Databases And Advanced Data Techniques (CM3010)

Course Notes

Felipe Balbi

April 20, 2021

Contents

Week 1

Key Concepts

- Find, describe and evaluate sources of data
- Understand different forms in which data may come
- Evaluate data-related access and reuse rights

1.005 Reading list

- Chen, P. 'The Entity-Relationship Model – Toward a Unified View of Data', ACM Transactions on Database Systems 1(1) 1976, pp.9–36.
- Codd, E. 'A relational model of data for large shared data banks', Comms of the ACM 13/6 1970, pp.377–87.
- Codd, E. 'Normalized data base structure: a brief tutorial'. In Proceedings of the 1971 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control (SIGFIDET'71). Association for Computing Machinery, New York, NY, USA (1971) pp.1–17
- Date, C.J. Database Design and Relational Theory. (Healdsburg, CA: Apress, 2019) Chapter 4. FDs and BCNF (informal)
- Härder, T and A. Reuter 'Principles of Transaction-Oriented Database Recovery', ACM Surveys, 15/4 1983
- Katie Rawson and Trevos Muñoz, 'Against Cleaning' from Matthew K. Gold and Lauren F. Klein Debates in the Digital Humanities, 5 (University of Minnesota Press, 2019).
- Lewis, D. CO2209 Database systems

1.101 Where does data come from?

Data can come from different sources:

New Data created for the sole purpose of the current application

Pre-existing Data data that already existed prior to the application being created. Perhaps it's internal *legacy* data, or it's external data that can be acquired from another supplier.

When it comes to new data, we can take different approaches:

Adding data on-demand For example, a hairdresser has bookings with clients. Either of these appointments is a new datum that gets added to the database *on-demand*, i.e. only the customer makes an appointment.

Bulk data entry Some systems can't afford to have only parts of the data available. In such cases, we can either pay for data entry services or rely on some form of crowd-sourcing.

Pre-existing data Whenever we have pre-existing data, it usually needs to be manipulated somehow in order to fit the new system. Some forms of data manipulation are:

Extraction data may already be in a spreadsheet or database and needs to be recovered, or extracted from the original source.

Conversion data may need to be converted into a new format or structure in order to fit new requirements.

Cleaning data may contain erroneous or unnecessary information. These need to be removed in order to prevent problems.

External sources of data are interesting because they amortize the cost of data entry or quality checks. When data is *purchased* from a supplier, it comes pre-cleaned and in a format that's easy to consume. Moreover, we can also have the opportunity of acquiring data produced by experts in a given field.

Conversely, when we acquire data from an external source, we relinquish control over the quality of the data and its structure. The data may also be incomplete and/or ambiguous from our point view; i.e. the level of detail to which a particular piece of information is encoded may be different from what we need. As a final concern, there may be concerns of trustworthiness with regards to the data.

1.103 Ordering some data: What's on the menu?

- Post 1: Trevor Munoz, 'What IS on the menu'
- Post 2: Trevor Munoz, 'Refining the problem'

1.105 What does your data look like?

When modelling real-life data, we must consider what sort of information is necessary for the application.

To motivate the problem, we look at the example of a book. The data required for a book may be:

Type	Book
Weight	557g
Height	172mm
Colour	Red and Green
Title	Gardener's Calendar
Authors	Thomas Mawe, John Abercrombie
Date	1803
Edition	17^{th}

Some questions arise when it comes to which form of e.g. the title to store. From the point of view of finding it in a shelf "Gardener's Calendar" is enough, from the point of view of comparison against other similar titles, a long form may be required.

1.201 Bringing data sources together

- Linked Jazz
- Pratt Institute, How Mapping Relationships Between Jazz Musicians Elevates Unsung Histories

1.203 Licenses, sharing and ethics

In academic and government circles, it's common to make data as openly available as possible. That, however, doesn't apply to all parts of government of commercial world. There are legal restrictions regarding the use of data which need to be considered.

The Linked Open Data Cloud project produces a graph of all the data openly available published in the Linked Data format. Considering the size of the graph which contains but a subset of all openly available data, the question to ask is *Why is so much data being shared for free if information is so valuable?*

To put into perspective, a furniture catalog from any given furniture company will contain many details about every item: price, sizes, materials, photos. In principle, the furniture could be copied from information that can be gathered from catalogues and manuals. However, the furniture company needs their products to be easy to find if they want to sell them. The same argument can be used for many other industries: music industry, electronics, streaming services, etc.

To summarize some of the reasons to share open data:

- To drive sales
- For the common good
- Contract requirements

- Interoperability

Conversely, here are some reasons **not** to share open data:

- Restrictions on source data
- Control of use
- Value of the data

1.204 Licensing

- Alex Ball, How to License research data

Week 2

Key Concepts

- Find, describe and evaluate sources of data
- Understand different forms in which data may come
- Evaluate data-related access and reuse rights

1.301 What shape is your data? Introduction

Data is structured in some form, and we have to be concerned about that. There are different *levels* of structure which can be considered:

Programming Languages Data types (`float`, `int`, `double`, etc) impose a certain structure to the data.

Data Models Relations between different data. Think databases.

Data Serialization Data formats used for transmission using e.g. a network connection.

Exchange Protocols Some form of standardization for information exchange using e.g. Unix Sockets, Named Pipes, shared memory or similar methods.

User Interfaces Data is user interfaces is structured in a way that's comfortable for humans to consume.

Some of the *shapes* of data we will deal with are:

- Tables
- Trees
- Graphs
- Media (raw data)
- Documents & objects

Table 1: Sample Table

Food	**Water (g)**	**Fat (g)**
Avocado	72.5	19.5
Butter	14.9	82.2

1.302 What shape is your data? Tables

A table has cells with a number of rows and columns. In our case, every row represents a *thing*. Each column represents a type of information about that *thing*. Table 1 shows an example of such a table:

Tables are easy to understand and structure. They're also very direct in how they communicate information. Tables are very important to Relational Databases. However, they're not very good at communicating or structuring data that branches or has hierarchy. A better suited representation for such data would be Trees.

1.304 What shape is your data? Trees

A tree is Computer Science is based on the metaphor of a real tree. Figure 1 below shows an example of a simple tree structure. Every tree has a root node, every branch in the tree has a path to the root.

Some vocabulary is necessary, the following refers to the tree from figure 1.

- The *root* of the tree is node a
- Nodes e, g, i, k l, m, n, o, p, r, s, and u are *leaf nodes*
- Node f is a parent of l, m, and n
- Nodes l, m, and n are children of node f
- Nodes a, and b are ancestor of nodes l, m, and n
- Nodes i, j, and k are sibblings
- Nodes b, c, d, h, and others are internal nodes

1.306 What shape is your data? Other

One limitation of trees is that each and every node can have a single parent node. What happens when we need to represent a node with more than one parent? Perhaps we can reach the same child node through different paths. If we were dealing with a filesystem, whenever we add a *symbolic link* to a file, we would break the representation of the filesystem as a tree. It's clear we need another structure to represent these sorts of structures. That structure is a graph.

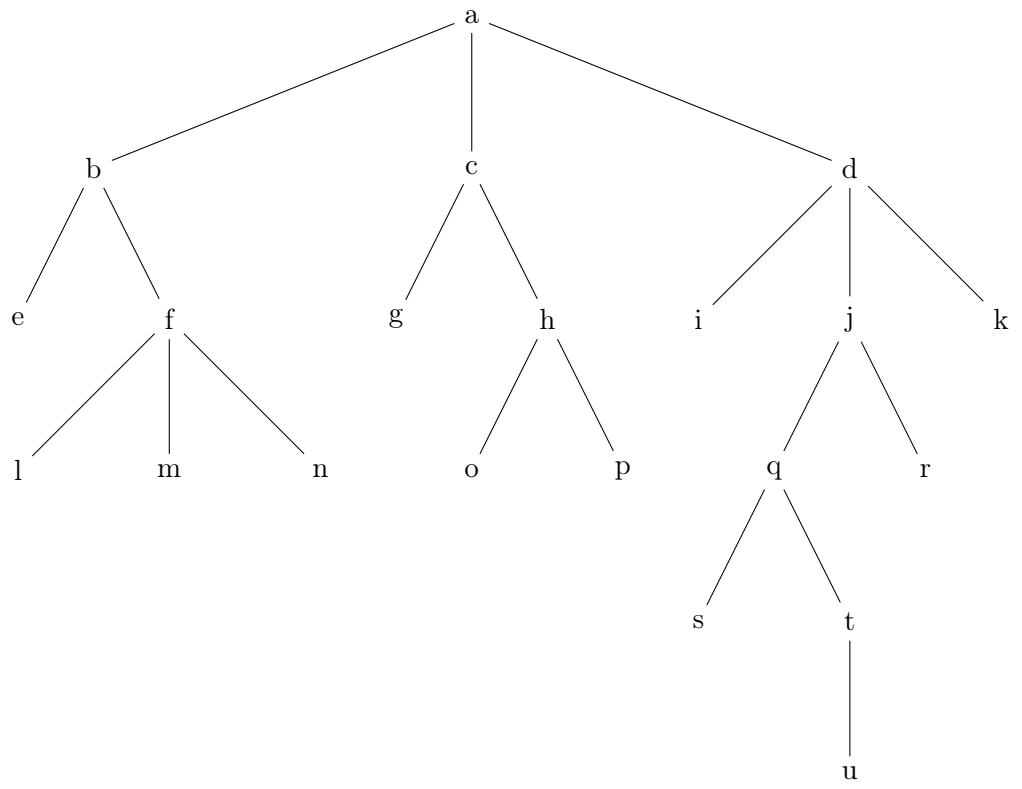

Figure 1: Sample Tree

Figure 2 shows a K_8 complete graph. A complete graph is that where each vertex is connected to every other vertex. The vertices in a graph could be web pages and the edges could be links between them, or perhaps each node is a file with the edges being a filesystem path.

Blobs are *raw* data representations without a perceivable structure. Raw sound samples fall into this category. Features are pieces of information derived from blobs, for example the sample rate from a raw audio file.

Table 2 shows a summary of the structures discussed so far:

Table 2: Summary of structures

Structure	**Description**
Table	General purpose
Tree	Heterogenous and hierarchical, structured data
Graph	Heterogenous, non-hierarchical, structured data
Blobs	Inaccessible data for storage
Features	Searchable information derived from blobs
Documents	Rich, but not interrelated data

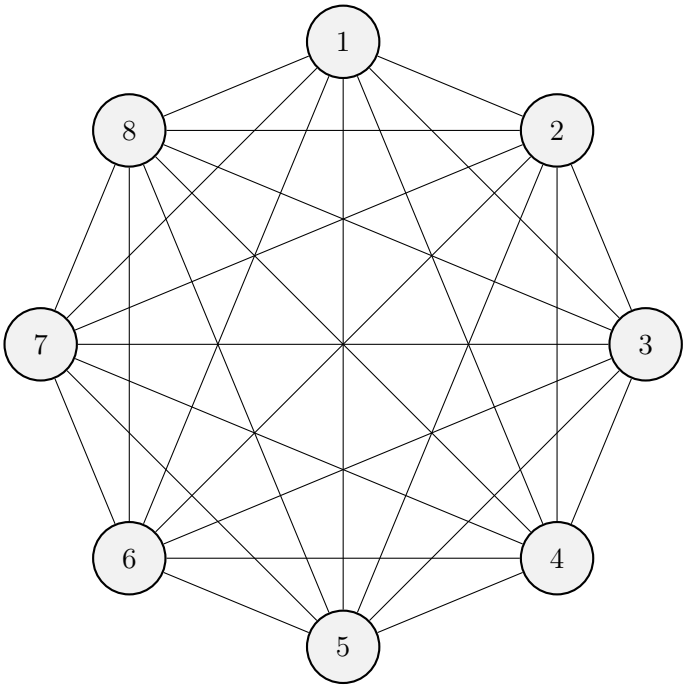

Figure 2: Complete graph

1.402 Further reading

- Katie Rawson and Trevos Muñoz, 'Against Cleaning' from Matthew K. Gold and Lauren F. Klein Debates in the Digital Humanities, 5 (University of Minnesota Press, 2019).

Week 3

Key Concepts

- Create and explore relational databases using SQL
- Design a database using Entity/Relationship diagrams
- Explain core concepts of relations and relational theory

2.001 Welcome to Relational Databases

A relational database implements the Relational Model. By model we mean by model is that it serves as an abstraction to the complex real-world. Usually we will abstract data.

A relational database uses tables to represent data. Relation, in this case, as defined by E. F. Codd., is a set of tuples $(d_1, \ldots, d_n)$ where each element d_j is a member of D_j

Relational Databases are an implementation of Relational Algebra, a theory for modelling data and defining queries on such data.

The following, summarizes a set of rules for Relational Databases.

1. Everything is a **relation**
    - All operations use the relational model
    - All data is represented and accessed as relations
    - Table and database structure is accessed and altered as relations
2. The system is unnaffected by its implementation
    - If the hardware changes
    - If the operating system changes
    - If the disks are replaced
    - If data is distributed

The Relational Model is not the same as the Entity Relationship Model. An ER Model helps us model concepts, usually as part of the design of a Relational Database.

SQL is a *partial* of the relational model.

2.101 Drawing a database I: Basic Entity-Relationship diagrams

An *Entity* is the thing we want to model, it must be uniquely idenfiable and it may have attributes. Of this, there are two subtypes:

Weak Entity its existence depends on the continued existence of other entities. For example, a customer's bank account depends on the existence of the account holder. This entity type is depicted in figure 3.

Strong Entity the one which is **not** weak. This entity type is depicted in figure 4.

Figure 3: Weak Entity Type

Figure 4: Strong Entity Type

An *Attribute* is an information that describes one aspect of an entity. Attributes can be characterised in various ways (described below and depicted in figure 5):

Simple vs composite A **simple attribute** is atomic or scalar (a simple integer or string). A **composite attribute** has internal structure that can be broken down into further attributes. For example, a *Date* attribute can be broken down into *day*, *month*, and *year*.

Single or multi-valued A **single-valued attribute** is one that won't change. For example a student is unlikely to have multiple full names at the same time. A **multi-valued attribute** is that which an entity can have multiples of it. It's depicted in a diagram with a double border line. An easy example is a phone number: a student can have multiple phone numbers.

Base or derived a **derived attribute** can be deduced from other attributes already present. They are depicted in a diagram with a dotted or dashed border line. An example would be someone's age can be computed from their date of birth. A **base attribute** cannot be deduced from other attributes.

Primary key an attribute that uniquely identifies an instance of the entity type. In a diagram, it's shown underlined.

A *Relationship* is a connection or dependency between two entities. Entities involved in a relationship are referred to as *participants*. A relationship is depicted as a diamond labelled with the name of the relationship. If one entity in the relationship is strong and the other weak, we draw the diamond with double line. Figure 6 shows the relationship types in a simple ER Diagram.

2.104 Speaking to databases I: Basic SQL

SQL has commands for manipulating structure, such as `CREATE`, `DROP`, `TRUNCATE`, `ALTER`; as well as commands for manipulating data, such as `INSERT`, `SELECT`, `UPDATE`, and `DELETE`.

To retrieve information from the database, we use the SQL `SELECT` command:

```
SELECT PlanetName FROM Planets;
```

We can add a constraint to this query:

```
SELECT PlanetName FROM Planets WHERE DayLength > 200;
```

To create a table in an existing database, we use the `CREATE` query:

```
CREATE TABLE Planets (
       PlanetName        CHAR(8),
       DayLength        INT,
       YearLength        INT,
       PRIMARY KEY (PlanetName)
);
```

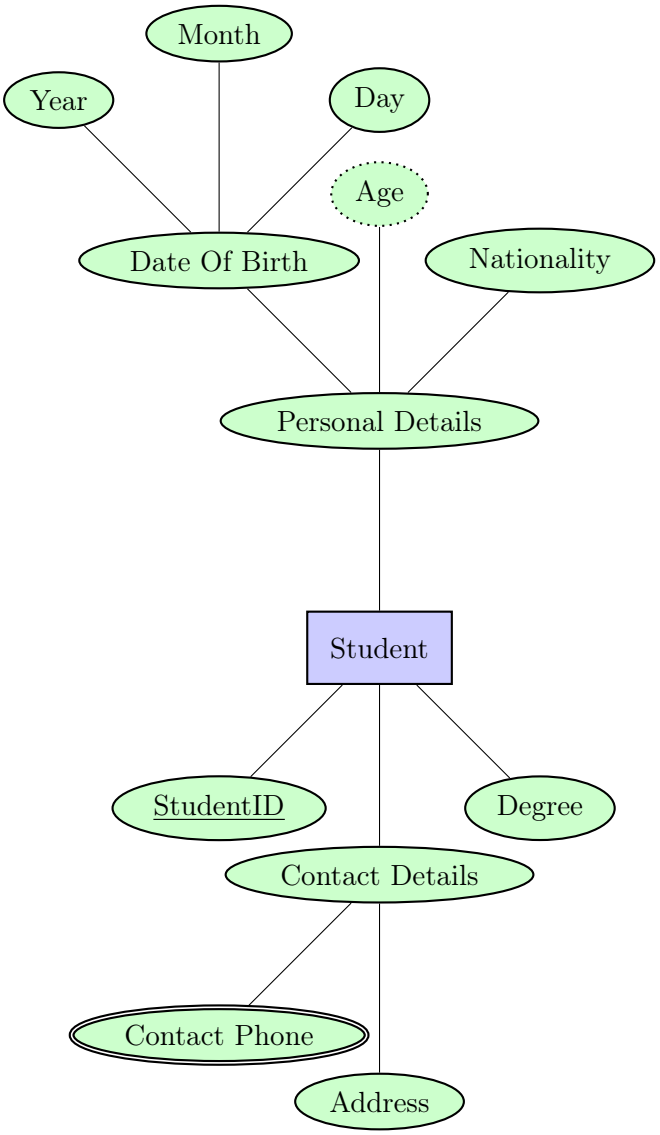

Figure 5: Attribute Types

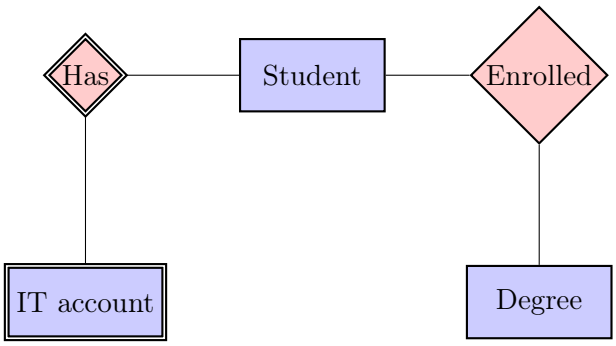

Figure 6: Relationships

Week 4

Key Concepts

- Create and explore relational databases using SQL
- Design a database using Entity/Relationship diagrams
- Explain core concepts of relations and relational theory

2.201 Introducing Joins

Joins are used to make queries that collect data from two tables. Figure 7 shows a diagram for the tables we will use to illustrate how to use Joins.

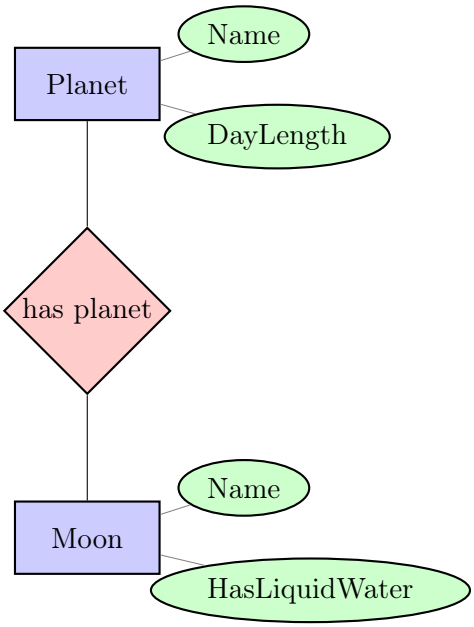

Figure 7: ER Diagram: Planets and Moons

The simplest form of a `JOIN`, called a *Cross Join* or a *Cartesian Join*, consists of simply listing all the tables we want to collect data from:

```
SELECT Lead.name, Rhythm.name,
       Bass.name, Drums.name
FROM   Lead, Rhythm, Bass, Drums;
```

The total number of results is the product of all entries in all tables, i.e. it essentially carries out a Cartesian Product of all the sets (tables) involved. E.g. if we have 3 Lead guitarists, 2 Rhythm guitarists, 5 Bass guitarists, and 7 Drum players, the total number of results will be $3 \cdot 2 \cdot 5 \cdot 7 = 210$ rows of results. Because the number of results grows so fast, we should carefuly consider our constraints in the `WHERE` clause to limit the results.

The example below is another way of executing a `JOIN`, called an *Inner Join*.

```
SELECT Planet.Name, Moon.Name, HasLiquidWater
FROM   Planet, Moon
WHERE  Planet.Name=Moon.HasPlanet
AND    DayLength < 11;
```

A more explicit version of the *Inner Join* is shown below

```
SELECT Planet.Name, Moon.Name, HasLiquidWater
FROM   Planet INNER JOIN Moon
ON     Planet.Name=Moon.HasPlanet
WHERE  DayLength < 11;
```

The *Outer Join* is another type of `JOIN`. Its syntax is shown below.

```
SELECT Planet.Name, Moon.Name, HasLiquidWater
FROM   Planet LEFT JOIN Moon
ON     Planet.Name=Moon.HasPlanet;
```

Figures 8, 9, and 10, give a visual representation of some what results will be returned for the joins.

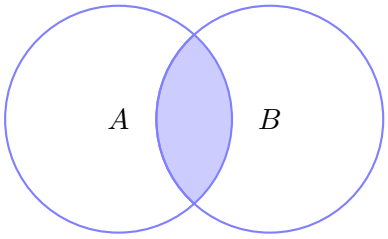

Figure 8: *SELECT * from A JOIN B ON A.B_id = B.id*

2.203 Drawing a database II: More about joins

Cardinality tells us how many rows in each of the tables participating in the join matches with how many rows on each of the table. It's often expressed in terms of a ratio, some of which are shown below:

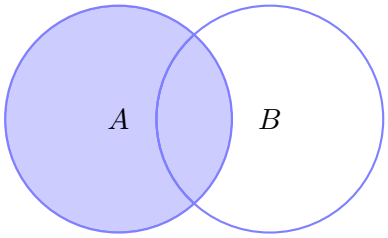

Figure 9: *SELECT * from A LEFT JOIN B ON A.B_id = B.id*

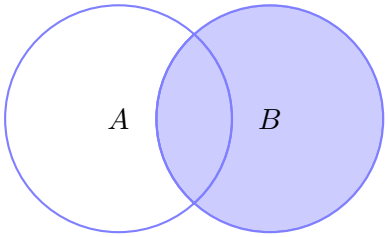

Figure 10: *SELECT * from A RIGHT JOIN B ON A.B_id = B.id*

1:n one row in table x joins with **zero, one or more** rows in table y.

In this case we want to use a Foreign Key by placing the primary key of the table x into the table y. Figure 11 shows a depiction of this case in an ER Diagram. If we want to show that at least one moon will be available, we use a double edge to connect the entity to the relation, as shown in figure 12.

In this situation, the Moon table would be declared like so:

```
CREATE TABLE Moons (
        MoonName   CHAR(20)
        PlanetName CHAR(10),
        Diameter   INT,

        PRIMARY KEY (MoonName),
        FOREIGN KEY (PlanetName)
        REFERENCES  Planets(PlanetName)
);
```

1:1 one row in x joins with exact one row in table y

This can be implemented as the case above, but we should consider implementing it as a single entity with attributes. Figure 13 shows an ER Diagram for this case.

The Project table would be declared like so:

```
CREATE TABLE Projects (
        Student VARCHAR(100)
        Title   VARCHAR(100),
```

```
4         Mark    INT,
5
6         PRIMARY KEY (Student),
7   );
```

m:n any number of rows from table x joins with any number of rows in table y

This is impossible to implement with the Relational Model. We must add a new entity/relation. A depiction of the ER diagram is shown in figure 14. The fix for this case is shown in figure 15, it consists of adding a *Link Table* to the model.

The link table is created as shown below:

```
1   CREATE TABLE TutorRole (
2         Student VARCHAR(100)
3         Tutor   VARCHAR(100),
4
5         PRIMARY KEY (Student, Tutor),
6   );
```

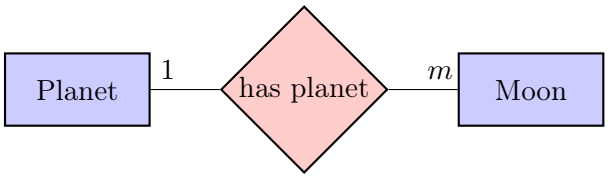

Figure 11: ER Diagram: Planets and Moons Cardinality $1 : n$

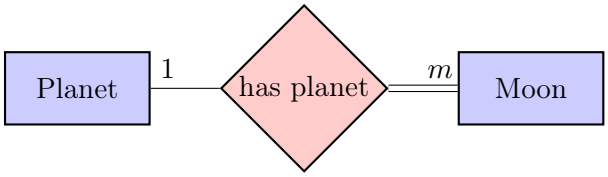

Figure 12: ER Diagram: Planets and Moons Cardinality $1 : n$, n at least 1

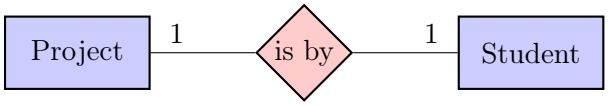

Figure 13: ER Diagram: Students and Projects Cardinality $1 : 1$

2.204 E/R diagrams summary

- Lewis, D. CO2209 Database systems. (London: University of London, 2016).

Figure 14: ER Diagram: Students and Tutors Cardinality $m : n$

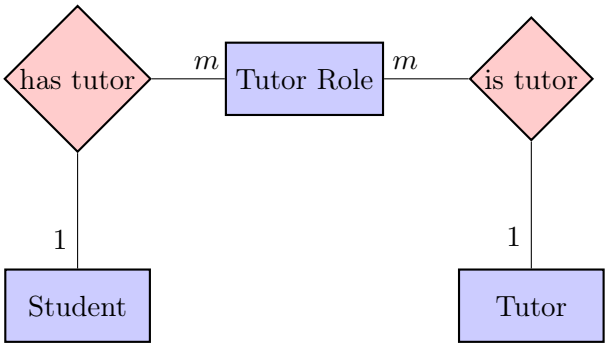

Figure 15: ER Diagram: Students and Tutors Cardinality $m : n$ with fix

2.301 Database integrity and the role of keys

By analyzing what could go wrong, we can design a database system that guarantees some error patterns won't happen. To motivate the discussion, we look at our planets and moons again. Assume we have entry shown below:

```
MoonName: Deimos
PlanetName: Mars
Area: 495
```

PlanetName could be mistyped as *Mers* or set to *NULL*. This planet doesn't exist, so our queries will produce erroneous results. *Area* should never negative, so we should disallow negative values. Some of these problems that can arise are detectable and preventable if we design the database for that.

2.303 Speaking to Databases II: SQL for joins and keys

Integrity Constraints can help us solve a few of the errors proposed before. Below we can find a list of common errors and their solution with integrity constraints.

Join fields must match We should use a `FOREIGN KEY`. A subsequent `INSERT` with wrong value will fail.

One some values of a field are valid Use `CHECK` column constraint.

Tables values should not be inconsistent Avoid repeating information in a database. `PRIMARY KEY` guarantees uniqueness. Avoid storing calculated values.

Changes should not cause inconsistency Use `FOREIGN KEY` rules to enforce correct behavior (i.e. `ON DELETE CASCADE`).

Table values should not be inconsistent Remove **functional dependencies**.

2.402 Further reading

- Chen, P. 'The Entity-Relationship Model – Toward a Unified View of Data', ACM Transactions on Database Systems 1(1) 1976, pp.9–36.