Relational Databases

4

What the reader will learn:

- The Origins and terminology of relational databases
- Database design—normalisation
- Database design—entity modelling
- Moving from design to implementation
- The basics of Structured Query Language

4.1 Origins

No discussion of relational databases would be complete without a reference to Edgar Codd's 1970 paper "A Relational Model of Data for Large Shared Data Banks". This was a mathematical description of what we now call Relational databases and operations to manipulate them. This is called relational algebra. Codd considered data could be organised into relations consisting of tuples, each with consistent attributes. A tuple containing 6 attributes would be called a 6-tuple. Most database professionals would translate this as meaning data can be organised into tables consisting of records (or rows), each with consistent attributes (see Fig. 4.1).

It is not intended to give a description of relational algebra here, but for those who want more information see Date (2005), Chap. 5.

Each record (or tuple) in a table is uniquely identified by a primary key which is stored as an attribute or combination of attributes. For example a key could consist of name, street, house number and postcode that could be put together to uniquely identify a person (assuming two people with the same name do not live at the same address). However, the key is more commonly an attribute created specifically for the purpose of uniquely identifying a record, its name often ending in -ID. A quick glance of my own data reveals I have many unique identifiers—my National Insurance Number, my employee ID number, my library ID, my driving licence number and my bank account number are but a small selection of all the unique identifiers associated with me.

P. Lake, P. Crowther, *Concise Guide to Databases*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-1-4471-5601-7_4,

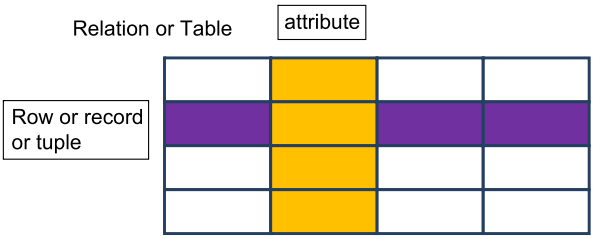

Fig. 4.1 Relational nomenclature

As well as describing the structure of the data, Codd also described a number of operations which could be performed on it. These included selection, projection and joins between tables. Selection and joins are common terminology in databases, but projection defines conditions on the data you want to retrieve, for example people with an age greater than 18.

Ultimately these operations were formalised into a structured query language or SQL. This was released in 1979 by what was to become Oracle Corporation.

4.2 Normalisation

Normalisation is a bottom up approach to database design concentrating on attributes and dependencies. Edgar Codd introduced the concept of normalisation. The primary aim of normalisation is to remove data redundancy, specifically repeating data in a single record. This reduced storage space and increased performance as duplicated data was kept to a minimum and only used to connect tables together.

A good example of why this is important can be seen by looking at the invoice in Fig. 4.2. At first glance this could be regarded as a single record and in paper filing systems it was often treated that way. However retrieving information from it was difficult and a series of indexing systems were developed. For example, there was often a card index with customer details record in it. This information changed very rarely but was a quick method of finding customers details.

The problem with documents like delivery notes, invoices and purchase orders is they consist of data which is from three or more separate tables. For example there is the customer information. This appears on every invoice for that customer. Secondly there is what the customer ordered—this repeats—just look at your supermarket bill. You don't want to store all this information every time you create a new delivery note, so you store it separately and develop relationships between it. Once repeating data is removed, you have your records in first normal form.

So, if we look in detail at Fig. 4.2 we see that although it is a single physical record it contains information about:

- Invoice
- Customer
- Stock item (often called a line item).

Because this is a company system there is no need to store information about the company (there would only be a single record!)

BETTER BOOKS AND BARGAINS
PO Box 2314
London SW3 5JK

Invoice for	**Billing Address**	**Shipping Address**
Your order of 11/12/12	Paul Crowther	Paul Crowther
Invoice ID 2134-5454398	8 Mien Pl	C/- Loxley University
Customer ID 5417- 678U	Sheffield S6 9JH	Loxley LO1 5XC
Invoice date 11/12/12		

Qty	Code	Item Description	Price	Vat	Total
1	9781906040130	Memory of Flames Armond Cabasson	£7.50	20%	£9.00
1	9781906040376	Strangled in Paris Claude Izner	£7.50	20%	£9.00
	Pack and post		£3.00		£3.00
	Total				£21.00

Fig. 4.2 A typical delivery note included with goods in an on-line purchase

Although you can divide the data into these separate tables you need a means of linking them together. So, to tell which customer an invoice belongs to you need to store something that identified them. These days that tends to be an identification number or ID whereas in the past it was more likely to be name and address. Earlier I mentioned how many different ID's were associated with me. How any ID numbers do you think you have associated with you? Each one of these will be specific to some system and you may be generating new ones everyday when you are shopping on-line. Try making a list.

4.2.1 First Normal Form (1NF)

Normalisation requires you to go through a series of well-defined steps. First, remove repeating groups and create a new table to store these attributes. Include a link to the table you have removed it from in the form of the key item. This will be the foreign key. The main reason for removing repeating groups is to make sure you don't end up with variable length records.

It is necessary to identify a primary key to uniquely identify records in the repeating group you have removed. Sometimes you may have to invent one, either for uniquely identifying a record, or because the combination of other attributes is cumbersome. For example if an invoice number did not exist, you could use a combination of the invoice date and billing name to uniquely identify an invoice (although this only works if a customer has a maximum of one invoice a day).

There may be more than one unique identifier—in this case you have candidate keys and you need to choose the most appropriate one. Sometimes it is a combination of attributes. In the example, the Invoice ID is a unique identifier for the invoice. The repeating group is the attributes of the items which were ordered on the invoice—often called line items. In this case the unique identifier is the code (which just happens to be the International Standard Book Number—ISBN). So you know which invoice the line item belongs to, you need to include that as part of the primary key. You then end up with a compound key consisting of Invoice ID and Code. Invoice ID is also a foreign key giving a link back to the invoice the line item relates to. The 2 new tables are therefore:

Invoice
- Order date
- Invoice ID (PK)
- Invoice Date
- Customer ID
- Billing Name
- Billing address
- Shipping Name
- Shipping Address

Line Item
- Invoice ID (PK, FK)
- Code (PK)
- Qty
- Item Description
- Price
- VAT
- Total

4.3 Second Normal Form (2NF)

Remove data that is only dependent on part of primary key (if there is no compound key, it is probably already in 2NF) and create a new table. What you want is for every non-key attribute to be dependent on the whole key

In this case much of the Line item is only dependent on the code, so Line item gets decomposed into Line Item and Stock Item: If you didn't do this you would have to store the item description, price and VAT multiple times—once every time it was included on an invoice—a very bad use of storage. The Line Item table therefore becomes 2 tables with the code giving the link (foreign key) to the new Stock Item table:

Line Item
- Invoice ID (PK, FK)
- Code (PK, FK)
- Qty
- Total

Stock Item
Code (PK)
Item Description
Price
VAT

4.4 Third Normal Form (3NF)

Remove any data that is dependent on a non-key field and create a new table. This happens when there is more than one candidate key and it would be unique. Create a new table with this as the primary key. What you are trying to do here is separate two entities which have become combined. They only appear once on the invoice in this example, but only some of the information contained in the invoice changes with every new invoice.

In the example Name and Address are dependent on the Customer ID, so this can be removed into a new table. The customers details are not going to change very often, so there is no point re-entering and storing them every time a new invoice is created. The Invoice would still need to contain the Customer ID so you would know which customer the invoice related to:

Invoice
Order date
Invoice ID (PK)
Invoice Date
Customer ID (FK)
Customer
Customer ID (PK)
Billing Name
Billing address
Shipping Name
Shipping Address

To achieve third normal form you also remove calculated field. In this case Total can be calculated so you don't need to store it. There are occasions where you might want to store data that can be calculated because of the overhead of doing the calculation is higher than that of storing the calculated value.

In Line item 'Total' can be calculated as you know the quantity, price and VAT, so it can be removed

Line Item
Invoice ID (PK, FK)
Code (PK, FK)
Qty

It is not unusual to find that once a table is in first normal form, it is also in third normal form. Despite this it is still worth going through the steps.

Invoice	Customer	Line Item	Stock Item
Order date	Customer ID (PK)	Invoice ID (PK, FK)	Code (PK)
Invoice ID (PK)	Billing Name	Code (PK, FK)	Item Description
Invoice Date	Billing address	Qty	Price
Customer ID (FK)	Shipping Name		VAT
	Shipping Address		

Fig. 4.3 The final tables

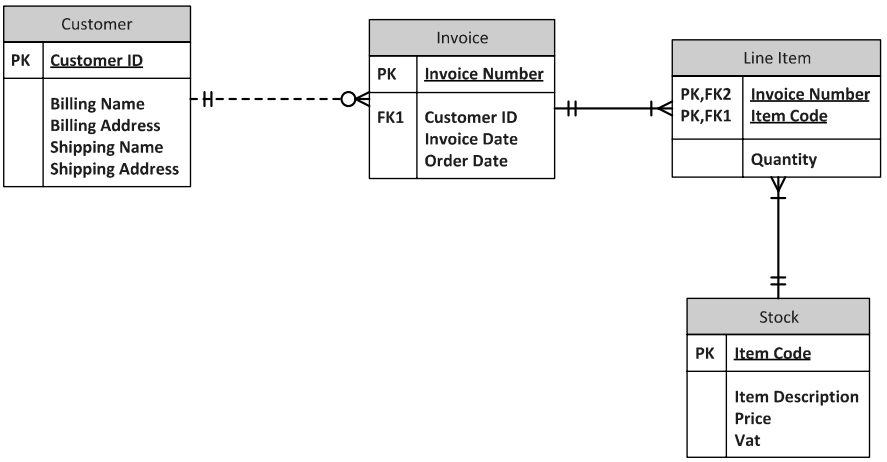

Fig. 4.4 Entity Relationship diagram using crows foot notation to show one to many relationships

The end result is there are 4 tables (see Fig. 4.3).

In all of them the tables in Fig. 4.3 all the attributes are fully dependent on the primary key and cannot be decomposed further.

Often this is drawn in as an ER (entity relationship) diagram. It should be noted there are a number of packages available to draw these diagrams. In this chapter they are drawn using Microsoft Visio (Fig. 4.4).

A number of concepts have been introduced in this example. The main one is the idea of a key. There are a number of different types of key. The first is Candidate Key. This is something that uniquely identifies a record. In some cases there may be more than one, for example in our exercise we had ISBN which uniquely identified an item, but it is possible there could have been a locally used stock code as well. It would be up to the developer to decide which was best to use as the main identifier of the record. That would become the Primary Key. The second type of Key is the Foreign Key. This provides a link to another table and often has the same name as a primary key in another table. Both primary and foreign keys can be composed of more than attribute. The criteria is they are the minimum needed to uniquely identify a record in the table.

4.5 Beyond Third Normal Form

Most developers stop at third normal form as this resolves most of the issues associated with making data storage and retrieval efficient. However it can be taken further to reduce other anomalies which can arise around keys, updates and temporal issues.

Bryce-Codd Normal Form (BCNF) deals with multiple overlapping candidate keys. In these a combination of attributes may create a candidate key. A different combination may form another. BCNF is not always achievable because it would mean losing the dependencies determined at third normal form.

Fourth Normal Form (4NF) is concerned with multivalued dependency. This can occur in a table with three or more attributes when all the attributes in a table are part of the composite key. It may in this case be necessary to decompose the table into 2 or more tables.

Consider the following example:

Garage	Speciality	Suburb Served
Speedy Motors	Ford	Nether Edge
Speedy Motors	Ford	Abbeydale
Speedy Motors	Ford	Hunters Bar
Speedy Motors	Volvo	Nether Edge
Speedy Motors	Volvo	Abbeydale
Speedy Motors	Volvo	Hunters Bar
A1 Service Centre	Peugeot	City
A1 Service Centre	Volvo	City
Sid's Super Service Station	Volvo	Nether Edge
Sid's Super Service Station	Volvo	Abbeydale
Sid's Super Service Station	Peugeot	Abbeydale

Each of the attributes of Garage, Speciality and Suburb Served are key values and form a composite key in third normal form, however the garage's speciality is not affected by the suburb served, so this table should be split into two as the speciality is dependent on the garage and the suburb served is dependent on the garage. As a result we end up with two tables:

Garage_Speciality

Garage	Speciality
Speedy Motors	Ford
Speedy Motors	Volvo
A1 Service Centre	Peugeot
A1 Service Centre	Volvo
Sid's Super Service Station	Volvo
Sid's Super Service Station	Peugeot

Garage_Region

Garage	Suburb Served
Speedy Motors	Nether Edge
Speedy Motors	Abbeydale
Speedy Motors	Hunters Bar
A1 Service Centre	City
Sid's Super Service Station	Nether Edge
Sid's Super Service Station	Abbeydale

Fifth Normal Form (5NF) is again related to multivalued dependency. A table is said to be in the 5NF if and only if every join dependency in it is implied by the candidate keys. Rarely does a 4NF table not be in 5NF.

Sixth Normal Form (6NF) relates to temporal databases and is intended to reduce database components into irreducible components. In the example database there is a problem with stock item. If the price of the item changes, the resulting change would be applied to all historical invoices which is clearly incorrect. Likewise, VAT can vary and this will be independent of price. There therefore needs to be extra tables containing historical price data with the item code and the date they were effective. This would also create a join to the invoice table.

A disadvantage of normalisation, particularly among novice database designers is that it provides a cook book approach which can be followed without any understanding of the individual attributes and their relationships to one another. Another issue is that of the performance hit caused by joins. In the current era where memory and disk space is relatively cheap denormalised forms such as those found in NoSQL (Chap. 5) and in-memory databases (Chap. 8) may be preferred for speed.

4.6 Entity Modelling

An alternative approach to developing a logical design of a database is to identify entities in the system and then map the relationships between them. This raises a number of issues, the first of which is how do we identify entities and secondly which ones are within the domain of the system.

4.7 Use Case Modelling

Use Cases are one part of the Unified Modelling Language (UML) which, although designed for object oriented systems (see Chap. 7), can be used to develop a relational database design. Use Case diagrams are essentially a view of what an external entity (either a user or an interfacing system) want to do with the data.

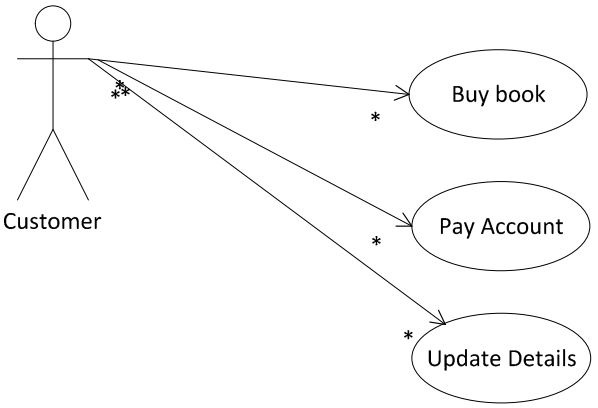

Fig. 4.5 Use case diagram

This allows the designer to work out both what the requirements of the user are and what entities are required in the system.

- A **Use Case** is a statement of functionality required of the software, expressed in the format
- **Actor** Action Subject
- An **Actor** represents a stimulus to the software system. The stimulus can be internal or external.
- An **Action** represents a capability of the software system.
- A **Subject** represents the item acted upon by an Action of the software system.

In the following example we may want to have a customer as an on-line user (Fig. 4.5).

From this we can identify a number of entities: Customer, Book (which could be generalised to stock' and account. Investigation of 'Details' would identify a number of attributes of the customer

What we are identifying here are regular entities which will be transformed into tables. Once these have been established, attributes can be investigated. Many will be simple attributes, but others will be composite attributes, For example Name, and address are both composite attributes. Name consists of Surname and given name (there is a decision to be made as to how many given names are to be stored—anything above 1 may be blank or null).

Occasionally a multivalued attribute may be identified, for example in an employee table you might want to store qualifications. In this case a new table should be created using Employee-ID and Qualification as a composite key. Other attributes may include year obtained and institution.

The example above is also an example of a weak entity. That means it does not have an independent existence. If the employee gets deleted, then the associated records in the qualification table also get deleted. Without the employee, to which there is a one to many relationship, they have no meaning.

works #	name	start date	leave date	car reg
w123	sam	1/1/1991	1/1/2008	a123xjz
w456	jill	1/7/2012		
w789	hal	1/1/2013		v21abc

NULL

Fig. 4.6 Heavy entity

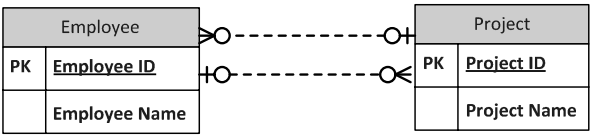

Fig. 4.7 Many-to-many relationship

You also need to be aware of 'heavy' entities. These are entities where there are many null attributes. This may come about because data in one attribute may mean that another attribute must be null. This often happens where the entity is actually a hierarchy of entities. A decision needs to be made as to whether to keep the heavy entity or map it so it reflects the hierarchy. An example of this comes later in this chapter (Fig. 4.6).

Once entities and their attributes have been identified binary relationships should be identified. Initially map one-to-many relationships. In this the item at the 'many' end will include the primary key of the entity at the 'one' end as a foreign key. For example, a customer may have many orders (although an order can only be associated with one customer). In this case the order contains the customer-id as a foreign key.

The more complex part is unravelling many-to-many relationships. For example, an order may contain many stock items, but a stock item may occur on many orders. Many-to-many relationships should not be part of a database design. They are usually resolved by creating an associative entity.

Consider the following situation. An employee may be assigned to several projects and a project may have one or more employees assigned to it (Fig. 4.7).

In this case a new entity called an associative entity needs to be created. The name of this entity will depend on what the database is used for. It may be as simple as 'Employee Assignment', or more complex, like 'Employee Time Tracking' where time spent and the activity could be stored as entities (Fig. 4.8).

In both cases the new entity would have a composite key of Employee_ID and Project-ID (although in rare cases a unique key may be created).

Occasionally you may find you have a three (or more) way many-to-many relationship. For example a many suppliers may supply many parts to many customers.

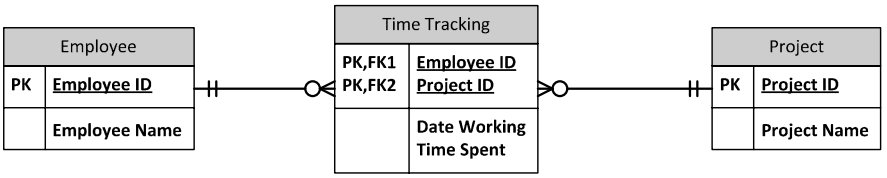

Fig. 4.8 Resolved many-to-many relationship

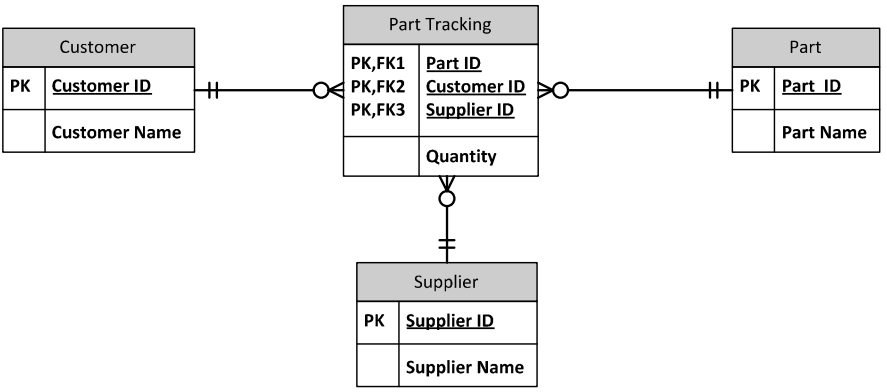

Fig. 4.9 Resolved three way relationship

Also, any supplier can supply any part to any customer. This can be resolved in much the same way as a many-to-many relationship by adding an associative entity. This will have a composite key composed of the primary keys of the customer, supplier and part tables (Fig. 4.9).

Generally this will solve most of the design issues, however there may be times when a number of other issues arise. One of these are unary relationships. The most often quoted example of this (and the one used in standard Oracle exercises) is where one employee is the manager of another employee. In this case a manager manages many employees, but an employee has only one manager. In practice this means a employee entity has an attribute, probably called 'manager' which is a foreign key pointing back to the same table. Ultimately one or more employees have a null value in this field because they have no manager (they are the top of the organisations hierarchy) (Fig. 4.10).

Rarely, there may be a unary many-to-many relationship. For example, in manufacturing a product may be composed of many parts. Those parts may be used in many products which themselves be a part of another product (Fig. 4.11).

To make this work both the components of the primary key of assembly reference the primary key of part. That way the assembly has a one to one relationship with part so the assembly can have a name, but at the same time there is another

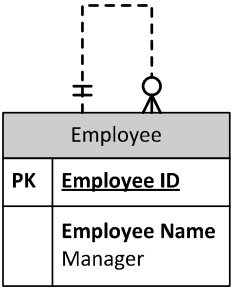

Fig. 4.10 Unary relationship—self join

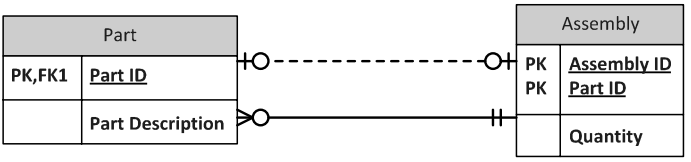

Fig. 4.11 Resolved unary many-to-many relationship

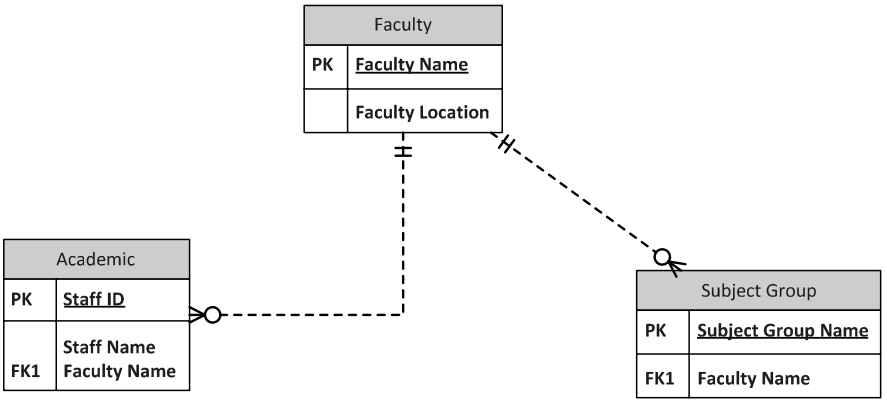

Fig. 4.12 Fan trap

relationship between assembly and part showing the assembly is made up of many parts.

There are two potential problems when modelling entities, the fan trap and the chasm trap. These situations usually arise because of missing relationships. They can often be identified by going back to the original use case diagrams and testing the relationships to see if the required processes can be achieved (Fig. 4.12).

In the above ER diagram it is not possible for an academic staff member to work out which subject group they are in.

This could be redrawn as Fig. 4.13, but this now gives us a chasm trap as not all staff are in subject groups, but all staff belong to a faculty. The way to resolve the problem is to add the missing relationship (Fig. 4.14).

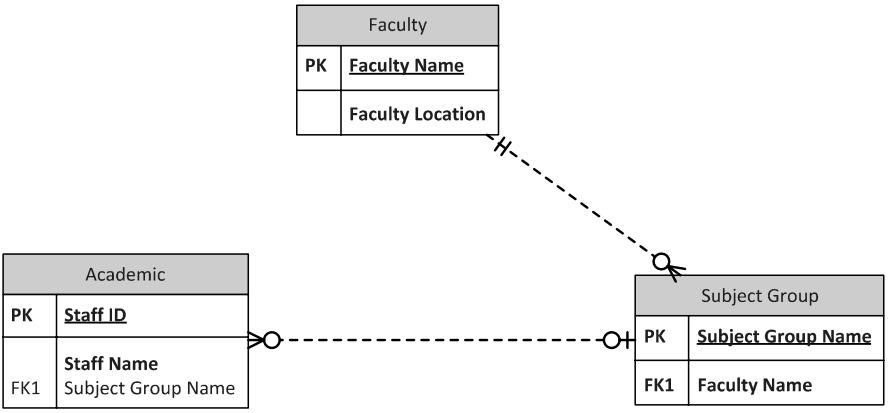

Fig. 4.13 Chasm trap

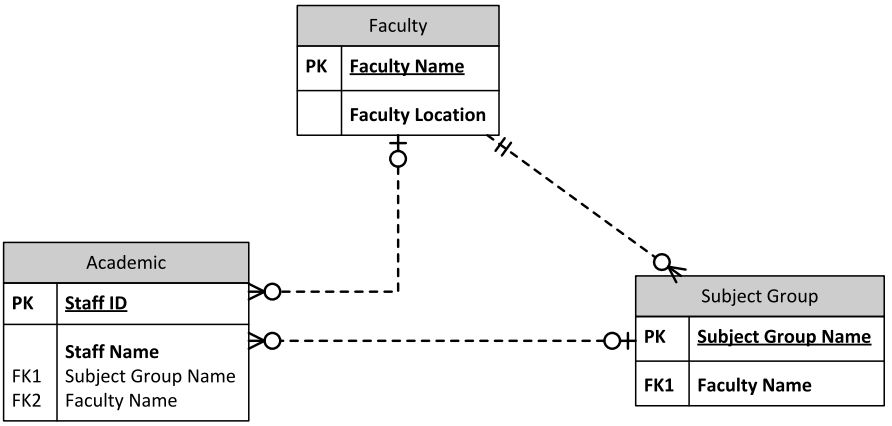

Fig. 4.14 Resolving the traps

The steps in entity modelling are therefore as follows:

1. Conduct a use case analysis to identify regular entities
   a. Identify composite attributes
   b. Identify multivalued attributes
   c. Map weak entities
   d. Identity 'heavy' entities and decide on a mapping resolution
2. Map Binary relationships
   a. Identify one to many relationships
   b. Identify Many-to-many relationships
   c. Map associative entities
3. Map unary relationships
4. Check for fan and chasm traps

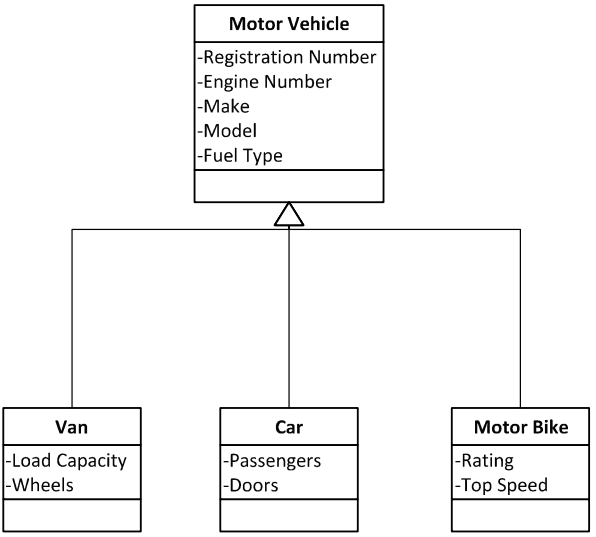

Fig. 4.15 Hierarchy diagram showing superclass (motor vehicle) and three subclasses

4.8 Further Modelling Techniques

With the advent of object oriented modelling, particularly where modellers are using UML toolkits, hierarchy models are sometimes constructed. This makes logical sense, but they can't be directly implemented in a relational model. An object relational database system is the usual way this kind of model is implemented and this will be discussed in depth in Chap. 7. However in the current context consider the following simple hierarchy (Fig. 4.15).

There are three ways this could be translated into relational tables:

1. Collapse all the sub types into the super type—the heavy entity approach (Fig. 4.16).

This has the advantage of creating a very simple structure—a single table. It has the disadvantage that for every record there will be four null fields, in other words we have created a heavy entity. For example if we had a record for a car, the data for a van and a motor bike could be null.

This approach is best used where there are more attributes in the super type than in the subtypes

2. Combine the attributes of the super type with the subtypes (Fig. 4.17).

This has the advantage that every distinct entity is now modelled separately and there are no null fields. The main disadvantage is that relationships start to become complex. For example if we had an entity called owner, each of the above entities would need a foreign key attribute to join to the owner attribute.

This approach is best used where the majority of the attributes are in the sub type.

Fig. 4.16 Heavy entity approach

Motor Vehicle	
PK	**Registration Number**
	Engine Number Make Model Fuel Ytpe Year of Manufacture Load Capacity Wheels Passengers Doors Rating Top Speed

Van	
PK	**Registration Number**
	Engine Number Make Model Fuel Ytpe Year of Manufacture Load Capacity Wheels

Car	
PK	**Registration Number**
	Engine Number Make Model Fuel Ytpe Year of Manufacture Passengers Doors

Motor Bike	
PK	**Registration Number**
	Engine Number Make Model Fuel Type Year of Manufacture Rating Top Speed

Fig. 4.17 Attributes moved down into subtypes

3. Implement the structure by making each of the types an entity in its own right and adding appropriate keys (Fig. 4.18).

This has the advantage of retaining most of the original hierarchy structure with the main disadvantage being the introduction of an extra attribute to allow joining of the tables.

This approach is best used where the number of attributes in all the types is about equal and the structure if the hierarchy wants to be maintained.

4.9 Notation

In this chapter we have been using the 'crows foot' notation of describing relationships. However many designers (and the packages they use) adopt the UML convention with the cardinality of the relationship expressed at each end. The diagram below shows the equivalences (Fig. 4.19).

A further notation form by Chen (1976) shows entities and relationships using the symbols, shown in Fig. 4.20.

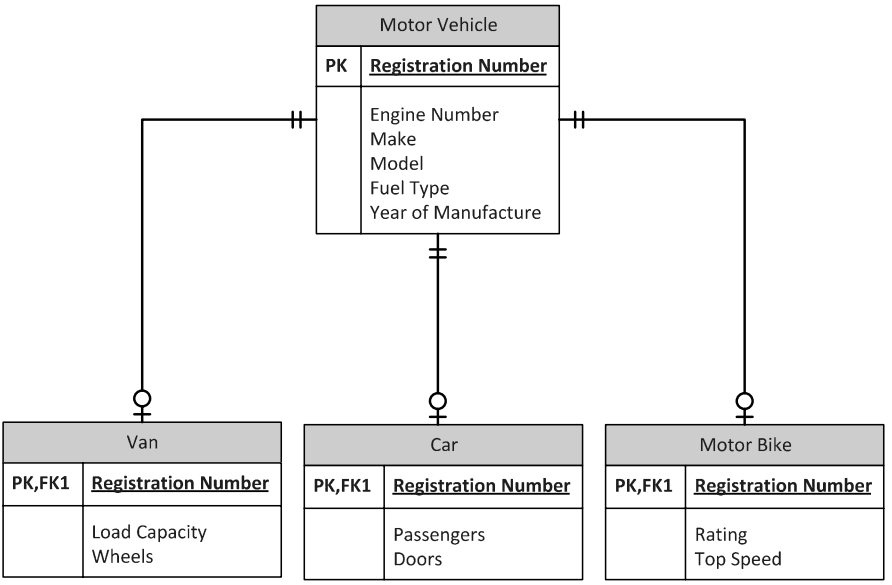

Fig. 4.18 All types mapped as entities

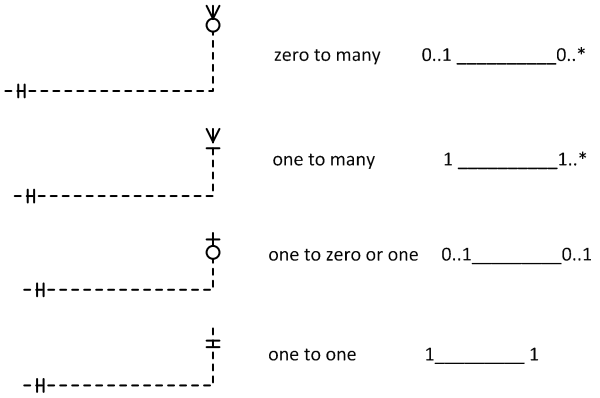

Fig. 4.19 Crows foot compared to UML notation

The bookshop example therefore can be redrawn in Chen's notation (Fig. 4.21). This can be compared with the modified entity-relationship diagram shown in Fig. 4.22.

Which the author finds particularly messy and complicated when compared to the notation used everywhere else in this chapter. The only real advantage is the diagram forces you to name relationships

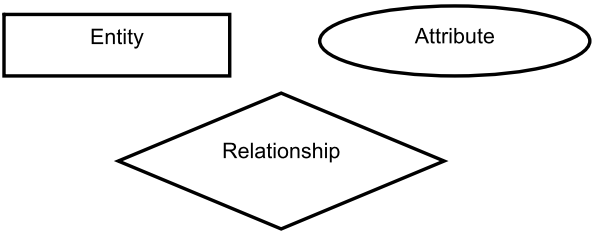

Fig. 4.20 Chen's notation

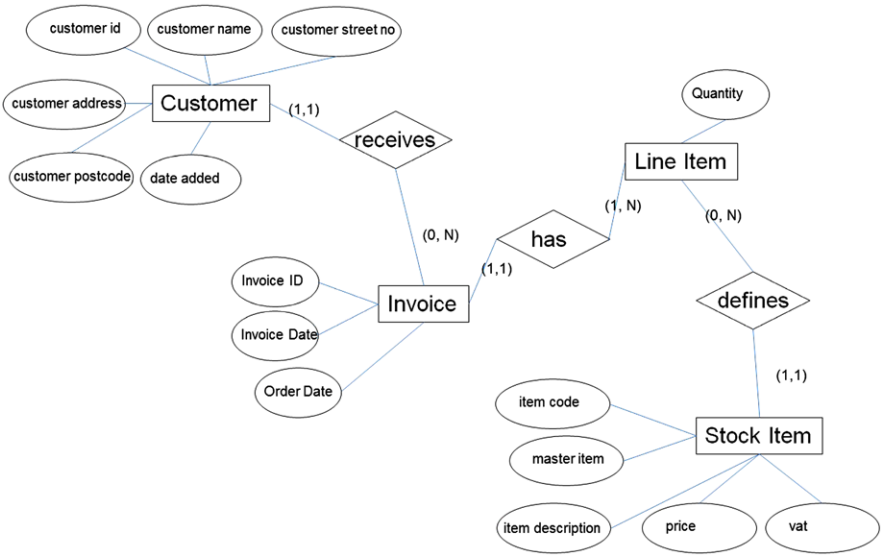

Fig. 4.21 The example using Chen's notation

4.10 Converting a Design into a Relational Database

Once you have created your database design using either of the methods mentioned above, you then need to convert it into a physical database. To do this there are some other questions to be answered.

Does any of the design need denormalising? You should look at any 1:1, 1:0 or 1:n relationships, particularly where n is 3 or less. This will reduce the number of tables and hence the complexity of the design. The overhead may mean some attributes with null values.

What are the datatypes that will be used for each of the attributes? This may be a decision influenced by the vendor package. For example not all RDMS support boolean datatypes. Oracle supports:

CHAR: This is a fixed length datatype which is backfilled if the data stored is less than the field size, or returns and error is the data is longer than the field size

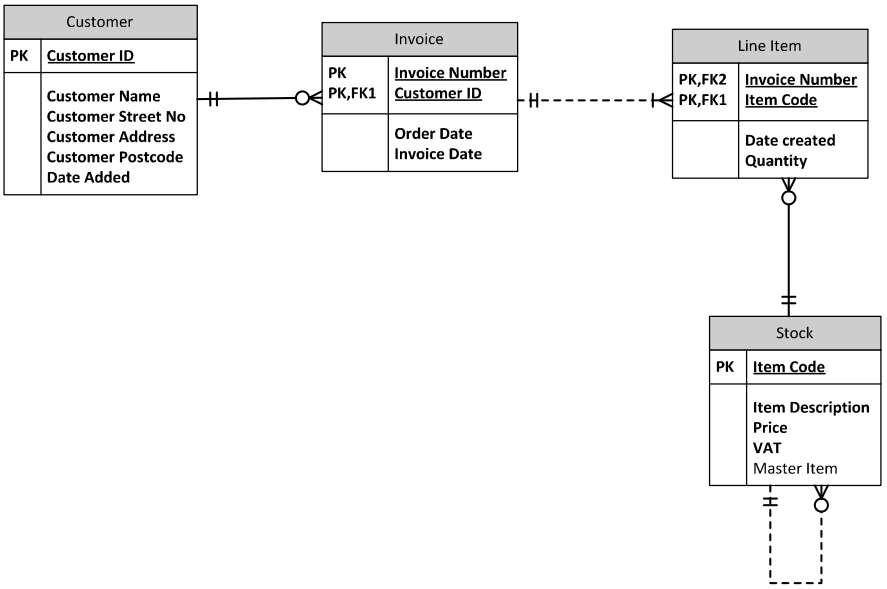

Fig. 4.22 Modified ER diagram

VARCHAR2: is a variable length data type which is more efficient than CHAR. However you still need to specify a maximum length and an error will be returned if this is exceeded

NUMBER: There may be issues with very large and very small numbers. Oracle stores numbers up to 38 significant digits (both positive and negative). This is more than adequate for most applications, but may be an issue if the database is storing scientific data

DATE

CLOB

BLOB

No matter how you develop your tables, either through normalisation or entity modelling, you also need to identify constraints. These will help maintain data integrity.

Primary Key constraint: This is basically identifying the primary key. It means values must be unique and they must not be blank (null)

Foreign Key constraint: Identifies the table the foreign key references

Not Null constraint: The entity must contain data, in other words it cannot be blank

Unique constraint: but that does not mean it is a primary key. You don't have to use this constraint on the primary key because that and NOT NULL are part of the primary key constraint.

Check constraints: These give rules as to what is valid data. They include mathematical constraints where data must be in a specific range and alphanumeric conditions where data must come from a specified list or contain (or not contain)

specific characters. There will be a more in depth discussion of this in Chap. 10 on security.

It is not uncommon for a particular entity to have several constraints.

Most database management system will allow you to name the constraints you create however if you do not name them, they will still be created and assigned a system name which will be almost impossible to retrieve. It is therefore recommended you name your constraints and use a systematic naming convention (for example foreign keys are prefixed with FK_). The name and the constraint type are held in the systems data dictionary.

4.11 Worked Example

The following exercise uses Oracles SQL Plus, but as few of Oracle specific operators will be used as possible and the syntax works in Microsoft's SQL Server except where stated. The customer details have been slightly altered to include a postcode and a date when the customer was added to the system. In practice the address and name attributes would probably be broken down further, for example, surname, given name and initial for name and street number, street, city for address.

4.12 Create the Tables

In this example we are going to create the database seen in Fig. 4.22. This has been modified from the original example in order to illustrate some features of SQL. Tables can be created in any order, however it is more efficient if you create tables which have no other tables dependent on them first. They can usually be identified by not having the foot of the crows foot notation attached to them, alternatively they are the 1 end of a 1:many relationship. Using this rule the first tables to be created are **Customer** and **Stock Item.** The next is **Invoice** because it is only dependent on **Customer** and finally **Line Item** because it is dependent on both **Invoice** and **Stock Item**

```
CREATE TABLE customer
    (customer_id VARCHAR(6),
    customer_name VARCHAR (24),
    customer_street_no NUMBER(3),
    customer_address VARCHAR (36),
    customer_postcode VARCHAR(6),
    date_entered (DATE));
```

This syntax would create the table, but does not identify any constraints. Customer ID should be identified as the primary key, the rest of the fields should have data in them, that is should be NOT NULL.

Therefore the CREATE statement becomes

```
CREATE TABLE customer
    (customer_id VARCHAR(6)
    customer_name VARCHAR (24) NOT NULL,
    customer_street_no NUMBER (3),
    customer_address VARCHAR (36) NOT NULL,
    customer_postcode VARCHAR(6),
    date_entered (DATE),
    CONSTRAINT customer _pk PRIMARY KEY (customer_id));
```

Constraints can be added and modified after the tables are created, but wherever possible it is best to include them here. It is also important to name constraints so that can be easily found in the data dictionary. If they are not named, they will still be stored, but will be assigned a system name making them difficult to retrieve. In this exercise a postscript convention to indicate the type of constraint is adopted:

_pk primary key constraint
_fk foreign key constraint
_uk unique constraint
_ck check constraint

The Stock item table is created in a similar way

```
CREATE TABLE stock_item
    (code VARCHAR(13),
    item_description VARCHAR (24) NOT NULL,
    price NUMBER (7,2) NOT NULL,
    vat NUMBER (2) NOT NULL,
    master_item VARCHAR(15),
    CONSTRAINT stock_item_pk PRIAMRY KEY (code),
    CONSTRAINT stock_item_fk FOREIGN KEY master_item REFER-
    ENCES stock_item(code),
    CONSTRAINT item_description_uk UNIQUE (item));
```

This create statement illustrates how numbers are handled. **price NUMBER (7,2)** defines price as being able to store numbers up to 9999.99. The 7 gives the size of the field including the decimal point, and the 2 gives the number of places after the decimal point. The VAT definition means two digits can be stored. Since this is normally a percentage, it could have been defined as **vat NUMBER (3,2).**

As well as a primary key constraint, this definition also contains a unique constraint which means no two descriptions can be the same. In theory this could have been used as the primary key, but from a labelling and scanning perspective which is how most items are handled today, the code (which happens to store the data on a bar code) is the better candidate.

The new master_item attribute is to be used where there may be several individual items making up another item. For example, three books may be sold as a box set. The books can be purchased individually or as a box set. Since not all items are grouped, the new master_item attribute must not have the '**NOT NULL**' clause associated with it. If they are a box set they will have a unique code. There is also a new foreign key constraint linking the table with itself.

The invoice table is created in a similar way, but in this case there is a foreign key constraint which allows a join between this table and the customer table

```
CREATE TABLE invoice
    (invoice_id NUMBER (8),
    invoice date DATE NOT NULL,
    order_date DATE NOT NULL,
    customer_id VARCHAR(6)
    CONSTRAINT invoice_id_pk PRIMARY KEY (invoice_id),
    CONSTRAINT customer_id_fk FOREIGN KEY (customer_id)
    REFERENCES customer (customer_id));
```

The foreign key constraint creates a link between customer and the invoice table. It also means you cant have an invoice unless there is an associated customer record in the customer table. Another impact of this constraint is you can't delete a customer record if there are still any associated invoices.

The final table to create is the line item table

```
CREATE TABLE line_item
    (invoice_id NUMBER (8),
    code VARCHAR(13),
    qty NUMBER (4),
    CONSTRAINT line_item_pk PRIMARY KEY (invoice_id, code),
    CONSTRAINT invoice_id_fk FOREIGN KEY (invoice_id)
    REFERENCES invoice (invoice_id),
    CONSTRAINT code_fk FOREIGN KEY (code)
    REFERENCES stock_item (code));
```

In this case there is a combined primary key and two foreign key clauses. This is common where a many-to-many relationship is being resolved. If any of the constraint clauses is violated, an error will be returned. If more data than the field you have defined is entered, an error will be returned. There are two ways of handling these errors. One is by displaying the error directly to the user. A better way is to have some programming code which handles the error in a more friendly way. This aspect is beyond the scope of this discussion.

4.13 CRUDing

When working with a database you normally want to do four things: Create, Retrieve, Update and Delete data. Update and Delete operations are usually done in conjunction with a Retrieve operation. It is always a good idea to show a user what they are about to either change or delete before they do it and give them the option of aborting the operation if they discover they have the wrong record. The rest of this section will take you through a selection of structured query language statements. It is not meant to be a complete set of examples, is provided to introduce SQL. Further details of SQL for Oracle and SQL Server can be found in the further reading section at the end of this chapter

4.14 Populate the Tables

Once a table is created, it needs to be loaded with data. What will be demonstrated below is how an initial load could be done. From a user point of view, the normal day to day entry of data would be done via web forms created using a package such as Oracles Developer or via web forms.

To insert a new record into the customer table you would use the syntax:

INSERT INTO customer VALUES ('OS3457', 'Warren Felsky', 8, 'Mien Place, Sheffield', 'S7 4GH', '20-APR-13');

It should be noted that different database management systems have different ways of handling time information. In the case of Oracle rather than **'20-APR-13'**, **TO_DATE ('APR 20 2013', 'MON DD, YYYY')** could be used where **TO_DATE** is an Oracle built in function.

Remember, you would not be able to insert records into the invoice table until you had records with customer ID's in the customer table. If you tried you get a foreign key constraint error.

4.15 Retrieve Data

The power of a relational database lies in the ability to easily search for data which meets a particular condition. This condition may require data to be retrieved from one or more tables. In the next section we will work from simple retrieval statements to more complex ones which show the power of SQL.

Perhaps the simplest statement is one where you want to retrieve all the records in a table. In this case we will use the customer table

SELECT * FROM customer;

This is not a particularly useful statement as in a production system it could retrieve hundreds of records.

```
SELECT * FROM customer
    WHERE billing_name = 'Warren Felsky';
```

Would return all customer records for the name Warren Felsky.

You can add multiple conditions, for example

```
SELECT * FROM customer
    WHERE billing_name = 'Warren Felsky' AND
    date_entered > '01-JAN-2010';
```

The * in **SELECT *** means all columns are retrieved. It may be that you only want data from certain columns to be to returned. To do this replace the * by the column names required. Some systems like Oracle will automatically label the output with these column names

```
SELECT billing_name, billing_address, billing_postcode, date entered FROM
    customer WHERE billing_name = 'Warren Felsky' AND
    date_entered > '01-JAN-2010';
```

4.16 Joins

Retrieving data from one table is seldom enough. For example we might want to retrieve the invoice date and order date of a customer. Unless the customers id number is know, this would require retrieving data from two tables. This is known as a table join. There are a number of types of joins:

```
SELECT billing_name, invoice_id, invoice_date
    FROM customer, invoice
    WHERE customer.customer_id = invoice.customer_id
    billing_name = 'Warren Felsky' AND
    date_entered > '01-JAN-2010';
```

The first part of the **WHERE** clause defines how the tables are going to be joined by linking the primary key in one table (in this case the customer table) with its equivalent foreign key in the invoice table. Because **customer_id** is common to both tables it must be prefixed by the table name to avoid ambiguity.

The above example is called an **Equijoin**. This means that only records which have matching values in both tables will be returned.

Inner Join (Also known as a simple join) returns those rows that satisfy the join condition

```
SELECT customer_name, invoice_id, invoice_date
    FROM customer
    INNER JOIN invoice
    ON customer.customer_id = invoice.customer_id;
```

This syntax would retrieve those customers who have invoices. Any customers without an invoice would not be shown. Note that '**INNER**' is an optional statement and does not need to be included.

In Oracle an **Equijoin** and an **Inner Join** are equivalent.

Outer Join Extends the idea of an inner join by including some or all records from the other table in the join which do not meet the join condition. There are three types of outer joins:

Left Outer Join This will return all records from the table on the 'left' (literally to the left of the **LEFT** clause) which have no matching records in the table on the right. A null will be displayed instead. In the following example customers including any which don't have any invoices associated with them will be returned. This will have spaces (null) in the **invoice_id** and **invoice_date** output column.

```
SELECT customer_name, invoice_id, invoice_date
    FROM customer
    LEFT OUTER JOIN invoice
    ON customer.customer_id = invoice.customer_id;
```

The word **'OUTER'** is optional in the syntax so **LEFT JOIN** would give the same result.

Right Outer Join This will return all records from the table on the 'right' (literally to the left of the **RIGHT** clause) which have no matching records in the table on the right. A null will be displayed instead. In the following example all stock items and their codes will be displayed regardless of whether they have appeared on an invoice. Those which have no associated invoice will have a null displayed instead of the **invoice_id.**

```
SELECT invoice_id, stock_item.code, item
    FROM line_item
    RIGHT JOIN stock_item
    ON line_item.code = stock_item.code;
```

Full Outer Join This will return all records from both the 'left' and the 'right' tables whether there is a matching record in the other table or not. The syntax uses **FULL JOIN.** Like the **LEFT** and **RIGHT** joins, any unmatched record will have nulls displayed.

Self Join Self joins are, as the name suggests a join that links a table to itself. Lets consider the definition of the **stock_item** table again where we set up the conditions for a self join:

```
CREATE TABLE stock_item
    (code VARCHAR(13),
    item VARCHAR (24) NOT NULL,
    master_item VARCHAR (24),
    price NUMBER (7,2) NOT NULL,
    price NUMBER (7,2) NOT NULL,
    CONSTRAINT stock_item_pk PRIMARY KEY (code),
    CONSTRAINT stock_item_fk FOREIGN KEY master_item REFER-
    ENCES stock_item(code),
    CONSTRAINT item_description_uk UNIQUE (item));
```

There is no specific self join syntax, but a **LEFT JOIN** can be used.

```
SELECT m.item, i.item,
    FROM stock_item AS m LEFT JOIN stock_item AS i
    ON m.master_item = i.item
```

In the example the syntax **AS m** and **AS i is** used to establish aliases to distinguish between instances of the table, in other words are we looking at the 'master

item' or 'ordinary' item? Remember the **master_item** field may be **NULL** because not all items are bundled into box sets. In this example only unmatched records on the left will be displayed. Since these are box sets, it is unlikely there will be any.

Cartesian Join A final type of join is the. In this case every record in one table is matched with every record in the joined table. So if you had two tables each of 10 records, you would end up with 100 records returned. This is obviously an error and usually happens where matching join records are not specified. About the only time this is likely to happen is if an **equijoin** is being used and the **WHERE** clause linking the associated primary and foreign keys is left out

Now we know how to retrieve data from tables and how to join tables together we can start looking at more complex data manipulation. This includes more complex WHERE clauses and grouping of data, for example, how could we find the total value of an invoice, particularly as we dropped that attribute as part of the normalisation process because it was a calculated field.

4.17 More Complex Data Retrieval

It is not unusual to know part of what we are looking for in our data base. For example we know that there is the word 'Paris' in the title of the book we are looking for. Most databases allow the use of wildcard searches. A wildcard is a character that takes the place of one or many other characters. It may be an $*$, or in the case Oracle it is % for may characters and _ for a single character. So to search for Paris in our stock_item table we can use:

```
SELECT code, item, price
    FROM stock_item
    WHERE item LIKE '%Paris%';
```

If you wanted to see the cheapest book with Paris in the title you could add the **ORDER** clause:

```
SELECT code, item, price
    FROM stock_item
    WHERE item LIKE '%Paris%'
    ORDER BY price DESC;
```

Omitting the **DESC** clause would make the order most expensive first. You can have several levels of ordering, for example, if you wanted to sort first by price, then by title the **ORDER** clause would become:

```
... ORDER BY price DESC, item;
```

A further requirement in most systems is to be able to calculate data. This is done on a group of records that are returned. For example how many invoices do customers have?

```
SELECT COUNT(invoice_id)
    FROM invoice
    GROUP BY customer_id;
```

This can get quite complicated when a number of tables are involved, for example, what is the total value of an invoice. This requires at least three of the tables in our database and there is an argument for all four if the customers' names is required as well. The joins here are all **INNER** joins because we are only concerned equal matches.

```
SELECT invoice_id, COUNT (code), SUM (price)
    FROM invoice JOIN line_item
    ON invoice.invoice_id = line_item.invoice_id,
    line_item JOIN stock_item
    ON line_item.code = stock_item.code
    GROUP BY invoice_id;
```

4.18 UPDATE and DELETE

The final two classes of operations in a relational database are **UPDATE** and **DELETE** where you want to make changes to the data. Let's say we want to update a customer's address. First it is a good idea to make sure you have the correct customer so we could use:

```
SELECT * FROM customer
    WHERE billing_name = 'Warren Felsky';
```

which we have seen before to retrieve all the customers with the name Warren Felsky. We see from the data returned that one customer, OS3457 is the one we want. We can then issue the UPDATE command to change the customers address:

```
UPDATE customer
    SET  customer_street_no = 6,
         customer_address = 'McGregor St, Sheffield'
         customer_postcode = 'S11 1OD'
    WHERE cutomer_id = 'OS3457';
```

The where clause can contain any of the conditions you have seen so far. It may have the effect of updating more than one row so it is essential you retrieve and verify records before you update them. It is also possible to get an integrity constraint error if you try and update a record with a constraint, for example if you try and update an attribute defined as a foreign key.

Removing data from a table is both very simple and very dangerous. Like with UPDATE you need to be sure that the record you are deleting is the one you want to delete. Lets say we want to remove Warren Felskys record

```
DELETE FROM customer
    WHERE customer_name = 'Warren Felsky';
```

Two things could go wrong here. If there is more than one warren Felsky, both will be deleted. It would have been better to use the customer_id. Secondly, if Warren Felsky still had any live invoices in the system there would be an integrity constraint error. In other words all invoices associated with Felsky would have to be deleted before the customer record could be removed.

The database is not permanently changed until a COMMIT command is issued and up to that point a ROLLBACK command could be issued to return the database to its original state. Most systems do a COMMIT when a user logs off.

This is related to ACID which was covered in Chap. 2 where we looked at transaction processing.

4.19 Review Questions

The answers to these questions can be found in the text of this chapter.

- What is a tuple?
- What is the difference between a primary key, candidate key and foreign key?
- What is CRUDing?
- What is a heavy entity?
- What is a weak entity?

4.20 Group Work Research Activity

This activity requires you to research beyond the contents of the book and can be tackled individually or as a discussion group.

Hallam Accommodation and Housing

Agency

The Agency currently rents out various types of domestic accommodation (flats and houses) to clients. The clients (e.g. students or groups of students) may require 1 year leases or (e.g. families) longer term lets.

Details are kept on each property and in particular:

Address (this must be searchable by postcode or partial post code or by district)
Owner details
Lead tenant details
Tenancy start
Tenancy end
Rent
Type of property (flat, detached house, terrace house etc.)
Furnished/Unfurnished (for Furnished, further details of the furnishings may be kept)

Number of bedrooms
Number of bathrooms
Number of reception rooms
Optionally also:
A textual description
Photographs.

A history of occupancy needs to be kept including periods were the property is unoccupied.

In order to make searching for appropriate properties easier it has been decided that details of individual rooms within a property will also be stored. This will include:

room type (bedroom/bathroom/kitchen etc.)
room dimensions (in feet and meters)
heating (e.g. radiator/fire)
for kitchens: appliances
for bathrooms: fittings
any special features e.g. patio windows.

The tourist business has started to boom in the area and the Agency wishes to move into the business of holiday lettings. This will involve much shorter lets and a much greater requirement for up to date availability information.

1. Using normalisation, create an entity relationship diagram for the scenario.

Was it possible to go beyond third normal form?

2. Repeat the exercise but this time use entity modelling.

Were the results the same?

3. Write the create statements necessary for the scenario.

What constraints are necessary?

4. Create a query that retrieves a list of vacant properties (tenancy end date is before today's date) and their owner. Try grouping the properties by owner.

References

Chen P (1976) The entity-relationship model—towards a unified view of data. ACM Trans Database Syst 1(1):9–36

Codd EF (1970) A relational model of data for large shared data banks. Commun ACM 13(6):377–387

Date CJ (2005) Database in depth: relational theory for practitioners. O'Reilly, Sebastopol

Further Reading

Chen P (2006) Suggested research directions for a new frontier: active conceptual modeling. In: Conceptual modeling—ER 2006. Lecture notes in computer science, vol 4215. Springer, Berlin/Heidelberg, pp 1–4

Microsoft (2013) Microsoft SQL server library. Available on line at http://msdn.microsoft.com/en-us/library/bb545450.aspx. Accessed 22/04/2013

Oracle® (2010) Database SQL language reference 11g release 1 (11.1). Available on line at http://docs.oracle.com/cd/B28359_01/server.111/b28286/toc.htm#BEGIN. Accessed 22/04/2013