CO130 - Databases

Prelude

The content discussed here is part of CO130 - Databases (Computing MEng); taught by Thomas Heinis, in Imperial College London during the academic year 2018/19. The notes are written for my personal use, and have no guarantee of being correct (although I hope it is, for my own sake).

Material Order

These notes are primarily based off the slides on CATe. This is the order in which they are uploaded (and I'd assume the order in which they are taught).

1. *Introduction.pdf*
2. *ER Modelling.pdf*
3. *ER to RM.pdf*
4. *Relational Algebra.pdf*
5. *Tutorial ER.pdf*
6. *Tutorial Translation.pdf*
7. *Relational Algebra.pdf*
8. *Functional Dependencies.pdf*
9. *Tutorial Relational Algebra.pdf*
10. *Normalisation.pdf*
11. *SQL.pdf*
12. *Data Definition.pdf*
13. *Data Manipulation.pdf*
14. *Advanced SQL.pdf*
15. *Functional Dependency Tutorial.pdf*
16. *Transactions.pdf*
17. *SQL Tutorial.pdf*
18. *Storage.pdf*
19. *Indexing.pdf*
20. *NoSQL.pdf*
21. *MapReduce.pdf*

Introduction

We use databases as it's more organised; hence it's easier to model and manage. It's more efficient, as it's fast to search, and update, and integration allows us to minimise data duplication. Concurrent (and therefore multi-user) access allows multiple people to access the database at the same time (will require some techniques).

Transactions are sequences of database actions that execute in a coherent, and reliable way - the classical properties are **ACID**. Consider the two transactions `T1`: $A = A - 100; B = B + 100$, and `T2`: $B = B - 100; A = A + 100$, we can observe ACID properties as follows;

- **atomicity** if one part of a transaction fails, the entire transaction fails

  on completion of `T1`, either $A' = A - 100$, and $B' = B + 100$, or $A' = A$, and $B' = B$, where the former is a successful transaction, and the latter is in the case of a failure.

- **consistency** transactions don't leave the database in an inconsistent state

  the sum of the balances must remain the same, such that $A' + B' = A + B$; we can also have more constraints such as keeping balances positive, or limiting the amount a transfer can do at once
- **isolation** transactions run as if no other transactions are running (may need to wait)

  given the two concurrent transactions `T1`, and `T2`, one has to be completed before the other can start
- **durability** results of successful transactions aren't lost on system failure

  the new values, A', and B' must persist if the transaction completes, even if the system fails (disk failure etc.)

A **Database Management System (DMBS)** creates new databases via a **Data Definition Language (DDL)**, which specifies the structure (**schema**). It also queries, and manipulates through a **Data Manipulation Language (DML)**. Examples of this include *PostgreSQL*, *MySQL*, *SQLite*, and can also fall under *NoSQL*, however SQL remains as the most widespread technology (as of writing this).

It lets us define, query, and manipulate databases with a high-levle declarative language (**Structured Query Language**). It's standardised by the ISO, but each DBMS implements its own variation of the standards, which may be costly if it's complex.

Relational Model

Consider the following model, represented as a table;

heading	`title:string`	`year:int`	`length:int`	`genre:string`
body	Gone with the Wind	1939	231	Drama
	Star Wars	1977	124	Science Fiction
	Wayne's World	1992	95	Comedy

We have the columns be the attribute, with the top row being the heading, and the rest being the body. The attributes are in the format `name:type`. The rows (of the body) are referred to as tuples. A relation is the heading as well as the body (the entire table). The heading is an **unordered set** of attributes, and an attribute is the name as well as the type (typically indivisible types). The body is an unordered set of tuples, and a tuple is the set of attribute values. The schema is for the entire relation is the name of the relation, and the heading, in this case, we'd have; `movies(title:string, year:int, length:int, genre:string)`, and a database is a collection of relations. A schema for a database is the schemas for all relations.

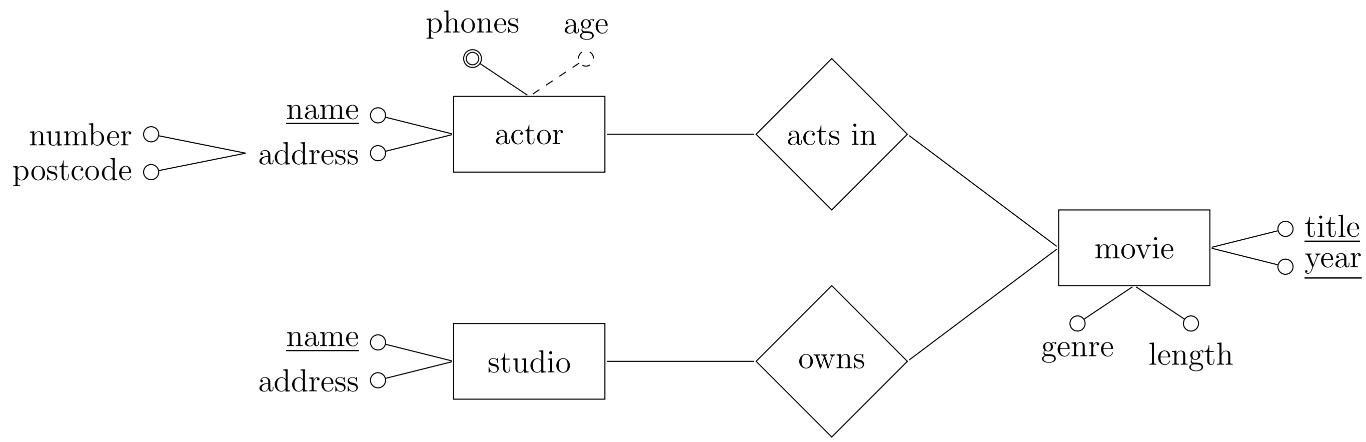
In mathematics, with a set of sets; $S_1, S_2, ..., S_n$, a relation R is a set of tuples $T_1, T_2, ..., T_n$, where $T_k \in S_k$, therefore $R \subset S_1 \times S_2 \times ... \times S_n$. As the idea of relations stems from set theory, it's important to note that the order in which we represent the attributes, and tuples in unimportant. In the Relational model we have **attributed** tuples, rather than **ordered**; R is the set of tuples $(A_1 : S_1 = T_1, ..., A_n : S_n = T_n)$, with $T_k \in S_k$. It's important to note that relations aren't 2-dimensional tables, even though it's more convineient to draw it on paper. We should instead consider them as a set of n-dimensional values, such that we have (`title:string`=StarWars, `year:int`=1997, `length:int`=127, `genre:string`=ScienceFiction), as a 4-dimensional movie value.

Entity Relationship Modelling

When a new database is being developed, it's important to try and model the real-world situation, instead of trying to refine it into an implementation, such as a relational model. In Entity-Relationship modelling, we try to create a diagram which represents the information needed for the database (the Entity Relationship Diagram). As there is no universally accepting notation for ER diagrams, we will use the following notation;

type	description	shape
entity sets	a set of distinguisable entries that share the same set of properties, can be physical (a room etc.), an event (flight, sale, etc.) - they normally correspond to nouns	rectangle
relationship sets	captures how **two or more** entity sets are related (e.g. owns, tutors), we can also have more than one relationship set between entity sets, and they can also have a relationship set on the same entity - they sometimes correspond to verbs	diamonds
attributes	properties of an entity; relationship sets can also have attributes, and primary keys are underlined	small circles

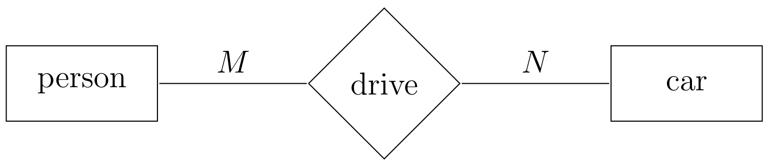
The movie example is represented below. Note that I've also extended it to contain different types of complex attributes. You can see that the address field is subdivided into number, and postcode, we have a multivalued attribute in phones, and a derived attribute in age (can be calculated from date of birth)



Cardinality Constraints

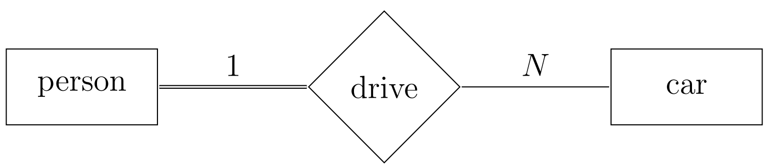
A relationship between two entity sets can be seen as one of the following (using the car example, where a person can drive N cars, and a car can be driven by M people);

- one-to-one $M = 1, N = 1$
- one-to-many $M = 1$
- many-to-many no restrictions

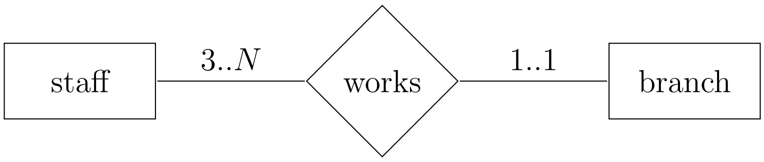


Participation

Using the same example, we can force participation, which means that all entities must participate in a relationship, with a double line. The example below, with the cars, suggests that everyone drives at least 1 car, and car can have at most 1 driver (note that it doesn't mean that every car has a driver).



Sometimes, instead of using double lines, E-R notation may allow explicit bounds. The below diagram suggests that staff work in exactly 1 branch, and that each branch must have at least 3 members of staff;



Fan, and Chasm Traps

See the diagrams in *ER Modelling.pdf* for examples.

We need to ensure that we do not allow ambigious paths between entities. For example, if we say `staff` works for `faculty` which operates `department`, if some member of staff works for Engineering, which operates both Computing, and EE, we can't follow a path from the staff member to their department. This can be solved by having the staff work for the department, which is operated by some faculty.

If we can't follow a path between two entities, we may need to add another relation between them to specify the relationship.

Weak Entities

Entities which cannot be uniquely identified by their own attributes are called **weak** entities, in contrast to **strong** entities which have primary keys. A weak entity has to be defined by a string entity. Weak entities are drawn as a double rectangle, and the relatioship to the strong entity is drawn as a double diamond. The key for a weak entity is formed by combining the primary key of the strong entity with attributes of the weak entity (denoted by a dashed underline). For example, a room in a building is a **weak** entity, with a room number, and can be uniquely identified with the name of the building combined with the room number.

ER to RM

Keys are used to join information when we do queries. THe primary key is the unique identifier of a tuple, and a foreign key is the primary key of **another** table.

Once we have an Entity Relationship Model, we can then map it to a relational schema. The schemas can then be refined with functional dependencies, and implemented with relational languages. A high level data model isn't the only concern for database design (considered a simpler aspect).

If we consider a strong entity set, with simple attributs, we can easily create a table from it (see the example, `movie`, drawn above).

```
movie(title, year, length, genre)

create table movie {
  title varchar(120),
  year int,
  length int,
  genre char(20),

  primary key(title, year)
}
```

Composite attributes are also fairly easy to represent, since we just flatten it, therefore store each of the sub-attributes as their own field - consider the actor example (note that if the primary key is composite, we mark all the sub-attributes as primary keys). This example also shows how multivalued attributes are stored - we create their own relation, and link back to the entity set with a foreign key constraint;

```
actor(name, number, postcode)
actor_phones(actorName, phoneID, other attributes)

create table actor {
  name varchar(60),
  number int,
  postcode varchar(10),

  primary key (name)
}

create table actor_phones {
  actorName varchar(60),
  phoneID varchar(10),
  ...
  primary key (actorName, phoneID),
  foreign key (actorName) references actor.name
}
```

The relational model doesn't allow us to specify derived attributes, and we're therefore expected to calculate them with queries.

We can also consider how many-to-many relationships can be represented (using the car example again). The first example is when a car can be driven by many people, and a person can drive many cars;

```
person(ID, other attributes)
car(regno, other attributes)
drive(personID, regno, other attributes)

create table drive {
  personID varchar(10),
  regno varchar(12),
  ...
  primary key (personID, regno),
  foreign key (personID) references person.ID on delete cascade,
  foreign key (regno) references car.regno on delete cascade
}
```

However, it's easier to represent a one-to-many relationship, as we can simply include the primary key of the "one" relation as a foreign key in the many (where a car can be driven by 1 person, and a person can drive many cars). The representation for a one-to-one is similar, but the foriegn key can be in either one, which is up to the designer;

```
person(ID, other attributes)
car(regno, personID, other attributes)

create table car {
    regno varchar(12),
    personID varchar(10),
    ...
    primary key (regno),
    foreign key (personID) references person.ID
}
```

In a weak entity, we include the primary key of the strong relation as a foriegn key, but also use it as part of the composite primary key for the weak entity. A delete cascade is also used, and finally the we use `not null` to ensure total participation;

```
building(name, other attributes)
room(no, buildingname, other attributes)

create table room {
    no varchar(120),
    buildingname varchar(50) not null,
    ...
    primary key (no, buildingname),
    foreign key (buildingname) references building.name on delete cascade
}
```

A multiway relationship can be mapped as several binary relationships, or we can generalise it to have the primary key consist of the relationship be composed of the primary keys of the many entity sets, and have the primary keys of the one entities be attributes. For roles, we map each role as a foriegn key attribute in the entity set;

```
movie(ID, other attributes)
sequelof(originalID, sequelID, other attributes)

create table sequelof {
    originalID int,
    sequelID int,
    ...
    primary key (orignalID, sequelID),
    foreign key originalID refereces movie.ID,
    foreign key sequelID refereces movie.ID
}
```

Extended Models

Newer models extend the classic model by supporting features from **object-oriented design**. We can look at how E-R models handle specialisation, or generalisation with **is-a heirarchies**. For example, we can specify that a `cartoon` **is-a** `movie`, which means that it inherits all the attributes of movie, but can have more attributes. and relations. Once this extends on multiple levels, we begin to form a heirarchy.

Relational Algebra

We can use **relational algebra** to construct new relations from existing ones, the operators include selection, projection, intersection, union, difference, and product. Consider the examples below, `a`, `b`, and `c` are all integers. Intersection, union, and difference will be omitted, since those are fairly self-explanatory.

R						$\pi_{\texttt{a,c,e}}(R)$			S			$\sigma_{\texttt{a>3\|b<5}}(S)$		
a	b	c	d	e	f	a	c	e	a	b	c	a	b	c
1	2	3	4	5	6	1	3	5	3	2	3	3	2	3
1	1	1	1	1	1	1	1	1	2	1	7	8	2	2
2	2	2	2	2	2	2	2	2	8	2	2			
1	2	3	4	5	8				1	2	9			

The syntax in use here for projection is $\pi_{\texttt{attributes}}(T)$, and likewise for selection it's $\sigma_{\texttt{condition}}(T)$. The former returns a relation with only the listed attributes, and the only takes rows which satsify the given condition. Neither will return duplicate rows. Here is an example of a Cartesian product;

R		S			$R \times S$				
a	b	c	d	e	a	b	c	d	e
1	2	1	2	3	1	2	1	2	3
3	4	4	5	6	1	2	4	5	6
		7	8	9	1	2	7	8	9
					3	4	1	2	3
					3	4	4	5	6
					3	4	7	8	9

We can take a natural join, where the resulting relation contains all the tuples that have matching attributes in R, and S. This is less common, and we'd typically use something closer to $R \bowtie_{R.\texttt{b}=S.\texttt{b}} S$. Note that in this case, they are the same, since that's the only attribute that overlaps;

R		S			$R \bowtie S$			
a	b	b	c	d	a	b	c	d
1	2	1	2	3	1	2	5	6
1	3	2	5	6	3	4	8	9
3	4	4	8	9	3	4	9	9
		4	9	9				

Attributes can also be renamed with the notation $\rho_{\texttt{new/old},\ldots}(R)$ notation, which is fairly self-explanatory.