Unit-3

Object database and xml

1.Definition and Overview of ODBMS

The **ODBMS** which is an abbreviation for **object oriented database management system**, is the data model in which data is stored in form of objects, which are instances of classes. These classes and objects together makes an object oriented data model.

Components of Object Oriented Data Model:

The OODBMS is based on three major components, namely: Object structure, Object classes, and Object identity. These are explained as following below.

1. Object Structure:

The structure of an object refers to the properties that an object is made up of. These properties of an object are referred to as an attribute. Thus, an object is a real world world entity with certain attributes that makes up the object structure. Also an object encapsulates the data code into a single unit which in turn provides data abstraction by hiding the implementation details from the user.

The object structure is further composed of three types of components: Messages, Methods, and Variables. These are explained as following below.

1. **Messages –**
   A message provides an interface or acts as a communication medium between an object and the outside world. A message can be of two types:
   - **Read-only message:** If the invoked method does not change the value of a variable, then the invoking message is said to be a read-only message.
   - **Update message:** If the invoked method changes the value of a variable, then the invoking message is said to be an update message.
2. **Methods –**
   When a message is passed then the body of code that is executed is known as a method. Every time when a method is executed, it returns a value as output. A method can be of two types:
   - **Read-only method:** When the value of a variable is not affected by a method, then it is known as read-only method.
   - **Update-method:** When the value of a variable changes by a method, then it is known as an update method.
3. **Variables –**
   It stores the data of an object. The data stored in the variables makes the object distinguishable from one another.

2. Object Classes:

An object which is a real world entity is an instance of a class. Hence first we need to define a class and then the objects are made which differ in the values they store but share the same class definition. The objects in turn corresponds to various messages and variables stored in it.

Example –

```
class CLERK
```

```
{ //variables
   char name;
   string address;
   int id;
   int salary;

  //messages
   char get_name();
   string get_address();
   int annual_salary();
};
```

In above example we can see, CLERK is a class that holds the object variables and messages.

An OODBMS also supports inheritance in an extensive manner as in a database there may be many classes with similar methods, variables and messages. Thus, the concept of class hierarchy is maintained to depict the similarities among various classes.

The concept of encapsulation that is the data or information hiding is also supported by object oriented data model. And this data model also provides the facility of abstract data types apart from the built-in data types like char, int, float. ADT's are the user defined data types that hold the values within it and can also have methods attached to it.

Thus, OODBMS provides numerous facilities to it's users, both built-in and user defined. It incorporates the properties of an object oriented data model with a database management system, and supports the concept of programming paradigms like classes and objects along with the support for other concepts like encapsulation, inheritance and the user defined ADT's (abstract data types).

2.Complex Data Types

An instance of a complex data type contains multiple values and provides access to its nested values. Currently, Oracle NoSQL Database supports the following kinds of complex values:

Table 2-3 Complex Data Types

Data Type	Description	Example
ARRAY	In general, an array is an ordered collection of zero or more items. The items of an array are	Type: ARRAY (INTEGER)

Data Type	Description	Example
(T)	called elements. Arrays cannot contain any NULL values. An instance of ARRAY (T) is an array whose elements are all instances of type T. T is called element type of the array.	Type Instance: [600004,560076,01803]
MAP (T)	In general, a map is an unordered collection of zero or more key-item pairs, where all keys are strings. The keys in a map must be unique. The key-item pairs are called fields. The keys are called fields names, and the associated items are called field values. Maps cannot contain any NULL field value. An instance of MAP (T) is a map whose field values are all instance of type T. T is called the value type of the map.	Type: MAP(INTEGER) Type Instance: { "Chennai":600004, "Bangalore":560076, "Boston":01803 }
RECORD (k1 T1 n1, k2 T2 n2, ……., kn Tn nn)	In general, a record is an ordered collection of one or more key-item pairs, where all keys are strings. The keys in a record must be unique. The key-item pairs are called fields. The keys are called fields names, and the associated items are called field values. Records may contain NULL as field value. An instance of RECORD (k1 T1 n1, k2 T2 n2, ……., kn Tn nn) is a record of exactly n fields, where for each field i (a) the field name is ki, (b) the field value is an instance of type Ti, and (c) the field conforms to the nullability property ni, which specifies whether the field value may be NULL or not. Contrary to maps and arrays, it is not possible to add or remove fields from a record. This is because the number of fields and their field names are part of the record type definition associated with a record value.	Type: RECORD(country STRING, zipcode INTEGER, state STRING, street STRING) Type Instance: { "country":"US", "zipcode":600004, "state":"Arizona", "street":"4th Block" }

Example 2-1 Complex Data Type

The following examples illustrate the difference between the way data get stored in various complex data types.

To store the zip codes of multiple cities when the number of zip codes is not known in advance, you can use arrays.

CopyDeclaration:

ARRAY(INTEGER)

CopyExample:

[600004,560076,01803]

To store the names of multiple cities along with their zip codes and the number of zip codes are not known, you can use maps.

CopyDeclaration:

MAP(INTEGER)

CopyExample:

{

"Chennai":600004,

"Bangalore":560076,

"Boston":01803

}

3.Structured Data Types in dbms

Structured Data Types -: A structured data type is a user defined data type with elements that are not atomic rather they are divisible and can be used either separately or as a single unit as per requirements. It is a form of user defined object that contains a sequence of attributes, each of which has a data type. An attribute is a property that helps to describe an instance of a particular type for example if we want to define a structured type called address to store addresses, in which city might be one of the attributes of this structured type.

A structured data type can be used as the type for a column in a regular table, the type for an entire table or as an attribute of another structured type. When used as the type for a table, the table is known as typed table.

CREATE TYPE statement is used to create a structured data type and DROP statement is used to delete the structured data type, Consider 'Emp-Dept' schema discussed previously. In this schema, table 'Emp ' is created with four column namely EmpNo, it is system generated identity column, Name contains name of the employee, Address which is used to hold the address of employee, it is a structured type column of type 'ADDRESS-T' and ProjNo which is a list that stores project number of project taken by employee and eImage that will contain images of an employee.

CREATE TYPE ADDRESS-T as Row (street varchar (12), city varchar (12), state varchar (12),postal code varchar (12))

Now, the 'Emp' table is created having 'ADDRESS-T' and jpeg-amage as data type of Address field as eimage as shown below:

CREATE TABLE EMP (Empno integer system generated, Name varchar (12),Address ADDRESS-T,Proj-no set of (varchar 12),Eimage jpeg-image);

Similarly, all the tables regarding 'company' database is also created by using **DDL statements** shown in below :

CREATE TABLE PROJECT
(Projno integer, Pname varchar (20),
Location REF (address-t) SCOPE LOCS,
Empno set of (integer));
CREATE TABLE LOCS OF ADDRESS-T REF is locid system generated.
CREATE TABLE DEPT
(Deptno integer, Dname varchar (20),
Dlocation REF (address-t) SCOPE LOCS,
Projno set of (integer));
CREATE TABLE CLIPS
(clipno integer, Cname varchar (20),
Objects varchar (20) ARRAY [20],
Budget float,
Projno integer Ctime time);

Above shown table LOCS has an attribute locid which is of ADDRESS-T type and is system generated.

PROJECT is a table which specifies the name, project number along with the location of the project which refers to the LOCS table and also specifies the no. of employees working on the project.

DEPT is a table which specifies name and deptno (unique value) along with location of department. It also specifies the project completed or undertaken by the department. CLIPS is a table which contains the information of the clip showing project demonstration like the objects used in the clip, time of the clip as well as the projno with which clip is associated.

Now suppose there arises a need to show the clip of the project 'HR Management' along with location of the project. Consider the company database, it has a number of inbuilt methods which provides different functionality to the scheme, among which a 'show' method is also available in the schema which shows the, clipno specified, in windows media player. Thus method is used in the following query to produce the result as described :

SELECT C.Cname, C.Projno,B.Location; show (Clipno)
FROM CLIP C PROJECT P

Where C.Projno = P.Projno and P.Pname = 'HR Management'; Suppose , there is a need to find the names of employees along with their images who are living in 'Gandhi Nagar' of 'Ahmedabad'.

SELECT Name, Image FROM EMP WHERE Address.street = 'Gandhinagar' and address.city = Ahmedabad';

The address attribute of EMP table is a structured user defined data type of which street and city are one of the attributes which are used as separate data item for comparison.

Structure Data Types in SQL 99 :-

SQL 99 allows user to define 'distinct' types but they are confined to relational model as these data types are atomic. SQL 99 also provides the facility of defining structured types which extends the relation model, that deals with atomic values only. When we create such structured type, SQL creates a constructor function for the type and creates both 'mutator' and 'observer' methods for the attribute of type.

Constructor function has the same name as structured type with which it is associated. The constructor function has no parameter and returns an instance of type with all of its attribute set to null values.

MUTATOR method exists for each attribute of a structured type. When a mutator method is invoked on an instance of structured type and specify a new value for its associated attribute method, returns a new instance with the attribute update to a new value.

When an 'OBSERVER' method is invoked on an instance of structured type, the method returns the value of attribute for that instance.

Structured types available in SQL 1999 are

ROW (f1t1,f2t2…….fntn)–

1. It represents a row, or a tuple of fields f1,f2,…..fn of types t1,t2,..tn respectively. 'ROW' data type specifies every table as a collection of rows or every table as set of rows or multi-set of rows for example, the 'address-t' is declared as of ROW data type as shown below which contains area, city and state as its components :-

CREATE TTOE address-t

AS ROW (area: varchar (20),city: varchar (20), state : varchar (20))

2. ARRAY [i] : It represents an array of 'i' items of 'base' type for example, the 'objects' field of CLIP table used an array of 10 objects, each of which is of varchar (20) type. A multidimensional array can not be created in SQL 99. An array can be used as component in ROW type as shown but not in array type :-**ROW (pno : integer, object : varchar (20) ARRAY [10])**

3. list of (base) : It represents a list of all items of 'base' type, fo example, PROJECT (Projno : integer, Pname: varchar (25), Emp no: list of (integer))

4. set of (base) : It represents a set of 'base' type items. A set does not contain duplicate elements unlike lists otherwise it is used in the same manner as list.

5. bag of (base) : It represents a bag or a multi-set of base type items. Collection type or build data types are types using list of, set of, bag of and ARRAY. But SQL does not provide any efficient method for manipulation of these collection type objects. So now we will discuss how these data types can be manipulated and also discuss the operations which can be applied on these data types.

Operation on Structured Data :-
Structured data can be manipulated using built in methods for types defined using type constructor. These methods are similar to operations used for data types (atomic) of traditional RDBMS.

1. Operations on Arrays
Array is used in the same manner as in traditional RDBMS. 'Array index' method is used to return the number of elements in the array for example. Suppose we want to find those projects whose clips contain more than 10 items or objects then following query can be used :

```
SELECT P.Pname, P.Projno
FROM project P, Clip C
WHERE CARDINALITY (C.Objects)>10 AND C.Projno = P.Projno
```

The above query select project name and projectno from "PROJECT" whose clips contain more then 10 items which can calculated by using CARDINALITY operation.

2. Operations on Rows
Row type is a collection of fields values whose each fields can be accessed by the same traditional notation for example, address-t.city specify the attribute 'city' of the type address-t. When operation is applied on collection of rows then result obtained is also a collection of values.
If a column or field whose type is ROW (f1t1, f2t2,…….fntn) and c1 fk gives us a list of values whose type is tk. If c1 is a set of rows or a bag of rows then c1 fk give us a set of values of type tk.
Consider 'Emp-Dept' schema in which we have to find the names of those employees who resides in 'Malviya Nagar' of 'New Delhi'.

```
SELECT E Empno,E.Name
FROM Emp E
WHERE E.Address.area ='Malviya Nagar' AND E.Address.city='New Delhi'
AND E.Address.city = 'New Delhi'
```

3. Operations on Sets and Multi-sets
Set and multisets are used in the traditional manner by using =,<,>,>,< comparison operators. An item of a set can be compared by other items using E (belongs to) relation. Two set objects can create a new object using U, (Union Operation). They can also create a new object by subtracting a set of elements from other set by using '-' (set difference operator). Multi-set also uses the same operations as used by the sets but the operations are applied on the number of copies of element into account.

4. Operations on Lists
List includes operations like 'append', 'concatenate', 'head', 'tail' etc. to manipulate the items of list for example, 'concatenate' or 'append' appends one list to another, 'head' returns the first element of list, 'tail' returns the list after removing the first element.

Inheritance is a concept from object-oriented databases. It opens up interesting new possibilities of database design.

Let's create two tables: A table cities and a table capitals. Naturally, capitals are also cities, so you want some way to show the capitals implicitly when you list all cities. If you're really clever you might invent some scheme like this:

```
CREATE TABLE capitals (
  name       text,
  population real,
  altitude   int,    -- (in ft)
  state      char(2)
);


CREATE TABLE non_capitals (
  name       text,
  population real,
  altitude   int     -- (in ft)
);


CREATE VIEW cities AS
  SELECT name, population, altitude FROM capitals
    UNION
  SELECT name, population, altitude FROM non_capitals;
```

This works OK as far as querying goes, but it gets ugly when you need to update several rows, to name one thing.

A better solution is this:

```
CREATE TABLE cities (
  name       text,
  population real,
  altitude   int     -- (in ft)
);


CREATE TABLE capitals (
  state      char(2)
) INHERITS (cities);
```

In this case, a row of capitals *inherits* all columns (name, population, and altitude) from its *parent*, cities. The type of the column name is text, a native PostgreSQL type for variable length character strings. State capitals have an extra column, state, that shows their state. In PostgreSQL, a table can inherit from zero or more other tables.

For example, the following query finds the names of all cities, including state capitals, that are located at an altitude over 500 ft.:

```
SELECT name, altitude
 FROM cities
 WHERE altitude > 500;
```

which returns:

```
  name    | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
 Madison   |      845
(3 rows)
```

On the other hand, the following query finds all the cities that are not state capitals and are situated at an altitude of 500 ft. or higher:

```
SELECT name, altitude
   FROM ONLY cities
   WHERE altitude > 500;
  name    | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
(2 rows)
```

4.Array and Multiset Types in SQL:

Array and Multiset Types in SQL

SQL supports two collection types
- arrays — SQL:1999
- multisets — SQL:2003

```
create type Publisher as
(name varchar(20),
branch varchar(20));
```

```
create type Book as
(title varchar(20),
author_array varchar(20) array[10],
pub_date date,
publisher Publisher,
keyword_set varchar(20) multiset);

create table books of Book;
```

Array and Multiset Types in SQL

Creating and Accessing Collection Values

An array of values can be created in SQL:1999 in this way:

```
array['Silberschatz', 'Korth', 'Sudarshan']
```

A multiset of keywords can be constructed as follows:

```
multiset['computer', 'database', 'SQL']
```

```
insert into books
values ('Compilers', array['Smith', 'Jones'],
new Publisher('McGraw-Hill', 'New York'),
multiset ['parsing', 'analysis'] );
```

But, How we can access or update elements of an array ?

Array and Multiset Types in SQL

Querying Collection-Valued Attributes

find all books that have the word "database" as one of their keywords

```
select title from books
where 'database' in (
unnest (keyword_set) );
```

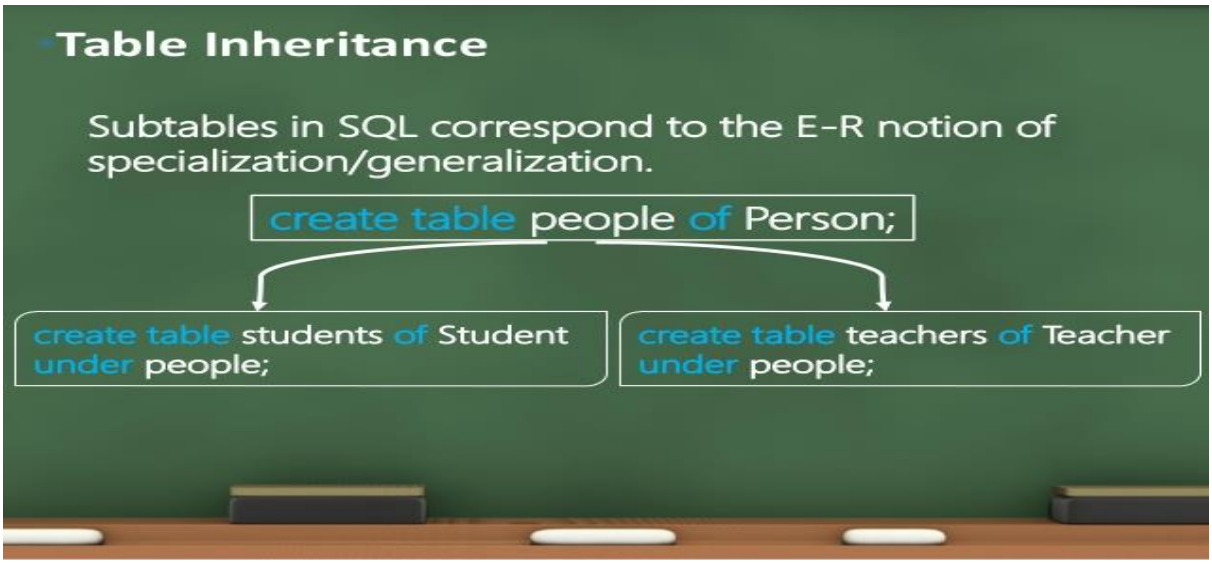
```
select author_array[1],author_array[2],author_array[3]
from books
where title =  'Database System Concepts';
```

Array and Multiset Types in SQL

Querying Collection-Valued Attributes

```
select B.title, A.author
from books as B, unnest (B.authorarray) as
A(author);
```

```
select title, A.author, A.position
from books as B,
unnest(B.author_array) with ordinality as A(author,
position);
```



Difference between RDBMS and OODBMS

RDBMS:
RDBMS stands for Relational Database Management System. It is a database management system based on the relational model i.e. the data and relationships are represented by a collection of inter-related tables. It is a DBMS that enables the user to create, update, administer and interact with a relational database. RDBMS is the basis for SQL, and for all modern database systems like MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access.

OODBMS:
OODBMS stands for Object-Oriented Database Management System. It is a DBMS where data is represented in the form of objects, as used in object-oriented programming. OODB implements object-oriented concepts such as classes of objects, object identity, polymorphism, encapsulation, and inheritance. An object-oriented database stores complex data as compared to relational database. Some examples of OODBMS are Versant Object Database, Objectivity/DB, ObjectStore, Caché and ZODB.

Difference Between RDBMS and OODBMS:

BASIS	RDBMS	OODBMS
Long Form	Stands for Relational Database Management System.	Stands for Object Orientedl Database Management System.
Way of storing data	Stores data in Entities, defined as tables hold specific information.	Stores data as Objects.

BASIS	RDBMS	OODBMS
Data Complexity	Handles comparitively simpler data.	Handles larger and complex data than RDBMS.
Grouping	Entity type refers to the collection of entity that share a common definition.	Class describes a group of objects that have common relationships, behaviors, and also have similar properties.
Data Handeling	RDBMS stores only data.	Stores data as well as methods to use it.
Main Objective	Data Independece from application program.	Data Encapsulation.
Key	A Primary key distinctively identifies an object in a table..	An object identifier (OID) is an unambiguous, long-term name for any type of object or entity.

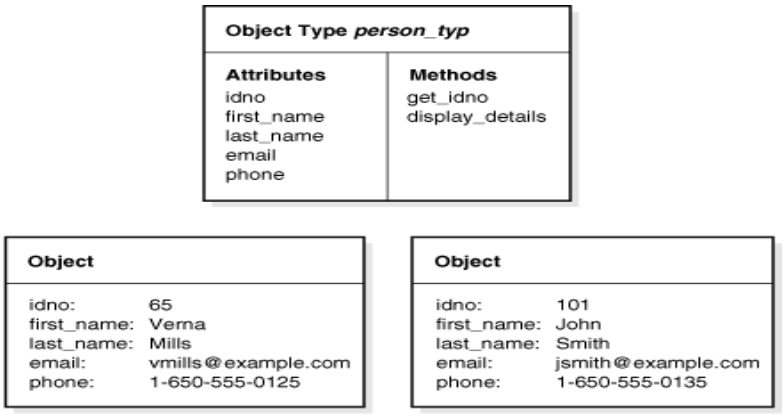
About Object Types

An object type is a kind of data type.

You can use an object in the same ways that you use standard data types such as NUMBER or VARCHAR2. For example, you can specify an object type as the data type of a column in a relational table, and you can declare variables of an object type. The value is a variable or an instance of that type. An object instance is also called an object.

Figure 1-1 shows an object type, person_typ, and two instances of the object type.

Figure 1-1 Object Type and Object Instances



Description of "Figure 1-1 Object Type and Object Instances"

Object types serve as blueprints or templates that define both structure and behavior. Object types are database schema objects, subject to the same kinds of administrative control as other schema objects. Application code can retrieve and manipulate these objects. See Managing Oracle Objects.

You use the CREATE TYPE SQL statement to define object types.

Example 1-1 shows how to create an object type named person_typ. In the example, an object specification and object body are defined. For information on the CREATE TYPE SQL statement and on the CREATE TYPE BODY SQL statement, see *Oracle Database PL/SQL Language Reference*.

Note:

Running Examples: Many examples on this subject can be run using the HR sample schemas. Comments at the beginning of most examples indicate if any previous example code is required.

Refer to *Oracle Database Sample Schemas* for information on how these schemas were created and how you can use them yourself.

Example 1-1 Creating the person_typ Object Type

```
Copy

CREATE TYPE person_typ AS OBJECT (
 idno         NUMBER,
 first_name     VARCHAR2(20),
 last_name     VARCHAR2(25),
 email         VARCHAR2(25),
 phone         VARCHAR2(20),
 MAP MEMBER FUNCTION get_idno RETURN NUMBER,
 MEMBER PROCEDURE display_details ( SELF IN OUT NOCOPY person_typ ));
/


CREATE TYPE BODY person_typ AS
 MAP MEMBER FUNCTION get_idno RETURN NUMBER IS
 BEGIN
```

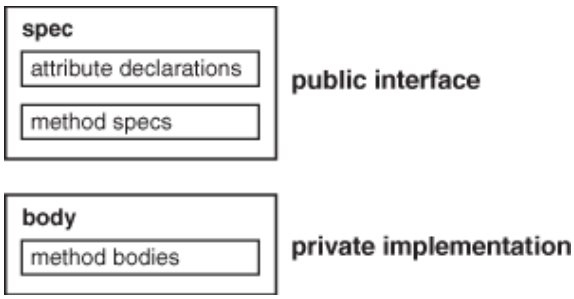
```
  RETURN idno;
 END;
 MEMBER PROCEDURE display_details ( SELF IN OUT NOCOPY person_typ ) IS
 BEGIN
  -- use the PUT_LINE procedure of the DBMS_OUTPUT package to display details
  DBMS_OUTPUT.PUT_LINE(TO_CHAR(idno) || ' ' || first_name || ' ' || last_name);
  DBMS_OUTPUT.PUT_LINE(email || ' '  || phone);
 END;
END;
/
```

Object types differ from the standard data types that are native to a relational database:

- Oracle Database does not supply predefined object types. You define the object types you want by combining built-in types with user-defined ones as shown in Example 1-1.
- Object types are composed of attributes and methods as illustrated in Figure 1-2.
    - Attributes hold the data about an object. Attributes have declared data types which can, in turn, be other object types.
    - Methods are procedures or functions that applications can use to perform operations on the attributes of the object type. Methods are optional. They define the behavior of objects of that type.

Figure 1-2 Object Attributes and Methods

Figure 1-2 shows the relationship of attributes and methods in the spec.



Description of "Figure 1-2 Object Attributes and Methods"

About Object Instances

A variable of an object type is an instance of the type, or an object.

An object has the attributes and methods defined for its type. Because an object instance is a concrete thing, you can assign values to its attributes and call its methods.

Defining an object type does not allocate any storage. After they are defined, object types can be used in SQL statements in most of the same places you use types such as NUMBER or VARCHAR2. Storage is allocated once you create an instance of the object type.

Example 1-2 shows how to create object instances of the person_typ created in Example 1-1, and define a relational table to keep track of these instances as contacts.

Example 1-2 Creating the contacts Table with an Object Type Column

```
Copy

-- requires existing person_typ fr. Ex 1-1

CREATE TABLE contacts (

 contact       person_typ,

 contact_date    DATE );


INSERT INTO contacts VALUES (

 person_typ (65, 'Verna', 'Mills', 'vmills@example.com', '1-650-555-0125'),

 to_date('24 Jun 2003', 'dd Mon YYYY'));
```

The contacts table is a relational table with an object type as the data type of its contact column. Objects that occupy columns of relational tables are called **column objects**.

About Object Methods

Object methods are functions or procedures that you can declare in an object type definition to implement behavior that you want objects of that type to perform.

The general kinds of methods that can be declared in a type definition are:

- Member Methods

  Using member methods, you can provide access to the data of an object, and otherwise define operations that an application performs on the data. To perform an operation, the application calls the appropriate method on the appropriate object.

- Static Methods

  Static methods compare object instances and perform operations that do not use the data of any particular object, but, instead, are global to an object type.

- Constructor Methods

  A default constructor method is implicitly defined for every object type, unless it is overwritten with a user-defined constructor. A constructor method is called on a type to construct or create an object instance of the type.

Example 1-3 show the get_idno() method, created in Example 1-1, to display the Id number of persons in the contacts table:

Example 1-3 Using the get_idno Object Method

```
Copy

-- requires Ex 1-1 and Ex 1-2

SELECT c.contact.get_idno() FROM contacts c;
```

How Objects are Stored in Tables

Objects can be stored in two types of tables:

- Object tables: store only objects

  In an object table, each row represents an object, which is referred to as a **row object**.

- Relational tables: store objects with other table data

  Objects that are stored as columns of a relational table, or are attributes of other objects, are called **column objects**. Example 1-2 shows the contacts table which stores an instance of the person_typ object.

Objects that have meaning outside of the relational database in which they are contained, or objects that are shared among more than one relational database object, should be made referenceable as row objects. That is, such objects should be stored as a row object in an object table instead of in a column of a relational table.

See Also:

- "About Storing Objects as Columns or Rows"
- "Creating and Using Object Tables"

Creating and Using Object Tables

You create object tables using a CREATE TABLE statement.

Example 1-4 shows a CREATE TABLE statement that creates an object table for person_typ objects.

Example 1-4 Creating the person_obj_table Object Table

```
Copy

-- requires Ex. 1-1

CREATE TABLE person_obj_table OF person_typ;
```

You can view this table in two ways:

- As a single-column table, in which each row is a person_typ object, allowing you to perform object-oriented operations.
- As a multi-column table, in which each attribute of the object type person_typ such as idno, first_name, last_name, and so on, occupies a column, allowing you to perform relational operations.

Performing Operations on Object Tables

You can perform various operations on object tables such as inserting objects into the table or selecting objects from the table.

Example 1-5 illustrates several operations on an object table.

Example 1-5 Operations on the person_obj_table Object Table

```
Copy

-- requires Ex. 1-1 and 1-4

INSERT INTO person_obj_table VALUES (

    person_typ(101, 'John', 'Smith', 'jsmith@example.com', '1-650-555-0135') );


SELECT VALUE(p) FROM person_obj_table p

     WHERE p.last_name = 'Smith';


DECLARE

 person person_typ;
```

```
BEGIN -- PL/SQL block for selecting a person and displaying details
  SELECT VALUE(p) INTO person FROM person_obj_table p WHERE p.idno = 101;
  person.display_details();
END;
/
```

The INSERT INTO SQL statement in Example 1-5inserts a person_typ object into the person_obj_table, treating person_obj_table as a multi-column table.

The SELECT SQL statement selects from person_obj_table as a single-column table, using the VALUE function to return rows as object instances.

The PL/SQL block selects a specific person and executes a member function of person_typ to display details about the specified person.

- .

Object Identifiers Used to Identify Row Objects

Object identifiers (OIDs) uniquely identify row objects in object tables.

You cannot directly access object identifiers, but you can make references (REFs) to the object identifiers and directly access the REFs, as discussed in "References to Row Objects".

There are two types of **object identifier**s.

- System-Generated Object Identifiers (default)

  Oracle automatically creates system-generated object identifiers for row objects in object tables unless you choose the primary-key based option.

- Primary-Key Based Object Identifiers

  You have the option to create primary-key based OIDs when you create the table using the CREATE TABLE statement.

Note:

Column objects are identified by the primary key of the row, and, therefore, do not need a specific object identifier.

References to Row Objects

A REF is a logical pointer or reference to a row object that you can construct from an object identifier (OID).

You can use the REF to obtain, examine, or update the object. You can change a REF so that it points to a different object of the same object type hierarchy or assign it a null value.

REFs are Oracle Database built-in data types. REFs and collections of REFs model associations among objects, particularly many-to-one relationships, thus reducing the need for foreign keys. REFs provide an easy mechanism for navigating between objects.

Example 1-6 illustrates a simple use of a REF.

Example 1-6 Using a REF to the emp_person_typ Object

```
Copy

CREATE TYPE emp_person_typ AS OBJECT (
 name     VARCHAR2(30),
 manager  REF emp_person_typ );
/
CREATE TABLE emp_person_obj_table OF emp_person_typ;


INSERT INTO emp_person_obj_table VALUES (
  emp_person_typ ('John Smith', NULL));


INSERT INTO emp_person_obj_table
 SELECT emp_person_typ ('Bob Jones', REF(e))
   FROM emp_person_obj_table e
   WHERE e.name = 'John Smith';
```

This example first creates the emp_person_typ John Smith, with NULL value for a manager. Then it adds the emp_person_typ Bob Jones as John Smith's supervisee.

The following query and its output show the effect:

```
Copy

COLUMN name FORMAT A10
```

```
COLUMN manager FORMAT A50

select * from emp_person_obj_table e;

CopyNAME      MANAGER

---------- --------------------------------------------------

John Smith

Bob Jones  0000220208424E801067C2EABBE040578CE70A0707424E8010

        67C1EABBE040578CE70A0707
```

Example 1-10 shows how to dereference the object, so that Manager appears as a name rather than an object identifier.

Using Scoped REFs

Scoped REF types require less storage space and allow more efficient access than unscoped REF types.

You can constrain a column type, collection element, or object type attribute to reference a specified object table. Use the SQL constraint subclause SCOPE IS when you declare the REF.

Example 1-7 shows REF column contact_ref scoped to person_obj_table which is an object table of type person_typ.

Example 1-7 Creating the contacts_ref Table Using a Scoped REF

```
Copy

-- requires Ex. 1-1, 1-4, and 1-5

CREATE TABLE contacts_ref (

 contact_ref   REF person_typ SCOPE IS person_obj_table,

 contact_date  DATE );
```

To insert a row in the table, you could issue the following:

```
Copy

INSERT INTO contacts_ref
```

```
SELECT REF(p), '26 Jun 2003'
  FROM person_obj_table p
  WHERE p.idno = 101;
```

A REF can be scoped to an object table of the declared type (person_typ in the example) or of any subtype of the declared type. If a REF is scoped to an object table of a subtype, the REF column is effectively constrained to hold only references to instances of the subtype (and its subtypes, if any) in the table..

Checking for Dangling REFs

Dangling REFs are REFs where the object identified by the REF becomes unavailable. Objects are unavailable if they have been deleted or some privilege necessary to them has been deleted.

Use the Oracle Database SQL predicate IS DANGLING to test REFs for dangling REFs.

You can avoid dangling REFs by defining referential integrity constraints.

See Also:

Dereferencing REFs

Accessing the object that the REF refers to is called dereferencing the REF.

There are various ways to dereference a REF, both with and without the DEREF command.

Topics:

- Dereferencing a REF with the DEREF Command
- Dereferencing a Dangling REF
- Dereferencing a REF Implicilty

Dereferencing a REF with the DEREF Command

This example shows how to use the DEREF command to derefence a REF.

Example 1-8 Using DEREF to Dereference a REF

```
CopySELECT DEREF(e.manager) FROM emp_person_obj_table e;

DEREF(E.MANAGER)(NAME, MANAGER)

------------------------------------------------------------

---
```

```
EMP_PERSON_TYP('John Smith', NULL)
```

This example shows that dereferencing a dangling REF returns a null object.

Dereferencing a Dangling REF

You can dereference a dangling REF with the DELETE command.

Dereferencing a dangling REF returns a null object.

Example 1-9 Dereferencing a Dangling Ref

```
Copy

DELETE from person_obj_table WHERE idno = 101;

/

SELECT DEREF(c.contact_ref), c.contact_date FROM contacts_ref c;
```

Dereferencing a REF Implicilty

Oracle Database provides implicit dereferencing of REFs.

For example, to access the manager's name for an employee, you can use a SELECT statement.

Example 1-10 follows the pointer from the person's name and retrieves the manager's name e.manager.name.

Example 1-10 Implicitly Dereferencing a REF

```
Copy

-- requires Ex. 1-6

SELECT e.name, e.manager.name FROM emp_person_obj_table e

  WHERE e.name = 'Bob Jones';
```

Dereferencing the REF in this manner is allowed in SQL, but PL/SQL requires the DEREF keyword as in Example 1-8.

Obtaining a REF to a Row Object

You obtain a REF to a row object by selecting the object from its object table and applying the REF operator.

- Select the object from its object table and apply the REF operator.

Example 1-11 shows how to obtain a REF to the person with an idno equal to 101.

The query returns exactly one row.

Example 1-11 Obtaining a REF to a Row Object

```
Copy

-- requires Ex. 1-1, 1-4, and 1-5

DECLARE

 person_ref REF person_typ;

 person person_typ;

BEGIN


 SELECT REF(p) INTO person_ref

  FROM person_obj_table p

  WHERE p.idno = 101;


  select deref(person_ref) into person from dual;

  person.display_details();


END;

/
```

REF Variables Compared

Two REF variables can be compared if, and only if, the targets that they reference are both of the same declared type, or one is a subtype of the other.

REF variables can only be compared for equality.

Oracle Collections Data Types

For modeling multi-valued attributes and many-to-many relationships, Oracle Database supports these two collection data types:

- Varrays
- Nested Tables

You can use collection types anywhere other data types are used. You can have object attributes of a collection type in addition to columns of a collection type. For example, a purchase order object type might contain a nested table attribute that holds the collection of line items for the purchase order.

To define a collection type, use the CREATE TYPE . . . AS TABLE OF statement.

Example 1-12 shows CREATE TYPE statements that define a collection and an object type.

Example 1-12 Creating the people_typ Collection Data Type

```
Copy

-- requires Ex. 1-1

CREATE TYPE people_typ AS TABLE OF person_typ;

/


CREATE TYPE dept_persons_typ AS OBJECT (

 dept_no    CHAR(5),

 dept_name  CHAR(20),

 dept_mgr   person_typ,

 dept_emps  people_typ);

/
```

Note the following about this example:

- The collection type, people_typ, is specifically a nested table type.
- The dept_persons_typ object type has an attribute dept_emps of people_typ. Each row in the dept_emps nested table is an object of type person_typ which was defined in Example 1-1.

Object Views Used to Access Relational Data

An object view is a way to access relational data using object-relational features.

An object view lets you develop object-oriented applications without changing the underlying relational schema.

You can access objects that belong to an object view in the same way that you access row objects in an object table. Oracle Database also supports **materialized view** objects of user-defined types from data stored in relational schemas and tables.

Object views let you exploit the **polymorphism** that a type hierarchy makes possible. A polymorphic expression takes a value of the expression's declared type or any of that type's subtypes. If you construct a hierarchy of object views that mirrors some or all of the structure of a type hierarchy, you can query any view in the hierarchy to access data at just the level of specialization you are interested in. If you query an object view that has subviews, you can get back polymorphic data—rows for both the type of the view and for its subtypes.

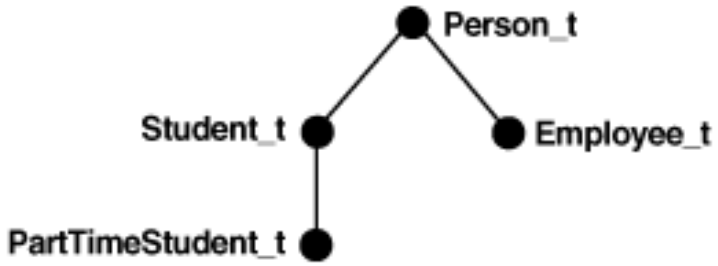
Use of Type Inheritance

Type inheritance enables you to create type hierarchies.

A type hierarchy is a set of successive levels of increasingly specialized subtypes that derive from a common ancestor object type, which is called a supertype. Derived subtypes inherit the features of the parent object type and can extend the parent type definition. The specialized types can add new attributes or methods, or redefine methods inherited from the parent. The resulting type hierarchy provides a higher level of abstraction for managing the complexity of an application model. For example, specialized types of persons, such as a student type or a part-time student type with additional attributes or methods, might be derived from a general person object type.

Figure 1-3 illustrates two subtypes, Student_t and Employee_t, created under Person_t, and the PartTimeStudent_t, a subtype under Student_t.

Figure 1-3 A Type Hierarchy



Description of "Figure 1-3 A Type Hierarchy"

5.SQL Object Types and References

Null Objects and Attributes

An object whose value is NULL is called atomically null.

An **atomically null object** is different from an object that has null values for all its attributes.

In an object with null values, a table column, object attribute, collection, or collection element might be NULL if it has been initialized to NULL or has not been initialized at all. Usually, a NULL value is replaced by an actual value later on. When all the attributes are null, you can still change these attributes and call the object's subprograms or methods. With an atomically null object, you can do neither of these things.

Example 2-1 creates the contacts table and defines the person_typ object type and two instances of this type.

Example 2-1 Inserting NULLs for Objects in a Table

```
Copy

CREATE OR REPLACE TYPE person_typ AS OBJECT (
 idno          NUMBER,
 name          VARCHAR2(30),
 phone         VARCHAR2(20),
 MAP MEMBER FUNCTION get_idno RETURN NUMBER,
 MEMBER PROCEDURE display_details ( SELF IN OUT NOCOPY person_typ ) );
/


CREATE OR REPLACE TYPE BODY person_typ AS
 MAP MEMBER FUNCTION get_idno RETURN NUMBER IS
 BEGIN
  RETURN idno;
 END;
 MEMBER PROCEDURE display_details ( SELF IN OUT NOCOPY person_typ ) IS
 BEGIN
  -- use the PUT_LINE procedure of the DBMS_OUTPUT package to display details
  DBMS_OUTPUT.PUT_LINE(TO_CHAR(idno) || ' - ' || name || ' - ' || phone);
 END;
```

```
END;
/
CREATE TABLE contacts (
  contact       person_typ,
  contact_date   DATE );

INSERT INTO contacts VALUES (
  person_typ (NULL, NULL, NULL), '24 Jun 2003' );

INSERT INTO contacts VALUES (
  NULL, '24 Jun 2003' );
```

Two instances of person_typ are inserted into the table and give two different results. In both cases, Oracle Database allocates space in the contacts table for a new row and sets its DATE column to the value given. But in the first case, Oracle Database allocates space for an object in the contact column and sets each of the object's attributes to NULL. In the second case, Oracle Database sets the person_typ field itself to NULL and does not allocate space for an object.

In some cases, you can omit checks for null values. A table row or row object cannot be null. A nested table of objects cannot contain an element whose value is NULL.

A nested table or array can be null, so you do need to handle that condition. A null collection is different from an empty one, one that has no elements.

Character Length Semantics

Lengths for character types CHAR and VARCHAR2 may be specified as a number of characters, instead of bytes, in object attributes and collections even if some of the characters consist of multiple bytes.

To specify character-denominated lengths for CHAR and VARCHAR2 attributes, you add the qualifier char to the length specification.

Like CHAR and VARCHAR2, NCHAR and NVARCHAR2 may also be used as attribute types in objects and collections. NCHAR and NVARCHAR2 are always implicitly measured in terms of characters, so no char qualifier is used.

For example, the following statement creates an object with both a character-length VARCHAR2 attribute and an NCHAR attribute:

Example 2-2 Creating the employee_typ Object Using a char Qualifier

```
Copy

CREATE OR REPLACE TYPE employee_typ AS OBJECT (
  name       VARCHAR2(30 char),
  language   NCHAR(10),
  phone      VARCHAR2(20) );
/
```

For CHAR and VARCHAR2 attributes whose length is specified without a char qualifier, the NLS_LENGTH_SEMANTICS initialization parameter setting (CHAR or BYTE) indicates the default unit of measure.

Defining Object Tables with Single Constraints

You can define constraints on an object table just as you can on other tables.

You can define constraints on the **leaf-level scalar attribute**s of a column object, with the exception of REFs that are not scoped.

Example 2-3 places a single constraint, an implicit PRIMARY KEY constraint, on the office_id column of the object table office_tab.

Example 2-3 Creating the office_tab Object Table with a Constraint

```
Copy

-- requires Ex. 2-1

CREATE OR REPLACE TYPE location_typ AS OBJECT (
  building_no  NUMBER,
  city         VARCHAR2(40) );
/


CREATE OR REPLACE TYPE office_typ AS OBJECT (
```

```
 office_id    VARCHAR(10),
 office_loc   location_typ,
 occupant     person_typ );/


CREATE TABLE office_tab OF office_typ (
        office_id      PRIMARY KEY );
```

The object type location_typ defined in Example 2-3 is the type of the dept_loc column in the department_mgrs table in Example 2-4.

Defining Object Tables with Multiple Constraints

You can define object tables with multiple constraints.

You can define object tables with multiple constraints.

Example 2-4 Creating the department_mgrs Table with Multiple Constraints

Example 2-4 defines constraints on scalar attributes of the location_typ objects in the table.

```
Copy
-- requires Ex. 2-1 and 2-3
CREATE TABLE department_mgrs (
 dept_no     NUMBER PRIMARY KEY,
 dept_name   CHAR(20),
 dept_mgr    person_typ,
 dept_loc    location_typ,
 CONSTRAINT  dept_loc_cons1
    UNIQUE (dept_loc.building_no, dept_loc.city),
 CONSTRAINT  dept_loc_cons2
     CHECK (dept_loc.city IS NOT NULL) );
```

```
INSERT INTO department_mgrs VALUES
        ( 101, 'Physical Sciences',
         person_typ(65,'Vrinda Mills', '1-1-650-555-0125'),
         location_typ(300, 'Palo Alto'));
```

Defining Indexes for Object Tables

You can define indexes on an object table or on the storage table for a nested table column or attribute just as you can on other tables.

Define indexes on **leaf-level scalar attribute**s of column objects. You can only define indexes on REF attributes or columns if the REF is scoped.

Example 2-5 Creating an Index on an Object Type in a Table

```
Copy
-- requires Ex. 2-1, 2-3,
CREATE TABLE department_loc (
  dept_no     NUMBER PRIMARY KEY,
  dept_name   CHAR(20),
  dept_addr   location_typ );

CREATE INDEX  i_dept_addr1
        ON  department_loc (dept_addr.city);

INSERT INTO department_loc VALUES
        ( 101, 'Physical Sciences',
         location_typ(300, 'Palo Alto'));
INSERT INTO department_loc VALUES
        ( 104, 'Life Sciences',
```

```
        location_typ(400, 'Menlo Park'));

INSERT INTO department_loc VALUES

       ( 103, 'Biological Sciences',

        location_typ(500, 'Redwood Shores'));
```

This example, Example 2-5, indexes city, which is a leaf-level scalar attribute of the column object dept_addr.

Wherever Oracle Database expects a column name in an index definition, you can also specify a scalar attribute of a column object.

For an example of an index on a nested table, see Storing Elements of Nested Tables.

Defining Triggers for Object Tables

You can define triggers on an object table just as you can on other tables.

You cannot define a trigger on the storage table for a nested table column or attribute. You cannot modify LOB values in a trigger body. Otherwise, there are no special restrictions on using object types with triggers.

Example 2-6 defines a trigger on the office_tab table defined in "Defining Object Tables with Single Constraints".

Example 2-6 Creating a Trigger on Objects in a Table

```
Copy

-- requires Ex. 2-1 and 2-3

CREATE TABLE movement (

   idno         NUMBER,

   old_office    location_typ,

   new_office    location_typ );


CREATE TRIGGER trigger1

 BEFORE UPDATE

        OF  office_loc
```

```
        ON  office_tab
  FOR EACH ROW
       WHEN  (new.office_loc.city = 'Redwood Shores')
  BEGIN
    IF :new.office_loc.building_no = 600 THEN
     INSERT INTO movement (idno, old_office, new_office)
      VALUES (:old.occupant.idno, :old.office_loc, :new.office_loc);
    END IF;
  END;/
INSERT INTO office_tab VALUES
   ('BE32', location_typ(300, 'Palo Alto' ),person_typ(280, 'John Chan',
      '415-555-0101'));


UPDATE office_tab set office_loc =location_typ(600, 'Redwood Shores')
  where office_id = 'BE32';


select * from office_tab;


select * from movement;
```

Rules for REF Columns and Attributes

Rules for REF columns and attributes can be enforced by the use of constraints.

In Oracle Database, a REF column or attribute can be unconstrained or constrained using a SCOPE clause or a referential constraint clause. When a REF column is unconstrained, it may store object references to row objects contained in any object table of the corresponding object type.

Oracle Database does not ensure that the object references stored in such columns point to valid and existing row objects. Therefore, REF columns may contain object references that

do not point to any existing row object. Such REF values are referred to as dangling references.

A SCOPE constraint can be applied to a specific object table. All the REF values stored in a column with a SCOPE constraint point at row objects of the table specified in the SCOPE clause. The REF values may, however, be dangling.

A REF column may be constrained with a REFERENTIAL constraint similar to the specification for foreign keys. The rules for referential constraints apply to such columns. That is, the object reference stored in these columns must point to a valid and existing row object in the specified object table.

PRIMARY KEY constraints cannot be specified for REF columns. However, you can specify NOT NULL constraints for such columns.

Name Resolution

There are several ways to resolve names in Oracle Database.

Oracle SQL lets you omit qualifying table names in some relational operations.

For example, if dept_addr is a column in the department_loc table and old_office is a column in the movement table, you can use the following:

```
Copy

SELECT * FROM department_loc WHERE EXISTS

 (SELECT * FROM movement WHERE dept_addr = old_office);
```

Oracle Database determines which table each column belongs to.

Using dot notation, you can qualify the column names with table names or table aliases to make things more maintainable. For example:

Example 2-7 Using the Dot Notation for Name Resolution

```
Copy

-- requires Ex. 2-1, 2-3, 2-5, and 2-6

SELECT * FROM department_loc WHERE EXISTS

 (SELECT * FROM movement WHERE department_loc.dept_addr = movement.old_office);



SELECT * FROM department_loc d WHERE EXISTS
```

```
(SELECT * FROM movement m WHERE d.dept_addr = m.old_office);
```

In some cases, object-relational features require you to specify the table aliases.

When Table Aliases Are Required

Table aliases can be required to avoid problems resolving references.

Oracle Database requires you to use a table alias to qualify any dot-notational reference to subprograms or attributes of objects, to avoid inner capture and similar problems resolving references.

Inner capture is a situation caused by using unqualified names. For example, if you add an assignment column to depts and forget to change the query, Oracle Database automatically recompiles the query so that the inner SELECT uses the assignment column from the depts table.

Use of a table alias is optional when referencing top-level attributes of an object table directly, without using the dot notation. For example, the following statements define two tables that contain the person_typ object type. person_obj_table is an object table for objects of type person_typ, and contacts is a relational table that contains a column of the object person_typ.

The following queries show some correct and incorrect ways to reference attribute idno:

```
Copy#1 SELECT idno FROM person_obj_table; --Correct

#2 SELECT contact.idno FROM contacts; --Illegal

#3 SELECT contacts.contact.idno FROM contacts; --Illegal

#4 SELECT p.contact.idno FROM contacts p; --Correct
```

- In #1, idno is the name of a column of person_obj_table. It references this top-level attribute directly, without using the dot notation, so no table alias is required.
- In #2, idno is the name of an attribute of the person_typ object in the column named contact. This reference uses the dot notation and so requires a table alias, as shown in #4.
- #3 uses the table name itself to qualify the reference. This is incorrect; a table alias is required.

You must qualify a reference to an object attribute or subprogram with a table alias rather than a table name even if the table name is itself qualified by a schema name.

For example, the following expression incorrectly refers to the HR schema, department_loc table, dept_addr column, and city attribute of that column. The expression is incorrect because department_loc is a table name, not an alias.

HR.department_loc.dept_addr.city

The same requirement applies to attribute references that use REFs.

Table aliases should uniquely pick out the same table throughout a query and should not be the same as schema names that could legally appear in the query.

Note:

Oracle recommends that you define table aliases in all UPDATE, DELETE, and SELECT statements and subqueries and use them to qualify column references whether or not the columns contain object types.

Restriction on Using User-Defined Types with a Remote Database

Objects or user-defined types (specifically, types declared with a SQL CREATE TYPE statement, as opposed to types declared within a PL/SQL package) are currently useful only within a single database.

Oracle Database restricts use of a database link as follows:

- You cannot connect to a remote database to select, insert, or update a user-defined type or an object REF on a remote table.

  You can use the CREATE TYPE statement with the optional keyword OID to create a user-specified object identifier (OID) that allows an object type to be used in multiple databases. See the discussion on assigning an OID to an object type in the *Oracle Database Data Cartridge Developer's Guide*.
- You cannot use database links within PL/SQL code to declare a local variable of a remote user-defined type.
- You cannot convey a user-defined type argument or return value in a PL/SQL remote procedure call.

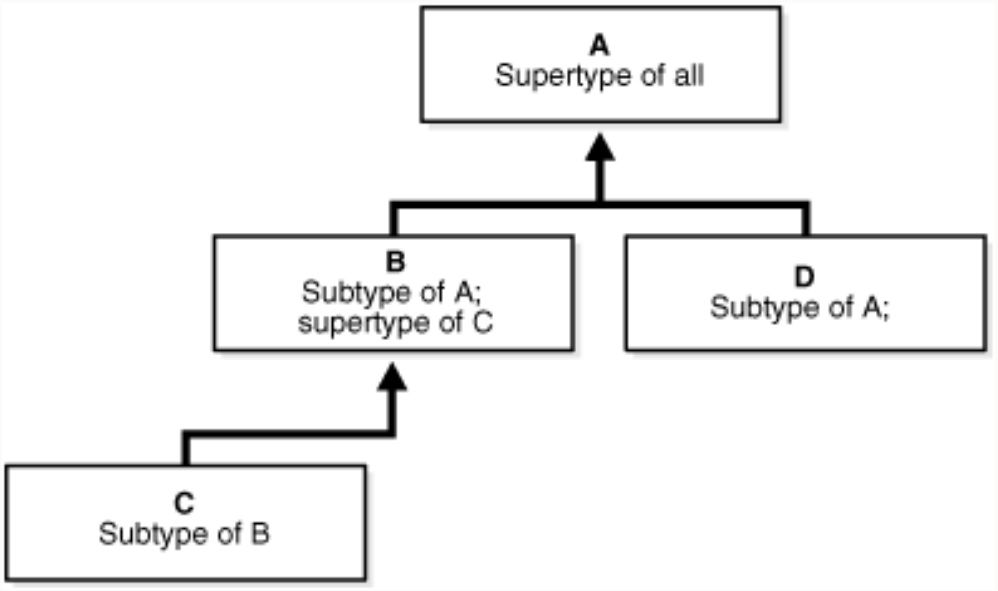
6.Inheritance in SQL Object Types

Supertypes and Subtypes

A subtype can be derived from a supertype either directly or indirectly through intervening levels of other subtypes.

A supertype can have multiple sibling subtypes, but a subtype can have at most one direct parent supertype (single inheritance).

Figure 2-1 Supertypes and Subtypes in Type Hierarchy



Description of "Figure 2-1 Supertypes and Subtypes in Type Hierarchy"

To derive a subtype from a supertype, define a specialized variant of the supertype that adds new attributes and methods to the set inherited from the parent or redefine (override) the inherited methods. For example, from a person_typ object type you might derive the specialized types student_typ and employee_typ. Each of these subtypes is still a person_typ, but a special kind of person. What distinguishes a subtype from its parent supertype is some change made to the attributes or methods that the subtype received from its parent.

Unless a subtype redefines an inherited method, it always contains the same core set of attributes and methods that are in the parent type, plus any attributes and methods that it adds. If a person_typ object type has the three attributes idno, name, and phone and the method get_idno(), then any object type that is derived from person_typ will have these same three attributes and a method get_idno(). If the definition of person_typ changes, so do the definitions of any subtypes.

Subtypes are created using the keyword UNDER as follows:

```
CREATE TYPE student_typ UNDER person_typ
```

You can specialize the attributes or methods of a subtype in these ways:

- Add new attributes that its parent supertype does not have.

  For example, you might specialize student_typ as a special kind of person_typ by adding an attribute for major. A subtype cannot drop or change the type of an attribute it inherited from its parent; it can only add new attributes.
- Add entirely new methods that the parent does not have.
- Change the implementation of some of the methods that a subtype inherits so that the subtype's version executes different code from the parent's.

For example, a ellipse object might define a method calculate(). Two subtypes of ellipse_typ, circle_typ and sphere_typ, might each implement this method in a different way.

The inheritance relationship between a supertype and its subtypes is the source of much of the power of objects and much of their complexity.

Being able to change a method in a supertype and have the change take effect in all the subtypes downstream just by recompiling is very powerful. But this same capability means that you have to consider whether or not you want to allow a type to be specialized or a method to be redefined. Similarly, for a table or column to be able to contain any type in a hierarchy is also powerful, but you must decide whether or not to allow this in a particular case. Also, you may need to constrain DML statements and queries so that they pick out just the range of types that you want from the type hierarchy.

FINAL and NOT FINAL Types and Methods for Inheritance

Object types can be inheritable and methods can be overridden if they are so defined.

For an object type or method to be inheritable, the definition must specify that it is inheritable. For both types and methods, the keywords FINAL or NOT FINAL are used are used to determine inheritability.

- **Object type:** For an object type to be inheritable, thus allowing subtypes to be derived from it, the object definition must specify this.

  NOT FINAL means subtypes can be derived. FINAL, (default) means that no subtypes can be derived from it.

- **Method:** The definition must indicate whether or not it can be overridden.

  NOT FINAL (default) means the method can be overridden. FINAL means that subtypes cannot override it by providing their own implementation.

Creating an Object Type as NOT FINAL with a FINAL Member Function

You can create a NOT FINAL object type with a FINAL member function as in Example 2-12.

Example 2-12 Creating an Object Type as NOT FINAL with a FINAL Member Function

```
Copy

DROP TYPE person_typ FORCE;

-- above necessary if you have previously created object


CREATE OR REPLACE TYPE person_typ AS OBJECT (
  idno        NUMBER,
```

```
  name        VARCHAR2(30),
  phone       VARCHAR2(20),
  FINAL MAP MEMBER FUNCTION get_idno RETURN NUMBER)
NOT FINAL;
/
```

Creating a NOT FINAL Object Type

You can create an object type as NOT FINAL.

Example 2-13 declares person_typ to be a NOT FINAL type and therefore subtypes of person_typ can be defined.

Example 2-13 Creating the person_typ Object Type as NOT FINAL

```
Copy
DROP TYPE person_typ FORCE;
-- above necessary if you have previously created object

CREATE OR REPLACE TYPE person_typ AS OBJECT (
  idno        NUMBER,
  name        VARCHAR2(30),
  phone       VARCHAR2(20))
NOT FINAL;
/
```

Changing a FINAL TYPE to NOT FINAL

You can change inheritance by changing a final type to a not final type and vice versa with an ALTER TYPE statement.

For example, the following statement changes person_typ to a final type:

```
Copy

ALTER TYPE person_typ FINAL;
```

You can only alter a type from NOT FINAL to FINAL if the target type has no subtypes.

Subtype Creation

You create a subtype using a CREATE TYPE statement that specifies the immediate parent of the subtype with the UNDER keyword.

Creating a Parent or Supertype Object

You can create a parent or supertype object using the CREATE TYPE statement.

Example 2-14 provides a parent or supertype person_typ object to demonstrate subtype definitions in Example 2-15, Example 2-18, and Example 2-19.

Note the show() in Example 2-14. In the subtype examples that follow, the show() function of the parent type is overridden to specifications for each subtype using the OVERRIDING keyword.

Example 2-14 Creating the Parent or Supertype person_typ Object

```
Copy

DROP TYPE person_typ FORCE;

-- if created

CREATE OR REPLACE TYPE person_typ AS OBJECT (

 idno        NUMBER,

 name         VARCHAR2(30),

 phone        VARCHAR2(20),

 MAP MEMBER FUNCTION get_idno RETURN NUMBER,

 MEMBER FUNCTION show RETURN VARCHAR2)

 NOT FINAL;

/
```

```
CREATE OR REPLACE TYPE BODY person_typ AS
 MAP MEMBER FUNCTION get_idno RETURN NUMBER IS
 BEGIN
  RETURN idno;
 END;
-- function that can be overriden by subtypes
 MEMBER FUNCTION show RETURN VARCHAR2 IS
 BEGIN
  RETURN 'Id: ' || TO_CHAR(idno) || ', Name: ' || name;
 END;


END;
/
```

Creating a Subtype Object

A subtype inherits the attributes and methods of the supertype.

These are inherited:

- All the attributes declared in or inherited by the supertype.
- Any methods declared in or inherited by supertype.

Example 2-15 defines the student_typ object as a subtype of person_typ, which inherits all the attributes declared in or inherited by person_typ and any methods inherited by or declared in person_typ.

Example 2-15 Creating a student_typ Subtype Using the UNDER Clause

```
Copy
-- requires Ex. 2-14
CREATE TYPE student_typ UNDER person_typ (
  dept_id NUMBER,
```

```
  major VARCHAR2(30),
  OVERRIDING MEMBER FUNCTION show RETURN VARCHAR2)
  NOT FINAL;
/

CREATE TYPE BODY student_typ AS
 OVERRIDING MEMBER FUNCTION show RETURN VARCHAR2 IS
 BEGIN
   RETURN (self AS person_typ).show || ' -- Major: ' || major ;
 END;

END;
/
```

The statement that defines student_typ specializes person_typ by adding two new attributes, dept_id and major and overrides the show method. New attributes declared in a subtype must have names that are different from the names of any attributes or methods declared in any of its supertypes, higher up in its type hierarchy.

Generalized Invocation

Generalized invocation provides a mechanism to invoke a method of a supertype or a parent type, rather than the specific subtype member method.

Example 2-15 demonstrates this using the following syntax:

```
Copy
    (SELF AS person_typ).show
```

The student_typ show method first calls the person_typ show method to do the common actions and then does its own specific action, which is to append '--Major:' to the value returned by the person_typ show method. This way, overriding subtype methods can call corresponding overriding parent type methods to do the common actions before doing their own specific actions.

Methods are invoked just like normal member methods, except that the type name after AS should be the type name of the parent type of the type that the expression evaluates to.

Using Generalized Invocation

In Example 2-16, there is an implicit SELF argument just like the implicit self argument of a normal member method invocation. In this case, it invokes the person_typ show method rather than the specific student_typ show method.

Example 2-16 Using Generalized Invocation

```
Copy

-- Requires Ex. 2-14 and 2-15

DECLARE

 myvar student_typ := student_typ(100, 'Sam', '6505556666', 100, 'Math');

 name VARCHAR2(100);

BEGIN

 name := (myvar AS person_typ).show; --Generalized invocation

END;

/
```

Using Generalized Expression

Generalized expression, like member method invocation, is also supported when a method is invoked with an explicit self argument.

Example 2-17 Using Generalized Expression

```
Copy

-- Requires Ex. 2-14 and 2-15

DECLARE

 myvar2 student_typ := student_typ(101, 'Sam', '6505556666', 100, 'Math');

 name2 VARCHAR2(100);

BEGIN
```

```
 name2 := person_typ.show((myvar2 AS person_typ)); -- Generalized expression

END;

/
```

Double parentheses are used in this example because ((myvar2 AS person_typ)) is both an expression that must be resolved and the parameter of the show function.

NOTE: Constructor methods cannot be invoked using this syntax. Also, the type name that appears after AS in this syntax should be one of the parent types of the type of the expression for which method is being invoked.

This syntax can only be used to invoke corresponding overriding member methods of the parent types.

Creating Multiple Subtypes

A type can have multiple child subtypes, and these subtypes can also have subtypes.

Example 2-18 creates another subtype employee_typ under person_typ in addition to the already existing subtype, student_typ, created in Example 2-15.

Example 2-18 Creating an employee_typ Subtype Using the UNDER Clause

```
Copy

-- requires Ex. 2-14


DROP TYPE employee_typ FORCE;

-- if previously created

CREATE OR REPLACE TYPE employee_typ UNDER person_typ (

   emp_id NUMBER,

   mgr VARCHAR2(30),

   OVERRIDING MEMBER FUNCTION show RETURN VARCHAR2);

/


CREATE OR REPLACE TYPE BODY employee_typ AS
```

```
 OVERRIDING MEMBER FUNCTION show RETURN VARCHAR2 IS
 BEGIN
  RETURN (SELF AS person_typ).show|| ' -- Employee Id: '
       || TO_CHAR(emp_id) || ', Manager: ' || mgr ;
 END;


END;
/
```

Creating a Subtype Under Another Subtype

A subtype can be defined under another subtype.

The new subtype inherits all the attributes and methods that its parent type has, both declared and inherited. Example 2-19 defines a new subtype part_time_student_typ under student_typ created in Example 2-15. The new subtype inherits all the attributes and methods of student_typ and adds another attribute, number_hours.

Example 2-19 Creating a part_time_student_typ Subtype Using the UNDER Clause

```
Copy
CREATE TYPE part_time_student_typ UNDER student_typ (
 number_hours NUMBER,
 OVERRIDING MEMBER FUNCTION show RETURN VARCHAR2);
/


CREATE TYPE BODY part_time_student_typ AS
 OVERRIDING MEMBER FUNCTION show RETURN VARCHAR2 IS
 BEGIN
  RETURN (SELF AS person_typ).show|| ' -- Major: ' || major ||
```

```
        ', Hours: ' || TO_CHAR(number_hours);
  END;


END;
/
```

Creating Tables that Contain Supertype and Subtype Objects

You can create tables that contain supertype and subtype instances.

You can then populate the tables as shown with the person_obj_table in Example 2-20.

Example 2-20 Inserting Values into Substitutable Rows of an Object Table

```
Copy
CREATE TABLE person_obj_table OF person_typ;


INSERT INTO person_obj_table
  VALUES (person_typ(12, 'Bob Jones', '650-555-0130'));


INSERT INTO person_obj_table
  VALUES (student_typ(51, 'Joe Lane', '1-650-555-0140', 12, 'HISTORY'));


INSERT INTO person_obj_table
  VALUES (employee_typ(55, 'Jane Smith', '1-650-555-0144',
                  100, 'Jennifer Nelson'));


INSERT INTO person_obj_table
  VALUES (part_time_student_typ(52, 'Kim Patel', '1-650-555-0135', 14,
```

```
    'PHYSICS', 20));
```

You can call the show() function for the supertype and subtypes in the table with the following:

```
Copy

SELECT p.show() FROM person_obj_table p;
```

The output is similar to:

```
CopyId: 12, Name: Bob Jones

Id: 51, Name: Joe Lane -- Major: HISTORY

Id: 55, Name: Jane Smith -- Employee Id: 100, Manager: Jennifer Nelson

Id: 52, Name: Kim Patel -- Major: PHYSICS, Hours: 20
```

Note that data that the show() method displayed depends on whether the object is a supertype or subtype, and if the show() method of the subtype is overridden. For example, Bob Jones is a person_typ, that is, an supertype. Only his name and Id are displayed. For Joe Lane, a student_typ, his name and Id are provided by the show() function of the supertype, and his major is provided by the overridden show() function of the subtype.

NOT INSTANTIABLE Types and Methods

Types and methods can be declared NOT INSTANTIABLE when they are created.

NOT INSTANTIABLE types and methods:

- NOT INSTANTIABLE Types

  If a type is not instantiable, you cannot instantiate instances of that type. There are no constructors (default or user-defined) for it. You might use this with types intended to serve solely as supertypes from which specialized subtypes are instantiated.

- NOT INSTANTIABLE Methods

  A non-instantiable method serves as a placeholder. It is declared but not implemented in the type. You might define a non-instantiable method when you expect every subtype to override the method in a different way. In this case, there is no point in defining the method in the supertype.

You can alter an instantiable type to a non-instantiable type and vice versa with an ALTER TYPE statement.

A type that contains a non-instantiable method must itself be declared not instantiable, as shown in Example 2-21.

Creating a Non-INSTANTIABLE Object Type

If a subtype does not provide an implementation for every inherited non-instantiable method, the subtype itself, like the supertype, must be declared not instantiable.

A non-instantiable subtype can be defined under an instantiable supertype.

Example 2-21 Creating an Object Type that is NOT INSTANTIABLE

```
Copy

DROP TYPE person_typ FORCE;

-- if previously created

CREATE OR REPLACE TYPE person_typ AS OBJECT (

 idno        NUMBER,

 name         VARCHAR2(30),

 phone        VARCHAR2(20),

 NOT INSTANTIABLE MEMBER FUNCTION get_idno RETURN NUMBER)

 NOT INSTANTIABLE NOT FINAL;/
```

Changing an Object Type to INSTANTIABLE

The ALTER TYPE statement can make a non-instantiable type instantiable.

In Example 2-22 an ALTER TYPE statement makes person_typ instantiable.

Example 2-22 Altering an Object Type to INSTANTIABLE

```
Copy

CREATE OR REPLACE TYPE person_typ AS OBJECT (

 idno        NUMBER,

 name         VARCHAR2(30),
```

```
  phone         VARCHAR2(20))

  NOT INSTANTIABLE NOT FINAL;/

ALTER TYPE person_typ INSTANTIABLE;
```

Changing to a Not Instantiable Type

You can alter an instantiable type to a non-instantiable type only if the type has no columns, views, tables, or instances that reference that type, either directly, or indirectly, through another type or subtype.

You cannot declare a non-instantiable type to be FINAL. This would actually be pointless.

Overloaded and Overridden Methods

A subtype can redefine methods it inherits, and it can also add new methods, including methods with the same name.

Overloading Methods

Adding new methods that have the same names as inherited methods to the subtype is called overloading.

Methods that have the same name but different signatures are called overloads when they exist in the same user-defined type.

A method signature consists of the method's name and the number, types, and the order of the method's formal parameters, including the implicit self parameter.

Overloading is useful when you want to provide a variety of ways of doing something. For example, an ellipse object might overload a calculate() method with another calculate() method to enable calculation of a different shape.

The compiler uses the method signatures to determine which method to call when a type has several overloaded methods.

In the following pseudocode, subtype circle_typ creates an overload of calculate():

```
CopyCREATE TYPE ellipse_typ AS OBJECT (...,

MEMBER PROCEDURE calculate(x NUMBER, x NUMBER),

) NOT FINAL;

CREATE TYPE circle_typ UNDER ellipse_typ (...,

MEMBER PROCEDURE calculate(x NUMBER),
```

```
...);
```

The circle_typ contains two versions of calculate(). One is the inherited version with two NUMBER parameters and the other is the newly created method with one NUMBER parameter.

Overriding and Hiding Methods

Redefining an inherited method to customize its behavior in a subtype is called overriding, in the case of member methods, or hiding, in the case of static methods.

Unlike overloading, you do not create a new method, just redefine an existing one, using the keyword OVERRIDING.

Overriding and hiding redefine an inherited method to make it do something different in the subtype. For example, a subtype circle_typ derived from a ellipse_typ supertype might override a member method calculate() to customize it specifically for calculating the area of a circle. For examples of overriding methods, see "Subtype Creation ".

Overriding and hiding are similar in that, in either case, the version of the method redefined in the subtype eclipses the original version of the same name and signature so that the new version is executed rather than the original one whenever a subtype instance invokes the method. If the subtype itself has subtypes, these inherit the redefined method instead of the original version.

With overriding, the system relies on type information contained in the member method's implicit self argument to dynamically choose the correct version of the method to execute. With hiding, the correct version is identified at compile time, and dynamic dispatch is not necessary. See "Dynamic Method Dispatch".

To override or hide a method, you must preserve its signature. Overloads of a method all have the same name, so the compiler uses the signature of the subtype's method to identify the particular version in the supertype that is superseded.

You signal the override with the OVERRIDING keyword in the CREATE TYPE BODY statement. This is not required when a subtype hides a static method.

In the following pseudocode, the subtype signals that it is overriding method calculate():

```
CopyCREATE TYPE ellipse_typ AS OBJECT (...,

MEMBER PROCEDURE calculate(),

FINAL MEMBER FUNCTION function_mytype(x NUMBER)...

) NOT FINAL;

CREATE TYPE circle_typ UNDER ellipse_typ (...,
```

```
OVERRIDING MEMBER PROCEDURE calculate(),

...);
```

For a diagram of this hierarchy, see Figure 2-2.

Restrictions on Overriding Methods

There are certain restrictions on overriding methods:

- Only methods that are not declared to be final in the supertype can be overridden.
- Order methods may appear only in the root type of a type hierarchy: they may not be redefined (overridden) in subtypes.
- A static method in a subtype may not redefine a member method in the supertype.
- A member method in a subtype may not redefine a static method in the supertype.
- If a method being overridden provides default values for any parameters, then the overriding method must provide the same default values for the same parameters.

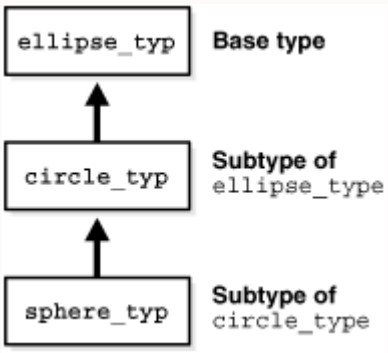
Dynamic Method Dispatch

Dynamic method dispatch refers to the way that method calls are dispatched to the nearest implementation at run time, working up the type hierarchy from the current or specified type.

Dynamic method dispatch is only available when overriding member methods and does not apply to static methods.

With method overriding, a type hierarchy can define multiple implementations of the same method. In the following hierarchy of types ellipse_typ, circle_typ, and sphere_typ, each type might define a calculate() method differently.

Figure 2-2 Hierarchy of Types



Description of "Figure 2-2 Hierarchy of Types"

When one of these methods is invoked, the type of the object instance that invokes it determines which implementation of the method to use. The call is then dispatched to that

implementation for execution. This process of selecting a method implementation is called virtual or dynamic method dispatch because it is done at run time, not at compile time.

The method call works up the type hierarchy: never down. If the call invokes a member method of an object instance, the type of that instance is the current type, and the implementation defined or inherited by that type is used. If the call invokes a static method of a type, the implementation defined or inherited by that specified type is used.

type Substitution in a Type Hierarchy

When you work with types in a type hierarchy, sometimes you need to work at the most general level, for example, to select or update all persons. But at other times, you need to select or update only a specific subtype such as a student, or only persons who are not students.

The (polymorphic) ability to select all persons and get back not only objects whose declared type is person_typ but also objects whose declared subtype is student_typ or employee_typ is called substitutability. A supertype is substitutable if one of its subtypes can substitute or stand in for it in a variable or column whose declared type is the supertype.

In general, types are substitutable. Object attributes, collection elements and REFs are substitutable. An attribute defined as a REF, type, or collection of type person_typ can hold a REF to an instance of, or instances of an instance of person_typ, or an instance of any subtype of person_typ.

This seems expected, given that a subtype is, after all, just a specialized kind of one of its supertypes. Formally, though, a subtype is a type in its own right: it is not the same type as its supertype. A column that holds all persons, including all persons who are students and all persons who are employees, actually holds data of multiple types.

In principle, object attributes, collection elements and REFs are always substitutable: there is no syntax at the level of the type definition to constrain their substitutability to some subtype. You can, however, turn off or constrain substitutability at the storage level, for specific tables and columns.

Column and Row Substitutability

Object type columns and object-type rows in object tables are substitutable, and so are views: a column or row of a specific type can contain instances of that type and any of its subtypes.

-

About Column and Row Substitutability

You can substitute object type columns and object type rows in object tables.

Consider the person_typ type hierarchy such as the one introduced in Example 2-14. You can create an object table of person_typ that contains rows of all types. To do this, you

insert an instance of a given type into an object table using the constructor for that type in the VALUES clause of the INSERT statement as shown in Example 2-20.

Similarly, Example 2-23 shows that a substitutable column of type person_typ can contain instances of all three types, in a relational table or view. The example recreates person, student, and part-time student objects from that type hierarchy and inserts them into the person_typ column contact.

Example 2-23 Inserting Values into Substitutable Columns of a Table

```
Copy

DROP TYPE person_typ FORCE;

-- if previously created


DROP TYPE student_typ FORCE; -- if previously created


DROP TYPE part_time_student_typ FORCE; -- if previously created

DROP TABLE contacts; if previously created

CREATE OR REPLACE TYPE person_typ AS OBJECT (
 idno         NUMBER,
 name         VARCHAR2(30),
 phone        VARCHAR2(20))
 NOT FINAL;/

CREATE TYPE student_typ UNDER person_typ (
   dept_id NUMBER,
   major VARCHAR2(30))
   NOT FINAL;
/

CREATE TYPE part_time_student_typ UNDER student_typ (
 number_hours NUMBER);
```

```
/

CREATE TABLE contacts (

 contact       person_typ,

 contact_date    DATE );


INSERT INTO contacts

 VALUES (person_typ (12, 'Bob Jones', '650-555-0130'), '24 Jun 2003' );


INSERT INTO contacts

 VALUES (student_typ(51, 'Joe Lane', '1-650-555-0178', 12, 'HISTORY'),

      '24 Jun 2003' );


INSERT INTO contacts

 VALUES (part_time_student_typ(52, 'Kim Patel', '1-650-555-0190', 14,

       'PHYSICS', 20), '24 Jun 2003' );
```

A newly created subtype can be stored in any substitutable tables and columns of its supertype, including tables and columns that existed before the subtype was created.

In general, you can access attributes using dot notation. To access attributes of a subtype of a row or column's declared type, you can use the TREAT function. For example:

```
Copy

SELECT TREAT(contact AS student_typ).major FROM contacts;
```

Using OBJECT_VALUE and OBJECT_ID with Substitutable Rows

You can access and identify the object identifier (OID) and value of a substitutable row.

Use the OBJECT_VALUE and OBJECT_ID pseudocolumns to allow access and identify the value and object identifier of a substitutable row in an object table as shown in Example 2-24.

Example 2-24 Using OBJECT_VALUE and OBJECT_ID

```
Copy

DROP TABLE person_obj_table; -- required if previously created

CREATE TABLE person_obj_table OF person_typ;


INSERT INTO person_obj_table

  VALUES (person_typ(20, 'Bob Jones', '650-555-0130'));


SELECT p.object_id, p.object_value FROM person_obj_table p;
```

Subtypes with Attributes of a Supertype

A subtype can have an attribute whose type is the type of a supertype. For example:

Example 2-25 Creating a Subtype with a Supertype Attribute

```
Copy

-- requires Ex 2-22

CREATE TYPE student_typ UNDER person_typ (

   dept_id   NUMBER,

   major     VARCHAR2(30),

   advisor   person_typ);

/
```

XML

xml stands for **E**xtensible **M**arkup **L**anguage. It is a text-based markup language derived from Standard Generalized Markup Language (SGML).

XML tags identify the data and are used to store and organize the data, rather than specifying how to display it like HTML tags, which are used to display the data. XML is not going to replace HTML in the near future, but it introduces new possibilities by adopting many successful features of HTML.

There are three important characteristics of XML that make it useful in a variety of systems and solutions −

- **XML is extensible** − XML allows you to create your own self-descriptive tags, or language, that suits your application.
- **XML carries the data, does not present it** − XML allows you to store the data irrespective of how it will be presented.
- **XML is a public standard** − XML was developed by an organization called the World Wide Web Consortium (W3C) and is available as an open standard.

XML Usage

A short list of XML usage says it all −

- XML can work behind the scene to simplify the creation of HTML documents for large web sites.
- XML can be used to exchange the information between organizations and systems.
- XML can be used for offloading and reloading of databases.
- XML can be used to store and arrange the data, which can customize your data handling needs.
- XML can easily be merged with style sheets to create almost any desired output.
- Virtually, any type of data can be expressed as an XML document.

What is Markup?

XML is a markup language that defines set of rules for encoding documents in a format that is both human-readable and machine-readable. So *what exactly is a markup language?* Markup is information added to a document that enhances its meaning in certain ways, in that it identifies the parts and how they relate to each other. More specifically, a markup language is a set of symbols that can be placed in the text of a document to demarcate and label the parts of that document.

Following example shows how XML markup looks, when embedded in a piece of text −

```
<message>
  <text>Hello, world!</text>
</message>
```

This snippet includes the markup symbols, or the tags such as <message>...</message> and <text>... </text>. The tags <message> and </message> mark the start and the end of the XML code fragment. The tags <text> and </text> surround the text Hello, world!.

Is XML a Programming Language?

A programming language consists of grammar rules and its own vocabulary which is used to create computer programs. These programs instruct the computer to perform specific

tasks. XML does not qualify to be a programming language as it does not perform any computation or algorithms. It is usually stored in a simple text file and is processed by special software that is capable of interpreting XML.

XML - Syntax

In this chapter, we will discuss the simple syntax rules to write an XML document. Following is a complete XML document −

```
<?xml version = "1.0"?>
<contact-info>
  <name>Tanmay Patil</name>
  <company>TutorialsPoint</company>
  <phone>(011) 123-4567</phone>
</contact-info>
```

You can notice there are two kinds of information in the above example −

- Markup, like <contact-info>
- The text, or the character data, *Tutorials Point* and *(040) 123-4567*.

The following diagram depicts the syntax rules to write different types of markup and text in an XML document.

Let us see each component of the above diagram in detail.

XML Declaration

The XML document can optionally have an XML declaration. It is written as follows −

```
<?xml version = "1.0" encoding = "UTF-8"?>
```

Where *version* is the XML version and *encoding* specifies the character encoding used in the document.

Syntax Rules for XML Declaration

- The XML declaration is case sensitive and must begin with "**<?xml>**" where "**xml**" is written in lower-case.
- If document contains XML declaration, then it strictly needs to be the first statement of the XML document.
- The XML declaration strictly needs be the first statement in the XML document.
- An HTTP protocol can override the value of *encoding* that you put in the XML declaration.

Tags and Elements

An XML file is structured by several XML-elements, also called XML-nodes or XML-tags. The names of XML-elements are enclosed in triangular brackets < > as shown below −

```
<element>
```

Syntax Rules for Tags and Elements

Element Syntax − Each XML-element needs to be closed either with start or with end elements as shown below −

```
<element>....</element>
```

or in simple-cases, just this way −

```
<element/>
```

Nesting of Elements − An XML-element can contain multiple XML-elements as its children, but the children elements must not overlap. i.e., an end tag of an element must have the same name as that of the most recent unmatched start tag.

The Following example shows incorrect nested tags −

```
<?xml version = "1.0"?>
<contact-info>
<company>TutorialsPoint
</contact-info>
</company>
```

The Following example shows correct nested tags −

```
<?xml version = "1.0"?>
<contact-info>
  <company>TutorialsPoint</company>
<contact-info>
```

Root Element − An XML document can have only one root element. For example, following is not a correct XML document, because both the **x** and **y** elements occur at the top level without a root element −

```
<x>...</x>
<y>...</y>
```

The Following example shows a correctly formed XML document −

```
<root>
  <x>...</x>
  <y>...</y>
</root>
```

Case Sensitivity − The names of XML-elements are case-sensitive. That means the name of the start and the end elements need to be exactly in the same case.

For example, **<contact-info>** is different from **<Contact-Info>**

XML Attributes

An **attribute** specifies a single property for the element, using a name/value pair. An XML-element can have one or more attributes. For example −

Tutorialspoint!

Here **href** is the attribute name and **http://www.tutorialspoint.com/** is attribute value.

Syntax Rules for XML Attributes

- Attribute names in XML (unlike HTML) are case sensitive. That is, *HREF* and *href* are considered two different XML attributes.
- Same attribute cannot have two values in a syntax. The following example shows incorrect syntax because the attribute *b* is specified twice

–

```
<a b = "x" c = "y" b = "z">....</a>
```

- Attribute names are defined without quotation marks, whereas attribute values must always appear in quotation marks. Following example demonstrates incorrect xml syntax

–

```
<a b = x>....</a>
```

In the above syntax, the attribute value is not defined in quotation marks.

1.Introduction to Oracle XML DB

Oracle XML DB provides Oracle Database with native XML support by encompassing both the SQL and XML data models in an interoperable way.

- Overview of Oracle XML DB
  Oracle XML DB is a set of Oracle Database technologies related to high-performance handling of XML data: storing, generating, accessing, searching, validating, transforming, evolving, and indexing. It provides native XML support by encompassing both the SQL and XML data models in an interoperable way.

- Oracle XML DB Benefits
  Oracle XML DB supports all major XML, SQL, Java, and Internet standards. It provides high performance and scalability for XML operations. It brings database features such as transaction control, data integrity, replication, reliability, availability, security, and scalability to the world of XML.

- Oracle XML DB Architecture
  Oracle XML DB gives you protocol and programmatic access to XML data in the form of local and remote XMLType tables and views. It provides a WebDAV repository with resource versioning and access control.

- Oracle XML DB Features
  Oracle XML DB provides standard database features such as transaction control, data integrity, replication, reliability, availability, security, and scalability, while also allowing for efficient indexing, querying, updating, and searching of XML documents in an XML-centric manner.

- Standards Supported by Oracle XML DB
  Oracle XML DB supports all major XML, SQL, Java, and Internet standards.

- Oracle XML DB Technical Support
  Besides the regular channels of support through your customer representative or consultant, technical support for Oracle Database XML-enabled technologies is available free through the discussion forums Oracle Technology Network (OTN).

- Oracle XML DB Examples
  The examples that illustrate the use of Oracle XML DB and XMLType are based on various database schemas, sample XML documents, and sample XML schemas.

- Oracle XML DB Case Studies and Demonstrations on OTN
  Visit Oracle Technology Network (OTN) to view Oracle XML DB examples, white papers, case studies, and demonstrations.

1.1 Overview of Oracle XML DB

Oracle XML DB is a set of Oracle Database technologies related to high-performance handling of XML data: storing, generating, accessing, searching, validating, transforming, evolving, and indexing. It provides native XML support by encompassing both the SQL and XML data models in an interoperable way.

Oracle XML DB is included as part of Oracle Database starting with Oracle9i Release 2 (9.2).

Oracle XML DB and the XMLType abstract data type make Oracle Database XML-aware. Storing XML data as an XMLType column or table lets the database perform XML-specific operations on the content. This includes XML validation and optimization. XMLType storage allows highly efficient processing of XML content in the database.

Oracle XML DB includes the following features:

- An abstract SQL data type, XMLType, for XML data.
- Enterprise-level Oracle Database features for XML content: reliability, availability, scalability, and security. XML-specific memory management and optimizations.
- Industry-standard ways to access and update XML data. You can use FTP, HTTP(S), and WebDAV to move XML content into and out of Oracle Database. Industry-

standard APIs provide programmatic access and manipulation of XML content using Java, C, and PL/SQL.

- Ways to store, query, update, and transform XML data while accessing it using SQL and XQuery.
- Ways to perform XML operations on SQL data.
- Oracle XML DB Repository: a simple, lightweight repository where you can organize and manage database content, including XML content, using a file/folder/URL metaphor.
- Ways to access and combine data from disparate systems through gateways, using a single, common data model. This reduces the complexity of developing applications that must deal with data from different stores.
- Ways to use Oracle XML DB in conjunction with Oracle XML Developer's Kit (XDK) to build applications that run in the middle tier in either Oracle Fusion Middleware or Oracle Database.

Oracle XML DB functionality is partially based on the Oracle XML Developer's Kit C implementations of the relevant XML standards, such as XML Parser, XSLT Virtual Machine, XML DOM, and XML Schema Validator.

XMLType Data Type

Using XMLType, XML developers can leverage the power of XML standards while working in the context of a relational database, and SQL developers can leverage the power of a relational database while working with XML data.

XMLType is an abstract native SQL data type for XML data. It provides PL/SQL and Java constructors for creating an XMLType instance from a VARCHAR2, CLOB, BLOB, or BFILE instance. And it provides PL/SQL methods for various XML operations.

You can use XMLType as you would any other SQL data type. For example, you can create an XMLType table or view, or an XMLType column in a relational table.

You can use XMLType in PL/SQL stored procedures for parameters, return values, and variables.

You can also manipulate XMLType data using application programming interfaces (APIs) for the Java and C languages, including Java Database Connectivity (JDBC), XQuery for Java (XQJ), and Oracle Data Provider for .NET (ODP.NET).

XMLType is an Oracle Database *object* type, so you can also create a table of XMLType object instances. By default, an XMLType table or column can contain any well-formed XML document.

You can constrain XMLType tables or columns to conform to an XML schema, in which case the database ensures that only XML data that validates against the XML schema is stored in the column or table. invalid documents are excluded.

XMLType Storage Models

XMLType is an *abstract* data type that provides different *storage models* to best fit your data and your use of it. As an abstract data type, your applications and database queries gain in flexibility: the same interface is available for all XMLType operations.

XMLType tables and columns can be stored in these ways:

- **Binary XML** storage (the default) – XMLType data is stored in a post-parse, binary format designed specifically for XML data. Binary XML is compact, post-parse, XML schema-aware XML data. This is also referred to as **post-parse persistence**.
- **Object-relational** storage – XMLType data is stored as a set of objects. This is also referred to as **structured** storage and **object-based persistence**.

XML Schema Support in Oracle XML DB

Support for the World Wide Web Consortium (W3C) XML Schema Recommendation is a key feature in Oracle XML DB.

XML Schema specifies the structure, content, and certain semantics of XML documents. It is described in detail at http://www.w3.org/TR/soap12-part0/.

The W3C Schema Working Group publishes a particular XML schema, often referred to as the **schema for schemas**, that provides the definition, or vocabulary, of the XML Schema language. An **XML schema definition** (**XSD**[Foot 2]), also called an **XML schema**, is an XML document that is compliant with the vocabulary defined by the schema for schemas.

An XML schema uses vocabulary defined by the schema for schemas to create a collection of XML Schema type definitions and element declarations that comprise a vocabulary for describing the contents and structure of a new class of XML documents, the **XML instance documents** that conform to that XML schema.

DTD Support in Oracle XML DB

An XML schema is in general a much more powerful way to define XML document structure than is a DTD. You can nevertheless use DTDs to some extent with Oracle XML DB.

Like an XML schema, A **DTD** is a set of rules that define the allowable structure of an XML document. DTDs are text files that derive their format from SGML. They can be associated with an XML document by using DTD element DOCTYPE or by using an external file through a DOCTYPE reference.

Oracle XML DB uses XML Schema, not DTDs, to define structured mappings to XMLType storage, but XML processors can still access and interpret your DTDs.

Inline DTD Definitions

When an XML instance document has an inline DTD definition, that definition is used during document parsing. Any DTD validations and entity declaration handling are done at this point. However, once parsed, the entity references are replaced with actual values and the original entity reference is lost.

External DTD Definitions

Oracle XML DB supports external DTD definitions if they are stored in Oracle XML DB Repository. Applications needing to process an XML document containing an external DTD definition such as /public/flights.dtd must first ensure that the DTD document is stored in Oracle XML DB at path /public/flights.dtd.

SQL/XML Standard Functions

Oracle XML DB provides the SQL functions that are defined in the SQL/XML standard.

SQL/XML functions fall into two groups:

- Functions that you can use to *generate* XML data from the result of a SQL query. In this book, these are called **SQL/XML publishing functions**. They are also sometimes called **SQL/XML generation functions**.
- Functions that you can use to *query* and *update* XML content as part of normal SQL operations. In this book, these are called **SQL/XML query and update functions**.

Using SQL/XML functions you can address XML content in any part of a SQL statement. These functions use XQuery or XPath expressions to traverse the XML structure and identify the nodes on which to operate. The ability to embed XQuery and XPath expressions in SQL statements greatly simplifies XML access.

Programmatic Access to Oracle XML DB (Java, PL/SQL, and C)

All Oracle XML DB functionality is accessible from C, PL/SQL, and Java.

You can build Web-based applications that take advantage of Oracle XML DB in various ways, including these:

- Using servlets and Java Server Pages (JSP). A typical API accesses data using Java Database Connectivity (JDBC).
- Using Extensible Stylesheet Language (XSL) plus XML Server Pages (XSP). A typical API accesses data in the form of XML documents that are processed using a Document Object Model (DOM) API implementation.

Oracle XML DB supports such styles of application development. It provides Java, PL/SQL, and C implementations of the DOM API. Applications that use JDBC, such as those based on servlets, need prior knowledge of the data structure they are processing. Oracle JDBC drivers allow you to access and update XMLType tables and columns, and call PL/SQL procedures that access Oracle XML DB Repository. Applications that use DOM, such as those based on XSLT transformations, typically require less knowledge of the data structure. DOM-based applications use string names to identify pieces of content, and must dynamically walk through the DOM tree to find the required information. For this, Oracle XML DB supports the use of the DOM API to access and update XMLType columns and tables. Programming to a DOM API is more flexible than programming through JDBC, but it may require more resources at run time.

2 Getting Started with Oracle XML DB

2.1 Oracle XML DB Installation

Oracle XML DB is installed automatically if Database Configuration Assistant (DBCA) is used to build Oracle Database using the general-purpose template.

You can determine whether or not Oracle XML DB is already installed. If it is installed, then the following are true:

- Database schema (user account) XDB exists. To check that, run this query:

```
Copy

SELECT * FROM ALL_USERS;
```

- View RESOURCE_VIEW exists. To check that, use this command:

```
Copy

DESCRIBE RESOURCE_VIEW
```

2.2 Oracle XML DB Use Cases

Oracle XML DB is suited for any application where some or all of the data processed is represented using XML.

Oracle XML DB provides for high-performance database ingestion, storage, processing and retrieval of XML data. It also lets you quickly and easily generate XML from existing relational data. Applications for which Oracle XML DB is particularly suited include the following:

- Business-to-business (B2B) and application-to-application (A2A) integration
- Internet
- Content-management
- Messaging
- Web Services

A typical Oracle XML DB application has at least one of the following characteristics:

- Large numbers of XML documents must be ingested or generated
- Large XML documents must be processed or generated
- High-performance searching is needed, both within a document and across large collections of documents

- High levels of security are needed
- Fine-grained security is needed
- Data processing must use XML documents, and data must be stored in relational tables
- Programming must support open standards such as SQL, XML, XQuery, XPath, and XSL
- Information must be accessed using standard Internet protocols such as FTP, HTTP(S)/WebDAV, and Java Database Connectivity (JDBC)
- XML data must be queried from SQL
- Analytic capabilities must be applied to XML data
- XML documents must be validated against an XML schema

2.3 Application Design Considerations for Oracle XML DB

When planning an Oracle XML DB application it can be worthwhile to consider some preliminary design criteria.

These include the following:

- The ways that you intend to store your XML data
- The structure of your XML data
- The languages used to implement your application
- The ways you intend to process your XML data

However, in general Oracle recommends that you start with the following Oracle XML DB features. For most use cases they are all that you need to consider.

- Storage model – binary XML
- Indexing – XML search index, XMLIndex with structured component
- Database language – SQL, with SQL/XML functions
- XML languages – XQuery and XSLT
- Client APIs – OCI, thin JDBC, SQL .NET

- ### XML Data Storage
  There are several ways to store XML data in Oracle Database.

- ### The Structure of Your XML Data
  How structured your XML data is, and whether it is based on an XML schema, can influence how you store it.

- ### Languages Used to Implement Your Application
  You can program your Oracle XML DB applications in Java (JDBC, Java Servlets) or PL/SQL.

- XML Processing Options
  Oracle XML DB offers a full range of XML processing options.
- Oracle XML DB Repository Access
  Design considerations for applications that use Oracle XML DB Repository include access method, security needs, and whether you need versioning.
- Oracle XML DB Cooperates with Other Database Options and Features
  Oracle XML DB is an integrated part of Oracle Database, and works well with other database options and features.

3.XQuery and Oracle XML DB

XPath Expressions Are XQuery Expressions

The XPath language is a W3C Recommendation for navigating XML documents. It is a subset of the XQuery language: an XPath expression is also an XQuery expression.

XPath models an XML document as a tree of nodes. It provides a set of operations that walk this tree and apply predicates and node-test functions. Applying an XPath expression to an XML document results in a set of nodes. For example, the expression /PO/PONO selects all PONO child elements under the PO root element of a document.

Table 4-1 lists some common constructs used in XPath.

Table 4-1 Common XPath Constructs

XPath Construct	Description
/	Denotes the root of the tree in an XPath expression. For example, /PO refers to the child of the root node whose name is PO.
/	Used as a path separator to identify the child element nodes of a given element node. For example, /PurchaseOrder/Reference identifies Reference elements that are children of PurchaseOrder elements that are children of the root element.
//	Used to identify all descendants of the current node. For example, PurchaseOrder//ShippingInstructions matches any ShippingInstructions element under the PurchaseOrder element.
*	Used as a wildcard to match any child node. For example, /PO/*/STREET matches any street element that is a grandchild of the PO element.
[ ]	Used to denote predicate expressions. XPath supports a rich list of

XPath Construct	Description
	binary operators such as or, and, and not. For example, /PO[PONO = 20 and PNAME = "PO_2"]/SHIPADDR selects the shipping address element of all purchase orders whose purchase-order number is 20 and whose purchase-order name is PO_2. Brackets are also used to denote a position (index). For example, /PO/PONO[2] identifies the second purchase-order number element under the PO root element.
Functions	XPath and XQuery support a set of built-in functions such as substring, round, and not. In addition, these languages provide for extension functions through the use of namespaces. Oracle XQuery extension functions use the namespace prefix ora, for namespace http://xmlns.oracle.com/xdb. See Oracle XQuery Extension Functions.

XQuery Expressions

XQuery expressions are case-sensitive. An XQuery expression is either a *simple* expression or an *updating* expression, the latter being an expression that represents data modification. More precisely, these are the possible XQuery expressions:

- **Basic updating expression –** an insert, delete, replace, or rename expression, or a call to an *updating function* (see the XQuery Update Facility 1.0 Recommendation).
- **Updating expression –** a basic updating expression or an expression (other than a transform expression) that contains another updating expression (this is a recursive definition).
- **Simple expression –** An XQuery 1.0 expression. It does not call for any updating.

The pending update list that results from evaluating a simple expression is empty. The sequence value that results from evaluating an updating expression is empty.

Simple expressions include the following:

- **Primary expression –** literal, variable, or function application. A variable name starts with a dollar-sign ($) – for example, $foo. Literals include numerals, strings, and character or entity references.
- **XPath expression –** Any XPath expression. The XPath 2.0 standard is a subset of XQuery.
- **FLWOR expression –** The most important XQuery expression, composed of the following, in order, from which FLWOR takes its name: for, let, where, order by, return.

- **XQuery sequence** – The comma (,) constructor creates sequences. Sequence-manipulating functions such as union and intersect are also available. All XQuery sequences are effectively **flat**: a nested sequence is treated as its flattened equivalent. Thus, for instance, (1, 2, (3, 4, (5), 6), 7) is treated as (1, 2, 3, 4, 5, 6, 7). A singleton sequence, such as (42), acts the same in most XQuery contexts as does its single item, 42. Remember that the result of any XQuery expression is a sequence.
- **Direct (literal) constructions** – XML element and attribute syntax automatically constructs elements and attributes: what you see is what you get. For example, the XQuery expression <a>33 constructs the XML element <a>33.
- **Computed (dynamic) constructions –** You can construct XML data at run time using computed values. For example, the following XQuery expression constructs this XML data: <foo toto="5"><bar>tata titi</bar> why? </foo>.

- Copy<foo>attribute toto {2+3},
- element bar {"tata", "titi"},
- text {" why? "}</foo>

In this example, element foo is a direct construction; the other constructions are computed. In practice, the arguments to computed constructors are not literals (such as toto and "tata"), but expressions to be evaluated (such as 2+3). Both the name and the value arguments of an element or attribute constructor can be computed. Braces ({, }) are used to mark off an XQuery expression to be evaluated.

- **Conditional expression –** As usual, but remember that each part of the expression is itself an arbitrary expression. For instance, in this conditional expression, each of these subexpressions can be any XQuery expression: something, somethingElse, expression1, and expression2.

- Copy if (something < somethingElse) then expression1 else expression2

- **Arithmetic, relational expression –** As usual, but remember that each relational expression returns a (Boolean[Foot 1]) value. Examples:

- Copy2 + 3
- 42 < $a + 5
- (1, 4) = (1, 2)
- 5 > 3 eq true()

- **Quantifier expression –** Universal (every) and existential (some) quantifier functions provide shortcuts to using a FLWOR expression in some cases. Examples:

- Copyevery $foo in doc("bar.xml")//Whatever satisfies $foo/@bar > 42
- some $toto in (42, 5), $titi in (123, 29, 5) satisfies $toto = $titi

- **Regular expression –** XQuery regular expressions are based on XML Schema 1.0 and Perl. (See Support for XQuery Functions and Operators.)
- **Type expression –** An XQuery expression that represents an XQuery type. Examples: item(), node(), attribute(), element(), document-node(), namespace(), text(), xs:integer, xs:string.[Foot 2]

  Type expressions can have **occurrence indicators**: **?** (optional: zero or one), ***** (zero or more), **+** (one or more). Examples: document-node(element())*, item()+, attribute()?.

  XQuery also provides operators for working with types. These include cast as, castable as, treat as, instance of, typeswitch, and validate. For example, "42" cast as xs:integer is an expression whose value is the integer 42. (It is not, strictly speaking, a type expression, because its value does not represent a type.)
- **Full-text contains expression** – An XQuery expression that represents a full-text search. This expression is provided by the XQuery and XPath Full Text 1.0 Recommendation. A full-text contains expression (FTContainsExpr) supported by Oracle has these parts: a **search context** that specifies the items to search, and a **full-text selection** that filters those items, selecting matches.

  The selection part is itself composed of the following:

  - **Tokens and phrases** used for matching.
  - Optional **match options**, such as the use of stemming.
  - Optional **Boolean operators** for combining full-text selections.
  - Optional constraint operators, such as **positional filters** (e.g. ordered window).

FLWOR Expressions

Just as for XQuery in general, there is a lot to learn about FLWOR expressions in particular. This section provides a brief overview.

FLWOR is the most general expression syntax in XQuery. FLWOR (pronounced "flower") stands for for, let, where, order by, and return. A FLWOR expression has at least one for or let clause and a return clause; single where and order by clauses are optional. Only the return clause can contain an updating expression; the other clauses cannot.

- **for** – Bind one or more variables each to any number of values, in turn. That is, for each variable, iterate, binding the variable to a different value for each iteration.

At each iteration, the variables are bound in the order they appear, so that the value of a variable $earlier that is listed before a variable $later in the for list, can be used in the binding of variable $later. For example, during its second iteration, this expression binds $i to 4 and $j to 6 (2+4):

```
Copy

 for $i in (3, 4), $j in ($i, 2+$i)
```

- **let** – Bind one or more variables.

  Just as with for, a variable can be bound by let to a value computed using another variable that is listed previously in the binding list of the let (or an enclosing for or let). For example, this expression binds $j to 5 (3+2):

```
Copy

let $i := 3, $j := $i + 2
```

- **where** – Filter the for and let variable bindings according to some condition. This is similar to a SQL WHERE clause.
- **order by** – Sort the result of where filtering.
- **return** – Construct a result from the ordered, filtered values. This is the result of the FLWOR expression as a whole. It is a flattened sequence.

  If the return clause contains an updating expression then that expression is evaluated for each tuple generated by the other clauses. The pending update lists from these evaluations are then merged as the result of the FLWOR expression.

Expressions for and let act similarly to a SQL FROM clause. Expression where acts like a SQL WHERE clause Expression order by is similar to ORDER BY in SQL. Expression return is like SELECT in SQL. Except for the two keywords whose names are the same in both languages (where, order by), FLWOR clause order is more or less opposite to the SQL clause order, but the meanings of the corresponding clauses are quite similar.

Using a FLWOR expression (with order by) is the *only* way to construct an XQuery sequence in any order other than document order.

4.Query and Update of XML Data

Using XQuery with Oracle XML DB

XQuery is a very general and expressive language, and SQL/XML functions XMLQuery, XMLTable, XMLExists, and XMLCast combine that power of expression and computation with the strengths of SQL.

You typically use XQuery with Oracle XML DB in the following ways. The examples here are organized to reflect these different uses.

- Query XML data in Oracle XML DB Repository.

  See Querying XML Data in Oracle XML DB Repository Using XQuery.

- Query a relational table or view as if it were XML data. To do this, you use XQuery function fn:collection, passing as argument a URI that uses the URI-scheme name oradb together with the database location of the data.

  See Querying Relational Data Using XQuery and URI Scheme oradb.

- Query XMLType data, possibly decomposing the resulting XML into relational data using function XMLTable.

  See Querying XMLType Data Using XQuery.

Example 5-1 creates Oracle XML DB Repository resources that are used in some of the other examples in this chapter.

Example 5-1 Creating Resources for Examples

```
Copy

DECLARE
 res BOOLEAN;
 empsxmlstring VARCHAR2(300):=
  '<?xml version="1.0"?>
   <emps>
    <emp empno="1" deptno="10" ename="John" salary="21000"/>
    <emp empno="2" deptno="10" ename="Jack" salary="310000"/>
    <emp empno="3" deptno="20" ename="Jill" salary="100001"/>
   </emps>';
 empsxmlnsstring VARCHAR2(300):=
  '<?xml version="1.0"?>
   <emps xmlns="http://example.com">
    <emp empno="1" deptno="10" ename="John" salary="21000"/>
    <emp empno="2" deptno="10" ename="Jack" salary="310000"/>
```

```
    <emp empno="3" deptno="20" ename="Jill" salary="100001"/>
  </emps>';
 deptsxmlstring VARCHAR2(300):=
  '<?xml version="1.0"?>
   <depts>
    <dept deptno="10" dname="Administration"/>
    <dept deptno="20" dname="Marketing"/>
    <dept deptno="30" dname="Purchasing"/>
   </depts>';
BEGIN
 res := DBMS_XDB_REPOS.createResource('/public/emps.xml',   empsxmlstring);
 res := DBMS_XDB_REPOS.createResource('/public/empsns.xml', empsxmlnsstring);
 res := DBMS_XDB_REPOS.createResource('/public/depts.xml',  deptsxmlstring);
END;
/
```

XQuery Sequences Can Contain Data of Any XQuery Type

XQuery is a general *sequence*-manipulation language. Its expressions and their results are not necessarily XML data. An XQuery sequence can contain items of any XQuery type, which includes numbers, strings, Boolean values, dates, and various types of XML node (document-node(), element(), attribute(), text(), namespace(), and so on).

Example 5-2 provides a sampling. It applies SQL/XML function XMLQuery to an XQuery sequence that contains items of several different kinds:

- an integer literal: 1
- a arithmetic expression: 2 + 3
- a string literal: "a"
- a sequence of integers: 100 to 102
- a constructed XML element node: <A>33</A>

Example 5-2 also shows construction of a sequence using the comma operator (**,**) and parentheses (**(**, **)**) for grouping.

The sequence expression 100 to 102 evaluates to the sequence (100, 101, 102), so the argument to XMLQuery here is a sequence that contains a nested sequence. The sequence argument is automatically flattened, as is always the case for XQuery sequences. The argument is, in effect, (1, 5, "a", 100, 101, 102, <A>33</A>).

Example 5-2 XMLQuery Applied to a Sequence of Items of Different Types

```
CopySELECT XMLQuery('(1, 2 + 3, "a", 100 to 102, <A>33</A>)'

            RETURNING CONTENT) AS output

 FROM DUAL;


OUTPUT

-------------------------

1 5 a 100 101 102<A>33</A>


1 row selected.
```

Querying XML Data in Oracle XML DB Repository Using XQuery

Examples are presented that use XQuery with XML data in Oracle XML DB Repository. You use XQuery functions fn:doc and fn:collection to query file and folder resources in the repository, respectively.

The examples here use XQuery function fn:doc to obtain a repository file that contains XML data, and then bind XQuery variables to parts of that data using for and let FLWOR-expression clauses.

Example 5-3 queries two XML-document resources in Oracle XML DB Repository: /public/emps.xml and /public/depts.xml. It illustrates the use of fn:doc and each of the possible FLWOR-expression clauses.

Example 5-4 also uses each of the FLWOR-expression clauses. It shows the use of XQuery functions doc, count, avg, and integer, which are in the namespace for built-in XQuery functions, http://www.w3.org/2003/11/xpath-functions. This namespace is bound to the prefix fn.

Example 5-3 FLOWR Expression Using for, let, order by, where, and return

```
CopySELECT XMLQuery('for $e in doc("/public/emps.xml")/emps/emp

            let $d :=

              doc("/public/depts.xml")//dept[@deptno = $e/@deptno]/@dname

            where $e/@salary > 100000

            order by $e/@empno

            return <emp ename="{$e/@ename}" dept="{$d}"/>'

           RETURNING CONTENT) FROM DUAL;


XMLQUERY('FOR$EINDOC("/PUBLIC/EMPS.XML")/EMPS/EMPLET$D:=DOC("/PUBLIC/DEPTS.
XML")

--------------------------------------------------------------------------------

<emp ename="Jack" dept="Administration"></emp><emp ename="Jill" dept="Marketing"

></emp>


1 row selected.
```

In this example, the various FLWOR clauses perform these operations:

- **for** iterates over the emp elements in /public/emps.xml, binding variable $e to the value of each such element, in turn. That is, it iterates over a general list of employees, binding $e to each employee.
- **let** binds variable $d to a *sequence* consisting of all of the values of dname attributes of those dept elements in /public/emps.xml whose deptno attributes have the same value as the deptno attribute of element $e (this is a join operation). That is, it binds $d to the names of all of the departments that have the same department number as the department of employee $e. (It so happens that the dname value is unique for each deptno value in depts.xml.) Unlike for, let never iterates over values; $d is bound only once in this example.
- Together, for and let produce a stream of tuples ($e, $d), where $e represents an employee and $d represents the names of all of the departments to which that employee belongs —in this case, the unique name of the employee's unique department.
- **where** filters this tuple stream, keeping only tuples with employees whose salary is greater than 100,000.

- **order by** sorts the filtered tuple stream by employee number, empno (in ascending order, by default).
- **return** constructs emp elements, one for each tuple. Attributes ename and dept of these elements are constructed using attribute ename from the input and $d, respectively. The element and attribute names emp and ename in the output have no necessary connection with the same names in the input document emps.xml.

Example 5-4 FLOWR Expression Using Built-In Functions

```
CopySELECT XMLQuery('for $d in fn:doc("/public/depts.xml")/depts/dept/@deptno

        let $e := fn:doc("/public/emps.xml")/emps/emp[@deptno = $d]

        where fn:count($e) > 1

        order by fn:avg($e/@salary) descending

        return

         <big-dept>{$d,

                <headcount>{fn:count($e)}</headcount>,

                <avgsal>{xs:integer(fn:avg($e/@salary))}</avgsal>}

         </big-dept>'

        RETURNING CONTENT) FROM DUAL;


XMLQUERY('FOR$DINFN:DOC("/PUBLIC/DEPTS.XML")/DEPTS/DEPT/@DEPTNOLET$E:=FN:D
OC("/P

--------------------------------------------------------------------------------

<big-dept deptno="10"><headcount>2</headcount><avgsal>165500</avgsal></big-dept>


1 row selected.
```

In this example, the various FLWOR clauses perform these operations:

- **for** iterates over deptno attributes in input document /public/depts.xml, binding variable $d to the value of each such attribute, in turn.

- **let** binds variable $e to a sequence consisting of all of the emp elements in input document /public/emps.xml whose deptno attributes have value $d (this is a join operation).
- Together, for and let produce a stream of tuples ($d, $e), where $d represents a department number and $e represents the set of employees in that department.
- **where** filters this tuple stream, keeping only tuples with more than one employee.
- **order by** sorts the filtered tuple stream by average salary in descending order. The average is computed by applying XQuery function avg (in namespace fn) to the values of attribute salary, which is attached to the emp elements of $e.
- **return** constructs big-dept elements, one for each tuple produced by order by. The text() node of big-dept contains the department number, bound to $d. A headcount child element contains the number of employees, bound to $e, as determined by XQuery function count. An avgsal child element contains the computed average salary.

Querying Relational Data Using XQuery and URI Scheme oradb

Examples are presented that use XQuery to query relational table or view data as if it were XML data. The examples use XQuery function fn:collection, passing as argument a URI that uses the URI-scheme name oradb together with the database location of the data.

Example 5-5 uses Oracle XQuery function fn:collection in a FLWOR expression to query two relational tables, regions and countries. Both tables belong to sample database schema HR. The example also passes scalar SQL value Asia to XQuery variable $regionname. Any SQL expression can be evaluated to produce a value passed to XQuery using PASSING. In this case, the value comes from a SQL*Plus variable, REGION. You must cast the value to the scalar SQL data type expected, in this case, VARCHAR2(40).

In Example 5-5, the various FLWOR clauses perform these operations:

- **for** iterates over sequences of XML elements returned by calls to fn:collection. In the first call, each element corresponds to a row of relational table hr.regions and is bound to variable $i. Similarly, in the second call to fn:collection, $j is bound to successive rows of table hr.countries. Since regions and countries are not XMLType tables, the top-level element corresponding to a row in each table is ROW (a wrapper element). Iteration over the row elements is unordered.
- **where** filters the rows from both tables, keeping only those pairs of rows whose region_id is the same for each table (it performs a join on region_id) and whose region_name is Asia.
- **return** returns the filtered rows from table hr.countries as an XML document containing XML fragments with ROW as their top-level element.

Example 5-6 uses fn:collection within nested FLWOR expressions to query relational data.

In Example 5-6, the various FLWOR clauses perform these operations:

- The outer **for** iterates over the sequence of XML elements returned by fn:collection: each element corresponds to a row of relational table oe.warehouses and is bound to variable $i. Since warehouses is not an XMLType table, the top-level element corresponding to a row is ROW. The iteration over the row elements is unordered.
- The inner **for** iterates, similarly, over a sequence of XML elements returned by fn:collection: each element corresponds to a row of relational table hr.locations and is bound to variable $j.
- **where** filters the tuples ($i, $j), keeping only those whose location_id child is the same for $i and $j (it performs a join on location_id).
- The inner **return** constructs an XQuery sequence of elements STREET_ADDRESS, CITY, and STATE_PROVINCE, all of which are children of locations-table ROW element $j; that is, they are the values of the locations-table columns of the same name.
- The outer **return** wraps the result of the inner return in a Location element, and wraps that in a Warehouse element. It provides the Warehouse element with an id attribute whose value comes from the warehouse_id column of table warehouses.
- Example 5-5 Querying Relational Data as XML Using XMLQuery

- CopyDEFINE REGION = 'Asia'
- SELECT XMLQuery('for $i in **fn:collection("oradb:/HR/REGIONS")**,
- $j in **fn:collection("oradb:/HR/COUNTRIES")**
- where $i/ROW/REGION_ID = $j/ROW/REGION_ID
- and $i/ROW/REGION_NAME = $regionname
- return $j'
- ASIAN_COUNTRIES
- ----------
- <ROW>
- <COUNTRY_ID>AU</COUNTRY_ID>
- <COUNTRY_NAME>Australia</COUNTRY_NAME>
- <REGION_ID>3</REGION_ID>
- </ROW>
- <ROW>
- <COUNTRY_ID>CN</COUNTRY_ID>
- <COUNTRY_NAME>China</COUNTRY_NAME>
- <REGION_ID>3</REGION_ID>
- </ROW>
- <ROW>
- <COUNTRY_ID>HK</COUNTRY_ID>
- <COUNTRY_NAME>HongKong</COUNTRY_NAME>
- <REGION_ID>3</REGION_ID>
- </ROW>
- <ROW>
- <COUNTRY_ID>IN</COUNTRY_ID>
- <COUNTRY_NAME>India</COUNTRY_NAME>

```
  <REGION_ID>3</REGION_ID>
</ROW>
<ROW>
  <COUNTRY_ID>JP</COUNTRY_ID>
  <COUNTRY_NAME>Japan</COUNTRY_NAME>
  <REGION_ID>3</REGION_ID>
</ROW>
<ROW>
  <COUNTRY_ID>SG</COUNTRY_ID>
  <COUNTRY_NAME>Singapore</COUNTRY_NAME>
  <REGION_ID>3</REGION_ID>
</ROW>

1 row selected.
```

Example 5-6 Querying Relational Data as XML Using a Nested FLWOR Expression

```
CopyCONNECT hr
Enter password: password

Connected.

GRANT SELECT ON LOCATIONS TO OE
/
CONNECT oe
Enter password: password

Connected.

SELECT XMLQuery(
        'for $i in fn:collection("oradb:/OE/WAREHOUSES")/ROW
         return <Warehouse id="{$i/WAREHOUSE_ID}">
                  <Location>
                    {for $j in fn:collection("oradb:/HR/LOCATIONS")/ROW
                     where $j/LOCATION_ID eq $i/LOCATION_ID
                     return ($j/STREET_ADDRESS, $j/CITY, $j/STATE_PROVINCE)}
                  </Location>
                </Warehouse>'
        RETURNING CONTENT) FROM DUAL;
```

This query is an example of using nested FLWOR expressions. It accesses relational table warehouses, which is in sample database schema oe, and relational table locations, which is in sample database schema HR. To run this example as user oe, you must first connect as user hr and grant permission to user oe to perform SELECT operations on table locations.

- This produces the following result. (The result is shown here pretty-printed, for clarity.)

- CopyXMLQUERY('FOR$IINFN:COLLECTION("ORADB:/OE/WAREHOUSES")/ROWRETURN<WAREHOUSEID="{$
- ----------
- <Warehouse id="1">
- <Location>
- <STREET_ADDRESS>2014 Jabberwocky Rd</STREET_ADDRESS>
- <CITY>Southlake</CITY>
- <STATE_PROVINCE>Texas</STATE_PROVINCE>
- </Location>
- </Warehouse>
- <Warehouse id="2">
- <Location>
- <STREET_ADDRESS>2011 Interiors Blvd</STREET_ADDRESS>
- <CITY>South San Francisco</CITY>
- <STATE_PROVINCE>California</STATE_PROVINCE>
- </Location>
- </Warehouse>
- <Warehouse id="3">
- <Location>
- <STREET_ADDRESS>2007 Zagora St</STREET_ADDRESS>
- <CITY>South Brunswick</CITY>
- <STATE_PROVINCE>New Jersey</STATE_PROVINCE>
- </Location>
- </Warehouse>
- <Warehouse id="4">
- <Location>
- <STREET_ADDRESS>2004 Charade Rd</STREET_ADDRESS>
- <CITY>Seattle</CITY>
- <STATE_PROVINCE>Washington</STATE_PROVINCE>
- </Location>
- </Warehouse>
- <Warehouse id="5">
- <Location>
- <STREET_ADDRESS>147 Spadina Ave</STREET_ADDRESS>
- <CITY>Toronto</CITY>
- <STATE_PROVINCE>Ontario</STATE_PROVINCE>
- </Location>
- </Warehouse>
- <Warehouse id="6">
- <Location>
- <STREET_ADDRESS>12-98 Victoria Street</STREET_ADDRESS>
- <CITY>Sydney</CITY>
- <STATE_PROVINCE>New South Wales</STATE_PROVINCE>
- </Location>

```
</Warehouse>
<Warehouse id="7">
 <Location>
  <STREET_ADDRESS>Mariano Escobedo 9991</STREET_ADDRESS>
  <CITY>Mexico City</CITY>
  <STATE_PROVINCE>Distrito Federal,</STATE_PROVINCE>
 </Location>
</Warehouse>
<Warehouse id="8">
 <Location>
  <STREET_ADDRESS>40-5-12 Laogianggen</STREET_ADDRESS>
  <CITY>Beijing</CITY>
 </Location>
</Warehouse>
<Warehouse id="9">
 <Location>
  <STREET_ADDRESS>1298 Vileparle (E)</STREET_ADDRESS>
  <CITY>Bombay</CITY>
  <STATE_PROVINCE>Maharashtra</STATE_PROVINCE>
 </Location>
</Warehouse>

1 row selected.
```

Example 5-7 Querying Relational Data as XML Using XMLTable

```
CopySELECT *
 FROM XMLTable(
      'for $i in fn:collection("oradb:/OE/WAREHOUSES")/ROW
       return <Warehouse id="{$i/WAREHOUSE_ID}">
              <Location>
               {for $j in fn:collection("oradb:/HR/LOCATIONS")/ROW
                where $j/LOCATION_ID eq $i/LOCATION_ID
                return ($j/STREET_ADDRESS, $j/CITY, $j/STATE_PROVINCE)}
              </Location>
             </Warehouse>');
```

This produces the same result as Example 5-6, except that each Warehouse element is output as a separate row, instead of all Warehouse elements being output together in a single row.

```
CopyCOLUMN_VALUE
------------------------------------------------------
<Warehouse id="1">
 <Location>
  <STREET_ADDRESS>2014 Jabberwocky Rd</STREET_ADDRESS>
```

```
    <CITY>Southlake</CITY>
    <STATE_PROVINCE>Texas</STATE_PROVINCE>
  </Location>
</Warehouse>
<Warehouse id="2">
  <Location>
    <STREET_ADDRESS>2011 Interiors Blvd</STREET_ADDRESS>
    <CITY>South San Francisco</CITY>
    <STATE_PROVINCE>California</STATE_PROVINCE>
  </Location>
</Warehouse>
. . .

9 rows selected.

  FROM DUAL;
```

Example 5-8 Querying an XMLType Column Using XMLQuery PASSING Clause

```
CopySELECT warehouse_name,
      XMLQuery(
        'for $i in /Warehouse
         where  $i/Area > 80000
         return <Details>
                <Docks num="{$i/Docks}"/>
                <Rail>{if ($i/RailAccess = "Y") then "true" else "false"}
                </Rail>
               </Details>'
        PASSING warehouse_spec RETURNING CONTENT) big_warehouses
  FROM oe.warehouses;
```

This produces the following output:

```
CopyWAREHOUSE_NAME
--------------
BIG_WAREHOUSES
--------------
Southlake, Texas


San Francisco


New Jersey
<Details><Docks num=""></Docks><Rail>false</Rail></Details>
```

- Seattle, Washington
- <Details><Docks num="3"></Docks><Rail>true</Rail></Details>
-
- Toronto
-
-
- Sydney
-
-
- Mexico City
-
-
- Beijing
-
-
- Bombay
-
-
- 9 rows selected.

- Example 5-9 Using XMLTABLE with XML Schema-Based Data

- CopySELECT xtab.COLUMN_VALUE
- FROM purchaseorder, XMLTable('for $i in /PurchaseOrder
- where $i/CostCenter eq "A10"
- and $i/User eq "SMCCAIN"
- return <A10po pono="{$i/Reference}"/>'
- PASSING OBJECT_VALUE) xtab;
-
- COLUMN_VALUE
- ----------
- <A10po pono="SMCCAIN-20021009123336151PDT"></A10po>
- <A10po pono="SMCCAIN-20021009123336341PDT"></A10po>
- <A10po pono="SMCCAIN-20021009123337173PDT"></A10po>
- <A10po pono="SMCCAIN-20021009123335681PDT"></A10po>
- <A10po pono="SMCCAIN-20021009123335470PDT"></A10po>
- <A10po pono="SMCCAIN-20021009123336972PDT"></A10po>
- <A10po pono="SMCCAIN-20021009123336842PDT"></A10po>
- <A10po pono="SMCCAIN-20021009123336512PDT"></A10po>
- <A10po pono="SMCCAIN-20021009123333894PDT"></A10po>
- <A10po pono="SMCCAIN-20021009123337403PDT"></A10po>
-
- 10 rows selected.

- Example 5-10 Using XMLQUERY with XML Schema-Based Data

```
CopySELECT XMLQuery('for $i in /PurchaseOrder
              where $i/CostCenter eq "A10"
                and $i/User eq "SMCCAIN"
              return <A10po pono="{$i/Reference}"/>'
             PASSING OBJECT_VALUE
             RETURNING CONTENT)
  FROM purchaseorder;

XMLQUERY('FOR$IIN/PURCHASEORDERWHERE$I/COSTCENTEREQ"A10"AND$I/USER
EQ"SMCCAIN"RET
--------------------------------------------------------------------------------
<A10po pono="SMCCAIN-20021009123336151PDT"></A10po>
<A10po pono="SMCCAIN-20021009123336341PDT"></A10po>
<A10po pono="SMCCAIN-20021009123337173PDT"></A10po>
<A10po pono="SMCCAIN-20021009123335681PDT"></A10po>
<A10po pono="SMCCAIN-20021009123335470PDT"></A10po>
<A10po pono="SMCCAIN-20021009123336972PDT"></A10po>
<A10po pono="SMCCAIN-20021009123336842PDT"></A10po>
<A10po pono="SMCCAIN-20021009123336512PDT"></A10po>
<A10po pono="SMCCAIN-20021009123333894PDT"></A10po>
<A10po pono="SMCCAIN-20021009123337403PDT"></A10po>

132 rows selected.
```

Using Namespaces with XQuery

You can use the XQuery declare namespace declaration in the prolog of an XQuery expression to define a namespace prefix. You can use declare **default** namespace to establish the namespace as the default namespace for the expression.

Example 5-15 illustrates use of a namespace declaration in an XQuery expression.

An XQuery namespace declaration has no effect outside of its XQuery expression. To declare a namespace prefix for use in an XMLTable expression outside of the XQuery expression, use the XMLNAMESPACES clause. This clause also covers the XQuery expression argument to XMLTable, eliminating the need for a separate declaration in the XQuery prolog.

In Example 5-16, XMLNAMESPACES is used to define the prefix e for the namespace http://example.com. This namespace is used in the COLUMNS clause and the XQuery expression of the XMLTable expression.

Example 5-15 Using XMLQUERY with a Namespace Declaration

```
CopySELECT XMLQuery('declare namespace e = "http://example.com";

ERROR:
```

ORA-01756: quoted string not properly terminated

```
        for $i in doc("/public/empsns.xml")/e:emps/e:emp
```

SP2-0734: unknown command beginning "for $i in ..." - rest of line ignored.

...

-- This works - do not end the line with ";".

```
SELECT XMLQuery('declare namespace e = "http://example.com"; for
          $i in doc("/public/empsns.xml")/e:emps/e:emp
        let $d :=
         doc("/public/depts.xml")//dept[@deptno=$i/@deptno]/@dname
        where $i/@salary > 100000
        order by $i/@empno
        return <emp ename="{$i/@ename}" dept="{$d}"/>'
       RETURNING CONTENT) FROM DUAL;
```

```
XMLQUERY('DECLARENAMESPACEE="HTTP://EXAMPLE.COM";FOR$IINDOC("/PUBLIC/EMPS
NS.XML"
--------------------------------------------------------------------------------
<emp ename="Jack" dept=""></emp><emp ename="Jill" dept=""></emp>
```

-- This works too - add a comment after the ";".

```
SELECT XMLQuery('declare namespace e = "http://example.com";  (: :)
        for $i in doc("/public/empsns.xml")/e:emps/e:emp
        let $d := doc("/public/depts.xml")//dept[@deptno=$i/@deptno]/@dname
        where $i/@salary > 100000
```

```
            order by $i/@empno
            return <emp ename="{$i/@ename}" dept="{$d}"/>'
           RETURNING CONTENT) FROM DUAL;

XMLQUERY('DECLARENAMESPACEE="HTTP://EXAMPLE.COM";(::)FOR$IINDOC("/PUBLIC/EM
PSNS.

--------------------------------------------------------------------------------

<emp ename="Jack" dept=""></emp><emp ename="Jill" dept=""></emp>


1 row selected.


-- This works too - tell SQL*Plus to ignore the ";".


SET SQLTERMINATOR OFF


SELECT XMLQuery('declare namespace e = "http://example.com";
            for $i in doc("/public/empsns.xml")/e:emps/e:emp
            let $d :=
              doc("/public/depts.xml")//dept[@deptno=$i/@deptno]/@dname
            where $i/@salary > 100000
            order by $i/@empno
            return <emp ename="{$i/@ename}" dept="{$d}"/>'
           RETURNING CONTENT) FROM DUAL
/
```

```
XMLQUERY('DECLARENAMESPACEE="HTTP://EXAMPLE.COM";FOR$IINDOC("/PUBLIC/EMPS
NS.XML"

--------------------------------------------------------------------------------

<emp ename="Jack" dept=""></emp><emp ename="Jill" dept=""></emp>
```

Example 5-16 Using XMLTABLE with the XMLNAMESPACES Clause

```
CopySELECT * FROM XMLTable(XMLNAMESPACES ('http://example.com' AS "e"),
                  'for $i in doc("/public/empsns.xml")
                   return $i/e:emps/e:emp'
                  COLUMNS name VARCHAR2(6) PATH '@ename',
                        id   NUMBER      PATH '@empno');
```

This produces the following result:

```
Copy

NAME          ID

------ ----------

John           1

Jack           2

Jill           3


3 rows selected.
```

It is the presence of qualified names e:ename and e:empno in the COLUMNS clause that necessitates using the XMLNAMESPACES clause. Otherwise, a prolog namespace declaration (declare namespace e = "http://example.com") would suffice for the XQuery expression itself.

Because the same namespace is used throughout the XMLTable expression, a default namespace could be used: XMLNAMESPACES (DEFAULT 'http://example.com'). The qualified name $i/e:emps/e:emp could then be written without an explicit prefix: $i/emps/emp.

Querying XML Data Using SQL and PL/SQL

You can query XML data from XMLType columns and tables in various ways.

- Select XMLType data using SQL, PL/SQL, or Java.
- Query XMLType data using SQL/XML functions such as XMLQuery. See Querying XMLType Data Using XQuery.
- Perform full-text search using XQuery Full Text. See Support for XQuery Full Text and Indexes for XMLType Data.

The examples in this section illustrate different ways you can use SQL and PL/SQL to query XML data. Example 5-17 inserts two rows into table purchaseorder, then queries data in those rows using SQL/XML functions XMLCast, XMLQuery, and XMLExists.

5.XMLType APIs

You can use Oracle XML DB XMLType PL/SQL, Java, C APIs, and Oracle Data Provider for .NET (ODP.NET) to access and manipulate X Table 11-1 PL/SQL APIs Related to XML

API	Documentation	Description
XMLType	*Oracle Database PL/SQL Packages and Types Reference*, chapter "XMLType"	PL/SQL APIs with XML operations on XMLType data – validation, transformation.
Database URI types	*Oracle Database PL/SQL Packages and Types Reference*, chapter "Database URI TYPEs"	Functions used for various URI types.
DBMS_METADATA	*Oracle Database PL/SQL Packages and Types Reference*, chapter "DBMS_METADATA"	PL/SQL API for retrieving metadata from the database dictionary as XML, or retrieving creation DDL and submitting the XML to re-create the associated

API	Documentation	Description
		object.
DBMS_RESCONFIG	*Oracle Database PL/SQL Packages and Types Reference*, chapter "DBMS_RESCONFIG"	PL/SQL API to operate on a resource configuration list, and to retrieve listener information for a resource.
DBMS_XDB_ADMIN	*Oracle Database PL/SQL Packages and Types Reference*, chapter "DBMS_XDB_ADMIN"	PL/SQL API for the management of Oracle XML DB Repository by database administrators.
DBMS_XDB_CONFIG	*Oracle Database PL/SQL Packages and Types Reference*, chapter "DBMS_XDB_CONFIG"	PL/SQL API for managing Oracle XML DB configuration sessions.
DBMS_XDB_CONSTANTS	*Oracle Database PL/SQL Packages and Types Reference*, chapter "DBMS_XDB_CONSTANTS"	PL/SQL constants for use with Oracle XML DB
DBMS_XDB_REPOS	*Oracle Database PL/SQL Packages and Types Reference*, chapter "DBMS_XDB_REPOS"	PL/SQL API for the use of Oracle XML DB Repository by application developers.
DBMS_XDBRESOURCE	*Oracle Database PL/SQL Packages and Types Reference*, chapter "DBMS_XDBRESOURCE"	PL/SQL API to operate on repository resource metadata and contents.

API	Documentation	Description
DBMS_XDBT	*Oracle Database PL/SQL Packages and Types Reference*, chapter "DBMS_XDBT"	PL/SQL API for creation of text indexes on repository resources.
DBMS_XDB_VERSION	*Oracle Database PL/SQL Packages and Types Reference*, chapter "DBMS_XDB_VERSION"	PL/SQL API for version management of repository resources.
DBMS_XDBZ	*Oracle Database PL/SQL Packages and Types Reference*, chapter "DBMS_XDBZ"	Oracle XML DB Repository ACL-based security.
DBMS_XEVENT	*Oracle Database PL/SQL Packages and Types Reference*, chapter "DBMS_XEVENT"	PL/SQL API providing event-related types and supporting interface..
DBMS_XMLDOM	*Oracle Database PL/SQL Packages and Types Reference*, chapter "DBMS_XMLDOM"	PL/SQL implementation of the DOM API for XMLType.
DBMS_XMLGEN	*Oracle Database PL/SQL Packages and Types Reference*, chapter "DBMS_XMLGEN"	PL/SQL API for transformation of SQL query results into canonical XML format.
DBMS_XMLINDEX	*Oracle Database PL/SQL Packages and Types Reference*, chapter "DBMS_XMLINDEX	PL/SQL API for XMLIndex.
DBMS_XMLPARSER	*Oracle Database PL/SQL Packages and Types Reference*, chapter "DBMS_XMLPARSER"	PL/SQL implementation of the DOM Parser API for XMLType.

API	Documentation	Description
DBMS_XMLSCHEMA	*Oracle Database PL/SQL Packages and Types Reference*, chapter "DBMS_XMLSCHEMA	PL/SQL API for managing XML schemas within Oracle Database – schema registration, deletion.
DBMS_XMLSCHEMA_ANNOTATE	*Oracle Database PL/SQL Packages and Types Reference*, chapter "DBMS_XMLSCHEMA_ANNOTATE"	PL/SQL API for adding and managing Oracle-specific XML Schema annotations.
DBMS_XMLSTORAGE_MANAGE	*Oracle Database PL/SQL Packages and Types Reference*, chapter "DBMS_XMLSTORAGE_MANAGE"	PL/.SQL API managing and modifying storage of XML data after XML schema registration.
DBMS_XMLSTORE	*Oracle Database PL/SQL Packages and Types Reference*, chapter "DBMS_XMLSTORE"	PL/SQL API for storing XML data in relational tables.
DBMS_XSLPROCESSOR	*Oracle Database PL/SQL Packages and Types Reference*, chapter "DBMS_XSLPROCESSOR"	PL/SQL implementation of an XSLT processor.

- PL/SQL APIs for XMLType
  There are several PL/SQL packages that provide APIs for XMLType.

- PL/SQL Package DBMS_XMLSTORE
  You can use PL/SQL package DBMS_XMLSTORE to insert, update, or delete data from XML documents stored object-relationally. It uses a canonical XML mapping similar to the one produced by package DBMS_XMLGEN. It converts the mapping to object-relational constructs and then inserts, updates or deletes the corresponding values in relational tables.

- Java DOM API for XMLType
  The Java DOM API for XMLType lets you operate on XMLType instances using a DOM. You can use it to manipulate XML data in Java, including fetching it through Java Database Connectivity (JDBC).

- C DOM API for XMLType
  The C DOM API for XMLType lets you operate on XMLType instances using a DOM in C.

- Oracle XML DB and Oracle Data Provider for .NET
  Oracle Data Provider for Microsoft .NET (ODP.NET) is an implementation of a data provider for Oracle Database. It uses Oracle native APIs to offer fast and reliable access to Oracle data and features from any .NET application.

PL/SQL DOM API for XMLType – Examples

Examples are presented of using the PL/SQL DOM API for XMLType.

Remember to call procedure freeDocument for *each* DOMDocument instance, when you are through with the instance. This procedure frees the document and all of its nodes. You can still access XMLType instances on which DOMDocument instances were built, even after the DOMDocument instances have been freed.

Example 11-1 creates a hierarchical, representation of an XML document in dynamic memory: a DOM document.

Example 11-1 uses a *handle* to the DOM document to manipulate it: print it, change part of it, and print it again after the change. Manipulating the DOM document by its handle also indirectly affects the XML data represented by the document, so that querying that data after the change shows the changed result.

The DOM document is created from an XMLType variable using PL/SQL function newDOMDocument. The handle to this document is created using function makeNode. The document is written to a VARCHAR2 buffer using function writeToBuffer, and the buffer is printed using DBMS_OUTPUT.put_line.

After manipulating the document using various DBMS_XMLDOM procedures, the (changed) data in the XMLType variable is inserted into a table and queried, showing the change. It is only when the data is inserted into a database table that it becomes persistent. Until then, it exists in memory only. This persistence is demonstrated by the fact that the database query is made after the document (DOMDocument instance) has been freed from dynamic memory.

Example 11-2 creates an empty DOM document, and then adds an element node (<ELEM>) to the document. DBMS_XMLDOM API node procedures are used to obtain the name (<ELEM>), value (NULL), and type (1 = element node) of the element node.

Example 11-1 Creating and Manipulating a DOM Document

```
CopyCREATE TABLE person OF XMLType;


DECLARE
 var      XMLType;
 doc      DBMS_XMLDOM.DOMDocument;
 ndoc     DBMS_XMLDOM.DOMNode;
 docelem   DBMS_XMLDOM.DOMElement;
 node     DBMS_XMLDOM.DOMNode;
 childnode DBMS_XMLDOM.DOMNode;
 nodelist  DBMS_XMLDOM.DOMNodelist;
 buf      VARCHAR2(2000);
BEGIN
 var := XMLType('<PERSON><NAME>ramesh</NAME></PERSON>');

 -- Create DOMDocument handle
 doc    := DBMS_XMLDOM.newDOMDocument(var);
 ndoc   := DBMS_XMLDOM.makeNode(doc);

 DBMS_XMLDOM.writeToBuffer(ndoc, buf);
 DBMS_OUTPUT.put_line('Before:'||buf);

 docelem := DBMS_XMLDOM.getDocumentElement(doc);

 -- Access element
 nodelist := DBMS_XMLDOM.getElementsByTagName(docelem, 'NAME');
 node := DBMS_XMLDOM.item(nodelist, 0);
```

```
 childnode := DBMS_XMLDOM.getFirstChild(node);


 -- Manipulate element
 DBMS_XMLDOM.setNodeValue(childnode, 'raj');
 DBMS_XMLDOM.writeToBuffer(ndoc, buf);
 DBMS_OUTPUT.put_line('After:'||buf);
 DBMS_XMLDOM.freeDocument(doc);
 INSERT INTO person VALUES (var);
END;
/
```

This produces the following output:

```
Copy
Before:<PERSON>
 <NAME>ramesh</NAME>
</PERSON>


After:<PERSON>
 <NAME>raj</NAME>
</PERSON>
```

This query confirms that the data has changed:

```
Copy
SELECT * FROM person;
SYS_NC_ROWINFO$
```

```
---------------
<PERSON>
 <NAME>raj</NAME>
</PERSON>


1 row selected.
```

Example 11-2 Creating an Element Node and Obtaining Information About It

```
CopyDECLARE
 doc   DBMS_XMLDOM.DOMDocument;
 elem  DBMS_XMLDOM.DOMElement;
 nelem DBMS_XMLDOM.DOMNode;
BEGIN
 doc := DBMS_XMLDOM.newDOMDocument;
 elem := DBMS_XMLDOM.createElement(doc, 'ELEM');
 nelem := DBMS_XMLDOM.makeNode(elem);
 DBMS_OUTPUT.put_line('Node name = ' || DBMS_XMLDOM.getNodeName(nelem));
 DBMS_OUTPUT.put_line('Node value = '|| DBMS_XMLDOM.getNodeValue(nelem));
 DBMS_OUTPUT.put_line('Node type = ' || DBMS_XMLDOM.getNodeType(nelem));
 DBMS_XMLDOM.freeDocument(doc);
END;
/
```

This produces the following output:

```
CopyNode name = ELEM
```

Node value =

Node type = 1