Mater Dei College
Tubigon, Bohol
2nd Semester, A.Y. 2022-2023
CC 105 – IM 101 (INFORMATION MANAGEMENT 1)

TOPIC 6
Database Design Phase 2- Conceptual, Logical, Physical Design

Database Design Phase 2: Conceptual Design

The design phase is where the requirements identified in the previous phase are used as the basis to develop the new system. Another way of putting it is that the business understanding of the data structures is converted to a technical understanding. The *what* questions ("What data are required? What are the problems to be solved?") are replaced by the *how* questions ("How will the data be structured? How is the data to be accessed?")

This phase consists of three parts: the conceptual design, the logical design and the physical design. Some methodologies merge the logical design phase into the other two phases.

Conceptual design

The purpose of the conceptual design phase is to build a conceptual model based upon the previously identified requirements, but closer to the final physical model. A commonly-used conceptual model is called an *entity-relationship* model.

Entities and attributes

Entities are basically people, places, or things you want to keep information about. For example, a library system may have the *book*, *library* and *borrower* entities. Learning to identify what should be an entity, what should be a number of entities, and what should be an *attribute* of an entity takes practice, but there are some good rules of thumb. The following questions can help to identify whether something is an entity:

- Can it vary in number independently of other entities? For example, *person height* is probably not an entity, as it cannot vary in number independently of *person*. It is not fundamental, so it cannot be an entity in this case.
- Is it important enough to warrant the effort of maintaining. For example *customer* may not be important for a small grocery store and will not be an entity in that case, but it will be important for a video store, and will be an entity in that case.
- Is it its own thing that cannot be separated into subcategories? For example, a car-rental agency may have different criteria and storage requirements for different kinds of vehicles. *Vehicle* may not be an entity, as it can be broken up into *car* and *boat*, which are the entities.
- Does it list a type of thing, not an instance? The video game *blow-em-up 6* is not an entity, rather an instance of the *game* entity.
- Does it have many associated facts? If it only contains one attribute, it is unlikely to be an entity. For example, *city* may be an entity in some cases, but if it contains only one attribute, *city name*, it is more likely to be an attribute of another entity, such as *customer*.

The following are examples of entities involving a university with possible attributes in parentheses.

- **Course** (name, code, course prerequisites)
- **Student** (first_name, surname, address, age)
- **Book** (title, ISBN, price, quantity in stock)

An instance of an entity is one particular occurrence of that entity. For example, the student Rudolf Sono is one instance of the student entity. There will probably be many instances. If there is only one instance, consider whether the entity is warranted. The top level usually does not warrant an entity. For example, if the system is being developed for a particular university, *university* will not be an entity because the whole system is for that one university. However, if the system was developed to track legislation at all universities in the country, then *university* would be a valid entity.

Relationships

Entities are related in certain ways. For example, a borrower may belong to a library and can take out books. A book can be found in a particular library. Understanding what you are storing data about, and how the data relate, leads you a large part of the way to a physical implementation in the database.
There are a number of possible relationships:

Mandatory

For each instance of entity A, there must exist one or more instances of entity B. This does not necessarily mean that for each instance of entity B, there must exist one or more instances of entity A. Relationships are optional or mandatory in one direction only, so the A-to-B relationship can be optional, while the B-to-A relationship is mandatory.

Optional

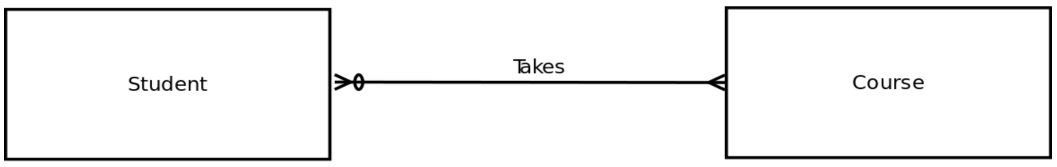
For each instance of entity A, there may or may not exist instances of entity B.

One-to-one (1:1)
This is where for each instance of entity A, there exists one instance of entity B, and vice-versa. If the relationship is optional, there can exist zero or one instances, and if the relationship is mandatory, there exists one and only one instance of the associated entity.

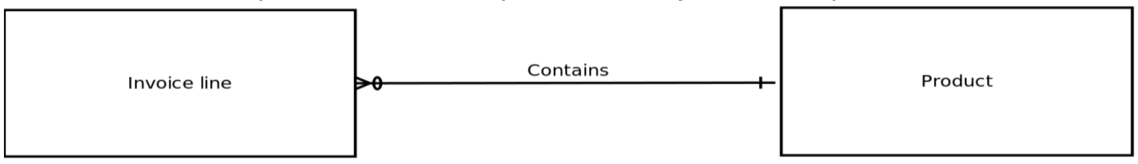
One-to-many (1:M)
For each instance of entity A, many instances of entity B can exist, which for each instance of entity B, only one instance of entity A exists. Again, these can be optional or mandatory relationships.

Many-to-many (M:N)
For each instance of entity A, many instances of entity B can exist, and vice versa. These can be optional or mandatory relationships.

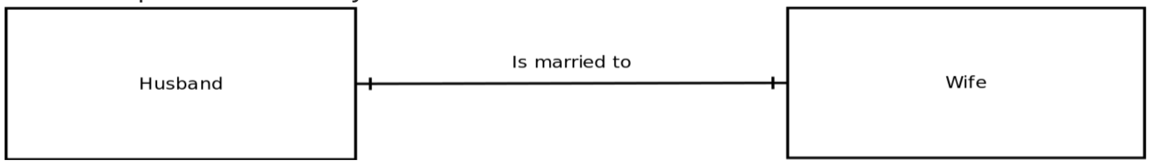
There are numerous ways of showing these relationships. The image below shows *student* and *course* entities. In this case, each student must have registered for at least one course, but a course does not necessarily have to have students registered. The student-to-course relationship is mandatory, and the course-to-student relationship is optional.



The image below shows *invoice_line* and *product* entities. Each invoice line must have at least one product (but no more than one); however each product can appear on many invoice lines, or none at all. The *invoice_line-to-product* relationship is mandatory, while the *product-to-invoice_line* relationship is optional.



The figure below shows husband and wife entities. In this system (others are of course possible), each husband must have one and only one wife, and each wife must have one, and only one, husband. Both relationships are mandatory.



An entity can also have a relationship with itself. Such an entity is called a *recursive entity*. Take a *person* entity. If you're interested in storing data about which people are brothers, you wlll have an "is brother to" relationship. In this case, the relationship is an M:N relationship.

Conversely, a *weak entity* is an entity that cannot exist without another entity. For example, in a school, the *scholar* entity is related to the weak entity *parent/guardian*. Without the scholar, the parent or guardian cannot exist in the system. Weak entities usually derive their primary key, in part or in totality, from the associated entity. *parent/guardian* could take the primary key from the scholar table as part of its primary key (or the entire key if the system only stored one parent/guardian per scholar).

The term *connectivity* refers to the relationship classification.

The term *cardinality* refers to the specific number of instances possible for a relationship. *Cardinality limits* list the minimum and maximum possible occurrences of the associated entity. In the husband and wife example, the cardinality limit is (1,1), and in the case of a student who can take between one and eight courses, the cardinality limits would be represented as (1,8).

Developing an entity-relationship diagram
An entity-relationship diagram models how the entities relate to each other. It's made up of multiple relationships, the kind shown in the examples above. In general, these entities go on to become the database tables.

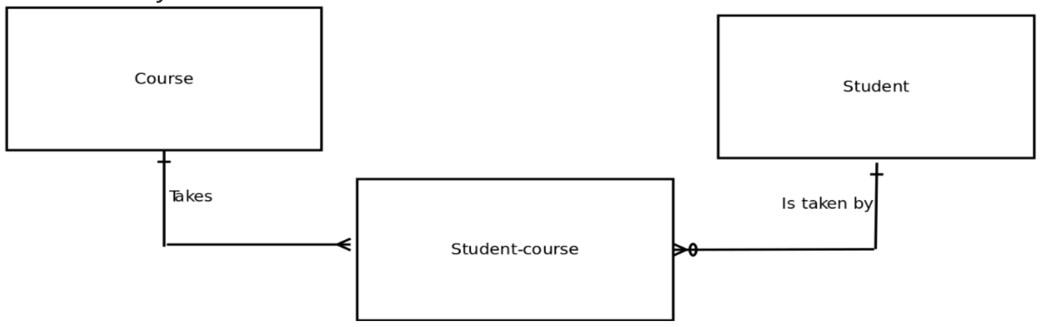
The first step in developing the diagram is to identify all the entities in the system. In the initial stage, it is not necessary to identify the attributes, but this may help to clarify matters if the designer is unsure about some of the entities. Once the entities are listed, relationships between these entities are identified and modeled according to their type: one-to-many, optional and so on. There are many software packages that can assist in drawing an entity-relationship diagram, but any graphical package should suffice.

Once the initial entity-relationship diagram has been drawn, it is often shown to the stakeholders. Entity-relationship diagrams are easy for non-technical people to understand, especially when guided through the process. This can help identify any errors that have crept in. Part of the reason for modeling is that models are much easier to understand than pages of text, and they are much more likely to be viewed by stakeholders, which reduces the chances of errors slipping through to the next stage, when they may be more difficult to fix.

It is important to remember that there is no one right or wrong answer. The more complex the situation, the more possible designs that will work. Database design is an acquired skill, though, and more experienced designers will have a good idea of what works and of possible problems at a later stage, having gone through the process before.

Once the diagram has been approved, the next stage is to replace many-to-many relationships with two one-to-many relationships. A DBMS cannot directly implement many-to-many relationships, so they are decomposed into two smaller relationships. To achieve this, you have to create an *intersection*, or *composite* entity type. Because intersection entities are less "real-world" than ordinary entities, they are sometimes difficult to name. In this case, you can name them according to the two entities being intersected.

For example, you can intersect the many-to-many relationship between *student* and *course* by a *student-course* entity.



The same applies even if the entity is recursive. The person entity that has an M:N relationship "is brother to" also needs an intersection entity. You can come up with a good name for the intersection entity in this case: *brother*. This entity would contain two fields, one for each person of the brother relationship — in other words, the primary key of the first brother and the primary key of the other brother.

Database Design Example Phase 2: Design

Based on the provided information, you can begin your logical design and should be able to identify the initial entities:

- Poet
- Poem
- Publication
- Sale
- Customer

The Poet's Circle is not an entity, or even of instance an a *publisher* entity. Only if the system were developed for many publishers would *publisher* be a valid entity.

Neither *website* nor *poetry community* are entities. There is only one website, and anyway, a website is merely a means of producing data to populate the database. There is also only one poetry community as far as this system is concerned, and there is not much you'd want to store about it.

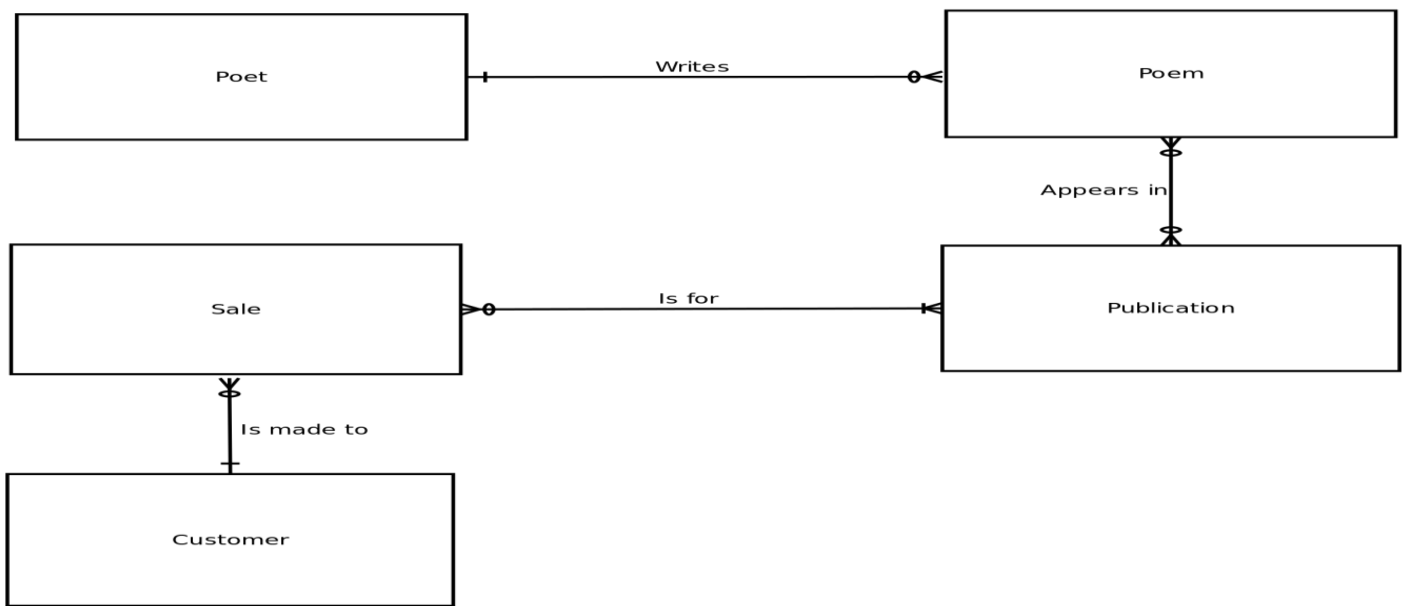
Next, you need to determine the relationship between these entities. You can identify the following:

- A poet can write many poems. The analysis identified the fact that a poet can be stored in the system even if there are no associated poems. Poems may be captured at a later point in time, or the poet may still be a potential poet. Conversely, many poets could conceivably write a poem, though the poem must have been written by at least one poet.
- A publication may contain many poems (an anthology) or just one. It can also contain no poems (poetry criticism for example). A poem may or may not appear in a publication.
- A sale must be for at least one publication, but it may be for many. A publication may or may not have made any sales.
- A customer may be made for many sales, or none at all. A sale is only made for one and only one customer.

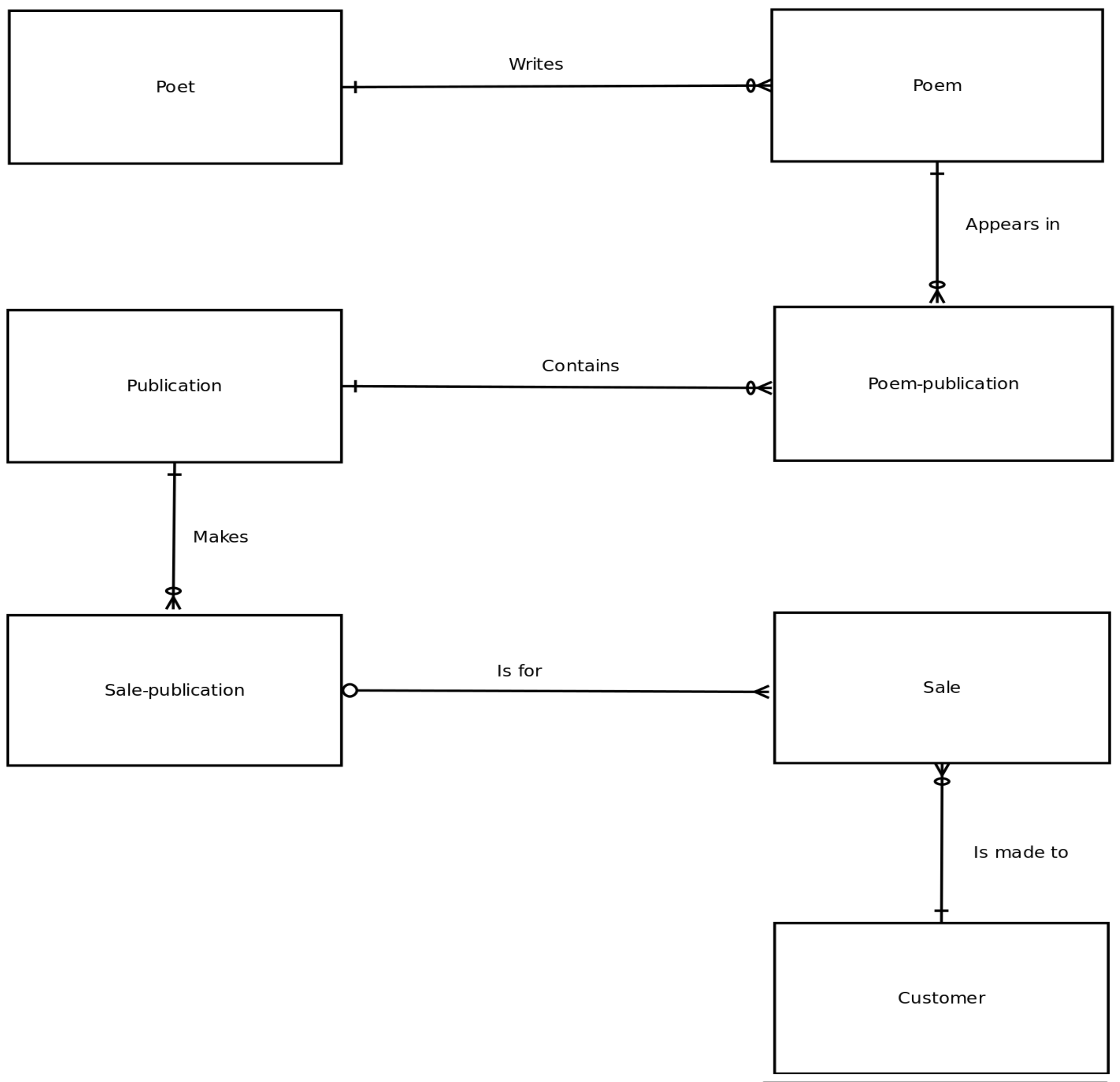
You can identify the following attributes:

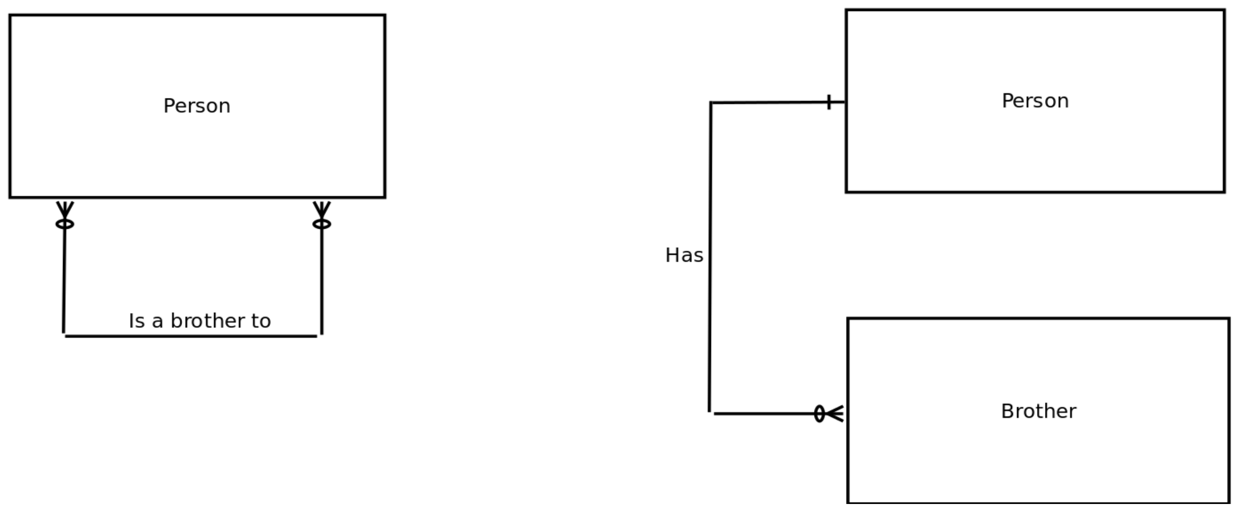
- **Poet**: first name, surname, address, email address
- **Poem**: poem title, poem contents
- **Publication**: title, price
- **Sales**: date, amount
- **Customer**: first name, surname, address, email address

Based on these entities and relationships, you can construct the entity-relationship diagram shown below:



There are two many-to-many relationships in the figure above. These need to be converted into one-to-many relationships before you can implement them in a DBMS. After doing so, the intersection entities *poem-publication* and *sale-publication* are created.





Database Design Phase 2: Logical and Physical Design

Overview

Once the conceptual design is finalized, it's time to convert this to the logical and physical design. Usually, the DBMS is chosen at this stage, depending on the requirements and complexity of the data structures. Strictly speaking, the logical design and the physical design are two separate stages, but are often merged into one. They overlap because most current DBMSs (including MariaDB) match logical records to physical records on disk on a 1:1 basis.

Each entity will become a database table, and each attribute will become a field of this table. Foreign keys can be created if the DBMS supports them and the designer decides to implement them. If the relationship is mandatory, the foreign key must be defined as *NOT NULL*, and if it's optional, the foreign key can allow nulls. For example, because of the invoice line-to-product relationship in the previous example, the product code field is a foreign key in the invoice to line table. Because the invoice line must contain a product, the field must be defined as *NOT NULL*. The default MariaDB storage engine, XtraDB, does support foreign key constraints, but some storage engines, such as MyISAM do not. The *ON DELETE CASCADE* and *ON DELETE RESTRICT* clauses are used to support foreign keys. *ON DELETE RESTRICT* means that records cannot be deleted unless all records associated with the foreign key are also deleted. In the invoice line-to-product case, *ON DELETE RESTRICT* in the invoice line table means that if a product is deleted, the deletion will not take place unless all associated invoice lines with that product are deleted as well. This avoids the possibility of an invoice line existing that points to a non-existent product. *ON DELETE CASCADE* achieves a similar effect, but more automatically (and more dangerously!). If the foreign key was declared with *ON CASCADE DELETE*, associated invoice lines would automatically be deleted if a product was deleted. *ON UPDATE CASCADE* is similar to *ON DELETE CASCADE* in that all foreign key references to a primary key are updated when the primary key is updated.

Normalizing your tables is an important step when designing the database. This process helps avoid data redundancy and improves your data integrity.

Novice database designers usually make a number of common errors. If you've carefully identified entities and attributes and you've normalized your data, you'll probably avoid these errors.

Common errors

- Keep unrelated data in different tables. People who are used to using spreadsheets often make this mistake because they are used to seeing all their data in one two-dimensional table. A relational database is much more powerful;

- Don't store values you can calculate. Let's say you're interested three numbers: /A, B and the product of A and B (A*B). Don't store the product. It wastes space and can easily be calculated if you need it. And it makes your database more difficult to maintain: If you change A, you also have to change all of the products as well. Why waste your database's efforts on something you can calculate when you need it?

- Does your design cater to all the conditions you've analyzed? In the heady rush of creating an entity-relationship diagram, you can easily overlook a condition. Entity-relationship diagrams are usually better at getting stakeholders to spot an incorrect rule than spot a missing one. The business logic is as important as the database logic and is more likely to be overlooked. For example, it's easy to spot that you cannot have a sale without an associated customer, but have you built in that the customer cannot be approved for a sale of less than $500 if another approved customer has not recommended them?

- Are your attributes, which are about to become field names, well chosen? Fields should be clearly named. For example, if you use *f1* and *f2* instead of *surname* and *first_name*, the time saved in less typing will be lost in looking up the correct spelling of the field, or in mistakes where a developer thought *f1* was the first name, and *f2* the surname. Similarly, try to avoid the same names for different fields. If six tables have a primary key of *code*, you're making life unnecessarily difficult. Rather, use more descriptive terms, such as *sales_code* or *customer_code*.

- Don't create too many relationships. Almost every table in a system can be related by some stretch of the imagination, but there's no need to do this. For example, a tennis player belongs to a sports club. A sports club belongs to a region. The tennis players then also belong to a region, but this relationship can be derived through the sports club, so there's no need to add another foreign key (except to achieve performance benefits for certain kinds of queries). Normalizing can help you avoid this sort of problem (and even when you're trying to optimize for speed, it's usually better to normalize and then consciously denormalize rather than not normalize at all).

- Conversely, have you catered to all relations? Do all relations from your entity-relationship diagram appear as common fields in your table structures? Have you covered all relations? Are all many-to-many relationships broken up into two one-to-many relationships, with an intersection entity?

- Have you listed all constraints? Constraints include a gender that can only be *m* or *f*, ages of schoolchildren that cannot exceed twenty, or email addresses that need to have an @ sign and at least one period (.; don't take these limits for granted. At some stage the system you will need to implement them, and you're either going to forget to do so, or have to go back and gather more data if you don't list these up front.

- Are you planning to store too much data? Should a customer be asked to supply their eye color, favorite kind of fish, and names of their grandparents if they are simply trying to register for an online newsletter? Sometimes stakeholders want too much information from their customers. If the user is outside the organization, they may not have a voice in the design process, but they should always be thought of foremost. Consider also the difficulty and time taken to capture all the data. If a telephone operator needs to take all this information down before making a sale, imagine how much slower they will be. Also consider the impact data has on database speed. Larger tables are generally slower to access, and unnecessary BLOB, TEXT and VARCHAR fields lead to record and table fragmentation.
- Have you combined fields that should be separate? Combining first name and surname into one field is a common beginner mistake. Later you'll realise that sorting names alphabetically is tricky if you've stored them as *John Ellis* and *Alfred Ntombela*. Keep distinct data discrete.

- Has every table gotten a primary key? There had better be a good reason for leaving out a primary key. How else are you going to identify a unique record quickly? Consider that an index speeds up access time tremendously, and when kept small it adds very little overhead. Also, it's usually better to create a new field for the primary key rather than take existing fields. First name and surname may be unique in your current database, but they may not always be. Creating a system-defined primary key ensures it will always be unique.

- Give some thought to your other indexes. What fields are likely to be used in this condition to access the table? You can always create more fields later when you test the system, but add any you think you need at this stage.
- Are your foreign keys correctly placed? In a one-to-many relationship, the foreign key appears in the *many* table, and the associated primary key in the *one* table. Mixing these up can cause errors.
- Do you ensure referential integrity? Foreign keys should not relate to a primary key in another table that no longer exists.

- Have you covered all character sets you may need? German letters, for example, have an expanded character set, and if the database is to cater for German users it will have to take this into account. Similarly, dates and currency formats should be carefully considered if the system is to be international
- Is your security sufficient? Remember to assign the minimum permissions you can. Do not allow anyone to view a table if they do not need to do so. Allowing malicious users view data, even if they cannot change it, is often the first step in for an attacker.
- Now, to begin the logical and physical design, you need to add attributes that can create the relationship between the entities and specify primary keys. You do what's usually best, and create new, unique, primary keys. The following tables show the structures for the tables created from each of the entities:

- ***Poet table***

Field	Definition
poet code	primary key, integer
first name	character (30)
surname	character (40)
address	character (100)
postcode	character (20)
email address	character (254)

Poem table

Field	Definition
poem code	primary key, integer
poem title	character(50)
poem contents	text
poet code	foreign key, integer

- ***Poem-publication table***

Field	Definition
poem code	joint primary key, foreign key, integer
publication code	joint primary key, foreign key, integer

Publication table

Field	Definition
publication code	primary key, integer
title	character(100)
price	numeric(5.2)

- ***Sale-publication table***

Field	Definition
sale code	joint primary key, foreign key, integer
publication code	joint primary key, foreign key, integer

Sale table

Field	Definition
sale code	primary key, integer
date	date
amount	numeric(10.2)
customer code	foreign key, integer

- ***Customer table***

Field	Definition
customer code	primary key, integer
first name	character (30)
surname	character (40)
address	character (100)
postcode	character (20)
email address	character (254)

- MariaDB will have no problem with this, and is selected as the DBMS. Existing hardware and operating system platforms are also selected. The following section looks at the implementation and the SQL statements used to create the MariaDB tables.

SOURCE: https://mariadb.com/kb/en/database-design-phase-2-conceptual-design/
https://mariadb.com/kb/en/database-design-phase-2-logical-and-physical-design/
https://mariadb.com/kb/en/database-design-example-phase-2-design/