Relational Databases
Summary and Revision

Relational databases consist of a set of **relations** (tables) in which

- columns are named and typed
- rows are identified only by the data they hold
- no two rows hold exactly the same data
- one or more unique columns are chosen to identify the rows
  - **primary key**

There are only two ways of relating data

- by putting the data in the same row of a table
- by relating rows by making them hold a common value
  - **foreign keys**

Designing a Relational Database

There are two main ways of proceeding:

- **conceptual modelling** (e.g. ER)
  - define a high level model of the data and derive the appropriate set of tables
- **normalisation**
  - list all the data in one table and then break it down into appropriate separate tables

The two techniques give rise to a similar set of tables

ER Modelling I

Identify the principal kinds of object you wish to hold data about – **entity types**

- these become tables

Identify the **attributes** of the entity types

- each attribute holds one or more 'simple' values usually a number or short string
- one or more attributes becomes the primary key of the entity and thus the table
  - most entity types are self-contained (**strong**) and have such a key
  - although sometimes you have to invent an artificial new attribute to be this key
  - but some (**weak**) entity types have no primary key, but are dependent for uniqueness on another entity type and so must import part of their key from an owning entity type
- these become columns of the entity types
- **multi-valued attributes** become tables in their own right

ER Modelling II

Identify the **relationships** between the entity types

- each of these represents a set of **connected entities**
- most are between two entity types and so consist of pairs of entities, one from each type
- we need to state for each entity type participating in a relationship
  - if **every** entity **must participate** or not (**total/partial** participation)
  - if **any** entity **can participate more than once**
- the latter gives rise to three situations
  - each entity can only participate once on each side **(1-1)**
    - these become a foreign key column in one side, preferably a totally participating side
  - on one side an entity can participate only once, but on the other an entity can participate many times **(1-N)**
    - these become a foreign key column in the N side
  - entities on both sides can participate more than once **(M-N)**
    - these become a separate table

Normalisation

Start by listing all the data as if it were in one big table and identify a primary key

Identify **functional dependencies**

Split the tables up in order to reduce the duplication of data without losing the functional dependencies

- if the primary key has more than one column in it and if any other column is dependent on **only part of the key** then a new table should be formed
  - (A, B, C) and A -> C means every repetition of an A redundantly repeats a C
  - make it (A, B) and (A, C) – note never throw away the key

2NF

- if there are columns which are **indirectly dependent** on the key then again there is redundant repetition of these columns
  - (A, B, C) and A-> B and B -> C means every repetition of B redundantly repeats C
  - make it (A, B) and (B, C)

3NF

Adding Constraints

The table structure constrains the data structure in some ways

It can further be constrained by further restrictions on the data:

- stating some columns are **unique** – (i.e. never repeated in the table)
- stating some columns are **not null** – (i.e. have a value for all rows)
- stating that one or more columns make up the **primary key**
  - which amounts to being both unique and non-null
- stating that the range of values in a column are restricted **(check)**
- stating that one or more columns is a **foreign key**
  - i.e. takes the values from an equivalent number of columns in a table – usually these are a primary key
  - e.g. if we have T1(A, B, C, ...) and we want to put a foreign key to this table in table T2, we have to add two columns, D and E, i.e.
    - T2(...., D, E, ...) and then
    - (D, E) **refers to** T1(A, B)

Implementing a Relational Database

Use a GUI to create the tables, their columns and their constraints

Use SQL Create Tables

Manipulating a Relational Database

Data is added by:

- importing data
- using SQL **insert**
- using an application with a GUI and a form interface
  - the application will have some way of doing inserts for you

Data is updated

- using SQL **update** and **delete**
- using an application with a GUI and a form interface

Querying a Relational Database

The user specifies queries directly using

- SQL **select** which specifies what we want in terms of:
  - the **tables** we must look in:
    - ones which have data to check
    - ones with data to print
    - together with any connecting tables

From

  - the DBMS will then consider each combination of records one from each of these tables
  - **tests on the data** to see which particular rows we want

Where

  - the **columns** we want to see

Select

- or an application with a GUI
  - which again will making use of SQL **select** in some way

Relational Algebra

The DBMS maps SQL select queries into relational algebra which is a series of steps

- each producing a new (virtual) table or **view**
- either reducing one table or view into some of the rows (**select** - σ) or some of the columns (**project** - Π)
- or connecting two tables
  - **joining** (⋈) two tables puts together each pair of records from the tables which share a common value to form a single records containing all the data from each
    - this is the reverse of normalisation
    - outer joins create rows out of unpaired data filling them out with nulls
  - a **union** of two tables with the same number and same types of columns creates a table with all of the records which are in each
  - **intersection** and **difference** take two similar tables and produce the table with records which are in both or in only one

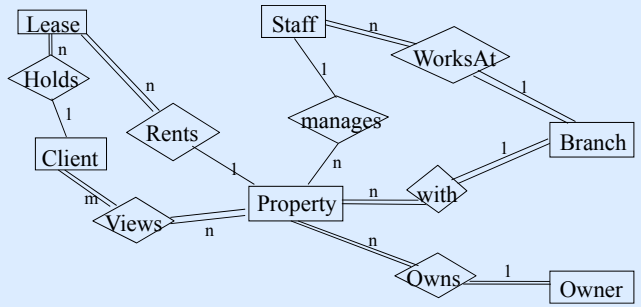
Case Study: Dream Home Property Management

(from "Database Systems" by Connolly and Begg)

The database must support the following requirements:

1. Data Requirements:
   - The company has a number of branches and staff working at those branches.
   - The company maintains a number of properties (each managed by a specific member of staff), their owners and a list of clients who rent these properties.
   - A client will view properties and may eventually commit to renting one, in which case a member of staff organises a lease.
2. Functional (Transaction) Requirements (extract):
   - List the addresses of all of the properties and those managed by a particular member of staff.
   - List all the clients viewing properties owned by a particular person.
   - Identify clients who have inspected all properties with 3 rooms.
   - Add the details of a new lease.

ER Diagram



Attributes on entity types:

- Staff - staff number, name, tel, sex, title
- Client - id, name, address, tel
- Branch - number, address, tel, fax
- Property - number, address, type, rooms, rent
- Owner - id, name, address, tel
- Lease - number, date_start, date_end, rent, payment_method, deposit

on relationships:

Views - date

Creation of Tables I

1. For each entity type, create a table, viz:
   Staff, Client, Branch, Property, Owner, Lease
2. Add columns for each attribute, e.g.:
   Staff(staffNumber: number, name: text, tel: text, sex: text, title: text)
3. Identify keys:

Staff – staffNumber	Client – id	Branch – number
Property - number	Owner - id	Lease - number

   But note, alternative candidate keys:

Branch - tel or fax	Property – address

4. Identify other constraints on columns, e.g.:
   uniqueness – none obvious here
   non-null – staff name, property address, etc.
   checks – staffNumber is between 1000 and 1999 sex is 'F' or 'M'

Creation of Tables II

5. For each 1-N relationship put a foreign key column in the N-side entity table, each of which refers to the primary key column:

   Manages - a column *managedBy* in *Property*

   Holds - a column *heldBy* in *Lease*

   Rents - a column *onProperty* in *Lease*

   Owns - a column *owner* in *Property*

   WorksAt - a column *worksAt* in *Staff*

   With - a column *branchOf* in *Property*

6. For each M-N relationship, create a new table:
   Views - with foreign key columns *client* and *property* and primary key also both columns, *client* and *property*.

7. For each relationship attribute, put a column in the table with the foreign key:
   In the *Views* table, add a column, *date*.

The Resulting Tables I

```
create table Staff(
    staffNumber: Number constraint pk_staff primary key,
    name: Text not null,
    tel: Text,
    sex: Text constraint ck-sex check sex in ('M', 'F'),
    title: Text,
    worksAt: Number constraint fk_worksat references Branch(number) )

create table Branch(
    number: Number constraint pk_branch primary key,
    address: Text not null,
    tel: Text unique not null,
    fax: Text unique not null)
```


The Resulting Tables II

```
create table Client(
    id: Number constraint pk_client primary key,
    name: Text not null,
    address: Text not null,
    tel: Text not null )

create table Property(
    number: Number constraint pk_property primary key,
    address: not null,
    type: not null,
    rooms: Number,
    rent: Number,
    managedBy: Number not null
                constraint fk_manages references Staff(staffNumber),
    branchOf: Number not null
                constraint fk_with references Branch(number) ,
    owner: Number not null
                constraint fk_owns references Owner(id)    )
```


The Resulting Tables III

```
create table Lease(   number: Number constraint pk_lease primary key,
    date_start: Text not null,
    date_end: Text,
    rent: Number not null,
    payment_method: Text not null,
    onProperty: Number constraint fk_on references Property(number),
    deposit: Number,
    heldBy: Number constraint fk_heldBy references Client(id) )

create table Owner(    id: Number constraint pk_owner primary key,
    name: Text not null,
    address: Text,
    tel: Text not null )

create table Views(
    client: Number constraint fk_vclient references Client(id),
    property: Number constraint fk_vprop references Property(number),
    date: Text,
    constraint pk_views primary key( client, property) )
```

The Role of Normalisation

The Universal Relation would be:

U(leaseNumber, staffNumber, staffName, staffTel, staffSex, staffTitle, branchNumber, branchAddress, branchTel, branchFax, date_start, date_end, leaseRent, payment_method, deposit, clientID, clientName, clientAddress, clientTel, propertyNumber, propertyAddress, type, rooms, propertyRent, ownerID, ownerName, ownerAddress, ownerTel, viewingDate)

The primary key is hard to find – the minimal set is:

clientID, propertyNumber

since everything is determined by these.

The Functional Dependencies

Functional dependencies include:

- staffNumber -> staffName, staffTel, staffSex, staffTitle, branchNumber, branchAddress, branchTel, branchFax
- branchNumber -> branchAddress, branchTel, branchFax
- clientID -> clientName, clientAddress, clientTel
- leaseNumber -> everything except viewing date
- propertyNumber -> propertyAddress, type, rooms, propertyRent, ownerID, ownerName, ownerAddress, ownerTel, managingStaffNumber, managingStaffName, etc.
- ownerID -> ownerName, ownerAddress, ownerTel
- clientID, propertyNumber -> viewingDate, leaseNumber

Second Normal Form

Second Normal Form splits off those relations dependent upon part of any candidate key:

Client(clientID, clientName, clientAddress, clientTel)

Property(propertyNumber, propertyAddress, type, rooms, propertyRent, ownerID, ownerName, ownerAddress, ownerTel)

ClientProperty(leaseNumber, staffNumber, staffName, staffTel, staffSex, staffTitle, branchNumber, branchAddress, branchTel, branchFax, date_start, date_end, leaseRent, payment_method, deposit, clientID, propertyNumber, viewingDate)

Third Normal Form

Third normal form splits off transitive dependencies, split *Property* into:

Property(propertyNumber, propertyAddress, type, rooms, propertyRent)

Owner(ownerID, ownerName, ownerAddress, ownerTel)

From *ClientProperty* split off :

Staff(staffNumber, staffName, staffTel, staffSex, staffTitle, branchNumber) // *aka worksAt*

Branch(branchNumber, branchAddress, branchTel, branchFax)

Lease(leaseNumber, staffNumber, date_start, date_end, leaseRent, payment_method, deposit)

leaving:

ClientProperty(clientID, propertyNumber, viewingDate, leaseNumber)

A Difference!

Note that this has created a slightly different set of tables from ER since we are using information not expressible in ER

- that there is only one lease for the pairing of a client and a property
- this moved the foreign key

From ER we got

Lease(leaseNumber, staffNumber, date_start, date_end, leaseRent, payment_method, deposit, clientID, PropertyNumber)

ClientProperty(clientID, propertyNumber, viewingDate)

From Normalisation we got

Lease(leaseNumber, staffNumber, date_start, date_end, leaseRent, payment_method, deposit)

ClientProperty(clientID, propertyNumber, viewingDate, leaseNumber)

Implementing the Transactions

Two methods:

1. SQL - a user-oriented language for managing the data in the database;
2. The Relational Algebra - the operations that the DBMS executes to implement the SQL queries.

Examples:

1. List the addresses of all of the properties and those managed by a particular member of staff, e.g. 'John Smith':

In SQL (all properties):

```
select address from Property
```

In Relational Algebra

$\pi_{address}$ (Property)

```
select address from Property, Staff
    where Staff.name = "John Smith"
        and staffNumber = managedBy
```

JS ← $\sigma_{name = "John Smith"}$ (Staff)

JSprops ← JS ⋈ Property **on** staffNumber = managedBy

$\pi_{address}$ (JSprops)

2. List all the clients viewing properties owned by a particular person, e.g. Jane Smith.

```
select Client.name
from Client, Views, Property, Owner
where Owner.name = "Jane Smith"
    and Property.owner = owner.id
    and Views.property =
            Property.number
    and Views.client - client.id
```

JS ← $\sigma_{name = "Jane Smith"}$ (Owner)

JSprops ← JS ⋈ Property **on** owner = id

JSviews ← JSprops ⋈ Views **on** property = number

JSvClients ← JSviews ⋈ Client **on** client = id

$\pi_{clientname}$ (JSviewingClients)

3. Identify clients who have inspected all properties with three rooms.

```
select C.name
from Client C
where not exists
((select P.number from Property P
        where P.rooms = 3)
        minus
(select V.property from Views V
        where C.id = V.client ))
```

3rooms ← $\sigma_{rooms=3}$ (Property)

3roomsNos ← π_{number} (3rooms)

reduceView ← $\pi_{client,property}$ (Views)

3roomClientNos ← reduceView ÷ 3roomsNos

3roomClients ← Client ⋈ 3roomClientNos **on** id = client

result ← π_{name} (3roomClients)

4. Add the details of a new lease.

The Relational Algebra doesn't deal with updates.

```
insert into Lease values( 1234, "1/1/05", "31/12/05", 240, "monthly", 432, 500,
    16543, 132 )
```

which creates a new Lease numbered 1234 made by staff member 16543 for client 132 on property 432.