Object and Object Relational Databases 7

What the reader will learn:

- The problems with data structures and relational databases
- What is an object
- How are objects handled in a database
- What is an object oriented database and its features
- What is an object relational databases and how it is implemented

7.1 Querying Data

One of the issues with data is that it has become more and more complex. As we saw in Chap. 2 there was a transition from early file based systems to more complex database systems which could be accessed using a query language. The format of data in these systems was alphabetic, numeric or alphanumeric with some special types such as date. This was stored in independent files which required end to end sequential processing. There was no need for a query language. The second transition was to be able to search and select data directly using so called random access files. These files were ultimately joined together to become databases which could be manipulated with a powerful query language. As seen in Chap. 4, relational databases became the de facto standard for this type of data. The next development was the requirement to store complex objects and retrieve them using a query language. Figure 7.1 gives a simple classification of data and queries. As will be seen in the text below, considering object databases to only be associated with simple queries is not correct.

7.2 Problems with Relational Databases

The first big problem with relational databases is that SQL only supports a restricted number of built in types which deal with numbers and strings. Initially the only complex object was BLOB (Binary Large OBject) but now vendors are including

P. Lake, P. Crowther, *Concise Guide to Databases*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-1-4471-5601-7_7,

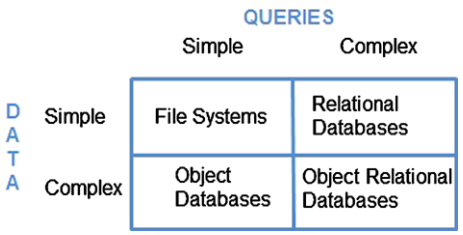


Fig. 7.1 A simple classification

other objects such as CLOB (Character Large OBject) and XML_Type. XML will be discussed later in this chapter. Increasingly there are requirements to deal with different complex objects such as graphics, video, audio and complete documents, all of which can be in a variety of different formats. The volume of this electronic data is rapidly increasing and there is a tendency to store anything that can be stored. Relational databases are not the best structures to store this type of data.

The second problem is that relational tables are essentially flat files linked by joins and do not easily support sets and arrays. Set theory means you can have data which is grouped by some criteria (say people aged between 20 and 29) which can be viewed in terms of its relationships with other groups of data (say smart phone ownership). In this example the intersection of two sets would be the data common to both sets (people aged between 20 and 29 who own a smart phone) while the union would be all the data in both sets (all smart phone owners and all people aged between 20 and 29). Arrays on the other hand bring us back to the first problem in that they store lots of data, often images which have a pixel format.

The third problem is there are certain types of relationships that cannot easily be represented without some kind of work around. For example in Chap. 4 the concept of a hierarchy was introduced where there was a superclass with one or more subclasses associated with it. Attributes and methods from the superclass were inherited by the subclasses. This structure had to be converted into relational tables to make it work in a relational database. Doing this always introduced some in efficiency either in the form of NULL fields or excessive table joins which has an impact on database performance. In the real world things are often organised into hierarchies. For example in a staffing system different classifications of staff will have different attributes, but some data, like staff number, name and address will be common to classes of staff and would be at the top of the hierarchy, to be inherited by the specialist subclass definitions.

The fourth and probably most important issue is that there is often a mismatch between the data access language (SQL) and the host language (for example java). This is termed the impedance mismatch. For example in object oriented programming one of the main concepts is encapsulation. Some encapsulated objects have their representation hidden. These are called private objects and are at odds with relational database representations where access is relative to need rather than an absolute characteristic of the data. Issues such as these are often solved by a programming work around.

Some vendors have solved at least some of these problems, but in a proprietary way rather than an industry standard way. Oracle, for example, has an implementation for sets and operators to manipulate them.

7.3 What Is an Object?

It is probably best to start with a definition and discussion of objects, then move on to object oriented databases, then look at the disadvantages of pure object oriented databases when dealing with lots of 'traditional data'. We will then look at the disadvantages of using relational databases when there are requirements to include and merge with object oriented technologies.

As we saw in Chap. 2, relational database theory is based in mathematics. Humans however tend to recognize 'objects' immediately in terms of their totality or 'wholeness'. Therefore our original relational example of an invoice would be viewed as a single object rather than being composed of a number of 'relations' by the average person.

From a programming point of view, an object is an encapsulation of data and the process which manipulate it. Only the data and methods which need to be seen on an interface are 'visible' to a user. For example, if we think of a television set as an object there are a restricted number of controls and data entry you as a user can interact with. The rest is 'hidden' inside the device.

In object oriented programming when an object instance ends, the data associated with it is lost. The problem with this definition is that the data may 'outlive' the processes. With a digital television, it 'remembers' the stations it has been tuned to, even when it is switched off. This phenomenon of data outliving the object is known as persistence.

In fact in a pure object world a relational database is often represented as a single persistent object. This fits with the concept of object orientation where the complexity of the object is hidden and only its 'public' data and processes can be 'seen'.

The discussion so far, that hasn't really defined an object in database terms. The word 'object' is really a shortened version of what we are talking about and there are two interpretations: object class and object instance. Depending on the object definition language you are using an object class is normally a description of the classes attributes, the messages to which the object responds and the methods which manipulate the object. The data itself is called an object instance. So if we have an object class student, an instance of the class could contain the attributes for the student 'Dorian'.

So far we are still looking at attributes which store simple data such as characters and numbers, however a further complication has been the rise of graphics, images, video and other large data items which require storing, linking and retrieving. As already mentioned relational databases deal with these via a single data type—BLOB (Binary Large OBject) although most vendors included other large

object data types. Relational databases also include other, normally character based attributes to assign keys. This was OK when dealing with something which fitted the relational structure such as a single image of a stock item. It was not such a good solution when dealing with the data required in a graphic based system of multiple related images. In these systems using the traditional primary/foreign key joins becomes highly complicated.

A solution to this is to assign an object an identity that uniquely identifies the object, but unlike a primary key is not stored as an attribute of the object.

7.4 An Object Oriented Solution

The Object Data Management Group (ODMG) which formed in 1991 and disbanded ten years later in 2001 developed a set of standards, the last of which was ODMG 3.0 in 2000. This formed the basis of an industry standard giving guidelines for a SQL like language to manipulate objects, Object Query Language (OQL).

About the same time the Object Oriented Database System Manifesto was produced by Atkinson et al. (1992). This proposed thirteen mandatory features:

- Complex Objects
- Object Identity
- Encapsulation
- Types and Classes
- Type and class hierarchies
- Overriding, overloading and late binding
- Computational completeness
- Extensibility
- Persistence
- Efficiency
- Concurrency
- Reliability
- Declarative query language

Taking each of these concepts in turn

Complex Objects: these are formed through constructor orthogonality. This means a small set of primitive constructs can be combined in a small number of ways to form more complex structures. In simple terms this means complex objects can be formed from other objects by a set of constructors.

Object Identity: Each object has a unique identity assigned by the system. Objects can be shared through references to their identity. This corresponds to the structure proposed by the ODMG.

Encapsulation: In object orientation an object consists of an interface and implementation. The interface defines the way the object looks to the environment. The implantation defines the object data and methods which are used for internal manipulation. The state of the object can only be altered through its interface

although the data structure can have declarative queries applied to it.

Classes: Developers should be able to develop their own classes, for example an 'Employee' class or 'Address' class.

Hierarchies: Many data structures can be regarded as hierarchies. We already saw this in Fig. 4.15 of Chap. 4 where we had a hierarchy consisting of 'Motor Vehicle' as the super class at the top of the hierarchy and a number of subclasses. Objects in the subclass automatically belong to and inherit all the attributes and methods of the superclass although the subclass can have attributes and methods in its own right.

Overriding, overloading and late binding: This is related to hierarchies. In method overriding, a method in the superclass is redefined in the subclass. This allows specialisation in the subclass while preserving the uniform interface defined in the superclass. Overloading is the effect caused by method overriding. This is because it is possible to have multiple methods in the same class that share the same name but have different parameter lists. They must however have the same number of return values. Late binding refers to the overloaded method that is selected at run time and will depend on where you are in the hierarchy.

Computational completeness is a requirement for the method implementation language. Basically it should be possible to express any computable function.

Extensibility: The database has a set of predefined types which developers can use to define new types. This relates to the mandatory feature of complex objects and constructor orthogonality.

Persistence: The data has to survive the program execution. In all object oriented applications a database is regarded as a persistent object.

Efficiency: The database must have index management, data clustering and data buffering to optimise queries.

Reliability: The database must conform to the principles of ACID (see Chap. 2 for details). In other words it must be resilient to user, software and hardware failures. Transactions must be all or nothing in terms of completeness and operations must be logged.

Declarative query language: Non-trivial queries must be able to be expressed consistently through a text or graphical interface. The query language must be vendor independent, in other words be able to work on any possible database.

This of course was a manifesto and together with the ODMG served as guide to the development of object oriented databases from the early 1990's. The resulting query language, Object Query Language (OQL) met the requirements of the manifesto and the ODMG. It looks very much like normal SQL but rather than naming tables in the SELECT clause, object classes are named. The language also has a concept of joins, but because the relationships are established with pointers rather than with a primary/foreign key reference, only the key word JOIN is required. Other structures such as hierarchies are also implemented.

In the 1990's a number of vendor implementations of object-oriented databases appeared. These included O2 (now owned by IBM), JADE and more recent open source products such as db4o.

As an example, in the O2 language (which follows ODMG OQL standard) a simple select clause becomes:

```
SELECT c FROM c IN BBB.customer
        WHERE date_added < '01 JAN 2013'
```

To retrieve data from two objects:

```
SELECT DISTINCT inv.what FROM cl IN BBB.customer,
        inv IN cl.invoice
        WHERE cl.customer_name = "Felsky"
```

For more details see the ODMG OQL User Manual Release 5.0—April 1998.

There have been a number of ways put forward to get around the impedance mismatch problem. Java Persistence Query Language (JP-QL) is an object query language designed for use with Java and to be platform independent. This is not designed to manipulate object oriented databases, but to allow an interface between java and relational databases. For example:

```
public List getStock() throws StockNotFoundException {
   try {
      return em.createQuery(
         "SELECT st FROM Stock st ORDER BY st.code").
            getResultList();
   } catch(Exception ex){
      throw new StockNotFoundException("Could not find stock:"
         + ex.getMessage());
   }
}
```

which has a relational SELECT clause embedded in the middle.

7.5 XML

A final solution presented here is XML (eXtensible Markup Language) which can be used to structure data. The central design goal for XML is to make it easy to communicate information between applications by allowing the semantics of the data to be described in the data itself. XML was designed to overcome the shortcomings of HTML by providing a way to have a domain specific markup language. Many industry specific XML standards have been proposed. In the chemical industry a standard called ChemML has been developed to represent chemicals and their properties. For example in ChemML water (H_2O) is described as:

```
<chem><molecule n ="2">
        <atom n ="2"> H < /atom>
        <atom> O < /atom>
        < /molecule>< /chem>
```

The primarily purpose of XML is to markup content, but it is claimed to have many advantages as a data format because:

- Utilizes unicode.
- Platform independent
- Human readable format makes which makes it easier for development and maintenance (although this is sometimes contested).
- Extensibility, so new information won't cause problems in applications that are based on older versions of the format.
- There exists a large number of off the shelf XML tools

Therefore it is not primarily a database definition language. However code can then be written to manipulate XML objects in the database. The are two main variants: XML Data Reduced (XDR) and XML Schema Definition (XSD). XDR was an interim standard adopted by Microsoft before the introduction of XSD which is the World Wide Web Consortium (W3C) specification. However, because XDR was introduced first (and by Microsoft) there are still a lot of applications which use it as a base.

The following XML Data Reduced (XDR) is in the a Microsoft based version and defines a customer:

```
<Schema name+"customer"
        xmlns="urn:schemas-microsoft-com:xml-data"
        xmlns="urn:schemas-microsoft-com:datatypes">
        <ElementType name="cutomer" model="closed"
                content="eltOny" order="seq">
          <AttributeType name="CustomerDescription" dt:type="string"
                required+"yes"/ >
          <attribute type="CustomerDescription"/ >
          <element type="Customer_Name" minOccurs="1"
                maxOccurs="*"/ >
        < /ElementType>
        <ElementType name="Customer_Name" model="closed"
                content="textOnly" dt:type="string">
          <AttributeType name="Customer_Address" dt:type="string"
                required="yes"/ >
        <attribute type="Customer_Address"/ >
        < /ElementType>
< /Schema>
```

XML Schema Definition (XSD) looks very similar. Here we have a definition for stock:

```
<xs:element name="stock" maxOccurs="unbounded">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="code" type="xs:string"/ >
```

```
        <xs:element name="description" type="xs:string" minOccurs="0"/>
        <xs:element name="price" type="xs:decimal"/ >
      < /xs:sequence>
   < /xs:complexType>
< /xs:element>
```

To search for a record a typical code fragment would look like:

```
FOR $b IN document("cust.xml")//customer
      WHERE $b/name = "Warren Felsky"
      AND $b/postcode = "S11 4RT"
      RETURN $b/customer_id
```

which would return the customer_id's of any customer called Warren Felsky.

Major vendors have already embraced XML as part of their products. Oracle has developed Oracle XML DB, a native XML storage and retrieval technology which is delivered as part of their standard database system. It provides support for all of the key XML standards, including XML, XML Namespaces, DOM, XQuery, SQL/XML and XSLT. This support enables XML-centric application development.

Although not as advanced, SQL Server provides support for XML including support for the XML data type and the ability to specify an XQuery query against XML data.

There are also a number of free and open source XML database systems available such as Sedna which is a free native XML database. It provides a full range of core database services including persistent storage, ACID transactions and flexible XML processing facilities including W3C XQuery implementation (see http://www.sedna.org/ last accessed 26/07/2013).

Another is BaseX which is also free. This is a scalable light-weight XML Database engine supporting XPath and XQuery Processing. It supports the latest W3C Update and Full Text Recommendations (see http://basex.org/home/ last accessed 26/07/2013).

7.6 Object Relational

Object oriented databases were never really adopted by the vendor community despite a flurry of products in the 1990's, but many of the principles ODMG defined were incorporated into relational database management systems resulting in what is often referred to as object relational databases. Oracle Corporation, for example started incorporating object features in Oracle 9i and steadily expanded them and associated development tools such as J developer. In the following examples Oracle statements will be used. Microsoft's SQL Server also has object relational features but these are not fully developed. For example constraints are not inherited in hierarchy structures.

PostgreSQL is another popular object relational database management system which was first devised in the 1980's. The advantage of this system is it is cross platform, free and open source. As a result it is popular with personal computer users. For more information see Obe and Hsu (2012).

It should be remembered that object relational is a work around to give some of the functionality of object orientation by building on top of the existing relational framework. As will be seen, there is always a relational table storing the objects.

7.7 What Is Object Relational?

Figure 7.2 will be used as the basis of the discussion and examples in the rest of the chapter. The code in bold can be typed in as you work through this section so it will be possible to create and experiment with object relational structures as you proceed.

Figure 7.2 shows a number of object classes. An object class consists of a number of attributes and the methods to manipulate them. In the example each of the object classes has a number of methods or operations associated with them. Generally there is one operation for create, retrieve, update and delete (so called CRUDing operations) but this is not a hard and fast rule. It should be noted that hierarchy under **shop_stock** contains only one operation (in subclass **magazine**). This is because each of the subclasses inherits the methods of **shop_stock**. **Magazine** is an exception because periodicals are not reordered based on stock levels instead being ordered on expected sales so returns are kept to a minimum. This operation would override the **add new stock** operation in the **shop_stock** class. This will be illustrated later.

It is probably a good idea at this point to define an instance of an object. This is roughly equivalent to a record in a relational database, so purchase order 45378 of the 3rd December 2012 is an instance of the **purchase_order** class.

The second thing to note about the class diagram is there are no primary or foreign keys. Classes are linked via associations implemented by pointers.

7.8 Classes

When a class is defined it can be used in any other definition where that class is used. In most commercial databases we often define a table called address, but there are many types of addresses for example: delivery address, home address, billing address. Each of these has the same format, but we define them individually in a relational database system. A much better approach would be to have a single data item called 'address' which we could then use whenever an address is needed.

```
CREATE TYPE addr_ty AS OBJECT
        (street         varchar2(60),
        city            varchar2(30),
        postcode        varchar(9));
```

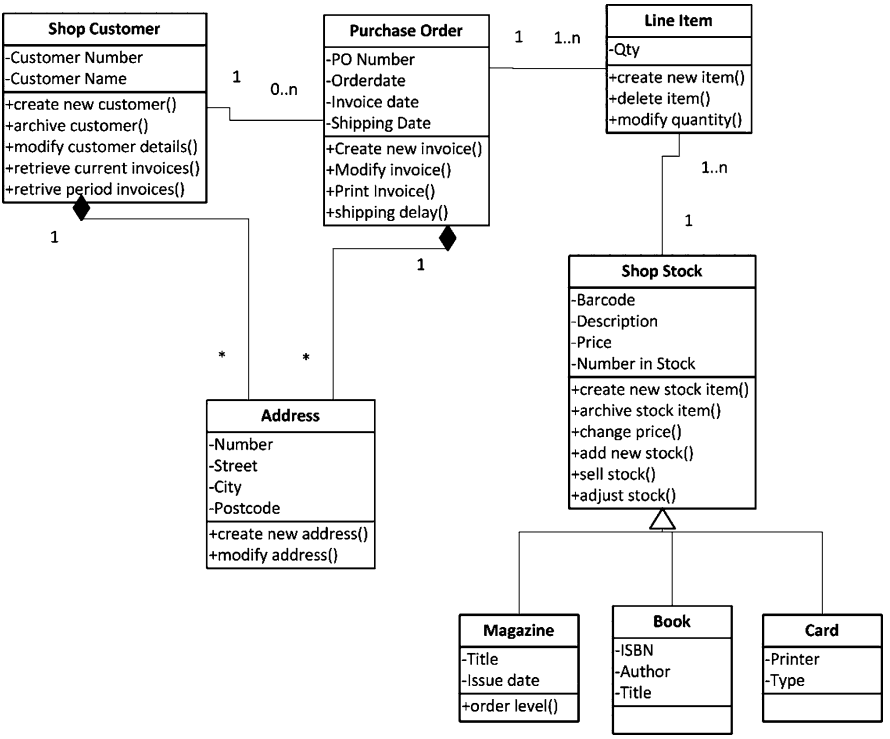


Fig. 7.2 A UML class diagram which will be implemented as an ORDB

The address type could then be used in the definition of 'Shop Customer':

```
CREATE TYPE shop_customer_ty AS OBJECT
        (customer_no    varchar(9),
        name            varchar2(25),
        address         addr_ty);
```

Obviously you must create something before you can use it in another creation statement so in this example you must create the address type first before you can use it in the shop customer type. Once you have created a type you can use it multiple times, for example address is used in purchase order.

Up until this stage the structure looks object oriented but actual data still needs to be stored in a table, so now we need a table to hold the object structures:

```
CREATE TABLE Shop_Customer OF Customer_objtyp
        (PRIMARY KEY (CustNo))
        OBJECT IDENTIFIER IS PRIMARY KEY;
```

The Shop_Customer table contains Customer objects. Each of these has its own object identifier or **OID**. You can either allow Oracle to generate this (**OBJECT IDENTIFIER IS SYSTEM GENERATED)** or specify the objects primary key to serve as it OID with **OBJECT IDENTIFIER IS PRIMARY KEY** which is what we have done in this case. **OBJECT IDENTIFIER IS PRIMARY KEY** should only be used if there is a naturally occurring attribute which lends itself to this role. If there is no natural candidate for a primary key, you should not create one but use **OBJECT IDENTIFIER IS SYSTEM GENERATED**.

Once the table has been created it can be populated with object instances:

```
INSERT INTO customer (shop_customer)
        VALUES
                ('0032478FT',
                'Warren Felsky',
                '8 Mien Place, Sheffield',
                'S1 6GH' );
```

The address information does not have to refer to the type (**addr_ty**) which was created. It is just another data type, only one that has been created by the user.

Once you have stored information of this object instance in this structure you need to be able retrieve it again. In this case you must state what attributes you want returned, but not their type:

```
SELECT c.shop_customer.customer_no ID, c.shop_customer.name NAME,
        c.shop_customer.address ADDRESS
        FROM customer c
        WHERE c.shop_customer.customer_no = '0032478FT'
```

We can use any of the selection constructs we saw in Chap. 4's discussion on SQL. In this case we are looking for the details of customer '**0032478FT**' which will result in the output:

```
ID              NAME                ADDRESS
---------       ------------------------ ------------
0032478FT       Warren Felsky       8 Mien Place, Sheffield.
```

It is important to note that in Oracle, aliases (in this case '**c**') must be used. This allows the substitution of '**c**' for '**customer**' otherwise the statement would look like:

```
SELECT customer.shop_customer.customer_no ID,
        customer.shop_customer.name NAME,
        customer.shop_customer.address ADDRESS
FROM customer
        WHERE customer.shop_customer.customer_no = '0032478FT'
```

which is not only longer but messy and confusing. **ID, NAME** and **ADDRESS** are also aliases for the column headings; otherwise the name of the selected attribute would be used, for example **c.shop_customer.customer_no** would be the column heading instead of the alias **ID**.

7.9 Pointers

In a traditional relational database, tables are linked via joins. The joins use a common primary key—foreign key attribute to link the two tables together. This is one of the hardest concepts for those new to the database field to master. There are two problems with this however. One is you may be creating an attribute for no other reason than to give the data a unique identifier (the primary key). Second you may include that data item in another table for no other reason than you want to join them—it may not logically be an attribute of that table.

In our example, the Purchase Order object contains a pointer to the associated customer object. That is a customer can have many orders but the order can only be associated with one customer—a classic one to many relationship. In a relational database this would be implemented with a table join involving the primary key of the customer table as the foreign key in the order table. In an object relational implementation the link must still be established but it is done with a pointer embedded in one class pointing to another. So an instance of purchase order must include a pointer to the instance of the customer class to which it belongs.

Creating the purchase order object is done as follows. Note this object has **Cust_ref** included to reference the customer object. A pointer is generated to the appropriate object instance in the **shop_customer** table. This will be seen later when we create the tables which hold the objects. Also note you could use the syntax **CREATE OR REPLACE** in the creation of **Address_objtyp** and **Customer_objtyp.** This means you do not have to first **DROP** an object before creating its replacement:

```
CREATE OR REPLACE TYPE Purchase_Order_objtyp AS OBJECT
        (PONo varchar2 (12),
        Cust_ref REF customer_objtyp,
        Orderdate       DATE,
        shipdate        DATE,
        shp_address_obj Address_objtyp);
```

Just like the customer table holds the customer object, we need a table to hold the purchase order object. There is a problem here as a foreign key constraint is required. This illustrates the issues of combining relational and object oriented structures. In this case the **Customer Number** needs to be added to **Purchase Order** as a foreign key. When the purchase order object was created we already specified the **Cust_ref** attribute was a pointer, now we specify which table it is pointing to.

```
CREATE TABLE purchase_order OF Purchase_Order_objtyp
        (PRIMARY KEY (PONo),
        FOREIGN KEY (Cust_ref) REFERENCES Shop_Customer)
        OBJECT IDENTIFIER IS PRIMARY KEY;
```

Now we have created the basic structure for **Customer, Address, Purchase Order** and **Shipping Address**. It also illustrates the trade off for having objects in what is still essentially a relational database environment.

The next stage is to add data:

```
INSERT INTO Shop_Customer
        VALUES (1234, 'Warren Felsky',
        Address_Objtyp ('2 Sherwood Place', 'Latrobe', 'L1 1AC'));
```

Adding data to a table with a **REF** construct to allow one object to reference another is more complex than a relational **INSERT** statement and requires the use of a **SELECT** as shown in the following example where a new purchase order is created. This makes sure that each **Purchase_Order** object instance has a pointer (an object ID or OID) to the associated **Shop_Customer** table object instance.

Once again the order in which you create things is important. You must create a **Shop_Customer** record before you can create a **Purchase_Order** record. In other words you can't have a purchase order with no associated customer.

```
INSERT INTO Purchase_Order
        SELECT '1004', REF(c),
        SYSDATE, '10-MAY-1999',
        address_objtyp( '8 Mien Place',
                'Sheffield',
                'S1 6GH' )
        FROM Shop_Customer c
        WHERE C.CustNo = 1234;
```

In the above example:

- The literal value "1004" is inserted into the **Purchase_Order** table.
- The **REF** function returns the OID (object identifier) from the query on the selected **Shop_Customer** object.
- The OID is now stored as a pointer to the row object in the **Shop_Customer** object table. This is the link from the purchase order object to the associated customer.
- **SYSDATE** captures the current date from the system clock.
- The delivery date is entered.
- The address details are stored.
- **FROM Shop_Customer c WHERE C.CustNo = 1234** is the query associated with the **REF** function.

The referenced value cannot be seen unless the **DREF** function is used. The **DREF** function takes the OID and evaluates the reference to return a value. Any attempt to select objects containing a **REF**, say with:

```
SELECT * FROM purchase_order;
```

would result in a long alphanumeric string representing the hexadecimal value of the pointer rather than a real value.

In the following example we want the customers name, purchase order numbers and shipping address:

```
SELECT (cust_ref).custno,
       (cust_ref).custname,
       p.pono,
       p.shp_address_obj.city
FROM Purchase_Order p
WHERE (cust_ref).custno = 1234;
```

If you don't use the **(cust_ref)** and try to use both tables and not put in a table join, we end up with a Cartesian join. **Cust_ref** is following the REF pointer to the customer.

Note that using **p.cust_ref** does not give the same value as **c.custno** as **cust_ref** is a pointer to the appropriate record in **Shop_Customer** table, not an attribute that stores 'real' data (try it and see!). To see the real value you must use **DEREF:**

```
SELECT DEREF(p.cust_ref)
       FROM Purchase_Order p
       WHERE p.pono = '1001'
```

7.10 Hierarchies and Inheritance

One of the problems in relational databases is you cannot directly implement inheritance. In our example we have items in our shop which share a lot of attributes, for example barcode, price and description. However different types of stock then have very specific attributes.

In the design in Fig. 7.2 we have an object called **shop_stock**. We could populate this with data, but when we consider different types of stock lower in the hierarchy, each has its own unique attributes which need populating.

In a relational structure, as we saw in Chap. 4, we would either include these in description of shop stock which would mean a number of NULL fields or we would create tables which duplicated shop stock but added a few extra attributes. Both of these solutions create either inefficiencies or complexity.

To create the objects involved in a hierarchy, you must start at the top grouping the attributes which are common to all objects. This is the most generalised part of the structure which is then inherited by lower level specialised objects. Therefore

the hierarchy involving the **shop_stock** object is created as follows:

```
CREATE OR REPLACE TYPE shop_stock_objtyp AS OBJECT
        (barcode                NUMBER (13),
        description             VARCHAR2(50),
        price                   NUMBER(7,2),
        number_in_stock         NUMBER(6) )
NOT FINAL;
```

The key words '**NOT FINAL**' means this object has subclasses associated with it. Because the key words '**AS OBJECT**' also appear, it means this is the most generalised type of superclass.

The subclasses are created next. The order is not important and in this case we will create a subclass to hold information about 'book'. **Note**: ***do not*** use the key words '**AS OBJECT**' here. You must however use the key word '**FINAL**' at the bottom of any hierarchy. This specifies that there are no more subclasses below this class.

```
CREATE OR REPLACE TYPE book_type_objtyp
        UNDER shop_stock_objtyp
        (ISBN                   NUMBER(16),
        Author                  VARCHAR2(50),
        Title                   VARCHAR2(30))
FINAL;
```

The book object will inherit all the attributes of the shop stock. Just like all the examples we have seen so far you must store data in tables, so you still have to create a table to hold the **shop_stock_object**. This will contain the structure for the complete hierarchy:

```
CREATE TABLE shop_stock OF shop_stock_objtyp
        (PRIMARY KEY (barcode))
        OBJECT IDENTIFIER IS PRIMARY KEY
```

You only need to worry about the object at the top of the hierarchy when creating the table. However, all the subclasses need to be created before the table is created.

Once you have created the table to hold the hierarchy you can start to populate it. Here you do need to worry about which subclass you are populating, so to insert a new book:

```
INSERT INTO shop_stock (book)
        VALUES('5023765013141', 'hard back', 12.95, 14,
        book_type_object (9781444712247, 'Marco Vichi', 'Death and the Ol
        ive Grove'));
```

The following two examples show how to retrieve data from either the superclass and the selected subclass, or individual attributes from a specified subclass. Is it not possible to retrieve all data from all subclasses with a simple SQL statement

```
SELECT s.barcode, s.price, s.number_in_stock,
        TREAT(VALUE (s) AS book_type_objtyp)
        FROM shop_stock s
        WHERE VALUE (s) IS OF (ONLY book_type_objtyp)
```

retrieves all the records which have an instance in the book subclass. The barcode, price and quantity in stock along with all the subclass attributes will be displayed.

```
SELECT s.barcode, TREAT(VALUE (s) AS book_type_objtyp).isbn
        FROM shop_stock s
        WHERE VALUE (s) IS OF (ONLY book_type_objtyp)
```

retrieves the attribute barcode from the shop_stock superclass and the isbn from the book subclass.

7.11 Aggregation

In object orientation, the structure '**is a part of**' is often required. If we go to our example, a line item is a part of the purchase order. Effectively this is a one to many relationship. In other words a purchase order is an aggregation of line items. In an object relational system this is once again created by using REF's to link individual line items to their purchase order. In this example there is also a reference to the **shop_stock** object so line item details can be retrieved.

Therefore, assuming we have both some purchase orders and shop stock we can create line_item by first creating a line item object:

```
CREATE OR REPLACE TYPE po_line_objtyp AS OBJECT
(PO_ref REF purchase_order_objtyp,
Qty NUMBER,
bar_ref REF shop_stock_objtyp)
```

then creating a table to hold the object (and ultimately the data) by:

```
CREATE TABLE line_item OF po_line_objtyp
        (FOREIGN KEY (PO_ref) REFERENCES purchase_order,
        FOREIGN KEY (bar_ref) REFERENCES shop_stock)
        OBJECT IDENTIFIER IS SYSTEM GENERATED;
```

You should note we haven't nominated an attribute as a **PRIMARY KEY**. We could have introduced an attribute **line_item_number**, but its sole purpose would be allow individual lines to be explicitly named. There is no requirement for this. However,

because there is no **PRIMARY KEY** we must use the **OBJECT IDENTIFIER IS SYSTEM GENERATED** clause.

Once we have created the table, then data can be added:

```
INSERT INTO line_item
        SELECT REF(p), '2', REF(s)
        FROM purchase_order p, shop_stock s
        WHERE p.pono = '1001'
        AND s.barcode = '5023765013141'
```

This code means insert a new line item which belongs to purchase order **1001** and contains an order for some stock with bar code **5023765013141** and we want two of them.

7.12 Encapsulation and Polymorphism

In a definition of an object oriented system, not only is data defined, but so are the methods to manipulate that data. This is called encapsulation. The method can be stored as part of the definition and then retrieved when required. In the following example a method to calculate a person's age will be demonstrated. Normally only date of birth is stored in a database as age can be calculated. Also, you are now older than when you started reading this section so age is highly volatile and if it is stored, it should be stored with a date/time stamp.

In the example here, the code used is written in PL/SQL, Oracles proprietary language. However in some of Oracles other products such as J Developer there has been a move towards java.

Creating the method is a two-step process with the first step being to create the object which holds the method or function:

```
CREATE OR REPLACE type newperson_ty as OBJECT
    (firstname      varchar2(25),
    lastname        varchar2(25),
    birthdate       date,
    MEMBER FUNCTION age(birthdate IN DATE) RETURN NUMBER);
```

We have created a member function called **age** and it uses the data stored in the attribute **birthdate**. It outputs a value in **NUMBER** format.

Once the object which is to hold the function has been created, then the function itself can be defined. Note that in the following example the return value has to be divided by 365 or it will return the persons age in days:

```
CREATE OR REPLACE type body newperson_ty as
        MEMBER FUNCTION age(birthdate in DATE) RETURN
                NUMBER IS
        BEGIN
                RETURN ROUND(SYSDATE - BirthDate)/365;
        END;
        END;
```

Assuming we have created a table called **newperson** and inserted one record with a birthdate of 25th February 1983 and that todays date (**SYSDATE**) is 19 January 2013:

```
SELECT p.person.age (p.person.birthdate) AGE IN YEARS
        FROM newperson p
        WHERE p.lastname = 'Felsky';
```

would result in the output of:

```
AGE IN YEARS
-------------------------------------
                        30
```

Because **age** is a function it can be selected directly like an attribute, but the **birthdate** must be specified as the input parameter.

7.13 Polymorphism

When using structures involving hierarchies and inheritance it is possible to implement ad-hoc polymorphism using function and method overloading. This allows a method in a sub type to override a method in a super type. For example if we had created the **shop_stock_objtyp** with functions:

```
CREATE OR REPLACE TYPE shop_stock_objtyp AS OBJECT
        (Barcode                NUMBER (13),
         Description            VARCHAR2(50),
         Price                  NUMBER(7,2),
        Number_In_Stock         NUMBER(6)
        MEMBER FUNCTION vat() RETURN NUMBER,
        MEMBER FUNCTION printme() return VARCHAR2) )
NOT FINAL;
```

and we create the card subclass:

```
CREATE OR REPLACE TYPE card_type_objtyp
        UNDER shop_stock_objtyp
        (Printer                VARCHAR2(30),
        Type                    VARCHAR2(50),
        Reorder_level           NUMBER(16)
        MEMBER FUNCTION
          number_before_reorder(number_in _stock, re_order_level)
          RETURN NUMBER,
        OVERRIDING MEMBER FUNCTION printme() RETURN
        VARCHAR2)
FINAL;
```

You would have to write the **printme()** function first as you can't override something that does not exist. Assuming you have created the **printme()** function, the **number_before_reorder** function would look like:

```
CREATE OR REPLACE type body card_type_objtyp as
        MEMBER FUNCTION number_before_reorder(number_in _stock,
                re_order_level) RETURN NUMBER IS
        BEGIN
                RETURN (number_in _stock - re_order_level);
        END;
        END;
```

On execution, whenever the **printme** function is called from the superclass the overriding function **number_before_reorder** will execute whenever a card type object is retrieved.

7.14 Support for Object Oriented and Object Relational Database Development

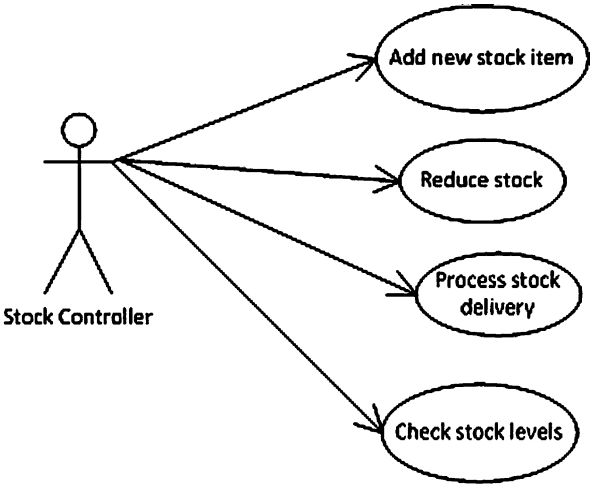
The most common support is in the form of UML (unified modeling language) toolkits. Many of the available UML toolkits claim to have database modeling capabilities. Generally these do not directly link to databases, an exception being Oracles J Developer. They do however provide a useful modeling tools to help with the design of a database. Arguably the two most useful are:

Use Case: A use case diagram shows how a user will interact with s system. This tool can be used to model what the data requirements are for the system and can even be used to define the requirements for the user interface (Fig. 7.3).

Class Diagram: Figure 7.2 is a class diagram and shows the components of a class and the links between classes.

Oracle has introduced JDeveloper which includes UML like tools to develop databases. These include a use case modeler to map user interactions and a class

Fig. 7.3 Use Case diagram



modeling like tool which can dynamically change the structure of underlying tables. It is supports java application development and is offered as an alternative to Oracles PL/SQL centric development tools. Oracle offer this tool as a free add on to its database management system (see http://www.oracle.com/technetwork/developer-tools/jdev/overview/index.html last accessed 26/07/2013).

7.15 Will Object Technology Ever Become Predominant in Database Systems?

Object-oriented capabilities have in effect turned the traditional relational model and the traditional view of normalization on its head in terms of design. Despite this, the relational principles of having entities which are complete and encapsulated remain. We are now adding methods to that encapsulation.

One of the biggest issues is the complexity of the SQL needed to manipulate object relational data despite the structure of the data being simplified. In object relational systems it is made more complex by having to store all structures in tables as ultimately data is still stored in tables rather than as true object instances.

A further issue is the number of systems which have been developed using relational database management systems. The cost of converting these would be high as would the retraining of database developers and administrators.

The most likely scenario is there will be a gradual shift from the relational to the object-oriented model. This is already happening in Oracle with object relational features becoming more common. It may be boosted by the need to store non-alphanumeric objects such as images and sound files.

7.15.1 Review Questions

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

- What is the difference between and object class and an object instance?
- What does an object class define?
- What is meant by persistent data?
- What is an object identifier and what is it used for?
- What is polymorphism?

7.15.2 Group Work Research Activities

These activities require you to research beyond the contents of the book and can be tackled individually or as a discussion group.

Activity 1 If you have not already done so, create the objects and tables using the examples in the text. Then populate and extend the example by reference to Fig. 7.2 and completing the following exercise:

Add two more records to the Shop_Customer table and four more records to the Purchase Order table. Don't forget that a Purchase Order must have an associated customer first, but a customer can have more than one Purchase Order.

The line_Item class in the original design is an association class. To serve its purpose in an ORDBMS it needs to be an object with foreign key links to the Purchase Order and the Shop Stock tables.

- Create the user defined type **Line_Item_objtyp**.
- Create a table to hold **Line_Item_objtyp**. This has two foreign keys, **PONo** to reference Purchase Order and **barcode** to reference the Barcode in Shop Stock. Use the **LineItemNo** as the primary key as trying to use **PONo** and **barcode** as a combined key will cause problems because they are defined with **REF**.)
- Populate **line_Item_objtyp** with at least 2 records for each purchase order.
- List all stock items.
- For purchase order 1001, list the shipping address and line items.
- Modify the previous exercise by adding stock item description.
- Create a new customer.
- Create a purchase order for the customer with one line item.
- Modify the customer's name.
- List the customers name and associated Purchase Order number(s).
- Delete the customer.

Define a function that calculates the number of items left in stock after a line item has been added (Number in **Stock – Qty**). Store the function in **Line_Item** even though you require data from **Shop_Stock**.

Activity 2 Refer to the case study at the end of Chap. 4.

If you have not already done so, download one of the free UML tools available.

1. Draw a class diagram for the scenario.
   Include both attributes and operations.
   Include relationships including hierarchies and aggregation.
2. Create the database.

References

Atkinson M, Bancilhon F, DeWitt D, Dittrich K, Maier D, Zdonik S (1992) The object-oriented database system manifesto. In: Building an object-oriented database system. Morgan Kaufmann, San Mateo

Obe R, Hsu L (2012) PostgreSQL: up and running. O'Reilly, Sebastopol. ISBN 1-4493-2633-1

ODMG OQL (1998) User manual release 5.0. Available at http://www.csd.uwo.ca/courses/CS4411b/pdfO2manuals/oql.pdf. Last accessed 29/04/2013.

Further Reading

Oracle Corporation (2008) A sample application using object-relational features. Available at http://docs.oracle.com/cd/B28359_01/appdev.111/b28371/adobjxmp.htm#BABCCIBC. Last accessed 07/12/2012

Visual paradigm for UML community edition. http://www.visual-paradigm.com/solution/freeumltool/. Last accessed 29/04/2013