Objectives

In this Lecture you will learn:

- How to derive a set of relations from a conceptual data model.
- How to validate a logical data model to ensure it supports the required transactions.
- How to ensure that the final logical data model is a true and accurate representation of the data requirements of the enterprise.

1. Introduction

In the previous lecture we introduced a methodology that describes the steps that make up the three phases of database design and then presented Step 1 of this methodology for conceptual database design.

In this lecture we describe Step 2 of the methodology, which translates the conceptual model produced in Step 1 into a logical data model.

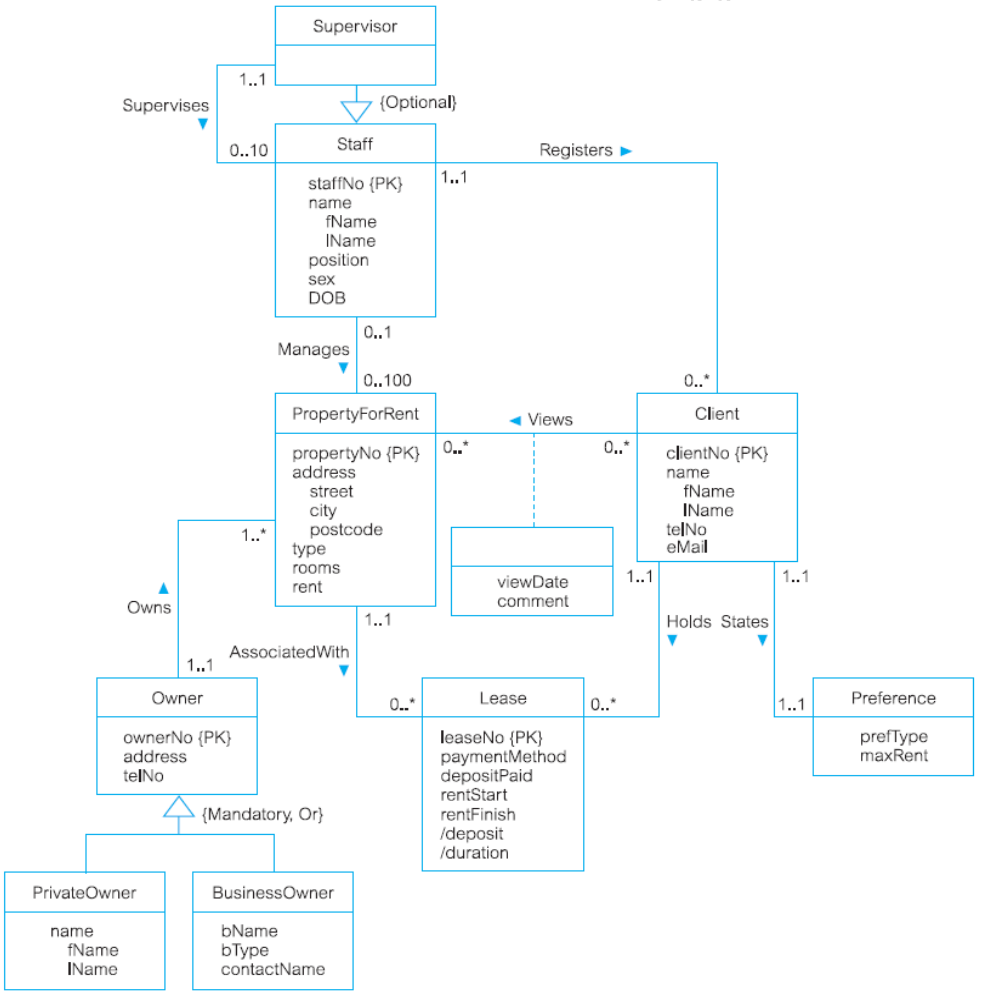


Fig. 1 . The final ER of Conceptual Database Desing steps.

2. Logical Database Design Methodology for the Relational Model

OBJECTIVE:

- To translate the conceptual data model into a logical data model.
- To validate this model and check that it is structurally correct and able to support the required transactions.

This objective is achieved by following these activities:

Step 1: Derive relations for logical data model.
Step 2: Check integrity constraints.
Step 3: Validate relations against user transactions.
Step 4: Review logical data model with user.

Step 1: Derive relations for logical data model

Objective: To create relations for the logical data model to represent the entities, relationships, and attributes that have been identified.

- The relationship that an entity has with another entity is represented by **the primary key/ foreign key** mechanism.
- We must first identify the "parent" and "child" entities involved in the relationship.
- The **parent** entity refers to the entity that posts a copy of its **primary key** into the relation that represents the child entity, to act as the foreign key.
- We describe how relations are derived for the following structures that may occur in a conceptual data model:
  (1) strong entity types;
  (2) weak entity types;
  (3) one-to-many (1:*) binary relationship types;
  (4) one-to-one (1:1) binary relationship types;
  (5) one-to-one (1:1) recursive relationship types;
  (6) superclass/subclass relationship types;
  (7) many-to-many (*:*) binary relationship types;
  (8) complex relationship types;
  (9) multi-valued attributes.

(1) Strong entity types

For each strong entity in the data model, create a relation that includes all the simple attributes of that entity.

Staff (staffNo, fName, lName, position, sex, DOB)
Primary Key staffNo

(2) Weak entity types

- For each strong entity in the data model, create a relation that includes all the simple attributes of that entity.

- The identification of the primary key of a weak entity cannot be made until after all the relationships with the owner entities have been mapped

Preference (prefType, maxRent)
Primary Key None (at present)

(3) One-to-many (1:*) binary relationship types

- The entity on the "one side" of the relationship is designated as the parent entity.
- The entity on the "many side" is designated as the child entity. Thus, we post a copy of the primary key attribute(s) of the parent entity into the relation representing the child entity, to act as a foreign key.

Post **staffNo** into **Client** to model 1:* ***Registers*** relationship

Staff (staffNo, fName, lName, position, sex, DOB)
Primary Key staffNo

Client (clientNo, fName, lName, telNo, eMail, staffNo)
Primary Key clientNo
Alternate Key eMail
Foreign Key staffNo **references** Staff(staffNo)

(4) One-to-one (1:1) binary relationship types

- Creating relations to represent a 1:1 relationship is slightly more **complex**.

- The cardinality cannot be used to help identify the parent and child entities in a relationship. Instead, the participation constraints: We consider how to create relations to represent the following participation constraints:
  (a) mandatory participation on both sides of 1:1 relationship;
  (b) mandatory participation on one side of 1:1 relationship;
  (c) optional participation on both sides of 1:1 relationship;

(a) *Mandatory* participation on *both* sides of 1:1 relationship

we should **combine** the entities involved into **one relation**. Choose one of the primary keys of the original entities to be the primary key of the new relation.

Example : The ***Client States Preference relationship*** is an example of a 1:1 relationship with mandatory participation on both sides. In this case, we choose to **merge** the **two** relations together to give the following Client relation:

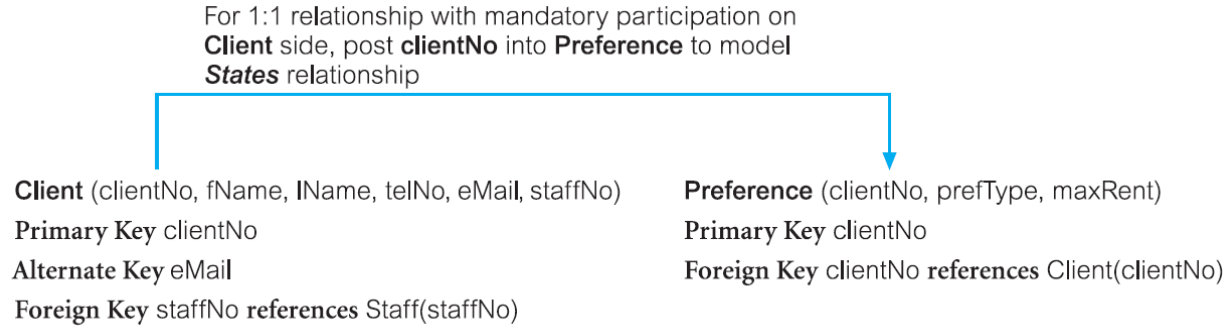
Client (clientNo, fName, lName, telNo, eMail, prefType, maxRent, staffNo)
Primary Key clientNo
Alternate Key eMail
Foreign Key staffNo **references** Staff(staffNo)

Note that it is possible to **merge two entities** into **one relation** only when there are no other direct relationships between these two entities that would prevent this, such as a **1:* relationship.**

(b) *Mandatory* participation on *one* side of a 1:1 relationship

- In this case we are able to identify the parent and child entities for the 1:1 relationship **using the participation constraints**.
- The entity that has **optional** participation in the relationship is designated as the **parent entity**.
- The entity that has **mandatory** participation is designated as the **child** entity.
- The idea is to prevent nulls in case if we combine the two in one relation.

Example: if the 1:1 ***Client States Preference relationship*** had partial participation on the Client side (***not every client specifies preferences***), then the **Client** entity would be designated as the **parent** entity and the **Preference** entity would be designated as the **child** entity.



(c) *Optional* participation on *both* sides of a 1:1 relationship

- In this case the designation of the parent and child entities is **arbitrary** unless we can find out more about the relationship that can help us make.

(6) Superclass/subclass relationship types

- For each superclass/subclass relationship in the conceptual data model, we identify the **superclass** entity as the **parent** entity and the **subclass** entity as the **child** entity.
- There are various options on how to represent such a relationship as one or more relations.
- In this case the most appropriate representation of the superclass/subclass relationship is **determined** by the **constraints** on this relationship.

Table 1. Guidelines for the representation of a superclass/subclass relationship based on the participation and disjoint constraints.

PARTICIPATION CONSTRAINT	DISJOINT CONSTRAINT	RELATIONS REQUIRED
Mandatory	Nondisjoint {And}	Single relation (with one or more discriminators to distinguish the type of each tuple)
Optional	Nondisjoint {And}	Two relations: one relation for superclass and one relation for all subclasses (with one or more discriminators to distinguish the type of each tuple)
Mandatory	Disjoint {Or}	Many relations: one relation for each combined superclass/subclass
Optional	Disjoint {Or}	Many relations: one relation for superclass and one for each subclass

Example, consider the ***Owner superclass/subclass relationship***, there are various ways to represent this relationship as one or more relations.

The options range from: Placing all the attributes into **one relation** with **two discriminators** ***pOwnerFlag*** and ***bOwnerFlag*** indicating whether a tuple belongs to a particular subclass (Option 1), to dividing the attributes into **three relations** (Option 4).

- The relationship that the Owner superclass has with its subclasses is *mandatory* and *disjoint* (each member of the Owner superclass must be a member of one of the subclasses (PrivateOwner or BusinessOwner) but cannot belong to both). We therefore **select Option 3**.
- Create a **separate relation** to represent each subclass, and include a copy of the primary key attribute(s) of the superclass in each.

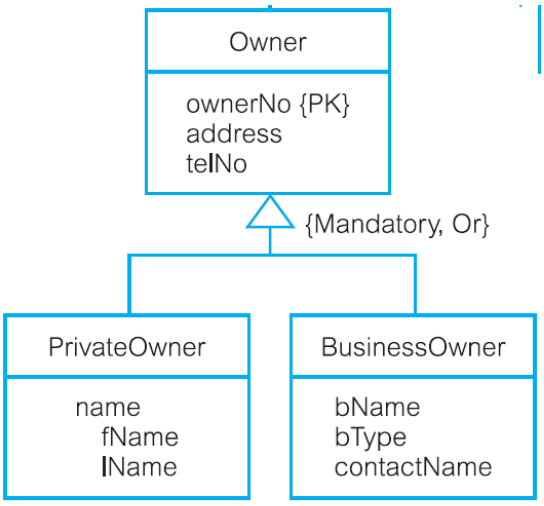


Fig. 2. Super/sub classes relations.

Option 1 – Mandatory, nondisjoint

AllOwner (ownerNo, address, telNo, fName, lName, bName, bType, contactName, pOwnerFlag, bOwnerFlag)
Primary Key ownerNo

Option 2 – Optional, nondisjoint

Owner (ownerNo, address, telNo)
Primary Key ownerNo

OwnerDetails (ownerNo, fName lName, bName, bType, contactName, pOwnerFlag, bOwnerFlag)
Primary Key ownerNo
Foreign Key ownerNo **references** Owner(ownerNo)

Option 3 – Mandatory, disjoint

PrivateOwner (ownerNo, fName, lName, address, telNo)
Primary Key ownerNo

BusinessOwner (ownerNo, bName, bType, contactName, address, telNo)
Primary Key ownerNo

Option 4 – Optional, disjoint

Owner (ownerNo, address, telNo)
Primary Key ownerNo

PrivateOwner (ownerNo, fName, lName)
Primary Key ownerNo
Foreign Key ownerNo **references** Owner(ownerNo)

BusinessOwner (ownerNo, bName, bType, contactName)
Primary Key ownerNo
Foreign Key ownerNo **references** Owner(ownerNo)

Fig. 3. Various representations of the Owner superclass/ subclass relationship based on the participation and disjointness constraints shown in Table1.

(7) Many-to-many (*:*) binary relationship types

For each *:* binary relationship,

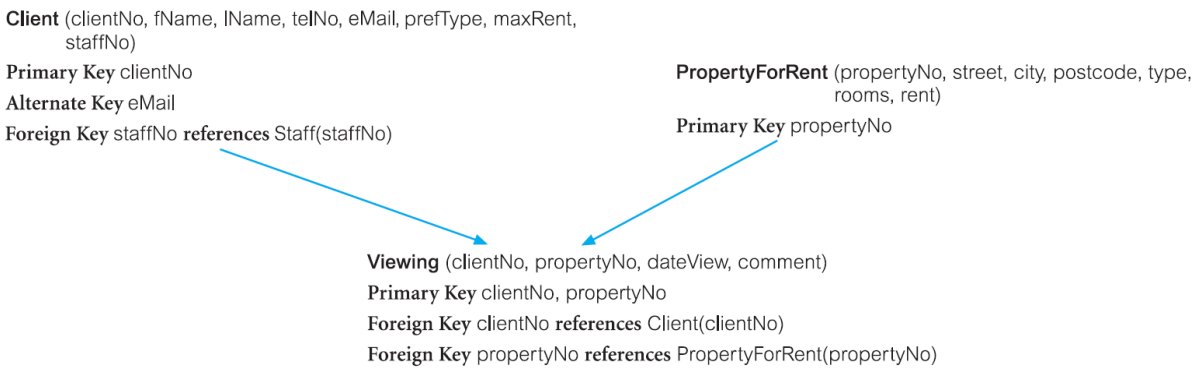
- create a **relation to represent the relationship** and include any attributes that are part of the relationship.

- We post a copy of the primary key attribute(s) of the entities that participate in the relationship into the new relation, to act as foreign keys.

- One or both of these foreign keys will also form the primary key of the new relation

Example, consider the *:* relationship **Client Views PropertyForRent**.

- The Views relationship has two attributes called ***dateView*** and ***comments***.
- We create **relations** for the strong entities **Client and PropertyForRent**.
- We create a relation **Viewing** to represent the relationship Views, to give



(8) Complex relationship types

- Create a **relation to represent the relationship** and include any attributes that are part of the relationship.

- We **post a copy of the primary key** attribute(s) of the entities that participate in the complex relationship **into the new relation**, to act as **foreign keys.**

- Any **foreign keys** that represent a "**many**" relationship (for example, 1..*, 0..*) generally will also form the primary key of this new relation, possibly in combination with some of the attributes of the relationship.

Example: the ternary **Registers relationship** in the Branch user views **represents the association between the member of staff who registers a new client at a branch**, as shown in Figure . To represent this, we **create relations** for the strong entities **Branch**, **Staff**, and **Client**, and we **create a relation Registration** to represent the **relationship Registers**, to give:

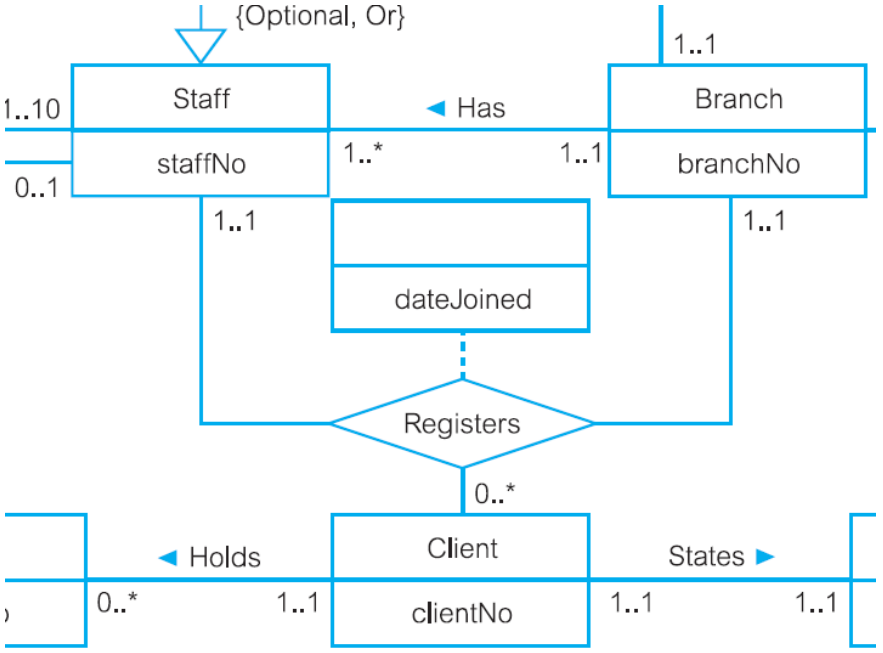


Fig. 4. Register as a complex relationship example.

Staff (staffNo, fName, lName, position, sex, DOB, supervisorStaffNo)
Primary Key staffNo
Foreign Key supervisorStaffNo **references** Staff(staffNo)

Branch (branchNo, street, city, postcode)
Primary Key branchNo

Client (clientNo, fName, lName, telNo, eMail, prefType, maxRent)
Primary Key clientNo
Alternate Key eMail

Registration (clientNo, branchNo, staffNo, dateJoined)
Primary Key clientNo, branchNo
Foreign Key branchNo **references** Branch(branchNo)
Foreign Key clientNo **references** Client(clientNo)
Foreign Key staffNo **references** Staff(staffNo)

(9) Multi-valued attributes

For each multi-valued attribute in an entity:

- **Create a new relation** to represent the multi-valued attribute.
- Include **the primary key of the entity** in the **new relation** to act as **a foreign key**.

Example:

- In the Branch user views to represent the situation where a **single branch has up to three telephone numbers.**
- The ***telNo*** attribute of the Branch entity has been defined as being a multi-valued attribute.

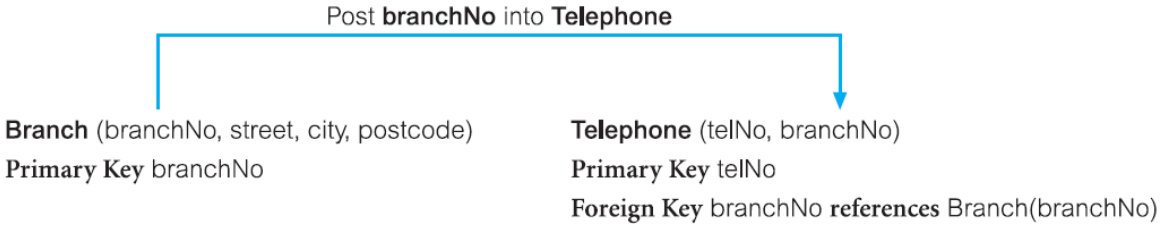


Table 2 Summary of how to map entities and relationships to relations.

ENTITY/RELATIONSHIP	MAPPING
Strong entity	Create relation that includes all simple attributes.
Weak entity	Create relation that includes all simple attributes (primary key still has to be identified after the relationship with each owner entity has been mapped).
1:* binary relationship	Post primary key of entity on the "one" side to act as foreign key in relation representing entity on the "many" side. Any attributes of relationship are also posted to the "many" side.
1:1 binary relationship: (a) Mandatory participation on both sides (b) Mandatory participation on one side (c) Optional participation on both sides	 Combine entities into one relation. Post primary key of entity on the "optional" side to act as foreign key in relation representing entity on the "mandatory" side. Arbitrary without further information.
Superclass/subclass relationship	See Table 17.1.
: binary relationship, complex relationship	Create a relation to represent the relationship and include any attributes of the relationship. Post a copy of the primary keys from each of the owner entities into the new relation to act as foreign keys.
Multi-valued attribute	Create a relation to represent the multi-valued attribute and post a copy of the primary key of the owner entity into the new relation to act as a foreign key.

Staff (staffNo, fName, lName, position, sex, DOB, supervisorStaffNo) Primary Key staffNo Foreign Key supervisorStaffNo references Staff(staffNo)	**PrivateOwner** (ownerNo, fName, lName, address, telNo) Primary Key ownerNo
BusinessOwner (ownerNo, bName, bType, contactName, address, telNo) Primary Key ownerNo Alternate Key bName Alternate Key telNo	**Client** (clientNo, fName, lName, telNo, eMail, prefType, maxRent, staffNo) Primary Key clientNo Alternate Key eMail Foreign Key staffNo references Staff(staffNo)
PropertyForRent (propertyNo, street, city, postcode, type, rooms, rent, ownerNo, staffNo) Primary Key propertyNo Foreign Key ownerNo references PrivateOwner(ownerNo) and BusinessOwner(ownerNo) Foreign Key staffNo references Staff(staffNo)	**Viewing** (clientNo, propertyNo, dateView, comment) Primary Key clientNo, propertyNo Foreign Key clientNo references Client(clientNo) Foreign Key propertyNo references PropertyForRent(propertyNo)
Lease (leaseNo, paymentMethod, depositPaid, rentStart, rentFinish, clientNo, propertyNo) Primary Key leaseNo Alternate Key propertyNo, rentStart Alternate Key clientNo, rentStart Foreign Key clientNo references Client(clientNo) Foreign Key propertyNo references PropertyForRent(propertyNo) Derived deposit (PropertyForRent.rent*2) Derived duration (rentFinish – rentStart)	

Fig. 5. Step 1 Output.

Step 2: Check integrity constraints

Objective To check whether integrity constraints are represented in the logical data model.

Integrity constraints are the constraints that we wish to impose in order to protect the database from becoming incomplete, inaccurate, or inconsistent.
We consider the following types of integrity constraint:

- required data;
- attribute domain constraints;
- multiplicity;
- entity integrity;
- referential integrity;

I. **Required data:**
Some attributes must always contain a valid value; in other words, they are not allowed to hold nulls.

II. **Attribute domain constraints:** Every attribute has a domain, that is, a set of values that are legal. For example, the sex of a member of staff is either "M" or "F," so the domain of the sex attribute is a single character string consisting of "M" or "F." These constraints should have been identified when we chose the attribute domains for the data model (Step 4 in Conceptual database design).

III. **Multiplicity** represents the constraints that are placed on relationships between data in the database.

IV. **Entity integrity**: The primary key of an entity cannot hold nulls. For example, each tuple of the Staff relation must have a value for the primary key attribute, staffNo. These constraints should have been considered when we identified the primary keys for each entity type (Step 5 in conceptual database design).

V. **Referential integrity**

- **A foreign key** links each tuple in the child relation to the tuple in the parent relation containing the matching primary key value.
- **Referential integrity** means that if the foreign key contains a value, that value must refer to an existing tuple in the parent relation.

- There are **two issues** regarding foreign keys that must be addressed:
  1. The first considers whether nulls are allowed for the foreign key.
     For example, can we store the details of a property for rent without having a member of staff specified to manage it—that is, can we specify a null staffNo? The issue is not whether the staff number exists, but whether a staff number must be specified.
     In general, if the participation of the child relation in the relationship is:
     • mandatory, then nulls are not allowed;
     • optional, then nulls are allowed.
  2. The second issue we must address is how to ensure referential integrity. we specify **existence constraints** that define conditions under which a primary key or foreign key may be inserted, updated, or deleted.
     For the 1:* ***Staff Manages PropertyForRent*** relationship, consider the following cases.

Case 1: Insert tuple into child relation (*PropertyForRent*): To ensure referential integrity, check that the foreign key attribute, ***staffNo***, of the new PropertyForRent tuple is set to **null** or **to a value of an existing Staff tuple**.

Case 2: Delete tuple from child relation (*PropertyForRent*): If a tuple of a child relation is deleted referential integrity is **unaffected**.

Case 3: Update foreign key of child tuple (*PropertyForRent*): This case is similar to Case 1. To ensure referential integrity, check whether the ***staffNo*** of the updated ***PropertyForRent*** tuple is set to **null** or **to a value of an existing Staff tuple**.

Case 4: Insert tuple into parent relation (*Staff*): Inserting a tuple into the parent relation (Staff) **does not affect referential integrity**; it simply becomes a parent without any children: in other words, a member of staff without properties to manage.

Case 5: Delete tuple from parent relation (*Staff*): If a tuple of a parent relation is deleted, **referential integrity is lost**. There are several strategies we can consider:

NO ACTION—Prevent a deletion from the parent relation if there are any referenced child tuples. In our example, "You cannot delete a member of staff if he or she currently manages any properties."

CASCADE—When the parent tuple is **deleted**, **automatically delete** any referenced child tuples. If any deleted child tuple acts as the parent in another relationship, then the delete operation should be applied to the tuples in this child relation, and so on in a cascading manner.

SET NULL—When a parent tuple is deleted, the foreign key values in all corresponding child tuples are **automatically set to null**.

SET DEFAULT—When a parent tuple is deleted, the foreign key values in all corresponding child tuples should **automatically be set to their default values**.

NO CHECK—When a parent tuple is deleted, do nothing to ensure that referential integrity is maintained.

Case 6: Update primary key of parent tuple (Staff): If the primary key value of a parent relation tuple is updated, **referential integrity is lost** if there exists a child ld primary key value; that is, if the updated member of staff currently manages one or more properties. To ensure referential integrity, the strategies described earlier can be used. In the case of CASCADE, the updates to the primary key of the parent tuple are reflected in any referencing child tuples, and if a referencing child tuple is itself a

primary key of a parent tuple, this update will also cascade to its referencing child tuples, and so on in a cascading manner. It is normal for updates to be specified as CASCADE.

Staff (staffNo, fName, lName, position, sex, DOB, supervisorStaffNo)
Primary Key staffNo
Foreign Key supervisorStaffNo **references** Staff(staffNo) ON UPDATE CASCADE ON DELETE SET NULL

Client (clientNo, fName, lName, telNo, eMail, prefType, maxRent, staffNo)
Primary Key clientNo
Alternate Key eMail
Foreign Key staffNo **references** Staff(staffNo) ON UPDATE CASCADE ON DELETE NO ACTION

PropertyForRent (propertyNo, street, city, postcode, type, rooms, rent, ownerNo, staffNo)
Primary Key propertyNo
Foreign Key ownerNo **references** PrivateOwner(ownerNo) and BusinessOwner(ownerNo)
ON UPDATE CASCADE ON DELETE NO ACTION
Foreign Key staffNo **references** Staff(staffNo) ON UPDATE CASCADE ON DELETE SET NULL

Viewing (clientNo, propertyNo, dateView, comment)
Primary Key clientNo, propertyNo
Foreign Key clientNo **references** Client(clientNo) ON UPDATE CASCADE ON DELETE NO ACTION
Foreign Key propertyNo **references** PropertyForRent(propertyNo)
ON UPDATE CASCADE ON DELETE CASCADE

Lease (leaseNo, paymentMethod, depositPaid, rentStart, rentFinish, clientNo, propertyNo)
Primary Key leaseNo
Alternate Key propertyNo, rentStart
Alternate Key clientNo, rentStart
Foreign Key clientNo **references** Client(clientNo) ON UPDATE CASCADE ON DELETE NO ACTION
Foreign Key propertyNo **references** PropertyForRent(propertyNo)
ON UPDATE CASCADE ON DELETE NO ACTION

Fig6. Referential integrity constraints for the relations in the StaffClient user views of *DreamHome*.

Step 3: Validate relations against user transactions.

Objective To ensure that the relations in the logical data model support the required transactions.

The objective of this step is to **validate** the logical data model to ensure that the model supports the required **transactions**. This type of check was carried out in Step 8 to ensure that the conceptual data model supported the required transactions.

Step 4: Review logical data model with user

Objective To review the logical data model with the users to ensure that they consider the model to be a true representation of the data requirements of the enterprise.

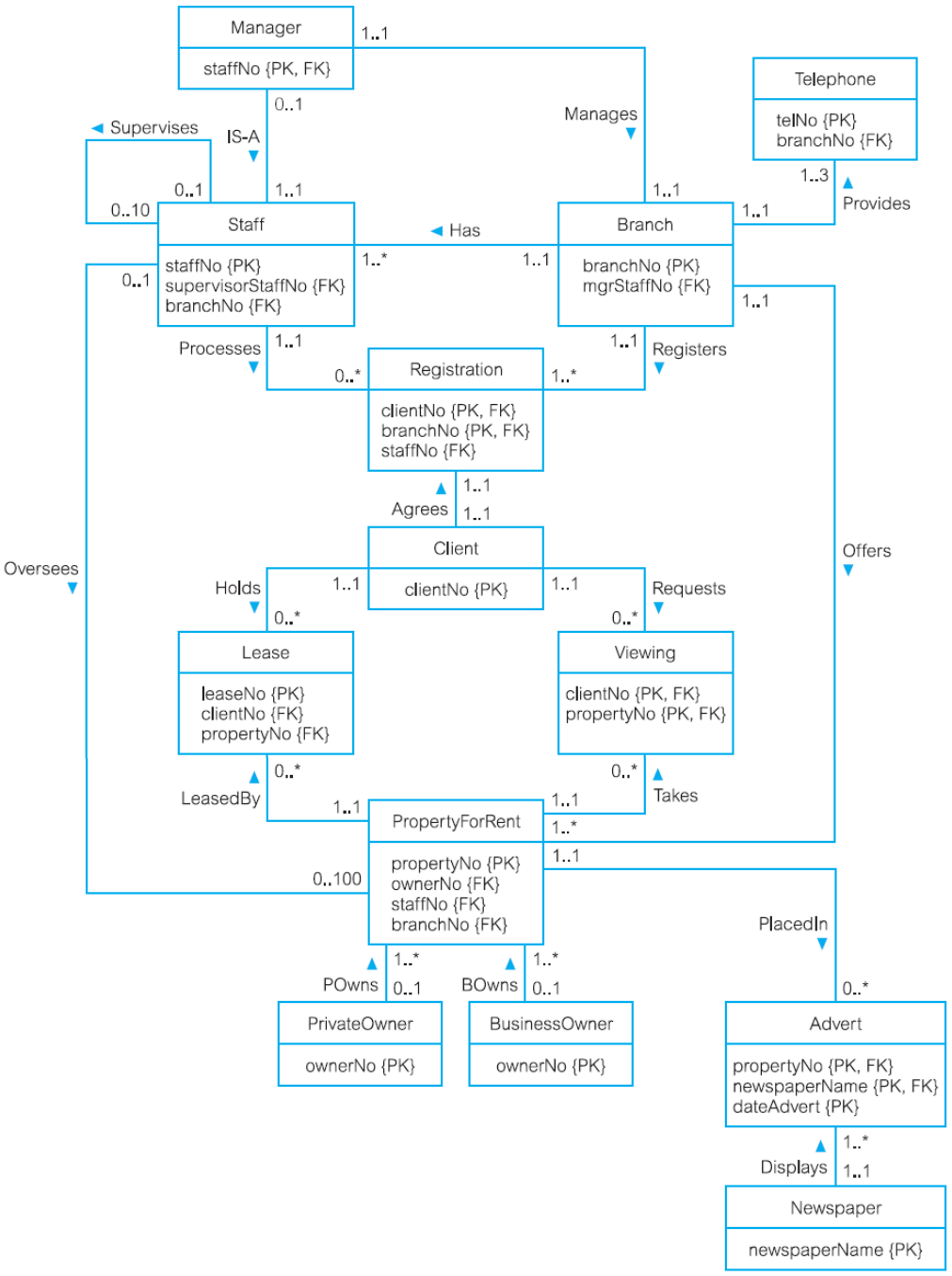


Fig. 7. Final Logical Database Design.

Checkpoints

I. The *University Accommodation Office* case study

Create and validate a logical data model from the conceptual data model for the *University Accommodation Office* case study created in previous Exercise.

II. The *EasyDrive School of Motoring* case study

Create and validate a logical data model from the conceptual data model for the *EasyDrive School of Motoring* case study created in previous Exercise.

III. The *Wellmeadows Hospital* case study

Create and validate the local logical data models for each of the local conceptual data models of the *Wellmeadows Hospital* case study created in previous Exercise.

IV. The *Parking Lot* case study

Describe the relational schema of the Parking Lot EER model, and redraw the Figure.

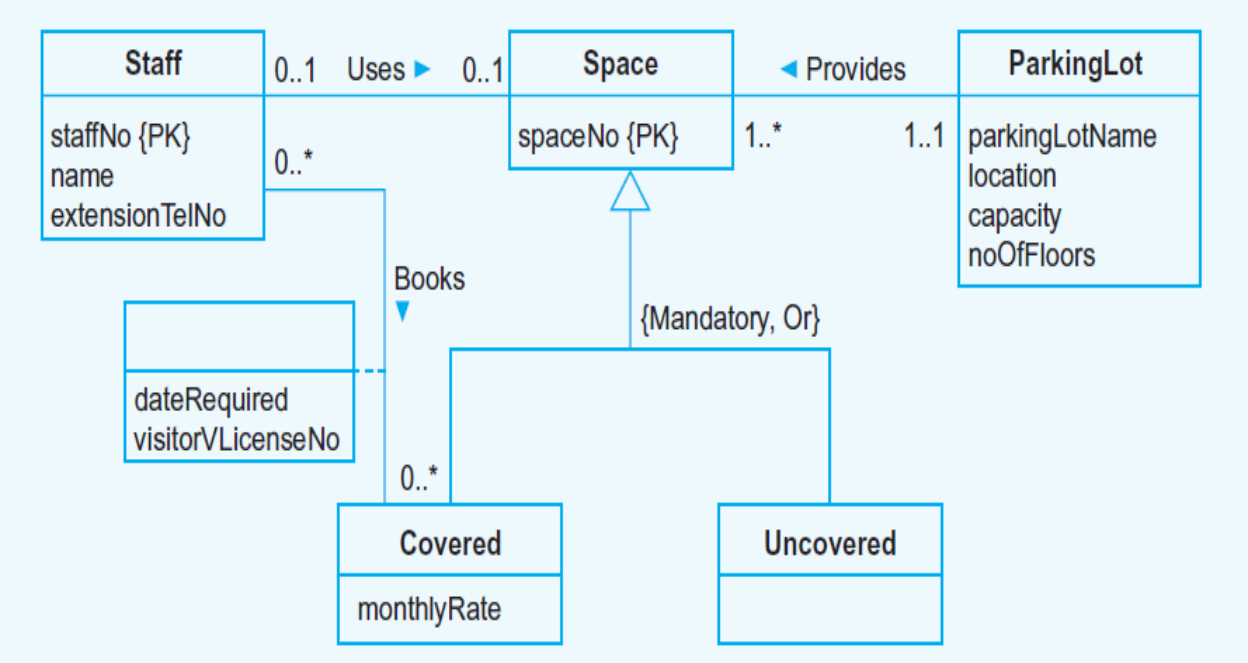


Fig. 8. Parking lot ER.

V. The *Library* case stud

Describe the relational schema mapped from the Library EER model, and redraw the Figure.

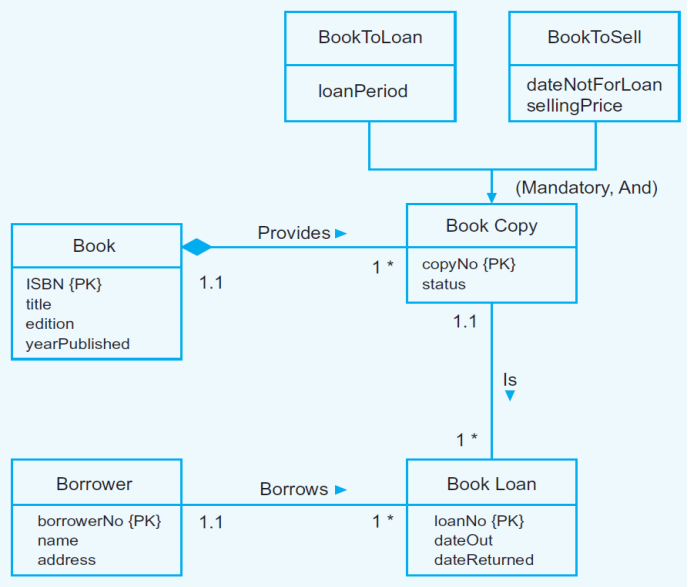


Fig. 9. Library ER.

VI. The ER diagram in Fig. 9 shows only entities and primary key attributes. The absence of recognizable named entities or relationships is to emphasize the rule-based nature of the mapping rules described previously in Step 1 of logical database design.

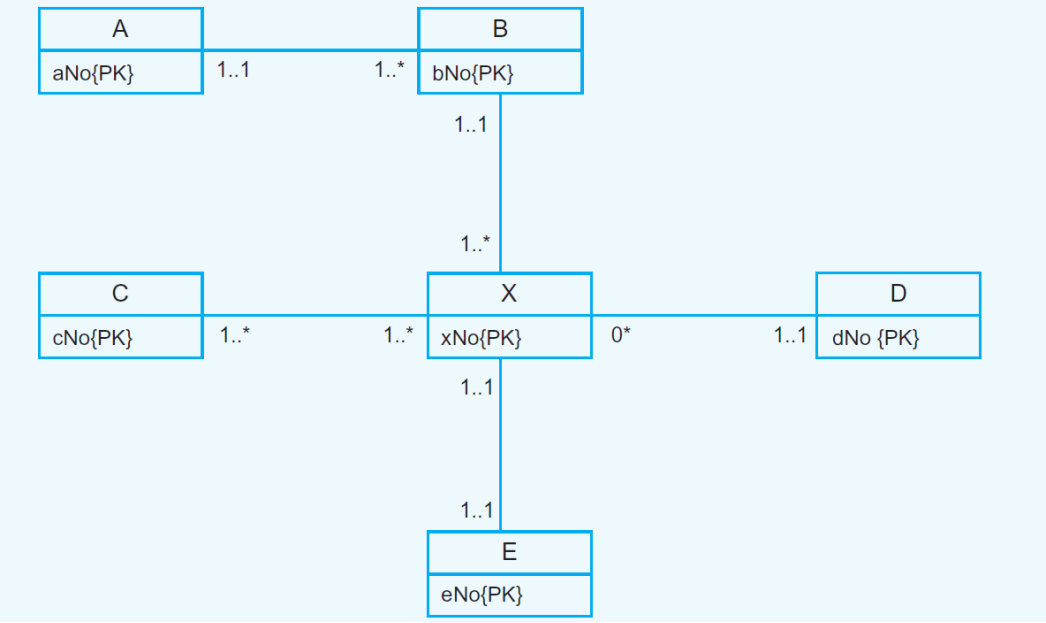


Fig. 9. An example ER model.

Answer the following questions with reference to how the ER model in Fig. 9 maps to relational tables.

(a) How many relations will represent the ER model?

(b) How many foreign keys are mapped to the relation representing X?

(c) Which relation(s) will have no foreign key?
(d) Using only the letter identifier for each entity, provide appropriate names for the relation mapped from the ER model.
(e) If the cardinality for each relationship is changed to *one-to-one* with total participation for all entities, how many relations would be derived from this version of the ER model?

Mohammed Dheyaa