DataBases

Contents

1 What are they?

Databases are structured collections of data. The are used to store information.
Databases are part of just about all aspects of business and digital technologies.
Retail , banking, social networks, any kind of reasonably sophisticated website, educational institutions , etc all use them.
Databases are made up of rows and columns of information and the simplest idea of a database is a table or spreadsheet.
In order to store large amounts of information the data is organised into many different tables with connections (or relationships) between the tables.
The data base must be designed so that information can be efficiently read , added, deleted.
As databases often store personal data maintaining the integrity and security of data is essential.

2 What do we want to do with them?

We want to:

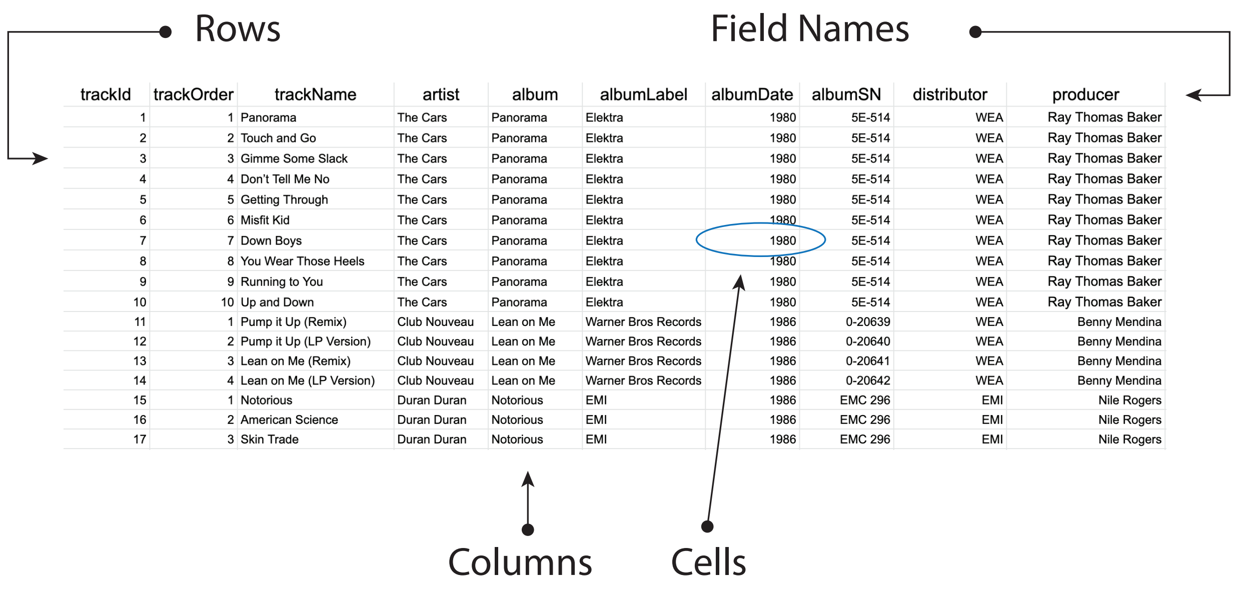
- Design the database so that it will efficiently hold information
- Put information in it
- Get information out of it
- Update information
- Delete information
- The above is sometimes acronymed to CRUD (Create Read Update Delete)

Data bases can be created and managed using programming languages such as Python and PHP.
We will work directly to a data base using the SQLite Browser
The most common language used to manage a database is SQL (Structured Query Language) (Which can be used in conjunction with the above languages).

3 Structure of a database table

4 Designing: part 1

We will go through a series of iterations to create progressively better database designs.
Our first example is a single table database.



trackId	trackOrder	trackName	artist	album	albumLabel	albumDate	albumSN	distributor	producer
1	1	Panorama	The Cars	Panorama	Elektra	1980	5E-514	WEA	Ray Thomas Baker
2	2	Touch and Go	The Cars	Panorama	Elektra	1980	5E-514	WEA	Ray Thomas Baker
3	3	Gimme Some Slack	The Cars	Panorama	Elektra	1980	5E-514	WEA	Ray Thomas Baker
4	4	Don't Tell Me No	The Cars	Panorama	Elektra	1980	5E-514	WEA	Ray Thomas Baker
5	5	Getting Through	The Cars	Panorama	Elektra	1980	5E-514	WEA	Ray Thomas Baker
6	6	Misfit Kid	The Cars	Panorama	Elektra	1980	5E-514	WEA	Ray Thomas Baker
7	7	Down Boys	The Cars	Panorama	Elektra	1980	5E-514	WEA	Ray Thomas Baker
8	8	You Wear Those Heels	The Cars	Panorama	Elektra	1980	5E-514	WEA	Ray Thomas Baker
9	9	Running to You	The Cars	Panorama	Elektra	1980	5E-514	WEA	Ray Thomas Baker
10	10	Up and Down	The Cars	Panorama	Elektra	1980	5E-514	WEA	Ray Thomas Baker
11	1	Pump it Up (Remix)	Club Nouveau	Lean on Me	Warner Bros Records	1986	0-20639	WEA	Benny Mendina
12	2	Pump it Up (LP Version)	Club Nouveau	Lean on Me	Warner Bros Records	1986	0-20640	WEA	Benny Mendina
13	3	Lean on Me (Remix)	Club Nouveau	Lean on Me	Warner Bros Records	1986	0-20641	WEA	Benny Mendina
14	4	Lean on Me (LP Version)	Club Nouveau	Lean on Me	Warner Bros Records	1986	0-20642	WEA	Benny Mendina
15	1	Notorious	Duran Duran	Notorious	EMI	1986	EMC 296	EMI	Nile Rogers
16	2	American Science	Duran Duran	Notorious	EMI	1986	EMC 296	EMI	Nile Rogers
17	3	Skin Trade	Duran Duran	Notorious	EMI	1986	EMC 296	EMI	Nile Rogers

There are "rules" (which can be open to interpretation) about how a database is design.
The process of implementing these rules is called **Normalisation**
We will rougly follow these rules.

First Normal Form

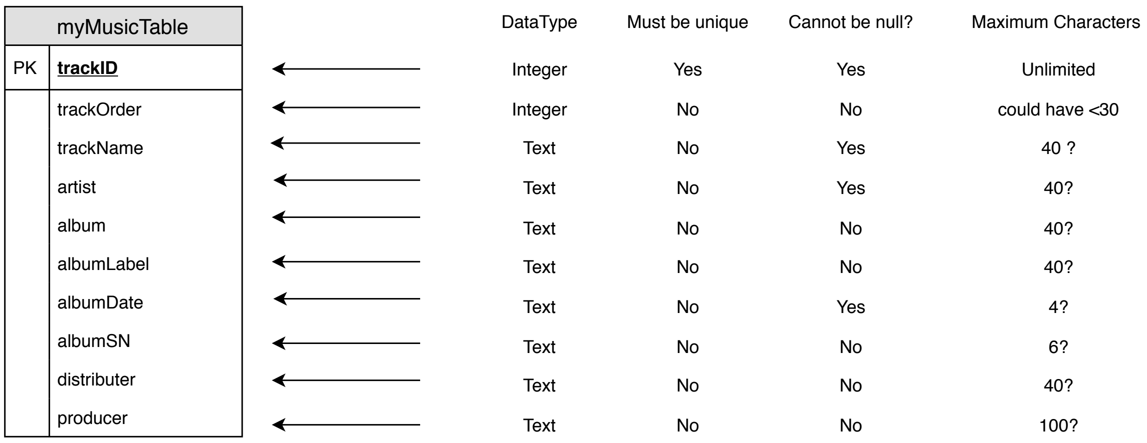
- Columns have a defined data type, and all pieces of data are of that type.
- Data must be a "single fact" (atomic).
- Each row must have some characteristic that makes it unique (cannot have two or more identical rows)

4.1 The Primary Key

In order to guarantee row uniqueness each table must have a primary key.
Primary keys can be a particular field (for example an email field for individual users is something we expect to be unique)
Primary keys could be made up of a combination of more than one field.
For our purposes we will have a field (which will always have "id" as part of its name) that is an **integer** field.
Seen from a larger perspective this may not always be ideal but will be fine in our case.
In order to guarantee the uniqueness of the field we always let the database manage the numbering.

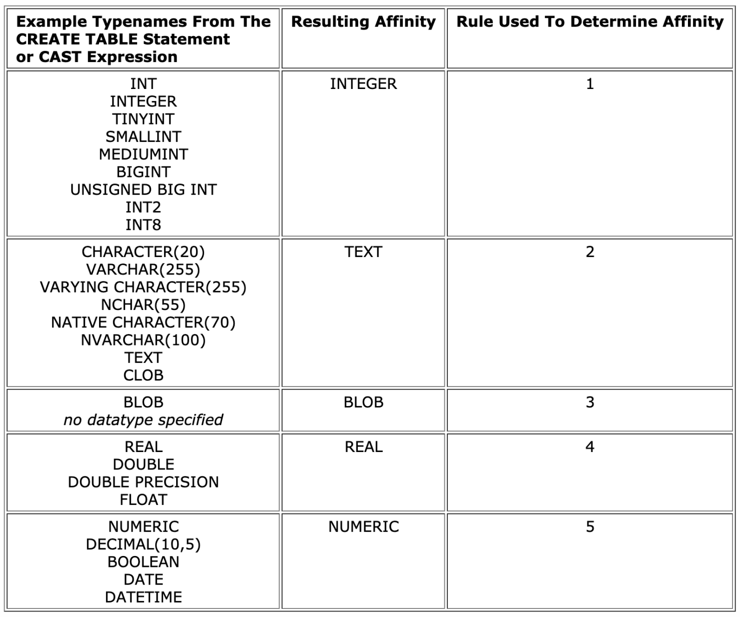
4.2 Basic Design

Looking at the design for a table of information about the LP records we could have a design like below.



myMusicTable		DataType	Must be unique	Cannot be null?	Maximum Characters
PK	**trackID**	Integer	Yes	Yes	Unlimited
	trackOrder	Integer	No	No	could have <30
	trackName	Text	No	Yes	40 ?
	artist	Text	No	Yes	40?
	album	Text	No	No	40?
	albumLabel	Text	No	No	40?
	albumDate	Text	No	Yes	4?
	albumSN	Text	No	No	6?
	distributer	Text	No	No	40?
	producer	Text	No	No	100?

The data types defined are very simple in the SQLite environment.



Example Typenames From The CREATE TABLE Statement or CAST Expression	Resulting Affinity	Rule Used To Determine Affinity
INT INTEGER TINYINT SMALLINT MEDIUMINT BIGINT UNSIGNED BIG INT INT2 INT8	INTEGER	1

CHARACTER(20)
VARCHAR(255)
VARYING CHARACTER(255)
NCHAR(55)
NATIVE CHARACTER(70)
NVARCHAR(100)
TEXT
CLOB
	TEXT	2

BLOB *no datatype specified*	BLOB	3
REAL DOUBLE DOUBLE PRECISION FLOAT	REAL	4
NUMERIC DECIMAL(10,5) BOOLEAN DATE DATETIME	NUMERIC	5

SQLite affinities

We also need to specify if the data must be unique in a particular field, and if the field can have empty cells).
Ideally empty cells is something we like to avoid.
The code for creating the table is below.
By defining a field as the primary key, this also will be understood as being unique and the database will automatically generate a primary key.

```sql
create table myMusicTable(
trackId integer  primary key,
trackOrder text,
trackName text not null,
artist text not null,
album text,
albumLabel text,
albumDate text not null,
albumSN text,
distributor text,
producer text
);
```

Listing 1: create myMusicTable

Once this has been created we can upload a csv of the data in our spreadsheet and copy it across into the myMusicTable (see video)

```sql
insert into myMusicTable(
trackOrder, trackName, artist, album, albumLabel, albumDate, albumSN,
distributor ,producer
)
select trackOrder, trackName, artist, album, albumLabel, albumDate, albumSN,
distributor ,producer
from temp
```

Listing 2: create myMusicTable

Ensure you delete the temporary table after you have copied the data across.

```sql
drop table temp;
```

Listing 3: delete a table

4.3 Basic Queries

At this point we have a simple database and it is worth exploring how data is read from it.
It is important to imagine that in any real data base, they may be many thousands or even millions of rows.
The idea of scrolling through the data is not an option.
Let's consder some things we might want to find from the data

- Find all tracks
- Find all tracks by The Cars (Showing the producer)
- Find all tracks before 1986 (include band and label in the output)

- Find the artist called Club “something” and show their albums
- Organise the tracks by Berlin in alphabetical order (include label and producer).
- Find all tracks produced by Georgio Moroder
- Find out how many different albums are in the data base.

Access https://www.w3schools.com/sql/ to reserach how to do this.

```sql
select trackName , album
from myMusicTable
```

Listing 4: query example

```sql
select trackName, producer
from myMusicTable
where artist = "The Cars"
```

Listing 5: query example

```sql
select trackName, artist, albumLabel
from myMusicTable
where albumDate < "1986"
```

Listing 6: query example

```sql
select trackName, artist, albumLabel, producer
from myMusicTable
where artist = "Berlin"
order by trackName asc;
```

Listing 7: query example

```sql
select trackName , producer
from myMusicTable
where producer like "%Mor%"
```

Listing 8: query example

```sql
select trackName
from myMusicTable
where trackName like "%Dan%"
```

Listing 9: query example

```sql
select count(trackName) as "Number of tracks", album, artist
from myMusicTable
group by album
```

Listing 10: query example

```sql
select   count(distinct album) as "number of albums"
from myMusicTable
```

Listing 11: query example

```sql
select trackName, artist
from myMusicTable
where trackName like "L%" or trackName like "D%"
```

Listing 12: query example

5 Designing Part 2

Our current database has limitations and the main one is that there is a large amount of repetition.
Apart from using up a lot of space, this is also problematic if we needed to alter the data (as each repeated block of data would have to be updated separately).
We could also consider the problems we would face if we wanted to add some further information fields about the albums (more repetition)

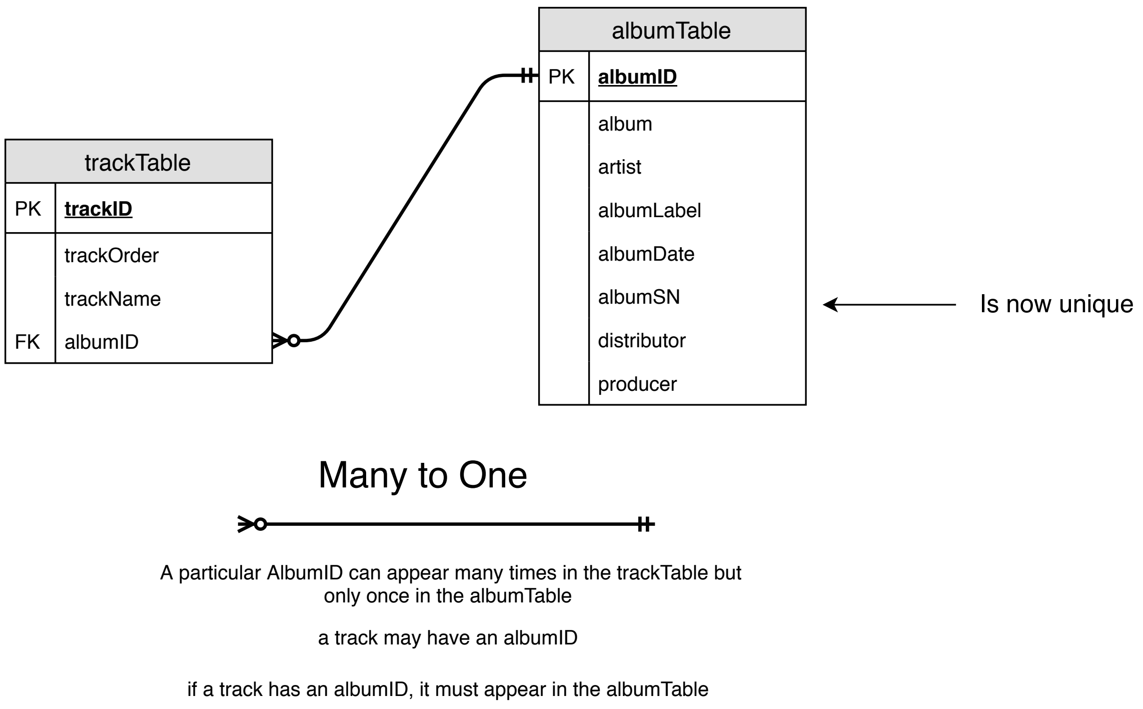
Second Normal Form

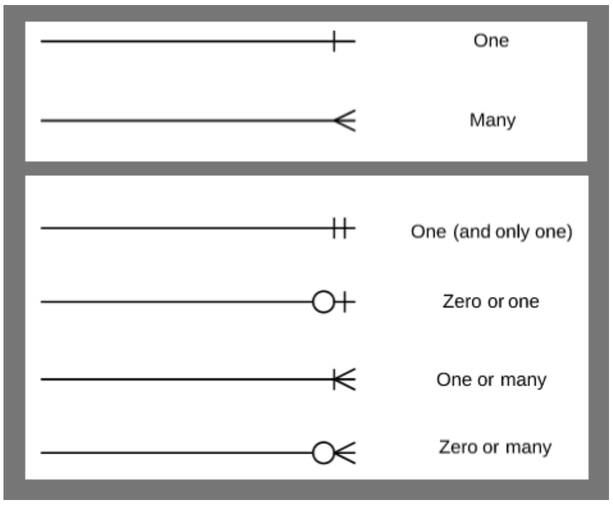
- The data must be in First Normal Form
- The database does not have distinct tuples with the same value.

An example of a distinct tuple in our case is: "The Cars,Panorama,Elektra,1980,5E-514,WEA"
It is debatable to include the producer but we can have more than one producer per album.

To solve this probem we move the albums to a separate table.
Our data base design now looks like the two boxes and the connector in the picture below.

In order to relate the trackTable with the album table, we need to use a unique key from one table and use it in the other.
It is logical use the primary key from one table and apply it as a **foreign key** in the other.
Data bases that relate tables in this way are called **relational databases**
To make the relationship clear we use identical field names in both.
How the foreign key and the primary key relate to each other is communicated using "Entity Relationship Connectors"
This can be discussed further in class but the key point at this stage is to either use connectors from the first group or the second group.

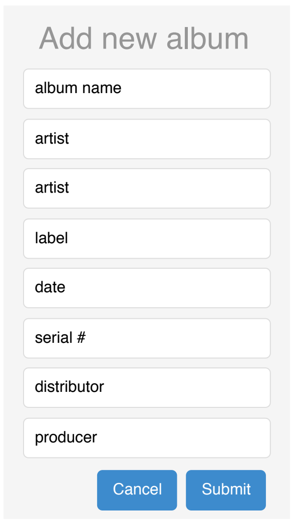

albumTable
PK albumID
album
artist
albumLabel
albumDate
albumSN
distributor
producer
Is now unique
trackTable
PK trackID
trackOrder
trackName
FK albumID
Many to One
A particular AlbumID can appear many times in the trackTable but only once in the albumTable
a track may have an albumID
if a track has an albumID, it must appear in the albumTable

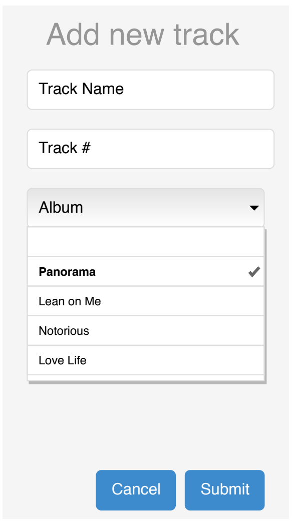


Entity Relationship Connectors

To build our database the best way is to upload the album data first and then write in the foreign keys before uploading the tracks data (see video).

If we were using forms to enter the data, we would probably set up the album information and then, when entering tracks we could access the appropriate album from a list of available albums.
In a database context a list of already available items is called a “lookup”.





5.1 Basic set-up code for the new database

```
create table albumTable(
albumId integer primary key,
album text not null,
artist  text not null,
albumType text not null,
albumLabel  text,
albumDate text,
albumSN text unique,
albumDistributer  text,
producer text
)
```

Listing 13: create album table

Upload the csv data and copy across

```
insert into albumTable(
album, artist, albumType, albumLabel  , albumDate , albumSN,albumDistributer
    ,producer
)
select album, artist, albumType, albumLabel , albumDate , albumSN,
    albumDistributer,producer
from temp
```

Listing 14: copy data from temp

```
drop table temp
```

Listing 15: delete temporary table

```
create table trackTable(
trackID integer primary key,
trackOrder integer,
trackName text not null,
albumID integer
)
```

Listing 16: create track table

Upload the csv data and copy across

```
insert into trackTable(
trackOrder, trackName,  albumID)
select trackOrder,  trackName, albumID
from temp
```

Listing 17: copy data from temp

```
drop table temp
```

Listing 18: delete temporary table

5.2 Joining

In order to execute queries across both tables, we need to perfom a **join**. If we wish to show all songs on the album "Love Life".

We

- define the fields we want to appear this time using dot syntax to identify which tables we are referring to
- we identify the first table we are taking from
- join the next table and identify the equal references (i.e primary key and foreign key)
- add any restrictions to the query

```
select trackTable.trackName, albumTable.album
from trackTable
join albumTable on trackTable.albumID = albumTable.albumId
where albumTable.album = "Love Life"
```

Listing 19: joining

find all owners of a Berlin album want album name and owner name

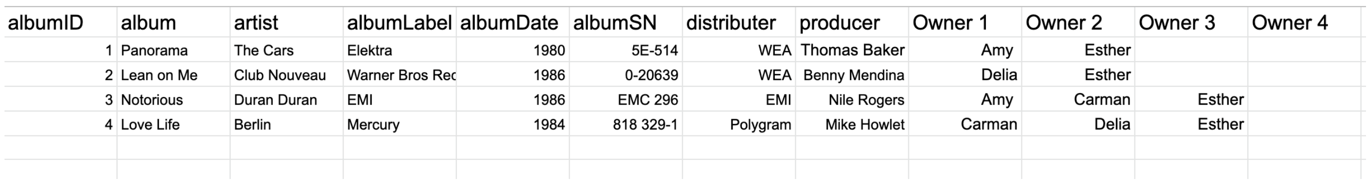
6 Designing part 3

6.1 Association tables

Let's extend our database further.
Consider we want to store information about people and which of the records they own.
Let's assume we are only interested whether the person owns an album or not (not whether they have more than one copy of the album).
We also assume that we are talking about physical records (discs) so they cannot own some of the tracks on an album.
We will have Amy, Belinda, Carman, Delia, Esther and Francesca who live in Northland, Karori, Wadestown, Northland, Thorndon and Karori respectively.

- Amy owns Panorama and Lean On Me
- Belinda owns no albums
- Carman owns Notorious and Love Life
- Delia owns Lean On Me and Love Life
- Esther owns Panorama, Lean On Me, Notorious and Love Life

This information needs to be stored in the data base.
Let's look at what happens in the database if we store this information in the albums table.



albumID	album	artist	albumLabel	albumDate	albumSN	distributer	producer	Owner 1	Owner 2	Owner 3	Owner 4
1	Panorama	The Cars	Elektra	1980	5E-514	WEA	Thomas Baker	Amy	Esther		
2	Lean on Me	Club Nouveau	Warner Bros Rec	1986	0-20639	WEA	Benny Mendina	Delia	Esther		
3	Notorious	Duran Duran	EMI	1986	EMC 296	EMI	Nile Rogers	Amy	Carman	Esther	
4	Love Life	Berlin	Mercury	1984	818 329-1	Polygram	Mike Howlet	Carman	Delia	Esther	

albumTable with owners added at the end

This is clearly very problematic,

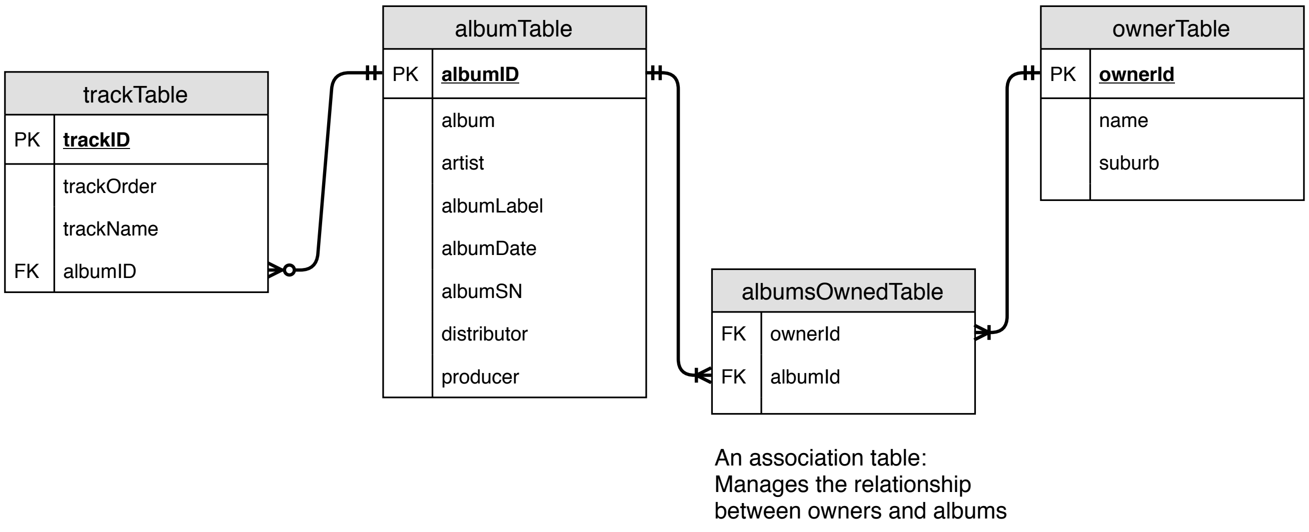
- Many empty cells.
- We have no idea what the maximum number of owners could be in the future.
- It makes things even more difficult if we want to include further information about the owners.
- There is a contradiction of Entities (Albums are one Entity and People are another).

- If our database is well designed we should be able to add new entities without interfering with those that are already included.

What we have have in this case is **repeating groups** (of owners) and although we have not seen it until now this is always the first problem that is dealt with when normalising a database.

It is also interesting to note that we have a many - to - many relationship between albums and owners.
Many albums can be owned by a particular owner and Many owners can have the same album.
(many albums can be owned by many owners and many owners can have many albums)

To resolve this design problem we clearly need a table for the owners (a distinct entity). But we also need another table to manage the **association** between the Albums and the Owners.
Unsurprisingly this is called an **association table**.



music database with association table

Assuming each person can only own a particular album or not (we are not interested in multiple copies), We could consider the **combination** of ownerId and albumId as the primary key.
So the primary key is a combination of two foreign keys.
You can see in one of the examples below how this is implemented.

6.2 Joining

Joining now becomes more complicated, but the basic rules of joining are

- you can have as many joins as you need
- start at one "end" and join your way to the other "end".

Below is some code examples for implementing the two new tables.

```sql
create table ownerTable(
ownerId integer primary key,
name text not null,
suburb text
);
```

Listing 20: creating owner table

```sql
insert into ownerTable(name, suburb)
select name, suburb
from tempowners;
```

Listing 21: copying across temporary table

```sql
create table albumsOwnedTable(
ownerId integer not null,
albumId integer not null,
primary key(ownerId,albumId)
);
```

Listing 22: creating table of albums owned (note primary key definition)

```sql
insert into albumsOwnedTable( ownerId, albumId)
select ownerId, albumId
from tempalbumsOwned;
```

Listing 23: copying across temporary table

```sql
insert into albumsOwnedTable(ownerId, albumId)
values( 1, 1)
```

Listing 24: trying to add a value pair that is already present in the albumsOwnedTable (this should produce a uniqueness error)

```sql
select albumTable.album, ownerTable.name
from albumTable
join albumsOwnedTable on albumsOwnedTable.albumId = albumTable.albumId
join ownerTable on albumsOwnedTable.ownerId = ownerTable.ownerId
```

Listing 25: all albums and owners

```sql
select trackTable.trackName, albumTable.album, ownerTable.name
from trackTable
join albumTable on albumTable.albumId = trackTable.albumID
join albumsOwnedTable on albumsOwnedTable.albumId = albumTable.albumId
join ownerTable on albumsOwnedTable.ownerId = ownerTable.ownerId
where ownerTable.name = "Amy"
```

Listing 26: all tracks and owners

```sql
select count( distinct albumTable.album) as "number of owned albums",
    ownerTable.name
from albumTable
join albumsOwnedTable on albumsOwnedTable.albumId = albumTable.albumId
join ownerTable on albumsOwnedTable.ownerId = ownerTable.ownerId
group by ownerTable.name
```

Listing 27: counting how many albums each person owns

```sql
insert into albumsOwnedTable(ownerId, albumId)
values(
(select  ownerId
from ownerTable
where name = "Francesca"),

(select albumId
from albumTable
where album = "Panorama"
)
);
```

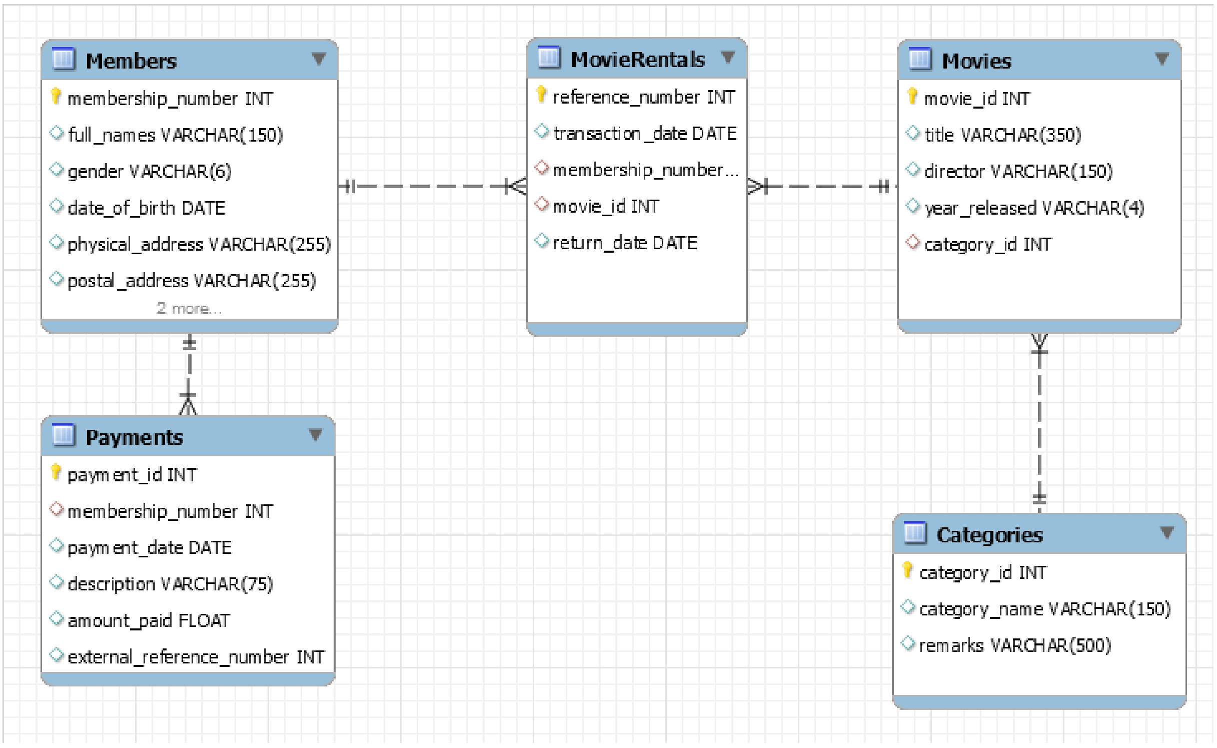
Listing 28: let's give Francesca a record.

```sql
delete from  albumsOwnedTable
where albumsOwnedTable.ownerId = ( select  ownerId
from ownerTable
where name = "Amy" );
delete from ownerTable
where name ="Amy";
```

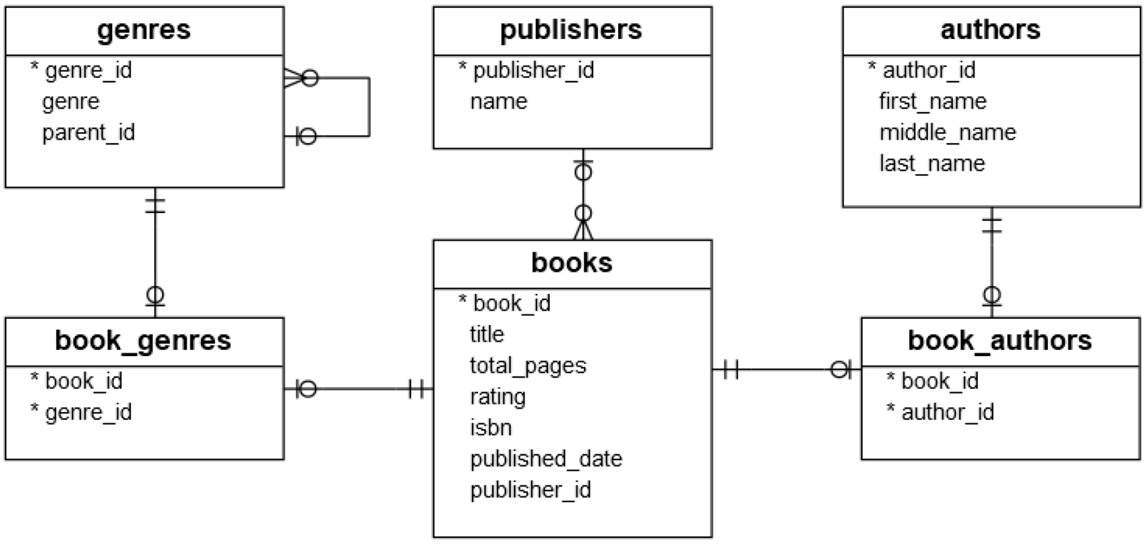
Listing 29: delete Amy from the system (so all foreign key references to Amy and then Amy herself).

```sql
-- reinstate Amy
insert into ownerTable(name, suburb)
values("Amy", "Karori");
insert into albumsOwnedTable(ownerId, albumId)
values(
(select  ownerId
from ownerTable
where name = "Amy"),
(select albumId
from albumTable
where album = "Panorama"
)
);
insert into albumsOwnedTable(ownerId, albumId)
values(
(select  ownerId
from ownerTable
where name = "Amy"),
(select albumId
from albumTable
where album = "Notorious"
)
);
```

Listing 30: re-entering Amy into the system (she will now have a new primary key).



Movie Rentals Entity Relationship Diagram



Book DataBase Entity Relationship Diagram