INFO20003 Week 3 Lab

ER Modelling with MySQL Workbench

Objectives:

In this lab you will:

- Learn about MySQL data types
- Choose proper data types for physical ER models
- Create a physical data model with the MySQL Workbench modelling tool

Section 1: Modelling participation constraints

In last week's lab you developed a simple music-related physical model with Song, Album and RecordCompany tables. It looked something like this:

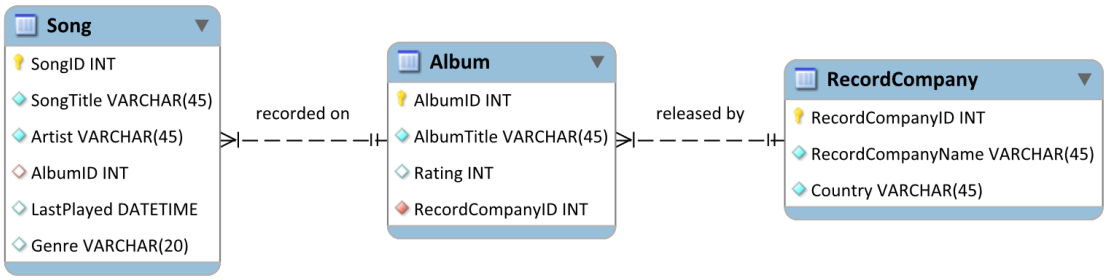

Figure 1: The Music model from Lab 2

◆ **Task 1.1** Re-open the .mwb model file you created last week. If you no longer have this file, you can download it from LMS.

Notice that all the relationships are shown as mandatory in this model:

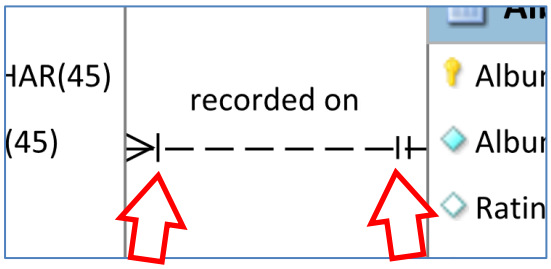

Figure 2: In Crow's foot notation, the innermost marker indicates mandatory (line) or optional (circle).

This is not realistic. Not all songs belong to an album, and a record company that has just started up will not have released any music yet. We need to change the participation constraints on this model.

◆ **Task 1.2** Double-click on the “recorded on” relationship between Song and Album.

The Relationship editor opens:

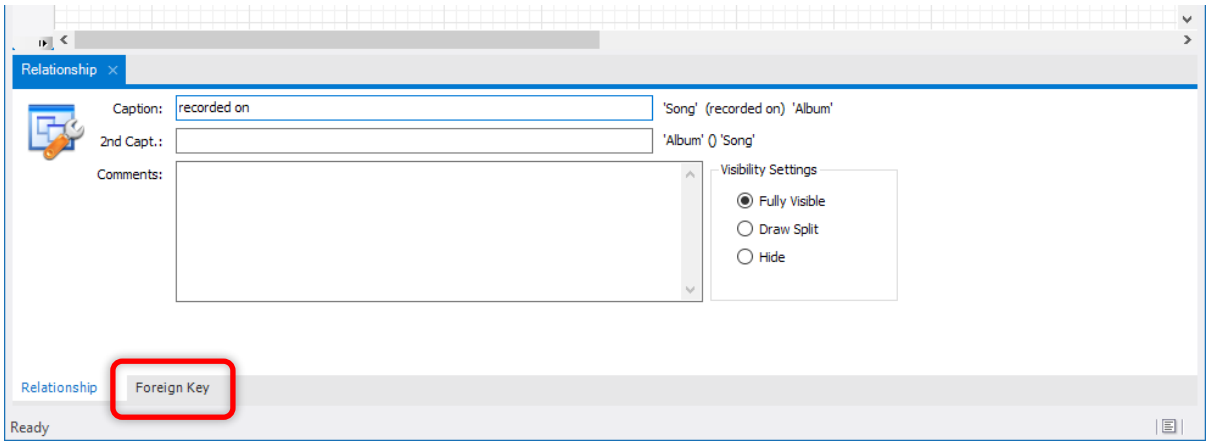

Figure 3: Editing the "recorded on" relationship

◆ **Task 1.3** Switch to the Foreign Key tab at the bottom of the window (circled in red above).

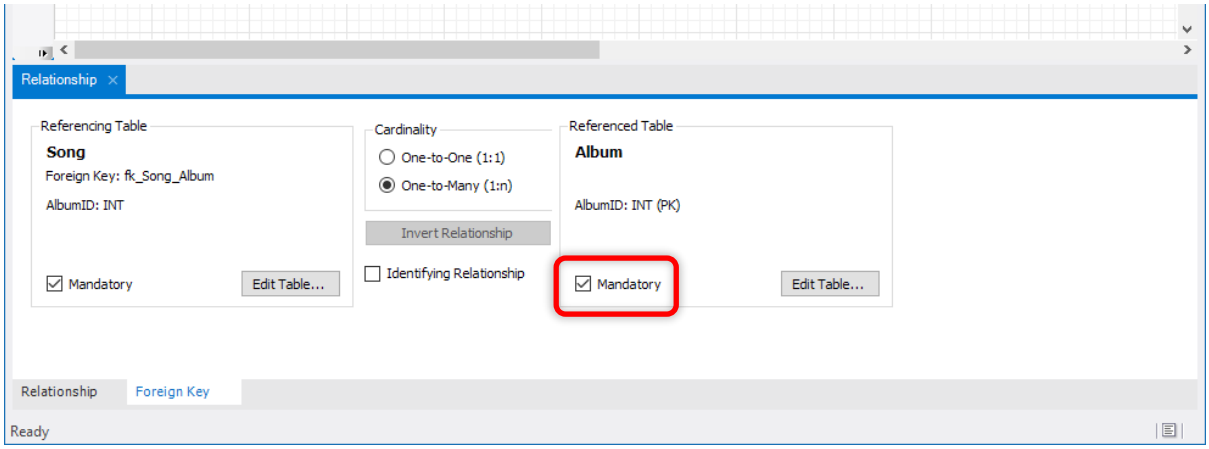

Figure 4: The Foreign Key tab of the Relationship editor

◆ **Task 1.4** Indicate that the Song table has optional participation in the relationship by unchecking the “Mandatory” box for Album.

This is somewhat counterintuitive – in order to change the participation constraint for the Song table, you change the Mandatory option for Album. Think of it this way: it *is not* mandatory for a Song to have an Album (hence Album is “not mandatory” from the point of view of Song), while it *is* mandatory for an Album to have at least one Song (hence Song is “mandatory” from the point of view of Album).

◆ **Task 1.5** Click in a blank part of the model to update the relationship.

The “mandatory” line has now changed to an “optional” circle.

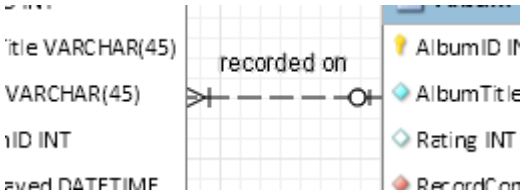

Figure 5: The "recorded on" relationship with the correct participation constraints applied

◆ **Task 1.6** Repeat tasks 1.2 to 1.5 to indicate that RecordCompany has optional participation in the “released on” relationship.

After completing Task 1.6 your model should look like this:

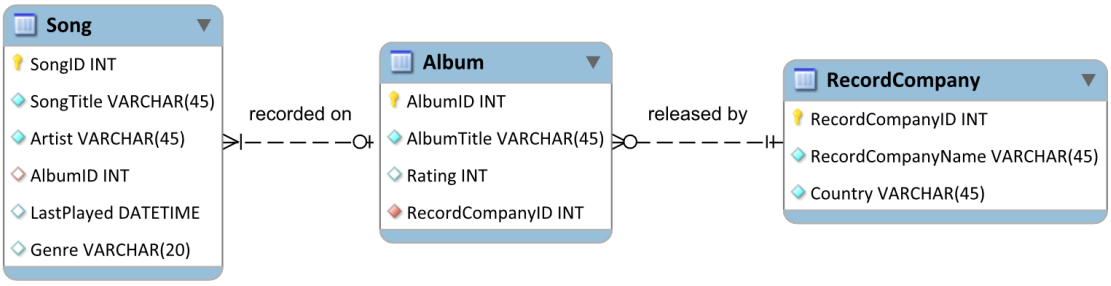

Figure 6: The final Music model, complete with participation constraints

Section 2: Drawing a physical model in MySQL Workbench

◆ **Task 2.1** Using MySQL Workbench, draw a physical model for the “cinema” case study from the Week 3 tutorial. You should already have developed a conceptual and logical model in class.

When modelling using Workbench, you should always draw **non-weak entities that are linked to weak entities** first. For this case study, it is suggested that you draw the tables in the following order: Cinema, Screen, Projector, Movie.

If you need help choosing data types, look at section 3 below. You might like to complete Task 3 to assist you.

Section 3: MySQL data types

NOTE: *Some students will not have time to complete this part during the scheduled lab. If this happens, please read the following material and complete Task 3 (on page 5) before next week’s class.*

So far, you have seen three fundamental MySQL data types: INT, VARCHAR and DATETIME. All relational databases support numbers (like INT), strings (like VARCHAR), and dates (like DATETIME). However, each relational database vendor (such as Oracle, Microsoft SQL Server and PostgreSQL) implements SQL standards in their own way. This becomes important when we move to physical (MySQL Relational Database) modelling.

Below is a quick revision of the three categories of data types and examples of how they are implemented in MySQL Server.

Strings

A string can be thought of as a piece of text we wish to store in a column. The typical string data type in MySQL is **VARCHAR(*n*)**, where *n* is the maximum length required. For example, if we expect that every employee's first name will be 40 characters or less, we might select the VARCHAR(40) data type for an EmployeeFirstName column.

VARCHAR data is stored flexibly: they only occupy as much space on disk as necessary. For example, storing the string `'abcd'` in a VARCHAR(50) column would take up 5 bytes – one byte to store the length, and four bytes to store the characters themselves.

The **CHAR(*n*)** data type can also be used for strings. Unlike VARCHAR, this data type always occupies *n* bytes even if the string being stored is shorter, so it is best for very short strings. For example, IATA airport codes (MEL, SIN, LAX) are always three letters long, so CHAR(3) would be a suitable data type. You could store a one- or two-letter code in this column if you needed to, but it would still take up 3 bytes.

If the string is limited to a very small and unchanging list of choices (such as movie ratings – G, PG, M, MA or R), use the **ENUM(…)** data type. For example, ENUM('G','PG','M','MA','R').

Finally, the **TEXT** data type can be used to hold longer passages of free-form text, such as the content of a web page. Each value in a TEXT column can contain up to 64 KB of text, while MEDIUMTEXT is a larger version that can store up to 16 MB.

Numbers

Various data types are available for storing numbers. In MySQL the most common numeric data types are INT, DOUBLE and DECIMAL.

Integers have their own family of data types in MySQL: TINYINT, SMALLINT, MEDIUMINT, INT, and BIGINT. Table 1 shows the ranges available. "Unsigned" data types allow positive values only; these can be used by writing TINYINT UNSIGNED, SMALLINT UNSIGNED etc, or in MySQL Workbench, checking the "U" box.

	Storage	**Min Signed**	**Min Unsigned**	**Max Signed**	**Max Unsigned**
TINYINT	1 byte	–128	0	127	255
SMALLINT	2 bytes	–32768	0	32767	65535
MEDIUMINT	3 bytes	–8388608	0	8388607	16777215
INT	4 bytes	–2147483648	0	2147483647	4294967295
BIGINT	8 bytes	-2^{63}	0	$2^{63} - 1$	$2^{64} - 1$

Table 1: The MySQL Integer family (Ref: MySQL Reference 11.2.1)

The **DOUBLE** data type is identical to the double data type in C and the float data type in Python. It occupies 8 bytes and is correct to about 15 significant figures. Because it is a floating-point data type, it is likely to experience rounding error.

The **DECIMAL(*n*,*p*)** data type is for storing exact-precision decimals (e.g. money). The value *n* is the total number of digits available, and *p* is the number of decimal places. So a DECIMAL(5,2) column can store values between –999.99 and 999.99. DECIMALs are stored exactly and do not suffer from floating-point rounding error.

Date

MySQL can store dates and time in a number of formats including **DATETIME**, **DATE**, **TIME** and **YEAR**. (There is also a TIMESTAMP data type, but this has certain undesirable properties and should normally be avoided.)

	Storage	**Format**	**Example**
YEAR	1 byte	'YYYY'	'1984'
DATE	3 bytes	'YYYY-MM-DD'	'1984-11-24
TIME	3 bytes [Note]	'HH:MM:SS'	'15:55:43'
DATETIME	5 bytes [Note]	'YYYY-MM-DD HH:MM:SS'	'1984-11-24 15:55:43'

[Note] TIME and DATETIME data types can store trailing fractions of a second up to microseconds (0.000001 of a second). Additional space is taken up when fractional seconds are stored.

The TIME data type can be used to represent the actual time of day or elapsed time. It can contain values from –838:59:59.000000 to 838:59:59.000000.

◆ **Task 3.1** Choose a suitable data type (or data types) for each of the following attributes. Where necessary, specify the length and precision (e.g. VARCHAR(50) instead of VARCHAR).

Attribute	**Data type**	**Why did you select this data type?**
Bank account balance		
Your full name		
Home address		
Postcode (Australian)		
LinkedIn page		
Website		
When a text message was sent		
When a person started working for a company		
Duration of a song		

Attribute	Data type	Why did you select this data type?
Unimelb room number (e.g. 109, 228A, G09)		
Unimelb assignment grade (H1, H2A, H2B, H3, P, N)		
A comment on a news article		

End of Lab 3