from a "one-side" table to a frequently joined "many-side" table.
     - **Example:** Adding `DeptName` from `DEPARTMENT` to `EMPLOYEE` table.
       - **Normalized:** `DEPARTMENT(DeptID, DeptName)`, `EMPLOYEE(EmpID, EmpName, DeptID)`
       - **Denormalized:** `EMPLOYEE_DENORM(EmpID, EmpName, DeptID, DeptName_Dup)`
  2. **Storing Derived/Calculated Values:** Storing pre-computed values (e.g., totals, averages) that are expensive to calculate on-the-fly.
     - **Example:** Adding `OrderTotal` to an `ORDERS` table instead of calculating from `ORDER_ITEMS` each time.
  3. **Combining Tables:** Merging tables with a 1:1 relationship or a very tight, frequently accessed 1:N relationship (few "many" side records per "one" side).
  4. **Repeating Groups (Limited Use):** Using multiple columns for a fixed, small number of similar attributes (e.g., `Phone1`, `Phone2`) instead of a separate table. Generally discouraged due to inflexibility.
  5. **Creating Reporting Tables/Data Marts:** Building separate, denormalized tables specifically for analytical queries and reporting, populated from the normalized operational database.
- **Denormalization Use Case Example (E-commerce Order Summary):**
  - **Requirement:** Fast display of order summary (OrderID, OrderDate, CustomerName, TotalItems, TotalOrderAmount).
  - **Normalized:** Requires joins (`Orders` to `Customers`) and aggregation (`OrderItems`).
  - **Denormalization:** Add `CustomerName_Dup`, `TotalItems_Calc`, `TotalOrderAmount_Calc` to the `Orders` table.
  - **Benefit:** Simple, fast `SELECT` from the denormalized `Orders` table.
  - **Cost:** Updates to `CustomerName` or `OrderItems` require complex logic/triggers to keep duplicated/calculated fields in `Orders` consistent.
- **Database Tuning [PYQ] (Conceptual Schema, Queries, Views):** (Ch 20.7, p.669-672)
  - **Tuning Conceptual Schema:**

        - Settling for a weaker normal form (e.g., 3NF instead of BCNF if BCNF decomposition impacts critical queries). (Ch 20.8.1, p.672)
        - Denormalization (as above).
        - Vertical partitioning (splitting a table's columns into multiple tables).
        - Horizontal partitioning (splitting a table's rows into multiple tables, e.g., by region, date).
    - **Tuning Queries and Views:**
        - Rewriting SQL queries to be more efficient, e.g., avoiding unnecessary joins, using sargable predicates (predicates that can use an index).
        - Optimizing view definitions.
        - Understanding and influencing the query optimizer's plan (e.g., using hints, ensuring statistics are up-to-date).

2. Database Security [PYQ Q8b]

- **Introduction to Database Security:** (Ch 21.1, p.697)
    - Protecting the database against unauthorized access, modification, or destruction.
    - **Objectives:**
        - **Secrecy/Confidentiality:** Preventing disclosure of information to unauthorized users.
        - **Integrity:** Ensuring data is accurate and consistent, and preventing unauthorized modification.
        - **Availability:** Ensuring authorized users can access data when needed.

[PYQ Q8b] Security considerations should be integrated throughout the database modeling and design process, not just as an afterthought.

1. **Conceptual Level (ER/EER Modeling):**
    - **Identify Sensitive Data:** During requirements gathering, clearly identify which entities and attributes contain sensitive information that requires protection (e.g., salaries, personal health information, credit card numbers).
    - **Define Access Privileges Conceptually:** Understand different user roles and what data they *should* be able to see or modify. This doesn't mean drawing security in the ERD, but documenting these requirements.
    - **Consider Views for Abstraction:** If certain user groups only need to see a subset of attributes or aggregated data from an entity, this can be noted conceptually, leading to view creation later.
    - **Data Ownership:** Clarify who "owns" different pieces of data, as this often dictates who can grant access.
2. **Logical Level (Relational Schema Design):**
    - **View Design for Security:** Design specific views that expose only necessary columns and/or rows to particular user roles.

        - **Column-level security:** A view selects only a subset of columns from a base table.
        - **Row-level security (Value-Dependent Access Control):** A view's `WHERE` clause filters rows based on user identity or attributes (e.g., a sales manager only sees orders for their region).
    - **Granular Privileges:** Plan for the types of privileges (`SELECT`, `INSERT`, `UPDATE`, `DELETE`, `REFERENCES`) that will be needed on tables and views.
    - **Separation of Duties:** Design the schema such that critical operations might require different roles or data access, reducing the risk of a single point of compromise.
3. **Physical Level (Implementation in DBMS):**
    - **GRANT/REVOKE (Discretionary Access Control - DAC):** Implement the conceptually defined access privileges using `GRANT` and `REVOKE` SQL statements on tables, views, stored procedures, etc.
    - **Role-Based Access Control (RBAC):** Create roles, grant privileges to roles, and then grant roles to users. This simplifies administration.
    - **Stored Procedures for Controlled Access:** Encapsulate DML operations within stored procedures. Grant users `EXECUTE` permission on the procedure, but not direct DML access to the underlying tables. The procedure can implement complex validation and security checks.
    - **Triggers for Auditing and Complex Constraints:** Use triggers to log access to sensitive data or to enforce complex security rules that cannot be defined by simple constraints.
    - **Mandatory Access Control (MAC) (if supported and required):** For high-security environments, implement MAC by assigning security labels to data and clearance levels to users (e.g., Bell-LaPadula model).
    - **Encryption:** Encrypt sensitive data at rest (in the database files) and in transit (over the network).
        - Column-level encryption for specific sensitive attributes.
        - Transparent Data Encryption (TDE) encrypts entire database files.
    - **Auditing:** Configure DBMS auditing features to log database activities, especially access to sensitive data or DDL changes.
    - **Authentication:** Strong mechanisms for verifying user identities (passwords, multi-factor authentication).
    - **Network Security:** Secure the network communication channels to the database server (e.g., using SSL/TLS).

- **Access Control:** (Ch 21.2, p.698) Mechanisms to control who can access what data and perform what operations.
    - **Discretionary Access Control (DAC):** [PYQ - Implied in GRANT/REVOKE] (Ch 21.3, p.699)
        - Based on privileges (access rights) granted to users or roles by object owners or DBAs.
        - Objects: Tables, views, columns, etc.

- Privileges: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `REFERENCES`, `USAGE`, etc.
- **`GRANT` Statement:** `GRANT privilege_list ON object_name TO user_list [WITH GRANT OPTION];`
  - `WITH GRANT OPTION`: Allows the grantee to further grant the privilege.
- **`REVOKE` Statement:** `REVOKE [GRANT OPTION FOR] privilege_list ON object_name FROM user_list [CASCADE | RESTRICT];`
  - `CASCADE`: Revokes privilege from user and any users they granted it to.
  - `RESTRICT`: Fails if privilege was passed on.
- **Roles:** A named group of privileges. Privileges granted to role, role granted to users. Simplifies privilege management. `CREATE ROLE`, `DROP ROLE`, `GRANT role TO user`. *(Authorization graph concepts for GRANT/REVOKE, Ch 21.3, p.701-702).*

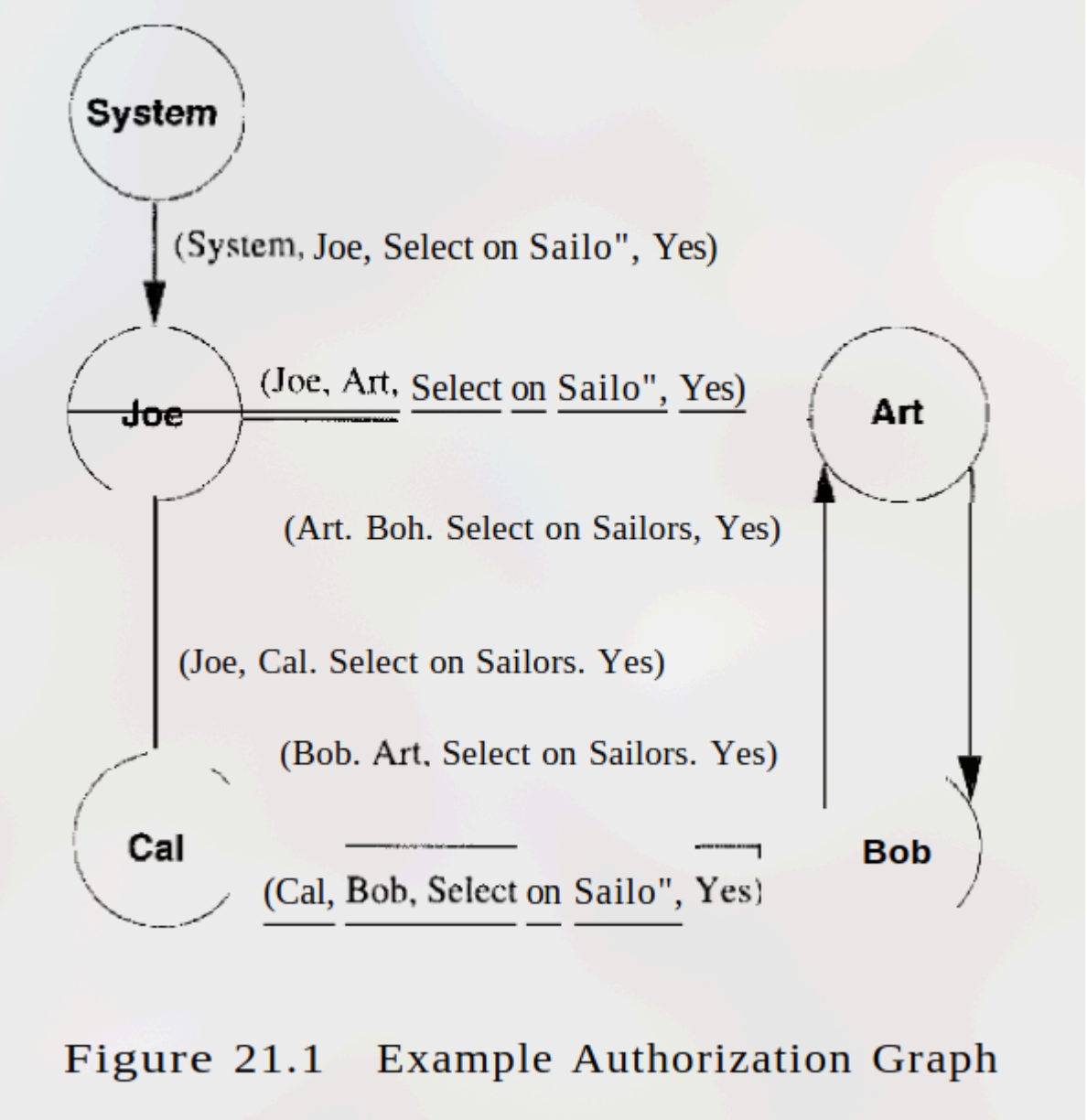


Figure 21.1 Example Authorization Graph

(Views as a security mechanism - restricting access to subset of data, Ch 21.3.1, p.704).

- **Mandatory Access Control (MAC):** (Ch 21.4, p.705)
  - Based on system-wide policies, not at owner's discretion.

    - Uses security classifications (labels) for data objects (e.g., Top Secret, Secret, Confidential, Unclassified) and clearance levels for subjects (users/processes).
    - **Bell-LaPadula Model:**
        - Simple Security Property (No Read Up): Subject S can read object O only if `class(S) >= class(O)`.
        - *-Property (Star Property - No Write Down): Subject S can write object O only if `class(S) <= class(O)`.
    - **Multilevel Relations and Polyinstantiation:** (Ch 21.4.1, p.707) Storing data with different security levels in the same table, potentially leading to different users seeing different versions of a row (polyinstantiation) to avoid covert channels.
    - Covert Channels: Indirect ways of inferring higher-level information.
    - DoD Security Levels (e.g., C2, B1).
- **DCL Commands (GRANT, REVOKE):** Covered under DAC.
- **Views as Security Mechanism:** (Ch 6.6, Ch 12.1.3, Ch 21.3.1)
    - Views can restrict users to specific rows (via `WHERE` clause in view definition) and specific columns (via `SELECT` list in view definition) of underlying base tables.
    - Users are granted privileges on the view, not directly on the base tables.

Unit 3 and 4 are smaller compared to Unit 1 and 2, easier to complete. Same as DBMS