CM12004 Databases
Condensed Notes

This revision guide is intended for internal use only and is not perfect, so if you find an error please report it to me via. email so I can improve this for years to come!

Dr. Andy Barnes
apb41@bath.ac.uk

Contents

Dr. Andy Barnes
apb41@bath.ac.uk

1. Database Systems

A database is an organised collection of data; from notepads to filing cabinets to enterprise-scale management systems in the cloud! The term database is broad but generally refers to a system which holds electronic data. The form of this data varies from one database to another but data generally falls into one of three categories:

Type of Data	**Description**
Structured	Data which falls nicely into a standardised format, the simplest form of structured data is that which can be stored in tables (columns and rows). Just like an Excel spreadsheet!
Semi-structured	Data which although uses a standardised format does not have a fixed schema. Examples of semi-structured data include web pages (html files), emails, JSON and XML documents.
Unstructured	Unstructured data does not use a standardised format and often contains data items of different native structures. For example, your operating system's file system contains audio files, images, video etc. These are all forms of unstructured data.

Most of the time when we discuss databases we are referring to structured databases. In the following sections we will look at relational databases which are a form of structured data database. However, it's important you are aware of alternative database approaches which can handle semi-structured and unstructured data, these are often referred to as NoSQL databases, a catch-all term for databases which aren't tied exclusively to using SQL.

2. Relational Databases

Relational databases are structured data stores which hold interrelated tables of data. Each tuple (row) in a table (relation) is allocated a unique ID called a key based on the attributes (columns) in the table and these keys can be used to maintain relationships between the rows in different tables. A good example of a relational database is that of a purchasing system; this could include tables such as Customer and Order. Each of these tables would have a key e.g. 'CustomerID' for the Customer table and 'OrderID' for the Order table. See below for an example of these tables.

Customer			
CustomerID	**Firstname**	**Surname**	**Email**
920384	Jason	Fernandes	JF93@gmail.com
...	...	...	...
102344	Melissa	Starcross	9kdslmm@aol.com

Order			
OrderID	**CustomerID***	**Item**	**Cost**
1	920384	Bottle of water	£0.99
...	...	...	...
2002	920384	Can of pop	£1.50

As shown, each table consists of rows and columns. Each row is a data item in the table (a tuple) and each column represents a real world attribute. The CustomerID attribute in Customer and the OrderID attribute in Order have been underlined to highlight that these have been chosen as the keys for the respective tables. Further to this, the Order table contains a CustomerID for each order, this indicates there is a relationship between the Customer and Order tables as they are linked through the CustomerID attribute. More on this later! But first let's discuss keys...

2.1 Keys

So far we've been introduced to the concept of a key, **a key is the unique identifier for a row in a given table**; for example CustomerID in the Customer table given above. However, there are a range of keys any database user and engineer must be aware of in order to make effective use of database systems. The key we've already been introduced to is known as a primary key, **a primary key is a unique identifier for a row in a given table.**

Usually primary keys consist of one attribute, however, sometimes they can be made of multiple attributes. For example, take the table below of Goods which can be supplied by a variety of suppliers. Try to identify a single-attribute primary key for this table. It can't be 'Title' because it repeats, it can't be 'Supplier' because it also repeats but it could be a combination of 'Title' and 'Supplier' (notice how both these attributes are underlined). Together 'Title' and 'Supplier' create a unique key which can be used to uniquely identify every row. **This is known as a composite key, a primary key consisting of multiple attributes**.

Goods		
Title	**Supplier***	**Cost**
Kettle	Hobbs	£15
Kettle	John Lewis	£14
...	...	...
Keyboard	Hobbs	£25

Next, we're going to look at how we maintain relationships between our tables. Refer back to the Customer and Order example from earlier, the CustomerID attribute is a primary key of the Customer table. However, the CustomerID attribute used in the Order table is not a primary key, instead it is known as a foreign key. **A foreign key is the primary key of one table referenced in another table**. This is why there is an asterisk after the attribute's name, this indicates it is a foreign key.

For another example let's add another table to the Goods example, below I've added a Supplier table with a primary key of 'Name'. This 'Name' attribute is referenced in the Goods table under a different name 'Supplier', despite the naming difference the Goods table 'Supplier' attribute is actually referencing the Supplier table. Therefore it is now known as a foreign key whilst still forming part of the composite primary key of the Goods table. This brings us to a special case of composite keys called compound keys.

Supplier	
Name	**Contact**
Hobbs	t@hobbs.com
...	...
John Lewis	j@jl.com

Compound keys are composite primary keys where at least one attribute in the composite key is a foreign key (reference to another table). In this case the Good primary key is a composite key as it contains multiple attributes but it is also a compound key as the 'Supplier' attribute is a foreign key to the Supplier table.

3. Entity-Relationship Diagrams

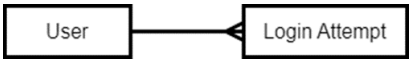
Entity-relationship diagrams (ERDs) as the name suggests described the relationships between entities in a domain. In database design ERDs describe the relationships between tables (entities). When designing a database system an ERD can prove vital in improving data consistency, reducing redundancy and improving maintainability of the resulting database.

3.1 Conceptual

In this section we will be looking over conceptual entity relationship modelling which is the first level in database design and should be the most high-level. To begin consider the scenario below:

In an online games sales platform users can attempt to login to the service by providing a username and password.

Here we have two entities User and Login Attempt (or Attempt) and these two entities hence have a relationship, a User can be related to multiple Login Attempts but each login attempt is only in reference to a single user. We can represent this relationship using crows feet syntax as follows:



In this simple example we have two entities and one relationship, this specific relationship is known as a one-to-many relationship, this use of ‘one’ or ‘many’ in the relationships is known as the cardinality. In general we can think of entities as tables in our database which are linked through relationships and foreign keys as shown in Section 2.1.

Not all relationships are one-to-many and there are two alternative relationships entities can have; these are one-to-one and many-to-many. However, these two relationships raise key questions regarding the databases’ design. For example, consider an addition to the scenario above:

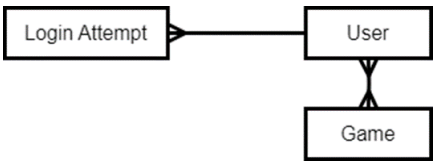
Each user is also affiliated with exactly one email address and each email address only one user.

Initially this seems simple enough. We can simply produce an ERD as given below, this shows a one-to-one relationship between User and Email Address. However, if we think about what this means in terms of table structures, does it really make sense to store email addresses in a separate table to the User details? In this case there is a clear argument for simplicity to store the email address in the User table. Hence the Email Address entity can be disregarded. This is a qualitative judgement to be made based on the scenario.

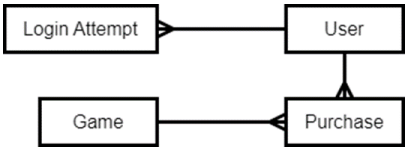
The second problematic relationship is many-to-many. To further expand the example above consider the following additional information:

Whilst each user can purchase many game titles, each game title can also be purchased by many users.

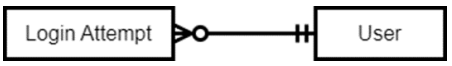
Here we reach the issue of a User having a many-to-many relationship to the Game entity, this is depicted below. Whilst this relationship at a conceptual level is not a problem, when we come to implement these relationships, we can run into several problems.



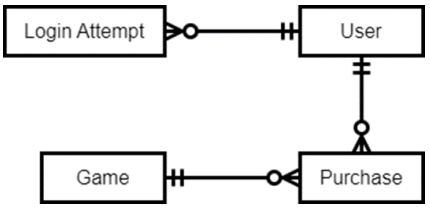
To resolve many-to-many relationships we often add what's known as a bridging table which points to the entities at each end of the many-to-many relationship, maintaining a 'bridge' between them, this is illustrated below using a new Purchase entity.



Now that we understand cardinality and relationship types we can now look at another aspect of relationships; their optionality. This means that we can define whether a relationship is optional for a given entity or not. For example, using the same example, a User can have multiple Login Attempts but every Login Attempt is associated with a User. This means that the relationship between User and Login Attempt is optional for the User but mandatory for the Login Attempt. We can depict this optionality as shown below:



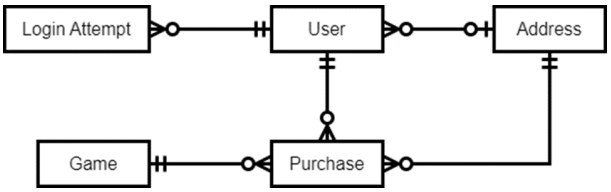
We can apply the same logic to the remaining relationships. Starting with User to Purchase, a User does not necessarily have to ever purchase anything and so the 'to-many' portions of this relationship is optional. However, every Purchase must be related to a User in some way so the 'one-to' portion of this relationship is mandatory. The same logic can be made for Game to Purchase, in that a Game may never be purchased but every Purchase references a Game. Below is the optionality approved ERD:



To show more examples of optionality consider the following addition to our scenario:

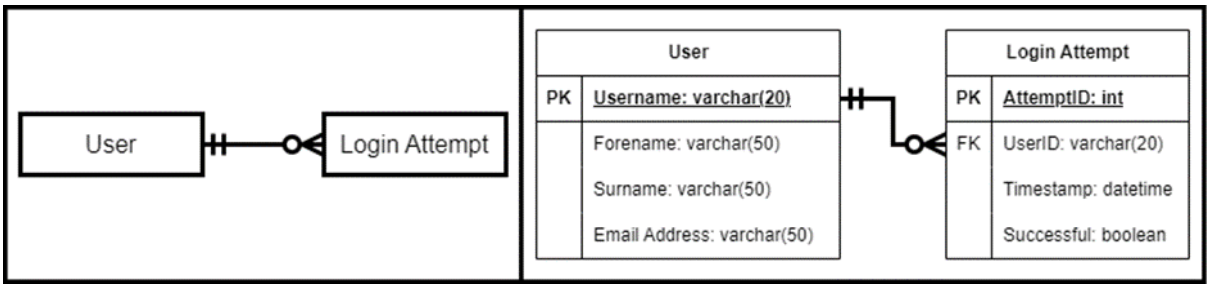
Whilst each user may or may not complete their home address details every purchase is made with a delivery address.

Here things get slightly complicated, so each User may or may not have one address and each address may be related to multiple Users. Further to this, every Purchase must be related to one address but each address may be used for multiple purchases. The depiction for this optionality is as follows:



3.2 Logical

Once a designer and database development team have settled and agreed a conceptual model for their data architecture it is then time to start the next step to produce a logical model of the domain. In order to do this we need to expand our conceptual design to include a few more details. Generally, moving from a conceptual to a logical model involves the addition of attributes, data types to those attributes, for example, on the left is the conceptual level User and Login Attempt entities and on the right are the logical level alternatives.



Firstly, notice the attributes are listed along with their relative data types, these data types are discussed more in section 3.2.1. Also added to some attributes are the terms 'PK' and 'FK' which relate to the terms Primary Key and Foreign Key. Despite the differences there are some similarities; for example, the names of the tables are not necessarily what will be implemented, having a table named 'Login Attempt' is not ideal due to the space and capitalisation which may not match the physical developers current naming standards but this is something resolved in physical modelling.

3.2.1 Data Types

The datatypes applied to attributes in database systems are relatively similar to that of most programming languages, with some caveats. A full list of datatypes are available in the Sqlite3 docs here: https://www.sqlite.org/datatype3.html. We will also be discussing these further in the lecture.

4. SQL

The language of relational databases is called SQL (Structured Query Language) and is intended to make defining, extracting and manipulating data easy without advanced knowledge of programming. It is a high level language and as such the commands and syntax are easily interpretable to anyone with a grasp of the English language. In this section we will highlight the key SQL commands you will need to both create and modify both the database and the data within it.

4.1 Data Definition Language

Data definition language is the part of SQL which allows you to define and modify the structure of the database. This is how we convert our ERDs and design into a physical database implementation.

4.1.1 Creating Databases

Despite most packages and libraries automatically generating databases for you when you attempt to connect it is sometimes necessary to create a database using SQL. To do this we use the command CREATE DATABASE, the following line of code creates a new, empty database called 'my_db'.

```
CREATE DATABASE my_db;
```

4.1.2 Creating Tables

An empty database is not very useful and so we need to populate our database with tables which can then hold data. To do this we can create a table based on our ERD designs using the CREATE TABLE command. The structure of the command is as follows:

```
CREATE TABLE test_table (
      <Attribute Name> <Data Type>,
      <Foreign Key Definitions (if any)>,
      PRIMARY KEY <Attribute Name>
);
```

This command allows us to create an empty table in the database with the given name (in this case 'test_table'), attributes as defined by entries of the attribute names and datatypes, we then provide any foreign key constraints before finishing off with our primary key definition. For example, let's take the Login Attempt entity from above and implement it below:

```
CREATE TABLE LoginAttempt(
      AttemptID INT,
      UserID VARCHAR(20),
      Timestamp DATETIME,
      Successful BOOLEAN,
      FOREIGN KEY(UserID) REFERENCES User(Username),
      PRIMARY KEY AttemptID
);
```

Here we see a table generated with four attributes of different types, a single foreign key constraint relating the LoginAttempt table to a User table and finally our primary key definition which indicates AttemptID is our primary key for this table.

4.1.3 Modifying Tables

There are several ways we can modify our table structures, we can add and remove columns, rename columns and even rename and delete tables.

4.1.3.1 Adding Columns

To add a column to a table we use the ALTER TABLE and ADD COLUMN commands as follows:

```
ALTER TABLE <Table Name> ADD COLUMN <Column Name> <Data Type>;
```

This allows us to 'modify' the table with the given <Table Name> by adding a new column with the name of <Column Name> and a type of <Data Type>.

4.1.3.2 Renaming Columns

To rename a table we again make use of the ALTER TABLE command but this time use the RENAME COLUMN command to indicate the action we wish to perform:

```
ALTER TABLE <Table Name> RENAME COLUMN <Old Name> TO <New Name>;
```

This will rename the column <Old Name> to have the name of <New Name>.

4.1.3.3 Removing Columns

Removing columns also uses the ALTER TABLE command, but the action we wish to perform to remove a column is the DROP COLUMN command. The use of DROP instead of DELETE is key as DELETE is a DML term (see section 4.2).

```
ALTER TABLE <Table Name> DROP COLUMN <Column Name>;
```

This statement will remove the column <Column Name> from the table <Table Name>.

4.1.3.4 Renaming Tables

Altering table names is slightly different, whilst we still use the ALTER TABLE command to indicate we're changing a table we now use the RENAME TO command to indicate what we want to rename is the table itself.

```
ALTER TABLE <Old Table Name> RENAME TO <New Table Name>;
```

This statement will rename the table <Old Table Name> to the new name provided by <New Table Name>.

4.1.3.5 Removing Tables

Removing tables is even more different in that we do not need to use the ALTER TABLE statement at all because we're not changing a table, rather removing it completely. To do this we use the DROP TABLE command.

```
DROP TABLE <Table Name>;
```

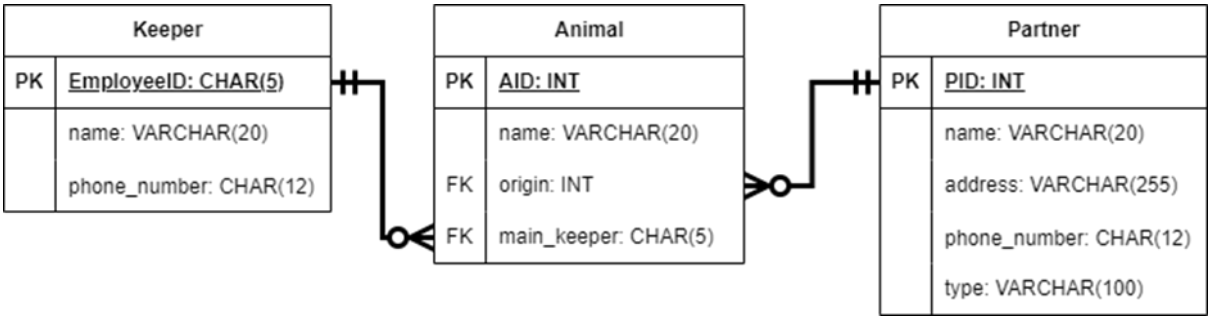
This statement will simply remove the table given by <Table Name> and any data within it.

4.2 Data Manipulation Language

Now you have a database structure we can now consider trying out some CRUD operations. Remember CRUD stands for:

- Create
- Read
- Update
- Delete

For these examples we will be using the Zoo structure as given by the ERD below.



4.2.1 Inserting Data

In order to try out the 'RUD' portion of CRUD we first need to add some data to our database. We can do this using the command INSERT as shown below:

```
INSERT INTO <Table> VALUES (Column 1, …, Column n);
```

An example of this could be adding a new Keeper to the database:

```
INSERT INTO Keeper

VALUES ("C927R", "Adam Jarvis", "4477882639");
```

You can also insert multiple rows at once; for example:

```
INSERT INTO <Table>

VALUES (Column 1, …, Column n),

(Column 1, …, Column n),

…,

(Column 1, …, Column n);
```

Sometimes you need to insert data without inserting the entire row at once. For example, in the example above we may want to insert a new Keeper who hasn't been assigned a contact number yet.

We can do this by specifying what columns we want to insert:

```
INSERT INTO <Table> (Column_Name_0, …, Column_Name_n)
VALUES (Data 1, …, Data n);
```

Applying this general syntax to our scenario:

```
INSERT INTO Keeper (EmployeeID, name)

VALUES ("O273P", "Melinda Cornwall");
```

Notice how we only need to provide two values this time. One caveat to this is that you must always provide the PRIMARY KEY attributes.

4.2.2 Retrieving Data

Once data is inside our database it is very useful to be able to retrieve it again, with this in mind this section covers getting our data back out of the database. Data can be extracted from the database using the following command:

```
SELECT <Columns> FROM <Table>;
```

Here the 'Select' command tells the database what columns we want to fetch and the 'From' indicates which of our tables (or entities) we want to get the data from. An example is given below:

```
SELECT name FROM Keeper;
```

This simple example would currently return two names 'Laudna Ericsson' and 'Oliver Cromwell' however to access this data we need to tell our cursor object that we actually want to retrieve the data. We can do that as follows:

```
curs.execute("SELECT name FROM Keeper;").fetchall()
```

The 'fetchall()' method means we want the execute to return any rows it finds. An alternative is the 'fetchone()' method which returns just the first item returned from the query. We an also return as many columns as we like, for example:

```
SELECT EmployeeID, name, phone_number FROM Keeper;
```

4.2.2.1 Conditionals

Sometimes it's not useful to retrieve all data from a table, instead you'll often be required to extract some data based on some conditions. In order to add these conditions we can use what is known as a WHERE clause.

A WHERE clause allows us to compare attributes of the table to only return rows which match the conditions. For example, the query below will return 'Oliver Cromwell'.

```
SELECT name FROM Keeper

WHERE EmployeeID="OB838";
```

We are able to use multiple conditions and mix/match our conditions using the standard binary operators AND, OR, XOR etc. See below for an example:

```
SELECT name FROM Keeper

WHERE EmployeeID=”OB838”

OR EmployeeID=”HJ837”;
```

In this case it will return both keepers but this serves as an illustration of multiple conditions.

For a full list of operators available please refer to the documentation here:
https://www.tutorialspoint.com/sqlite/sqlite_operators.htm

The most important ones to be aware of are as follows:

Operator	**Description**
A = B	Checks whether A and B hold the same value.
A != B	Checks whether A and B hold different values.
A > B	Checks whether A is bigger than B.
A < B	Checks whether A is smaller than B.
A >= B	Checks whether A is smaller than or equal to B.
A <= B	Checks whether A is smaller than or equal to B.
A BETWEEN B AND C	Checks whether A is between or equal to B and C.
A IN (x, y, z)	Checks whether A exists in a list of values.
A IS NULL	Checks whether A is NULL (empty).

4.2.2.2 Joining Tables

In order to retrieve data from different but related tables we use joins, there are primarily four types of join you will need to be aware of:

1. INNER JOIN
2. LEFT JOIN
3. RIGHT JOIN
4. FULL OUTER JOIN

Each of these joins tables slightly differently and are best illustrated through a simpler example. Take the following two tables:

T1			**T2**	
x	**y***		**i**	**j**
4	1		1	7
5	2		3	6
6	3		9	5

T1 has a primary key of ‘x’ and T2 has a primary key of ‘i’ as indicated by the underlines. Now let’s assume the foreign key in T1 (‘y’) is a reference to ‘i’ in T2. This now enables us to visualise the joins.

INNER JOIN

Now, let’s consider an INNER JOIN, an inner join returns only rows which have a matching reference in both tables. For example, to INNER JOIN T1 and T2 on the y and i attributes we can use the following code:

```
SELECT *
FROM T1
INNER JOIN T2 ON T1.y = T2.i;
```

Notice how we need to select from T1 first and then setup our inner join. We then indicate what attributes we want to join on and in this case the output will be as follows:

x	**y**	**i**	**j**
4	1	1	7
6	3	3	6

Here we see only the combinations of rows in T1 and T2 where i and y are matching.

LEFT & RIGHT JOIN

The left join enables us to extract all combinations of rows which have a matching attribute as well as any rows without a matching attribute in the initial (left) table. Here is an example of the LEFT JOIN:

```
SELECT *
FROM T1
LEFT JOIN T2 ON T1.y = T2.i;
```

The output of the above code would be as follows:

x	y	i	j
4	1	1	7
6	3	3	6
5	2		

Notice how the i and j columns of the final row are blank (or NULL). A right join, does the same operation but returns non-matching rows from the right hand side of the join, in this case T2:

```
SELECT *
FROM T1
RIGHT JOIN T2 ON T1.y = T2.i;
```

x	y	i	j
4	1	1	7
6	3	3	6
		9	5

FULL OUTER JOIN

The final type of join we will discuss is the full outer join which returns matching combinations as well as both unmatching rows from the left AND right tables of the join. For example:

```
SELECT *
FROM T1
FULL OUTER JOIN T2 ON T1.y = T2.i;
```

The results of the above expression are as follows:

x	y	i	j
4	1	1	7
6	3	3	6
		9	5
5	2		

This is like a combination of the LEFT and RIGHT joins!

4.2.3 Updating Data

Sometimes data in our system is incorrect and needs updating, (this is one of the rights data subjects have regarding their personal data!) in order to do this in SQL we use the UPDATE command as follows:

```
UPDATE <Table> SET <Column>=<Value>;
```

This statement on it's own would update the value of the given column in every row unless we specify which rows we'd like it to apply to using a where clause:

```
UPDATE <Table> SET <Column>=<Value>

WHERE <Condition>;
```

An example of this would be updating the name of a keeper with an EmployeeID of S923J to "Steven Lewis":

```
UPDATE keeper SET name="Steven Lewis"

WHERE EmployeeID="S923J";
```

4.2.4 Deleting Data

The final operation we need to be aware of when manipulating data is the DELETE command, one which allows us to remove rows of data. Its general use is as follows:

```
DELETE FROM <Table>;
```

Again, in this form the query would delete all rows from the given table and we will need to add our conditional WHERE clause to it to ensure we only delete specific rows. For example, deleting any keeper entries with an EmployeeID of 'J923L':

```
DELETE FROM keeper WHERE EmployeeID="J923L";
```

5. Relational Algebra

Relational algebra is considered a procedural language which allows us to query data from relational databases, the language consists of a number of operators described below. The examples provided follow the keeper, animal and partner setup used throughout section 4.

Name	Symbol	Description	Example
Projection	$\Pi_x(y)$	Extracts the columns given by x from table y.	$\Pi_{name}(keeper)$
Selection	$\sigma_x(y)$	Extracts rows from y based on criterion x.	$\sigma_{origin<100}(animal)$
Cross Product	$x \times y$	Gives the cartesian product of tables x and y.	$animal \times keeper$
Conditional Join	$x \bowtie_{i=j} (y)$	Joins tables x and y where attribute i matches attribute j.	$animal \bowtie_{origin=PID} (partner)$
Left Join	$x ⟕_{i=j} (y)$	Left joins tables x and y where attribute i matches attribute j.	$animal ⟕_{origin=PID} (partner)$
Right Join	$x ⟖_{i=j} (y)$	Right joins tables x and y where attribute i matches attribute j.	$animal ⟖_{origin=PID} (partner)$
Full Outer Join	$x ⟗_{i=j} (y)$	Joins tables x and y where attribute i matches attribute j but allowing unmatching rows from x and y.	$animal ⟗_{origin=PID} (partner)$

It is rarely the case that a single operator will be used alone, they are more often used in tandem to generate complex queries just like in SQL. The operators can actually be matched pretty easily to SQL syntax and this is something you must be able to do. Here are three examples:

SQL	Relational Algebra
`SELECT AID, name` `FROM animal;`	$\Pi_{AID,\, name}(animal)$

`SELECT EmployeeID`
`FROM keeper`
`WHERE name=”Simon Johnson”;`
	$\Pi_{EmployeeID}(\sigma_{Name="Simon\ Johnson"}(keeper))$

`SELECT phone_number`
`FROM keeper`
`INNER JOIN animal ON`
`EmployeeID=main_keeper`
`WHERE animal.name=”Pete the sheep”;`
	$\Pi_{phonenumber}(\sigma_{animal.name="Pete\ the\ sheep"}($ $keeper \bowtie_{EmployeeID=mainkeeper} (animal)))$

Note, final relational algebra statement:
$\Pi_{phonenumber}(\sigma_{animal.name="Pete\ the\ sheep"}(keeper \bowtie_{EmployeeID=mainkeeper} (animal)))$

There are a range of other operations available however these are not required by the course, if you're interested in exploring these further please refer to:
https://www.geeksforgeeks.org/introduction-of-relational-algebra-in-dbms/

6. Legal & Ethical issues

While it is important to be able to build and design advanced database systems, it is also important to understand the legal and ethical frameworks in which we need to operate. In this section we discuss the legal frameworks and the ethical conundrums we face as database designers and developers.

6.1 Legislation

There are several key pieces of legislation related to information and data which are key to protecting the rights of individuals and ensuring data is made available where relevant. Most legislation revolves around the concept of personal data which is defined under GDPR as:

"'personal data' means any information relating to an identified or identifiable natural person ('data subject'); an identifiable natural person is one who can be identified, directly or indirectly, in particular by reference to an identifier such as a name, an identification number, location data, an online identifier or to one or more factors specific to the physical, physiological, genetic, mental, economic, cultural or social identity of that natural person."

Whilst you do not need to remember that entire definition you must understand the key parts of it. For example, personal data only refers to data relating to an identified or identifiable natural person who is known as the 'data subject'. You should also be able to identify types of personal data where possible. In the following subsections we discuss how this concept of personal data relates to both your human rights and the commercialisation of big data.

6.1.1 Privacy Law

According to the Human Rights Act of 1998: Article 8 you have a right to privacy regarding your:

- Private Life - Sexuality, body, personal identity, relationships with others and personal information.
- Family Life - The right to maintain family relationships.
- Home - You cannot be prevented from entering your home and may enjoy it 'peacefully'.
- Correspondence - Privacy for letters, emails, phone calls etc.

This means you cannot interfere with someone's parental rights, perform surveillance of someone's home or tap their phone lines/physical mail. However, as this is a qualified right some public authorities can interfere with this right under some circumstances:

- The protection of others' rights.
- National security
- Prevention of crime
- Protection of health
- Public safety

Whilst there is a bit of crossover here, generally public authorities can interfere with your article 8 rights if it's for the public's safety.

6.1.2 Freedom of Information

Where you have the right to privacy you also have the right to request information and data from public authorities. This means you have the ability to access and request data and information from public authorities such as:

- The NHS
- Government departments
- Schools/Universities
- The Police
- Etc.

Retrieving this information generally takes one of two forms, either: (i) The public authorities already have a legal obligation to publish information regarding their services, e.g. health statistics; and (ii) Members of the public are entitled to request information from a public authority (freedom of information request). When making a freedom of information request you can request a range of data such as:

- Documents (Drafts, published material, meeting notes etc.)
- Emails
- CCTV recordings
- Telephone conversations
- Meta-data of documents (authors and dates of writing etc.)

However, one thing you can't request is the undocumented thoughts or memories of an individual within a public authority.

6.1.3 GDPR / DPA 2018

The General Data Protection Regulations implemented through the Data Protection Act 2018 are an extensive protection of the rights to individuals regarding the way their personal data is processed and requested. There are several roles that one may have relevant to GDPR such as:

- Data subjects - The individual whose personal data is being processed.
- Data Controller – Any organisation, person or body processing personal data.
- Data Processor – Any body processing data on behalf of the data controller.
- Data Protection Officer – Required appointment for large scale processing.

However, before any of the above individuals (bar the data subjects) can access your personal data they must either obtain a legal exemption and request the data subject's informed consent. This means that in order to obtain consent they must provide you with the following information:

1. The identity of the data controller.
2. What personal data will be processed
3. Why is your personal data being processed in this way.
4. How you can withdraw consent.

Without providing you these four pieces of information you cannot give your informed consent. Alongside this it is also vitally important that data subjects understand their rights

- **Right to be informed** - Data subjects must be informed about the collection and use of their personal data.
- **Right of access** - Data subjects have a right to access and receive a copy of their personal data.
- **Right of rectification** - Data subjects have a right to have their personal data corrected.
- **Right to erasure** - Data subjects have the right to have their data erased.
- **Right to restrict processing** - Data subjects have the right to request the restriction of processing their personal data.
- **Right to data portability** - Data subjects have the right to obtain and reuse their personal data.
- **Right to object** - Data subjects have the right to object to the processing of their personal data.

Understanding these rights will not only help you in the exam but will also help you in your personal lives!

6.2 Ethical Considerations

Whilst a section on ethical considerations could expand into a whole new book it is generally a good principle to follow the 4 W's:

- Who
    - Who is processing the data?
    - Who is the data from?
    - Who is the data about?
- What
    - What data are you processing?
    - What processing techniques are you using?
    - What data isn't needed for the purpose of your system?
    - What legal considerations have been made?
- Why
    - Why are you processing this data?
    - Why have you selected these datasets?
    - Why have you selected this data entry process?
- Where
    - Where is the data being processed?
    - Where is the data from?
    - Where are the outcomes being used?
    - Where are the ethical checkpoints in the service?

Whilst this list is not exhaustive it would be a good starting point when looking at situations. The best way to prepare for questions regarding ethics is to read news articles about current tech trends through a critical eye and ask these questions to yourself. Try reading a range of sources to better understand the different perspectives through which these questions can be answered.