Relational Extensions

Object-Relational and XML Extensions

Oscar Romero
Roberto García
Rosa M. Gil Iranzo

PID_00179808

UOC
Universitat Oberta de Catalunya

Index

Introduction

The relational model has shown its versatility and potential all over these years. Indeed, it dates back to the 70s and, since then, many real scenarios have been captured and modeled by means of this model.

The relational model

The relational model was introduced by E. F. Codd in 1970:

Codd, E. F. (June 1970). "A relational model of data for large shared data banks". *Communications ACM* (13(6), pp. 377–387). ACM.

For many years, any scenario demanding data persistence was properly mapped and implemented according to the relational principles. Mainly:

- Data must be organized in tables and rows.
- Relationships between tables must be expressed with the referential integrity constraint (foreign key – candidate key relationships).

The simplicity of the relational model is complemented by a strong foundation on the set theory, which provides algebraic and calculus-based techniques for querying relational data (the core behind the SQL language).

For years, modeling business (or operational) data in tabular form (i.e., by means of rows and columns) was proven to be natural and rather easy. For example, we all know that users' data can be naturally represented in a table, representing each row a user and each column one of his/her attributes. We can proceed similarly to capture most operational data.

Soon, the relational model succeeded as de facto standard to implement data persistence and many people experimented implementing this model in many other areas, such as engineering design, geographic information or, in general, to store complex data.

At this point, some drawbacks regarding the tabular format behind the relational model arose. Indeed, it was crystal clear that the relational model did not suit some other areas that nicely, and some reengineering and design techniques had to be applied. For example, consider the two-dimensional arrays used in many areas, like in remote sensing used in ocean and atmospheric simulation. Mapping arrays to tabular data is not easy, and database administrators (DBAs) had to come up with ad hoc tabular array representations that could be later easily stored in databases.

It didn't take DBAs much to realize that tabular representation was affecting performance, because retrieving large amounts of data from these ad hoc tabular representations required many joins. Around that time, object-oriented programming languages (OOPLs) started to appear defining the concept of user-defined classes, with classes' attributes, methods and encapsulation, which did really suit the DBAs necessities.

Two main trends dominated by this point: those claiming that the relational model was not enough to model any kind of data / environment (e.g., object-oriented applications) and new systems had to be developed, and those saying that the relational model simply needed some extensions to nicely fit them in. Nowadays, we can say that the second trend succeeded (mainly because of the support of major relational software vendors) and we can talk about relational extensions to adapt other paradigms to relational. Relational extensions mainly focus on incorporating complex objects into the database. These objects are stored in relational tables and thus, by definition, they violate the first normal form (which states that any database field should not be decomposable).

In this module we focus on two of these extensions: the object-oriented extension and the XML extension. The first one appeared in the 90s, whereas the second one was introduced a bit later, at the beginning of the new century. These two extensions are, nowadays, the most popular extensions and also the two first widely accepted by vendors and the database community.

Although relational extensions succeeded versus native applications that dealt with OO or XML application, there were some native proposals that co-existed during all this time with relational approaches. While it is true that, in the past, these alternatives were a minority, nowadays, this trend is increasing thanks to the NOSQL wave.

Objectives

The main objective of this module is to get familiar with the two most popular relational extensions; namely, object-oriented and XML relational extensions. Specifically:

1. Explain the historical background / needs behind object data models (ODMs) for databases.

2. Enumerate the main features an object data model must provide; similarly, name the main standard object-oriented features from SQL:1999.

3. Discuss about advantages and disavantages of using a purely relational or a purely object-oriented approach. For example, elaborate on the dichotomy attributes vs. methods, OIDs vs. PKs, REFs vs. FKs, etc.

4. Discuss the object-oriented layer implementation in Oracle, with regard to the underlying relational technology.

5. Formulate simple (i.e., basic syntax), correct SQL statements (in Oracle syntax) for the following standard object-oriented features: Row type, UDTs, inheritance, object tables, REFs and collections.

6. Justify the suitability to store objects in columns, object tables or use object views, for Oracle.

7. For a given specification, justify the suitability (at least, four reasons) of using VARRAYs instead of NESTED TABLEs for modeling multi-valued attributes (i.e., collections) in Oracle.

8. Understand the fundamentals of XML, its syntax, its underlying structure and the namespaces mechanism that avoids naming clashes.

9. Know the way to add structure to XML documents using XML Schemas. They make possible to model a domain using XML Schemas and use them to constraint the way XML documents are created to capture data for that domain.

10. Create XQuery requests to retrieve data from XML documents and also to generate output XML documents meeting the requirements for those queries.

11. Comprehend the different clauses, functions and operators that constitute a XQuery, especially XPath to build paths across XML documents to select the relevant parts from them to the XQuery at hand.

12. Be aware of how XML data is stored in a particular database, Oracle, and how XML Schema and XQuery can be used in the context of that particular database.

1. The Object-Relational Extension

1.1. Background

Shortly after the introduction and wide acceptance of the relational model, back to the 70s, databases (and the relational model) began to be used in many other areas besides operational and business scenarios, which could not always be easily adapted to the relational model.

When object-oriented programming languages (OOPLs) were a reality, many people wondered why such approach could not be extended to databases and develop an object-data model that would replace the current (and by then already successful) relational model. During the mid-80s, the first object-oriented database systems (OODBSs) were developed.

Manifesto

A manifesto is a public declaration of intentions, opinions, objectives, or motives, as one issued by a government, sovereign, or, as in the case of the OODBS manifesto, by an organization.

This school of thought reached its climax when the *Object-Oriented Database System Manifesto* was published. This manifesto, signed by some of the most reputed researchers of that time, claimed that OODBSs better supported the emerging programming languages and gave support for complex structures or objects, which were the basis of non-operational (i.e., based on complex data) applications.

1.1.1. Object-Oriented Database Systems

Specifically, they claimed that OODBSs differ from relational databases in the use of internal object identifiers or OIDs (instead of candidate keys) and provided a natural integration between OOPLs and stored data (i.e., the database). At the same time, it was claimed that relational databases could not fill the gap with the object-oriented (OO) paradigm because of the mismatch between set-oriented data storage promoted by the relational model and the iterative one-record-at-a-time language access (this is known as the *impedance mismatch* between SQL and procedural, high-level languages). In OODBSs, the OOPL provides a uniform yet natural way to access object-oriented data (both at the application and database level), whereas relational databases had to develop specific object-oriented interfaces to communicate with SQL, a declarative high-level language, which was the only way to access the database.

Bibliography

The Object-Oriented Database System Manifesto:

M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik (December 1989). "The Object-Oriented Database System Manifesto". In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases* (pages 223-240). Kyoto, Japan.

The manifesto organized the object-oriented features that any OODBMS should contain in four categories:

- The Golden Rules, or core features, which were divided in two categories: those coming from the object-oriented data model and those from the DBMS field,

- Optional features, which advocated for advisable characteristics and

- Open issues, which embraced open problems not solved back then.

Specifically, each category can be described as follows (summarized in table 1):

- Golden Rules: These rules define an OODBS. A system that integrates the traditional database management with the object-oriented data model.
  - OO Data Model: Data must be internally organized according to the object-oriented data model; i.e., objects as means to represent data and universally identified by an object identifier (OID). Encapsulation, in terms of functions, is mandatory as well as traditional object-oriented features such as inheritance and extensibility, overriding / overloading operators and late binding. All in all, providing computational completeness (i.e., any algorithm can be implemented following this paradigm).
  - DBMS: A database management system should provide data persistence and therefore, it must deal with issues such as secondary storage management, concurrency, recovery and ad hoc query facilities.

Importantly, the same high-level language used to develop applications must be used to access the database. Thus, the golden rules advocate for access uniformity (i.e., OOPLs as single language to access the database and implement database applications).

- Optional features: These features are considered to be appealing, but the manifesto acknowledges the difficulty to provide them so they are categorized as optional. It includes multiple inheritance, type checking and type inferring, data distribution (in the sense of distributed database management systems), transactions and versioning (any modification and not only the last version must be stored).

- Finally, open features refer to those features still representing a challenge for the community, mainly because of the lack of standards. For example, several object-oriented languages co-existed or were developed back then. The manifesto claimed that it was not possible to deploy a different OODBS for each existing high-level language. Thus, some agreement should be reached to provide uniformity. In the manifesto, they identify three main issues on which to agree; a programming paradigm, an internal represen-

tation system and a type system. These open issues were never solved, however, as discussed in section 1.1.4.

Table 1. The Object-Oriented Database System Manifesto in a nutshell

The Golden Rules		**Optional**	**Open**
OO Data Model	**DBMS**		
Complex Objects	Persistence	Multiple Inheritance	Programming Paradigm
Object Identity	Secondary Storage Management	Type Checking and Inferring	Representation System
Encapsulation	Concurrency	Distribution	Type System
Types / Classes	Recovery	Design Transactions	Uniformity
Inheritance	Ad hoc Query Facility	Versions	
Overriding, Overloading, Late Binding			
Extensibility			
Computational Completeness			

Importantly, you should not think of the relational databases of the 80s as you currently know them. Relational technology was rather primitive back then and, for example, ODBC / JDBC interfaces were not available. Instead, special APIs had to be developed by programmers to bridge the OO view of data in their applications and the relational view of data stored in the database.

As access language, relational databases already counted with SQL (a declarative language), which was in a very primitive stage (for example, it could not deal with primary or foreign keys until SQL-89) and for years, it was source of many criticisms towards the relational technology. Mainly, because of two reasons: it was not computationally complete and because of the impedance mismatch with procedural high-level languages.

SQL

SQL was developed at IBM Research Laboratories in the 70s, based on the relational data model defined by E. F. Codd in 1970. It is a *de iure* international standard named Database Language SQL.

1.1.2. Object-Relational Database Systems

As previously discussed, the OO school of thought was not alone in their crusade. In parallel, relational pundits claimed that, although the OO paradigm provided a revolutionary concept for data modeling, all the efforts put in the relational technology should not be put aside. Instead, relational databases should be extended with object features to deal with OO principles but always keeping the essence of their predecessors: **data independence as a must**.

Bibliography

The Third Generation Database System Manifesto:
Stonebraker, M.; Rowe, L.A.; Lindsay, B.G.; Gray, J.; Carey, M.J.; Brodie, M.L.; Bernstein, P.A.; Beech, D. (September 1990). "Third-generation database system manifesto". *SIGMOD Record* (19(3), pages 31-44). ACM.

Soon, they counter-attacked with the Object-Relational Database Systems (ORDBs) manifesto, also known as the *Third Generation Database System Manifesto*. According to this manifesto, ORDBSs should provide the following features: user-defined data types, object tables formed from user-defined types, hierarchies and support for OIDs (also as a mean to relate objects).

Specifically, the manifesto is structured in 3 different tenets (see table 2):

- Traditional DBMS services and support for richer object structures and rules (i.e., integrity constraints),
- Must subsume second generation DBMSs and
- Must be open to other subsystems.

Table 2. The Third Generation Database System Manifesto in a nutshell

Object and Rule Management	DBMS Function	Towards an Open System
Rich Type System	Non-procedural, high-level access language: SQL	Accessible from multiple high-level languages
Multiple Inheritance	Suppport collections: enumeration of members or using the query language	Enhancement of DBMS – programming language interfaces
Encapsulation		Queries and answers as the lowest level of communication (client / server)
OIDs and PKs	Updatable views	
Rules Enforcement	Data independence	

The manifesto elaborates on each of the three tenets as follows:

- Object and rule management: These features extend the relational model to deal with objects and rules (i.e., more expressive integrity constraints). Basically, they combine relational and object-oriented features. For example, they mainly talk about rich types, although objects are also supported and consequently, encapsulation. Note, however, that multiple inheritance is mandatory and, interestingly, object identifiers (OIDs) are allowed as an alternative to primary keys. Rule management was dealt as a first-class citizen in order to deal with richer semantics. Accordingly, additional means (to those already provided by the relational model) to enforce constraints are identified as mandatory (this point was the seed for trig-

gers and procedures, which were not available for relational databases at that time).

- DBMS Function: This set of features extends those principles upon which relational database systems were built. From the database point of view, a non-procedural high-level language should be available to access the database (i.e., SQL) instead of a procedural one (as claimed in the OODBSs manifesto). For better or worse, they say, a non-procedural high-level access language is needed, and cannot be replaced by object-oriented programming languages. Their ultimate objective behind this statement was preserving data independence. However, back then, SQL was heavily criticized for not being computationally complete. In this sense, the ORDBS manifesto proposed additional features to complement SQL; namely: updatable views (in their own words, *dynamic* views) and collections (to deal with procedural arrays).

- Towards an open system: By an open system they meant that any application (i.e., high-level languages) should be able to access the database, but always through programming interfaces (this was the seed for what nowadays is known as high-level language connectors; e.g., ODBC or JDBC). SQL remains as the only access language and, again, they recall us that high-level languages should not directly access the physical structures of the database (to preserve data independence). Instead, they should *query* the database and receive an *answer* by means of SQL.

1.1.3. Object-Oriented Data Model vs. Object-Relational Data Model

What we have presented in previous sections are the essentials behind the object-oriented data model (OODM) and the alternative object-relational data model (ORDM). In this section, we discuss the main differences and agreements between both models, which can be summarized with the following claims (A √ stands for agreement, X for disagreement and ~ for a partial agreement):

- ~ **Complex Objects, Type Classes, Extensibility etc. vs. Rich Type System**: Concepts from the ORDM parallel those in OODBs, with the notions of user-defined data types, object tables formed from user-defined types, hierarchies of user-defined types and object tables, rows of object tables with internal object identifiers, and relationships between object tables that use object identifiers as references. However, in a object-relational database all these concepts are finally mapped to the relational structure. Thus, they are introduced as a relational extension.

- ~ **Inheritance vs. Multiple Inheritance**: Inheritance is essential for the OODM, but the ORDM goes even further by asking for multiple inher-

itance, whereas supporting multiple inheritance is not mandatory for OODBS (see section 1.1.1).

- √ **Encapsulation**: Both models fully agree on the need for encapsulation, which refers to the fact that class methods must also be stored in the database, altogether with the object itself.

- X **Object Identity (OID) vs. OIDs and PKs**: Whereas the OODM claims for the necessity of OIDs and refer to them by means of object pointers, the ORDM claims for the co-existence of OIDs and object references with primary keys and foreign keys. It must be up to the database designer to decide which mechanism better suits their necessities.

- √ **DBMS main features (persistence, concurrency, recovery etc.)**: There is no argument about what a database management system means, and both models advocate for the same principles.

- ~ **HLLs Computational Complexity vs. SQL, Constraints enforcement, updatable views and HLLs Interfaces**: High-level languages (HLLs in short) are computationally complete (i.e., you can implement any algorithm), whereas SQL by itself was not. The ORDM proposes to complement SQL with constraints enforcement (basically, triggers and procedures), updatable views (to support dynamic views over data) and also HLLs interfaces to access data (but always through a query-answer manner). Furthermore, people behind the third generation database system manifesto proposed this solution because a non-procedural high-level language untied to physical structures (such as SQL) was mandatory to preserve data independence.

Note

SQL was widely criticized by the OO community because of the impedance mismatch between SQL and HLLs. Exchanging data between a procedural and a declarative language was seen as too complex, and they claimed it affected global optimization. This problem is related to the fact that SQL is not computationally complete.

- √ Open Systems: By this claim, both models argue that databases must be able to communicate with other systems or applications.

- ~ OODBSs vs. ORDBSs: All in all, the OODM is supported by OODBSs, whereas the ORDM is supported by ORDBSs, being incompatible with each other.

Summing up, both models agree on the necessity to provide rich types (i.e., give support to define object types), encapsulation (in the form of object methods to also be stored in the database), inheritance between objects, reusability (of objects and methods; i.e., code) and also the necessity to be open to other systems (i.e., applications should easily access data in the database). However, they are confronted because of their approach to model data. In other words, it is a matter of *how* to implement it; OODBSs or ORDBSs.

Specifically, the ORDM is an extension of the relational model and claims to reuse all efforts previously devoted to databases. Contrarily, the OODM claims for a complete rewrite, because neither SQL nor the relational model properly

support the OO paradigm. Consequently, they claim for native OO databases (i.e., designed from scratch). This main disagreement has many consequences. For example, the OODM promotes a uniform way to access data and develop applications and, therefore, the user should be able to navigate physical structures and data in the database through a procedural, high-level OO language. The ORDM model, however, claims for data independence and therefore, users and applications (through HLL interfaces) must exclusively access data (in the database) by means of a non-procedural high-level language (namely, SQL).

1.1.4. Object-Oriented Database Systems Today

During the 80s, many OODBSs were developed. The pioneer was a research project called ORION, back to the early 80s, which gave rise to Versant, a well-known OODBS still around as Versant Object Database.

Unfortunately, most of those DBMSs have been discontinued. With the perspective of time, we could identify the lack of a common data model as the main reason for their decline. Although the OODM was shaped in the Object Oriented Database System Manifesto, there was no consensus on how to develop it mainly because:

- They lacked a strong theoretical framework (like the set theory behind the relational model) behind the OODM,
- And because OODBSs bloomed with an incredibly strong experimental activity, with many systems and proposals being around but with *no de facto standard* (instead, relational systems were supported by major relational vendors).

Nowadays, this movement is somehow gaining relevance with the arrival of alternative non-relational solutions embraced under the umbrella of what is known as NOSQL.

The NOSQL wave

The NOSQL wave is somehow repeating the same steps as the OODBSs back in the 80s, and it claims that the relational model is not the solution to every possible scenario (expressed with the *one world does not fit all* motto). However, the approach is subtly different, as NOSQL arises as an alternative to relational databases, whereas OODBSs claimed for replacing relational databases.

The OODBSs movement is still alive as an open source movement whose main reference is the http://www.odbms.org/Introduction website. There, you will find the Db4o project, which is an open source object database project. As a curiosity, you are also allowed to download and test some object databases such as Objectivity/DB, ObjectStore or Versant's ODBMS.

1.1.5. Object-Relational Database Systems Today

One of the people behind the Third Generation Database System Manifesto was Michael Stonebraker, who developed Postgres, the first object-relational database system in 1987.

However, object-relational features were not standarized (i.e., included in the SQL standard) until SQL-99. Object-relational features included in SQL-99 are far away from the ambitious propositions claimed in the manifesto. Indeed, some of them, such as updatable views, are still open problems not solved and thus, a chimera for current relational databases. The following features were added in SQL-99 to support the ORDM:

- LOB (Binary Large Object) type: the LOB type was conceived as a "relational" feature to support object-oriented databases. Internally, LOB serializes objects as byte streams.

- Row type (ROW): The row type allows a column to contain several attributes and, thus, violates the first normal form. These attributes are accessed by means of the dot notation (e.g., `person.age`). Relevantly, ROWs cannot be references not reused externally, which makes them useless.

- User defined types (UDTs): UDTs were introduced in two different ways, as distinct types and structured types. Importantly, both are types, not objects. Distinct types attach semantics to already built-in datatypes (e.g., `CREATE DISTINCT TYPE age AS Integer`) but they do neither provide inheritance nor methods. Structured types (most commonly known as user defined types in databases) were thought to support objects, as discussed later in section 1.2.

- Typed tables: Tables where each row is an object (i.e., an instance of a UDT).

- Inheritance: Inheritance is allowed at UDT level. However, multiple inheritance (as claimed in the manifesto) is not supported.

- REF Type and object pointers, as an alternative to primary and foreign keys.

- Collection type (ARRAY): This collection type implements an array that could contain multiple values.

SQL-99 introduced many other interesting features, although not exclusively object-oriented but answering some of the problems stated in the manifesto. For example, triggers are introduced, as well as the DISTINCT keyword. Recursive queries were also introduced at this revision (by means of the WITH RECURSIVE keywords), which improved SQL expressiveness.

1.2. The Object-Relational Model

A model is composed of a data structure, a set of integrity constraints and a set of operators to handle data. As explained, the object-relational model is an extension of the relational model and, thus, it relies on it.

Importantly, and according to the discussions undertaken in the 80s, the operators used to manipulate the object-relational data are those already available in the relational model, that is the relational algebra and, at a higher level of abstraction, SQL. In this sense, thus, there is no new contribution.

Regarding additional (object-oriented) extensions on the relational data structure and integrity constraints, the SQL-99 introduced a set of elements (see previous section). Unfortunately, there is no RDBMS in the market that implements the standard. Every system has decided to model the ORDM in its own way and none of them is even close to what the standards states. For this reason, we will overlook the SQL-99 standard and focus on a specific relational system, namely Oracle, and elaborate on the object-relational features introduced to extend the relational data structure.

In Oracle, the OODM mainly consists of user-defined types (UDTs), REFs and means to implement collections. Oracle UDTs are different from the standard in that they resemble C++ or Java classes. Thus, they are not types, as proposed in SQL-99. REFs are rather similar to the standard concept and collections are supported by means of VARRAYs (similar to the ARRAY concept in the standard) and nested tables (a new concept with no correspondence in the standard). LOBs are currently implemented in Oracle as two different types: BLOBs and CLOBs, which replaced the deprecated LONG RAW type. They are designed to store large objects of any kind: XML files, multimedia files etc. All of them are up to 4 gigabytes long, and they differ from each other on the character set used.

This section is structured as follows. Each new feature is introduced individually: user-defined types, REFs and collections, together with some examples in Oracle syntax. For each category, we also identify some practical issues.

1.2.1. User Defined Types (UDTs)

In Oracle, structured types (also known as user defined types) are a layer of abstraction built on top of relational technology. Consequently, the object-relational data structures and integrity constraints are eventually translated in terms of tables and columns. As a general rule, objects are translated as relational tables (each object attribute as a column), but with the following overheads:

- An extra column to store the mandatory OID and
- Extra space to store NULL pointers (see section 1.2.2.).

OIDs are, by definition, 16 bytes long.

Oracle calls them *object types* to distinguish them from traditional basic types and also from UDTs as described in the SQL-99 (there, they were described as types, not objects) and therefore, they can have instances (i.e., objects).

Object types can be created from any built-in database type and / or any previously created type, object-references (i.e., REFs, see section 1.2.2.) and / or collection types (either Nested Tables or VARRAYs, see section 1.2.3.). As presumed, they can contain methods (member, static or constructor) that are stored with the object.

Oracle keeps track of metadata related to these types in the catalog, which can be accessed through SQL, PL/SQL or HLLs interfaces.

Note

You can drop types by using a DROP statement.

UDTs syntax in Oracle is as follows[1]:

```
CREATE TYPE name AS OBJECT (
   List of attributes,
   List of procedure specifications,
   List of function specifications
);
```

[(1)]This module presents a simplified syntax for UDTs and we will proceed similarly for the rest of concepts introduced in this module. We recommend readers to refer to Oracle manuals for a full description of the object-oriented features syntax.

We introduce the syntax with an example:

```
CREATE TYPE fullname AS OBJECT (
   given VARCHAR2(10),
   surname VARCHAR2(30),
   initials VARCHAR(5),
   MEMBER PROCEDURE generate_full_name
    (SELF IN OUT NOCOPY fullname),
   MAP MEMBER FUNCTION get_id RETURN VARCHAR2
);
```

The `CREATE TYPE ... AS OBJECT` syntax tells Oracle it is an object type. Thus, besides regular attributes it contains two method headers in the form of a function (i.e., `get_id`) and a procedure (i.e., `generate_full_name`). In this module we do not intend to revisit the object-oriented paradigm and thus, we will not get into details when talking about object methods and class properties, because the mechanism is similar to the class mechanism found in most object-oriented programming languages, such as C++ or Java.

Note

The examples in this section can be found in file 1.2.2. UDTs-code.

Nevertheless, find below an example of how to define methods in Oracle. It uses the `CREATE TYPE BODY` statement, which must be used to define the member methods (i.e., procedures and functions) specified in the object definition. The syntax is as follows:

```
CREATE TYPE BODY name AS
   List of subprogram declarations
END;
```

For example:

```
CREATE TYPE BODY fullname AS
   MAP MEMBER FUNCTION get_id RETURN VARCHAR2 IS
       BEGIN
        RETURN given || ' ' || surname;
       END;
   MEMBER PROCEDURE generate_fullname (SELF IN OUT
       NOCOPY fullname) IS
       BEGIN
         DBMS_OUTPUT.PUT_LINE(given || ' ' ||
         initials || ' ' || surname);
       END;
END;
```

In this example, the `CREATE TYPE BODY ... AS` is used to define those type methods (i.e., functions and procedures) declared in the fullname object type definition.

DBMS_OUTPUT

The `DBMS_OUTPUT` procedure is provided by Oracle and it is useful to print information. Here, the `put_line` function is used to print every field of the input fullname object.

Now, we can use this type as any other type. For example:

```
CREATE TYPE person AS OBJECT (
   name VARCHAR2(20),
   realName fullname);

CREATE TYPE release AS OBJECT (
   artist person,
   title VARCHAR2(30)
);
```

Note

By now, do not worry about when an attribute should be defined as a basic or an object type. We will tackle this issue later in this module.

In this example, the `fullname` type is nested in the `person` type that, in turn, is nested in the `release` type.

Now, we can create a table containing such type as an attribute:

```
CREATE TABLE my_songs (
  song release,
  rating NUMBER
);
```

This creates a relational table with an object type as a datatype of one of its columns (this is what is known as *column objects*).

To insert data in such table we should proceed as follows:

```
INSERT INTO my_songs VALUES (
release (person ('Adele', fullname ('Adele','Atkins','L.B.') ),'Chasing Pavements'), 10);
```

Note that it is mandatory to use the object type constructor in order to properly insert data.

Now, we can retrieve the id of the artist and the rating of those songs rated better than 8, by issuing the following SQL query (which calls the `get_id() function` procedure previously defined):

```
SELECT s.song.artist.realname.get_id(), s.rating
FROM my_songs s
WHERE s.rating > 8
ORDER BY s.rating;
```

Furthermore, UDTs can benefit from inheritance and type evolution as shown in next sections.

Type inheritance

The `UNDER` clause can be used to define subtypes. The syntax is as follows:

Note

The examples in this section can be found in file 1.2. Inheritance_alter_type_code.

```
CREATE TYPE name UNDER parentType (
  List of attributes,
  List of procedure specifications,
  List of function specifications
) [NOT FINAL, NOT INSTANTIABLE];
```

By definition, subtypes inherit the features of the parent type and extend its definition with new attributes and / or methods (possibly, redefining methods inherited). Furthermore, we can define it as `FINAL/NOT FINAL` (so it cannot be extended with any subclass) or `INSTANTIABLE/NOT INSTANTIABLE` (i.e., whether it is an abstract type or not). For example, suppose we want to extend the `person` object type previously defined as follows:

```
CREATE TYPE artist UNDER person (
   country VARCHAR2(10),
   genre VARCHAR(20)
);
```

`Artist` automatically inherits all methods and attributes from `person`, and extends it with its own attributes: `country` and `genre`.

Inheritance is an interesting feature that provides more semantics than object references or foreign keys.

See also

For further details, see section "Practical issues on REFs".

Type evolution

Object types can be modified with `ALTER TYPE` statement which can be used to:

- Add / drop attributes or methods from an object type,
- Modify a numeric attribute (length, precision or scale) or a varying length character attribute length or
- Change a type's FINAL and INSTANTIABLE properties.

For example:

```
ALTER TYPE artist ADD ATTRIBUTE (
   subgenre VARCHAR(20)) CASCADE;
```

With the cascade keyword, we automatically extend all artist objects previously defined.

Object tables

In an object table each row represents an object. You could be tempted to think on this kind of tables as a single-column table, but, as previously discussed in section 1.2.1., they are internally implemented as a multi-column where each object attribute plus the object OID are stored in a different column.

Note

The examples in this section can be found in file 1.2. objects_tables_code.

Object tables accept indexes, constraints and triggers (except for indexes and constraints over unscoped REFs; see section 1.2.2.).

You can define an object table by using the following syntax:

```
CREATE TABLE name OF type;
```

Where `name` is the table name and `type` is an already existing type. For example:

```
CREATE TYPE single UNDER release (
   album VARCHAR2(20),
   releaseDate DATE,
   chartPosition NUMBER,
   MEMBER PROCEDURE display_details
);
```

```
CREATE TABLE singles OF single;
INSERT INTO singles VALUES
(
   single (artist ('Adele', fullname ('Adele','Atkins',
   'L.B.')'UK','soul','none'),'Chasing Pavements', '19', '11-01-2008', 2)
);
```

We can now query this table by means of SQL or PL/SQL. For example:

```
//SQL
SELECT VALUE(s) FROM singles s WHERE s.artist.name = 'Adele';
// PL/SQL block
DECLARE song single;
   BEGIN
     SELECT VALUE(s) INTO song
     FROM singles s
     WHERE s.person.name = 'Adele';

     s.display_details;
   END;
```

SQL statement

In a SQL statement, the VALUE function takes as its argument a correlation variable (i.e., a table alias) for an object table or object view and returns object instances corresponding to rows of the table or view. The VALUE function may return instances of the declared type of the row or any of its subtypes.

Object views

Object views are used to access relational data using object-related features without modifying it. In this way, we can access objects that belong to an object view in the same way as if they were objects in an object table.

Polymorphism

Object views allow exploiting polymorphism through type hierarchies. Thus, a polymorphic expression can take the declared type or any of its subtypes as value.

In this way, you can benefit from object-relational features without modifying pre-existing relational data.

For example, suppose we had the following relational table:

```
CREATE TABLE songs (
   artist VARCHAR2(30),
   title VARCHAR2(30),
   album VARCHAR2(30),
   length NUMBER(3,2),
   genre VARCHAR(10),
   subgenre VARCHAR(10)
);
```

Note

The examples in this section can be found in file 1.2. object_view_code.

First, we must create an object type mapping the relational data we want to manipulate mirroring the object-oriented features. For example:

```
CREATE TYPE adele_song AS OBJECT (
   title VARCHAR2(30),
   album VARCHAR2(30),
   length NUMBER(3,2)
);
```

We can now create an object-view mapping the desired relational data. The general syntax is as follows:

```
CREATE VIEW name OF type AS
   SELECT statement;
```

Where `name` is the view name, `type` is an already existing type and the `SELECT` statement maps the relational data onto the type specified. Following our example:

```
CREATE VIEW adele_songs OF adele_song WITH OBJECT IDENTIFIER (title, album) AS
   SELECT s.title, s.album, s.length
   FROM songs s
   WHERE s.artist = 'Adele';
```

The `WITH OBJECT IDENTIFIER` keywords tell us the object id. Optionally, we can materialize this object-view by using the `CREATE MATERIALIZED VIEW` statement. Now, we can query this view in an object-relational fashion hiding the relational nature of these data.

Object types can be used in three different ways: as column objects (i.e., as an object attribute in a relational table), in object views (wrapping relational data in an object-oriented fashion) or in object tables (where the whole row is an object itself).

Practical Issues on UDTs

In this section we intend to provide some guidelines to decide whether a full or partial object-relational approach better suits our necessities.

As introduced in section 2.2.1., objects are eventually translated to tables. One could argue, though, that performance could be affected by this semantic layer built on top of the relational engine. In practice, nowadays, many people overlook object-relational features because of that.

In operational scenarios, the object-relational layer provides means to bridge the impedance mismatch previously discussed. Nothing more, nothing less. Now, you have the option to take advantage of object-oriented features within your database. If so, you can do it in two different ways:

- You can benefit from object-oriented features while continuing to work with most of your data relationally (object views), or
- You can entirely go over to an object-oriented approach (i.e., object tables and object views).

On the one hand, you still benefit from key features of the relational model: transactions and concurrency, backup and recovery, row-level locking, read consistency, partitioned tables, parallel queries, cluster databases, etc. On the other hand, you have means to deal with your stored data in an object-oriented fashion. Thus, your data is stored as it is handled by your applications

(e.g., purchase or customer instead of tables and columns) and, accordingly, the object-oriented layer provides higher-level means to organize and access data and a way to store and share code.

Although this higher level of abstraction is appealing, nowadays, most developers are used to mapping between the object-oriented and relational model and decline to use these features.

Data Mappers

Nowadays we can find some advanced tools, such as Hibernate, which automatically map objects defined in HLLs to relational.

While this is true for operational environments (which naturally suit the tabular representation behind the relational model), there are many other scenarios where this mapping is far away from being natural (not to say rather impossible to automate). In these scenarios, object-oriented features provide something else than syntactic sugar.

For example, consider a data mining tool nurtured from a specific database. Traditional data mining approaches usually extract data from the database, format it according to their needs and batch process it. However, data mining algorithms deal with enormous amounts of data and it is well known that extracting data from the database (and formatting it) is a bottleneck for such tools.

In such scenario, object-oriented features have been successfully proven to outperform a classical relational approach. To tackle this problem, novel approaches proposed to move the data mining algorithms within the database, instead of moving data out of the database. By means of UDTs, data mining algorithms can be encapsulated as object methods and efficiently handle data within the database. Some tests show a gain of several orders of magnitude between both approaches.

Thus, you should be able to analyze your scenario and consider if object-oriented features might facilitate your task. In general, object types are efficient, because types and methods are stored altogether with data (thus, programmers can benefit from reusability). In this way, a set of objects can be fetched from disk as a single I/O operation.

Do not underestimate object-relational features. In most cases, it is true that they do not provide anything else but syntactic sugar. However, they have been proved to be of great value in some scenarios where relational approaches do not naturally fit.

Suppose you decide to go for an object-relational implementation. Yet, there are many issues to be considered. For example:

1) Objects are provided with an OID by default. If you implement object tables, the only available identifier will be the object OID. Oracle OIDs, though, are 16 bytes long and therefore, can affect performance due to its size. Alternatively, you can store the object as an object column within a relational table, and better use the table primary key for your queries. Which approach is better? It depends on the following criteria:

- The OID is always 16 bytes long. The primary key has an ad hoc size depending on the built-in type used to implement it. For example, an integer is usually 4 bytes long.
- The primary key provides semantics regarding the scenario. For example, an ID card has clear semantics. However, an OID is an artificial identifier, with no semantics at all.
- The OID is guaranteed to be unique. The primary key, however, could just be unique in terms of our database. For example, the uniqueness of a given ID card from Spain cannot be guaranteed if other countries are considered (i.e., another country could have issued the same ID number).

If you decide to go for primary keys, be sure it is unique not only internally in the table but at the domain level.

2) You can decide to materialize derived attributes or use methods to compute them. Materializing them you gain query efficiency but it may result in data inconsistencies, whereas methods are slower but guarantee data consistency.

3) As middle ground solution, you might go for object views, which provide both models advantages (your data is still relational but your applications communicate with the database in an object-oriented fashion), but also disadvantages (for better or worse, an object view suffers from the same problems as any other view and thus, well-known problems related to views, such as view updating and query rewriting, also hold for them).

4) As a drawback to bear in mind, object types cannot benefit from parallelism unless you provide the map member function, which tells Oracle how to compare objects.

See also

See the fullname type definition at the beginning of section 1.2.

Summing up, you should go for an object-relational approach (i.e., use object types) whenever you are interested in hiding the relational layer to your applications or whenever the object-relational features might improve your system performance (either because your data structures cannot be easily mapped to relational or because you can better benefit from the object-oriented paradigm). As always, a middle ground solution mixing both approaches is an alternative to assess.

1.2.2. References between Objects

The REF type allows referencing objects and it is implemented as an object pointer. This type contains:

- The OID of the referenced object (16 bytes long) and
- The OID of the table or view containing that object (16 bytes long).

Optionally, it can also contain the rowid value (10 bytes long) for the referenced object.

RowID

The Oracle rowid value returns the row physical address.

REFs can be defined as scoped or unscoped ones. Scoped REFs are strongly typed and point to an object table. Therefore, the object type referred is known beforehand. Note that a scoped REF can point to an object of the scope type or to any of its subtypes.

Oppositely, the pointed object type is unknown for unscoped REFs. For this reason, scoped REFs are, in principle, more efficient, since the optimizer can get into play. Furthermore, unscoped REFs neither accept constraints nor indexes.

See also

See section "Practical issues on REFs" for further details.

For example, consider yet another possibility to model the `song` and `artist` types:

```
CREATE TYPE artist UNDER person (
   country VARCHAR2(10),
   genre VARCHAR(20)
);
CREATE TABLE artists OF artist;

CREATE TABLE single (
   artist_ref REF artist SCOPE IS artists,
   title VARCHAR2(20),
   album VARCHAR2(20)
);

INSERT INTO single SELECT REF(a),'Chasing Pavements','19'
FROM artists a
WHERE a.name = 'Adele';
```

Note

The examples in this section can be found in file 1.2. REFs_code.

In this example, a type `artist` is created and referred from the relational table storing singles. Thus, to fill the REF attribute properly, we use the REF keyword.

An alternative to obtain a REF is the following example in PL/SQL:

```
DECLARE artist_ref REF artist;
BEGIN
   SELECT REF(a) INTO artist_ref
   FROM artists a
   WHERE a.name = 'Adele';
END;
```

Additionally, REFs can be deferenced and therefore, navigate through them to the object containing the REF type. Following with our example:

```
SELECT DEREF(s.artist_ref), s.title
FROM single s;
```

Indeed, implicit deferences are possible, as shown below:

```
SELECT s.artist_ref.name, s.title
FROM singles s
WHERE s.album = '19';
```

In the example, `s.artist_ref.name` follows the pointer from the single's artist, and retrieves the artist name. Note, however, that implicit deferences are allowed in SQL but not in PL/SQL.

Finally, we call a *dangling pointer* to those REFs storing a value no longer pointing to an object. An OID stored by a REF is no longer available if:

- The referred object has been deleted from the database (this is possible because REFs only guarantee that the referenced table does exist) or
- By revoking privileges.

We can check if a REF is dangling by means of the `IS DANGLING` keywords. As said, REFs only guarantee that the pointed table does exist, but allows deleting the object pointed. If we want to enforce that the object exists, we can add the referential integrity to REFs (whenever it is not defined in a nested table). For example, we could redefine the previous single table as follows:

See also

See section "Nested Tables".

```
CREATE TABLE single (
   artist_ref REF artist REFERENCES artists,
   title VARCHAR2(20),
   album VARCHAR2(20)
);
```

REF cannot be used to point at column objects.

Practical issues on REFs

REFs are an essential mechanism to reference objects and therefore, a basic tool to enforce integrity constraints, like foreign keys in the relational model. Remember, though, that REFs co-exist in the object-relational model with foreign keys. So, we should deal with the main differences between both concepts.

Foreign keys point to candidate keys (i.e., `UNIQUE NOT NULL`), whereas REFs point to objects. In the object-relational model, though, dangling REFs are allowed (it means that the pointed object could no longer exist). However, a foreign key is never dangling and guarantees that the candidate key pointed does exist.

Anyway, as previously discussed, we can enforce the referential constraint in a REF and therefore, note that it is exactly the same notion as the foreign key but for the object-oriented paradigm. It means that still some differences exist; for example, a REF is an object type and it has some available pre-defined methods and, importantly, a foreign key `NULL` value (i.e., a bit) is dealt with in a different way than `NULL` pointers for REFs (i.e., it is treated as an object pointer).

NULL object pointers

Oracle distinguishes between two kind of `NULL` object pointers: objects whose values are `NULL` and those whose attributes are all `NULL`. For the second kind, space is allocated and set to `NULL` for each of their attributes. In the first case, Oracle does not allocate space and stores the `NULL` value in a bit.

Finally, note that, although scoped REFs are intended to be more efficient (the optimizer can be used as well as indexing techniques), they do not exploit the rowid attribute. The Oracle rowid value is a physical pointer to the tuple and therefore, unscoped REFs can outperform scoped ones by using the rowid for traversal searches.

Note that foreign keys have been traditionally used in relational databases to simulate hierarchies (i.e., generalizations / specializations). However, foreign keys (or REFs) do not provide as much semantics as inheritance, but just referential integrity (indeed, in the relational model, specializations did not inherit the parents' attributes at all, but we were able to reach them by navigating the mandatory foreign key between both relations).

Oppositely, the object-relational inheritance provides real inheritance of both attributes and methods.

1.2.3. Collections

Oracle provides two types to support collections: the VARRAY and nested tables. While the first one nicely suits the ARRAY type defined in the SQL-99 standard, nested tables happen to be much more interesting and provide new and powerful alternatives.

Note

The examples in this section can be found in file 1.2.3. VARRAYs_code.

VARRAY

A VARRAY is an ordered collection of elements where each element is associated to an index. In other words, it maps the traditional array concept of object-oriented programming languages. VARRAYs demand to specify, at design time, the maximum number of elements they can store (however, note that it can be modified later, as shown below).

The general syntax to define VARRAYs is as follows:

```
CREATE TYPE name AS VARRAY(limit) OF datatype | type;
```

Where `name` is the name of the new VARRAY type, `limit` is the number of elements this collection can contain and finally, after the `OF` keyword, we must specify a built-in datatype or user-defined type this collection is made of.

For example:

```
CREATE TYPE list_of_songs AS VARRAY(100) OF VARCHAR2(30);
CREATE TABLE adele_fans (
    fan_name VARCHAR2(30),
    favourite_songs list_of_songs
);
```

Alternatively, we could create the `song` object type and declare a VARRAY of songs:

```
CREATE TYPE song AS OBJECT (
 artist VARCHAR2(30),
 title VARCHAR2(30),
 rating NUMBER(2)
);
CREATE TYPE array_of_songs AS VARRAY(100) OF song;
CREATE TABLE adele_fans (
   fan_name VARCHAR(30),
   favourite_songs array_of_songs
);
```

We can now insert data in this table by properly calling the type constructor. For example:

```
INSERT INTO adele_fans  VALUES ('John Smith',
  array_of_songs(
    song ('Adele', 'Chasing Pavements', 8),
    song ('Creedence Clearwater Revival ', 'Susie Q', 9),
    song ('The Eagles', 'Hotel California', 10))
);
```

Importantly, note that VARRAYs require, at definition time, the maximum number of elements they can store. However, this limit can be replaced by a greater number by means of the `ALTER TYPE ... MODIFY LIMIT` statement:

```
ALTER TYPE array_of_songs MODIFY LIMIT 200 CASCADE;
```

Importantly, VARRAYs are stored inline (that is, as any other attribute, within the row). If the VARRAY happens to be too large (more than 4000 bytes), it is internally stored as a `LOB` attribute. VARRAYs of nested tables are immediately considered as `LOBs`.

Note that VARRAYs provide a set of interesting pre-defined methods you can use for your convenience. For example: `EXISTS` (to know if an element is already in the collection), `COUNT`, `FIRST`, or `LAST` (to respectively get the first / last element of the collection).

Nested Tables

A nice feature of the Oracle ORDM is that the type of a column can be a table-type. That is, the value of an attribute in one tuple can be an entire relation. Nested tables are defined as follows:

```
CREATE TYPE object table name AS TABLE OF type;
CREATE TABLE name (
  List of attributes,
  List of nested tables,
)
NESTED TABLE nested table column STORE AS nested table name;
```

Where `name` is the name of the table containing the `NESTED TABLE` columns. This table defined a set of attributes (some of them being nested tables). To define a nested table we need to previously create an object table. Now, we simply have to provide a column name of that object table type (i.e., the object table type previously defined). Later, we must specify that this `NESTED TABLE` column is stored in a relation whose name is provided after the `STORE AS` keywords. For example:

```
 CREATE TYPE single AS OBJECT (
  title VARCHAR2(30),
  album VARCHAR2(20),
);
CREATE TYPE single_table AS TABLE OF single;

CREATE TABLE singers (
   name VARCHAR2(30),
   releasedSingles single_table
)
NESTED TABLE releasedSingles STORE AS single_nt;
```

NESTED TABLE

Be careful with the syntax of nested tables. Note that there is just one semicolon in the definition of the singers table and it goes after both the parenthesized list of attributes and the `NESTED TABLE` column).

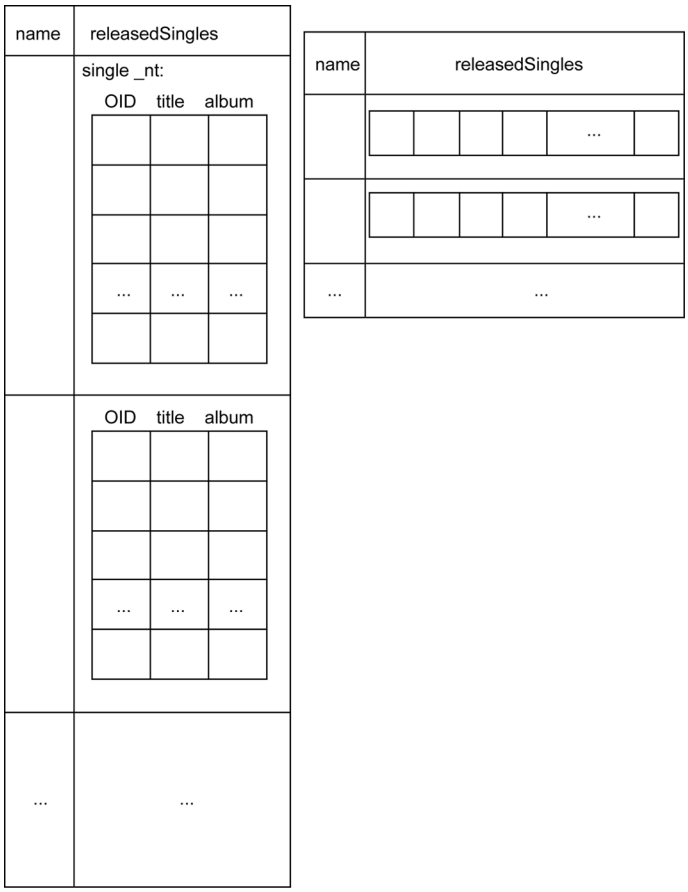
Nota

The examples in this section can be found in file 1.2.3. nested_tables.

To better understand how it works, figure 1 (left-side) sketches the nested table introduced in the example above. Furthermore, figure 1 confronts nested tables and VARRAYs. On its right side you can see the sketched representation of an alternative implementation of the above example by using VARRAYs.

Nested tables can have any number of elements (unlike VARRAYs, which require a maximum number of elements at definition time) and interestingly, they are handled as an ordinary table. Thus, we can insert tuples in it, index it, create triggers, enforce integrity constraints, etc.

Figure 1. Implementing collections in the object-relational model.



Relevantly, nested tables are not stored inline. Instead, they are stored as individual relations, whose name must be declared at definition time (after the `STORE AS` keywords). In our example above, sketched in figure 1, the whole nested table (i.e., `single_nt`) is stored in a relation apart. For example, consider two different `singer names` (e.g., Adele and The Eagles). All their singles (stored in the nested table in the `releasedSingles`) are stored in the same relation (`single_nt`) that is stored outside the `singers` table. Thus, there would not be two different tables, one for Adele singles and another one for The Eagles singles, but just one.

However, we cannot refer to this relation in any sense. When inserting data in a nested table, note that we use the nested relation type constructor (similarly to previous examples). For example:

```
INSERT INTO singers (name) VALUES ('Adele');
UPDATE singers SET releasedSingles = single_table (
   single('Chasing Pavements', '19'),
    single('Daydreamer', '19'),
    single('Cold Shoulder', '19')
)
WHERE name = 'Adele';
```

Optionally, when defining the nested table, we can use the `DEFAULT` keyword to automatically call the `single_table` constructor. For example:

```
CREATE TYPE single (
   title VARCHAR2(30),
   album VARCHAR2(20),
 );
CREATE TYPE single_table AS TABLE OF single;

CREATE TABLE singers (
    name VARCHAR2(30),
    releasedSingles single_table DEFAULT single_table()
 )
NESTED TABLE releasedSingles STORE AS single_nt;

INSERT INTO singers (name) VALUES ('Adele');
```

This way, after just inserting the name, the `single_table` constructor is automatically called.

Since nested tables are relations. We can use triggers, checks or any other statement available for regular tables. For example, we can create indexes over it:

```
  CREATE INDEX album_idx ON single_nt(album);
```

Importantly, note that we refer to the name provided in the `NESTED TABLE` statement to create the index.

Nested table

A table can contain as many nested tables as desired. Furthermore, nesting several tables is also allowed.

All in all, nested tables represent a powerful tool and provide an alternative to store objects (until now, we have only seen object tables), which open a whole new bunch of alternatives. For example, note that we can think of nested tables as attribute grouping and therefore, use them to simulate vertical fragmentation (not supported by Oracle natively). For example:

```
CREATE TYPE single (
   title VARCHAR2(30),
   album VARCHAR2(20),
 );
CREATE TYPE single_table AS TABLE OF single;
CREATE TYPE award (
   name VARCHAR2(30),
   category VARCHAR2(5),
 );
CREATE TYPE award_table AS TABLE OF award;

CREATE TABLE singer (
   name VARCHAR2(25),
   singles single_table,
   awards award_table
 )
NESTED TABLE singles STORE AS singer_singles_nt
NESTED TABLE awards STORE AS singer_awards_nt;
```

Each nested table is stored independently and therefore, by definition, we are performing a vertical fragmentation (whenever we access the singles nested table, we are accessing data related to singles and only that data).

Practical Issues on Collections

Collections are useful to implement multi-valued attributes. VARRAYs store multi-valued attributes column-wise (i.e., as an attribute in a relational row), while nested tables do it row-wise (i.e., store each element in a row).

Regarding design considerations, consider the following storage features for VARRRAYs:

- The VARRAY size depends on the number of elements it can hold (specifically, number of elements * size + overhead, where overhead are `NULL` values) and it is always stored as `RAW` data.
- According to the `LIMIT` value defined, the VARRAY is either stored inline or in `LOBs`.
- If the whole collection is manipulated at once, it behaves much better than nested tables (it is fetched at once, unlike rows in a nested table).

With regard to nested tables, the following considerations hold:

- It is exactly stored as a relation.
- If the nested table has a primary key, it is organized as an index-organized table (IOT).

IOT

An IOT is the Oracle version for clustered indexes.

As a common storage feature, none of them allocate memory at definition time.

Nevertheless, we can also implement collections in a fully relational fashion. The next table summarizes the options we have and the features provided by each solution. The table is structured as follows; we distinguish between relational and object-relational solutions and, within them, between column-wise or row-wise collections (see figure 2).

For example, the first column proposes to implement the collection column-wise within a relational table. The second one also goes for a relational solution, but storing it row-wise. The two last columns implement collections taking advantage of object-relational features and they correspond to the VARRAY and nested table types we have already been discussing at the beginning of this section.

Table 3. Alternatives to implement collections in both the relational and the object-oriented model.

Relational		Object-relational	
Per column	**Per row**	**VARRAY**	**Nested Tables**
Fixed number of values	Variable number of values	Fixed number of values	Variable number of values
Few values	Many values	Many values (LIMIT required)	Many values
Generates nulls	There are no null values	One null	There are no null values
One I/O	Many I/O	One I/O	Many I/O
Global processing	Partial processing	Global processing	Partial processing
Natural PK	Artificial PK	Natural PK + Indexes	OIDs
Less space	More space	Less space	More space
Hard to aggregate	Easy to aggregate	Hard to aggregate (Methods/Extensibility)	Easy to aggregate
Many CHECKs	One CHECK	No CHECKs	One CHECK (not for REFs)
Lower concurrency	Higher concurrency	Lower concurrency	Higher concurrency

Each row must be read as follows:

- The first one tells us whether the chosen implementation accepts a variable number of values or not. For example, a column-oriented relational implementation and VARRAYs need to know, at design time, the number of elements to be stored as maximum. We can obviously modify it later (by means of an `ALTER TABLE` and an `ALTER TYPE`, respectively) but the other two options can handle this issue dynamically.

- The second row tells us if, compared to the other approaches, that implementation can store a few or many values. The relational column-oriented solution is clearly the less flexible according to this criterion, because we need to define as many columns as elements in the collection. VARRAYs could eventually store large numbers of values, but the maximum number of values to store have to be expressed at definition time by means of the `LIMIT` keyword.

- The third row reflects if `NULLs` are generated. The relational row-oriented implementation and nested tables do not generate `NULLs` at all, but the first one does, if we do not straightforward fill the whole collection.

- The fourth row accounts for the number of I/O operations needed to retrieve the whole collection. Column-oriented solutions are able to read the whole collection with one I/O. If the collection is stored as row-oriented, we might need to read many blocks to retrieve all the rows. Note, however, that the VARRAY type is moved out of the table when it gets too

large and needs to be stored as a LOB. Therefore, it is not true anymore that a single I/O operation retrieves the whole collection.

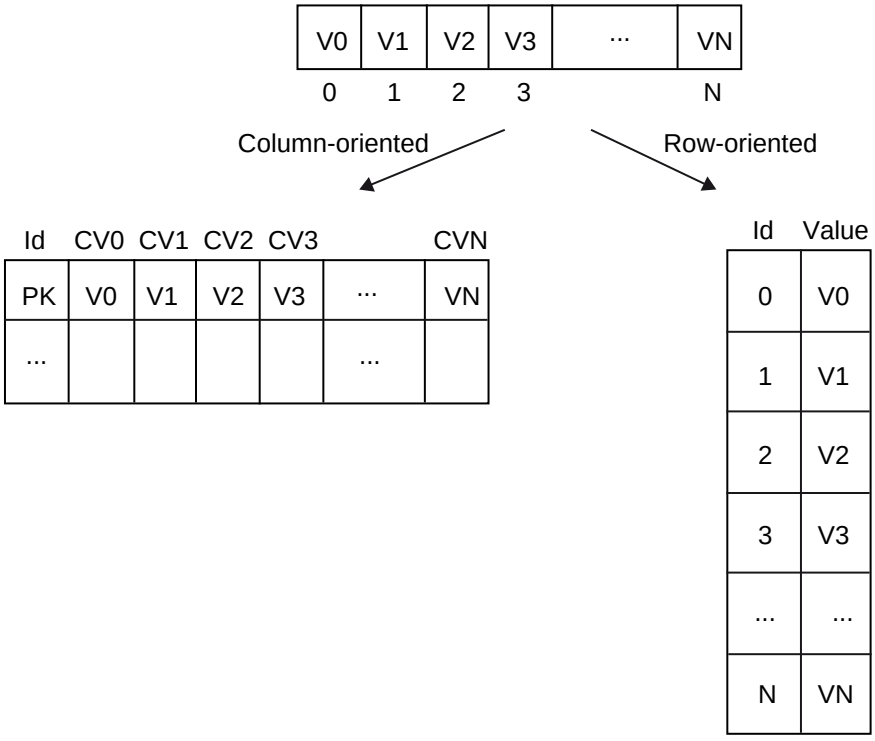
- The fifth row tells us if this solution is appropriate to deal with all the elements at the same time or if it is better suited for dealing with partial collections. Column-oriented solutions perform better when reading the whole collection at once, and row-oriented solutions deal better with partial processing.

- The sixth row tells us what kind of identifier we can use in each solution. In the first case, nothing prevents us to add a new column containing a natural primary key for such collection. Later, we can access each element according to its own name, which is, again a natural identifier. VARRAYs behave similarly, except for the use of indexes to access the elements. Implementing the collection per rows in a relational table means that we need to come up with an artificial primary key (for example, surrogates) to identify each element. Nested tables, however, use OIDs.

- The seventh row tells us the amount of space to be used: inline solutions (i.e., column-oriented) are cheaper, because no extra space is needed.

- The eighth row tells us whether performing aggregations over the collection is easy or not. Row-oriented solutions behave better in this case, because we can solve the aggregation by means of a SQL statement. In column-oriented ones we should iterate the collection over instead.

Figure 2. Implementing collections in the relational model.



- The ninth row focuses on enforcing constraints. Thus, it tells us how many CHECKs we do need to enforce a constraint over all the elements of the collection. In row-oriented approaches we can use a single `CHECK` to be enforced in all rows, whereas column-oriented ones require a `CHECK` for each element in the collection.

- Finally, the last row tells us the degree of concurrency provided by such solution. In this sense, row-oriented solutions benefit from row-locking (thus, we only lock one element of the collection). Column-oriented ones block the whole collection when accessing a single element, because the whole collection is retrieved at once.

In general, if you need to store a fixed number of items, loop through the elements in order or you (usually) retrieve and manipulate the entire collection as a value then, use a VARRAY (or a column-oriented relational approach). However, if you need to run efficient queries on a collection, handle arbitrary numbers of elements, or perform mass insert, update, or delete operations, then use a nested table (or its row-oriented relational counterpart).

2. The XML Extension

Thanks to the emergence of telecommunications and computer networks, information systems have evolved from quiet islands, usually just connected to other systems in the same organizations or in tightly related ones, to highly connected systems.

Lately, the scope for these information systems has become even planetary as a result of their integration into the World Wide Web (WWW). This is a great opportunity but also carries one a great challenge. Until recently, all data in and out these systems were generated inside and under the control of the IT department of the organization. Consequently, it was easier to define a common schema for the data.

However, when operating at a global scale, the sources of data to be integrated are no longer under your control. In this context, the rigid schema of a relational database does not seem appropriate to store data. Something more flexible was needed.

The proposal here was to go from the table-based model of relational databases to a more flexible tree-based model. This model is enhanced with a flexible schema language, XML Schema, which integrates reuse by extension and restriction mechanisms, so it is easier to share and reuse schemas, anticipate the structure of input data and accommodate it into existing schemas.

Another key element are namespaces, that allow partitioning the names used in these schemas into naming spaces controlled by the data publisher so naming clashes can be more easily avoided.

Finally, there is the XQuery query language that leverages all the flexibility of the underlying data model and schemas using an SQL-shaped syntax that makes the transition from relational databases easier. XQuery allows selecting the required information from XML documents while reducing the commitment to a pre-established structure of the data. Moreover, it also allows generating XML as output, enabling an XML-based pipeline-oriented processing engine that can be even used to implement a whole WWW application.

The drawback of XML and related technologies is that though more flexible, they are less efficient. They require more storage space, which becomes unsustainable for binary data, and more processing time due to less mature technologies and the increased complexity of the involved data structures and schemas.

2.1. XML Fundamentals

XML stands for eXtensible Markup Language and it is a very flexible text format derived from SGML (ISO 8879). It provides a set of rules for encoding documents in a machine-readable form. Thus, each individual piece of information is 'marked up' (a marker shows the meaning of the associated data) with a tag attached that is called an element.

Note

The XML Specification is available from http://www.w3.org/TR/REC-xml

This element consists of a start tag, text, and an end tag, like in HTML (HyperText Markup Language). When required, attributes can be associated to the start tag of an element, allowing more detailed information to be assigned to the data. This basic syntax is illustrated in figure 3.

Figure 3.The basic XML structure

```
<artist type="Person"> Adele </artist>
```

Inside a tag, the element name for the start and end tags must match and lower case/upper case characters are differentiated in element names. Moreover, blank spaces, tabs, carriage returns, or line feeds cannot be included between the '<' character and the element name. The same restrictions apply to the '/' character immediately following the end tag '<' character, as well as to the '/' character and the element name immediately following. No spaces may exist between characters in an element name.

Attributes are properties that provide additional information about the element they are associated to. They are always marked with single or double quotes and stand as an additional value to a label `<a attribute="attribute_value"/>`. For the same element, attribute names are unique, so no more than one attribute with the same name can be attached to the same element.

Based on these building blocks, different combinations of them can be made. For instance:

- Elements with just text:

```
<a> Text </a>
```

- Elements containing other elements and attributes:

```
<a atr="val"> <b> </b> </a>
```

- Elements with attributes, text and subelements:

```
<a atr="val"> <b> </b> text </a>
```

- Empty elements, which can be written down in compact form like:

```
<a/>
```

However, with so much freedom, as shown in Code 1, when should we use an attribute and when an element to model a piece of information?

Code 1. Two choices for modeling an artist with two properties, his/her type and name

```
<artist type="Person" name="Adele">
</artist>
```

or

```
<artist>
  <type>Person</type>
  <name>Adele</name>
</artist>
```

Two complementary guidelines can be used to help making this kind of choices in a more systematic and consistent way:

1) Conceptual guideline, choose:

- Attribute for values without their own identity, for instance age, or
- Subelement for values with their own identity, for instance date of birth.

2) Content vs. Metadata guideline, choose:

- Attribute for metadata or descriptive information about content. For instance the length of a piece of content or its language, or
- Subelement for content, for instance a title.

The two guidelines can be combined to help decide how to model `type` and `name` in the example in Code 1. For `type`, though the category of things called `Person` might have its own identity, in this case the attribute type is used to characterize the entity `artist` being modeled so the best choice seems to follow the second guideline and model `type` as and attribute because it is providing metadata for the main content. Similarly, the choice for `name` seems clearer following the second guideline though in this case, as a piece of fundamental content, it seems more convenient to model `name` as a subelement. The result after applying the guidelines is shown in Code 2.

Code 2. Chosen syntax after applying modeling guidelines for the alternatives in Code 1

```
<artist type="Person">
  <name>Adele</name>
</artist>
```

Some additional notes about XML syntax:

- A piece of XML is marked as such by starting with the following expression:
  `<?xml version="1.0" encoding="UTF-8"?>`

- In addition to the XML version, it also specifies the encoding used to codify characters, which is especially relevant for those specific to a particular language. XML supports the following codification schemes:
  UTF-8, UTF-16, ISO-10646-UCS-2, ISO-10646-UCS-4, from ISO-8859-1 to ISO-8859-9, ISO-2022-JP, Shift_JIS and EUC-JP. ISO-8859-1 is the typical choice for "Western European" languages. However for broader coverage XML defaults to UTF-8, which is the common choice.

2.1.1. Well-formed XML

Following with this basic syntax, the XML specification defines an XML document as a text that is well formed, i.e. it satisfies a list of syntax rules provided in the specification. The main rules that a well-formed XML document should satisfy are:

- It contains only properly encoded legal Unicode characters.

Unicode

Unicode is a computing industry standard for the consistent encoding, representation and handling of text. Unicode can be implemented by different character encodings being the most common UTF-8.

- None of the special syntax characters such as '<' and '&' appear except when performing their mark-up delineation roles.
  When you need any of these special characters but you do not want them to be interpreted as part of the XML syntax, as part of the content you put between tags or in attributes, use the corresponding entities as shown in Code 3. They are replaced with their corresponding character after the XML syntax has been processed.
  For instance, if you want to include a piece of XML inside the content of an element but it should not be interpreted as XML, you can replace all special characters with the corresponding entities, like in Code 4.
  Another alternative is to use the `<![CDATA[…]]>` construct to mark a set of characters that should not be interpreted as XML markup. An alternative to Code 4 based on this option is shown in Code 5.

Code 3. XML entities for XML special characters

```
Character      Entity
  &            &amp;
  <            &lt;
  >            &gt;
  '            &apos;
  "            &quot;
```

Code 4. Encoding XML syntax to include it in element content

```
<example>
   &lt;artist
      type=&quot;Person&quot;&gt;
      Adele
   &lt;/artist&gt;
</example>
```

Code 5. Encoding XML syntax to include it in element content using CDATA region

```
<example>
   <![CDATA[
      <artist
      type="Person">
      Adele
   </artist>]>
</example>
```

- The beginning, end and empty-element tags that mark the elements are correctly nested, with none missing and none overlapping, i.e. not as shown in Code 6.

Code 6. Examples of missing name closing element and overlapping artist and name elements

```
<artist><name>Adele</artist>
<artist><name>Adele</artist></name>
```

- The element tags are case-sensitive; the beginning and end tags must match exactly.

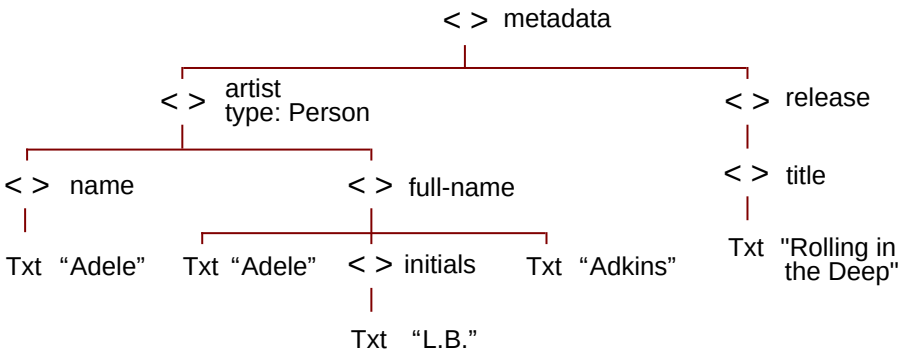
- Tag names cannot contain any of the characters " # $ % & ' ( ) * + , / ; < = > ? @ [\ ] ^ ` { | } ~, nor a space character, and cannot start with - . or a numeral.

- There is a single root element that contains all the other elements, which might be nested. Altogether, the XML syntax encodes an underlying tree data structure, as shown in Figure 2 for the XML syntax in Code 7.

Codi 7. A piece of XML syntax that corresponds to the tree structure in Figure 4

```
<metadata>
   <artist type="Person">
      <name>Adele</name>
      <full-name>
         Adele<initials>L.B.</initials>Adkins
      </full-name>
   </artist>
   <release>
      <title>Rolling in the Deep</title>
   </release>
</metadata>
```

Figure 4. Tree structure for the XML example in Code 7



2.1.2. Namespaces

Now, continuing with the example in Code 7, let's imagine that you want to combine that piece of XML with another piece you have generated. Unfortunately, you have also used the element name `metadata` in your XML but for a different purpose and thus with a different structure. In order to avoid these name clashes, XML incorporates naming spaces.

A namespace is identified by a URI and as long as you define a unique URI for your namespace, you can then use whatever element and attribute names inside it. The best way to get a unique URI is to use part of your organization or own domain name to build a custom URL for your namespace. For instance, for someone at the UOC it is possible to pick the `uoc.edu` domain name and build a URI for the namespace like `http://www.uoc.edu/subjects/adb/ns/custom#`.

Uniform Resource Identifier

A Uniform Resource Identifier (URI) is a string of characters used to identify something on the Internet. URIs include names (URNs) and locators (URLs). A URN is a URI that works as a identifier, like the ISBN of a book that can be used to build the URN urn:isbn:0-486-27557-4. This URN just identifies a book while a URL also provides access to it, like the HTTP URL http://www.bookfinder.com/dir/i/Romeo_and_Juliet/0486275574.

As shown in Code 8, it is possible to combine the namespaces for the original data, defined in a MusicBrainz namespace, and the custom XML data without clashes by defining the namespace each element name or attribute belongs to. In order to avoid typing the whole namespace URI each time, it is possible to define aliases, short names for the namespaces that are prepended to the element and attribute names together with ':'.

These alias are usually defined at the beginning of the XML document, thought they can be defined for any element and then apply to that element and all its subelements. Aliases are defined using a special attribute that starts with `xmlns:` and then the alias. The value of the attribute is the URI it refers to. Finally, there is the default namespace, which is defined using the `xmlns` attribute and applies to all elements that do not explicitly define their namespace. Elements and attributes define their namespace by prepending the namespace alias plus ':' to the element or attribute name.

Attribute

Attributes without an explicit namespace are not in the default one, they simply do not have one. In other words, an attribute is only in a namespace if it has a proper prefix declared as an alias for an XML namespace. Attribute-name clashes are only relevant in the rare case when attributes with the same name appear for the same element; consequently it is very uncommon to define namespaces for attributes.

To conclude the scenario posed at the beginning of this section, Code 8 shows the piece of XML in Code 7 plus a piece of custom XML that provides additional metadata for the release. To avoid name clashes, the original XML from MusicBrainz is defined in the `http://musicbrainz.org/ns/mmd-2.0#`, which

is the default namespace, so it applies to every element that does not specify an alias. The custom XML is in the `http://www.uoc.edu/subjects/adb/ns/custom#` namespace, which is linked to the custom alias. Consequently, the elements in this namespace have their names prepended with `custom:`.

Code 8. A piece of XML that combines elements from different namespaces

```
<metadata
   xmlns="http://musicbrainz.org/ns/mmd-2.0#"
   xmlns:custom="http://www.uoc.edu/subjects/adb/ns/custom#">
   <artist type="Person">
      <name>Adele</name>
      <full-name>
         Adele<initials>L.B.</initials>Adkins
      </full-name>
   </artist>
   <release>
      <title>Rolling in the Deep</title>
      <custom:metadata>Must buy</custom:metadata>
   </release>
</metadata>
```

2.1.3. Full XML Example

This section introduces the first full XML example. This is a real piece of XML that can be obtained from the MusicBrainz online service using a simple link. The link encodes a call to the MusicBrainz API asking for data about a particular music release including information about the release artist and the included recordings.

Bibliography

Web recommended

This piece of XML data is available from: http://musicbrainz.org/ws/2/release/79ef6f41-51a3-3ee5-8b2f-7347de023e30?inc=artist-credits%2Brecordings

The response from this service is an XML document representing the data MusicBrainz has about the release, titled Rolling in the Deep, plus information about the artist, called Adele, and the two tracks contained in the release.

The first line in the document indicates that it is an XML file and that the UTF-8 encoding is used. Then, the root element `metadata`, introduces a pair of namespace definitions. The first one to the default namespace used by the MusicBrainz XML data. The second one is for a standard namespace from which the `schemaLocation` attribute is used.

This attribute is used to point to the namespace and location of the schema that the XML document instantiates, i.e., points to the schema defining the structure of the XML document. In this case, it is the one for the vocabulary used by the MusicBrainz service. More details about schemas are provided in the next section.

Note

XML document from the MusicBrainz service for the Rolling in the Deep music release by Adele. This example is available as file "Adele.xml".

```
<?xml version="1.0" encoding="UTF-8"?>
<metadata xmlns="http://musicbrainz.org/ns/mmd-2.0#"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://musicbrainz.org/ns/mmd-2.0#
    musicbrainz_mmd-2.0.xsd">
    <release id="79ef6f41-51a3-3ee5-8b2f-7347de023e30">
```

```
            <title>Rolling in the Deep</title>
            <status>Official</status>
            <quality>normal</quality>
            <text-representation>
                <language>eng</language>
                <script>Latn</script>
            </text-representation>
            <artist-credit>
                <name-credit>
                    <artist id="cC2c9c3c-b7bc-4b8b-84d8-4fbd8779e493">
                        <name>Adele</name>
                        <sort-name>Adele</sort-name>
                        <disambiguation>UK Soul/Jazz singer</disambiguation>
                    </artist>
                </name-credit>
            </artist-credit>
            <date>2011-01-17</date>
            <country>GB</country>
            <barcode>634904152123</barcode>
            <asin>B004DK49WI</asin>
            <medium-list count="1">
                <medium>
                    <position>1</position>
                        <track-list count="2" offset="0">
                            <track>
                                <position>1</position>
                                <length>229706</length>
                                <recording
                                    id="1a13c710-4b7e-4701-8968-cd61f2e58110">
                                    <title>Rolling in the Deep</title>
                                    <length>229000</length>
                                </recording>
                            </track>
                            <track>
                                <position>2</position>
                                <title>If It Hadn't Been For Love</title>
                                <length>186933</length>
                                <recording
                                        id="addd7af6-36c5-4626-a623-aC23b8bc3d2e">
                                    <title>If It Hadn't Been for Love</title>
                                    <length>188000</length>
                                </recording>
                            </track>
                        </track-list>
                    </medium>
            </medium-list>
        </release>
```

```
    </metadata>
```

2.1.4. Storing XML Documents in Oracle XML DB

Oracle XML DB is the name for a set of Oracle Database technologies that provide XML support by encompassing both SQL and XML data models in an interoperable manner. The Oracle XML DB Repository is the component of Oracle Database that handles XML data. It contains resources, which can be either folders or files. Each resource is identified by a path and name. In the case of files, their content can be XML data but need not be.

Thanks to the Oracle XML DB Repository, Oracle provides a hierarchically organized repository that can be queried and through which XML content can be managed. A hierarchical index speeds up folder and path traversals.

A URL is used to locate an XML document and XPath is used to access and update content contained within XML documents. Both URLs and XPath expressions are based on hierarchical metaphors. A URL uses a path through a folder hierarchy to identify a document, whereas XPath uses a path through the node hierarchy of an XML document to access part of an XML document.

One key decision to make when using Oracle XML DB for persisting XML documents is whether to use structured or unstructured storage:

- Unstructured storage provides highest throughput when inserting and retrieving entire XML documents. It also provides the greatest degree of flexibility in terms of the structure of the XML that can be stored. These throughput and flexibility benefits come at the expense of less performance when working with documents at a finer granularity level. There is little the database can do to optimize queries or updates on XML stored using a Character Large Object (CLOB), Binary Large Object (BLOB), Binary File (BFILE), or VARCHAR column.

- Structured storage is based on the XMLType, a new datatype that makes the database aware that XML is being stored. It has a number of advantages, including optimized memory management, reduced storage requirements, B-tree indexing that optimize XPath queries and in-place updates. These advantages are at a cost of a greater processing overhead during ingestion and retrieval and reduced flexibility in terms of the structure of the XML.

Bibliography

Shreding is the process of mapping the data in an XML document to table rows and columns in a relational database. More details are available from:

Ethan V. Munson (Springer, 2009). "Document Representations (Inclusive Native and Relational)". In: Ling Liu and M. Tamer Özsu (Eds.). *Encyclopedia of Database Systems* (pp. 942-946; ISBN 978-0-387-35544-3).

See also

XPath is described in Section 3.1.

Each of these approaches implies different procedures to load XML data into Oracle. For instance, we will consider the piece of XML corresponding to Adele's basic information and the list of all her release groups, which, in MusicBrainz terms, means the list of all her singles, albums and live recordings. Each one of them is a group of releases because it might have one or more actual releases, where each release might have different format, release country or label.

Web recommended

This piece of XML data is available from: http://musicbrainz.org/ws/2/artist/cc2c9c3c-b7bc-4b8b-84d8-4fbd8779e493?inc=release-groups+releases

To load it into Oracle XML DB using an unstructured approach:

```
DECLARE
    res BOOLEAN;
 BEGIN
    res := DBMS_XDB.createResource('adele-releasegroups.xml',
       HTTPURITYPE.createuri(
 'http://musicbrainz.org/ws/2/artist/cC2c9c3c-b7bc-4b8b-84d8-
 4fbd8779e493?inc=release-groups').getClob());
 END;
 /
```

Note

This code is available as file 1.4a.txt.

To load it into Oracle XML DB using the structured approach:

```
 CREATE TABLE table1 OF XMLType;
 INSERT INTO table1 VALUES (XMLType(HTTPURITYPE.createuri(
 'http://musicbrainz.org/ws/2/artist/cC2c9c3c-b7bc-4b8b-84d8-
 4fbd8779e493?inc=release-groups').getXML()));
```

Note

This code is available as file 1.4b.txt.

2.2. XML Schema

As we have seen in section 1.1., a XML document is well formed if it follows some syntactic rules. However, this is just a small constraint on the structure of XML documents that makes them processable but allows building quite meaningless documents because these rules say nothing specific about the structure of the document; they do not define a schema.

Web recommended

For a detailed description of Oracle's XML storage features see http://docs.oracle.com/cd/B19306_01/appdev.102/b14259/xdb03usg.htm

The main purpose of XML is to provide a way to make applications communicate and in order to attain this, there should be some sort of understanding, some way of being able to anticipate the structure of the document and know how to interpret the pieces of data in that structure. In other words, there is a need for some sort of shared vocabulary for that particular communication context.

For this purpose, when dealing with XML, it is necessary to define all the element and attribute names that may be used. Moreover, to define the structure: what values an attribute may take, which elements must occur within other elements, how many times, in which order etc.

If a XML document is well formed and it is structured following one of these XML vocabularies, it is said to be a valid XML document. There are two main ways of defining XML vocabularies:

- **DTD** (Document Type Definition): this is the first standardized way of doing so. It is simpler but less expressive so most of the modern standards based on XML, which define a XML vocabulary to build meaningful documents in the particular application domain of the standard, use the next alternative instead.
- **XML Schema**: this second option is more complex than DTD but it is more expressive and thus allows defining vocabularies that better capture the particularities of the application domain. We will concentrate on this alternative, which is detailed in the next section.

2.2.1. Basic Concepts

As we have seen, XML Schema offers a richer language for defining XML vocabularies than DTD. The syntax of XML Schema is based on XML itself, which means that there is a XML Schema that defines the XML Schema vocabulary. This is the set of elements and attributes, plus the way of combining them, to define a XML vocabulary.

Web recommended

The XML Schema Specification is available from http://www.w3.org/TR/xmlschema-0

This is the first advantage over DTD, which is not based on XML, because this approach provides a significant improvement in readability, and, most importantly, it allows significant reuse of existing XML technology and refining schemas.

However, the fundamental advantage of a XML Schema over a DTD is that in addition to the element and attribute names and their structure, a schema can specify more sophisticated rules for the content of elements and attributes.

Any XML Schema builds on top of a set of built-in datatypes, called simple types, like string, boolean or integer. These primitives can be combined to build complex types, which can be then assigned as the content of an element-containing subelement. Simple types can be also restricted to derive new custom simple types. For example, the schema can restrict dates to those after the year 2000. Therefore, users can derive their own data types from the built-in data types or other derived types to define new types that meet the requirements of the vocabulary modeling process.

2.2.2. XML Schema Root

An XML schema is a document with an opening root element like:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://musicbrainz.org/ns/mmd-2.0#"
      xmlns:ext="http://musicbrainz.org/ns/ext#-2.0"
      xmlns:mmd-2.0="http://musicbrainz.org/ns/mmd-2.0#">
```

Note

The MusicBrainz XML Schema, used in the examples in this section, is available as file musicbrainz_mmd-2.0.xsd.

The element defines the namespace of XML Schema, found at the W3C web site. It is common practice to use the `xsd` extension to denote the namespace of that schema. It is the foundation on which new schemas can be built.

The new XML Schema is created in the context of the namespace set by the `targetNamespace` attribute. This means that all the elements and types defined by the new schema are placed in that namespace. Consequently, as seen in XML example[2], an instance XML document based on this schema will use as the first part of the `schemaLocation` attribute that namespace. The second part corresponds to a path or URL pointing to the file where the schema is stored.

[(2)]XML document from the MusicBrainz service for the Rolling in the Deep music release by Adele. This example is available as file Adele.xml.

Note

If an XML schema does not specify a `targetNamespace`, elements and types defined by the XML schema are associated with the NULL namespace.

Just after the XML Schema root element, it is possible to define other XML Schemas to be imported. This way, it is possible to reuse the types, elements and attributes defined by them for the current schema and derive new types, elements or attributes from them. This is done using the `import` element that specifies the schema location and optionally the namespace for the imported schema.

```
<xsd:import schemaLocation="local.xsd"/>
   <xsd:import namespace="http://musicbrainz.org/ns/ext#-2.0"
schemaLocation="extensions.xsd"/>
```

The XML Schema document starts from this point to provide definitions and declarations for the vocabulary terms to be defined. Definitions create new types. These can be complex types, which allow elements in their content and may carry attributes, and simple types, which cannot have element content or attributes. On the other hand, declarations enable elements and attributes with specific names and types, both simple and complex, to appear in document instances.

2.2.3. Complex Types

Complex types are defined using the `complexType` element, and their definitions contain a set of element declarations, element references and attribute declarations.

The declarations are associations between an element name and the complex type, which defines the constraints that govern the content of the element, the subelements and attributes, in documents instantiating the schema. Elements are declared using the `element` element, and attributes are declared using the `attribute` element.

Complex types are defined from existing data types combining them using one of the following primitives:

- **Sequence**: the subelements for the complex type structure defined inside of a sequence should appear in the appearing order in instance XML documents.
- **All**: the collection of subelements must appear, but the order is not relevant, they might appear in the instance XML documents in a order different from the order they appear in the XML Schema complex type.
- **Choice**: from the collection of subelements in the choice just one will be chosen in the instance XML documents.

By default, the elements defined in a complex type are required to appear just one time. However, it is possible to define their cardinality. The `minOccurs` attribute sets the minimum number of times that the element should appear and the maximum number of times an element may appear is determined by the value of the `maxOccurs` attribute.

This value must be a positive integer or the term unbounded to indicate there is not a maximum number of occurrences. The default value for both the `minOccurs` and the `maxOccurs` attributes is 1.

Attributes may appear once or not at all. Consequently, the syntax for attributes cardinality is different from the syntax for elements. Attributes can be declared with a `use` attribute to indicate whether the attribute is `required` or `optional`. Attributes are optional by default and to make them mandatory their declaration should have the `use` attribute with its value set to `required`. Moreover, if they are optional, they can have a default value defined using the `default` attribute. Finally, attribute content is a simple type, which is set using the `type` attribute.

2.2.4. Example

For example, in the MusicBrainz case the intended vocabulary should define references to the artists associated to a recording. To this end, a complex type is defined, which can be later associated to a new element like `name-credit`. The structure defined by the complex type is a sequence of an optional `name` subelement followed by a mandatory `artist` subelement.

The subelements are referenced using the `name` or the `ref` attributes. The former means that the subelement is declared at that point so it is also necessary to define its content using the `type` attribute. The latter means that the element is declared elsewhere in the schema (or imported schemas) and reused here. Finally, there is also an attribute declaration.

```
<xsd:complexType name="name-creditType">
   <xsd:sequence>
       <xsd:element minOccurs="0" name="name" type="xsd:string"/>
   ....<xsd:element ref="mmd-2.0:artist"/>
   </xsd:sequence>
   <xsd:attribute name="joinphrase"
   ....type="xsd:string" default=" &amp; "/>
</xsd:complexType>
```

As a result of this definition, any element whose type is declared to be `name-creditType`, such as `name-credit` in the following example, will optionally contain a `name` subelement but allways preceding the `artist` subelement. The element will optionally also have the `joinphrase` attribute.

```
<xsd:element name="artist-credit">
   <xsd:complexType>
   ....<xsd:sequence>
   ....   <xsd:element maxOccurs="unbounded"
   ....       name="mmd-2.0:name-credit"
   ....       type="mmd-2.0:name-creditType"/>
   ....</xsd:sequence>
   </xsd:complexType>
</xsd:element>
```

2.2.5. Simple Types

As we have seen in the previous section, elements can be declared to have content constrained to a defined complex type. The complex type defines the element subelements and attributes. The subelements, in turn, can also be constrained to complex types or to simple types.

In any case, at last, all complex types are rooted in simple types, which also define attributes content. The simple types can be those built into the XML Schema, shown in Table 4, or be derived from these built-ins, as detailed next.

Table 4. The simple types built-in the XML Schema standard

Simple Types	Examples (delimited by commas)	Notes
string	Confirm this is electric	--
normalizedString	Confirm this is electric	Newline, tab and carriage-return characters are converted to space characters
token	Confirm this is electric	As normalizedString, and adjacent space characters are collapsed to a single space character, and leading and trailing spaces are removed.
byte	-1, 126	
unsignedByte	0, 126	
base64Binary	GpM7	
hexBinary	0FB7	
integer, int	-126789, -1, 0, 1, 126789	
positiveInteger	1, 126789	
negativeInteger	-126789, -1	

Simple Types	Examples (delimited by commas)	Notes
nonNegativeInteger	0, 1, 126789	
nonPositiveInteger	-126789, -1, 0	
unsignedInt	0, 1267896754	
long	-1, 12678967543233	
unsignedLong	0, 12678967543233	
short	-1, 12678	
unsignedShort	0, 12678	
decimal	-1.23, 0, 123.4, 1000.00	
float	-INF, -1E4, -0, 0, 12.78E-2, 12, INF, NaN	equivalent to single-precision 32-bit floating point, NaN is Not a Number
double	-INF, -1E4, -0, 0, 12.78E-2, 12, INF, NaN	equivalent to double-precision 64-bit floating point
Boolean	true, false 1, 0	
time	13:20:00.000, 13:20:00.000-05:00	
dateTime	1999-05-31T13:20:00.000-05:00	May 31st 1999 at 1.20pm Eastern Standard Time which is 5 hours behind Co-Ordinated Universal Time
duration	P1Y2M3DT10H30M12.3S	1 year, 2 months, 3 days, 10 hours, 30 minutes, and 12.3 seconds
date	1999-05-31	
gMonth	--05--	May
gYear	1999	1999
gYearMonth	1999-02	the month of February 1999, regardless of the number of days
gDay	---31	the 31st day
gMonthDay	--05-31	every May 31st
Name	shipTo	XML 1.0 Name type
QName	po:USAddress	XML namespace QName
NCName	USAddress	XML namespace NCName, that is, QName without the prefix and colon
anyURI	http://example.com/doc.html#ID5	
language	en-GB, en-US, fr	valid values for xml:lang as defined in XML 1.0 that specify the language
ID, IDREF, IDREFS, ENTITY, ENTITIES, NOTATION		XML 1.0 ID attribute type
NMTOKEN	US, Canada	XML 1.0 NMTOKEN attribute type, a normalised string without spaces

Simple Types	Examples (delimited by commas)	Notes
NMTOKENS	US UK, Canada Mexique	XML 1.0 NMTOKENS attribute type, that is, a white-space separated list of NMTOKEN values

New simple types are defined by deriving them from existing built-in or previously derived simple types. The derivation is based on restricting the range of values of the derived simple type to a smaller set of intended values for the new simple type.

The definition is built using the `simpleType` element that sets the name for the new simple type. Then, the `restriction` subelement indicates the existing base simple type. Finally, there is a set of subelements that allow specifying in what way the base simple type set of values is restricted. XML Schema defines restriction facets. The main ones are:

Web recommended

For an exhaustive list of restriction facets, refer to Appendix B of the specification at http://www.w3.org/TR/xmlschema-0

- **Range**: this restriction can be applied to simple types derived from numeric or temporal types. The lower limit of the range is defined using the `minInclusive` or `minExclusive` subelement and the `value` attribute, whose value is included in the range if inclusive or not if exclusive. For the upper limit the subelements are `maxInclusive` and `maxExclusive`. For instance, to define a new simple type for the range of integers between 10 and 99, both included in the range, the new simple type is derived from `integer` and the `minInclusive` facet is set to 10 while the `maxInclusive` one is set to 99:

```
<xsd:simpleType name="myInteger">
    <xsd:restriction base="xsd:integer">
       <xsd:minInclusive value="10"/>
       <xsd:maxInclusive value="99"/>
    </xsd:restriction>
</xsd:simpleType>
```

- **Pattern**: this restriction allows defining a regular expression that constraints the desired combinations of characters from the simple type string. The constraint is set using a facet called `pattern` in conjunction with the regular expression `[A-Z]{2}` that is read "from the range of upper-case letters between A and Z, take two of them".

Bibliography

For a complete reference for regular expressions:

Stubblebine, T. (2007). *Regular expression pocket reference*. O'Reilly Media, Inc.

```
<!-- A two-letter country code like 'DE', 'UK',... so
 it does not include 'SPA' or 'USA' -->
<xsd:simpleType name="def_iso-3166">
   <xsd:restriction base="xsd:string">
      <xsd:pattern value="[A-Z]{2}"/>
   </xsd:restriction>
</xsd:simpleType>
```

Another example of pattern restriction from the MusicBrainz schema is about defining a simple type for ISWC (the international standard for identifying music creations):

```
<!-- An ISWC code:
   C           - single-letter prefix character
   NNN.NNN.NNN - 9-digits separated by "." grouped 3x3
   C           - check digit
   -->
<xsd:simpleType name="def_iswc">
   <xsd:restriction base="xsd:string">
     <xsd:pattern
       value="[A-Z]-[0-9]{3}\.[0-9]{3}\.[0-9]{3}-[0-9]"/>
   </xsd:restriction>
</xsd:simpleType>
```

- **Enumeration**: it can be used to constrain the values of almost every simple type, except the boolean type. The enumeration facet limits the new type to a set of distinct values. For example, the quality levels, derived from NMTOKEN (a normalized string without spaces), whose value must be `low`, `normal` or `high`:

```
<xsd:simpleType name="def_quality">
   <xsd:restriction base="xsd:NMTOKEN">
       <xsd:enumeration value="low"/>
       <xsd:enumeration value="normal"/>
       <xsd:enumeration value="high"/>
   </xsd:restriction>
</xsd:simpleType>
```

- **Length**: another way to limit the set of possible values for a simple type is by restricting their length to a particular quantity or to define a minimum or maximum length. This is done using the `length`, `minLength` and `maxLength` restriction facet elements.

```
<xsd:element name="password">
   <xsd:simpleType>
       <xsd:restriction base="xsd:string">
      <xsd:minLength value="5"/>
      <xsd:maxLength value="8"/>
       </xs:restriction>
   </xsd:simpleType>
</xsd:element>
```

The previous method allows for defining atomic simple types, which are simple type whose values are indivisible. Another option is to define compound simple types using lists or unions.

For example, the NMTOKEN `low` value is indivisible in the sense that no part of it, such as the character l, has any meaning by itself. In contrast, list types are comprised of sequences of atomic types and consequently the parts of a sequence (the atoms) themselves are meaningful. For example, NMTOKENS is a list type, and an element of this type would be a white-space delimited list of NMTOKEN values, such as `low normal`. XML Schema has three built-in list types:

- NMTOKENS
- IDREFS
- ENTITIES

In addition to using the built-in list types, you can create new list types by derivation from existing atomic types. You cannot create list types from existing list types, nor from complex types. For example, to create a list of lengths based on a list of integers:

```
<xsd:element name="listOfLengths">
<xsd:simpleType>
   <xsd:list itemType="myInteger"/>
</xsd:simpleType>
</xsd:element>
```

An instance XML document including a `listOfLenghts` element based on the previous definition can be:

```
<listOfLengths>189 187 191</listOfMyInt>
```

Several of the previous simple types restriction facets can be applied to list types: `length`, `minLength`, `maxLength` and `enumeration`.

In addition to atomic types and list types, which enable an element or an attribute value to be one or more instances of one atomic type, there are also union-based simple types. The union operation enables element or attribute values to be from the union of multiple atomic and list types.

For instance, it is possible to define a union type for country codes that combines the two-letters and two-letters plus subdivision simple types:

```
<!-- Two-letter country code like 'DE', 'UK', 'FR' etc. -->
<xsd:simpleType name="def_iso-3166">
   <xsd:restriction base="xsd:string">
       <xsd:pattern value="[A-Z]{2}"/>
   </xsd:restriction>
</xsd:simpleType>
<!-- Two-letter country code followed by a 3 letter subdivision -->
<xsd:simpleType name="def_iso-3166-2">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="[A-Z]{2}\-[A-Z]{2}"/>
    </xsd:restriction>
</xsd:simpleType>
<!-- The union of the previous simple types -->
<xsd:simpleType name="countryCodes">
   <xsd:union memberTypes="def_iso-3166 def_iso-3166-2"/>
</xsd:simpleType>
```

2.2.6. Registering an XML Schema in Oracle XML DB

An XML Schema must be registered with the Oracle XML DB before it can make use of it. To register an XML Schema, the user should call PL/SQL procedure `DBMS_XMLSCHEMA.register_schema`, like shown in the following example that loads the file containing the MusicBrainz schema from the examples directory:

```
BEGIN
  DBMS_XMLSCHEMA.registerSchema(
    SCHEMAURL => 'http://musicbrainz.org/ns/mmd-2.0#',
    SCHEMADOC => bfilename('examplesDir',
    'musicbrainz_mmd-2.0.xsd') );
END;
/
```

Note

This code is available as file 2.5.txt.

When XML schemas are registered with Oracle XML DB, a set of default tables are created and used to store XML instance documents associated with the schemas. These documents can be viewed and accessed in Oracle XML DB Repository.

XMLType is a native server datatype that lets the database understand that a column or table contains XML. This is similar to the way that date and timestamp datatypes let the database understand that a column contains a date. Datatype XMLType also provides methods that allow common operations such as XML schema validation.

XMLType tables or columns can be constrained and conform to an XML schema. This has several advantages:

- The database will ensure that only XML documents that validate against the XML schema can be stored in the column or table.
- Since the contents of the table or column conform to a known XML structure, Oracle XML DB can use the information contained in the XML schema to provide more intelligent query and update processing of the XML.
- Constraining the XMLType to an XML schema provides the option of storing the content of the document using structured-storage techniques.

Structured storage decomposes the content of the XML document and stores it as a set of SQL objects rather than simply storing the document in an unstructured way, such as text in a CLOB.

The main reason for using XML schemas with Oracle XML DB is to validate that instance documents conform to a given XML Schema. The XMLType datatype methods `isSchemaValid()` and `schemaValidate()` allow Oracle XML DB to validate the contents of an instance document stored in an XMLType.

The XML schema is registered under a URL. This URL is meant to identify the XML Schema internally. Oracle XML DB does not require access to the target of the URL when registering an XML Schema or when validating documents that conform to the schema. It is assumed that any instance document associated with the XML schema will provide the URL used to register the XML schema as the way to identify the schema.

2.3. XQuery

In the context of this module, the easiest way to introduce XQuery might be to say that it is to XML what SQL is to relational databases. Like SQL, XQuery is a query language but in this case it lets you define queries or complex traversals on collections of XML data and return the pieces of those XML documents that meet the query conditions.

XQuery is also a declarative language, so it is independent from the way XML data are traversed or where they are stored. Consequently, the same XQuery can work with different XML sources, like a XML file or a relational database that features a mechanism to provide an XML view of its records.

However, despite similarities between XQuery and SQL, the data model that supports XQuery is very different from the relational data model on which SQL is based. XML includes concepts like hierarchy and data order that are not present in the relational model.

In XQuery, the order in which data appear is important and decisive, because it is not the same to look for a tag <B> in an <A> tag, like <A><B></A>, than to look for all <B> tags anywhere in the document. XQuery has been built on the basis of XPath, which is a declarative language for the location of nodes and pieces of information in XML trees. XQuery is based on this language for information selection and iteration through an XML-based data set.

Bibliography

For a detailed description of XPath:

Kay, M. (2004). *XPath 2.0 programmer's reference*. John Wiley and Sons.

2.3.1. XPath

XPath is a standard for accessing and obtaining data from XML documents. It takes into account that they are structured hierarchically as a tree. By defining paths across the tree, it allows identifying specific parts of an XML document. It includes:

- A syntax for defining the parts of an XML document.
- A set of expressions that select parts of an XML document.
- A set of standard functions for manipulating strings, numbers, dates etc.

In general, depending on what it selects, an XPath expression can return:

- A sequence of nodes
- A boolean value, true or false
- A number
- A string of characters

Node

A node can be an element node, an attribute node, a text node, or any other of the Node Types. More definitions are available from the glossary included at the end of this module.

Path Expressions

A path expression locates nodes inside the hierarchical structure of an XML document. Each expression includes one or more steps across the tree, each one connected with '/'.

```
step1/step2/...
```

If the path expression starts with '/', then its evaluation starts from the root element of the XML document.

```
/step1/step2/...
```

Each step might include one axe, one node test and one or more predicates (axes, node tests and predicates are described in the next sections):

```
axe::nodetest[predicate1][predicate2]...
```

Steps are evaluated in relation to the set of nodes produced by the previous step in the expression, the one on the left of the current one. For instance, the following expression selects, from the specified XML document, the `text` content of the `name` elements inside `artist` elements that are `children` of the `metadata` root element:

```
$doc/child::mmd:metadata/child::mmd:artist/
child::mmd:name/child::text()
```

It is important to note that for all the XPath and XQuery examples in this section, it is necessary to define both the `mmd` namespace (this is where all the MusicBrainz schema elements are defined) and the `$doc` variable, which points to the XML document we are working with. Consequently, all the examples should start with the following two lines:

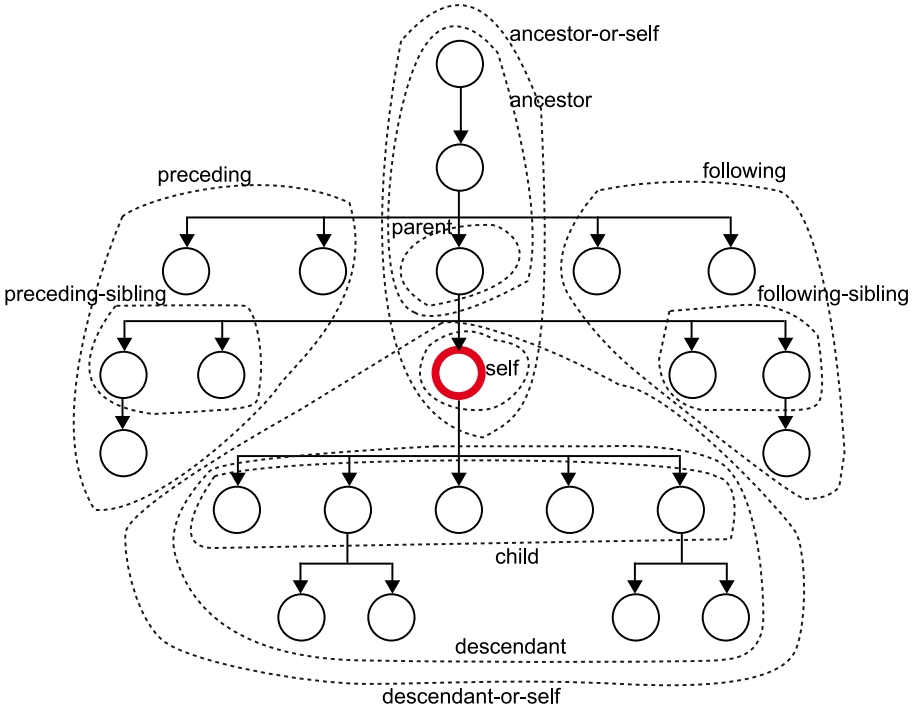
```
declare namespace mmd="http://musicbrainz.org/ns/mmd-2.0#";
declare variable $doc := doc("http://musicbrainz.org/ws/2/artist/
cC2c9c3c-b7bc-4b8b-84d8-4fbd8779e493?inc=release-groups+releases");
```

Axes

The axes in a path expression step specify the direction in which the evaluation is going to proceed, this might be up or down in the hierarchy, whether it is going to include the current node or not etc. The axes are presented next and their application illustrated in figure 4 (except for attribute and namespace nodes and axes):

- **ancestor**: selects all ancestors (parent, grandparent etc.) of the current node.
- **ancestor-or-self**: selects all ancestors (parent, grandparent etc.) of the current node and the current node itself.
- **attribute**: selects all attribute nodes of the current node.
- **child**: selects all children of the current node.
- **descendant**: selects all descendants (children, grandchildren etc.) of the current node.
- **descendant-or-self**: selects all descendants (children, grandchildren etc.) of the current node and the current node itself.
- **following**: selects everything in the document after the closing tag of the current node.
- **following-sibling**: selects all siblings after the current node.
- **namespace**: selects all namespace nodes of the current node.
- **parent**: selects the parent of the current node.
- **preceding**: selects everything in the document that is before the start tag of the current node.
- **preceding-sibling**: selects all siblings before the current node.
- **self**: selects the current node.

Figure 5. Application of axes in relation to the current node, the one selected by self



The more common axes can be abbreviated or, in some cases, omitted:

- **child**: this is the default axis so it can be omitted.
  For instance, to get just the `text` contained in all `name` elements inside an `artist` node inside the `metadata` root element of the specified XML documents, it is possible to just build the expression:

```
$doc/mmd:metadata/mmd:artist/mmd:name/text()
```

- **attribute**: it can be abbreviated as '@'.
  For instance, to get the attribute named `type` for all `artists` in the XML document:

```
$doc/mmd:metadata/mmd:artist/@type
```

- **self::node()**: is equivalent to a point ('.').

- **parent::node()**: can be replaced with two points ('..').

- **descendant-or-self::node()**: is equivalent to '//'.
  For instance, to get the attribute named `count` for the parent node of any `release` element, wherever in the XML document:

```
$doc//mmd:release/../@count
```

Node Tests

These test are used to include or exclude nodes selected by an axe. The result after applying a node test is a subset of the nodes selected by the axe, those that satisfy the test. The available tests are:

- **node-name**: where node-name is the actual name of the nodes to be selected. For instance `release-group` will select all nodes names like that.
- **node**(): matches any node. It can be abbreviated using '*', for instance `child::*`.
- **text**(): matches any text node.
- **comment**(): matches any comment node.
- **element**(): matches any element node.
- **attribute**(): matches any attribute node.
- **attribute(price)**: matches any attribute whose name is price.

Predicates

A predicate further restricts the set of nodes selected by the combination of an axe and a node-test it is attached to. It sets the conditions that should be evaluated true by the set of nodes selected by the axe and the node-test. The predicates are included in the expression between '[' and ']'. In order to build the logical expressions in predicates, it is possible to combine axes and node-test with operators and functions defined by the XPath specification. Table 5 presents a subset of the operators and functions provided by XPath.

Web recommended

For and exhaustive list of operators refer to the specification: World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition) W3C Recommendation, 14 December 2010.

http://www.w3.org/TR/xpath-functions

Table 5. A subset of the operator and functions defined by XPath

Operator	Description	Example
|	Computes two node-sets	//release | //release-group
+	Addition	6 + 4
-	Subtraction	6 – 4
*	Multiplication	6 * 4
div	Division	8 div 4
=	Equal	count = 9
!=	Not equal	count != 9
<	Less than	count < 9
<=	Less than or equal to	count <= 9
>	Greater than	count > 9
>=	Greater than or equal to	count >= 9
or	Or	count < 8 or count > 10
and	And	count > 8 and count < 10

Operator	Description	Example
mod	Modulus (division remainder)	5 mod 2

For instance, the following two expressions restrict the set of nodes selected by the axes plus node-tests to just those with the attribute `type` valued Album or the attribute `count` valued greater than 10.

```
$doc//mmd:release-group[@type="Album"]
$doc//mmd:release-list[@count>10]
```

2.3.2. Queries

A query in XQuery is an expression that reads a sequence of data as XML and returns as a result another sequence of data in XML. In XQuery, expressions and returned values are dependent on the context. For example, the nodes of the result depend on the namespaces, the position inside the document when evaluating the query expression, etc.

XQuery queries are composed of five different kinds of clauses that are called **FLWOR** (pronounced flower) that stands for **for**, **let**, **where**, **order by** and **return**. Each FLWOR expression is generating a tuple stream:

tuple stream

A tuple stream is an ordered sequence of zero or more tuples. Each tuple is a set of zero or more named variables, each of which is bound to a value. More definitions are available from the glossary at the end of this module.

- **for**: it links one or more variables to expressions written in XPath, creating a flow of tuples (rows) where each tuple contains the values for all the variables defined by the `for`.
- **let**: it binds a variable to the result of an expression, which might involve variables defined by a `for`. There might be more than one `let` in an XQuery. The variable values are added to the tuples generated by a `for` clause or, if there is no `for` clause, creating a unique tuple containing these links.
- **where**: it filters the tuples by removing all those that do not meet the conditions.
- **order by**: it sorts the tuple stream according to the given criterion.
- **return**: it is evaluated for each tuple in previously filtered and reordered stream. It builds the result XML of the query by concatenating the results of all return evaluations. It can combine XML code with XPath expressions, the latter are surrounded with '{' and '}' if mixed with XML.

The XQuery syntax defines that each query must have at least one `for` or `let` clause. The query may optionally have `where` and `order by` clauses, but it must end with a `return`.

Simple For Expression Example

In the following `for` clause example, the variable `$rg` will go through all the `release-group` elements in the XML document.

```
 for $rg in $doc//mmd:release-group
return <release-group id="{$rg/@id}"/>
```

Note

This XQuery is available as file 3.2.1.txt.

Query details:

- The `for` clause iterates over all `release-group` elements in `adele-releasegroups.xml`, loaded in Section 1.4, binding variable `$rg` to the value of each such element, in turn. That is, it iterates over `release-group` elements, binding `$rg` to each `release-group`, independently of where they appear in the XML. The result at this stage of query evaluation is:

```
<release-group   type='Single'   id='29716e9a-4496-4d3f-8570-
  2833001cdd9e'>
   <title>Cold Shoulder</title>
   <first-release-date>2008-04-21</first-release-date>
</release-group>
<release-group   type='Live'   id='36e41dc0-2a0c-4ff7-b043-
  097534d52bf6'>
   <title>Adele Live at the Royal Albert Hall</title>
   <first-release-date>2011-11-28</first-release-date>
</release-group>
...
<release-group type='Album' id='e4174758-d333-4a8e-a31f-
  dd0edd51518e'>
   <title>21</title>
   <first-release-date>2011-01-19</first-release-date>
</release-group>
```

- The `return` clause constructs `release-group` elements, one for each tuple. Attribute `id` of these elements is constructed using attribute `id` from the input, resulting in the following output from the query:

```
<release-group id='29716e9a-4496-4d3f-8570-2833001cdd9e'/>
<release-group id='36e41dc0-2a0c-4ff7-b043-097534d52bf6'/>
...
<release-group id='e4174758-d333-4a8e-a31f-dd0edd51518e'/>
```

Conditional Expression Example

The following query returns the titles of the Adele release groups that were first released during the second half of 2011 ordered by title.

```
for $rg in $doc//mmd:release-group
where $rg/mmd:first-release-date>="2011-07-01" and
   $rg/mmd:first-release-date<"2012-01-01"
order by $rg/mmd:title
return $rg/mmd:title
```

Note

This XQuery is available as file 3.2.2.txt.

Query details:

- The for clause iterates over all release-group elements in adele-releasegroups.xml, loaded in Section 1.4, binding variable $rg to the value of each such element, in turn and independently of where they appear in the XML. The results is:

```
<release-group   type='Single'   id='29716e9a-4496-4d3f-8570-
2833001cdd9e'>
   <title>Cold Shoulder</title>
   <first-release-date>2008-04-21</first-release-date>
</release-group>
<release-group   type='Live'   id='36e41dc0-2a0c-4ff7-b043-
  097534d52bf6'>
   <title>Adele Live at the Royal Albert Hall</title>
   <first-release-date>2011-11-28</first-release-date>
</release-group>
...
<release-group   type='Album'   id='e4174758-d333-4a8e-a31f-
  dd0edd51518e'>
   <title>21</title>
   <first-release-date>2011-01-19</first-release-date>
</release-group>
```

- The `where` clause filters the tuple stream of release groups, keeping only tuples with a subelement `first-release-date` greater or equal than 2011-07-01 and smaller than 2012-01-01. The result after filtering is:

```
<release-group type='Live' id='36e41dc0-2a0c-4ff7-b043-
097534d52bf6'>
   <title>Adele Live at the Royal Albert Hall</title>
   <first-release-date>2011-11-28</first-release-date>
</release-group>
<release-group   type='Live'   id='763f800f-4284-432b-b056-
  7f6e0aa26bfe'>
   <title>iTunes Festival: London 2011</title>
   <first-release-date>2011-07-13</first-release-date>
</release-group>
<release-group   type='Single'   id='7C2071cb-598d-4a0c-b1d5-
  a53e2cb9b5f8'>
   <title>Set Fire to the Rain</title>
   <first-release-date>2011-07-04</first-release-date>
</release-group>
```

- The `order by` clause sorts the filtered tuple stream by the value of the subelement `title` in ascending order, the default, and upper-case before lower-case. The previous output is consequently ordered resuting in:

```
<release-group type='Live' id='36e41dc0-2a0c-4ff7-b043-
097534d52bf6'>
   <title>Adele Live at the Royal Albert Hall</title>
   <first-release-date>2011-11-28</first-release-date>
</release-group>
<release-group   type='Single'   id='7C2071cb-598d-4a0c-b1d5-
  a53e2cb9b5f8'>
   <title>Set Fire to the Rain</title>
   <first-release-date>2011-07-04</first-release-date>
</release-group>
<release-group   type='Live'   id='763f800f-4284-432b-b056-
  7f6e0aa26bfe'>
   <title>iTunes Festival: London 2011</title>
   <first-release-date>2011-07-13</first-release-date>
</release-group>
```

- Finally, the `return` clause concatenates at the output the `title` elements for all the filtered tuples and in the appropriate order. The final output for the query is:

```
<title>Adele Live at the Royal Albert Hall</title>
<title>Set Fire to the Rain</title>
<title>iTunes Festival: London 2011</title>
```

Differences Between For and Let Clauses

To first view `for` and `let` clauses may seem equal but, although their aim is in both cases to link variables to values, they do so differently. For the query:

```
 for $t in $doc//mmd:release-group/mmd:title
 return <titles>{$t}</titles>
```

Note

This XQuery is available as file 3.2.3a.txt.

The `for` clause assigns each of the release title nodes that appear anywhere in the XML document to the `$t` variable. The result is repeated as many times as `title` elements so the result is:

```
<titles><title>Cold Shoulder</title></titles>
<titles><title>Adele Live at the Royal Albert Hall</title>
</titles>
...
<titles><title>Data on the Web</title></titles>
```

On the other hand, the `let` clause binds a variable to the whole results of evaluating an expression. Consequently, if we replace the `for` clause with a `let` one in the previous query:

```
let $t:=$doc//mmd:release-group/mmd:title
return <titles>{$t}</titles>
```

Note

This XQuery is available as file 3.2.3b.txt.

```
The result obtained is:
<titles>
<title>Cold Shoulder</title>
<title>Adele Live at the Royal Albert Hall</title>
...
<title>Data on the Web</title>
</titles>
```

In this case, the `$t` variable is linked only once to all titles of all release groups so the `<titles>` tag appears just once.

Combining For and Let

If a `let` clause appears in a query that already has one or more `for` clauses, the values of the variable bound by the `let` clause are added to each of the rows generated by the `for` clause.

For instance, the following query returns the titles of the release groups that have more than seven releases with the same title than the release group. Release group, in MusicBrainz terms, means the list of all her singles, albums and live recordings.

```
for $rg in $doc//mmd:release-group
let $r:=//mmd:release[mmd:title=$rg/mmd:title]
let $c:=count($r)
where $c>7
return
   <release-count count="{$c}">
      {$rg/mmd:title/text()}
   </release-count>
```

Note

This XQuery is available as file 3.2.4a.txt.

Query details:

- The `for` clause iterates over all `release-group` elements in adele-releasegroups.xml, loaded in Section 1.4, binding variable `$rg` to the value of each such element, in turn and independently of where they appear in the XML. The result at this stage of query evaluation is:

```
<release-group   type='Single'   id='29716e9a-4496-4d3f-8570-2833001cdd9e'>
   <title>Cold Shoulder</title>
   <first-release-date>2008-04-21</first-release-date>
</release-group>
...
<release-group   type='Album'   id='e4174758-d333-4a8e-a31f-dd0edd51518e'>
   <title>21</title>
   <first-release-date>2011-01-19</first-release-date>
</release-group>
```

- The `let` clause binds variable `$r` to the sequence of all of the releases whose title is equal to the title of the release group associated for that tuple to `$rg` (this is a join operation). Note that, unlike `for`, `let` does not iterate over values, `$r` is bound once per `$rg` value. Consequently, for each `release-group` in the output there is a set of release, as shown in the next table:

<release-group type='Single' id='29716e9a-4496-4d3f-8570-2833001cdd9e'> <title>Cold Shoulder</title> <first-release-date>2008-04-21</first-release-date> </release-group>	<release id="907aaa31-d6d1-3831-9993-886c70c20c1d"> <title>Cold Shoulder</title> <status>Official</status> <quality>normal</quality> <date>2008-04-21</date> <country>GB</country> <barcode>634904035877</barcode> </release> <release id="99c27905-350e-4ab2-998a-7e087ad122cf"> <title>Cold Shoulder</title> <status>Promotion</status> <quality>normal</quality> <date>2008-04-21</date> <country>GB</country> </release> <release id="a48d3be1-915a-42c8-bed9-0b9d4407b28d"> <title>Cold Shoulder</title> <status>Official</status> <quality>normal</quality> <date>2008-04-21</date> <country>GB</country> <barcode>634904035822</barcode> </release>
...	...
<release-group type='Album' id='e4174758-d333-4a8e-a31f-dd0edd51518e'> <title>21</title> <first-release-date>2011-01-19</first-release-date> </release-group>	

Together, `for` and `let` produce a stream of tuples (`$rg`, `$r`), where `$rg` represents a `release-group` and `$r` represents all of the releases with the same title than `$rg`, for each `$rg`.

- The `where` clause filters this tuple stream, keeping only tuples with `$r` having 7 or more elements.

- Finally, the `return` clause concatenates the output for each `return` clause, which combine XML code with XPath expression to mix it with the releases count and the release title. Just the release group for the album titled 19 has more than 7 releases with the same title, so the output from the previous query is:

```
<release-count count="8">19</release-count>
```

If the query contains more than one `for` clause or `for` clauses with more than one variable, the result is the Cartesian product of the involved variables. For instance, the following query explores all the combinations of release groups and releases, though just those whose titles do not match are picked.

The result contains all the combinations of titles among release groups and releases that do not have the same title:

```
for $rg in $doc//mmd:release-group, $r in $doc//mmd:release
where $rg/mmd:title != $r/mmd:title
return <different>{$rg/mmd:title, $r/mmd:title}</different>
```

```
<different>
  <title>Cold Shoulder</title>
  <title>Adele Live at the Royal Albert Hall</title>
</different>
<different>
  <title>Cold Shoulder</title>
  <title>Adele Live at the Royal Albert Hall</title>
</different>
...
<different>
  <title>21</title>
  <title>Make You Feel My Love</title>
</different>
<different>
  <title>21</title>
  <title>Chasing Pavements</title>
</different>
```

Note

This XQuery is available as file 3.2.4b.txt.

Additional Conditional Expressions

In addition to the `where` clause, XQuery also supports conditional expression of the form if-then-else with the same behavior than similar expressions in most common programming languages.

The where part in FLWOR expressions allows filtering the rows that will appear in the result, whereas a conditional expression allows creating alternative outputs depending on how the boolean part of the expression, the if, is evaluated.

For instance, it is possible to process all release groups and generate one kind of output if they are albums and another one if they are not.

```
for $rg in $doc//mmd:release-group
return
if ($rg/@type = "Album") then
<album>{$rg/mmd:title/text()}</album>
else
<other>{$rg/mmd:title/text()}</other>
```

Note

This XQuery is available as file 3.2.5.txt.

We get the following result:

```
<other>Cold Shoulder</other>
<other>Adele Live at the Royal Albert Hall</other>
...
<album>21</album>
```

Existential Quantifiers

XQuery supports two existential quantifiers: `some` and `every`. These quantifiers allow defining queries that filter just the tuples for which the stated condition is met for all the specified nodes, this is the case for `every`, or that is met by at least one of the specified nodes, this is the case for `some`.

For instance, with every it is possible to define a XQuery that selects those release groups whose releases with the same title are all with status set to `Official`.

```
for $rg in $doc//mmd:release-group
where every $r in $doc//mmd:release[mmd:title=$rg/mmd:title]
satisfies ($r/mmd:status = "Official")
return <all-official>{$rg/mmd:title/text()}</all-official>
```

Note

This XQuery is available as file 3.2.6a.txt.

We get the following result:

```
<all-official>Adele Live at the Royal Albert Hall</all-official>
<all-official>2011-02-25: Morning Becomes Eclectic, KCRW-FM, Santa
Monica, CA, USA</all-official>
...
<all-official>21</all-official>
```

Alternatively, we can broaden the query to select those release groups for which there is at least one release with the same title and status set to `Official`.

```
for $rg in $doc//mmd:release-group
where some $r in $doc//mmd:release[mmd:title=$rg/mmd:title]
satisfies ($r/mmd:status = "Official")
return <some-official>{$rg/mmd:title/text()}</some-official>
```

Note

This XQuery is available as file 3.2.6b.txt.

```
We get the following result:
<some-official>Cold Shoulder</some-official>
<some-official>Make You Feel My Love</some-official>
<some-official>Chasing Pavements</some-official>
<some-official>iTunes Live From SoHo</some-official>
<some-official>19</some-official>
<some-official>Hometown Glory</some-official>
```

Operators and Functions

XQuery supports different kinds of operators and functions. The set of standard operations and functions is shared between XPath and XQuery. Consequently, some of them where already introduced together with XPath in Section 3.1.4. Here, a more detailed overview of these operators and functions is provided, though, for the full set, the reference is the corresponding W3C standard document:

Web recommended

For and exhaustive list of operators and functions refer to the specification: World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition) W3C Recommendation, 14 December 2010.

http://www.w3.org/TR/xpath-functions

- Mathematical: +, –, * , div(), mod(),…
- Comparison: =, !=, <, >, <=, >=
- Boolean: not(), true(), false()
- Rounding: round(), floor(), ceiling(),…
- Aggregate functions: count(), min(), max(), avg(), sum()
- String functions: concat(), string-length (), startswith(), ends-with (), substring(), upper-case (), lower-case, string(),…
- Context functions: position(), last(), current-time(),…
- Date, time and duration: duration-equal(), time-equal(), hours-from-duration(), day-from-date(),…
- Sequence: union (|), intersect, except, distinct-values…
- Etc.

For instance, using the operator on sequences `distinct-values`, it is possible to get the list of all the releases titles without repeated values.

```
for $t in
   distinct-values($doc//mmd:release/mmd:title/text())
return <distinct-title>{$t}</distinct-title>
```

Note

This XQuery is available as file 3.2.7a.txt.

We get the following result:

```
<distinct-title>19</distinct-title>
<distinct-title>Hometown Glory</distinct-title>
...
<distinct-title>iTunes Live From SoHo</distinct-title>
```

Another interesting function is `except`, which allows getting a release node with all its children nodes except for the barcode node. In this case, the `$r/@*` construct retrieves all attributes of the original release node to pass them to the new output `release-without-barcode` node. On the other hand, `$r/*` retrieves the set of all child nodes of the original release node, from where the barcode node is removed using `except`.

```
for $r in $doc//mmd:release
return
   <release-without-barcode>
       {$r/@* }
       {$r/* except $r/mmd:barcode}
   </release-without-barcode>
```

Note

This XQuery is available as file 3.2.7b.txt.

The previous XQuery, for each release node, obtains first all its attributes and then all the child nodes except for the barcode one. The result of this query is like this:

```
<release-without-barcode
       id='1596501c-e332-366d-9ad5-b1923bab1005'>
    <title>19</title>
    <status>Official</status>
    <quality>high</quality>
    <text-representation>
       <language>eng</language>
       <script>Latn</script>
    </text-representation>
    <date>2008-11-17</date>
    <country>FR</country>
</release-without-barcode>
```

2.3.3. Comments

The comments in XQuery, unlike in XML, are enclosed between smiling faces, as it is shown below.

```
(: this is a comment :)
```

2.3.4. XQuery in Oracle XML DB

When performing XPath queries in Oracle, on XML stored using an unstructured approach, queries are evaluated by parsing the XML documents from CLOBs, BLOBs, BFILEs or VARCHARs. This can be very expensive when performing operations on large collections of documents. For structured storage, XPath operations may be evaluated using XPath rewrite, leading to significantly improved performance, particularly with large collections of documents.

Oracle XML DB can rewrite SQL statements that contain XPath expressions to purely relational SQL statements, which can be processed more efficiently. In this way, the rewrite insulates the database optimizer from having to understand XPath/XQuery and the XML data model. The database optimizer simply processes the rewritten SQL statement in the same manner as other SQL statements.

This means that the database optimizer can derive an execution plan based on conventional relational algebra. Consequently, there is little overhead and Oracle XML DB can execute XPath-based queries at near-relational speed, while preserving the XML abstraction. In certain cases, the rewrite is not possible because there is no SQL equivalent of parts of the XQuery. In this situation Oracle XML DB performs a functional evaluation like in the case of unstructured storage.

In general, functional evaluation of a SQL statement is more expensive than XPath rewrite, particularly if the number of documents that needs to be processed is large. However the major advantage of functional evaluation is that it is always possible, regardless of the complexity of the XPath expression.

This is an example of an XMLQuery performed on a XML document previously loaded into Oracle XML DB. The evaluation is conduced differently depending on how the document was stored. If it was stored in an unstructured way, it is parsed and then the XQuery is evaluated using a functional XQuery interpreter or evaluation engine, which itself has been compiled into the database. Otherwise, an XQuery rewrite is conduced. The procedure is transparent to the user, who will never need to change the code in any way to take advantage of available XQuery optimizations.

```
SELECT XMLQuery('
for $rg in doc("adele-releasegroups.xml")//release-group
return <release-group id="{$rg/@id}"/>
   'RETURNING CONTENT) FROM DUAL;
```

Note

This code is available as file 3.4.txt.

Summary

In this module we have introduced two relational extensions; object-relational and XML. Each of these extensions tackles a precise problem but in essence, both pursue the same objective: extend the relational model to other areas where the relational model does not naturally fit in.

On the one hand, we have seen that the object-relational data model succeeded in its confrontation, back in the 80s, with the object-oriented data model. After a long debate, major relational software vendors tip the balance and, nowadays, most relational databases follow this paradigm, whereas most OODBSs have been discontinued.

However, it was not until SQL-99 that object-oriented features were included in the standard. Unfortunately, this release was not only late, but also short in terms of features. Many of the propositions claimed in the third generation database systems manifesto have never been included in the standard.

Current software products, alternatively, have developed their own object-oriented data model. All of them substantially differ from each other, but they keep a common feature: the object-relational layer is built on top of the underlying relational engine. About features provided, we can mostly talk about user-defined types, object tables, inheritance, REFs and support to collections. In this module, we have introduced them in terms of Oracle.

All in all, the object-oriented layer is an interesting extension that can help to bridge the gap between the relational model and the object-oriented paradigm most HLLs follow.

On the other hand, the XML extension is based on a tree data structure that is serialized using the XML syntax. This syntax is based on tags, with open and close tags, which capture the tree structure through their nesting and each one corresponds to a tree node. Consequently, as trees have just one root node, XML documents start with an open tag and the corresponding close tag is at the end of the document, thus enclosing all the rest of the tags.

Tags can contain other tags or text, which also correspond to nodes of the tree. The other main components of XML documents are attributes. They are name-value pairs that are attached to tags and thus to non-textual tree nodes. A XML document that follows the XML syntax rules is said to be well formed.

However, XML on its own is of little help because it does not put any constraints on how tags, text and attributes are combined, neither on which tag or attribute names to use. The XML Schema language allows creating domain

specific vocabularies defining tag and attribute names, how they are combined, their allowd values etc. so any XML-based tool can easily check if a XML document based on a schema conforms to its constraints and can be thus considered valid.

XML Schema also provides reuse mechanisms by extension or restriction of existing schema elements. This is combined with namespaces, which define naming contexts that avoid the ambiguities of tags with the same name, thus facilitating the reuse of existing XML Schemas at a global scale.

A set of XML documents can be queried using XQuery, the XML query language standard. It allows retrieving data from XML documents and also generating output XML documents meeting the requirements for those queries. An XQuery is based on the FLWOR structure, where each letter stands for one of the query parts:

- **For:** links variables to expressions written in XPath and creates a tuple stream where each tuple is composed by variable values.
- **Let:** binds a variable to the result of an expression.
- **Where:** filters the tuples by removing all those that do not meet the conditions.
- **Order by:** sorts the tuple stream according to the given criterion.
- **Return:** builds the output XML of the query, combining XML templates and variable values.

XPath is the XML standard to build paths across XML documents to select the relevant parts from them. Its syntax defines the steps to follow to find the tags, attributes or values to be retrieved.

Self-evaluation

Object-relational

1. Give two reasons why object methods may suit better than using attributes.

2. Enumerate two pros and cons of choosing OIDs instead of PKs in Oracle.

3. Consider the following relational tables:

```
CREATE TABLE airport (
 IATA CHAR(3) PRIMARY KEY,
 city VARCHAR2(25) NOT NULL,
 country VARCHAR2(20) NOT NULL,
 region VARCHAR2(14) NOT NULL
);

CREATE TABLE passenger (
 id CHAR(10) PRIMARY KEY,
 EUcitizen CHAR NOT NULL CHECK (EUcitizen = 'Y' OR EUcitizen = 'N'),
 membership VARCHAR2(9) NOT NULL CHECK (membership IN ('none',
 'frequent', 'business', 'gold', 'vip'))
);

CREATE TABLE ticket (
 IATA1 CHAR(3) CONSTRAINT ticket_FK_origin REFERENCES airport,
 IATA2 CHAR(3) CONSTRAINT ticket_FK_destination REFERENCES airport,
 id CHAR(10) CONSTRAINT ticket_FK_passenger REFERENCES passenger,
 price NUMBER(9,2) NOT NULL,
 discount INTEGER NOT NULL,
 PRIMARY KEY (IATA1, IATA2, id)
);
```

And the following (already available) type:

```
CREATE TYPE travel AS OBJECT (
 departureAirport CHAR(3),
 destinationAirport CHAR(3),
 idPassenger CHAR(10),
 passengerMembership VARCHAR2(9),
 price NUMBER(9,2),
 discount INTEGER);
```

Create an object view to allow external applications to access your relational data according to the type travel.

4. Suppose you are asked to design a database to contain the following data: for each school in Barcelona, its name, code (provided by the Generalitat) and list of students. For each student, in turn, we want to know their name, current course and average mark.

You know that your database will be accessed by an application defining the following classes:

```
CLASS centre
   Name VARCHAR2(20)
   Code INTEGER
   List of students

CLASS student
   Name VARCHAR2(30),
   Surname1 VARCHAR2(30),
   Surname2 VARCHAR2(30),
   Course VARCHAR2(30),
   AverageMark INTEGER
```

You are asked to:

Create the UDTs to map both classes but:

- First, think of a solution where the list of students is described as a VARRAY.
- Then, provide a solution where the list of students is a NESTED TABLE.

- Briefly discuss which solution would be better for this scenario.

XML

We recommend you now to try to develop the XQueries for the following exercises. You can test your solutions using different XQuery tools before looking at the solutions we propose for them in the next section. By testing your solutions before looking at the proposed one, you can get a better idea of the effects of your changes and get a deeper understanding of how XQuery works.

The input XML for these exercises is the same XML document from MusicBrainz than the one used along this module. To test your solutions, you can use Oracle XML DB, as detailed in the corresponding sections, or use an online service that allows you to load the XML input and perform your queries interactively.

XML document from MusicBrainz

The XML document for Adele containing all her releases and groups of releases is available from: http://musicbrainz.org/ws/2/artist/cc2c9c3c-b7bc-4b8b-84d8-4fbd8779e493?inc=release-groups+releases

For instance, the XQuery Demo is a web page that allows testing XQueries and defining a custom input XML document against which the queries will operate. The custom XML document is defined using the "Load a context document input", where the URL pointing to the MusicBrainz XML document should be placed. Then, the XQuery text area can be used to write the XQueries.

In any case, whatever the XQuery tool you use, do not forget to define the `$doc` variable pointing to the input XML document and the MusicBrainz XML Schema namespace:

```
declare namespace mmd="http://musicbrainz.org/ns/mmd-2.0#";
declare variable $doc :=
doc("http://musicbrainz.org/ws/2/artist/cc2c9c3c-b7bc-4b8b-84d8-
4fbd8779e493?inc=release-groups+releases");
```

XQuery Demo

The XQuery Demo web page is available at: http://www.semwebtech.org/xquery-demo/

This is the list of proposed exercises:

5. Modify the part of the MusicBrainz schema related to the `release-group` element so it also incorporates a mandatory `quality-list` subelement that point to a list of qualities, based on the already defined quality simple type. The `release-group` element and `quality` definitions are included here for convenience.

```
<xsd:element name="release-group">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element minOccurs="0" ref="mmd-2.0:title"/>
        <xsd:element minOccurs="0" ref="mmd-2.0:disambiguation"/>
        <xsd:element minOccurs="0" ref="mmd-2.0:comment"/>
        <xsd:element minOccurs="0" ref="mmd-2.0:first-release-date"/>
        <xsd:element minOccurs="0" ref="mmd-2.0:artist-credit"/>
        <xsd:element minOccurs="0" ref="mmd-2.0:release-list"/>
        <xsd:element minOccurs="0" maxOccurs="unbounded"
   ref="mmd-2.0:relation-list"/>
        <xsd:element minOccurs="0" ref="mmd-2.0:tag-list"/>
        <xsd:element minOccurs="0" ref="mmd-2.0:user-tag-list"/>
        <xsd:element minOccurs="0" ref="mmd-2.0:rating"/>
        <xsd:element minOccurs="0" ref="mmd-2.0:user-rating"/>
        <xsd:group ref="mmd-2.0:def_release-group-element_extension"/>
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:anyURI"/>
     <xsd:attribute name="type" type="xsd:anyURI"/>
      <xsd:attributeGroup ref="mmd-2.0:def_release-group-
   attribute_extension"/>
    </xsd:complexType>
  </xsd:element>

<xsd:simpleType name="def_quality">
    <xsd:restriction base="xsd:token">
      <xsd:enumeration value="low"/>
      <xsd:enumeration value="normal"/>
      <xsd:enumeration value="high"/>
    </xsd:restriction>
  </xsd:simpleType>
```

6. List the release title and release date for those releases dated before 2008.

7. Get the amount of releases for each country, taking into account the country where each release has happened, but just considering those releases before 2008-06-01. The output

should be a list of `country` tags with an attribute `name` and a `text` subelement with the number of releases in that country, sorted in descending order by `number`.

8. List all release groups that have an empty `first-release-date`.

9. Get all the release groups that have a `first-release-date` that do not follow the YYYY-MM-DD format.

10. Fix all release groups that have a date that does not follow the YYYY-MM-DD format. If the format is YYYY, make it the first day of the corresponding year using the YYYY-MM-DD format. If the `first-release-date` does not follow the YYYY format, remove the element completely.

Answer key

Object-relational

1. Object methods can be used to replace derived attributes (i.e., attributes whose value is derived from other attributes). In such scenarios:

- object methods provide up-to-date data (if the base attributes upon which the derived attribute is built change we must refresh the derived attribute and while it is not done, we would access and old value),
- we save space, as we do not materialize this data but compute it on-the-fly whenever the method is called.

```
These exercises can be found in the exs_solution_3 and exs_solution_4 files, respectively.
```

2. We can spot out the following main differences between OIDs and PKs:

- The primary key has an ad hoc size depending on the built-in type used to implement it. For example, an integer is usually 4 bytes long. On the contrary, OIDs are always 16 bytes long. This can affect performance (for example, when building indexes).
- The primary key provides semantics regarding the scenario. However, an OID is an artificial identifier, with no semantics at all.
- The OID is guaranteed to be unique. The primary key, however, could just be unique in terms of our database.

3. The object-view must fill all the ticket attributes properly. In this case, the solution would look like this:

```
CREATE VIEW travels OF travel WITH OBJECT IDENTIFIER (departureAirport, destinationAirport,
idPassenger ID) AS
   SELECT t.IATA1, t.IATA2, t.id, p.membership, t.price, t.discount
   FROM ticket t, passenger p
   WHERE t.id = p.id;
```

4. Solution with VARRAY:

```
CREATE TYPE student AS OBJECT (
Name VARCHAR2(30),
Surname1 VARCHAR2(30),
Surname2 VARCHAR2(30),
Course VARCHAR2(30),
AverageMark INTEGER);

CREATE TYPE list_of_students AS VARRAY(5000) OF student;

CREATE TYPE centre AS OBJECT (
   Name VARCHAR2(20),
   Code INTEGER,
   Students list_of_students);
```

Solution with NESTED TABLE:

```
CREATE TYPE student AS OBJECT (
Name VARCHAR2(30),
Surname1 VARCHAR2(30),
Surname2 VARCHAR2(30),
Course VARCHAR2(30),
AverageMark INTEGER);

CREATE TYPE students_t AS TABLE OF student;

CREATE TABLE centre AS OBJECT (
   Name VARCHAR2(20),
   Code INTEGER,
   Students students_t)
   NESTED TABLE Students STORE AS students_nt;
```

Discussion:

According to table 4 (discussed in section “Practical Issues on Collections”), nested tables do not require specifying the number of elements beforehand and in this case, the variation between different centers might be too high. Furthermore, we will mostly access students

according to which center they belong to (i.e., we mostly will partially access the collection) and nested tables also allow higher concurrency. Furthermore, with nested tables we can easily answer questions of the kind: count the number of students with an average mark higher than X per center and it is also easier to declare CHECKs, if needed.

XML

These are the proposed solutions for the previous exercises. Note that each solution is just one among many different ways to solve the same problem.

```
The solutions for these exercises are available as files Solution2a.txt, Solution2b.txt,
Solution3.txt, Solution4a.txt, Solution4b.txt, Solution4c.txt, Solution5.txt and Solution6.txt.
```

5. The release group element is modified to add a new entry for the `quality-list` element. This is a just a reference because the element is defined below, in a separate definition. The reference does not include any occurrence attribute because the default is a minimum cardinality of one. The added line is highlighted in bold.

```
<xsd:element name="release-group">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element minOccurs="0" ref="mmd-2.0:title"/>
        <xsd:element minOccurs="0" ref="mmd-2.0:disambiguation"/>
        <xsd:element minOccurs="0" ref="mmd-2.0:comment"/>
        <xsd:element minOccurs="0" ref="mmd-2.0:first-release-date"/>
        <xsd:element minOccurs="0" ref="mmd-2.0:artist-credit"/>
        <xsd:element ref="mmd-2.0:release-list"/>
        <xsd:element minOccurs="0" ref="mmd-2.0:release-list"/>
        <xsd:element minOccurs="0" maxOccurs="unbounded"
      ref="mmd-2.0:relation-list"/>
        <xsd:element minOccurs="0" ref="mmd-2.0:tag-list"/>
        <xsd:element minOccurs="0" ref="mmd-2.0:user-tag-list"/>
        <xsd:element minOccurs="0" ref="mmd-2.0:rating"/>
        <xsd:element minOccurs="0" ref="mmd-2.0:user-rating"/>
        <xsd:group ref="mmd-2.0:def_release-group-element_extension"/>
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:anyURI"/>
      <xsd:attribute name="type" type="xsd:anyURI"/>
      <xsd:attributeGroup ref="mmd-2.0:def_release-group attribute_extension"/>
      </xsd:complexType>
    </xsd:element>
```

The previous addition is just a reference to the `quality-list` element. It is defined in its own, on top of the existing quality simple type that it reuses.

```
    <xsd:element name="quality-list">
    <xsd:simpleType>
       <xsd:list itemType="mmd-2.0:def_quality"/>
    </xsd:simpleType>
    </xsd:element>
```

6.

```
    for $r in $doc//mmd:release
    where $r/mmd:date<"2008"
    order by $r/mmd:date
    return <release>{$r/mmd:title,$r/mmd:date}</release>
```

The proposed solution uses 2008 instead of 2008-01-01 because there are some entries in the input XML data that do not follow the YYYY-MM-DD format for dates. This causes that when filtering for dates before 2008-01-01, releases that have just the year 2008 are also included in the output, as shown below.

```
    for $r in $doc//mmd:release
    where $r/mmd:date>"2008-01-01"
    order by $r/mmd:date
    return $r/mmd:title
```

If 2008-01-01 is used for filtering in the where clause, the output is the following, which includes the releases that just specify the year 2008 as their date.

```
<release>
   <title>Hometown Glory</title>
   <date>2007-10-29</date> </release>
<release>
   <title>2008-09-22: BBC Radio 1's Live Lounge: London, UK </title>
   <date>2008</date>
</release>
<release>
   <title>Hometown Glory</title>
   <date>2008</date>
</release>
```

7.

```
for $c in distinct-values($doc//mmd:country)
let $rc := $doc//mmd:release[mmd:country=$c and mmd:date<
   "2008-06-01"]
let $nc := count($rc)
where $nc > 0
order by $nc descending
return <country name="{$c}">{$nc}</country>
```

The query has a pair of particularities. First of all, the use of the `distinct values` function in the `for` clause to avoid as many occurrences of the same country in the output as its appearances in the input. The other one is the `where` clause. It is necessary to avoid that countries without any release appear in the output. The output of the query is:

```
<country name="GB">12</country>
<country name="FR">1</country>
<country name="JP">1</country>
<country name="DE">1</country>
```

8.

```
for $rg in $doc//mmd:release-group
where not(exists($rg/mmd:first-release-date/text()))
return $rg
```

The same result can be obtained omitting the `exists` function, because the `not` operator also operates on sequences being the empty one equivalent to false.

```
for $rg in $doc//mmd:release-group
where not($rg/mmd:first-release-date/text())
return $rg
```

It is even possible to reduce the query to just the XPath expression.

```
$doc//mmd:release-group[not(mmd:first-release-date/text())]
```

In all cases we obtain the same result.

```
<release-group type="Live" id="37367c39-1c91-4a8c-baa7-0a09c3df4b6b">
   <title>2011-02-25: Morning Becomes Eclectic, KCRW-FM,
     Santa Monica, CA, USA</title>
   <first-release-date/>
</release-group>
```

9.

```
for $rg in $doc//mmd:release-group
where not(matches($rg/mmd:first-release-date,"\d{4}-\d{2}-\d{2}"))
return $rg
```

The proposed solution uses a regular expression matching function called `matches`. It matches the value to be checked against the provided regular expression. In this case, the regular expression is four digits followed by a hyphen ('-') followed by two digits another hyphen ('-') character and two final digits. The result from the `matches` function is negated and the result includes release groups whose first release date does not follow this format, which includes those that do not have a value for this element.

```
<release-group type="Live" id="11ce3c93-0325-439e-8de7-fab397ba839c">
   <title>2008-09-22: BBC Radio 1's Live Lounge: London, UK </title>
   <first-release-date>2008</first-release-date>
```

```
</release-group>

<release-group type="Live" id="37367c39-1c91-4a8c-baa7-0a09c3df4b6b">
   <title>2011-02-25: Morning Becomes Eclectic, KCRW-FM,
     Santa Monica, CA, USA</title>
   <first-release-date/>
</release-group>
```

10.

```
for $rg in $doc//mmd:release-group
where not(matches($rg/mmd:first-release-date,"\d{4}-\d{2}-\d{2}"))
return
<mmd:release-group>
    {$rg/@* }
   {$rg/* except $rg/mmd:first-release-date}
   {if ($rg/mmd:first-release-date/text() and matches())
     then <mmd:first-release-date>
             {$rg/mmd:first-release-date/text()}-01-01
      </mmd:first-release-date>
     else ()}
</mmd:release-group>
```

The proposed solution is built on the proposed one for Exercise 4. For each of the releases that do not follow the YYYY-MM-DD format, all the attributes and subelements are copied in the output except for the `first-release-date` one.

For that particular case an if-then-else conditional is defined. If the first release date follows the YYYY format, the first-release-date element is generated and its text subelement is the result of concatenating the year and the string 01-01. This way, the output date is the first day of the corresponding year. Otherwise, the `else` part of the conditional clause, which is mandatory, returns the empty sequence. Consequently, there is no `first-release-date` element in the output because it was previously omitted using the `except` operator and it has not been generated in the if-then-else conditional.

The result of this query is:

```
<mmd:release-group type="Live" id="11ce3c93-0325-439e-8de7-fab397ba839c">
   <mmd:title>2008-09-22: BBC Radio 1's Live Lounge:
      London, UK </mmd:title>
   <mmd:first-release-date>2008-01-01</mmd:first-release-date>
</mmd:release-group>

<mmd:release-group type="Live" id="37367c39-1c91-4a8c-baa7-0a09c3df4b6b">
   <mmd:title>2011-02-25: Morning Becomes Eclectic, KCRW-FM,
      Santa Monica, CA, USA</mmd:title>
</mmd:release-group>
```

Glossary

API Application Programming Interface.

Attribute A characteristic or property of an element. Attributes are represented as name value pairs on an element tag.

DBA Database Administrator.

DBMS Database Management System.

Document Object Model (DOM) An API that provides an object representation of an XML document. The DOM API represents an XML document as a tree of nodes. Nodes may be created, queried, updated and deleted.

Document Type Definition (DTD) Describes the structure of XML documents.

Document An XML structure containing a root element and its subelements.

Element A component of the tree structure defined in a Document Type Definition (DTD) or Schema. An element may be composed of text, attributes and other elements.

Final In Object-Oriented programming languages, a final class is that which cannot be extended.

Instantiable In Object-Oriented programming languages, a class which can be instantiated and, therefore, creates an object of that kind.

Item An item is either an atomic Value or a Node.

JDBC Java Database Connectivity.

Metadata Metadata are data about data or data that describe other data.

Namespace A feature of XML for using multiple vocabularies in a single XML document and avoid name clashes.

Node Type The types of nodes are, in addition to Element, Attribute and Text, Document, DocumentFragment, DocumentType, ProcessingInstruction, EntityReference, CDATASection, Comment, Entity and Notation.

Node A node can be an element node, an attribute node, a text node or any other of the Node Types.

ODBC Open Database Connectivity.

OO Object-Oriented.

OODBS Object-Oriented Database System.

OODM Object-Oriented Data Model.

OOPL Object-Oriented Programming Language.

ORDBS Object-Relational Database System.

ORDM Object-Relational Data Model.

Parser A tool that reads XML data and breaks it up into elements and attributes, usually structured as a Document Object Model (DOM).

PL/SQL Procedural Language/Structured Query Language.

Root The outermost element in an XML document that contains all other elements. It is the top node in a tree structure.

Schema Defines the structure of XML documents. Schemas address deficiencies in DTDs such as specifying data types.

Sequence A sequence is an ordered collection of zero or more Items.

Shred The process of mapping the data in an XML document to table rows and columns in a relational database.

Surrogate An artificial primary key created and maintained by the system with no semantics extracted from the domain.

Tag The markup language used to describe an XML element. An XML tag is represented by the element name enclosed by angle brackets.

Tuple Stream A tuple stream is an ordered sequence of zero or more tuples.

Tuple It is a set of zero or more named variables, each of which is bound to a Value.

Value In the XQuery data model, a value is always a Sequence.

Vertical Fragmentation The problem of breaking a relation into smaller pieces by grouping attributes.

Vocabulary A dialect or set of XML tags used to describe a particular data structure. A vocabulary is defined using a DTD or Schema.

Bibliography

Object-Oriented

Atkinson M.; Bancilhon F.; DeWitt D.; Dittrich K.; Maier D.; Zdonik S. (1990). *The Object-Oriented Database System Manifesto*. In Proc. 1st Int. Conf. on Deductive and Object-Oriented Databases, North Holland.

Stonebraker M.; Rowe L.; Lindsay B.; Gray J.; Carey M.; Brodie M.; Bernstein P.; Beech D. (1990). "Third generation database system manifesto".*ACM SIGMOD* Rec., 19(3).

Rowe L.; Stonebraker M. (1987). *The Postgres Data Model*. In Proc. 13th Int. Conf. on Very Large Data Bases.

Dietrich S.W.; Urban S.D. (2005). *An Advanced Course in Database Systems: Beyond Relational Databases*. Prentice Hall, Upper Saddle River, NJ.

Liu, L.; Ozsu, M. T. (Eds.) (2009). *Encyclopedia of Database Systems*. Springer.

XML

Ray, E.T. (2003). *Learning XML* (2nd Edition). Sebastopol, CA: O'Reilly Media.

Harold, E. R.; Jeans, W. S. (2004). *XML in a Nutshell*, 3rd Edition. Sebastopol, CA: O'Reilly Media.

Hunter, D.; Rafter, J.; Fawcett, J.; Vlist, E. van der; Ayers, D.; Duckett, J.; Watt, A., et al. (2007). Beginning XML, 4th Edition. Indianapolis, IN: Wrox.

Vlist, E. van der. (2002). *XML Schema: The W3C's Object-Oriented Descriptions for XML*. Sebastopol, CA: O'Reilly Media.

Walmsley, P. (2002). *Definitive XML Schema*. Upper Saddle River, NJ: Prentice Hall.

Walmsley, P. (2007). *XQuery*. Sebastopol, CA: O'Reilly Media.

Brundage, M. (2004). *XQuery: the XML query language*. Boston, MA: Addison-Wesley Professional.

Additional references available on-line

Object Database Management Systems: The Resource Portal for Education and Research, http://odbms.org

http://www.mulberrytech.com/quickref/XMLquickref.pdf