Database Design

The steps to take

Critical success factors

- Work interactively with the users as much as possible.
- Follow a structured methodology throughout the data modelling process.
- Employ a data-driven approach.
- Incorporate structural and integrity considerations into the data models.
- Use a Database Design Language.
- Build a data dictionary to supplement the data model diagrams.
- Be willing to repeat steps.

Logical Database design

- Step 1
  - Create and check ER model.
    - (ER means Entity-Relationship)
- Step 2
  - Map ER model to tables.

Create and check ER model

1. Identify entities
2. Identify relationships
3. Identify and associate attributes with entities or relationships
4. Determine attribute domains
5. Determine candidate, primary, and alternate key attributes.
6. Specialize / Generalize entities (optional)
7. Check model for redundancy
8. Check model supports user transactions
9. Review model with users.

Map ER model to tables

1. Create tables
2. Check table structures using normalisation
3. Check tables support user transactions
4. Check business rules.
5. Review logical database design with users.

Identify entities

- Define the main objects that are of interest to the system.
- Examine the requirements specification and identify nouns / noun phrases that are mentioned.
  - Group nouns that are qualities of other nouns together, around that noun.
- Look for objects that exist in its own right.
- Abstract specifics to give general details.

Identify entities

- Synonyms
  - Words that have the same meaning; e.g. staff member and employee, or client and customer.
- Homonyms
  - Words that have more than one meaning; e.g. program; order date
- Document entities
  - As you find entities, give them names and estimate how many of them there'll be in the system.
  - If an entity has synonyms, define them as aliases.

Identify relationships

- Look at the requirements specification. Relationships can be indicated by verbs or verbal expressions:
  - Branch *has* staff
  - Branch *stocks* product
  - Client *places* order
- Be careful only to model relationships that are required, or you may end up with everything related to everything else.

Identifying relationships

- Most relationships are binary, but some are non-binary:
- *Supervises* represents a recursive relationship (reflexive) between staff.
- *Meets* represents a relationship between, for example, a manager, a client, a room and a time.

Identify relationships

- Determine the multiplicity of each relationship.
  - i.e. 1:1, 1:many, 0/1:1, 0/1:many, many:many
  - If specific maxima / minima are known, document them.
- A model with multiplicity constraints is a more accurate model.
- These constraints are used to check and maintain the quality of the data.
- They can be applied when the database is updated, to determine whether or not the updates follow business rules.

Checks on relationship models

- Document relationships in a grid:

Entity	Multiplicity	Relationship	Multiplicity	Entity
Customer	1..1	places	0..*	order
Staff	1..1	Issues	0..*	order
Supplier	1..1	Supplies	0..*	stock

Identify and associate attributes with entities or relationships

- Identify the types of facts about the entities and relationships, by looking for nouns or noun phrases.
- The attributes can be identified where the noun or noun phrase is a property, quality, identifier or characteristic of one of the entities or relationships that you've previously found.

Attributes

- Simple / composite attributes:
  - A composite attribute is one that is made up of simple attributes;
    - e.g. address can be a string, or a composite of street, area, city, postcode.
  - Attributes should only be composite if their individual attributes are required.
- Single / multi-value attributes:
  - If an attribute contains multiple values, it should be separated out into another table.
- Derived attributes
  - Do not eliminate derived attributes until all components required to derive them are available.

Attributes

- Check through the requirements specification and make a list of attributes.
- As you allocate them to entities, eliminate them from the list.
- If an attribute appears in several entities:
  - You may have identified several entities that are all really the same entity
  - You may have identified a relationship between entities.
- Attributes of relationships
  - A user logs on to a lab computer, using an account name and password, at a time.

Documenting attributes

- Record the following:
  - Attribute name and description
  - Data type and length
  - Any aliases of the attribute
  - Whether the attribute must always be specified.
  - Whether the attribute is multi-valued
  - Whether it is composite
  - Whether it is derived and how
  - Default values for the attribute.

Attribute grid

Entity	Attributes	Description	Data type and length	Nulls	Multi-valued
Customer	Customer name	Name of customer	Up to 40 characters	No	No
	Customer Address	Customer address	Up to 80 characters	No	No
	Telephone no	Telephone number with international code	Up to 12 characters	Yes	Yes

Determine attribute domains

- The attribute domain is a pool of values from which the attribute draws its value.
  - A valid age for a school-child is between 4 and 19.
  - A valid value for AliensExist is TRUE or FALSE.
  - A valid student number is a letter followed by 8 digits.
  - Etc.
- As you identify attribute domains:
  - record their names and characteristics in the data dictionary.
  - Update the data dictionary entries for attributes to record their domain, in place of the datatype.

Determine candidate, primary, and alternate key attributes.

- To choose the primary key, pick the candidate key that
  - has the minimal set of attributes.
  - Is less likely to have its value changed.
  - Is less likely to lose uniqueness in the future.
  - With
    - fewest characters (character type).
    - the smallest maximum value (numeric type)
  - That is easiest to use from the user's point of view.

Note whether an entity is strong or weak

- Strong entity is one to which you can identify a primary key.
- A weak entity is one that is identified by another entity:
  - E.g. orderline.
- Not all entities (even strong ones) have an appropriate primary key.
  - Often the only identification can be a name, that may not be unique.
  - In this case, a key needs to be introduced.

Specialize / Generalize entities (optional)

- This step can be necessary when there are a number of alterations to an entity:
- E.g. a transaction could be:
  - Transaction type
  - Account number
  - Amount.
- This works for deposits and withdrawals.
- For credit transfers, extra information is required: the destination account.
  - This is a specialised version of the transaction entity.

Check model for redundancy

- Re-examine 1:1 relationships
- Remove redundant relationships
- Consider the time dimension when assessing redundancy.
  - E.g. an order may eventually become an invoice.

Check model supports user transactions

- Describe each transaction
  - This will give you an idea of what data you need to add / amend / delete / select.
- Use transaction pathways
  - Trace the relationship pathways that you need for the transaction onto the ER diagram.

Review model with users

- Go back to the user and review
  - the ER model.
    - The data dictionary
    - The ER diagram(s)
    - Any additional documentation.
- Repeat the previous steps until the user is happy.

Extended Entity Relationship (EER) Models

- The EER model includes all the modeling concepts of the ER model.
- It also includes the concepts of
  - subclass and superclass;
  - specialisation and generalisation.
  - Inheritance of attributes and relationship.

Constraints on Specialisation and Generalisation

- Predicate (or condition) defined
  - In some specialisations the entities that will become members of each subclass can be determined by placing a condition on the value of some attribute of the superclass.
    - E.g. an attribute called job-type could be the defining predicate.
  - This is a constraint describing exactly which members are in the subclass.
- Attribute defined specialisation
  - All subclasses have their membership defined on the same attribute of the superclass.
  - The attribute is called the defining attribute.
- User-defined
  - There is no condition for determining membership in a subclass.
  - Membership is determined by the database users when they apply the operation to add an entity to the subclass.
  - Membership is specified individually for each entity by the user.

Other constraints

- Disjointedness constraint
  - Specifies that all subclasses of the specialisation should be disjoint.
  - i.e. the entity can be a member of at most one of the subclasses of the specialisation.
  - A specialisation that is attribute-defined implies disjointedness.
- Completeness constraint
  - Total specialisation specifies that every entity in the superclass must be a member of at least one subclass.
  - Partial specialisation allows an entity not to belong to any of the subclasses.

Disjointedness and completeness

- These are independent constraints.
- There are four combinations:
  - Disjoint, total
  - Disjoint, partial
  - Overlapping, total
  - Overlapping, partial

Insertion and Deletion

- Deleting an entity from a superclass implies that it is automatically deleted from all the subclasses to which it belongs.
- Inserting an entity into a superclass
  - implies that the entity is mandatorily inserted in all predicate-defined subclasses for which the entity satisfies the defining predicate.
  - Of a total specialisation implies that the entity is mandatorily inserted into at lease one subclass of the specialisation.

Multiple Inheritance

- A subclass with more than one superclass is called a shared subclass.
- The subclass inherits from > 1 superclass.

Specialisation Hierarchy

- A subclass may have further subclasses specified on it, forming a hierarchy or lattice of specialisations.
- Hierarchy
  - Every subclass participates as a subclass in only one class/subclass relationship.
- Lattice
  - A subclass can be a subclass in more than one class / subclass relationship.

Categories

- If a single superclass / subclass relationship has more than one superclass, the superclasses may represent different entity types.
- The subclass represents a collection of objects that is a subset of the UNION of distinct entity types.
- This subclass is called a **union type** or **category**.

Guidelines for specialisation / generalisation

- Many specialisations and subclasses can be defined to make the model accurate, but they can clutter it.
- If a subclass has few specific attributes and no specific relationships, it should be merged into the superclass, with tuples without subclasses having nulls for those attributes.
- If all subclasses of a generalisation have few specific attributes and no specific relationships, they can be merged into the superclass and replaced with one or more *type* attributes that specify the subclass to which they belong.
- Union types and categories should be avoided unless absolutely necessary.
- The choice of disjoint / overlapping and total / partial constraints on specialisation / generalisation is driven by the rules in the domain being modelled.
  - If the requirements don't indicate constraints, the default is overlapping and partial.

Mapping EER Model constructs to relations

- There are four ways of mapping specialisation or generalisation.
  - Multiple relations – superclass and subclasses
  - Multiple relations – subclass relations only
  - Single relation with one type attribute
  - Single relation with multiple type attributes.

In discussion:

- Attrs(R) means attributes of Relation R.
- PK(R) means primary key of R.
- We want to convert each specialisation with *m* subclasses $\{S_1, S_2, \ldots, S_n\}$ and generalised superclass C, where the attributes of C are $\{k, a_1, a_2, \ldots, a_n\}$ and k is the primary key.

Multiple relations – superclass and subclasses

- Create a relation L for C with attributes Attrs(L) = {k, a_1, a_2,…, a_n} and PK(L)=k.
- Create a relation L_i for each subclass S_i, 1≤i≤m, with the attributes Attrs(L_i)={k}u{attributes of S_i } and PK(L_i) = k.
- This option works for any specialisation.

Multiple relations – subclass relations only

- Create a relation L_i for each subclass S_i, $1 \leq i \leq m$, with the attributes $Attrs(L_i) = \{k, a_1, a_2, \ldots, a_n\} \cup \{\text{attributes of } S_i\}$ and $PK(L_i) = k$.
- This option only works for a specialisation whose subclasses are *total*
  - (every entity in the superclass must belong to at least one of the subclasses).
  - If the specialisation is *overlapping* an entity may be duplicated in several relations.

Multiple relations – superclass and subclasses

- Create a relation L for C with attributes Attrs(L) = {k, a_1, a_2,…, a_n} and PK(L)=k.
- Create a relation L_i for each subclass S_i, 1≤i≤m, with the attributes Attrs(L_i)={k}u{attributes of S_i } and PK(L_i) = k.
- This option works for any specialisation.

Single Relation with one type attribute

- Create a single relation schema L with attributes Attrs(L) =
- {k, a_1, a_2,…, a_n} u{attributes of S_i }u…u {attributes of S_m } u{t} and PK(L_i) = k.
- t is a type (or discriminating) attribute that indicates the subclass to which each tuple belongs, if any.
- This works only for a specialisation whose subclasses are disjoint, and has the potential for generating many NULL values if many specific attributes exist in the subclasses.

Single relation with multiple type attributes

- Create a single relation schema L with attributes Attrs(L) =
- {k, a_1, a_2,…, a_n} u{attributes of S_i }u…u {attributes of S_m } u{t_1, t_2,…, t_n} and PK(L_i) = k.
- Each t_i, $1 \leq i \leq m$, is a Boolean type attribute that indicates whether or not the tuple belongs to the subclass S_i.
- This works only for a specialisation whose subclasses are overlapping, but will also work for a disjoint specialisation.