Topic 06: Logical Database Design

ICT285 Databases
Dr Danny Toohey

About this topic

- In this topic we continue our coverage of the database design process by looking at how we can convert an ERD produced in the conceptual design stage to a set of normalised tables that supports the organisation's business processing requirements. You have already covered the relational model in some depth, and the conversion from an ERD follows some simple rules to ensure you achieve a good design.

Topic learning outcomes

After completing this topic you should be able to:

- Describe the activities in logical database design
- Convert an ERD to a relational schema in 3NF
- Validate a relational schema against the business transactions it is required to support
- Identify and document all integrity constraints for the logical model: required data, attribute domain constraints, entity integrity, referential integrity and enterprise constraints
- Define appropriate referential actions (‘foreign key rules’) to ensure that referential integrity is maintained when the database is updated, inserted or deleted
- Document the logical database design in a data dictionary

Resources for this topic

READING

- Text, Chapter 6: Transforming data models into database designs
- Skim the section on Data Types for now (we will come back to it later)
- You can also skim the section on ‘Representing Ternary and Higher Order Relationships’. And just try to get the basics from ‘Designing for Minimum Cardinality’

The first four labs provided you with a lot of practice in creating and querying Oracle tables, and in the previous lab you used Visio to create entity-relationship models. This lab is designed to help you understand how the two are connected, through the database design process that moves from a conceptual model (the ERD) to a set of normalised relations that can be implemented in a DBMS.

LAB 6 – CONVERTING AN ENTITY-RELATIONSHIP MODEL TO TABLES

Topic outline

- Where logical design fits into database design
- Convert ERD to tables
- Integrity constraints
- Confirm logical design
- Document the logical design in the data dictionary

Topic 06: Part 01 - Where logical design fits into database design

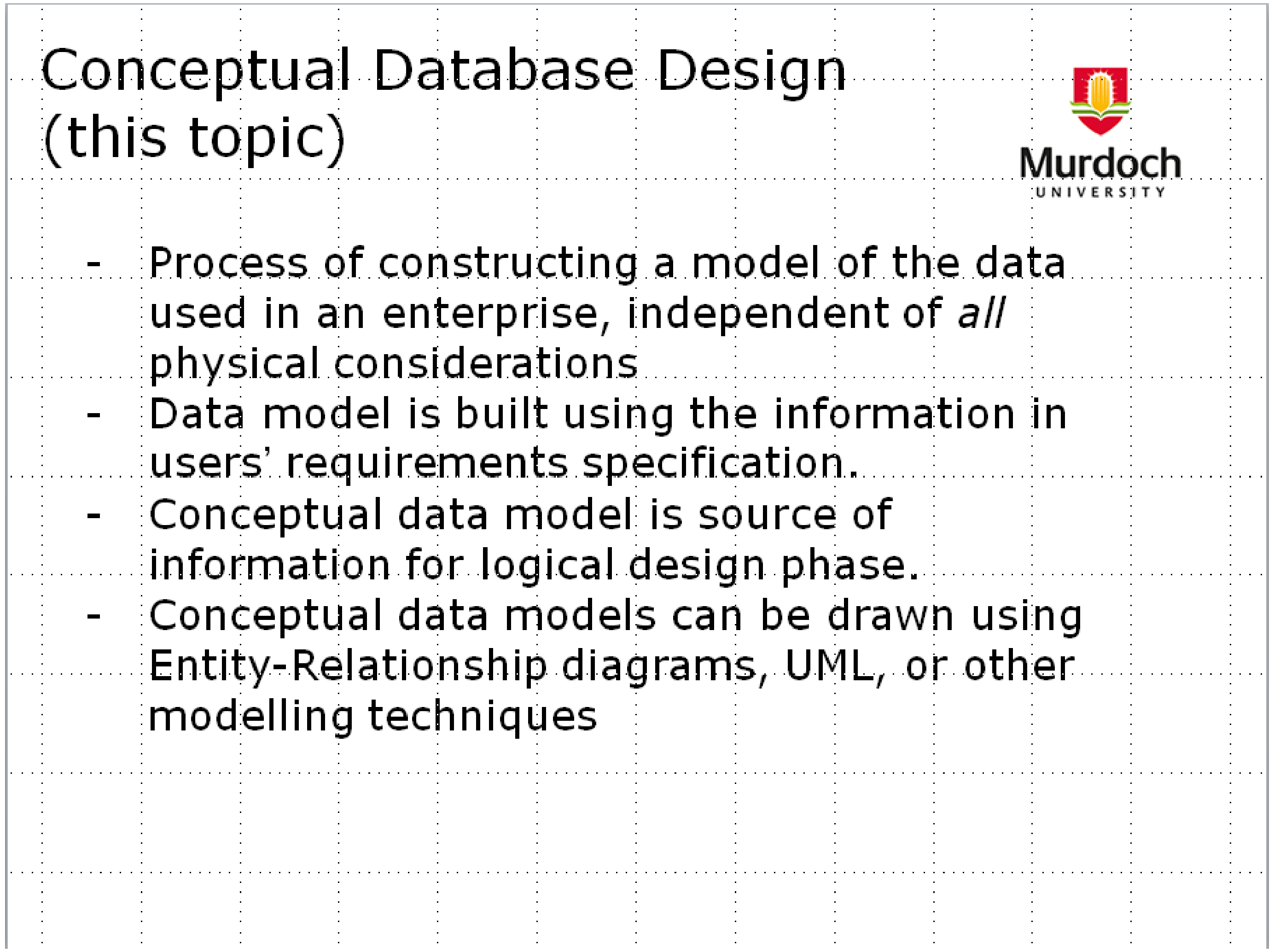
Database Design

Process of creating a design for a database that will support the enterprise's mission statement and mission objectives for the required database system

Three phases of database design:

- Conceptual database design
- Logical database design
- Physical database design

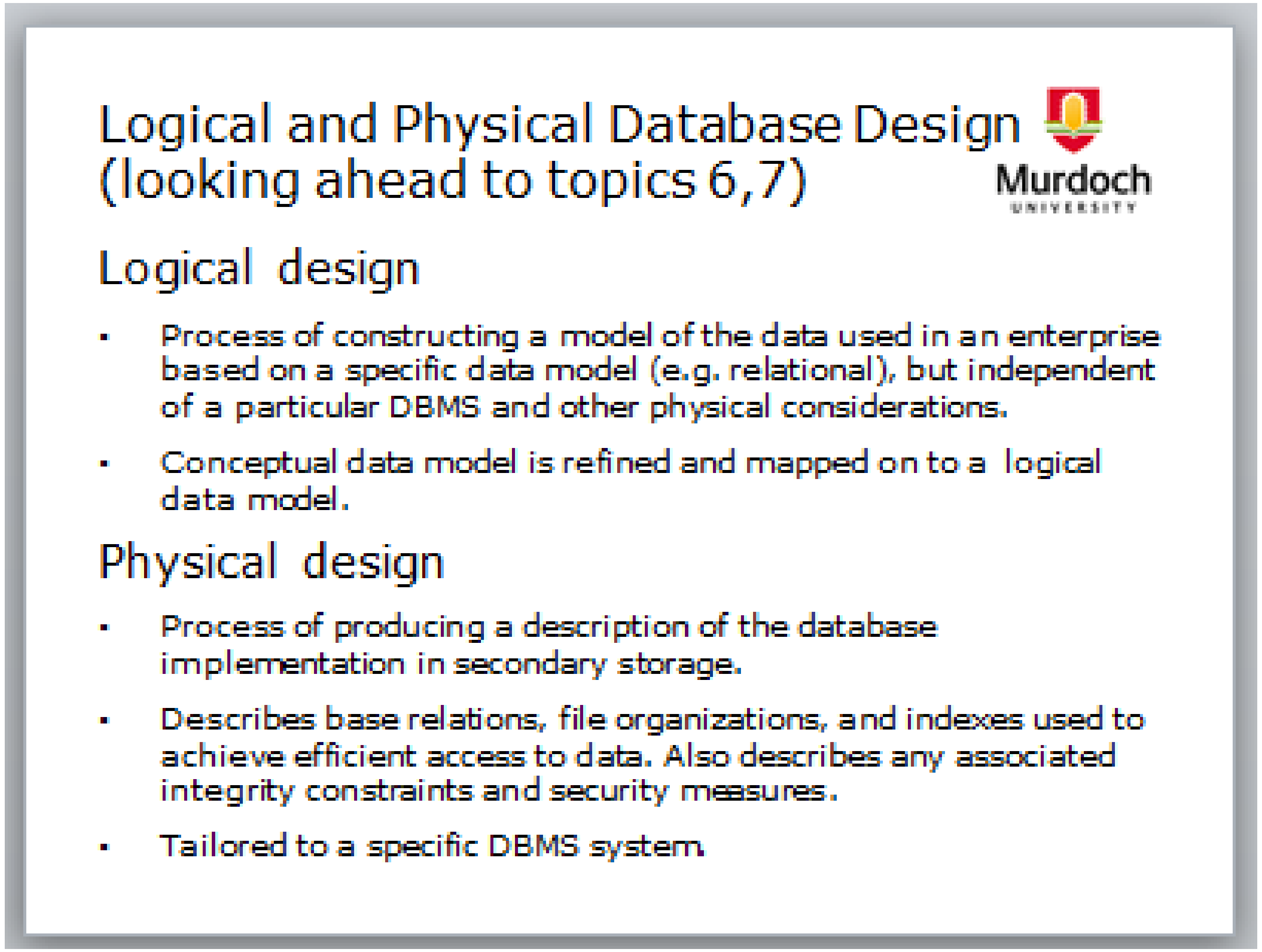
Reminder (topic 5)



Conceptual Database Design (this topic)

- Process of constructing a model of the data used in an enterprise, independent of *all* physical considerations
- Data model is built using the information in users' requirements specification.
- Conceptual data model is source of information for logical design phase.
- Conceptual data models can be drawn using Entity-Relationship diagrams, UML, or other modelling techniques

Reminder (topic 5)



Logical and Physical Database Design (looking ahead to topics 6,7)

Murdoch UNIVERSITY

Logical design

- Process of constructing a model of the data used in an enterprise based on a specific data model (e.g. relational), but independent of a particular DBMS and other physical considerations.
- Conceptual data model is refined and mapped on to a logical data model.

Physical design

- Process of producing a description of the database implementation in secondary storage.
- Describes base relations, file organizations, and indexes used to achieve efficient access to data. Also describes any associated integrity constraints and security measures.
- Tailored to a specific DBMS system.

Logical database design

- Create a model of the data based on a particular data model (EG- Relational model for this lecture/course). The DBMS (Oracle/MYSQL) and other physical constraints are not factors at this phase
- So once we have our conceptual data model from previous phase we refine and map it to our logical data model
- Logical database design produces a relational schema and associated documentation, which can be converted to a physical design in the chosen DBMS and eventually implemented
  - *What advantages are there in separating the conceptual design from the logical design process?*
    - *We are designing the data in the most appropriate way. Provides freedom to change course from not appropriate data model (relational model) to a more appropriate data model*
    - Bad logical database design results in bad physical database design, and generally results in poor database performance. EG queries are slow, might not be able to get data we need due to poor design

Why bother?

- **"Bad logical database design results in bad physical database design, and generally results in poor database performance.** So, if it is your responsibility to design a database from scratch, be sure you take the necessary time and effort to get the logical database design right. Once the logical design is right, then you also need to take the time to get the physical design right.
- Both the logical and physical design must be right before you can expect to get good performance out of your database. If the logical design is not right before you begin the development of your application, it is too late after the application has been implemented to fix it. No amount of fast, expensive hardware can fix the poor performance caused by poor logical database design…"

http://www.sql-server-performance.com/database_design.asp

Logical design for the relational model

- We will assume that we have decided to implement our design using the relational model:
  - The vast majority of new databases are implemented in DBMS based on the RM
  - You are already familiar with the RM
- When we convert an ERD to a logical relational schema, we must represent all of the features of the ERD in tables (relations)
- We must also ensure that the *business requirements* we began to capture in the conceptual modelling stage continue to be represented in the logical design

A logical database design methodology Activities

1. Represent entities as tables
2. Represent relationships using foreign keys
3. Validate model using normalisation
4. Validate model against user transactions
5. Define and check integrity constraints
6. Review logical model with users
7. Merge local models into a global model
8. Check for future growth
9. Document the logical design

Adapted from Connolly & Begg

Take-aways...

- Logical database design is the second step in the database design process
- It involves constructing a model of the data used in an enterprise based on a specific data model (e.g. relational), but independent of a particular DBMS and other physical considerations
- The outcome of the logical design process is a relational schema and associated documentation that can then be converted to a physical design in the chosen DBMS and eventually implemented

Topic 06: Part 02 - Convert ERD to tables

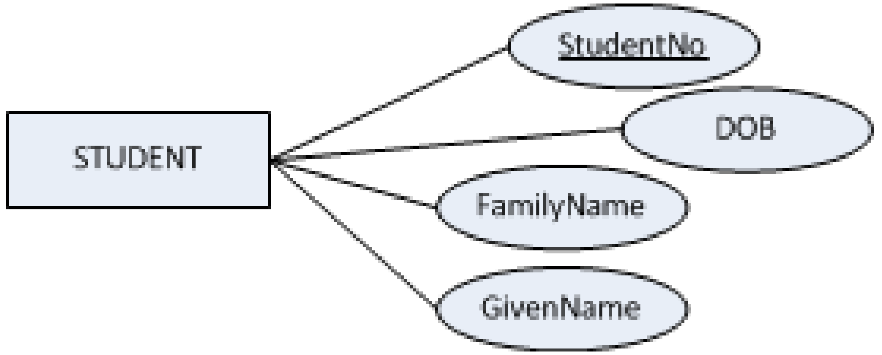
Convert ERD to tables

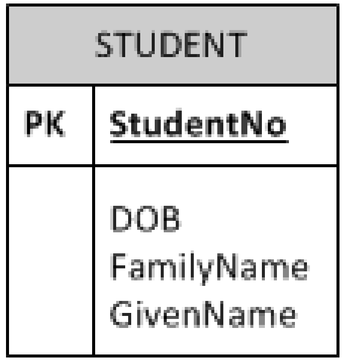
- Represent entities as tables
- Represent relationships using foreign keys
- Subtypes and Supertypes
- Verify using normalisation (3NF)

Represent entities as tables

Create a table for each entity

- Create a table that includes all the attributes of the entity
- The name of the entity becomes the name of the table
- The name of the attributes become the names of the columns
- The key attribute(s) of the entity becomes the primary key of the table
- If necessary, normalise the table further to 3NF



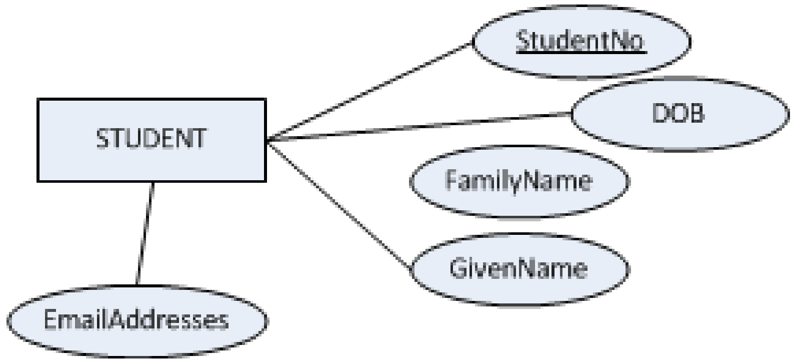


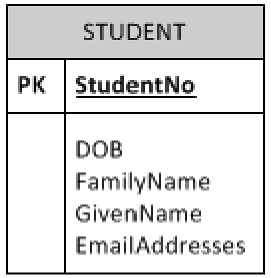
STUDENT	
PK	**StudentNo**
	DOB FamilyName GivenName

STUDENT (StudentNo, DOB, FamilyName, GivenName)

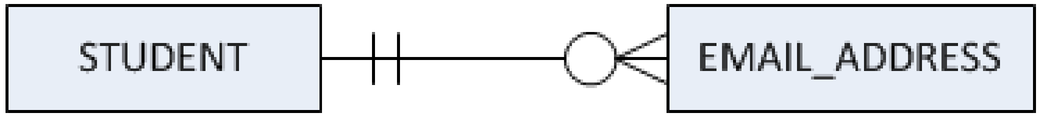
Multi-valued attributes

- If you find your ERD still has multi-valued attributes, correct them now
- Often the solution is to create a new entity in a 1:M relationship





STUDENT	
PK	StudentNo
	DOB FamilyName GivenName EmailAddresses

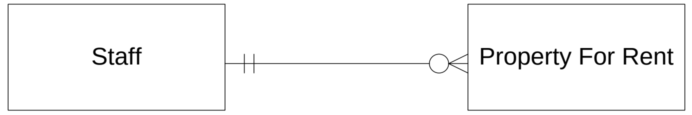


STUDENT_EMAIL (<u>StudentNo, Email</u>, EmailType)

Representing 1:N (1 To Many) relationships

Representing 1:N (1 to Many)relationships

- Represent each entity as a table
- **Include the primary key of the 1-side (parent) table as foreign key in the N-side (child) table**



The entities:

STAFF(<u>StaffNo</u>, Name, Position, …)

PROPERTYFORRENT (<u>PropertyNo</u>, Address, Type, Rooms,…)

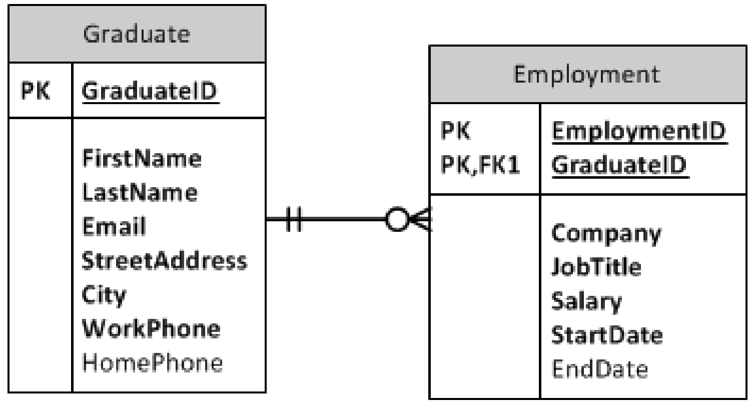
Become the relations:

STAFF (<u>StaffNo</u>, Name, Position, …)

PROPERTYFORRENT (<u>PropertyNo</u>, Address, Type, Rooms, …, **StaffNo**)

1:N relationships with ID-dependent entities

- In a 1:N relationship involving an ID-dependent entity with a composite key, the foreign key attribute is *already in* the child table, as it is part of the PK

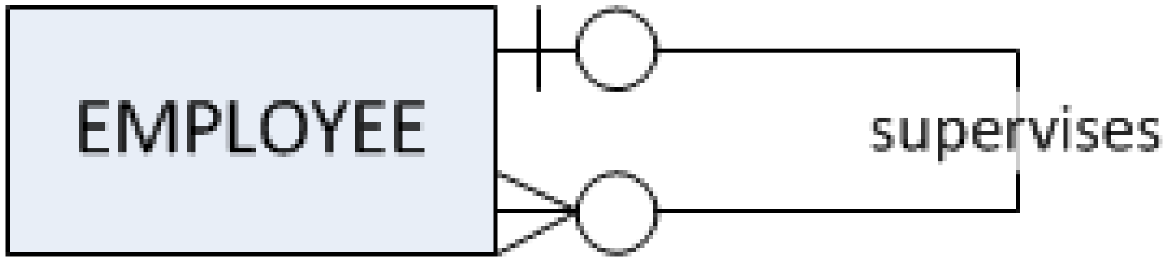


GRADUATE (<u>GraduateID</u>, FirstName, LastName, ….)

EMPLOYMENT (<u>EmploymentID, **GraduateID**</u>, Company, JobTitle, …)

1:N recursive relationships

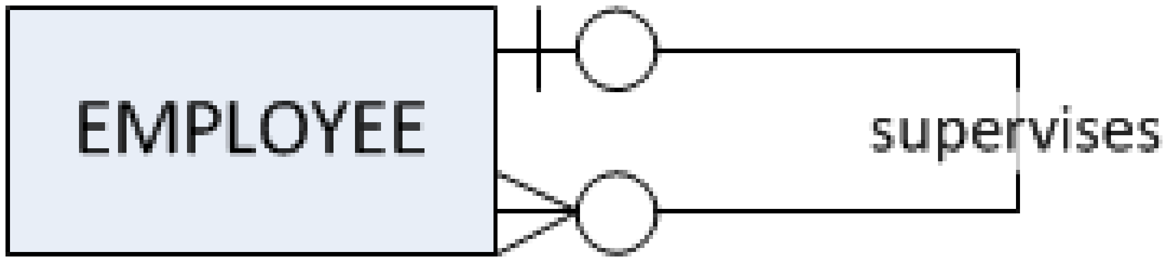
- Can put the foreign key in the *same* table, in a different role
- In this example any SupervisorID (FK) is found as Employee ID (PK) in another record



EMPLOYEE (<u>EmployeeID</u>, Name, … **SupervisorID**)

1:N recursive relationships

- Can also create a *new* table for the relationship, consisting of the PK and FK attributes



SUPERVISION (**EmployeeID, SupervisorID**)

- The PK of SUPERVISION is EmployeeID, SupervisorID
- And **EmployeeID** and **SupervisorID** are each separate FKs to the EMPLOYEE table

Creating another table to represent a 1:N relationship

It's also possible to create a third table to represent a 1:N relationship. This could be a good idea if:

- There is a possibility that the business rules may change so that the relationship becomes M:N in the future
- If the N side has *optional* participation – this may result in a table with many nulls in the FK field (e.g. if a property might not be managed by a member of staff)

PROPERTY (PropertyNo, Address, Rooms, …)
STAFF (StaffNo, Name, …)
MANAGES (**PropertyNo, StaffNo**)

Representing 1:1 relationships

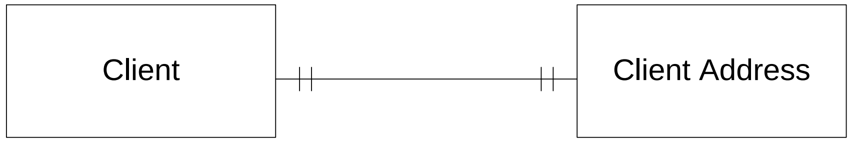
Representing 1:1 Relationships

- Unlike 1:N relationships, there is a choice of where the FK will be placed
- This choice will depend largely on the nature of the relationship:
  - Mandatory on both sides of the relationship?
  - Mandatory on one side only of the relationship?
  - Optional on both sides of the relationship?

Mandatory on *both* sides of the relationship

- In this case, we may be able to combine the entities into a **single** relation

  - In the example, IF each client must have one and only one address, then we would put the address attributes in the CLIENT relation



The entities:

CLIENT(<u>ClientNo</u>, Name, Tel …)

CLIENTADDRESS (Street, Suburb, City, …)

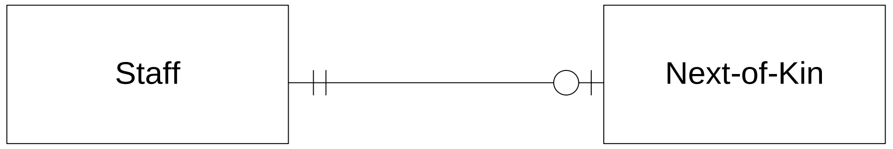
Become the single relation:

CLIENT(<u>ClientNo</u>, Name, Tel, Street, Suburb, City …)

Mandatory on *one side* only of the relationship

- In this case, we would put the FK in the relation that has the **optional** participation

- In this way, we remove the nulls that would result if we put the FK on the mandatory side



The entities:

STAFF(<u>StaffNo</u>, Name, Position …)

NEXT-OF-KIN (Name, Address, Telephone …)

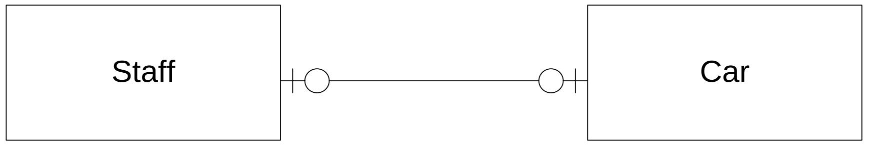
Become the relations:

STAFF(<u>StaffNo</u>, Name, Position …)

NEXT-OF-KIN (Name, Address, Telephone, **<u>StaffNo</u>** …)

Optional on *both sides* of the relationship

- In this case, the positioning of the FK is arbitrary
  - As a rule of thumb, it would be placed on the side of the relationship where it would result in *fewer* nulls
  - In the example, we have chosen to put the FK in the CAR relation as the majority of cars are used by staff, but only a small number of staff will use a car



The entities:

STAFF(<u>StaffNo</u>, Name, Position …)

CAR (<u>Registration</u>, …)

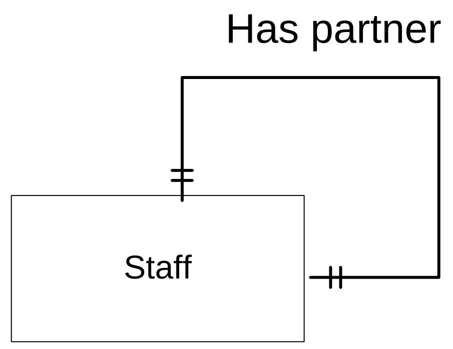
Become the relations:

STAFF(<u>StaffNo</u>, Name, Position …)

CAR (<u>Registration</u>, **StaffNo**, …)

One-to-one recursive relationships

- For a 1:1 recursive relationship, follow the same guidelines as for a 1:1 relationship
  - **e.g. mandatory participation on both sides,** represent the recursive relationship as a single relation with two copies of the primary key (one renamed)



Becomes the relation:

STAFF (<u>StaffNo</u>, Name, Position …, **PartnerNo**)

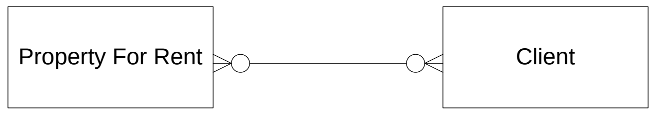
Representing M:N relationships

M:N relationships

A M:N relationship in an ERD *cannot* be represented directly as a M:N relationship in the relational model

Instead, replace it with **two 1:M relationships:**

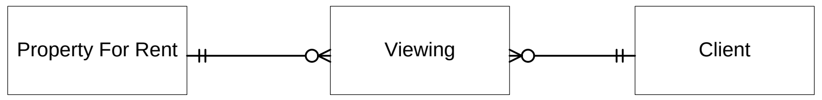
Create a relation to represent the relationship and include any attributes that are part of the relationship



The entities:

PFR(<u>PropertyNo</u>, Address, Type …)

CLIENT (<u>ClientNo</u>, Name, …)



Become the relations:

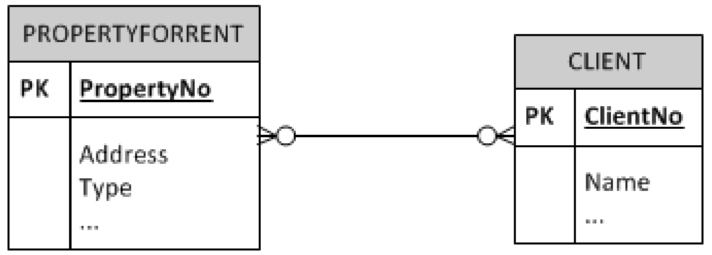
PFR(<u>PropertyNo</u>, Address, Type …)

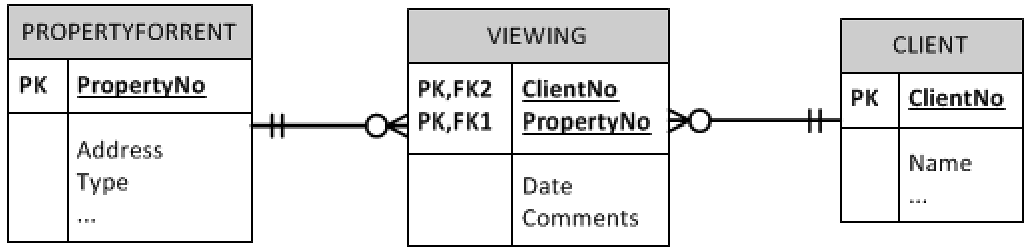
VIEWING (<u>**ClientNo,PropertyNo**</u>, Date, Comments)

CLIENT (<u>ClientNo</u>, Name, …)

M:N relationships

- It is good practice to ensure all M:N relationships are converted to pairs of 1:N relationships in the ERD, BEFORE commencing logical design



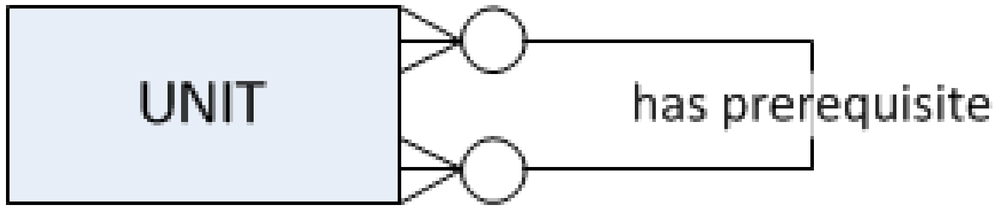


- This often reveals additional attributes or relationships belonging to the associative (intersection) entity

Recursive M:N relationships

Treat as other M:N relationships by creating an intersection entity

The primary keys of the intersection entity/table will need to be named to indicate their roles



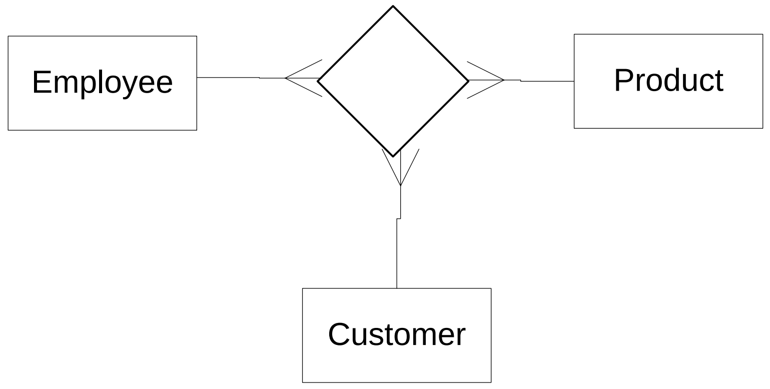
UNIT (<u>UnitCode</u>, UnitTitle, …)

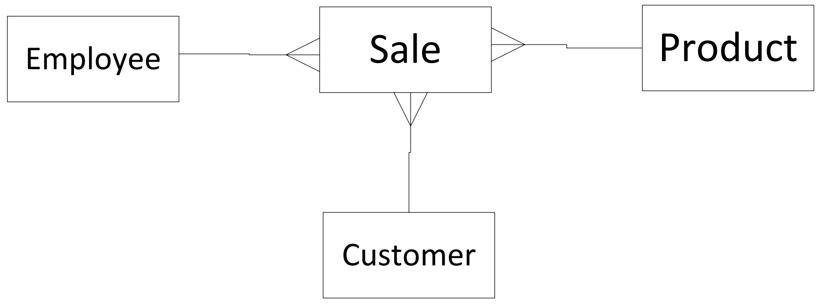
PREREQUISITE (**<u>UnitCode, PrereqUnitCode</u>**)

Ternary (and higher) relationships

A ternary relationship is a relationship involving *three* entities

- Should already have been resolved into pairs of 1:M relationships and treated as per earlier discussion of M:N relationships





SALE (<u>**Employee, Customer, Product**</u>, Date)

Take-aways...

All features of the ERD must be represented in a relational schema:

- Entities become tables, and attributes of the entity become attributes of the table
- Relationships between entities are represented by foreign keys in tables
- There are rules and guidelines about how to convert 1:N, 1:1 and M:N relationships to tables with FKs

Representing subtypes and supertypes

Subtypes and supertypes - reminder

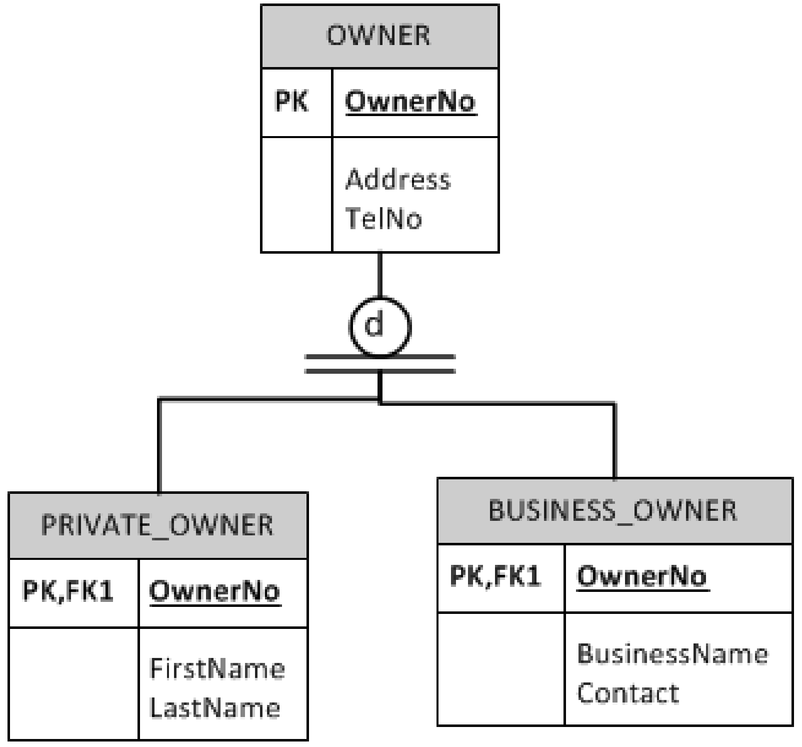
- We can model generalisation/specialisation hierarchies with subtypes and supertypes
    - e.g. STUDENTS may be Undergraduate or Postgraduate

Constraints on subtypes include:

- **Mandatory/optional** – every member of the supertype MUST be one of the subtypes, or may not be
- **Disjoint/overlapping** – an entity is one **or** other of the subtypes, but can’t be both (or can be)

Subtypes and supertypes - diagramming

- Visio 2010 ‘category’ symbol

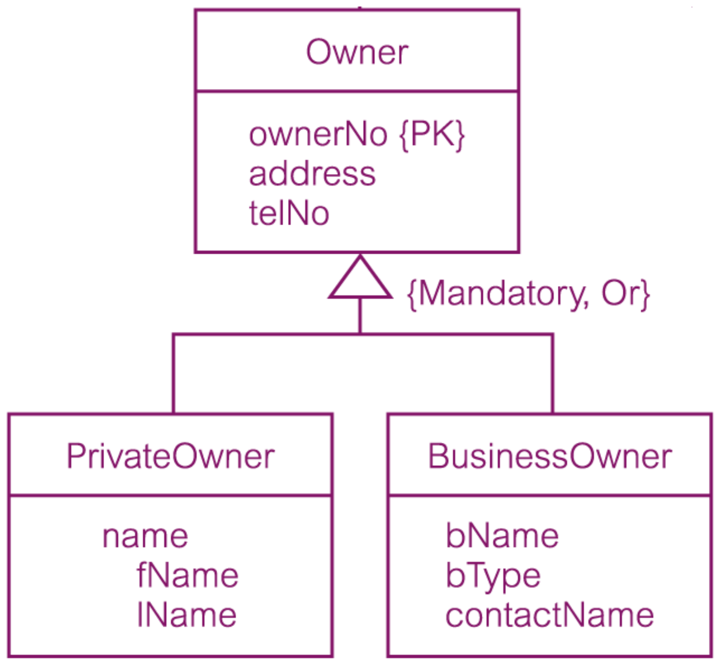


This shows that the subtypes are disjoint (the ‘d’) and that they are mandatory (the double line)

Subtypes and supertypes - diagramming

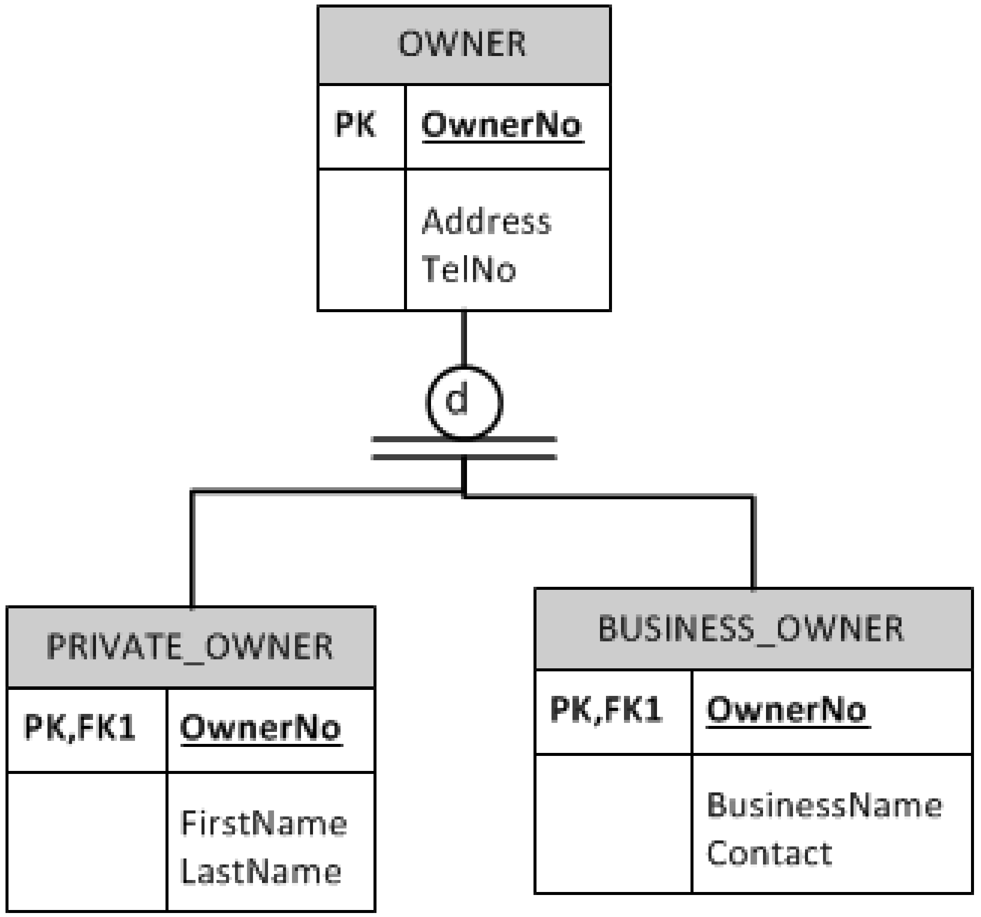
- As the 'category' symbol has been discontinued in later versions of Visio, you could use the UML symbol instead and write the constraint information beside it

This shows that the subtypes are disjoint (the 'or') and that they are mandatory



Primary keys and subtypes

- Normally the primary key of the supertype and the key of the subtype are the same domain, although they may have different names
- The supertype primary key is included as foreign key in the subtype table



Representing subtypes as tables

There are various ways in which we can represent subtypes, for example:

- A single table including all subtype and supertype attributes
- A table for the supertype and a table for each subtype
- A table for each subtype, each including the shared attributes from the supertype

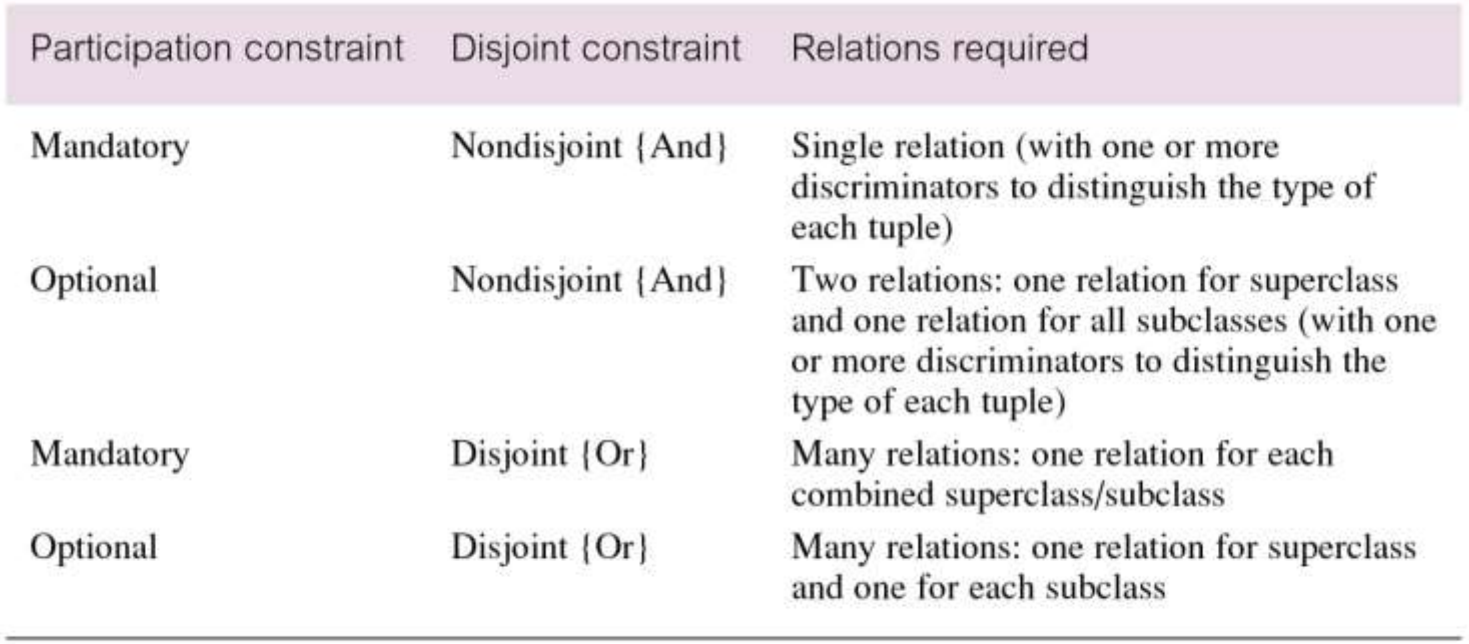
- There are no hard and fast rules that apply in every situation, but we can distinguish some general guidelines (next)

Representing subtypes as tables

- The nature of the subtype-supertype relationship and its context in the data model will together determine the most appropriate option
- Generally: if it’s possible to be in *more than one* subtype (overlapping), a single table is more appropriate, while separate tables are more appropriate for *disjoint* (exclusive) subtypes
- We also need to consider whether it is *mandatory* to belong to a subtype or optional
- Other features such as the number of shared vs distinct attributes and relationships with other entities also play a part

Some guidelines based on participation/disjointness

- Representing Subclasses in Tables (from Connolly & Begg, 2005)

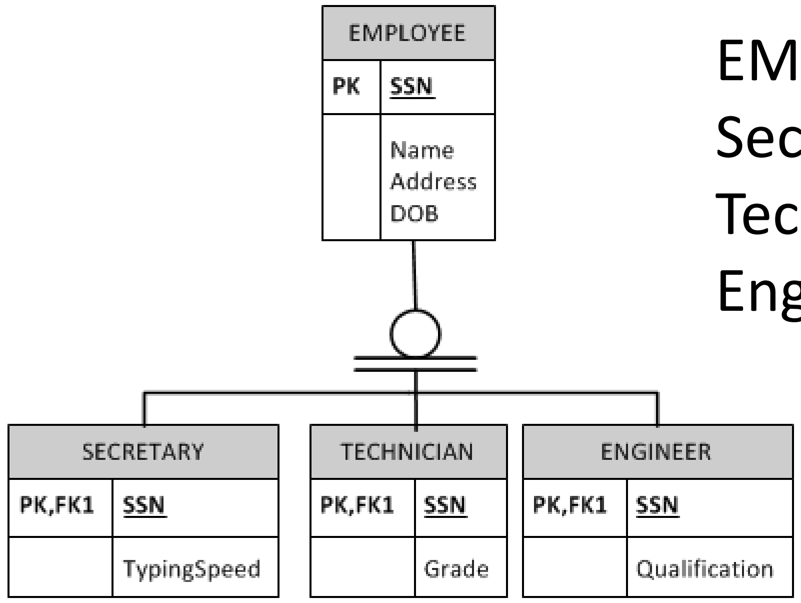


Participation constraint	Disjoint constraint	Relations required
Mandatory	Nondisjoint {And}	Single relation (with one or more discriminators to distinguish the type of each tuple)
Optional	Nondisjoint {And}	Two relations: one relation for superclass and one relation for all subclasses (with one or more 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

Representing subtypes: option 1

Mandatory, overlapping (non-disjoint)

Create a single relation including all subtype and supertype attributes, plus an attribute(s) to identify the subtype

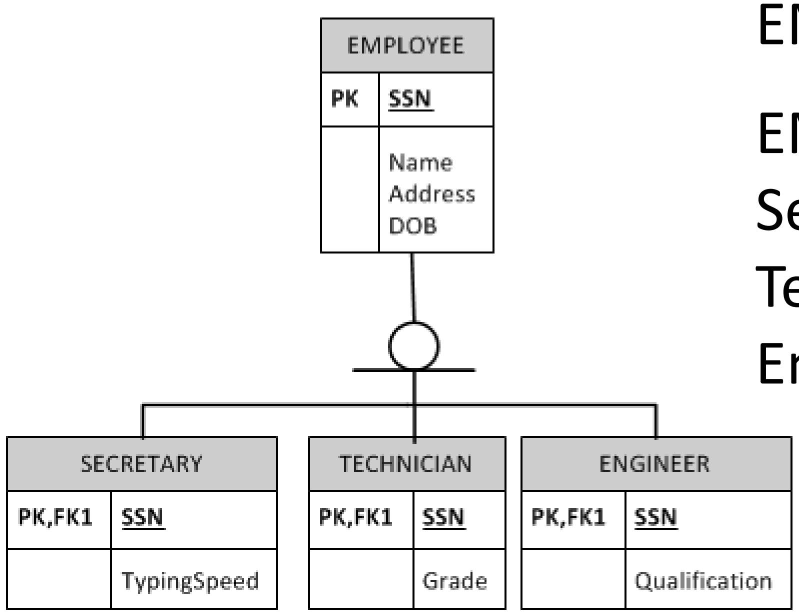


EMPLOYEE (<u>SSN</u>, Name, Address, DOB, SecretaryFlag, TypingSpeed, TechnicianFlag, Grade, EngineerFlag, Qualification)

Representing subtypes: option 2

Optional, overlapping

Create a relation for the supertype, and a single table for all subtypes including an attribute(s) to identify the subtype



EMPLOYEE (<u>SSN</u>, Name, Address, DOB)

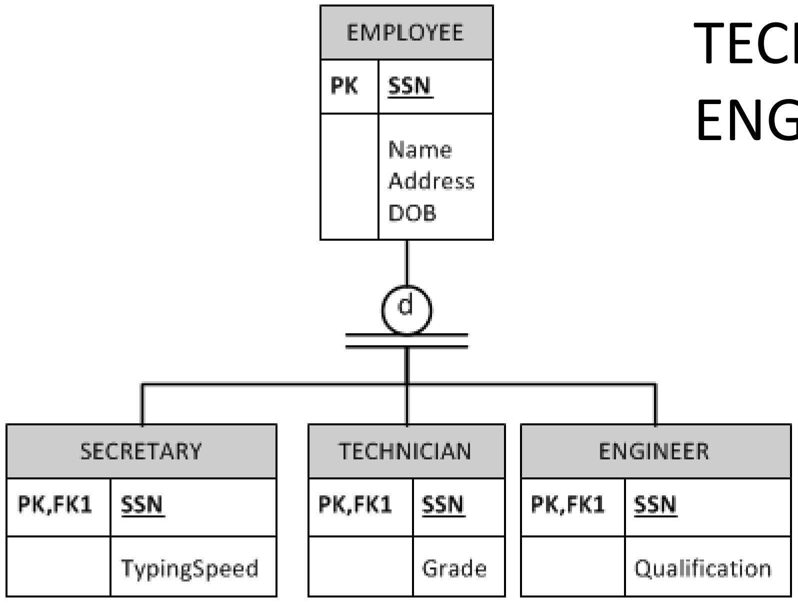
EMPLOYEEDETAILS (**<u>SSN</u>**, SecretaryFlag, TypingSpeed, TechnicianFlag, Grade, EngineerFlag, Qualification)

(could also implement in a single table)

Representing subtypes: option 3

Mandatory, disjoint

Create separate relations for each subtype containing the complete set of subtype attributes:

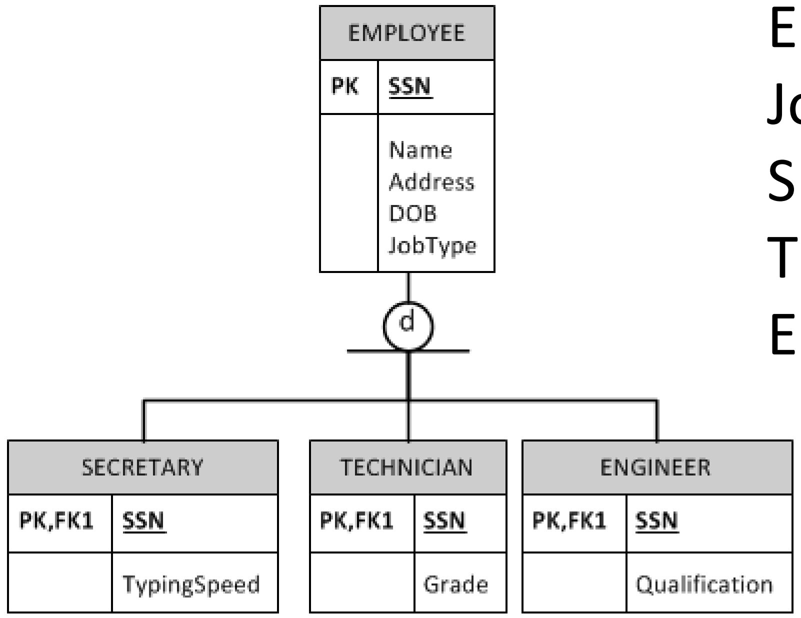


SECRETARY (<u>SSN</u>, Name, Address, DOB, TypingSpeed)
TECHNICIAN (<u>SSN</u>, Name, Address, DOB, Grade)
ENGINEER (<u>SSN</u>, Name, Address, DOB, Qualification)

Representing subtypes: option 4

Optional, disjoint

Create one relation for the supertype entity containing the shared attributes and an attribute to identify the subtype, and a separate relation for each subtype containing the subtype attributes:



EMPLOYEE (<u>SSN</u>, Name, Address, DOB, JobType)

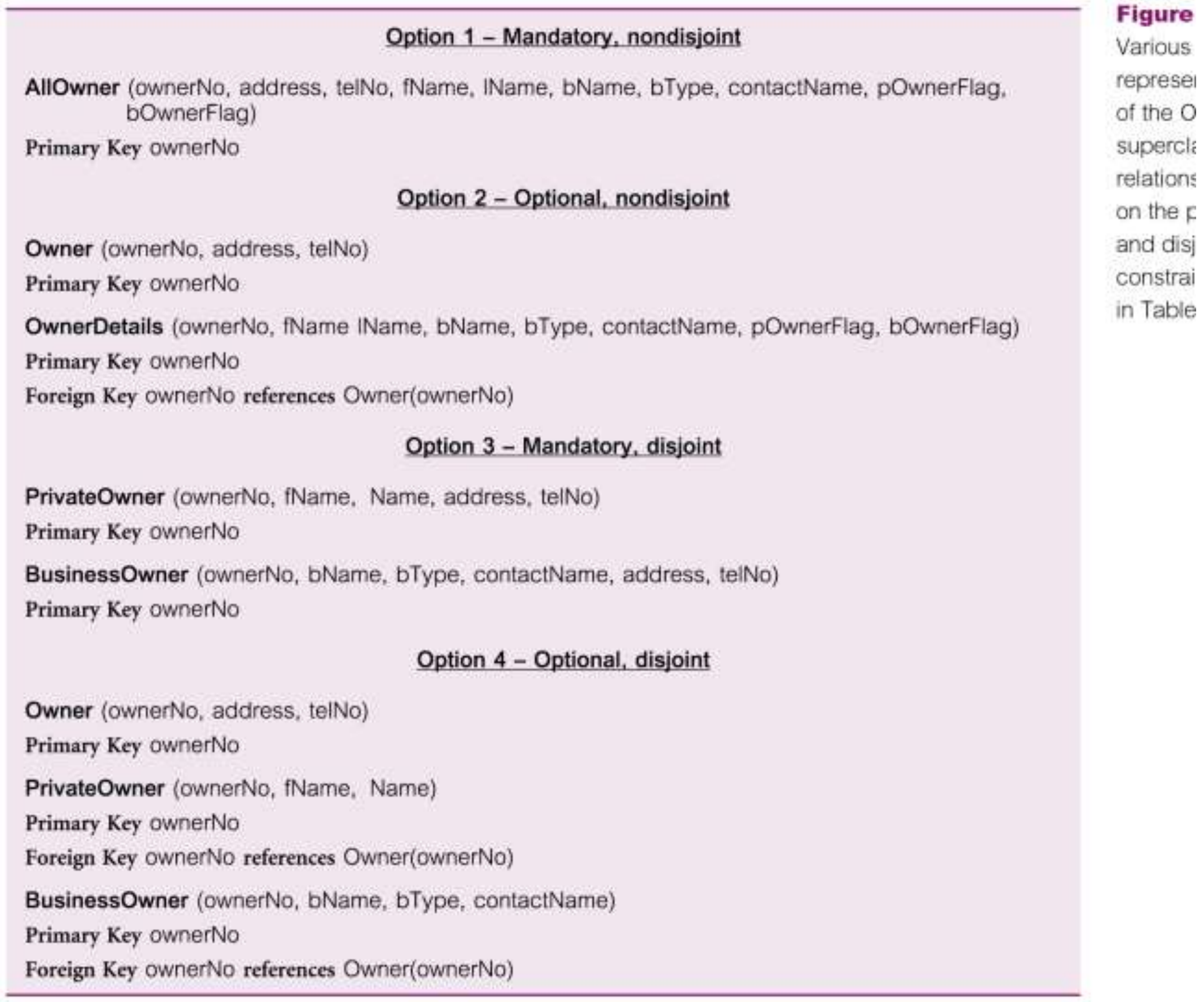
SECRETARY (<u>**SSN**</u>, TypingSpeed)

TECHNICIAN (<u>**SSN**</u>, Grade)

ENGINEER (<u>**SSN**</u>, Qualification)

More examples

(from Connolly & Begg, 2005)



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, Name, 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, Name)
Primary Key ownerNo
Foreign Key ownerNo **references** Owner(ownerNo)

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

Figure
Various represe
of the O
supercla
relations
on the p
and disj
constrai
in Table

Take-aways…

- Subtypes and supertypes can be converted to tables in a number of ways, depending on the business requirements of the system
- Generally: if it's possible to be in *more than one* subtype (overlapping), a single table is more appropriate, while separate tables are more appropriate for *disjoint* (exclusive) subtypes
- We also need to consider whether it is *mandatory* to belong to a subtype or optional
- Other features such as the number of shared vs distinct attributes and relationships with other entities also play a part

Validate model using normalisation

Validate model using normalisation {reworded]

Recall from Topic 4 that normalisation is the process of grouping attributes together because there is a logical relationship between them

- Look at conceptual data model (ERD) and verify the relations currently in 3NF are appropriate .
- Look at the schema for relations not in 3NF. Any relations not in 3NF we must correct them and update the ERD or we understand why those relations are not in 3NF and have a reason for it
- A normalised schema should resist having modification anomalies
  - 1NF: Removes repeating groups
  - 2NF: Removes partial functional dependencies
  - 3NF: Removes transitive functional dependencies

Validate model against user transactions

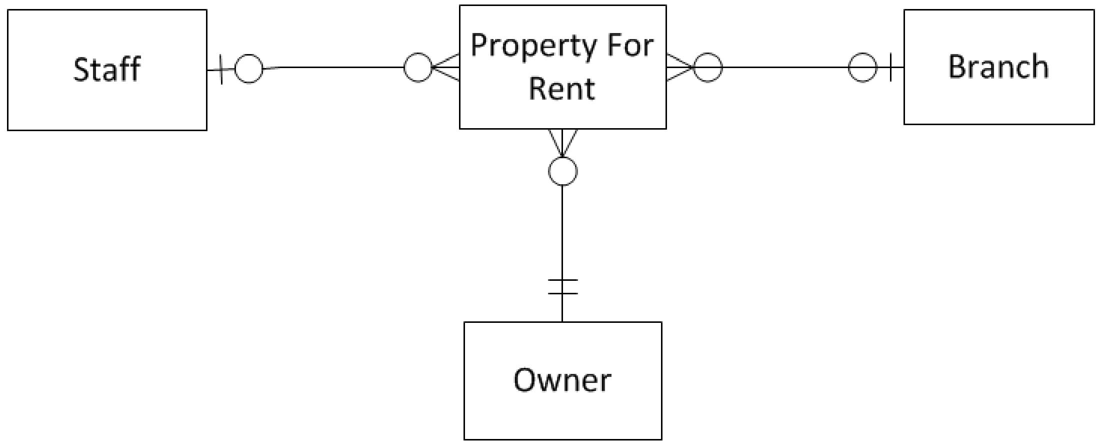
Validate model against user transactions

Verify whether the final database supports all users requested transaction.

- At all stages the database will be validated against the transaction since the transaction and database are designed in parallel
- If any problems are identified the ERD has to be amended, and the logical design rebuilt from the amended ERD

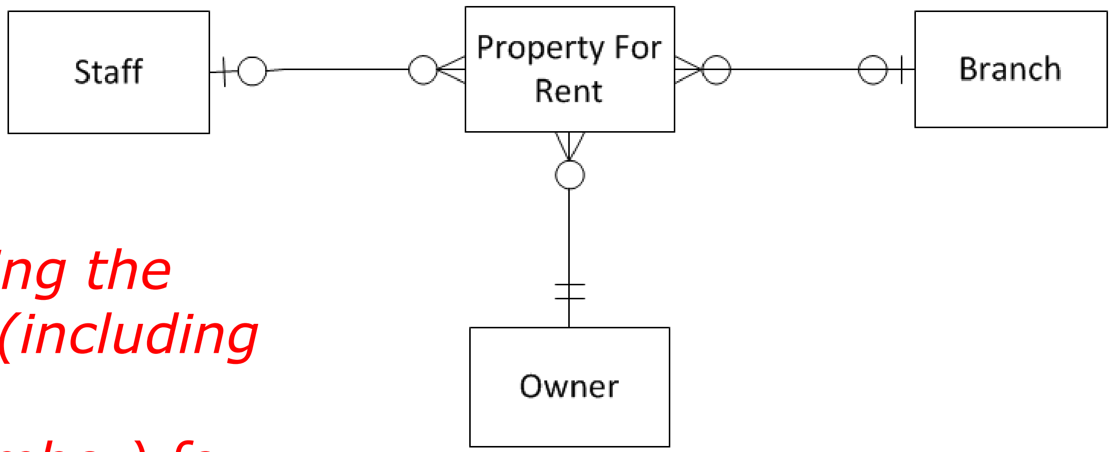
Identify any errors that arose when converting conceptual design to relational schema

Example:



How would you validate the following user transaction on this ERD? Is the transaction possible?

Produce a report listing the details of properties (including names of owner and responsible staff member) for rent at the Perth branch



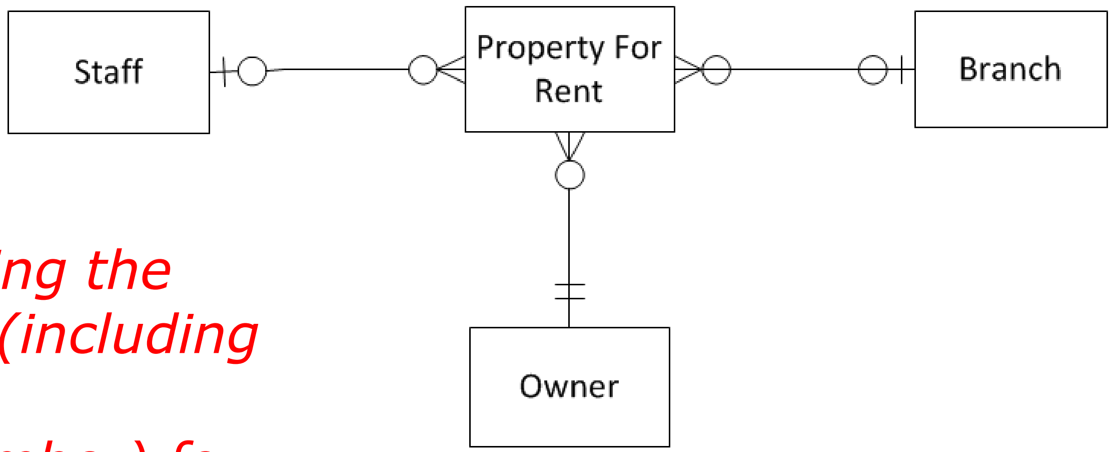
Example:

Produce a report listing the details of properties (including names of owner and responsible staff member) for rent at the Perth branch

- Starting from BRANCH, we can retrieve all the PROPERTIES FOR RENT it manages (there may be many).
- From each of those PFR, we can find one OWNER, and one STAFF member (or none, if it doesn't have one)

So yes, this transaction is possible

Example:



Produce a report listing the details of properties (including names of owner and responsible staff member) for rent at the Perth branch

STAFF (<u>StaffID, …</u>)
BRANCH (<u>BranchID, …</u>)
OWNER (<u>OwnerID</u>, …)
PFR (<u>PropertyID</u>, **StaffID**, **BranchID**, **OwnerID**, …)

- We can join on PK–FK to get the information for a particular Property

So yes, this transaction is possible

Take-aways…

- It's important to check that the logical schema supports the transactions required of the database, before going any further with the design
- This can be done in the ERD, by tracing the path of the required transactions
- Or consider whether it is possible to construct the equivalent SQL (or relational algebra) query from the schema
- If any of the transactions aren't possible, check the original design
- Also check for errors in the mapping to the logical schema – did you convert the ERD correctly?

Topic 06: Part 03 - Integrity Constraints

- Domain constraints (including Required)
- Entity integrity constraints
- Referential integrity constraints
- Enterprise constraints
- Cardinality

Define and check integrity constraints

- Integrity refers to the validity and consistency of the stored data
- Integrity is expressed in terms of constraints, which are consistency rules that the database is not permitted to violate
- In other words, in designing the database, we need to ensure that it correctly represents the system modelled, and *remains* consistent when data is added or deleted
- In the logical design stage we need to ensure **ALL** constraints are documented
- We will decide HOW to implement them later

Integrity constraints - reminder from Topic 2 …

The relational database model has a number of *constraints* that keep the data correct and consistent:

- The **domain constraint** states that the value of a particular attribute always comes from the same (specified) domain
- The **entity integrity constraint** states that the value of the primary key must be unique and not null
- The **referential integrity constraint** states that the value of a foreign key must match an existing primary key, or be null
- **Enterprise constraints** specify constraints relating to business rules that must hold true across multiple attributes or relations

Domain constraints

The **domain constraint** states each attribute within a relation must be from a single domain that limits its data to particular set of *allowable values*

- GPA must be a numeric value between 0-4
- Final grade must be one of {HD, D, C, P, N}

When we define attribute constraints for the logical design we must also specify:

- Required (**not null**) if an attribute must ALWAYS hold a value
- **Data type** (character, numeric, date, Boolean, etc)
- Any **default values** should also be specified

Entity integrity – primary key

The entity integrity constraint says that **primary key values are unique and cannot be null**

These constraints are automatically enforced when a primary key is defined a primary key – you don't have to define them separately

Choosing a primary key - some principles:

Uniqueness **must** be guaranteed

Value is not likely to change

Domain should be big enough for expansion

Key names should be recognisable, e.g. by using –ID

A primary key may be a single attribute or compound (=composite, concatenated)

e.g. JOB_HISTORY (<u>StaffID, StartDate</u>, JobNo, EndDate)

Surrogate primary keys

An artificial primary key created to simplify retrieval – e.g. if you have a very long concatenated candidate key

- Only used for implementation, usually created automatically by the DBMS

Advantages:

- Short, numeric, fixed – therefore ‘good’ primary key
- Simpler when used as foreign key in another table

Disadvantages:

- Meaningless to user
- May need further queries to retrieve ‘meaningful’ data
- May not be unique when multiple databases are merged

Enterprise constraints

Also known as **business rules**

- These are additional constraints that apply to the particular system being modelled
    - A student must have passed the prerequisite for a unit before enrolling in it
    - A student must have passed 18 points at Part 1 before enrolling in a Part 2 unit
    - End date must be later than start date
- Enterprise constraints usually have to be implemented in the application or using code in the DBMS
- Must be **documented**

Enforcing referential integrity

Referential integrity constraint

- The referential integrity constraint says that the value of a foreign key must reference an existing primary key, or be totally null
- Referential integrity can be violated in a number of ways: it is possible any time there is an:
    - **Insertion**
    - **Deletion**
    - **Update**

  of a record in the database
- So we have to document what must happen in each case in order to *preserve* referential integrity and enforce this solution

Choosing referential actions (‘foreign key rules’)

The foreign key constraint clause in the CREATE TABLE statement specifies *what should happen when changes are made in that table or the one that it references:*

- INSERT – a record in the ‘child’ table (the one with the FK)
- UPDATE, DELETE – a record in the ‘parent’ table (the one with the PK)

Violating referential integrity 1 -INSERT

Inserting a new record with a foreign key that doesn't reference a primary key in another table

- **Solution**: Disallow the insert

- If the foreign key is defined in the CREATE TABLE statement, then this will happen automatically

Violating referential integrity 2 - DELETE

Deleting a record whose primary key is referenced by records in other tables

- Solution:

- Disallow the delete

    on delete no action

- **or** Cascade the delete to delete all referencing records in the other table as well

    on delete cascade

- **or** Set foreign keys to another, allowable value (such as null or the default), then permit the delete

    on delete set null

Violating referential integrity 3 - UPDATE

Updating the primary key of a record that is referenced by records in other tables

- **Solution:**

- Disallow the update

  on update no action (NOT IN oracle)

- **or** Cascade the update to all FKs - change their values to match the new primary key

  on update cascade

- **or** Set foreign keys to another, allowable value (such as null or the default), then permit the update

  on update set null

Choosing referential actions (‘foreign key rules’)

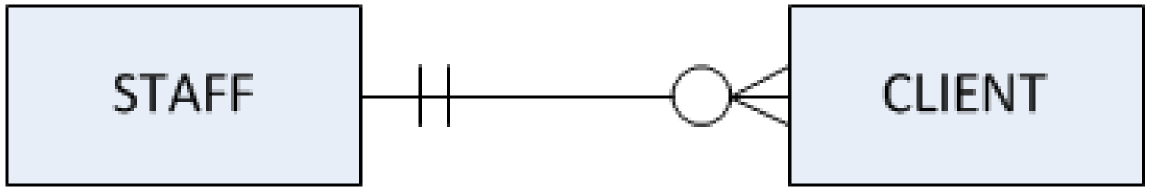
What to choose when??

The action that should be chosen to preserve referential integrity in any given case depends on the particular business context, but we can identify some rules of thumb (next)

Choosing referential actions: INSERT

Inserts to the N-side (child) table that would violate referential integrity:

- Disallow the insert, as discussed



Choosing referential actions UPDATE

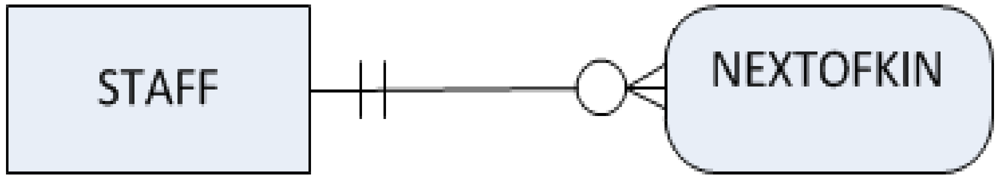
Updates to the primary key of the 1-side (parent) table:

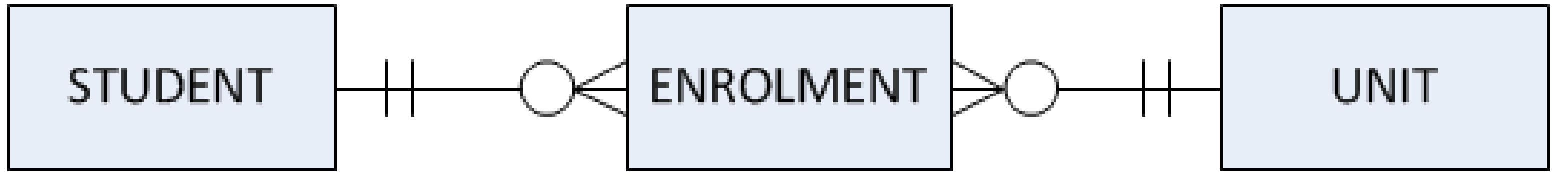
- Normally would *cascade* the update to the foreign key in the child table
- If the PK is a system-generated surrogate, would normally disallow any changes

Choosing referential actions DELETE

Deletes from the 1-side (parent) table:

- If the N-side is from a weak entity or an associative entity (existence dependent)
  - Can usually cascade the delete, unless there are business reasons for disallowing it

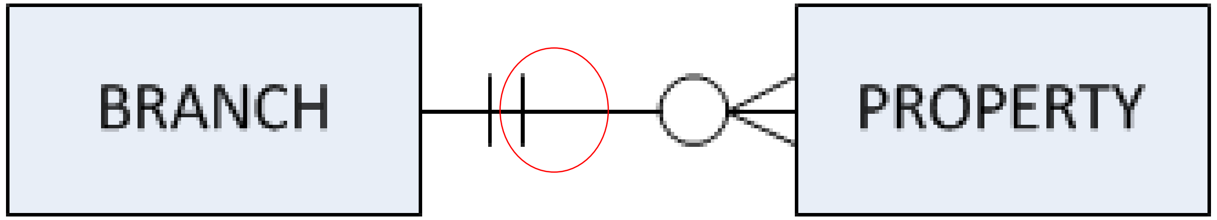




Choosing referential actions DELETE

Deletes from the 1-side (parent) table:

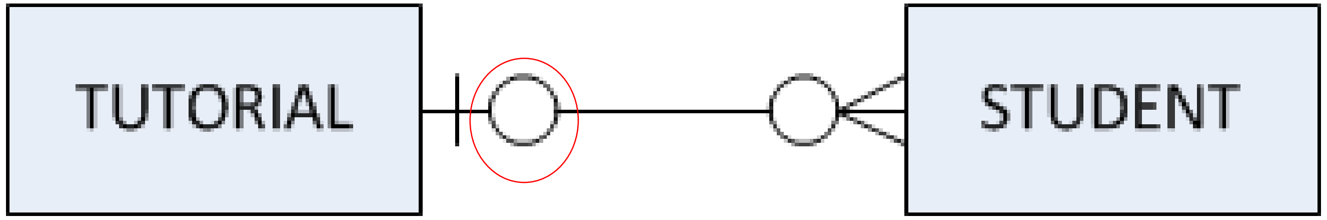
- If the 1-side is mandatory to the N-side
  - disallow delete if there are child records
  - or reassign child records to another parent then allow the delete



Choosing referential actions DELETE

Deletes from the 1-side (parent) table:

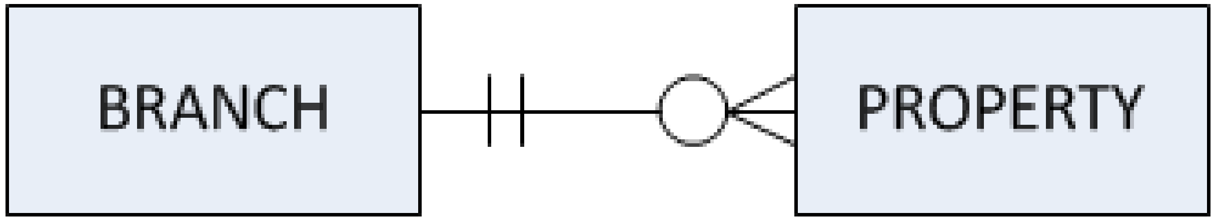
- If the 1-side is optional to the N-side
    - set foreign keys in child records to null then allow the delete
    - or reassign child records to another parent then allow the delete



Enforcing cardinality

Enforcing cardinality constraints

How do we ensure that the information about cardinality in the ERD is implemented in the database?



- We need to consider both *maximum* cardinality and *minimum* cardinality, for both entities involved in the relationship

Enforcing maximum cardinality

Maximum cardinality is either is either 1 or many, and is enforced by the foreign key:

- If **maximum cardinality is many**, all this requires is that the foreign key is placed in the 'many' side table, as usual
- If the **maximum cardinality is 1**, the foreign key is set to be unique

Enforcing minimum cardinality

The textbook goes through this in some detail, as it can become quite complex; however it can be summarised as:

If minimum cardinality is 0 no problem – nothing extra to do

If **minimum cardinality** is 1 (mandatory), then:

- If the **parent is mandatory to the child**, making the foreign key NOT NULL ensures every child record *must* have a parent
- If the **child is mandatory to the parent**, must add parent + child records in the same transaction via the application to ensure every parent has at least one child record. Must also ensure last child of a parent is not deleted
- If the **parent and child are mandatory to each other** … it gets very tricky because ‘circular’

Take-aways…

- The logical design needs to document all the **integrity constraints** required both for the system being modelled and the relational model itself:
  - Entity integrity, referential integrity, domain constraints and enterprise constraints
- We also need to specify how referential integrity is to be maintained when data in the database is changed through insert, delete, and update
  - These are known as **referential actions or** foreign key rules and must be documented
- We must also consider how to ensure minimum and maximum **cardinality** specifications

Topic 06: Part 04 - Confirm Logical Design

Review with users

Check for future growth

Merge local logical models into global model

Review logical data model with user

- ERD and schema should be complete and documented
- We need to get the model and supporting documentation and go through it with the user

Check for future growth

- It’s vital to develop a model that is extensible in case there are future changes in the requirements
  - Ask: Can we predict any potential changes that may arise
  - Our logical model must be able to deal with the potential changes
- e.g. Number of teaching periods at Murdoch has increased significantly over the years (trimesters, winter, etc)

  - what would be the best way to model teaching period??

Merge local data models to create global model

- If we have developed separate logical data models for different user views, they must be merged into a single global model
- The activities in this step include:
  - Merge local logical data models into global model
  - Validate global logical data model
  - Review global logical data model with users.

Merge local logical data models into global model

Tasks typically include:

(1) Review the names and contents of entities/relations and their candidate keys.
(2) Review the names and contents of relationships/foreign keys.
(3) Merge entities/relations from the local data models
(4) Include (without merging) entities/relations unique to each local data model
(5) Merge relationships/foreign keys from the local data models.
(6) Include (without merging) relationships/foreign keys unique to each local data model.
(7) Check for missing entities/relations and relationships/foreign keys.
(8) Check foreign keys.
(9) Check Integrity Constraints.
(10) Draw the global ER diagram
(11) Update the documentation.

Take-aways…

- Before going on, it’s important to review the logical design:
  - Confirm with users
  - Check for flexibility and future growth
  - Merge local models into a global model if required

- 6. Document the logical design in the data dictionary

Topic 06: Part 05 - Document the logical design in the data dictionary

Documenting the conceptual and logical database design process

- Both the relational schema and the ERD must have appropriate and complete documentation attached with it
- A **data dictionary** of all aspects of the system should be maintained at all stages as the design is developed
    - can be done by hand (e.g. in a series of Word tables) or using a design/development tool (e.g. ERWin; Oracle SQL Developer)
- The data dictionary should be made when you design a database and the data dictionary should be updated regularly for people who will be using the database

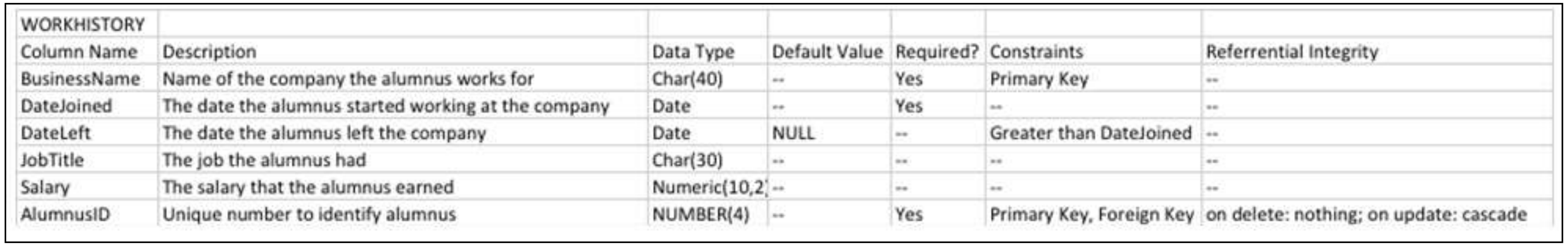
The Data Dictionary

Formats for the data dictionary vary, but should normally include the following:

- **The ERD**, including all entities, attributes, relationships, cardinality, primary keys and foreign keys (if shown)
- **Relation schemas** derived from ERD, including attributes, primary keys and foreign keys
- **Attributes** - attribute name, alias, owner table, description, data type and size, required?, derived (and how computed)? domain (allowable values), format, input mask ('picture')
- **Primary key** and any alternate key(s) for each table
- **Referential actions** (foreign key rules) for each foreign key
- Any additional constraints such as cardinality or **enterprise constraints** that have not already been documented

Data dictionary examples for attribute definitions using format in textbook

Column characteristics for the Employment table				
Column Name	**Type**	**Key**	**NULL Status**	**Remarks**
DateJoined	DATE	Primary	NOT NULL	(BETWEEN 01-JAN-1900 and 01-JAN-2999)
AlumID	NUMBER(8)	Primary, Foreign	NOT NULL	
EmployerName	VARCHAR(35)	No	NOT NULL	
JobTitle	VARCHAR(35)	No	NOT NULL	
DateLeft	DATE	No	NULL	(DateLeft > DateJoined) (BETWEEN 01-JAN-1900 and 01-JAN-2999)
Salary	NUMBER(8,2)	No	NOT NULL	(Amount > 0)



WORKHISTORY						
Column Name	Description	Data Type	Default Value	Required?	Constraints	Referential Integrity
BusinessName	Name of the company the alumnus works for	Char(40)	--	Yes	Primary Key	--
DateJoined	The date the alumnus started working at the company	Date	--	Yes	--	--
DateLeft	The date the alumnus left the company	Date	NULL	--	Greater than DateJoined	--
JobTitle	The job the alumnus had	Char(30)	--	--	--	--
Salary	The salary that the alumnus earned	Numeric(10,2	--	--	--	--
AlumnusID	Unique number to identify alumnus	NUMBER(4)	--	Yes	Primary Key, Foreign Key	on delete: nothing; on update: cascade

(student examples from last year's assignment)

Take-aways…

- A data dictionary is an essential part of documenting the logical design, and provides the basis for the subsequent physical design and implementation
- The exact format is determined by organisational practice or the software tool being used, but will include at least:
    - The complete ERD
    - The relational schema
    - Description of each table and each attribute in each table
    - Any business rules not already captured

Topic 06: Part 06 - Conclusion

Topic learning outcomes revisited

After completing this topic you should be able to:

- Describe the activities in logical database design
- Convert an ERD to a relational schema in 3NF
- Validate a relational schema against the business transactions it is required to support
- Identify and document all integrity constraints for the logical model: required data, attribute domain constraints, entity integrity, referential integrity and enterprise constraints
- Define appropriate referential actions (‘foreign key rules’) to ensure that referential integrity is maintained when the database is updated, inserted or deleted
- Document the logical database design in a data dictionary

What’s next?

Next time, we complete our set of topics on the database design process, by looking at physical design – where we define exactly how we will implement the database in the target DBMS. In the labs, we’ll return to Oracle to look in more detail at the decisions that need to be made and implemented during the physical database design phase.