Complementing Database Design for Microsoft Access with MySQL Workbench

Jerzy Letkowski

Western New England University

Abstract

Microsoft Access is a popular computer program used to teach database design, development and applications in many business curricula. Students learn how to develop database tables and their relationships. They also learn how to populate the tables with data, using forms, generate cool reports, and develop quite sophisticated queries. More advanced topics get them into application development, using VBA. Consequently, they can appreciate lots of appealing features and tools offered by Access. Unfortunately, Access has a very weak point. It has insufficient support for one of the most important steps of the database life cycle: the logical data design. Yes, database experts, having a good understanding of the business context, can design databases by sketching logical data model [mentally] and going straight into developing a physical designs. Such experts also understand that a logical data model is the most critical task and, if well done, it will lead to a robust physical design. They also recognize that, in general, logical data models are developed by interdisciplinary teams, including subject-matter experts and end-users who do not necessarily have a good understating of the database jargon, requirements and restrictions. Thus, in order to complete the logical phase of the database design, the teams opt for working with less intimidating languages and tools, including more intuitive graphical/visual tools. This paper shows how to fill the Access's "logical gap", using MySQL Workbench. It uses a simple example of a data model expressed in UML and implemented in the Workbench. The model is then used to automatically generate an SQL schema. Finally, it gets to the physical design by executing the schema, using the SQL Query tool in Access. This approach results in a database, consisting of proper tables and relationships without requiring any additional steps.

Keywords: database, design, logical data model, table, relationship, query.

LOGICAL DATA MODEL FOR A RELATIONAL DATABASE

A database can be developed in many different ways. An experienced database developer, having a good grasp of the business specifications (including requirements, rules and constraints) can get right to the physical design using SQL or table and relationship building tools (all available in Access). Such an expert can usually get by without a logical data model or s/he can visualize the model without using any special (automatic) tools.

Mere mortals or those dealing with complex data structures have to work harder. In general, even if one believes that the logical model is not necessary, the analysis of the business side of the

design can contribute to better results. In many situations, getting a correct result depends decisively on accurate interpretation of business requirements and rules.

A classical approach to the database design involves three levels or views: external, conceptual and internal (Date, 2000, p. 33). The design process consists of several phases such as requirement analysis, conceptual (high level) design, and logical design—all leading to a physical design (Elmasri, et al., 2004, p. 51). This paper shows how to build a logical data model that is a combination of the conceptual and extremal views, resulting from the business requirement analysis. The model includes a complete set of entities, attributes, keys, relationships, cardinality and participation constraints that are necessary to accurately define a physical database (Silberschatz, et al., 2001, p. 27-37). It is structured as an Enhanced Entity Relationship Diagram (EERD), using MySQL Workbench. Such a model can be automatically transformed into a physical design (database schema) expressed in SQL (Letkowski, 2014).

In many situations, successful development of an ERD data model, using systems like MySQL Workbench, depends more on the business domain expertise than on the database skills. A subject-matter expert, who understands simple concepts of an entity (or class), instance (object) and relationship can decidedly contribute to development of the data model.

PROBLEM SPECIFICATIONS

Nick is a Web developer, who wants to run a Web site, hobbysta.org, specializing in sharing information among hobby enthusiasts all over the World. He asked Karolina, a teacher at a local business college, to design a prototype database in Access so that he could easily experiment with the database before integrating it with his Web site. Nick understands that his target database server will be MySQL or a similar database system, but he loves the QBE environment for experimenting with all kinds of queries that might be later used by his Web applications. He wants to start with a simple database that would allow for member registration and hobby selection. He wants to develop and maintain a logical data model that would be flexible enough to accommodate future expansions. To get started, he wants only simple information about members (given name, family name, and email) and hobbies (hobby title and short description). Karolina decided to get some help from her student, Emma, majoring in Entertainment Management. Emma does not have any database background, beyond what she learned in her spreadsheet class.

HIGH LEVEL DATA MODEL

The entire process of the database design is presented below in from of discussion among three individuals: Nick, Karolina, and Emma. The author of this paper (Author) provides additional comments.

Karolina: Having learned about Nick's situation, I recommend to develop a logical data model and to generate an SQL schema, using MySQL Workbench. The resulting schema will be very close to what Access needs in order to create tables and relationships. Before delving into *painting* the logical model, I want to verify my understanding of the Nick's specifications by writing short, but precise, English statements that will describe and bind members and hobbies. I suppose, a member can have at least one hobby and a hobby can be enjoyed by one or by many members. Actually, we should relax the strict requirements and allow for storing member records even if they do not have hobby and to have hobbies that are not involved with any member. Let use names Member and Hobby to represent collections of members and hobbies, respectively. At the initial (or high) level of the database design process, we refer to such names as entities or classes. Thus we have entity Member, a set of member records, and entity Hobby, a set of hobby records.

Nick: I am impressed. This is exactly what he had in mind. However, I do not fully understand that there could be hobbies not associated with any members and vice versa.

Emma: So do I!

Karolina: OK, supposed that you add a member who has a unique hobby (not picked by any other member). What do you want to do if the member quits. Do you want to also remove the hobby, or you want to keep it so that when another member picks the hobby, it will be right there among all the elements of the Hobby set? Another possible situation, would be to add the most popular hobbies to entity Hobby so that they would be ready to be selected by the members. This should simplify Web interfaces and minimize errors. Obviously, members will be able to pick additional (new) hobbies at any time. There may be also situations when a member wants to register with hobbysta.org but s/he prefers to pick a hobby or hobbies later. Finally, if we do not want to delete the record of a member who has dropped all her/his hobbies, then we should allow for the record to exist without connection to any hobby.

Author: They all agreed that an initial list (set) of the popular hobbies would be an excellent choice. It has also become clearer why we should allow for existence of orphan members and/or hobbies.

Karolina: Let me also explain why we are dealing here [initially] with two data entities: Member and Hobby. They are simply sets of objects (members and hobbies, respectively). A good practice is to name entities, using singular nouns written in a title case. To refer to instances of the entities we use a lower case. Thus members constitute the Member set and hobbies—the Hobby set. By the way, entities are also referred to as classes and instances as objects (as in object-oriented systems). To distinguish entities and their instances from other things, from now on, we will write them, using Currier fonts.

Author: Since there has been no database term used, Nick and especially Emma could follow everything so far.

Karolina: Emma, why don't you prepare instance samples of the entities, for example, in a spreadsheet (Excel). In the meantime, let me wrap their discussion and analysis in a more formal way, using an annotated `SUBJECT + PREDICATE + OBJECT` format:

`Member(0..*) - has - (0..*)Hobby`

What it means is that a given `member` (instance of set `Member`) can have zero, one, or more (`0..*`) `hobbies` and a given `hobby` (instance of set `Hobby`) can be selected by zero, one, or many `members` (`0..*`).

Emma: When you say it, it is all clear to me but if you wanted me to repeat it, I would have a hard time expressing exactly the same thing.

Karolina: Well, this is exactly what I suggested before but expressed in a more concise and crisp way. Let me try to explain it graphically (Figure 1). Bubbles represent objects (instances of the entities) and oval rectangles—sets of the objects (entities). Arrows connect (associate) `Member` bubbles with some `Hobby` bubbles. For simplicity, `members` are identified by their first names and `hobbies`—by their titles (other attributes are hidden). `Don` and `Tom` (instances of entity `Member`) have just one (1) `hobby` (collecting `Golf Balls`). `Pat` has two (*) `hobbies` (`Golf Balls` and `Coke`). Eve has three (*) `hobbies` (`Golf Balls`, `Stamps` and `Coke`). `Golf Ball` is picked by three (*) `members` (`Don`, `Pat`, and `Eve`). Both `Coke` and `Stamps` are selected by two (*) members (**`Pat`**, **`Eve`** and **`Eve`**, **`Tom`**, respectively). **`Post Cards`** (a `hobby` for collecting `Post Cards`) and `Kim` are an orphan objects. No `member` has picked this `hobby`. Thus it is associated with no (0) `members`. Similarly, member `Kim` has not selected any `hobby`. This it is assiciated with no (0) `hobbies`. Notice that the asteriscs shown above denote *many* instances.

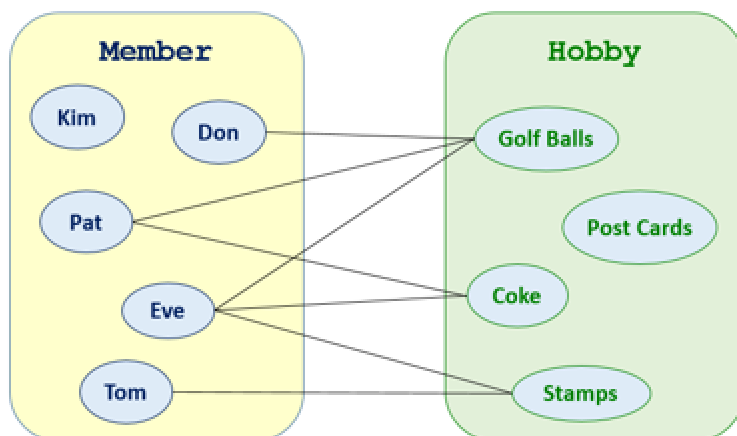

Figure 1 Entities, their instances and relationships.

Since there exist associations to many (*) instances on both sides (`Member` and `Hobby`), the relationship between entities `Member` and `Hobby` is referred to as a *`many-to-many`*

relationship. Such a relationship is not database friendly. It requires at least one extra [associative] entity to be placed between entity `Member` and entity `Hobby` but let's learn more about the nature of our data model before we can resolve this complex relationship.

Karolina (continuing): I can see that `Emma` has not wasted her time. She captured the above entities and their instances in `Excel` (Table 1).

Table 1 Initial sample data for entities Member and Hobby.

Member				**Hobby**	
givenName	**familyName**	**email**		**title**	**description**
Pat	Patyk	pp@hobbysta.org		Golf Balls	Collect golf balls.
Don	Donski	dd@hobbysta.org		Coke	Coca Cola memarobilia.
Eve	Evening	ee@hobbysta.org		Post Cards	Collect post cards.
Tom	Tomas	tt@hobbysta.org		Stamps	Collect postage stamps.
Kim	Kimura	kk@hobbysta.org			

Each bubble shown in Figure 1 in now represented by one row in the above tables. Woops, I said it but I am not supposed to say it, not yet. *Table* is a database term and we are not supposed to use it until we get to the physical design. We are still developing a logical model. Nonetheless, *table* is a popular term and it will help us move forward with two important issues that must be resolved. The first deals with the so called entity integrity and the second one—with the `many-to-many` relationship. The entity integrity requires that each entity instance be unique. Each entity must have one or more attributes that uniquely identify each of its instances. Such an attribute or a combination of attributes is referred to as a *candidate key*. Since there may be more than one candidate key, the one that we choose for *operational* purpose is call a *primary key*. If there is no natural and/or convenient primary key, we can create an *artificial* one, a *surrogate key*. Traditionally (and also because of efficiency) the best primary (surrogate) keys have values taken from natural numbers (1, 2, 3, ...). They are the Integer type.

Emma: I get it. Let me add those keys to my Excel tables. Should I add them as the left-most columns?

Karolina: Yes, and name them `mid` and `hid` (for member ID and hobby ID). As you edit the tables, why don't you also try to reflect the relationships shown in Figure 1?

Emma (after a few minutes): I did my best (Table 2).

Karolina: You took care of the entity integrity brilliantly but, unfortunately, the `memHobby` attribute show the relationships correctly but it is not *relationally* acceptable. All relational entities must have single valued (atomic) attributes. This requirement (as related to database tables) is reinforced by a database normalization rule, the so called First Normal Form (Date,

2000, p. 127). We could fix this problem right in Excel, but it will be easier and more professionally to do it in MySQL Workbench.

Table 2 Revised tables (entities) Member and Hobby.

Member				
mid	**givenName**	**familyName**	**email**	**memHobby**
1	Pat	Patyk	pp@hobbysta.org	1,2
2	Don	Donski	dd@hobbysta.org	1
3	Eve	Evening	ee@hobbysta.org	1,2,4
4	Tom	Tomas	tt@hobbysta.org	4
5	Kim	Kimura	kk@hobbysta.org	0

Hobby		
hid	**title**	**description**
1	Golf Balls	Collect golf balls.
2	Coke	Coca Cola memarobilia.
3	Post Cards	Collect post cards.
4	Stamps	Collect postage stamps.

We will start with two entities, `Member` and `Hobby`, having attributes shown in Table 2 except of the `memHobby` one. As mentioned, these entities are part of the high-level model expresses as:

`Member(0..*) - `*`has`*` - (0..*)Hobby.`

Author: Because of a limited space, this paper only shows outcomes of major steps performed with MySQL Workbench and later with Access. For detail instructions, consult the resource page specifically developed to complement this paper (Letkowski, 2016).

LOGICAL DATA MODEL

Karolina: Our first job is to place on the canvas (diagram board) the two entities (tables): **`Member`** and **`Hobby`**. Next, we add attributes to the entities (Figure 2).

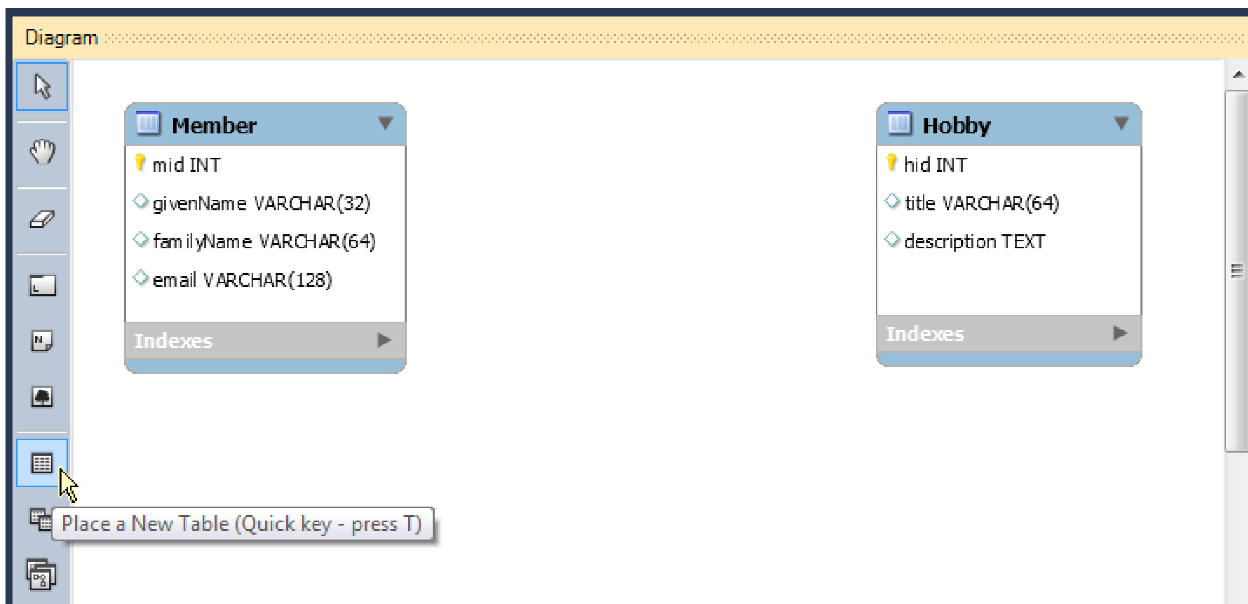

Figure 2 Creating entities with attributes in MySQL Workbench.

To this end, we use the **table** tool (or press **T**) and then click anywhere on the canvas. Next, we open the table's property panel by double-clicking the table. This will allow us to change the name of the table, and add the attributes and specify their types. By default, MySQL Workbench defines the first attributes as Primary Key which is what we want in order to ensure the entity integrity. Of course, if one wants different keys, they can be manually redefined.

Notice that MySQL Workbench does not use term *entity*. Instead, the entities are referred to as *tables*. Still, the diagram is called EERD (Enhanced Entity Relationship Diagram) rather than ETRD (Enhance Table Relationship Diagram). I guess, we can live with this slight intrusion of the database jargon.

It is interesting to note that, if we were to do it in Access, the tables would have a very similar structure (Figure 3). However, we would not be able to directly bind the tables (create a relationship) since Access is unable to resolve [automatically] many-to-many relationships. Moreover, in Access, in addition to the primary key, *foreign keys* would have to be defined.

Member

Field Name	Data Type
mid	Number
givenName	Short Text
familyName	Short Text
email	Short Text

Hobby

Field Name	Data Type
hid	Number
title	Short Text
description	Long Text

Figure 3 Creating tables Member and Hobby in Access.

Emma: I am not sure I understand. What is a foreign key?

Karolina: Well, I messed things up a little but I did it on purpose so you can see that going forward with the design in Access would require a higher level of the database expertise. I want to complete the logical design without those database buzzwords. Bear with me, you will soon see how foreign key are born and what they do.

Emma: This is quite suspenseful!

Karolina: I am glad you are having fun with this technical stuff. We are just one step from completing the logical design. All we need is to connect [in Workbench] the entities with a *many-to-many* relationship tool (Figure 4).

To this end, select (click) the many-to-many (`n:m`) button (on the tool bar), then connect the entities by clicking each one after another. Notice that the button shows a Craw's Foot notation of the relationship. The diagram on the canvas uses an UML notation which can be set or changed via a menu command Model > Relationship Notation. By the way, I have been using different types of diagram formats in the past but I believe UML diagrams are the simplest and

most expressive. UML happens to be #1 choice for designing Object-Oriented (OO) applications. Since many database developers also develop applications, using Object-Oriented languages, it is convenient for them to express the model, using the Object-Oriented design language—UML (Naiburg, Maksimchuk, 2001).

As you can see, the three clicks have produced one extra entity (`Member_has_Hobby`) with two attributes annotated with a reddish icon ( ): `Member_mid` and `Hobby_hid`. Such an entity is referred to as an *Associative* entity. The entity's attributes bind members (instances of entity `Member`) with his/her hobbies (instances of entity `Hobby`). Now, I can say it officially, these attributes are referred to a *foreign keys* and they are required to take on values of the corresponding *primary keys*, this way supporting the so called *Referential Integrity*. The lines, connecting the entities show the relationships graphically. They are kind of redundant since the relationships are already established by the foreign key and their related primary keys. On the other hand, the lines help to better visualize the relationships.

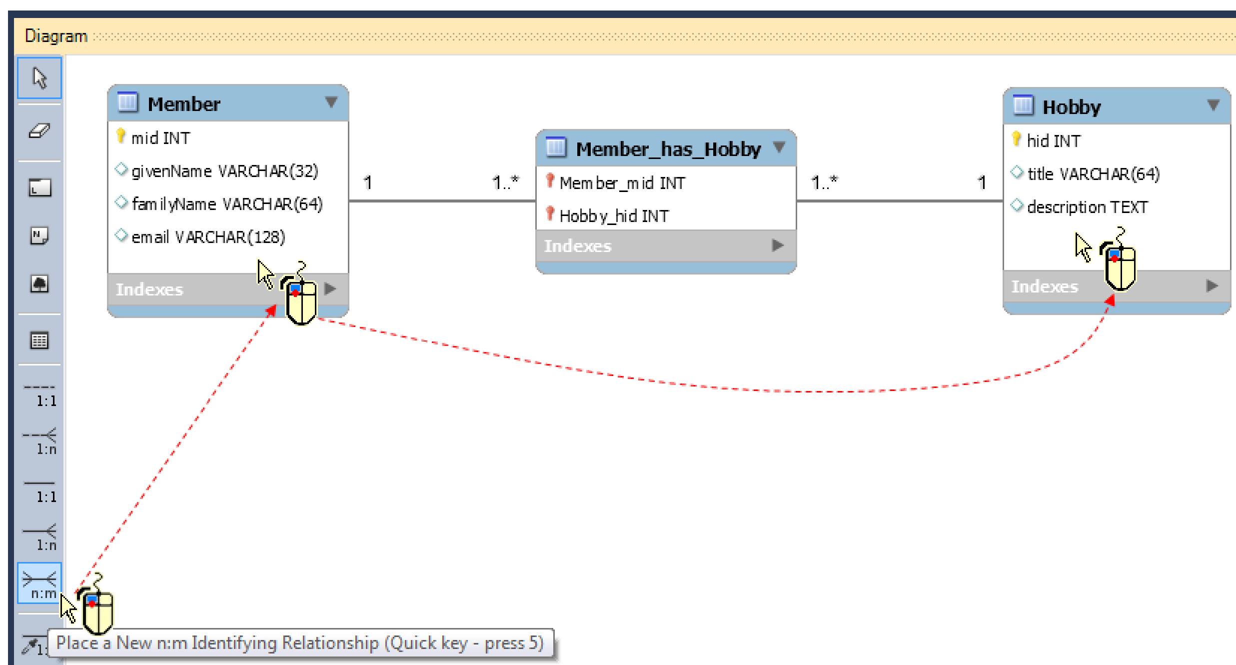

Figure 4 Add a many-to-many relationship in MySQL Workbench.

Emma: Wow, this is cool! I am impressed! By the way, I did identify the relationships but, unfortunately, in a wrong way.

Nick: I am impressed too! It is like having a Genie touch a magic wand and create something really beautiful.

Karolina: Indeed, it is like a magic and let me improve it a little by editing the new entity. First I recommend to change the name of the entity. Why don't we name it `ActiveHobby` and change the attribute names to simply `mid` and `hid` (same as their related primary key names). These changes will make our future queries more clear and compacts. Do you think we should better describe the new entity? What about adding some extra attributes?

Emma: Two extra attributes come to my mind: the starting date and estimated annual budget of the members' hobbies.

Nick: I was thinking about the starting date too.

Karolina: Great! Let's add the two attributes named as `startDate` and `annualBudget`. Please notice that I consistently follow the `OO` naming convention (entities—in a title case, and attributes—starting with a lower case). We select a `DATE` type for `startDate` and a default `FLOAT` type for `annualBudget`.

Nick: What is the largest value that can be expressed, using the MySQL's FLOAT type?

Karolina: A good question! Unless Mr. Elon Musk joins the site we should be OK with budgets up to `$99,999,999,00`. I you prefer a larger allowable maximum, we can select either `DOUBLE` (it defaults to the 16 significant digits and 4 decimal places) or `DECIMAL(m,d)` with sufficient number of significant digits (`m`) and decimal places (`d`).

Emma: I guess a FLOAT should work well.

Nick: Ditto!

Karolina: We can do it all by double-clicking the entity and making the changes in the table form (Figure 5).

It is also a good idea to annotate the relationship lines so that the diagram can be interpreted as a collection of S+P+O (Subject + Predicate + Object) statements. By the way, we can now express our original S+P+O model:

Member(0..*)-*has*-(0..*)Hobby

as:

Member(1)-*has*–(0..*)ActiveHobby(0..*)–*involves*–(1)Hobby.

This means that, if there exists an `active hobby` (an instance of entity `ActiveHobby`), it must belong to one and only one `member` (an instance of entity `Member`) and it also must involve exactly one `hobby` (an instance of entity `Hobby`). Going from the external entities, a `member` has zero, one or many `active hobbies` and a `hobby` gets involved in zero, one or many `active hobbies`. We need those zero constraints since there may be `hobbies` that are not chosen by any `members` and there may be `members` without any `hobby`.

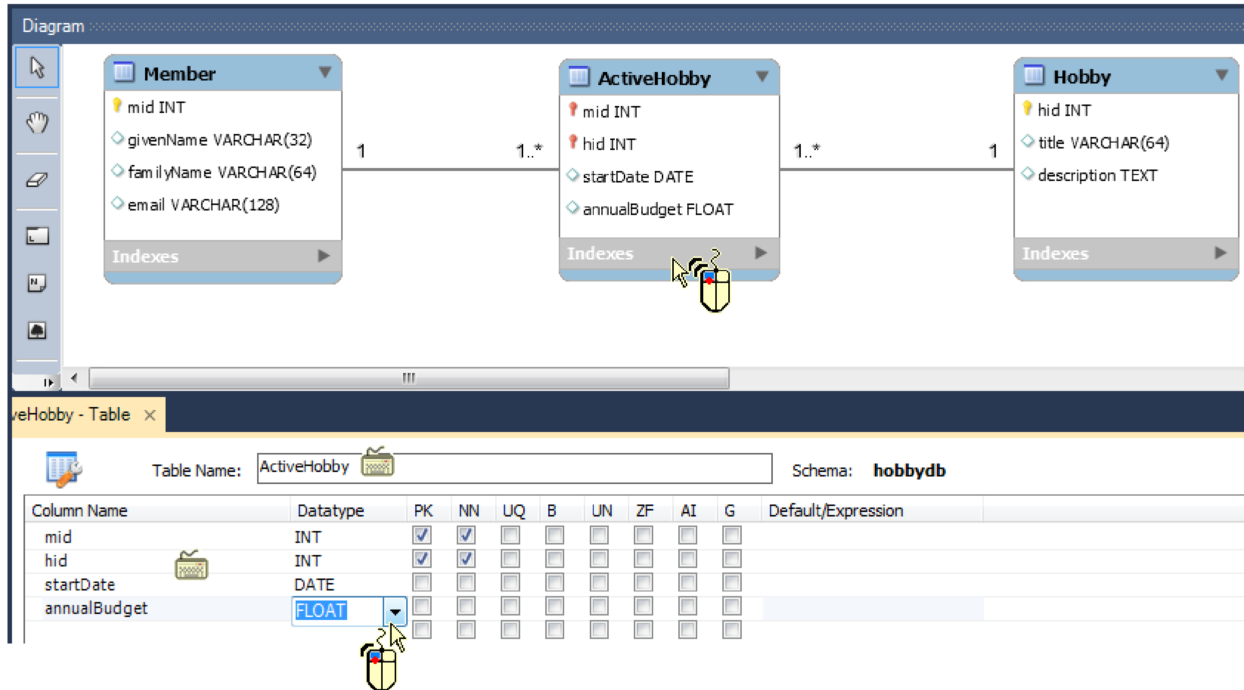

Figure 5 Customizing the associative entity.

Let's do our final touches to our diagram (relationship annotation and their cardinality constraints). We will use the text tool to add labels the relationship lines.

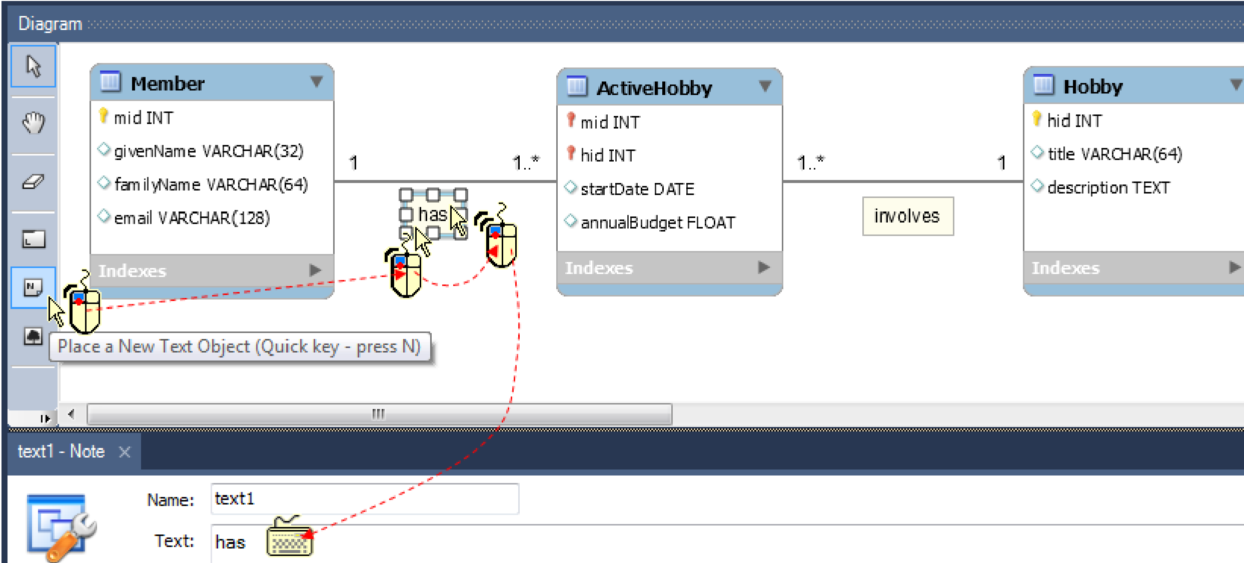

Figure 6 Annotating the *has* relationship.

As shown in Figure 6, you choose (click) the Text Object tool. Next, you click near the `Member` – `ActiveHobby` relationship line. MySQL Workbench adds a text box. You then double-click the box and change its name to [here] `has`. In a similar way, you add label `involves` to the second relationship.

Emma: I can now read the diagram as if it was written in English.

Nick: For me this is probably the best way to present a [rational] data model.

Karolina: Wait a moment, we are not done yet. We need to fix the cardinality constraints. At the **`ActiveHobby`** entity, we should have **`0..*`** rather than **`1..*`**. (Recall that we agreed that there may be **`members`** without **`hobbies`** and vice versa.) All we need is to double-click the relationship line, click the **`Foreign Key`** tab and uncheck box **`Mandatory`** (Figure 7).

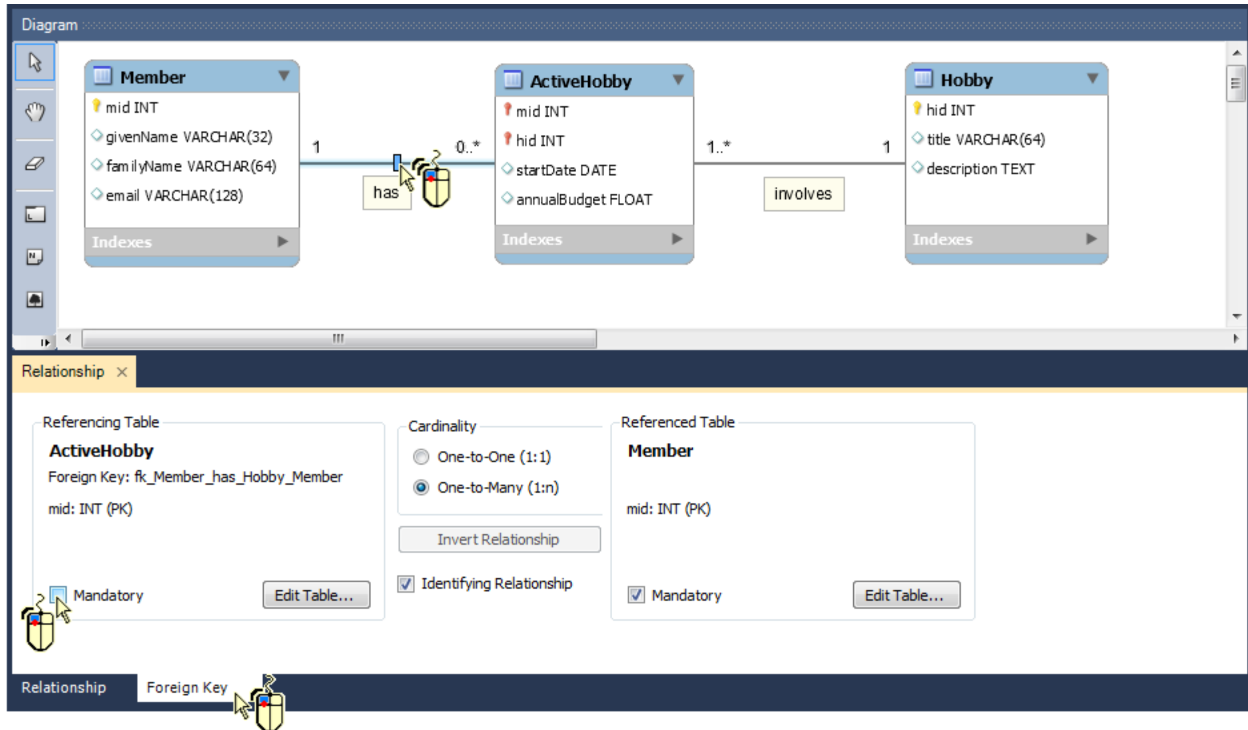

Figure 7 Relaxing the 1..* cardinality constraint by making it optional (0..*).

We will get our final design, when we repeat this procedure for the ***`involves`*** relationship.

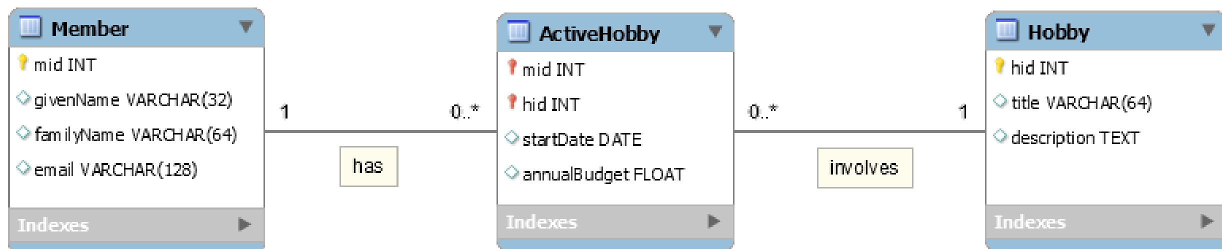

Figure 8 The final, relational data model.

FORWARD ENGINEERING

The logical data model is complete. MySQL Workbench refers to it as an enhanced model (EERD) since it can be used to generate a physical design (an SQL model) or to directly create the database and tables on MySQL Server.

Emma: Now what? What do we do with this gem? Can we now bring it to Access?

Nick: Yes, I can't wait to see it in Access.

Karolina: Hold your horses! I wish Access could import databases from diagrams like this. Access has lots of importing power but not with this. However, Access can create tables directly from an SQL schema and MySQL Workbench can help us generate the schema. All we need is to invoke properly a menu service: Database > Forward Engineer (Crtl+D) or File > Export > Forward Engineer SQL Create Script … (Shift+Ctrl+D). Let's try the latter. A series of steps with some options set on (Omit Schema Qualifier in Object Names, Generate Separate CREATE INDEX Statements, Disable FK Checks for Inserts, and Export MySQL Table Objects) will lead to the following script:

```
CREATE TABLE IF NOT EXISTS `Member` (
  `mid` INT NOT NULL,
  `givenName` VARCHAR(32) NULL,
  `familyName` VARCHAR(64) NULL,
  `email` VARCHAR(128) NULL,
  PRIMARY KEY (`mid`))
ENGINE = InnoDB;

CREATE TABLE IF NOT EXISTS `Hobby` (
  `hid` INT NOT NULL,
  `title` VARCHAR(64) NULL,
  `description` TEXT NULL,
  PRIMARY KEY (`hid`))
ENGINE = InnoDB;

CREATE TABLE IF NOT EXISTS `ActiveHobby` (
  `mid` INT NOT NULL,
  `hid` INT NOT NULL,
  `startDate` DATE NULL,
  `annualBudget` FLOAT NULL,
  PRIMARY KEY (`mid`, `hid`),
  CONSTRAINT `fk_Member_has_Hobby_Member`
    FOREIGN KEY (`mid`)
    REFERENCES `Member` (`mid`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_Member_has_Hobby_Hobby1`
    FOREIGN KEY (`hid`)
    REFERENCES `Hobby` (`hid`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;
```

Code 1 The generated SQL script.

Emma: This is amazing: such a complicated code was created from our diagram. I have just one question. Could you explain why this procedure is called “Forward Engineer”?

Karolina: Certainly! In the database context, Forward Engineering is the process of transforming an ERD model into an executable SQL script (McLaughlin, 2013). A graphical model is transformed into an SQL model that can be used directly to develop a physical database. I would also consider MySQL Workbench to be a light-weight CASE tool, where CASE stands for Computer –Aided Software Engineering (Wikipedia, 2017). It captures quite a significant portion if the database life cycle and help in transition from the logical to physical design.

Nick: I also have a question. Can we run this script now in Access?

Karolina: We have a MySQL executable script but it will not work [directly] in Access. We need to get rid of a few pieces: “`” (grave accent),”`ENGINE = InnoDB`”, “`IF NOT EXISTS`”, “`ON DELETE NO ACTION`”, and “`ON UPDATE NO ACTION`”. They all can be eliminated in any text editor by using a Search and Replace command. To further simplify the script also segments “``CONSTRAINT `fk_Member_has_Hobby_Member` ``” and “``CONSTRAINT `fk_Member_has_Hobby_Hobby1` ``” can be removed. Such a cleanup will produce three statements that can be executed in Access’s SQL-Query panel:

```
CREATE TABLE Member (
  mid INT NOT NULL,
  givenName VARCHAR(32) NULL,
  familyName VARCHAR(64) NULL,
  email VARCHAR(128) NULL,
  PRIMARY KEY (mid)
);

CREATE TABLE Hobby (
  hid INT NOT NULL,
  title VARCHAR(64) NULL,
  description TEXT NULL,
  PRIMARY KEY (hid)
);

CREATE TABLE ActiveHobby (
  mid INT NOT NULL,
  hid INT NOT NULL,
  startDate DATE NULL,
  annualBudget FLOAT NULL,
  PRIMARY KEY (mid, hid),
  FOREIGN KEY (mid) REFERENCES Member (mid),
  FOREIGN KEY (hid) REFERENCES Hobby (hid)
);
```

Code 2 The simplified SQL script (meeting Access requirements).

CREATING THE TABLES IN ACCESS

Assuming an empty Access database (`HobbyDB.accdb`) is already open, the team led by Karolina will add tables and data to this database.

Karolina: If we started creation of the database in Access (as shown in Figure 3), we would have to physically add table `ActiveHobby` along with its primary and foreign keys. We would then open the Relationship diagram, add the three tables and connect the primary keys to their respective foreign key along with setting the referential integrity. Instead, we will use the SQL-Query panel and execute our CREATE TABLE statements, one at a time.

On the CREATE toolbar, select Query Design command, close the Show Table dialog window and select the SQL View option. Next, copy the first CREATE TABLE statement from (Code 2) and paste it into the Query tab. Finally, click the Run button to add this table to the database. The table should be show on the right, in the Navigation panel. Save the query as `CreateTableMember` (Figure 9).

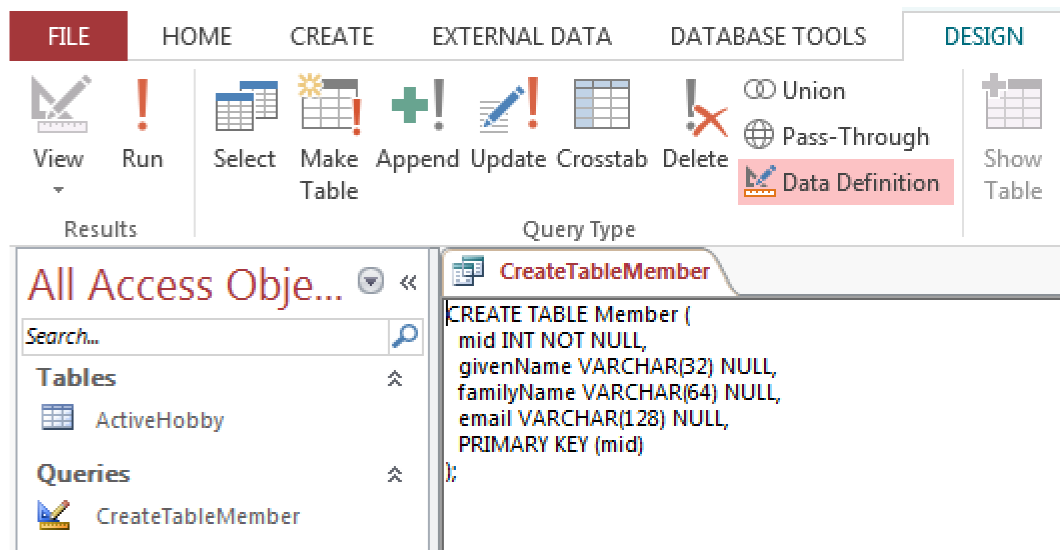

Figure 9 Add table Member to the Access database.

Repeat this procedure for the other two tables. The complete database schema, along with its Relationships diagram, is shown in Figure 10.

Nick: It would be great if we populated the tables with data and tried to explore them a little!

Emma: I already have some data in Excel. I can add the third table and maybe we could then copy the Excel rows into the Access table records.

Karolina: This is an excellent idea. Having identically structured table in Excel we could import them into Access or simply copy in Excel and past them in Access.

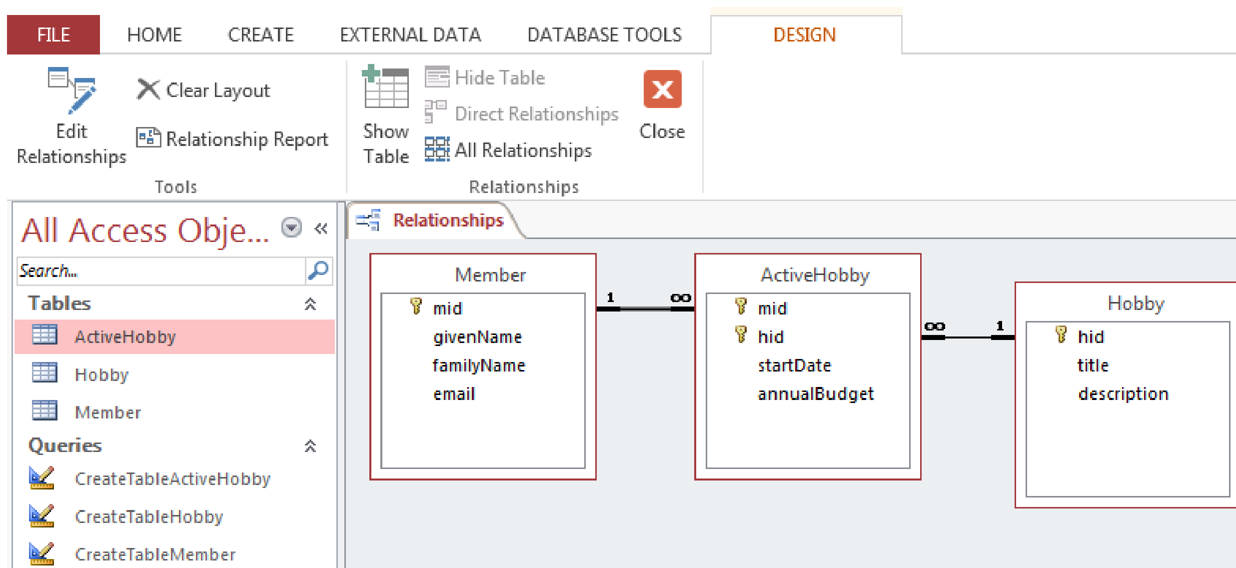

Figure 10 The HobbyDB database schema in Access.

Emma: Are the tables shown in Figure 11 good enough? Just in case I named them as `Member`, `Hobby`, and `ActiveHobby`.

Member

mid	givenName	familyName	email
1	Pat	Patyk	pp@hobbysta.org
2	Don	Donski	dd@hobbysta.org
3	Eve	Evening	ee@hobbysta.org
4	Tom	Tomas	tt@hobbysta.org
5	Kim	Kimura	kk@hobbysta.org

Hobby

hid	title	description
1	Golf Balls	Collect golf balls.
2	Coke	Coca Cola memarobilia.
3	Post Cards	Collect post cards.
4	Stamps	Collect postage stamps.

ActiveHobby

mid	hid	startDate	annualBudget
1	1	05-14-2016	$500
1	2	07-25-2016	$750
2	1	07-28-2016	$1,200
3	1	08-03-2016	$200
3	2	08-22-2016	$350
3	4	09-16-2016	$800
4	4	10-14-2016	$100

Figure 11 Data for database HobbyDB in Excel.

Nick: It all looks great! I can take care of the data transfer. Since your Excel workbook is already open, I will copy the table rows in Excel (Figure 11) and paste them to the Access tables.

Karolina: You guys are a great team. Now, as we have a complete database in Access (Figure 10), Nick can play with his queries. Why don't you write your first query to show members and their hobby information for those who declared their hobbies before August 15, 2017 and have budget above $400?

Nick: Right away. We will need the three tables with criteria set for `startDate < #08/15/2016#` and `annualBudget > 400`. Figure 12 shows the setup in the QBE panel.

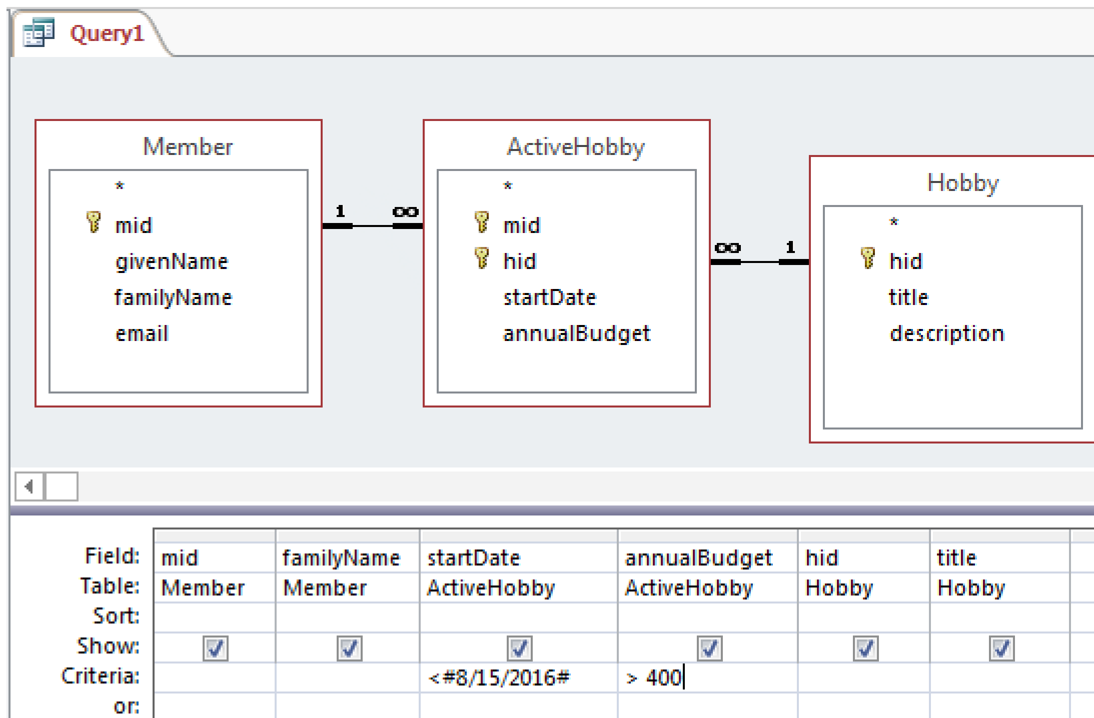

Figure 12 An Access query example.

Karolina: Very good! I guess it will do. Nick, you can now add more data and start playing with the database. Remember, you can always add more entities and/or modify the existing ones in the logical model and follow up for the Forward Engineering procedures to generate SQL script. Do you have any questions?

Nick: I am good for now! Thank you Karolina and Emma for your help.

Karolina: You very welcome! It was a pleasure doing it all with you.

Emma: It was fun working with you. Thank you for this great opportunity to learn more about designing databases. Now, I can better understand how I will try to design my own database for my domain, I mean, for an Entertainment application.

FINAL REMARKS

Whether one follows a traditional software development life cycle (SDLC) or rapid development techniques (RAD), one will always run into the design phase (Gupta, et al., 2011). This phase (including its logical part) is inevitable and of an outmost importance (Smith, 2004).

Unfortunately, Microsoft Access, while being incredibly capable and rich tool, fails to fully support the logical design (Letkowski, 2015).

A question should be raised, especially in the academic context. Is it really important for business students to learn about the logical database design? Having taught this subject to different audiences (Business, Computer Science, Information Technology, Master Degree) for more than 30 years, the author strongly believes that the logical data modeling is one of the most critical knowledge components and skills one would expect from a business graduate. One of the important reasons is that business (accounting, finance, management, marketing, etc.) graduates (excluding Computer related majors) are more likely in the future to act as domain (subject-matter) rather than application development experts. Needless to say that the logical design comes very close to data modeling which has become recently one of the most demanding skills sought by businesses (Blaha, 2014). Demands for this skill are particularly fueled by the increasing popularity of business analytics (Chen, et al., 2012) and proliferation of data structures that have to a great extend successfully competed with the relational databases (Letkowski, 2017).

This paper shows how to realize a complete SDLC by complementing Microsoft Access's apparatus with MySQL Workbench's database design tools. A complete analysis of database topics that should be taught in bossiness curricula is beyond this paper's scope. However, the author strongly believes that the data modeling, in general, and logical database design, in particular, should become a solid part of the business graduates' expertise. This can be accomplished by adopting both MySQL Workbench and Microsoft Access into the business productivity software classes. If one is convinced that the business student should focus only on critical topics, such as database design, management, and SQL queries, then MySQL Workbench along with MySQL Server (both free—kudos to Oracle), can do it all superbly.

As a follow-up of this paper, the author intends to develop a complete set of notes for teaching the vital database topics to business students, utilizing MySQL or a similar database, more fully supporting the SDLC.

REFERENCES

Blaha. M. (2014). Data Models Have Many Benefits. Here Are 10 of Them. Retrieved from: http://www.dataversity.net/data-models-many-benefits-10/ (mirrored at: http://mars.wne.edu/~jletkows/conference/2017/nedsi/Blaha2014.pdf)

Chen, H., Chiang, R. H., Storey, V. C. (2012). Business Intelligence and Analytics: From Big Data to Big Impact. MIS Quarterly Vol. 36 No. 4/December 2012.

Date, C. J., (2000). An Introduction to Database Systems, 7th edition, Addison Wesley Longman, Inc.

Elmasri, R., Navathe, S. B. (2004). Fundamentals of Database Systems. Boston: Pearson / Addison Wesley.

Gupta, P., Mata-Toledo, R., Monger, M. (2011) Database development life cycle. Journal of Information Systems & Operations Management 2011, Vol. 5 Issue 1.

Letkowski, J. (2014). Doing database design with MySQL. Journal of Technology Research. ISSN Online: 1941-3416 (http://www.aabri.com/jtr.html) Volume 6.

Letkowski, J. (2015). Challenges in database design with Microsoft Access. Academic and Business Research Institute Conference, Orlando 2015, January 1-3.

Letkowski, J. (2016). Complementary instructions and resources to accompany this paper. Retrieved from: http://mars.wne.edu/~jletkows/conference/2017/nedsi/howto.html (mirrored at: http://letkowski.us/conference/2017/nedsi/howto.html)

Letkowski, J. (2017). Beyond Relational Databases. IACB,ICE, & ISEC Conference Proceedings, 2017 Maui, Hawaii, USA, January 1-5 2017, p.192-1, http://cibusinesshawaii.com/.

McLaughlin, M. (2013). MySQL Workbench: Data Modeling & Development. McGraw Hill Education and Oracle Press.

Silberschatz, A., Korth, H. F., Sudarshan S. (2001). *Database System Concepts*. Boston: Osborne/McGraw-Hill.

Smith, A. M. (2004) How to Learn Data Modeling. Retrieved from: http://tdan.com/how-to-learn-data-modeling/5181 (mirrored at: http://mars.wne.edu/~jletkows/conference/2017/nedsi/Smith2004.pdf)

Wikipedia (2017). Computer-aided software engineering. Retrieved from: https://en.wikipedia.org/wiki/Computer-aided_software_engineering