2. Database Design

Slides adapted from John Dowell & Pearson Ed.

Work to be done before designing a database

- Requirements gathering: understanding…
  - what data will be used,
  - who will use the system,
  - what features the system will need to implement.
- In your coursework, this has largely been done in the write-up itself.

Database design process

- Two steps for designing database:
  - Conceptual design: creating a model of the database without considering any physical aspects
    - Using ER modeling / diagrams.
  - Logical design: defining structure of data in more detail having decided on a particular data model (relational, object-oriented, graph-based, etc.)
    - Using normalization and translation procedure for relational DBs.
- Conceptual design = “blue sky thinking” where only limitations are those imposed by the type of data being collected.
- Logical design: Additional constraints imposed by structure of the database chosen.

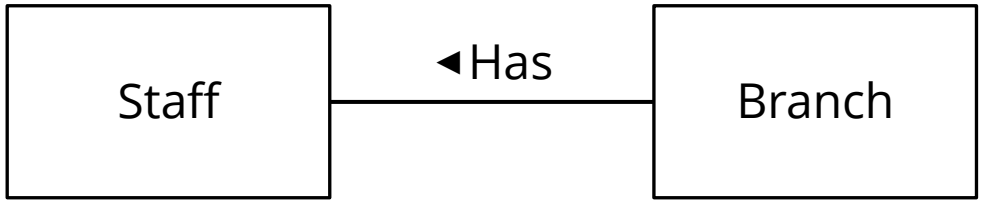
Conceptual design

Conceptual data model

- Create a picture to lay out all parts of the database and describe the properties of the data and how different data are related to each other.
- Can be structured in multiple ways
  - Focus on **entity-relationship** (ER) model: everything is either
    - an entity (a thing; not necessarily physical),
    - a relationship (a connection between two or more entities), or
    - an attribute (a property describing one of the above).

ER modeling with UML

- Common system for creating ER diagrams (ERDs) is the **Unified Modeling Language** (UML).
- Entities: in rectangle boxes with UpperCamelCase names.
- Relationships: arrows between entities with UpperCamelCase names followed by an arrow indicating reading direction of relationship.

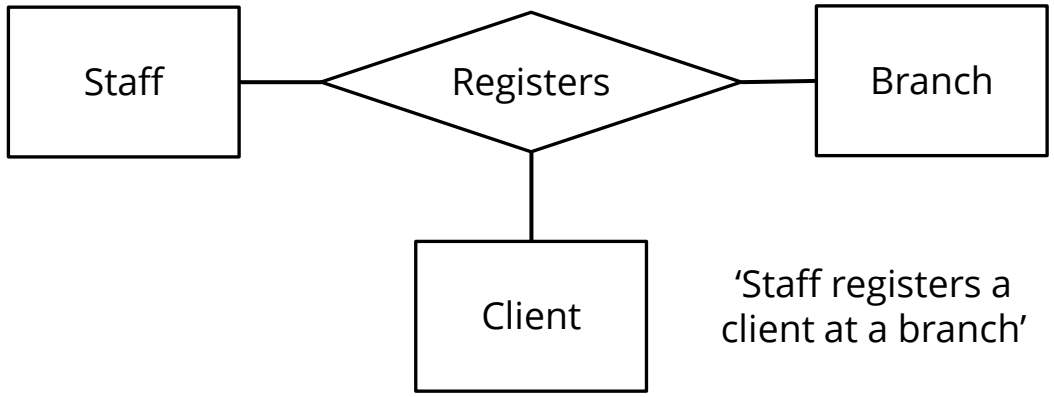


Relationships: Degree

- Degree of a relationship refers to how many entities are involved in that relationship.
- Simplest relationship are degree 2 (binary).
- Degree 3 = ternary; degree 4 = quaternary; etc.

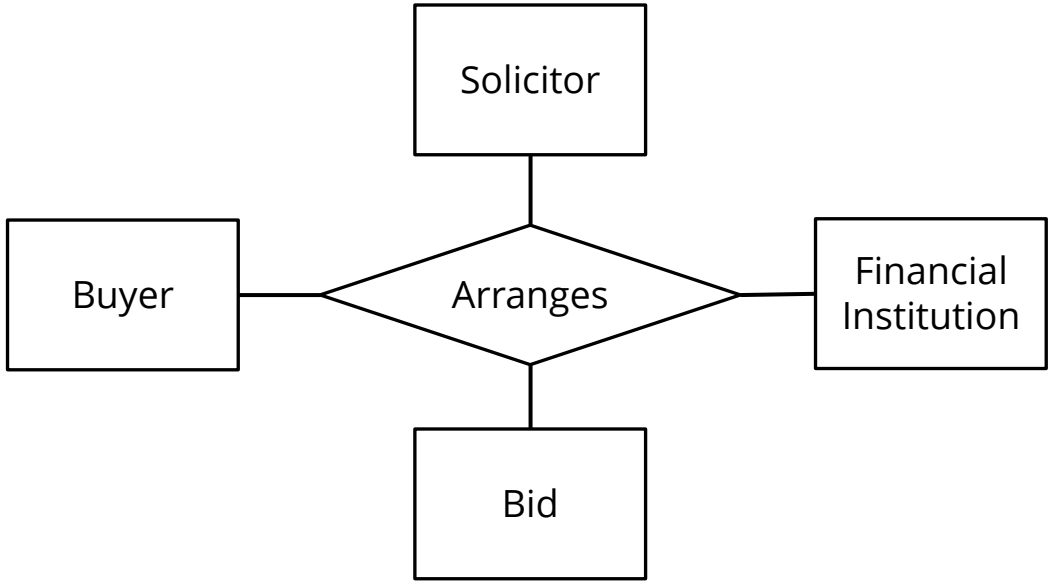
Modeling complex relationships

- For degree 2+, surround the relationship name with a diamond and connect it to all relevant entities.
- Can no longer use arrows.
- Relies on intuitive spatial arrangement & longer supplemental descriptions.



Modeling complex relationships

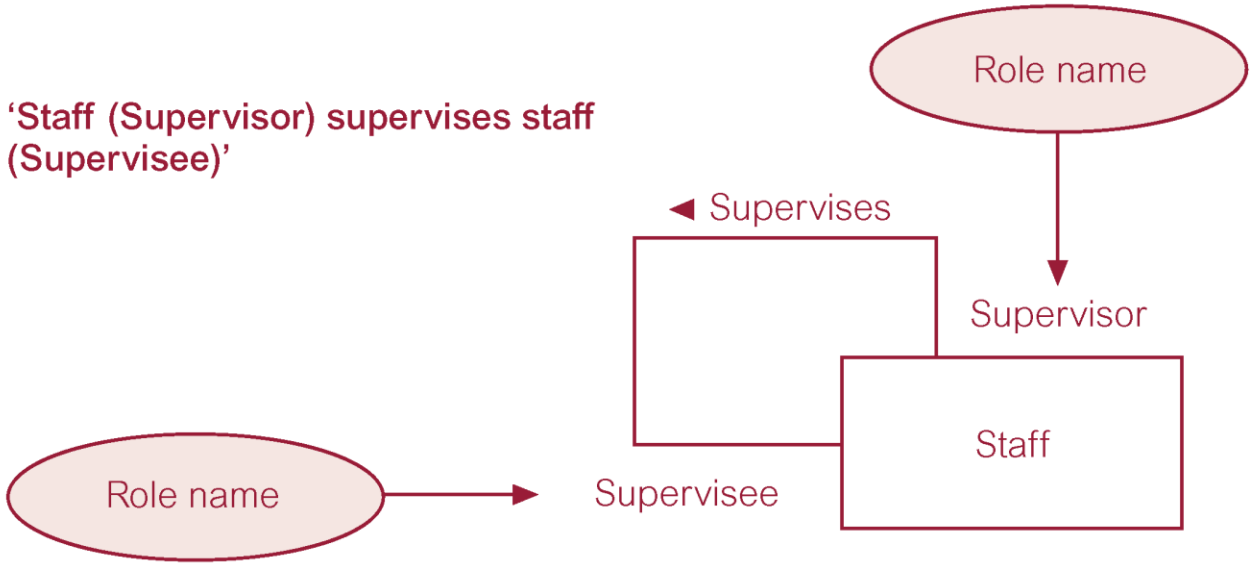
- For degree 2+, surround the relationship name with a diamond and connect it to all relevant entities.
- Can no longer use arrows.
- Relies on intuitive spatial arrangement & longer supplemental descriptions.



'A solicitor arranges a bid on behalf of a buyer supported by a financial institution'

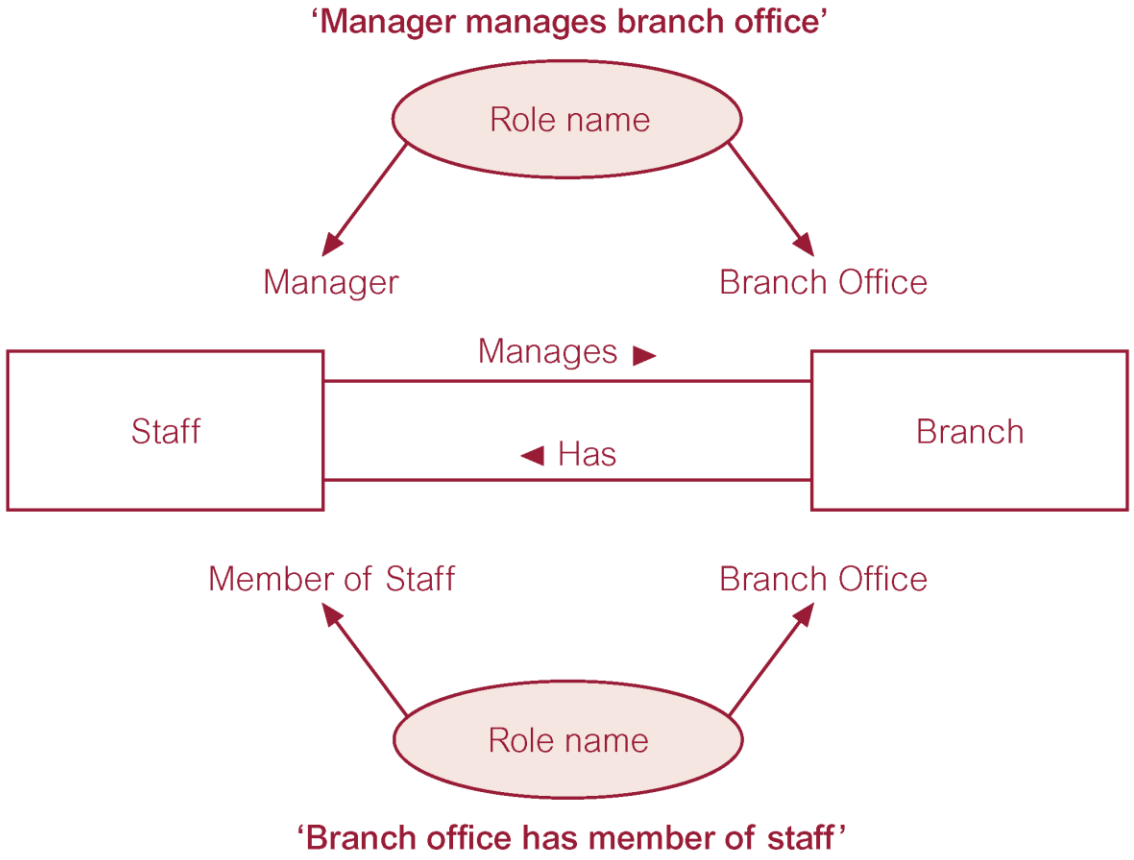
Relationships: Role names

- Can add labels (role names) on the ends of the arrow to clarify the relationship more.
  - Useful for *self-loops* and *multiple relationships* existing between the same set of entities.



Relationships: Role names

- Can add labels (role names) on the ends of the arrow to clarify the relationship more.
  - Useful for *self-loops* and *multiple relationships* existing between the same set of entities.

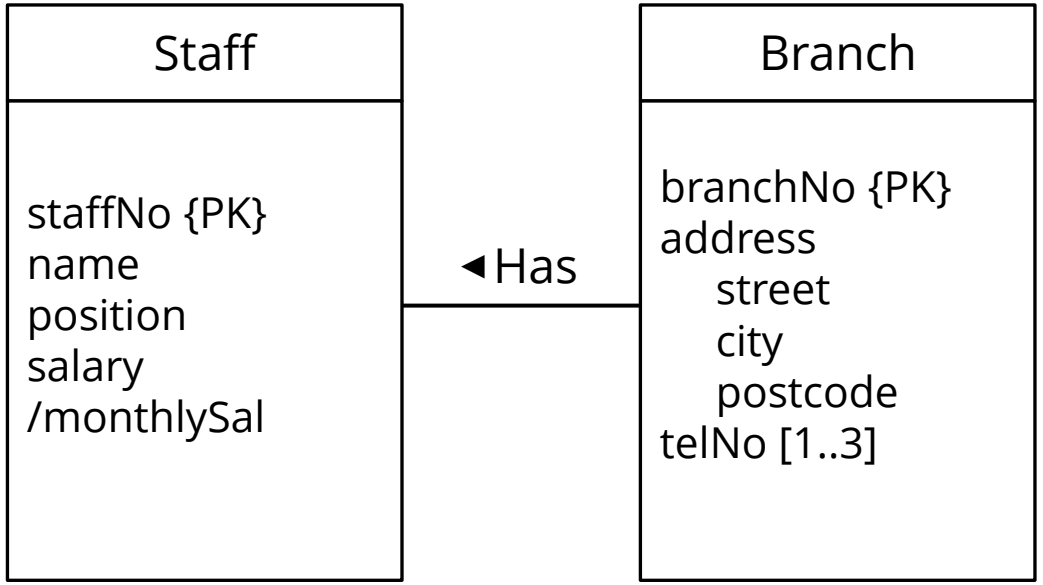


Attributes: Entity attributes

- Properties of entities in your model.
- Types:
  - **Candidate key:** one attribute or a minimal set of attributes that uniquely identifies a specific instance of an entity.
    - Composite key: when it's a set of attributes that work together to ID an entity.
  - **Primary key:** a key that has been selected from the above options to be THE way of identifying instances of an entity.
  - **Composite attributes:** attributes themselves made up of multiple pieces of data (e.g. address) – contrast with *simple attributes.*
  - **Multi-valued attributes:** attributes that can contain a list of data all of the same type (e.g. list of phone numbers) – contrast with *single-valued attrs.*
  - **Derived attributes:** attributes that can be entirely calculated from another attribute.

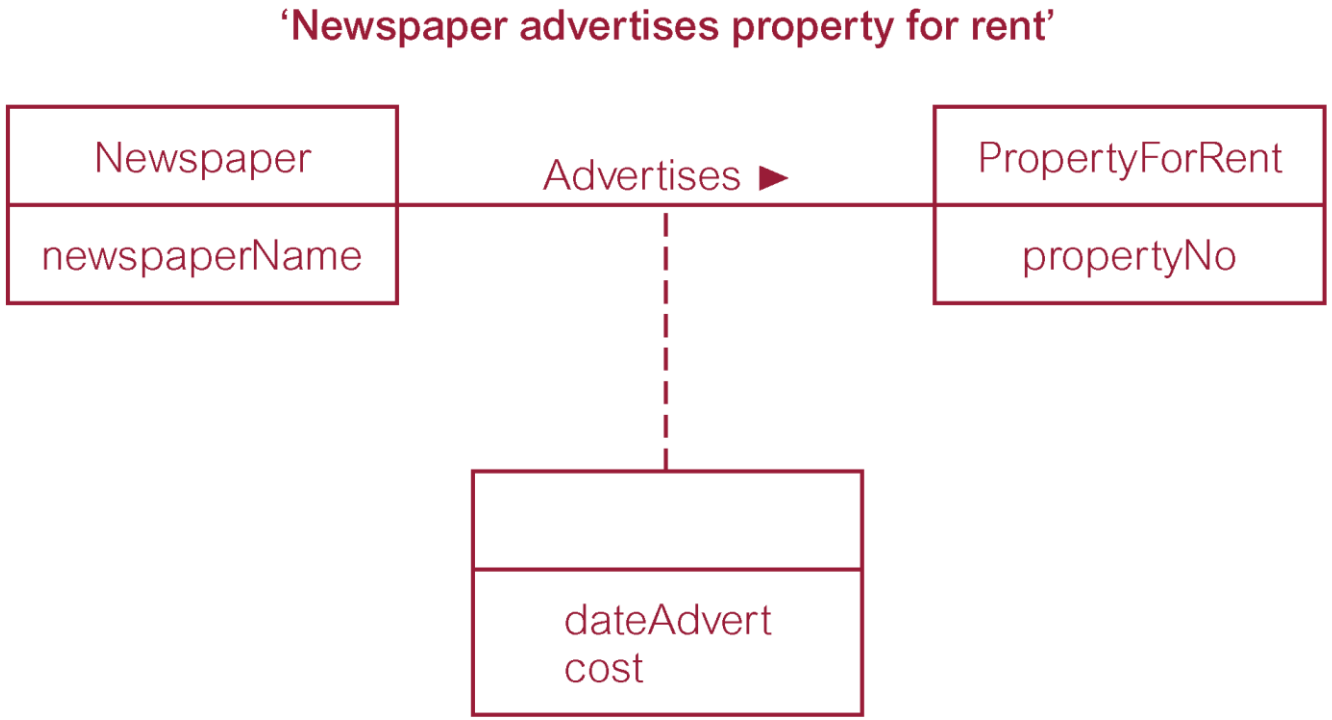
Diagramming entity attributes

- Entity attributes: list under the entity name with lowerCamelCase names.
  - Primary keys: the only attr, the first attr, or marked with {PK}.
    - Composite primary key: use {PPK} for the parts.
  - Candidate key: marked with {CK}.
  - Composite attributes: list parts indented underneath.
  - Multi-value attributes: specify how many values allowed e.g. [0..5].
  - Derived attributes: prefix with '/'.



Attributes: Relationship attributes

- Relationship attributes: boxed list connected to relationship name with a dashed line.

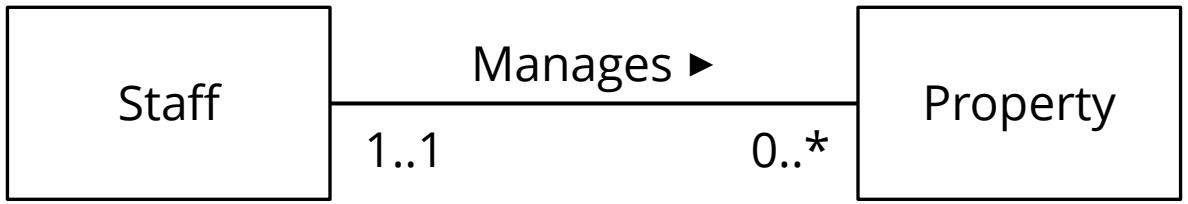


Structural constraints

- Properties of entities and relationships that must be true at all times.
- Main relationship constraint that can be visualized in an ERD is **multiplicity** (how many occurrences of other entities are allowed to be related to one occurrence of this entity).
- Multiplicity has two parts:
  - Cardinality: upper bound.
  - Participation: lower bound (0 = optional participation; 1+ = mandatory participation).

Diagramming multiplicity

- Multiplicity: ranges at the ends of a relationship line saying how many [near] entities a specific [far] entity is allowed to have.
  - '*' means infinite (no bound).
  - '1..1' can be written as just '1'.
  - '0..*' can be written as just '*' (any amount).

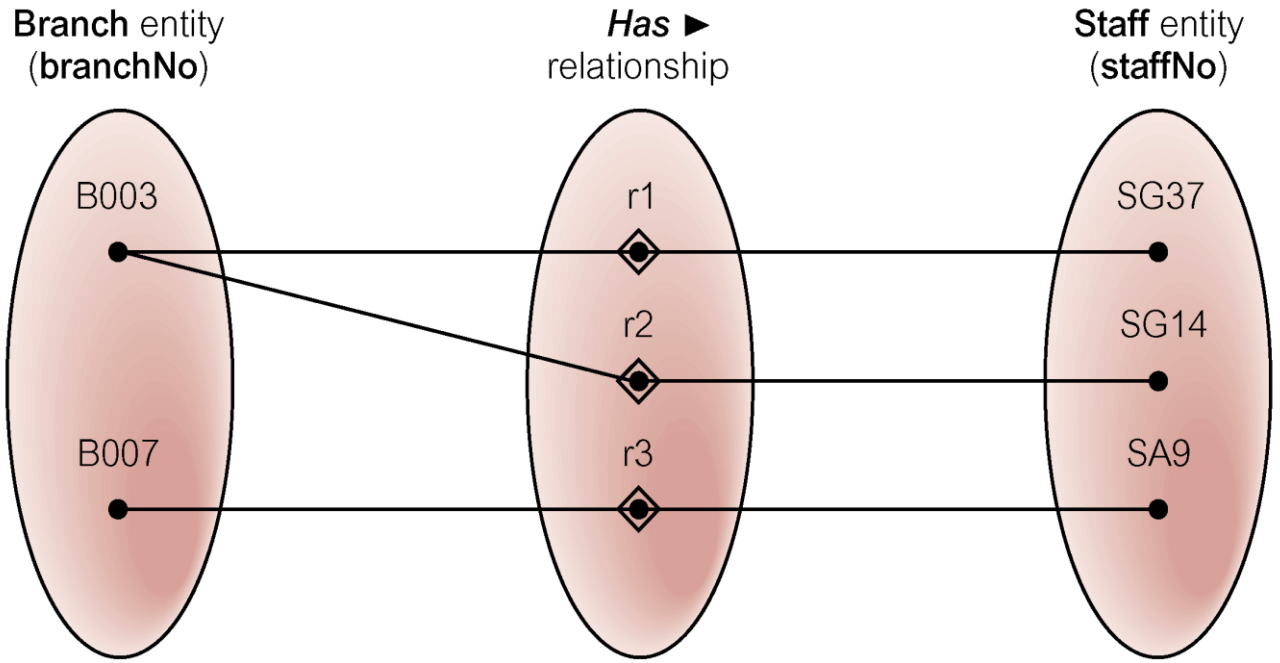


Types of binary relationships

- For relationships between two entities, can group possible multiplicity into three types:
  - One-to-one (1:1) – for both entities, one entity of each type can be associated with (at most) one entity of the other type.
  - One-to-many (1:*) – one [subject] entity can be associated with many [object] entities, but each [object] entity has (at most) one [subject] entity.
  - Many-to-many (*:*) – entities of both types can be associated with multiple entities of the other type.
- Note: doesn't make any claims about *participation*, just the *cardinalities* of the relationship from the point of view of each entity involved in the binary relationship.

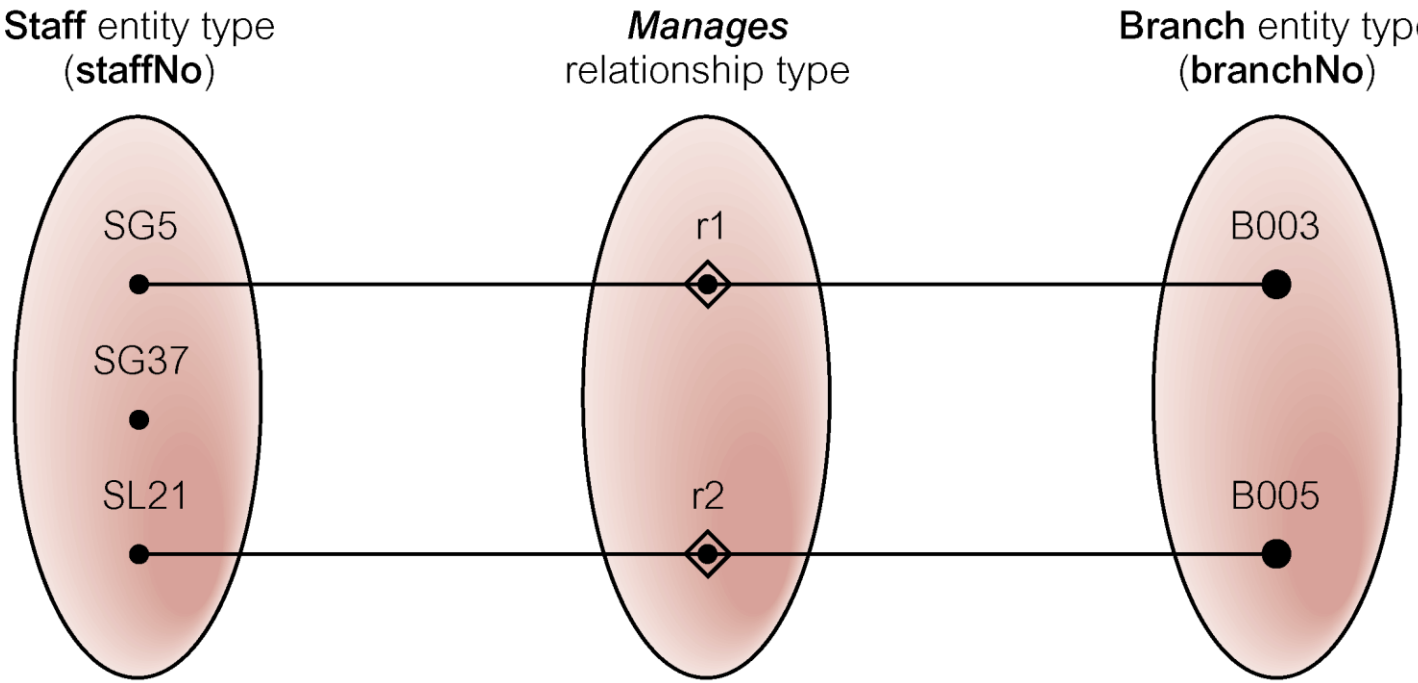
Using semantic nets to visualize relationships

- NOT the same as UML / ER diagram.
  - ER diagram shows generic entities + relationships; semantic nets use sample data.
- Useful for visualizing multiplicity.

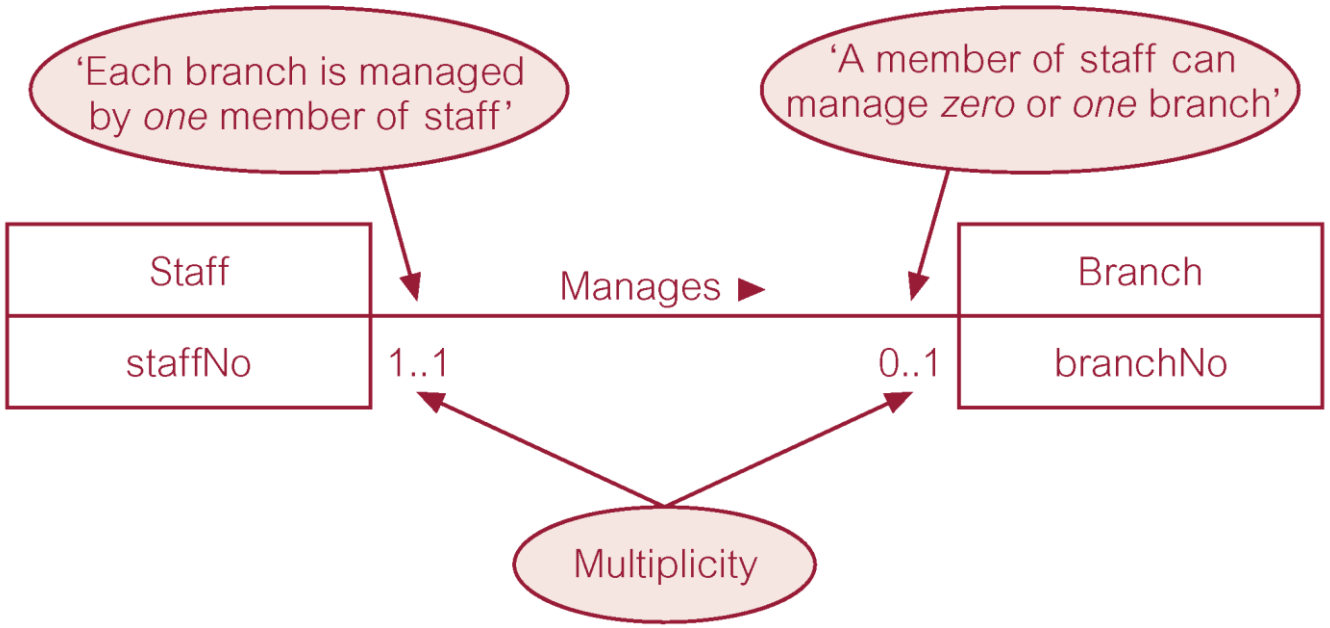


Visualizing a 1:* relationship

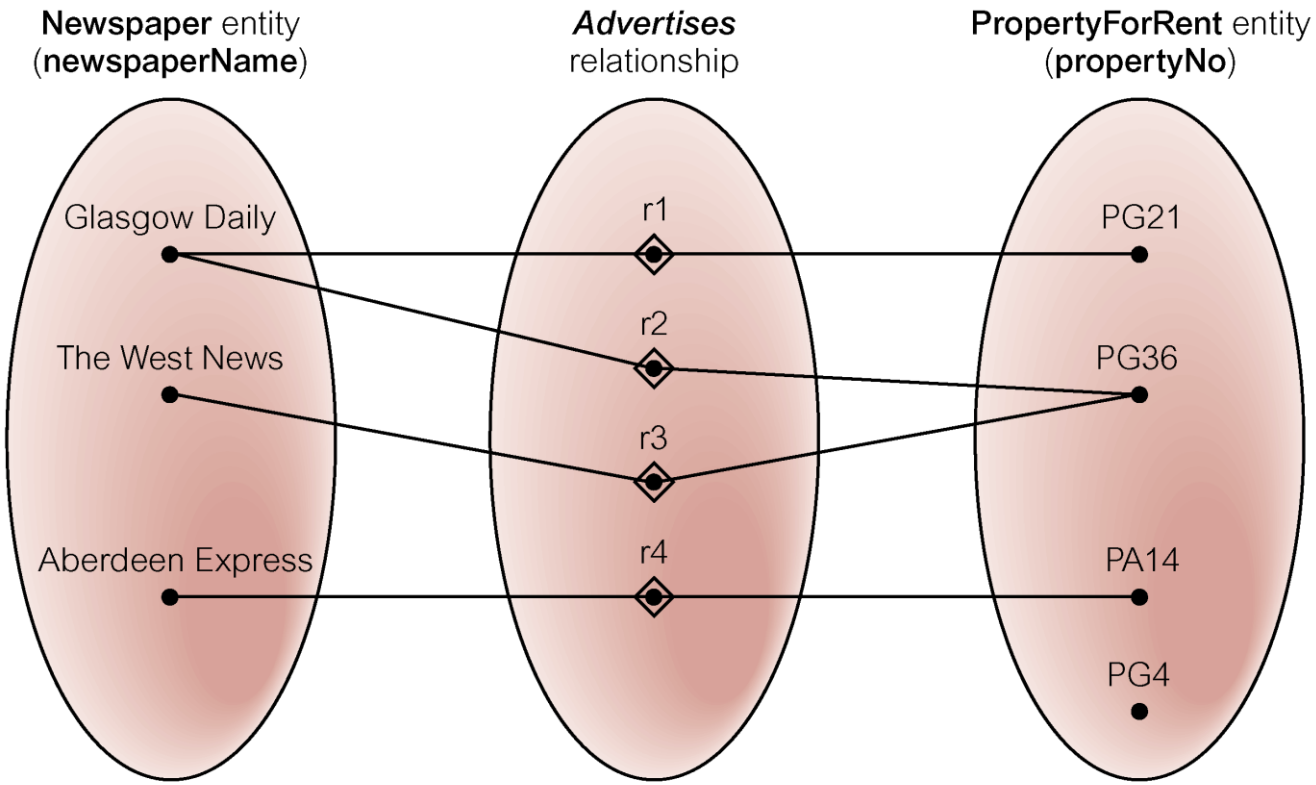
Semantic net: 1:1 relationship



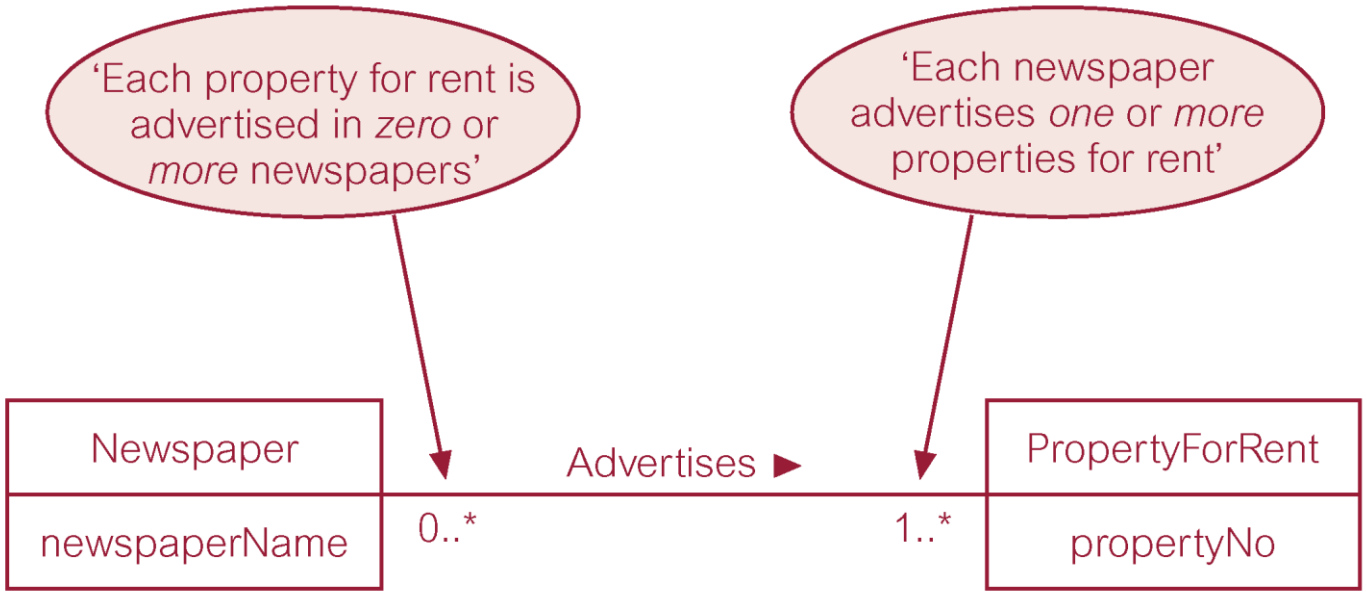
1:1 relationship as an ER diagram



Semantic net: *:* relationship

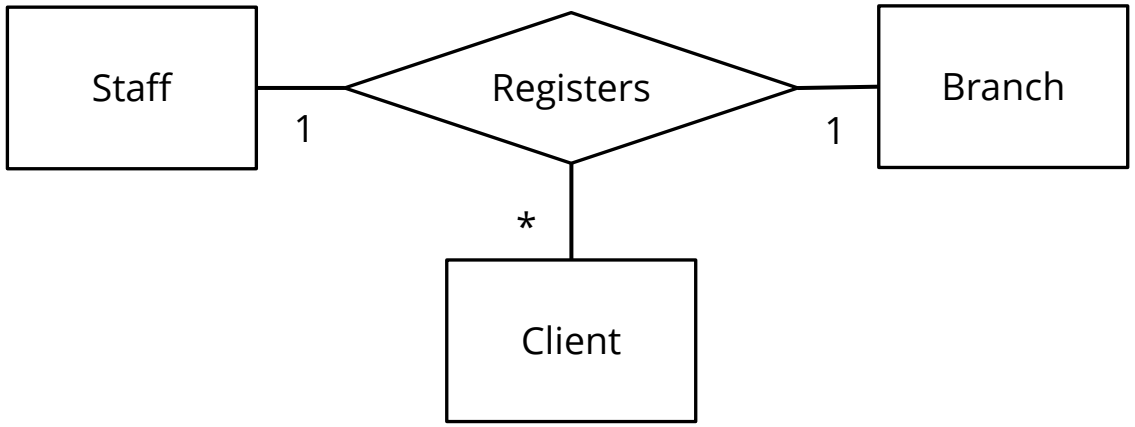


: relationship as an ER diagram



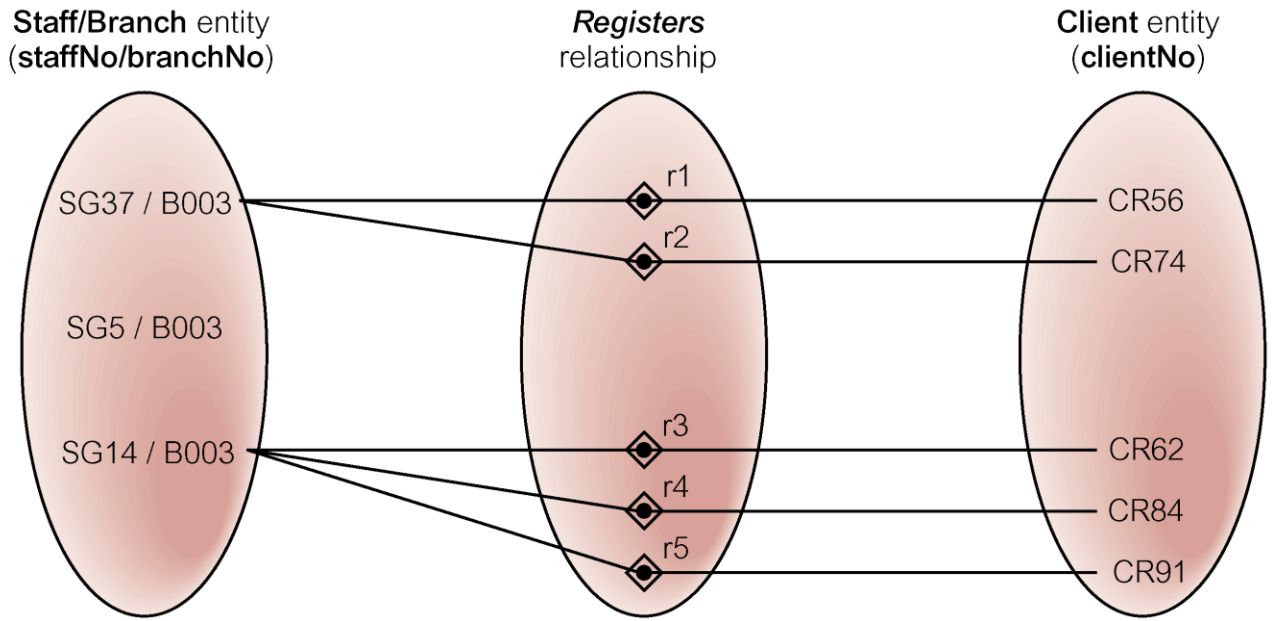
Higher order relationships

- Label each endpoint of a relationship with how many [near] entities a particular value of [far] entities is allowed to have.
  - Generalization of the simple case.
- How to read:
  - “A member of staff can register any number of clients at a single branch”.
  - “A staff member can only register a client at 1 branch”.



Semantic nets can be used for higher-order relationships

- Right side: one entity (Client).
- Left side: fixed *combinations* of possible values for all other entities (Staff/Branch).
- Middle: individual connections.

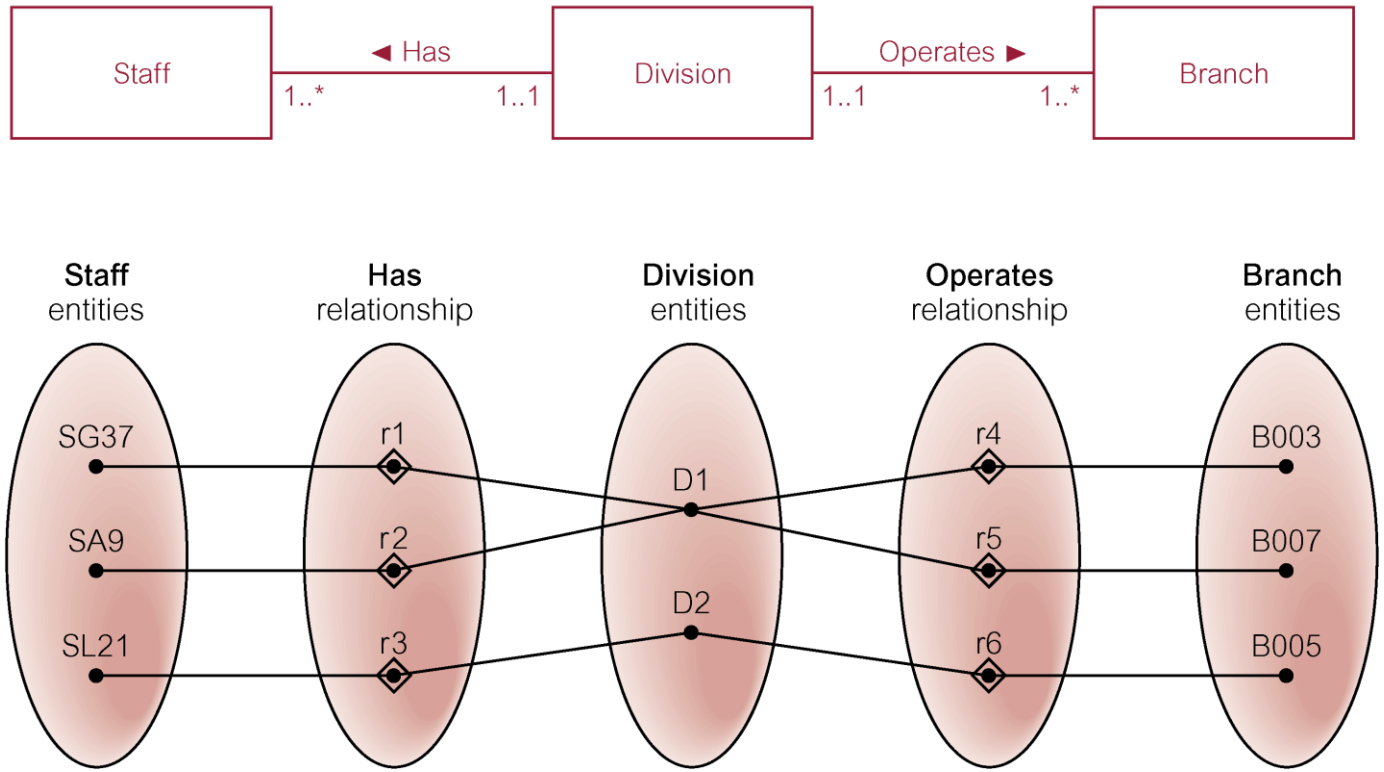


Traps in ERDs

- Connection trap: when an ER diagram is laid out in a way that makes understanding its relationships difficult.
- Examples:
  - Fan trap: When two 1:* relationships fan out from one entity.
    - Obscures potential relationships between the two.
  - Chasm trap: When a series of optional relationships allows what should be a mandatory relationship to also become optional.

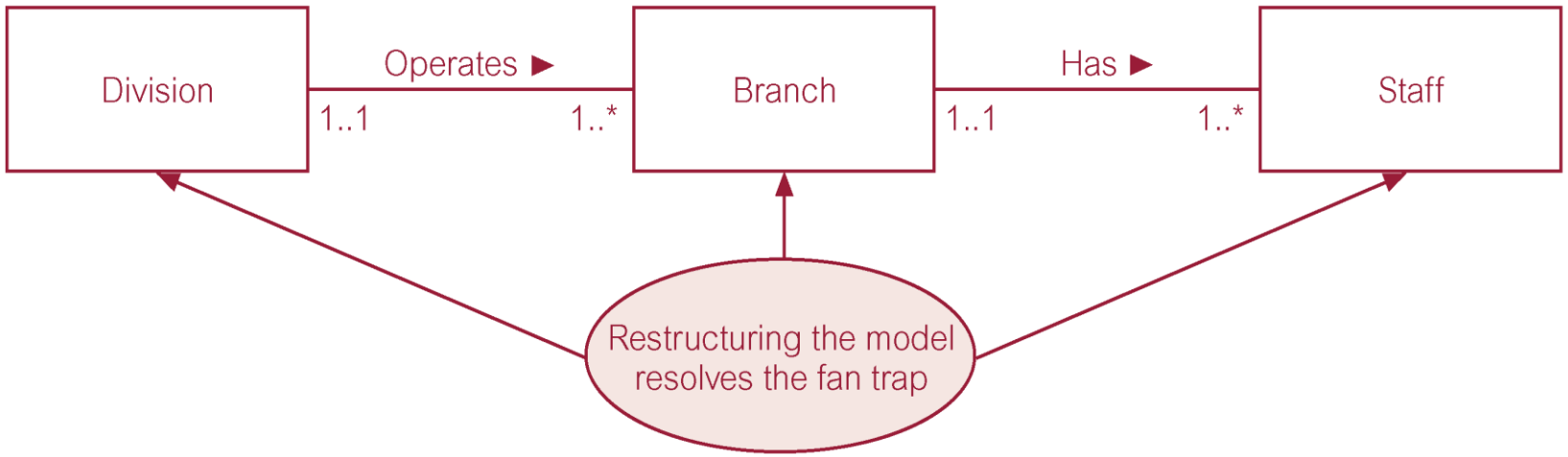
Fan trap

- Layout obscures relationship between staff members and the branch they work at.



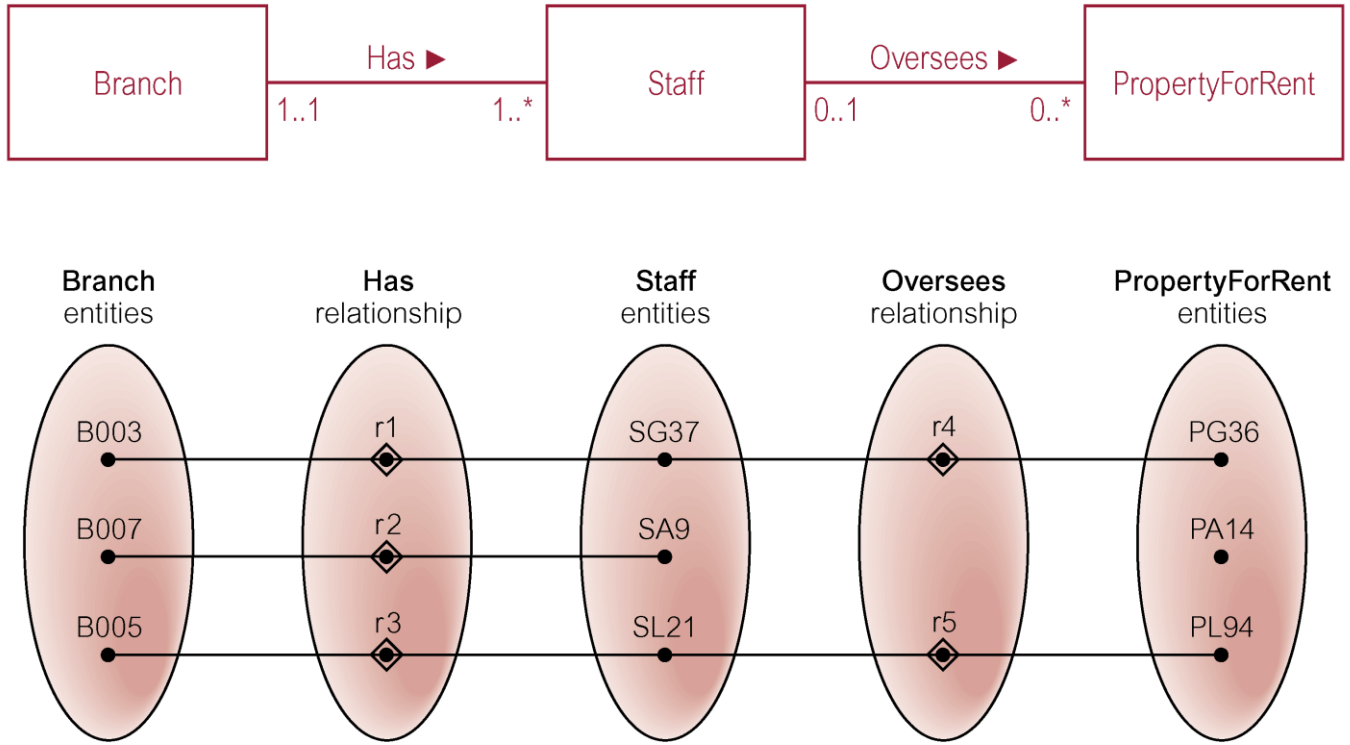
Fan trap: fix

- Rearrange which entity is the parent.



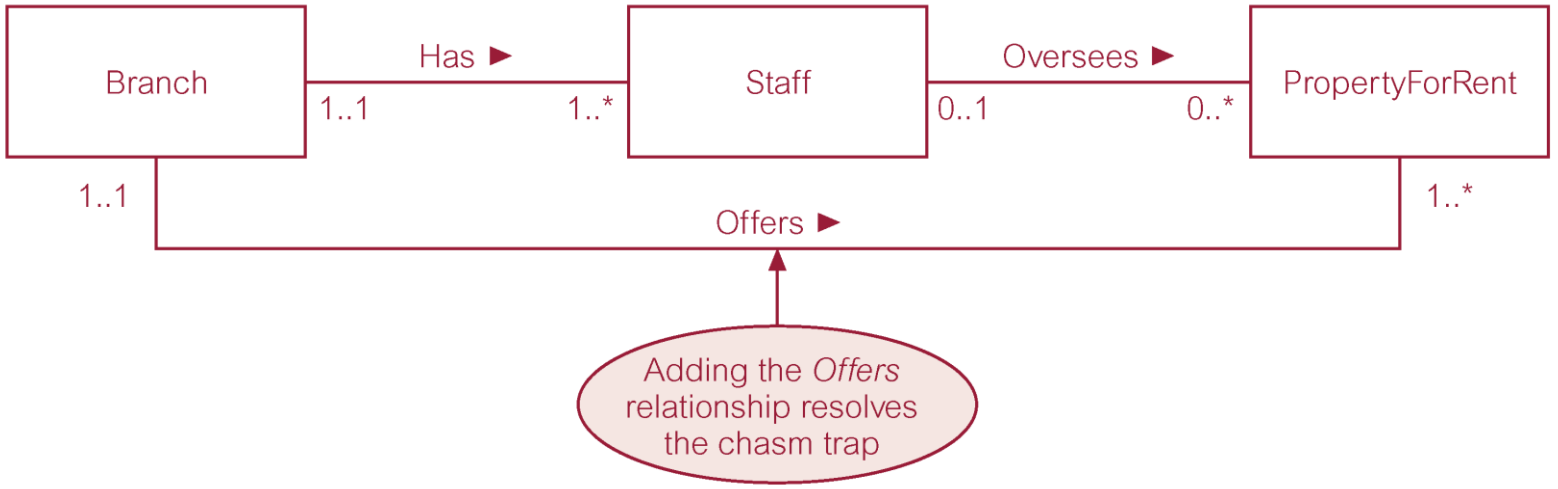
Chasm trap

- Not all staff oversee a property (participation must be optional).
- But keeping it optional also allows a property to be connected to nothing.



Chasm trap: fix

- Supplement with a direct mandatory relationship.



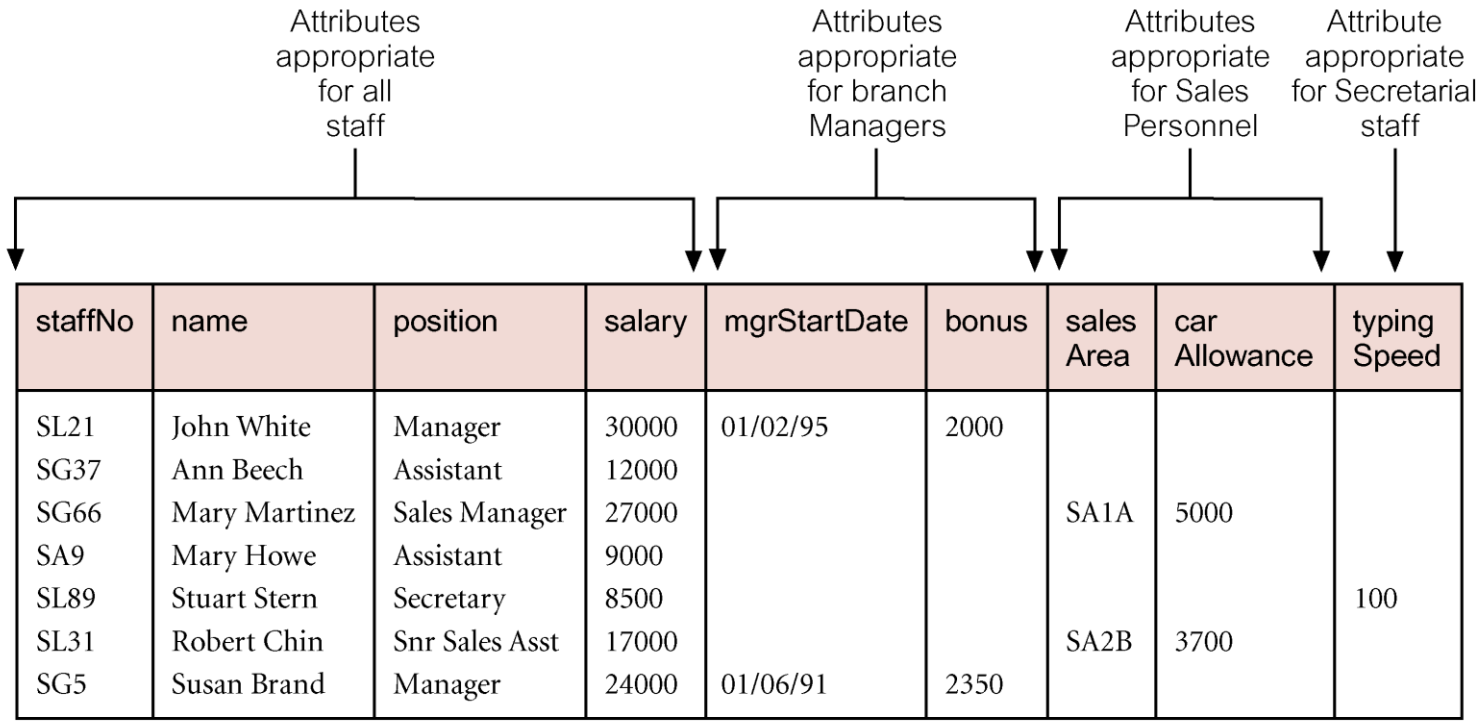
Enhanced ER modeling

Enhanced entity relationship (EER) modelling

- ER with ability to visualize object-oriented concepts (inheritance, super-/subclasses, etc.).
- Inheritance: child classes that receive all their parent class's attributes and functions.
- Specialization: thinking about differences between types of entities in order to break a superclass into child classes.
- Generalization: thinking about commonalities between related classes to extract a parent class they can all inherit from.

Example: subtypes of staff

- All staff have number, name, etc.
- Managers, salespeople, and secretaries have special data.



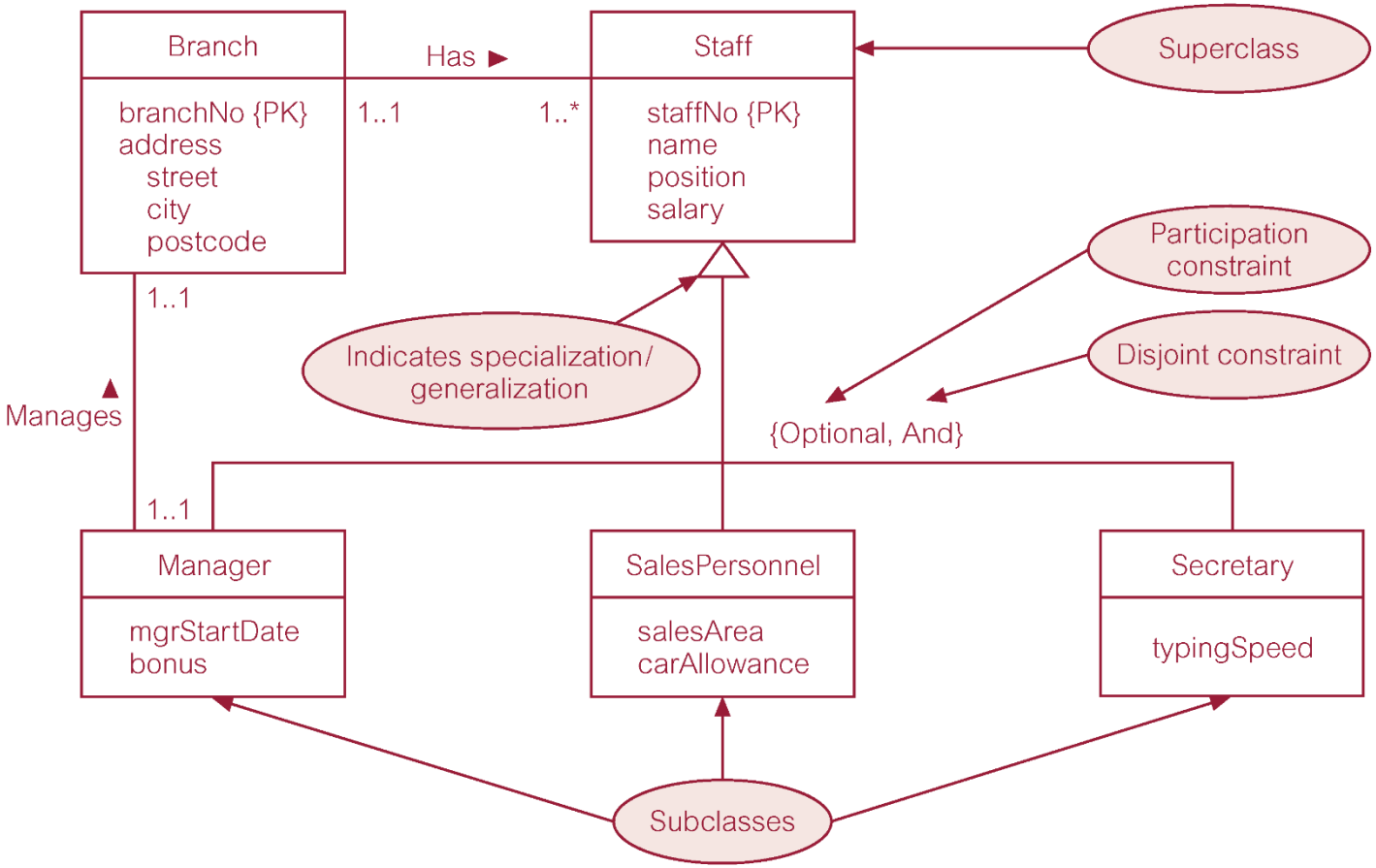
staffNo	name	position	salary	mgrStartDate	bonus	sales Area	car Allowance	typing Speed
SL21	John White	Manager	30000	01/02/95	2000			
SG37	Ann Beech	Assistant	12000					
SG66	Mary Martinez	Sales Manager	27000			SA1A	5000	
SA9	Mary Howe	Assistant	9000					
SL89	Stuart Stern	Secretary	8500					100
SL31	Robert Chin	Snr Sales Asst	17000			SA2B	3700	
SG5	Susan Brand	Manager	24000	01/06/91	2350			

Structural constraints on the inheritance relationship

- Participation constraint: whether a member of a superclass is required to be a member of at least one subclass (*mandatory* vs. *optional*).
- Disjoint constraint: whether an entity instance is allowed to belong to more than one subclass (*and*) or at most one subclass (*or*).
  - Similar to cardinality.
- Four types:
  - Mandatory and disjoint (must belong to exactly one subclass).
  - Optional and disjoint (can belong to up to one subclass).
  - Mandatory and nondisjoint (must belong to at least one subclass).
  - Optional and nondisjoint (can belong to any number of subclasses).

Diagramming inheritance

- Subclasses: line with an arrow pointing from children class(es) to parent.
  - Constraints: a label on the line in curly brackets.



Logical design

Logical design

- After you've chosen the data model you will use, logical design involves designing the schema needed to implement your database in that model.
  - E.g. for a relational database, logical design involves deciding on all the tables in your database.
  - Independent of any particular DBMS software.
- For relational databases, two strategies:
  - Translate from an ER diagram (top-down).
  - Start with one big data table and normalize (bottom-up).

ER translation for relational data models

ER diagram translation

- Input: a conceptual data model.
- Output: relational schema.

Entities: Strong vs. weak entities

- Strong: not dependent on the existence of another entity for existence.
- Weak: dependent on the existence of another entity for existence.
- Example: Users (keeps track of all users in a system – strong) and UserPreferences (stores each user's settings – weak).
- 'Parent/child' type relationship where a child is linked to parent data.

Translating strong and weak entities

- Strong entities: create one table per entity; columns are the entity's attributes.
  - Flatten composite attrs by storing only their constituent parts.
  - Example: Client (clientNo, fName, lName, telNo)
- Weak entities: same as strong entities, except its primary key may not be as clear (could be parent's primary key and possibly some other attribute).
  - Example: Preference (clientNo, prefType, maxRent)

1:* relationships

- Model as a parent (1)-child (*) relationship.
  - Child entity gets a foreign key pointing to the parent entity as an attribute, as well as all of the *relationship* attributes (if any).
  - Example: Staff registers potentially many Clients (at a Branch).
    - Staff (staffNo, fName, lName, position, sex, DOB)
    - Client (clientNo, fName, lName, telNo, staffNo)

1:1 relationships

- Unclear which entity is the parent so use *participation* constraints to inform your actions.
- Three possibilities with different advice:
  - Mandatory participation on both sides.
  - Mandatory participation on one side.
  - Optional participation on both sides.

Mandatory participation on both sides

- All entity instances are forced to participate and match one instance on the other side.
- Best to combine both entities into one table that includes all entity attributes AND relationship attributes.
  - Use one of the entities' PK for the table's PK.
  - Example: mandatory UserPreferences could all be included as part of the User table.
- If this is not feasible, see the 'optional on both sides' advice.

Mandatory participation on one side

- One entity must participate while the other might participate.
- The entity with the choice (optional participant) is the parent and the forced entity is the child.
- Like with 1:* relationship: two entities, and add a foreign key and all relationship attributes to the child.
  - FK works as a (simple) PK in this case.

Optional participation on both sides

- Unclear which one is the better parent without more info.
- Choose one arbitrarily to be the parent (the *more optional* one i.e. the one where fewer instances participate, if applicable).

1:1 recursive relationships

- Recursive relationship = members of a table relates to other members of the same tables.
- Similar advice:
  - Mandatory participation on one or both sides: add another copy of the primary key to the table indicating the entity this one is linked to (also, relationship attributes).
  - Optional participation on one or both sides: create a new table representing the relationship.
    - Table contains two copies of the original table's PK as FKs, showing which pairs of members are related to each other (also, relationship attributes).

: relationships

- Create a dedicated table for the relationship .
  - Include FKs and relationship attributes, which may be (part of) this relationship table's PK.
  - Example: Clients view Properties.
    - Viewing (clientNo, propertyNo, dateView, comment)

Higher-order relationships

- Relationships involving more than two entities.
- Strategy: again, dedicated table for the relationship including FKs and relationship attributes.
- For the new table's primary key, favor the PK of the entity with the "many" (*) cardinality near it (holding values of the other entities fixed, many instances of this entity are possible)
  - Example: Staff registers Clients at a Branch:
    - Registration (clientNo, branchNo, staffNo, dateJoined)

Super/subclasses

- Many options depending on:
  - Structural constraints,
  - Relationships that subclasses are participating in,
  - Number of super/subclasses involved,
  - Etc.

Super/subclass recommendations

- Possible strategies based on *participation constraint* and *disjoint constraint*:
  - Mandatory And: all superclass members have to be at least one subclass.
    - Put all super/subclass attributes in one table with flags to indicate subclass membership; subclass attrs filled in or missing as appropriate.
  - Mandatory Or: all superclass members are exactly one subclass as well.
    - One table for each super + subclass combination possible.
  - Optional And: a superclass member may belong to multiple subclasses.
    - One superclass table with its minimal set of attributes; a second combined table of all subclass attributes with flags indicating membership.
  - Optional Or: a superclass member may belong to up one subclass.
    - One superclass table + one table per subclass referencing superclass table with FK.

Multi-valued attributes

- When an entity is allowed to have multiple values of the same type in an attribute (e.g. multiple phone numbers).
- List values in a new table with owner entity's PK as a FK.

Summary of recommendations

Element	**Action to take**
Strong entity	Create table consisting of simple attrs
Weak entity	Create table consisting of simple attrs (PK to-be-determined)
1:* relationship	Add “1”’s PK to “many” as a FK
1:1 relationship	
Mandatory on both sides	Combine entities into one table
Mandatory on one side	Add “optional”’s PK to “mandatory” as a FK
Optional on both side	Choose one as parent and add its PK as FK to other
: relationship, complex relationships	Create dedicated relationship table with PKs from participating entities
Super/subclasses	Create 1, 2, or many tables depending on constraints
Multi-valued attributes	Create dedicated table with FK linking to original entity

Result schema

- After translation, decide on PKs for weak entities.
- Result: a schema / set of tables with primary and foreign keys that is in **third normal form** (more on this later).
  - Should check that normal form properties hold in case of an error during translation.

Normalization

Normalization

- Goal of normalization is to minimize unnecessary **duplicated data**, which can cause **update anomalies**.
  - Inconsistent data when data is updated in one storage location and not another.
  - Unintended loss of data when deleting some data results in related data being lost.
- Can be used to clean up a ER translation or to design a database from a single big table containing all interesting data.

Different levels of normalization

- Four commonly-used normal forms.
  - First (1NF), second (2NF), third (3NF), and Boyce-Codd normal form (BCNF).
- Each form guarantees that a nice property holds.
- Each form also guarantees properties from previous normal forms.
  - i.e. all models in 3NF are also in 1NF and 2NF; a model in BCNF is also in 3NF.

Normalization ensures two properties of decomposition

- “Lossless-join” property:
  - The exact set of rows in the original table can be reconstructed from a series of joins on the decomposed tables (natural joins on foreign keys).
- “Dependency preservation” property:
  - Any constraint that was imposed on the original table can be imposed by equivalent constraint(s) on decomposed tables.

UNF → 1NF

- Unnormalized form (UNF) has no special properties.
- First normal form (1NF) imposes the requirement that single cells in the table can't contain repeating groups (multi-valued attribute or set of attributes).
  - For a single row and a single column in your tables, there should only be one piece of data stored there.

How to fix UNF tables

- Two strategies:
  - Flattening: separating the compound data into multiple rows, copying data as necessary.
    - No longer “tiers” where some data is summarized and some data is detailed.
  - Breaking the multi-valued data into its own table using the original table’s primary key as the new table’s primary key and giving each piece of data its own row.
- Doesn’t matter which you do – will end up with the *same set of tables* in the end.

1NF → 2NF

- Second normal form (2NF) guarantees there are no partial dependencies on the primary key in a table.

Functional dependencies

- For two attributes or sets of attributes A and B, we say A ® B, A → B, or A **functionally determines** B if fixing A to a specific value causes B to be determined.
  - Example: For Staff, staffNo ® position, but position ~~®~~ staffNo.
- A value of (left side) is only associated with one value of (right side).
- Relationship has to hold theoretically, not just for a particular instance/snapshot of the table.
  - staffNo ® fullName, but fullName ~~®~~ staffNo because it is theoretically possible (though rare) for multiple members of staff to have the same name.

Full vs. partial dependencies

- Full functional dependency: B is determined by the full set of A's values.
- Partial functional dependency: Only *part* of A is needed to determine the value of B.

Returning to 1NF → 2NF

- 2NF means there are no non-primary-key attributes that are fully determined by a *part* of the primary key.
  - So: only needs to be checked for relationships with a composite primary key.

How to fix 1NF

- Whenever you find an attribute(s) that is dependent on (functionally determined by) only part of the primary key, create a new table with (1) a copy of the primary key part, and (2) the attributes it fully determines.
- Repeat process until no partial dependencies on PKs exist.
- Example:
  - Lease (clientNo, propNo, cName, address, leaseStart)
  - Break into three tables:
    - Client (clientNo, cName)
    - Property (propNo, address)
    - Lease (clientNo, propNo, leaseStart)

2NF result

- Set of tables that contain only full functional dependencies between primary key and other attributes.

2NF → 3NF

- Third normal form (3NF) ensures that there are no non-primary-key attributes transitively dependent on the primary key.

Transitive dependencies

- A transitive dependency is when you have A ® B and B ® C (meaning that A ® C).
  - But B ~~®~~ A and C ~~®~~ A (that would be uninteresting).
- Example:
  - propertyNo, inspectionDate ® staffNo
  - staffNo ® staffName
  - “From a property ID and an inspection date, you can deduce staff ID; from staff ID you can deduce staff name”.
    - From property ID and inspection date, you can deduce staff name.
  - If all four attributes are in the same table, this creates issues.

How to fix 2NF

- For primary key A, if you have A ® B and B ® C, split B and C into their own table (primary key: B).
  - Keep B (but not C) in the original table along with A. B functions as a foreign key to the new table.

3NF result

- Functional dependencies are captured by the primary key vs. other attributes relationship of a table.
  - Not “hidden” among non-primary-keys in a table.

3NF → BCNF

- Boyce-Codd normal form (BCNF) requires every functional dependency determinant (left side) to be a candidate key for the table.
  - If you have a functional dependency that violates this property, it's possible to get update anomalies where changing the attribute on the righthand side requires updating it in multiple places for rows sharing the same lefthand side value.

BCNF violation example

- Consider a table holding information about pupils, the subjects that they study, and the teachers who teach them. In this scenario, assume that:
  - For a given subject, each pupil is taught by one teacher.
  - Each teacher only teaches one subject, but a subject may be taught by several teachers.
- So:
  - Given a pupil + subject combo, teacher is known (pupil, subject → teacher)
  - Given a teacher (+pupil), the subject is known (pupil, teacher → subject; teacher → subject).

BCNF violation example (cont.)

- In this Teaching table with pupils, subjects and teachers, we have:
  - pupil, subject → teacher
  - pupil, teacher → subject
  - teacher → subject
- Teacher is the left side (determinant) of a FD not a candidate key for the relation.
- Reflects the fact that:
  - Duplication of data (that a teacher teaches a particular subject) means updating data in one place requires updating it in multiple places.
  - The hidden relationship (between teachers and subjects) can be lost if pupils are deleted.

How to fix 3NF

- Spin that problematic functional dependency into its own table.
  - Retain the determinant (left side) as a foreign key in the original table.
    - Example: Teaching (<u>pupil</u>, <u>subject</u>, teacher) becomes
      Tutor (<u>pupil</u>, <u>teacher</u>) and TaughtSubject (<u>teacher</u>, subject)

BCNF result

- Less redundancy.
- Downside: It is possible to split a functional dependency across two tables when you make this change causing it to “disappear” and be difficult to enforce
  - Need to trade off less redundancy (BCNF) vs. preserving all constraints (3NF)

Keeping track of functional dependencies

- Being able to identify functional dependencies is important for normalization, but finding and working with all of them is tedious/hard.
- Need to instead work with a minimal/simplified set that stands in for the full set.
  - Keep left side of dependencies minimal.
  - Don’t have dependencies that are the composition of two simpler dependencies.
  - Don’t bother with trivial dependencies.