# <Library>

## Milestone 1: Requirements Analysis & Conceptual Design

Purpose:

This database project is based around a fictional book library that is both multi-lingual and multi-modal in that it provides books in physical, digital, and audio formats while also providing translations of those books in different languages. To that end, the project strives to imitate only the most crucial properties of a real-life library, such as book categorisation and user identification as well as online browsing and lending functionalities. Emphasis is placed on intricate searching capabilities and problem-free borrowing, rather than, let's say, delivery, procurement, or employee management. The backbone of the library database management system is to be built up to accommodate a multitude of different higher-level features (e.g. email reminders, recommendation algorithms, etc.) but no extra care is to be taken to incorporate those features as they go beyond the scope of the project.

Data Requirements:

The database should capture information on **item mediums** as well as **item genres** each of which should be used to enrich the information that is stored on each **library item**. This should be done in such a way to ensure that every library item corresponds to one of several mediums at most, while simultaneously belonging to one, many, or no genres.

For the sake of simplicity, only library items (and eventual republications or additional editions of those items) should be given unique **ISBN**s and should be entered into the central **library item** relation, while translations and narrations of those items are only to be assigned their corresponding language and narrators, respectively, and to be added to separate child relations of the **library item** relation. This is intended to simplify data entry and to avoid overloading one relation with many entries while leaving others largely empty. This, in end effect, should make it so that only a handful of unique library items can correspond to a multitude of unique translations and narrations of the same items; translations and narrations are then to be uniquely identified using their language and narrators, respectively, in combination with the ISBN codes of the books they correspond to. This simplification should in no way impact the user experience.

In line with the project's purpose, every library item should also possess the capability of existing in multiple formats, outside of its **library item** relation, and namely, in digital and physical form. This, combined with the data described in the previous paragraph, enables each library item to exist in a variety of additional states such as physical and/or digital and/or audio, each of which can be further offered in a variety of languages. All this is done without severely overloading the **library item** relation by making use of cross-referencing and by delegating this data to the item's child relations, one representing the **physical** and/or **digital** format, the other, the **audiobook** format. Furthermore, both child relations are to allow their entries to exist in multiple **languages**. This implies that only **physical** copies of books are to be assigned a **section** within a **library**, further implying that only physical copies are to be **borrowed**, while audio and digital should be available for download.

To accommodate physical storage of books, data is also to be stored on each **library**, things such as **address** and **name**, while keeping the data of each **section** of each library separately, so as to allow for one library to be made up of multiple sections, any one of which can share its name with the section of another library but without losing the ability for unique identification in the process.

Apart from that, again, for the sake of simplicity, the **authors** of each item are to be connected to that **library item**, rather than to its children, eliminating the need to store the names of translators of books, instead simply connecting any translation to its original author. It should also be made possible to store data on which **author**, if any, **influenced** another; this could potentially be used to enable user recommendations based on the authors whose books a user has been known to borrow.

Lastly, minimal amounts of data are to be stored about **users**, only those needed for their unique identification and mutual cross referencing, which would include the user's **name,** unique **email address**, and **password** – for purposes of registration. The library should give its users the ability to **borrow** and **return** books, while ensuring that no physical copy of a book can be borrowed by two users at any one time. A time window for borrowing should also be enforced for each borrow, so as to prevent users from taking any other books in case they haven't returned their overdue books. Finally, a user should have the freedom to **delete** their data from the database, but only after they have returned all their books.

Notes:

- A user should be allowed to borrow the same book (uniquely identified by its *language* and *ISBN*) multiple times, but not at the same time, in other words, a user cannot hold two copies of the same book at any one time.

- It should be possible for a book to be borrowed by multiple users simultaneously, although only as long as there are available copies of that book in the library. As soon as one copy is returned, it is up for grabs again.

- A user is to be allowed to have many different books in their possession at any point time. User-initiated requests for a book to be returned are unfortunately not to be implemented due to time constraints, nor are waiting lines or book reservations. If a book of interest is in somebody else's possession, the user interested in borrowing that book should be prompted to swing by the library and request its return.

Technicalities (skip this)

Due to the fact that the authors of a book are not used for its unique identification, there is a theoretical possibility for the existence of two or more books that differ only in their ISBN codes and are otherwise completely identical. As this is indeed a real-life possibility, this database does not strive to eliminate it and therefore relies on the validity and authenticity of the data before that data is inserted into the database.

As part of one of the simplifications mentioned in the previous section, translations of a book or audio narrations of the same book are no longer uniquely identified within the library item relation, but instead delegated to its children relations, *book* and *audiobook*, where they are now uniquely identified by *language* and *narrator*, respectively. This database is therefore constructed on a fictional premise, namely that narrations and translations of a book do not have their own ISBNs that are different from that of the corresponding edition of that book written in its native language.

Note: republications of a book are still uniquely identified and can be assigned a whole new, unique batch of narrations and translations.

Security

Security is not taken into account and therefore user *emails* and *passwords* are simply stored as attribute values of that **user** entity (unencrypted).

Generalisation:

All standalone entities of the parent relation *lib_item* are abstract in that they can't be read or borrowed by users unless they exist in physical, digital, and/or audio form, that is, unless they are linked to entities within the *phys_book* and/or *audio_book* relations. A standalone entry in the lib_item relation that does not (yet) correspond to any relations in the phys_book or audio_book relations represents an item that has been ordered by the library and is awaiting delivery to the library while also being unavailable in audio or digital format. This usually symbolizes pre-ordered books that have yet to be printed.

The parent/child relationship between *lib_item* and its children *phys_book* and *audio_book* is partial in that, as clarified in the paragraph above, not all entries in lib_item are required to be linked to existing entries in phys_book or audio_book. What's more, the relationship is also non-disjoint in that a library item can exist in both audio and physical format.

Normalisation Process

The database schema was normalised using the ER model diagram for reference; below are all the steps involved in the process:

- The composite attribute **author.name** was not decomposed into three separate attributes but simply joined into one string attribute that contains the first, middle, and last name of the author while still remaining unique within the **author** relation. This would be a non-atomic attribute if any of the three parts of the author's name were to be used individually elsewhere, but, as that isn't the case in this database, the full name of the author is treated like non-atomic attribute of its own. Decomposing it would have introduced additional overhead for no reason as the individual parts of the authors' names are not used to build recommendation or comparison algorithms.

- Both child entities of the **library item** relation include their parent's primary key as part of their own primary keys.

- The multi-valued **genre** attribute in the **library item** relation was created into a separate relation and then cross referenced.

- The composite attribute **address** in the **library** relation was simply decomposed into separate attributes.

- The **time period** relation was merged with the **borrowed_by** relation which represents the ternary relationship in the diagram. This was done as there was no practical reason to maintain a separate relation for the time slots. Thanks to various conditions in PHP, the many-to-many-to-many relationship is maintained and not lost. This was delegated to the PHP scripts because, as it is widely known, SQL is not the best at dealing with temporal data and relationships that vary across time.
- The **language** attribute within the **audio_book** and **phys_book** relations was created into a relation of ist own and then cross referenced so as to avoid repetition across the two relations.
- The identifying relationship between the **library** and its **sections** was implemented by giving **sections** a compound primary key which contains the primary key of the corresponding library.

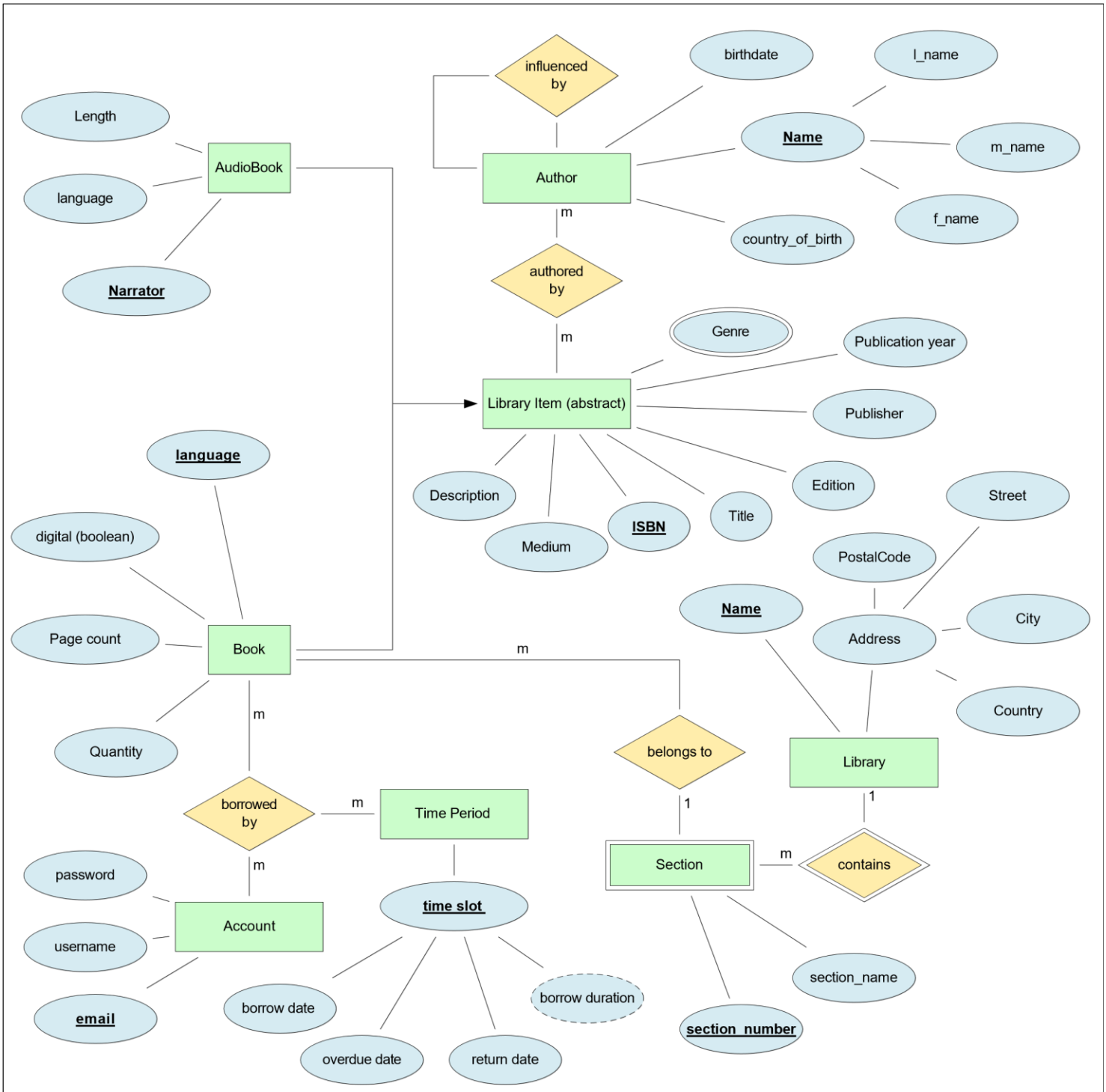


**Figure 1: Entity Relationship Diagram**

## Milestone 2: Logical Desgin

*composite keys are always enclosed by brackets

**Item_medium** (medium_id, medium_name)
PK: medium_id
FK:
Candidate Keys: medium_id, medium_name

**Language** (lang_id, lang_name)
PK: lib_id
FK:
Candidate Keys: lang_id, lang_name

**Library** (lib_id, lib_name, lib_street, lib_city, lib_country, lib_postcode)
PK: lib_id
FK:
Candidate Keys: lib_id, (lib_street, lib_city, lib_country, lib_postcode)

**Lib_section** (lib_id, section_id, section_name)
PK: (lib_id, section_id)
FK: Lib_section.lib_id ◊ Library.lib_id
Candidate Keys: (lib_id, section_id)

**Item_genre** (genre_id, genre_name)
PK: genre_id
FK:
Candidate Keys: genre_id, genre_name

**Author** (author_id, author_name, author_birthdate, country_of_birth)
PK: author_id
FK:
Candidate Keys: author_id

**Account** (account_id, username, email, password)
PK: account_id
FK:
Candidate Keys: account_id, email

**Lib_item** (item_id, ISBN, title, edition, publisher, publication_year, item_desc, *medium_id*)
PK: item_id
FK: Lib_item.medium_id ◊ Item_medium.medium_id
Candidate Keys: item_id, ISBN

**Audio_book** (*item_id, narrator*, *lang_id*, length)
PK: (item_id, narrator)
FK: Audio_book.item_id ◊ Lib_item.item_id, Audio_book.lang_id ◊ Language.lang_id
Candidate Keys: item_id, narrator

**Phys_book**(*item_id*, page_count, dig, quantity, *lang_id*, *lib_id, section_id*)
PK: (item_id, lang_id)
FK: Phys_book.item_id ◊ Lib_item.item_id, Phys_book.(lib_id, section_id) ◊ Lib_section.(lib_id, section_id), Phys_book.lang_id ◊ Language.lang_id
Candidate Keys: (item_id, lang_id)

**Influenced_by** (*author_id1, author_id2*)
PK: (author_id1, author_id2)
FK: Influenced_by.author_id1 ◊ Author.author_id, Influenced_by.author_id2 ◊ Author.author_id
Candidate Keys: (author_id1, author_id2)

**Authored_by** (*item_id, author_id*)

```
PK: (item_id, author_id)
FK: Authored_by.item_id ◊ Lib_item.item_id, Authored_by.author_id ◊ Author.author_id
Candidate Keys: (item_id, author_id)
```

```
Borrowed_by (borrow_id, account_id, item_id, lang_id, borrow_date, return_date,
overdue_date)
PK: borrow_id
FK: Borrowed_by.account_id ◊ Account.account_id, Borrowed_by.(item_id, language) ◊
Phys_book.(item_id, lang_id)
Candidate Keys: borrow_id
```

```
Belongs_to (item_id, genre_id)
PK: (genre_id, item_id)
FK: Belongs_to.genre_id ◊ Item_genre.genre_id, Belongs_to.item_id ◊ Lib_item.item_id
Candidate Keys: (genre_id, item_id)
```

## Milestone 3: Physical Design and Start of Implementation

See the DDL file for the data definition queries responsible for the creation of all tables, 5 views, 1 stored procedure, 1 sequence, and 1 trigger. The create table statements also contain all of the prescribed integrity, referential, and domain constraints. Furthermore, the 5 views mentioned above make use of the prescribed aggregate functions, namely, GROUP BY… HAVING, as well as JOINS such as INNER and OUTER joins. Other aggregate functions such as COUNT are embedded in the databasehelper.php file.

See the DELETE sql file for all of the corresponding DELETE i.e. DROP statements that can be used to rebuild the database whenever needed.

Lastly, see the DML.sql file for a small portion of testing data contained within INSERT and UPDATE statements; the testing data should be sufficient to verify all of the prescribed functionalities of the database, particularly those articulated in the *Requirements* section, without the need to launch the java data generator program.

## Milestone 4: Stored Procedure, Trigger, Sequence

Java

The java app is very basic; I originally tried to import some open-source library data but due to time constraints eventually decided to take the simpler route of generating my own (ensuring that each book corresponds to its ISBN and is correctly linked to all of its authors turned out to be more time-consuming than I expected).

The data I used for each library item , i.e. book titles, originates from the exported library of *my books* from my GoodReads.com account page (goodreads allows users to export their reading lists in csv format), to which I added a randomly selected list of around 70 authors who wrote abstracts and papers for the *Conference of Neural Information Processing Systems* between 1987 and 2017 (this I obtained from a dataset on Kaggle using the *Open Database License* – link: https://www.kaggle.com/benhamner/nips-papers).

Therefore, the data added to the library, although realistic in appearance, does not correspond to reality as none of the books are actually authored by their specified authors and the ISBNs are also not verified. Furthermore, things such as the authors' birthplaces were inserted at random, and

narrators were simply assigned numbers (e.g. Narrator no 23 or Narrator no 2032) instead of names because I couldn't find a good list of narrators to use. The data is nevertheless perfectly suited for testing purposes.

The only other relation that is read from a csv file is medium data, which is just a list of around 15 mediums and their key values I wrote in csv personally, and all other attribute values are generated in java directly.

PHP

All of the 15 or so PHP files that support the library web page are filled with comments which should thoroughly explain all pieces of code contained within them. There may be some portions of redundant code that are repeated and could have been avoided but stylistic concerns were heavily overlooked due to time constraints. Nevertheless, communication with the database is achieved with the use of object-oriented PHP and all the functions are contained within the databasehelper.php file.

The final product required extensive testing to ensure expected behaviour in a variety of different scenarios – especially because session functions were not used – but there is always a slight possibility of unexpected events occurring, particularly because not all SQL errors are caught in PHP.

The only thing I am not particularly pleased with is the fact that all links that aren't pure <a> hyperlinks are actually <input> elements whose values are *hidden* in HTML; this effectively allows the user to edit the values sent to the server simply accessing the developer tools in their browser and modifying the values manually, thereby allowing them to upload illegal values potentially leading to undefined behaviour or SQL injections. I found no workaround to this that did not involve JavaScript.

A screenshot of the databasehelper file where all the class function are stored:

```php
class DatabaseHelper
{
    const renew_incr = 7;
    const return_date = 14;
    const renew_allowed = 2;
    // Since the connection details are constant, define them as const
    // We can refer to constants like e.g. DatabaseHelper::username
    const username = ''; // your username
    const password = ''; // use your oracle db password
    const con_string = '' // your local server uri i.e. port;

    // Since we need only one connection object, it can be stored in a member varia
    // $conn is set in the constructor.
    protected $conn;

    // Create connection in the constructor
    public function __construct()  {…
    public function __destruct() {…

    public function SearchQuery($author_name, $item_title, $medium_id, $genre_id, $

    public function MediumDropdown() {…

    public function GenreDropdown() {
```