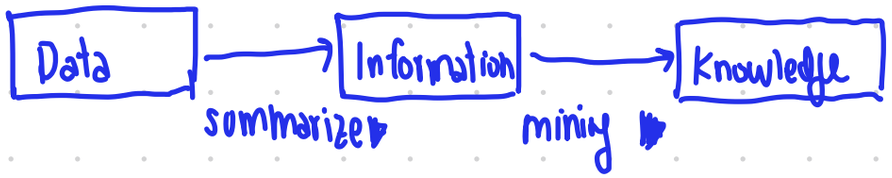
CS5325 Database System Midterm Summary

Introduction (L1)

Type of DB Application
- ① Traditional Application → Numerical & Textual DB
- ② Recent Application → Multimedia / GIS / Warehouses

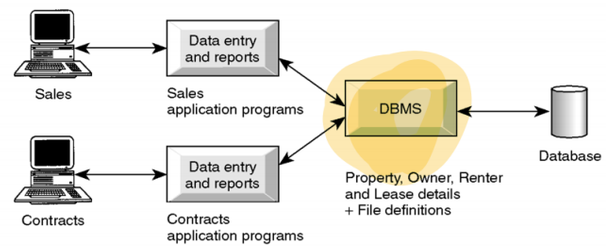
Data & Information



File-based System
- → decentralized / predecessor of DBS
- → each program defines & manages its own data

Critique!
- ① Data Management : extensive programming (3GL) / Time consuming / island of information
- ② Data dependence : file structure defined in program code
- ③ Structural dependence : change in file structure require change in program.
- ! ④ Data Redundancy : Different & confilting version of same data.
- ⑤ Seperation & Isolation of data ⑥ Incompatible format. ⑦ Fixed Queries

Database Management System (DBMS)
- → Database is shared / integrated
  - ↳ end user data / metadata
  - ↳ Manage structure / Control access. / Query Language



database : shared collection of logically related data (& description of it), design to meet info. need of organization
- → promote: data independence → "System Catalog / Metadata / dictionary) → provide description of data
- → comprises of entities / attributes / relationship.

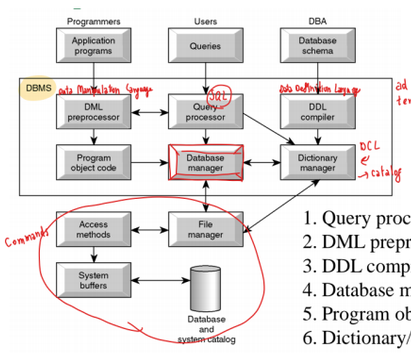
- database system: A system that occupies a database as basic stroage.
- DBMS: A software system that enables users to define/create/maintain database which provides controlled access to this database

• Typical DBMS Functionality
→ define: data types/structures/constraints
→ construct/Load: initial database contents on secondary medium.
→ Manipulate: Retrieval: Query / Modification: Insert/Delete/Update/Access

• DBMS Functions
① Stroage/Retrieval/Update
② User-accessible catalog (description of data)
③ Transaction support ┌ all updates are made
└ none update are made
④ Concurrency: do query in sequential control
⑤ Recovery service: Rollback/Rollforward
⑥ Authorization & Security management
⑦ Integrity services ⑧ Data Commu.
!⑨ Promote data independence

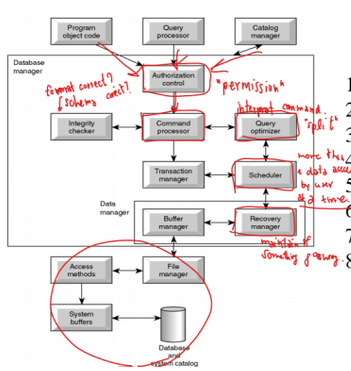
* VIEW Mechanism
→ provides user w/ only data they want/need
SQL: CREATE VIEW S1 AS
(Query: SELECT)

• Components of DBMS



1. Query processor
2. DML preprocessor
3. DDL compiler
4. Database manager (DM)
5. Program object code
6. Dictionary/Catalog manager

• Database Manager

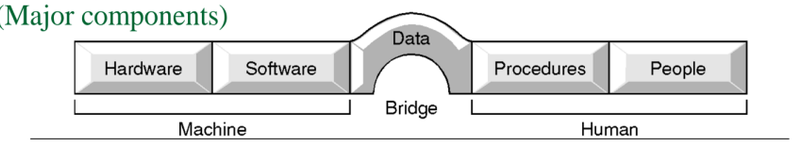


1. Authorization control
2. Command processor
3. Integrity checker
4. Query optimizer
5. Transaction manager
6. Scheduler
7. Recovery manager
8. Buffer manager

• Advantages
- Control Redundancy
- Control access to data
- Persistent Stroage
- Index for efficient query processing
- Backup & Recovery

• Disadvantage
- Complexity - Size
- * Higher impact of failure.

DBMS Environment



- USERS ① Data Admin :→ manage of data resource + planning / dev / maintainance of design
  ② Database Admin :→ manage physical realization of a database application + set security monitor etc.
  ③ Database Designer ④ Programmer ⑤ End users

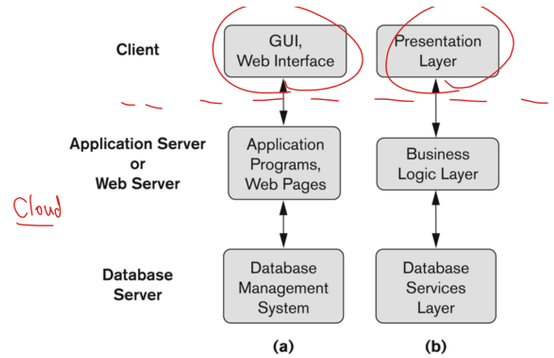
DBMS Server

provide db query & transaction services
Relational DBMS → SQL server
Application on client utilize APIs to access server database.

Architecture

① 2-tier → user can connect to several data sources

② 3-tier → web application.
[+ Application Server / Web Server]
Enhance security



Data Independence

Logical Data Independence
→ can change conceptual schema w/o change to external / associate program
(add / remove entities)

Physical Data Independence
→ change internal w/o change conceptual.
(using different file organization)

Data model

→ concepts to describe structure of DB / operations / constraints

Database Language
① DDL : Definition
② DML : Manipulation

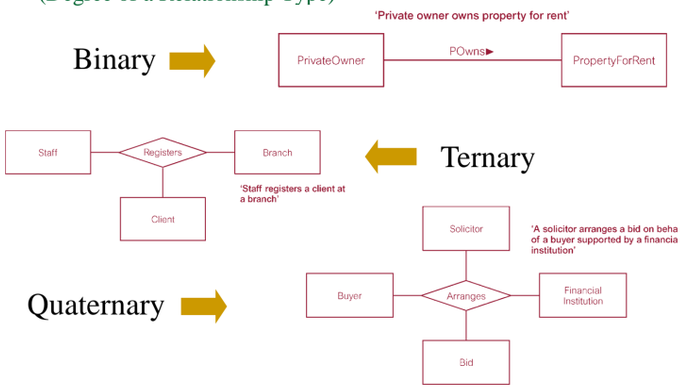
Model → ① Structural Part.
② Manipulative.
③ Integrity rules

Entity-Relationship Modeling (L2)

Entity → group of object w/ same properties / relationship → set of meaningful association / Attribute → property of entity.

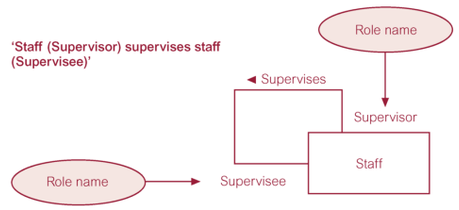
ER Diagram → entity — relationship — entity

Binary, Ternary and Quaternary
(Degree of a Relationship Type)



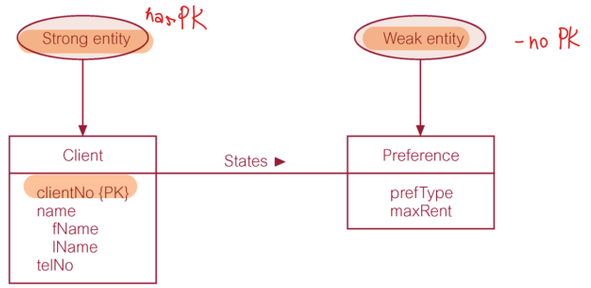
Recursive Relationship Types

- Relationship type where same entity type participates more than once in different roles.
- Relationships may be given role names to indicate purpose that each participating entity type plays in a relationship.

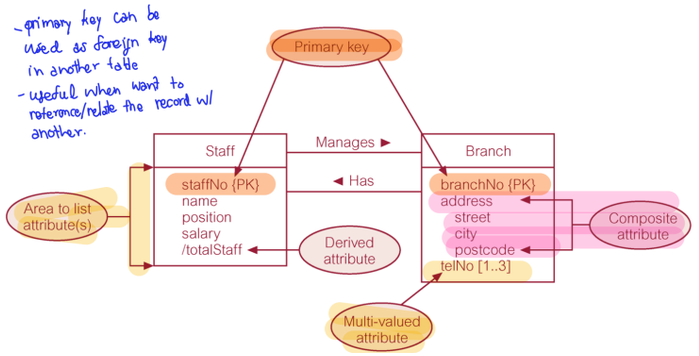


Two Kinds of Entity Types

- Strong Entity Type
  - Entity type that is not existence-dependent on some other entity type.
- Weak Entity Type
  - Entity type that is existence-dependent on some other entity type.



An Example of ER Diagram
(entities, attributes, relationships, primary keys)



* Keys

① Candidate key
attributes that uniquely identifies records

② Primary key
candidate key to identify each unique record.

③ Composite key
candidate key consists of 2 or more attribut

④ Superkey
set of attr. that uniquely identify records.

⑤ Alternate key
what can be primary key alternatively

Multiplicity → constraint of relationship.
↳ range of possible occurence of entity type related to a particular relationship.

- The most common degree for relationships is binary.
  - one-to-one (1:1)
  - one-to-many (1:*)
  - many-to-many (*:*)

"Cardinality" → maximum possible occurrences 0..1 1..1 0..* 1..*

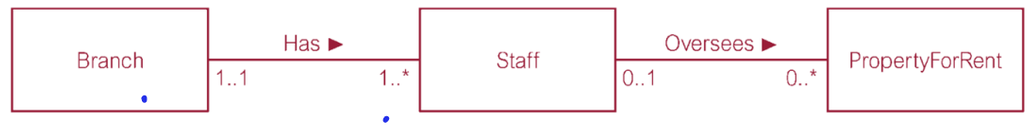
"Participation" → minimum occurrences

Problems w/ ER models

① Fan Trap

Solve

② Chasm trap



pathway b/w entity is ambiguous) suggest existence but pathway doesn't exist.

Specialization & Generalization

Superclass / Subclass

① participation constraints

→ Optional → doesn't always be in subclass

→ Mandatory → must be in subclass

② disjoint constraints

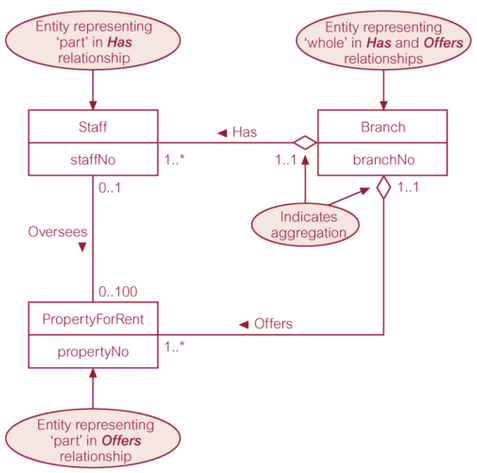
→ AND → more than or equal to 1 subclass

→ OR → only 1 subclass

{participation, disjoint} : {optional, OR}, {mandatory, OR} ---

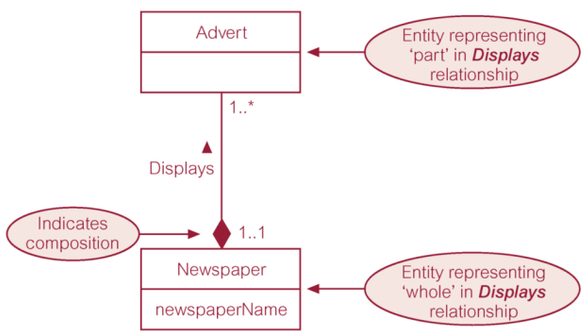
Aggregation

- Represents a 'has-a' or 'is-part-of' relationship between entity types, where one represents the 'whole' and the other 'the part'.



Composition

- Specific form of aggregation that represents an association between entities, where there is a strong ownership and coincidental lifetime between the 'whole' and the 'part'.

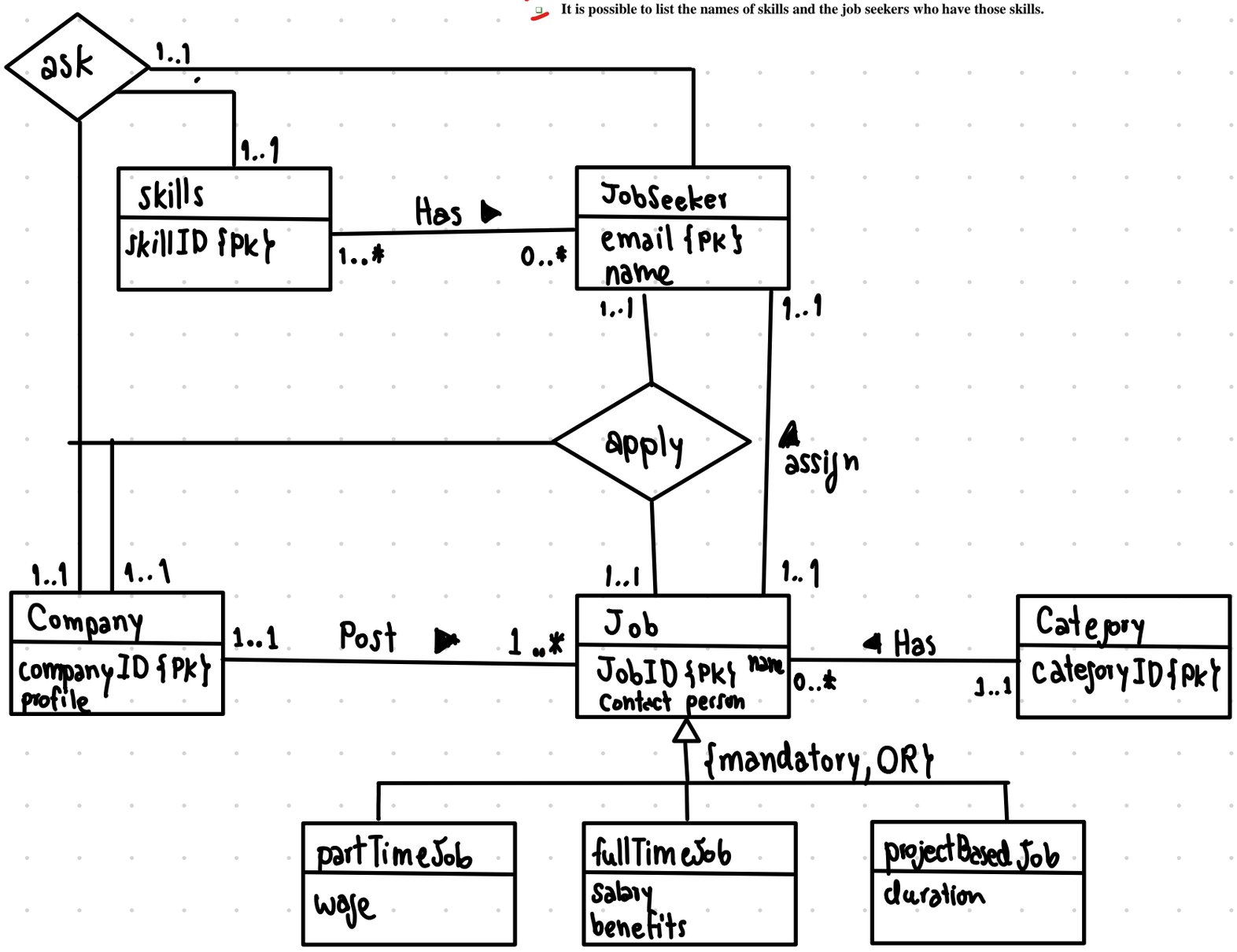


Construct EER model

- i'm not sure whether this is correct or not

Construct an EER model for a database for a recruitment company. This database keeps information of persons who want to find a job, as well as information of companies that are finding some qualified employment candidates. Some detail requirements are given below. Note that the EER model needs to include the primary key for each entity as well as multiplicity (cardinality and participation) for each relationship. (16 marks)

- **The model stores the personal information of a job seeker (a person who is finding a job).**
- **The model stores the company profile of a company which is finding some personnel.**
- **A company can post a set of jobs it provides.**
- **A job seeker can post a set of skills he/she has.**
- **There are three types of jobs: part-time jobs, full-time jobs and project-based jobs.**
- **Each job type may have different information. For example, a part time job has wage information, a full-time job has salary and other benefit information and a project-based job has project duration (period).**
- **Jobs are organized into categories, e.g. 'Computer', 'Construction', 'Food', 'Art' and so on.**
- **A job seeker can apply for a job that a company posted.**
- **A company can ask a job seeker who has a skill that the company requires.**
- **Each job can be assigned to only one job seeker.**
- **It is possible to list the names of jobs and their contact person.**
- **It is possible to list the names of skills and the job seekers who have those skills.**



Data Manipulation SQL (L3+4)

SELECT Statement

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

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.
SELECT	Specifies which columns are to appear in output.
ORDER BY	Specifies the order of the output.

(*) → all columns.

→ condition (WHERE)

=	equal	<>	not equal
>	Greater	<	less
>=	Greater or equal	<=	less or equal

IN / BETWEEN __ AND __

Membership

IN (in set)
NOT IN (not in set)

Strings!

wildcards

LIKE % anything (0 and more character)
_ any1 character

"Pattern matching"

LIKE '%a%' → any strings contain a
'%a' → any strings ends w/ a
'a%' → — " — starts w/ a
'__a%' → any strings that has 3rd Character is a
'%a_' → second-to-last character is a

ORDERING

DESC → descending
ASC → ascending (default)

Aggregates functions

COUNT	returns number of values in a specified column.
SUM	returns sum of values in a specified column.
AVG	returns average of values in a specified column.
MIN	returns smallest value in a specified column.
MAX	returns largest value in a specified column.

- Aggregate functions can be used only in **SELECT** list and in **HAVING** clause.

Sub queries

- List staff who work in branch at '163 Main St'.

```
SELECT staffNo, fName, lName, position
FROM Staff
WHERE branchNo IN
      (SELECT branchNo
       FROM Branch
       WHERE street = '163 Main St.');
```

subquery

Subquery Rules

- **ORDER BY** clause may not be used in a subquery (although it may be used in outermost **SELECT**).
- Subquery **SELECT** list must consist of a single column name or expression, except for subqueries that use **EXISTS**.
- By default, column names refer to table name in **FROM** clause of subquery. Can refer to a table in **FROM** using an *alias*.
- When subquery is an operand in a comparison, subquery must appear on right-hand side.
- A subquery may not be used as an operand in an expression.

COUNT, SUM, MIN, MAX, AVG

- Find number of Managers and sum of their salaries.

```
SELECT COUNT(staffNo) AS count,
       SUM(salary) AS sum
FROM  staff
WHERE position = 'Manager';
```

count	sum
2	54000.00

- Find minimum, maximum, and average staff salary.

```
SELECT MIN(salary) AS min,
       MAX(salary) AS max,
       AVG(salary) AS avg
FROM staff;
```

min	max	avg
9000.00	30000.00	17000.00

Use of GROUP BY

- Find the number of staff in each branch and their total salaries.

```
SELECT branchNo,
       COUNT(staffNo) AS count,
       SUM(salary) AS sum
FROM Staff
GROUP BY branchNo
ORDER BY branchNo;
```

ANY and ALL

- **ANY** and **ALL** may be used with subqueries that produce a single column of numbers.
- If subquery preceded by **ALL**, condition will only be true if it is satisfied by all values produced by subquery.
- If subquery preceded by **ANY**, condition will be true if it is satisfied by any values produced by subquery.
- If subquery is empty, **ALL** returns true, **ANY** returns false.
- ISO standard allows **SOME** to be used in place of **ANY**.

Use of ANY/SOME

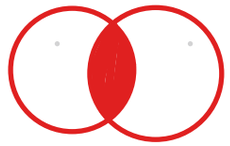
- Find staff whose salary is larger than salary of at least one member of staff at branch B003.

```
SELECT staffNo, fName, lName, position, salary
FROM   staff
WHERE  salary > SOME (SELECT salary FROM staff
                      WHERE  branchNo = 'B003');
```

>12000
ALL >24000

Multiple table queries
⇒ JOIN

Simple Join



- List names of all clients who have viewed a property along with any comment supplied.

```
SELECT c.clientNo, fName, lName,
       propertyNo, comment
FROM   client c, viewing v
WHERE  c.clientNo = v.clientNo;
```

Equivalent to equi-join in relational algebra.

```
FROM Client c JOIN Viewing v ON c.clientNo = v.clientNo
FROM Client JOIN Viewing USING clientNo
FROM Client NATURAL JOIN Viewing
```

Right Outer Join

- List branches and properties in same city and any unmatched properties.

```
SELECT b.*, p.*
FROM branch1 b RIGHT JOIN
     propertyForRent1 p ON b.bCity = p.pCity;
```

- Right Outer join includes those rows of second (right) table that are unmatched with rows from first (left) table.
- Columns from first table are filled with NULLs.

Outer Joins (Left Outer Joins)

- Result table has two rows where the cities are the same.
- There are no rows corresponding to branches in Bristol and Aberdeen.
- To include unmatched rows in the result, use an outer join.
- Ex.: List branches and properties that are in same city along with any unmatched branches.

```
SELECT b.*, p.*
FROM branch1 b LEFT JOIN
     propertyForRent1 p ON b.bCity=p.pCity;
```

Full Outer Join

- List branches and properties in same city and any unmatched branches or properties.

```
SELECT b.*, p.*
FROM branch1 b FULL JOIN
     propertyForRent1 p ON b.bCity = p.pCity;
```

- Includes rows that are unmatched in both tables.
- Unmatched columns are filled with NULLs.

Query using EXISTS

- Find all staffs who work in a London branch.

```
SELECT staffNo, fName, lName, position
FROM   staff s
WHERE EXISTS
      (SELECT * FROM Branch b
       WHERE s.branchNo = b.branchNo AND
             city = 'London');
```

Use of UNION

- List all areas where there is either a branch or a property.

```
(SELECT city
 FROM   branch
 WHERE  city IS NOT NULL )
UNION
(SELECT city
 FROM   propertyForRent
 WHERE  city IS NOT NULL );
```

```
(SELECT *
 FROM   branch
 WHERE  city IS NOT NULL )
UNION CORRESPONDING BY city
(SELECT *
 FROM   propertyForRent
 WHERE  city IS NOT NULL );
```

Use of INTERSECT

- List all cities where there is both a branch office and a property.

```
(SELECT city FROM Branch)
INTERSECT
(SELECT city FROM PropertyForRent);
```

```
(SELECT * FROM Branch)
INTERSECT CORRESPONDING BY city
(SELECT * FROM PropertyForRent);
```

city

Use of EXCEPT

- List of all cities where there is a branch office but no properties.

```
(SELECT city FROM Branch)
   EXCEPT
(SELECT city FROM PropertyForRent);
```

Or

```
(SELECT * FROM Branch)
EXCEPT CORRESPONDING BY city
(SELECT * FROM PropertyForRent);
```

INSERT

INSERT

```
INSERT INTO TableName [ (columnList) ]
   VALUES (dataValueList)
```

- columnList is optional.
- If omitted, SQL assumes a list of all columns in their original **CREATE TABLE** order.
- Any columns omitted must have been declared as **NULL** when table was created, unless **DEFAULT** was specified when creating column.

UPDATE

UPDATE

```
UPDATE TableName
SET columnName1 = dataValue1
    [, columnName2 = dataValue2...]
[WHERE searchCondition]
```

DELETE

DELETE

```
DELETE FROM TableName
[WHERE searchCondition]
```

→Schema

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)] [,…] })
```

DROP TABLE

```
DROP TABLE TableName [RESTRICT | CASCADE]
```

- For example,

```
DROP TABLE PropertyForRent;
```

ALTER TABLE (III)

- Remove constraint from PropertyForRent that staff are not allowed to handle more than 100 properties at a time.
- Add new column to Client table.

```
ALTER TABLE PropertyForRent
   DROP CONSTRAINT StaffNotHandlingTooMuch;

ALTER TABLE Client
   ADD prefNoRooms INT;
```

```
ALTER TABLE TableName
{ADD [COLUMN] colName dataType [NOT
  NULL][UNIQUE]
[DEFAULT defaultOption]
[CHECK searchCondition] [,...]}
[DROP [COLUMN] colName [RESTRICT|CASCADE]]
[ADD [CONSTRAINT [ConstrName]] tblConstrDef]
[DROP CONSTRAINT ConstrName [RESTRICT|CASCADE]]
[ALTER [COLUMN] SET DEFAULT DefaultOption]
[ALTER [COLUMN] DROP DEFAULT]
```