



UNIVERSIDAD CATÓLICA DEL NORTE  
FACULTAD DE INGENIERÍA Y CIENCIAS GEOLÓGICAS  
DEPARTAMENTO DE INGENIERÍA DE SISTEMAS Y  
COMPUTACIÓN

**GUÍA PARA LA SELECCIÓN DEL TIPO DE BASE DE DATOS  
EVALUANDO EL MODELO RELACIONAL, MODELO OBJETO  
RELACIONAL Y DATOS SEMI ESTRUCTURADOS EN XML**

Memoria para optar al Título de Ingeniero de Ejecución en  
Computación e Informática

**Ingeborg Andrea Muñoz Carnot**

Profesores Guía: **Luis Lobos Flores**  
**Loreto Telgie Bendek**

Antofagasta, Chile  
Enero de 2014

# Agradecimientos

Al finalizar este periodo de mi vida no queda más que agradecer a todas las personas que ayudaron a que esto fuera posible.

Primero quisiera agradecer a mi madre, que sin su apoyo incondicional no hubiera podido lograr esta tan anhelada meta. Gracias por confiar en mí hasta el final, eres mi ejemplo, eres mi luz. Además desearía agradecer a mi padre, muchas gracias por brindarme de tu sabiduría, por tu apoyo, por alentarme a lo largo de mi vida. Asimismo quisiera agradecer a mi abuelita, quien siempre ha sido un pilar en mi vida, su total seguridad en mí me ha dado constantemente fuerzas para continuar.

Finalmente y no menos importante, me gustaría dar gracias a mis profesores guía Loreto Telgie y Luis Lobos, que por su gran apoyo, enseñanza y confianza se pudo realizar con éxito este trabajo de título. Gracias por guiarme, darme la mano una vez más y profundizar mi pasión por el área de bases de datos.

# Índice General

	Página
Índice de Tablas . . . . .	V
Índice de Figuras . . . . .	VI
Resumen . . . . .	VIII
<b>I INTRODUCCIÓN . . . . .</b>	<b>1</b>
1.1 Conceptos básicos . . . . .	1
1.2 Problema a resolver . . . . .	3
1.3 Justificación . . . . .	4
1.4 Objetivos . . . . .	5
1.5 Descripción . . . . .	6
1.6 Resultados esperados . . . . .	7
1.7 Metodología . . . . .	7
1.8 Recursos necesarios . . . . .	9
<b>II MARCO TEÓRICO . . . . .</b>	<b>11</b>
2.1 La evolución del modelado de bases de datos . . . . .	11
2.2 Datos semi estructurados . . . . .	27
2.3 Algunos motores con soporte objeto relacional y XML . . . . .	41
2.4 Oracle . . . . .	57
<b>III CASO DE ESTUDIO . . . . .</b>	<b>87</b>
3.1 Descripción del Caso de Estudio . . . . .	87
3.2 Diagrama entidad relacionamiento . . . . .	92
3.3 Modelo relacional . . . . .	93
3.4 Notación UML para el modelo entidad relacionamiento . . . . .	96
3.5 Implementación modelo objeto relacional . . . . .	99

3.6	Implementación de modelo de datos semi estructurado . . . . .	118
3.7	Comparación de resultados . . . . .	127
<b>IV</b>	<b>CONCLUSIONES . . . . .</b>	<b>136</b>
4.1	Recomendaciones en la elección del tipo de base de datos para una aplicación particular . . . . .	136
4.2	Cumplimiento de los objetivos . . . . .	138
4.3	Contratiempos en la planificación . . . . .	142
4.4	Experiencia personal . . . . .	142
4.5	Trabajo futuro . . . . .	143
	<b>Bibliografía . . . . .</b>	<b>144</b>
	<b>Anexo A Detalles, sintaxis y ejemplos de algunos motores de base de datos con soporte objeto relacional . . . . .</b>	<b>148</b>
A.1	PostgreSQL . . . . .	148
A.2	SQLServer . . . . .	156
A.3	DB2 . . . . .	163
A.4	Oracle . . . . .	167
	<b>Anexo B Documentación caso de estudio . . . . .</b>	<b>193</b>
B.1	Documentación DER . . . . .	193
B.2	Documentación modelo relacional . . . . .	205
B.3	Código SQL de creación de tablas del modelo relacional . . . . .	218
B.4	Mapeo de caso de estudio de entidad relacionamiento / UML a la capa objeto relacional . . . . .	229
B.5	Mapeo de caso de estudio de la capa objeto relacional a la capa de persistencia . . . . .	240
B.6	Detalles de implementación de nuevo requerimiento utilizando XML	242
	<b>Anexo C Desempeño de consultas SQL relacionales y objeto-relacionales en Oracle . . . . .</b>	<b>244</b>
	<b>Anexo D Mapeo objeto relacional (ORM) . . . . .</b>	<b>249</b>
D.1	Tipos de mapeadores . . . . .	250
D.2	Frameworks de persistencia . . . . .	251

# Índice de Tablas

	<b>Página</b>
Tabla II-1 Tipos colección: Varray versus Nested Table . . . . .	60
Tabla III-1 Reglas de transformación a Modelo Objeto relacional . . . . .	103
Tabla III-2 Ventajas y desventajas del modelo relacional . . . . .	133
Tabla III-3 Ventajas y desventajas del modelo objeto relacional . . . . .	134
Tabla III-4 Ventajas y desventajas de XML/SQL . . . . .	135
Tabla A-1 Valores que puede tener el tipo base en SQLServer . . . . .	157
Tabla A-2 Valores del parámetro flags de OPENXML en SQLServer . . .	161
Tabla A-3 Tabla empleado con datos de ejemplo . . . . .	166
Tabla A-4 Comparación de los diferentes modelos de almacenamiento .	191
Tabla D-1 Herramientas ORM populares en la actualidad . . . . .	252

# Índice de Figuras

	<b>Página</b>
Figura 1.1 Entorno de un sistema de bases de datos simplificado . . . .	3
Figura 2.1 La evolución de las técnicas de modelos de bases de datos .	13
Figura 2.2 Relaciones modelo jerárquico de bases de datos . . . . .	15
Figura 2.3 Modelo de red de base de datos . . . . .	17
Figura 2.4 Modelo relacional . . . . .	18
Figura 2.5 Base de datos que almacena estudiantes y cursos . . . . .	20
Figura 2.6 Tercera generación de bases de datos . . . . .	21
Figura 2.7 Modelo de bases de datos orientado a objetos . . . . .	22
Figura 2.8 Ejemplo de objeto . . . . .	23
Figura 2.9 Tipos de datos internos de DB2 soportados . . . . .	54
Figura 2.10 Estructura de un tipo de objeto . . . . .	62
Figura 2.11 Ejemplo de herencia . . . . .	77
Figura 2.12 Diagrama de flujo de decisión para almacenar XML en Oracle	86
Figura 3.1 Diagrama entidad relación . . . . .	92
Figura 3.2 Diagrama UML del modelo relacional . . . . .	98
Figura 3.3 Entidad Personal con atributo compuesto . . . . .	100
Figura 3.4 Ejemplo objetos relacionamiento N : N . . . . .	102
Figura 3.5 Ejemplo objeto intersección relacionamiento N : N . . . . .	102
Figura 3.6 Entidad Personal con atributo multivaluado . . . . .	104
Figura 3.7 Entidad Personal con atributo derivado . . . . .	105
Figura 3.8 Relacionamiento 1 a N . . . . .	106
Figura 3.9 Relacionamiento N a N . . . . .	108
Figura 3.10 Relacionamiento N a N con atributos . . . . .	110
Figura 3.11 Entidad débil . . . . .	112
Figura 3.12 Generalización de entidades . . . . .	114

Figura 3.13 Agregaciones . . . . .	115
Figura 3.14 Resultado de consulta a columna XMLTYPE (con tags) . . .	123
Figura 3.15 Resultado a columna XMLTYPE nodos . . . . .	124
Figura 3.16 Resultado a columna XMLTYPE después UPDATE . . . . .	126
Figura 3.17 Resultado a columna XMLTYPE antes UPDATE nodos . . .	126
Figura 3.18 Resultado a columna XMLTYPE después UPDATE nodos . .	127
Figura 3.19 Resultado a columna XMLTYPE después DELETE nodos . .	127
Figura 3.20 Resultado de consulta a listado de celulares . . . . .	129
Figura 3.21 Cantidad tablas modelo relacional . . . . .	131
Figura 3.22 Cantidad tablas modelo objeto relacional . . . . .	132
Figura 3.23 Tabla resumen de criterios . . . . .	133
Figura A.1 Ejemplo estructura de esquema XML (Documento XSD) . . .	184
Figura A.2 Ejemplo XPath 1 . . . . .	188
Figura A.3 Ejemplo XPath 2 . . . . .	188
Figura A.4 Ejemplo XPath 3 . . . . .	189
Figura C.1 Tasa de bloques leídos para GROUP BY sencillo . . . . .	245
Figura C.2 Tasa de lecturas de disco para GROUP BY sencillo . . . . .	246
Figura C.3 Tasa de bloques leídos para GROUP BY HAVING y ORDER BY . . . . .	246
Figura C.4 Tasa de lecturas de disco en GROUP BY HAVING y ORDER BY . . . . .	247
Figura C.5 Tasa de bloques leídos para las subconsultas con EXISTS .	248
Figura C.6 Tasa de bloques leídos para las subconsultas con la cláusula IN . . . . .	248

# Resumen

El presente trabajo de título aborda la evaluación del sistema de administración de base de datos Oracle 11g, con el objetivo de investigar las capacidades que este posee. Para ello se diseñará un caso de estudio que luego será implementado en una base de datos relacional, base de datos objeto relacional y el manejo de datos semi estructurados como XML, con el propósito de obtener como resultado un documento que indique las ventajas y desventajas de cada implementación considerando criterios como facilidad de modelamiento y facilidad de aprendizaje por parte del desarrollador.

En el presente informe se hablará de los distintos conceptos nombrados como lo son bases de datos relacionales, bases de datos objeto relacional y datos semi estructurados como XML. Dentro de la introducción se justificará el propósito de la realización de esta investigación, los objetivos más relevantes, una descripción detallada del trabajo a realizar, junto a los resultados esperados de dicho trabajo, especificando la metodología de investigación. Los recursos tanto como hardware, software y personas involucradas. Se detallará a fondo un marco teórico de cada tema antes mencionado, además se mencionará brevemente algunos motores de bases de datos con capacidades objeto relacional y sus ejemplos correspondientes. Por último dentro del marco teórico se especificará en profundidad las capacidades que posee el motor Oracle, detallando tanto teoría como ejemplos de implementación.



En el caso de estudio se presentará el problema a desarrollar con sus respectivos diagramas de entidad relacionamiento y UML, también se propondrá un algoritmo para transformar desde el modelo relacional al objeto relacional y se especificará cada paso de la implementación, las dificultades a la que se vio enfrentada la implementación del modelo objeto relacional y las soluciones propuestas. Se incluirá un nuevo requerimiento al caso de estudio para implementar los datos semi estructurados en el modelo objeto relacional ya implementado, se detallará cada paso de este proceso. Finalmente se realizará una comparación de los resultados obtenidos según los criterios definidos y se presentarán las ventajas y desventajas de cada implementación. Por último en las conclusiones se darán las recomendaciones para la elección del tipo de base de datos según la aplicación a desarrollar, el cumplimiento de los objetivos del trabajo de titulación, todos los contratiempos que afectaron la planificación inicial del trabajo, la experiencia personal por parte de la memorista y el trabajo a futuro.

# **CAPÍTULO I**

## **INTRODUCCIÓN**

El presente capítulo tiene por objetivo entregar una pequeña base teórica, en lo que respecta a los conceptos básicos de bases de datos, además de entregar los detalles del problema a resolver, justificación de la realización de la investigación, objetivo general y específico(s), descripción del problema, los resultados esperados, metodología que se aplicará en la investigación y los recursos a utilizar como hardware, software y humanos.

### **1.1. Conceptos básicos**

Las bases de datos y la tecnología de bases de datos tienen mucha culpa del uso creciente de los computadores. Es justo decir que las bases de datos juegan un papel fundamental en la mayoría de las áreas en las que se utilizan computadores, como en el sector empresarial, en el comercio electrónico, ingeniería, medicina, justicia, educación y bibliotecas [2].

Una base de datos es un repositorio para datos. En otras palabras, se puede almacenar una gran cantidad de información en éstas. Una base de datos relacional utiliza estructuras llamadas tablas, las cuales están unidas entre sí mediante lo que se denominan relaciones. Se puede construir tablas con relaciones entre las

tablas, no sólo para organizar los datos, sino que también permite una recuperación posterior de la información desde la base de datos [3].

Un sistema de administración de datos (SABD) es un conjunto de programas que posibilita a los usuarios a crear y mantener una base de datos. El SABD es un software que simplifica los procesos de definición, construcción, manipulación y compartición de bases de datos entre varias aplicaciones y usuarios. La construcción de la base de datos es el proceso estable y permanente en almacenar los datos en algún medio de almacenamiento controlado por el SABD. La manipulación de ésta incluye funciones como consultas para recuperar datos específicos, actualización para reflejar los cambios introducidos en el mini mundo y la generación de reportes a través de los datos. Compartir una base de datos posibilita que varios usuarios y programas accedan de forma simultánea [2].

Una aplicación accede a la base de datos enviando solicitudes o consultas de datos al SABD. Una consulta generalmente ocasiona la recuperación de datos; en cambio una transacción puede provocar la escritura o lectura de datos.

La Figura 1.1 muestra algunos de los conceptos que se han explicado, cómo los usuarios realizan consultas a la base de datos y el SABD se encarga de procesar dichas consultas, accediendo a los datos almacenados.

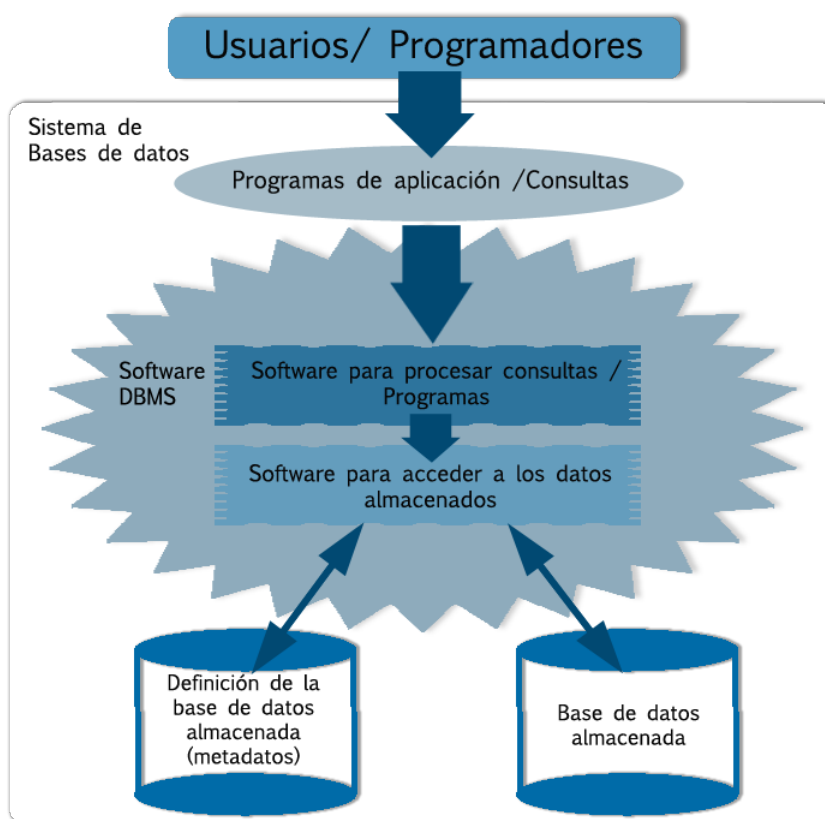


Figura 1.1. Entorno de un sistema de bases de datos simplificado [2]

## 1.2. Problema a resolver

El estudio que se realizará estará centrado en la implementación de un caso de estudio propuesto en una base de datos relacional, una base de datos objeto relacional y el manejo de datos semi estructurados con XML, con el fin de generar una guía para seleccionar el tipo de base de datos más adecuada según la aplicación a desarrollar y de esta forma tener claro las ventajas y desventajas de cada implementación, por lo que se realizarán pruebas en cada una de éstas, con el propósito de analizar los resultados obtenidos.

### 1.3. Justificación

La realización de esta investigación tiene como propósito adquirir la capacidad de examinar los conceptos de bases de datos relacional, bases de datos objeto relacional y datos semi estructurados, sin embargo, implica mucho más que la utilización de dichos conceptos, ya que consiste en poder seleccionar el modelo de datos más apropiado a implementar en un sistema y / o aplicación en particular. Y de esta forma lograr responder la pregunta ¿Por qué este modelo de base de datos es más adecuado para el desarrollo de mi software?

Es sumamente importante para los desarrolladores sacarle provecho a las capacidades que se han ido incorporando a los servidores de base de datos, ya que es relevante modelar según las aptitudes de las bases de datos, puesto que tienen más semántica y los datos no están fragmentados como en una base de datos relacional pura, por lo que no se restringe a la primera forma normal y a causa de esto pueden existir atributos no atómicos.

Es válido acotar que la capa de persistencia de datos del software que se desea desarrollar, puede variar según la implementación seleccionada de la guía a generar mediante los resultados obtenidos en esta investigación.

Cabe mencionar que el obtener resultados efectivos permitirá cooperar con documentación y casos de estudio para la asignatura de base de datos.

A modo personal del tesista, se busca adquirir experiencia y conocimientos de avanzada para el desarrollo profesional en el área de base de datos, específicamente en las áreas relacionadas con los sistemas de administración de base de datos, bases de datos objeto relacional y datos semi estructurados como XML. Así mismo, adquirir un mayor nivel de conocimiento en las capacidades que poseen los servidores de base de datos, como es la implementación de procedimientos almacenados utilizando lenguajes de programación sin tener que imple-

mentar fuera del servidor del sistema de administración de base de datos.

## **1.4. Objetivos**

### **Objetivo general**

Evaluar y comparar la implementación de una base de datos relacional con respecto a una base de datos objeto relacional y datos semi estructurados en el sistema de administración de base de datos Oracle, utilizando las funciones disponibles en el servidor del SABD, con el fin de generar una guía que en base a criterios claros permita determinar cuál es el tipo de base de datos más adecuada a utilizar bajo ciertas características.

### **Objetivos Específicos**

- Estudiar, seleccionar y utilizar funciones del sistema de administración de base de datos relacional Oracle 11G, que permitan la implementación de bases de datos objeto relacional y XML.
- Comprender los múltiples métodos de implementación de los modelos mencionados con anterioridad para ser capaz de seleccionar adecuadamente entre ellos según la aplicación a desarrollar.
- Diseñar el caso de estudio para luego implementarlo con Oracle 11G en una base de datos relacional, objeto relacional y datos semi estructurados como XML.
- Definir criterios de comparación para las implementaciones tales como: facilidad de modelamiento, complejidad de aprendizaje por parte del desarrollador y/o DBA, entre otros.

- Realizar un análisis de resultados, en donde se compare cada una de las implementaciones, con el fin de estructurar una guía con los beneficios y desventajas dependiendo de la aplicación a desarrollar, según los criterios de comparación.

## 1.5. Descripción

El problema a dar solución es generar la capacidad de comprender los múltiples conceptos y métodos de implementación relacionados con bases de datos relacionales, objeto relacional y datos semi estructurados como XML y poder justificar la selección del modelo más adecuado para diferentes aplicaciones. Para lograr este objetivo se pretende implementar dichos modelos sólo utilizando las características nativas presentes en el servidor del SABD Oracle 11G, con el fin de crear una guía para seleccionar el tipo de base de datos más adecuada según la aplicación a desarrollar.

La primera etapa contempla hacer una investigación exhaustiva de los sistemas de administración de base de datos que tengan soporte para implementar bases de datos objeto relacional y datos semi estructurados como XML. A continuación en su segunda etapa, se diseñará un caso de estudio, el cual será implementado en una base de datos relacional en Oracle.

Posteriormente en la tercera etapa, se implementará el caso de estudio en una base de datos objeto relacional y datos semi estructurados como XML. En la cuarta etapa, se realizarán pruebas con una cierta cantidad de datos en cada versión, para luego analizar los resultados obtenidos bajo ciertos criterios que serán definidos. Por último se generará un documento guía con la información recolectada.

## 1.6. Resultados esperados

Se obtendrán tres productos finales:

- **Documento con marco teórico:** El documento de memoria contendrá todo el marco teórico asociado a la creación de una base de datos relacional, base de datos objeto relacional, el manejo de datos semi estructurados como XML desde el punto de vista del intercambio de información.
- **Caso de estudio implementado:** Implementar un caso de estudio en una base de datos relacional, objeto relacional y datos semi estructurados como XML en Oracle.
- **Comparación de implementaciones:** Generar un documento que indique las ventajas y desventajas de cada implementación considerando facilidad de modelamiento, facilidad de aprendizaje por parte del desarrollador, entre otros. Y con esto poder tomar mejores decisiones en cuanto a la elección del tipo de SABD a seleccionar y el manejo de datos semi estructurados dependiendo del tipo de aplicaciones a desarrollar.

## 1.7. Metodología

A continuación se describirá de forma cronológica la metodología que se realizará en esta investigación:

- **Metodología de investigación**

En un inicio para lograr establecer el estado del arte del tema de investigación, se hará una revisión de la literatura sobre los diversos SABD con la capacidad de implementar bases de datos objeto relacional y XML. Para



luego explorar en profundidad los métodos y / o funciones imprescindibles para crear cada implementación en Oracle 11G.

#### ■ **Metodología de diseño de modelo de datos**

Para el diseño del caso de estudio que se implementará en cada base de datos se aplicarán los métodos adquiridos en la asignatura. A continuación se detalla cada paso a realizar:

- Diseñar un caso de estudio.
- Modelar la base de datos relacional y hacer la documentación asociada incluyendo el modelo relacional.
- Modelar la base de datos objeto relacional y documentar utilizando UML.

#### ■ **Metodología de implementación y comparación**

- Implementación del caso de estudio utilizando una base de datos relacional pura.
- Implementación del caso de estudio utilizando una base de datos objeto relacional y datos semi estructurados como XML, disponibles en el SABD Oracle.

Primero se definirán criterios para realizar la comparación, estos pueden ser:

- Facilidad de modelamiento.
- Complejidad de aprendizaje por parte del desarrollador y/o DBA.
- Utilización de recursos.

Procedimientos iniciales para la realización de las comparaciones:

- Ingreso de datos.
- Consulta de datos.
- Eliminación de datos.
- Actualización de datos.

Luego de realizar las comparaciones se procederá a estudiar y analizar los resultados obtenidos para generar la guía, en donde se detallen los beneficios y desventajas de cada implementación de acuerdo al tipo de aplicación que se desea desarrollar.

## 1.8. Recursos necesarios

### Hardware

- Servidores del DISC.
- Laptop personal con las siguientes características.
  - Marca: Dell
  - Modelo: XPS 15
  - Procesador: Intel ® Core ™ i7 CPU Q740 @1.73 GHz
  - Memoria Ram: 8 GB.

### Software

- **Sistema de administración de bases de datos:** Oracle 11g Enterprise Edition.
- **Software de modelado:** Dia.

- **Cliente base de datos:** Oracle Sql\*Plus, Oracle Sql Developer.
- **Sistemas operativos a utilizar:** GNU/Linux / Windows.

### **Recursos Humanos**

- Alumno: Ingeborg Muñoz Carnot.
- Profesor Guía: Lotero Telgie Bendek.
- Profesor Guía: Luis Lobos Flores.

# **CAPÍTULO II**

## **MARCO TEÓRICO**

Este capítulo habla sobre cómo ha evolucionado el modelado de bases de datos a lo largo de la historia y detalla los conceptos de datos semi estructurados y XML. Además presenta algunos motores de bases de datos que poseen soporte objeto relacional, en donde se especifican los métodos principales de utilización de tipos y XML. Se enfatiza en especial al sistema de administración de base de datos Oracle Database 11g, ya que es el motor elegido para implementar el caso de estudio que se definirá, este SGDB posee las ediciones: Oracle Database 11g Express Edition, Standar Edition y Enterprise Edition. La edición a evaluar en la presente memoria será Oracle Database 11g Enterprise Edition, ya que con ésta es posible la gestión de la información de forma eficiente, confiable y segura en aplicaciones transaccionales indispensables, almacenes de datos con elevado tráfico de consultas [1].

### **2.1. La evolución del modelado de bases de datos**

#### **2.1.1. Generalidades**

Los modelos de datos son medios formales para representar los datos asociados a una situación real y para manipular tal representación. Las componentes de

todo modelo de datos son las siguientes:

- Las estructuras básicas son los elementos básicos o tipos de objetos que conforman el modelo.
- Las reglas que es el conjunto de lineamientos que expresan las propiedades estáticas del modelo. Ellas son:
  - Las reglas de formación
  - Las restricciones
- Los operadores que permiten cambiar el estado de una base de datos modificando su contenido. Ellos están asociados a las propiedades dinámicas de los elementos.

Los modelos de datos se pueden clasificar en modelos de alto nivel o semánticos y modelos de bajo nivel o básicos:

- **Los modelos semánticos** capturan un mayor significado de los datos e intentan representar la estructura real de los datos independientemente de las características de almacenamiento, es decir ellos están orientados a las aplicaciones. Existen, hoy en día, numerosos y muy variados modelos semánticos, entre ellos se encuentran: el modelo Entidad-Relación, el modelo Entidad-Relación-Extendido (ERE) y el modelo IFO.
- **Los modelos básicos** constituyen el grupo de modelos que han sido diseñados orientándose al computador, sobre ellos se han desarrollado la mayoría de los SGBD. Ellos son: el modelo jerárquico, el modelo redes, el modelo relacional, el modelo orientado por objetos y el objeto relacional.

Muchos modelos semánticos han sido propuestos, pero pocos de ellos han atraído el interés de los desarrolladores de sistemas de base de datos, esto tal vez es

debido a la complejidad de tales modelos y a su dificultad para ser plasmados con los modelos básicos actuales. La mayoría de los conceptos del modelado semántico de datos han sido muy bien representados en el modelo ERE, el cual goza de gran prestigio y popularidad en el ambiente comercial, jugando un rol muy importante en la mayoría de las herramientas CASE (Computer Aided Software Engineering).

Existieron dos modelos de datos antes del modelo relacional de base de datos, el modelo jerárquico y modelo de red, que fueron soluciones parciales de problemas interminables de como almacenar datos y cómo hacerlo eficientemente. El modelo relacional de base de datos es actualmente la mejor solución tanto para almacenamientos como recuperación de datos.

La evolución de los modelos de base de datos tuvo lugar cuando cada modelo de base de datos mejoró del anterior. La solución inicial no fue una base de datos si no que un sistema de archivos, dependiente del sistema operativo. Se pueden examinar archivos en el sistema de archivos del sistema operativo ejecutando el comando “dir” en DOS, un comando “ls” en UNIX, o buscando a través del Explorador de Windows en Microsoft Windows. El problema que se presenta utilizando un sistema de archivos es que no existe una estructura de base de datos.

La figura 2.1 muestra el proceso de evolución en el tiempo desde los finales de 1940.

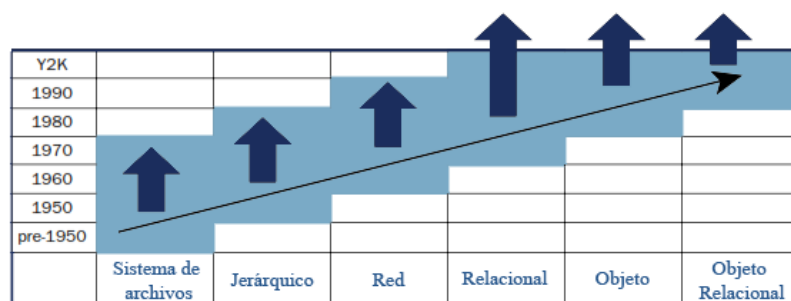


Figura 2.1. La evolución de las técnicas de modelos de bases de datos [3]

### **2.1.2. Sistema de archivos**

El uso de un sistema de archivo como modelo de base de datos implica no aplicar técnicas de modelado y que la base de datos es almacenada en un archivo plano en un sistema de archivos, utilizando la estructura del sistema operativo.

Cualquier búsqueda de datos a través de archivos planos debe ser explícitamente programada. La ventaja de diferentes modelos de bases de datos es que proporcionan algo de esta programación. Para una base de datos de sistema de archivos, los datos pueden ser almacenados en archivos individuales o en múltiples archivos. Similar a buscar a través de archivos planos, cualquier relación y validación entre diferentes archivos planos tendrían que ser programadas y probablemente con una capacidad limitada.

### **2.1.3. Modelo jerárquico de base de datos**

El modelo jerárquico de base de datos fue desarrollado como una solución a las necesidades inmediatas de aplicaciones reales a mediados de los 60s. El sistema más antiguo e importante de base de datos jerárquico es IMS de IBM y fue desarrollado para organizar y almacenar información necesaria para el proyecto de alunizaje del Apolo. IBM y la aviación norteamericana trabajaron juntos para producir la primera versión de IMS en 1968. Las versiones posteriores de IMS fueron diseñadas para usarse con dispositivos de cintas magnéticas, pero los posteriores discos magnéticos se convirtieron en el estándar. IMS rápidamente se convirtió en el sistema de base de datos jerárquico dominante en el mercado y fue por muchos años el DBMS más ampliamente usado, hasta que fue remplazado por los sistemas relacionales. Muchas mejoras fueron hechas a IMS después del 68, resultando en nuevas versiones que obtenían ventajas de las mejoras en hardware y software, proporcionando nuevas características como comunicación

de datos y máximo desempeño. IMS era capaz de procesar grandes cantidades de datos de modo eficiente. Usaba una estructura de árbol familiar para los programadores acostumbrados a trabajar con archivos. La estructura lógica en la que se sustenta es el árbol, el cual se compone de un nodo raíz y varios nodos sucesores, ordenados jerárquicamente, como se observa en la figura 2.2. Cada nodo representa una entidad (tipo de registro) y las relaciones entre entidades son las conexiones entre los nodos. El nodo superior es el nodo padre y los inferiores son los nodos hijos. Las conexiones entre archivos no dependen de la información contenida en ellos, se definen al inicio y son fijos (punteros). Las interrelaciones entre registros permiten que un padre tenga muchos hijos, pero un hijo sólo puede tener un padre. Los datos se representan como estructuras de árbol y el árbol representa la jerarquía de registros de datos. La navegación es top-down.



Figura 2.2. Relaciones modelo jerárquico de bases de datos

Desventajas:

- No modela sencillamente las relaciones Muchos a Muchos.



- Genera anomalías de inserción.
- Genera anomalías de borrado.
- Genera anomalías de actualización.
- Se pueden dar consultas inconsistentes.

#### **2.1.4. Modelo de red de base de datos**

El modelo de base de datos de red es esencialmente un refinamiento del modelo jerárquico. El modelo de red permite a las tablas hijas tener más de un padre, por lo tanto creando a una estructura de tabla como red. Múltiples tablas padres para cada hijo permite relaciones de N a N (muchos es a muchos), en adición a las relaciones de 1 a N (1 a muchos). En un ejemplo de modelo de red mostrado en la figura 2.3, existe una relación N a N entre empleados y tareas. En otras palabras. Un empleado puede ser asignado a muchas tareas, y una tarea puede ser asignada a diferentes empleados. Por lo tanto, muchos empleados tienen muchas tareas y vice versa [3].

La figura 2.3 muestra como los administradores pueden ser parte de departamentos y compañías. En otras palabras, el modelo de red está tomando en cuenta que no sólo cada departamento dentro de una empresa tiene un administrador, sino que también cada compañía tiene un administrador general (en la vida real, un jefe ejecutivo o CEO). La figura 2.3 además muestra la incorporación de tipos de tablas donde los empleados puede ser definidos de diferentes tipos (tal como tiempo completo, part-time, entre otros.). Lo más importante de notar es la nueva tabla “Asignaciones” la cual permite asignar tareas a los empleados. La creación de la tabla “Asignaciones” es el resultado directo de la adición de la capacidad de padres múltiples entre los modelos jerárquicos y de red. Como ya se señaló, la relación entre las tablas empleados y tareas es de N a N, donde cada empleado

puede ser asignado a múltiples tareas y cada tarea puede ser asignada a múltiples empleados. La tabla asignación resuelve el problema de la relación N a N, permitiendo una definición única para la combinación de empleado y tarea. Sin esa definición única, encontrar una asignación sería imposible [3].



Figura 2.3. Modelo de red de base de datos [3]

Desventajas:

- Resulta difícil definir nuevas relaciones.
- Es complicado darle mantenimiento, ya que cualquier cambio en la estructura requiere una descarga de los datos.
- Representa desperdicio de recursos.
- Genera anomalías de inserción.
- Genera anomalías de borrado.

### 2.1.5. Bases de datos relacionales

Las bases de datos relacionales mejoraron en la restricción de la estructura jerárquica, no abandonando completamente la jerarquía de datos, como se muestra en la figura 2.4, en donde un tipo de empleado tiene muchos empleados asociados o dentro de una compañía existen muchos departamentos. Cualquier tabla puede ser accedida directamente sin tener que acceder a todos los objetos padres [3].

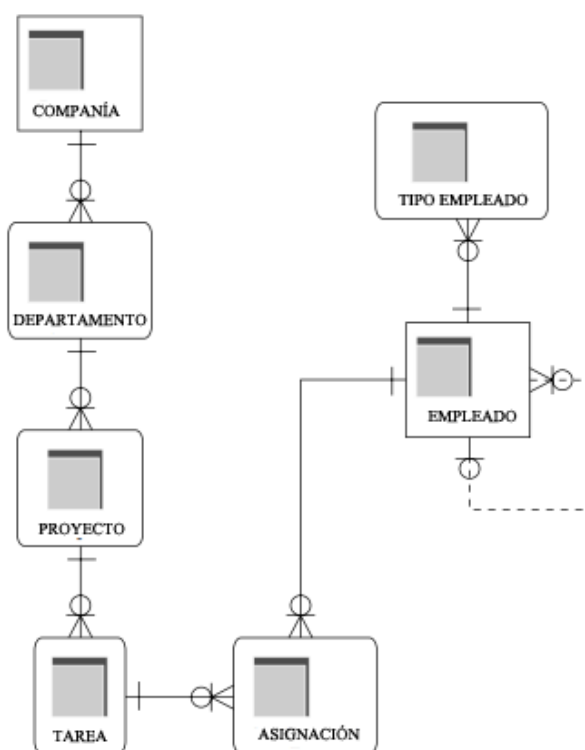


Figura 2.4. Modelo relacional [3]

El modelo de datos relacional introdujo lenguajes de consulta de alto nivel que suministraban una alternativa a las interfaces de lenguaje de programación; por lo que era mucho más rápido escribir consultas nuevas. La representación relacional de los datos se parece al ejemplo presentado en la figura 2.5. Los sistemas

relacionales estaban destinados inicialmente a las mismas aplicaciones que los primitivos sistemas, pero estaban pensados para ofrecer flexibilidad en el desarrollo de nuevas consultas y para reorganizar la base de datos cuando cambiaran los requisitos. Su rendimiento mejoró con el desarrollo de nuevas técnicas de almacenamiento e indexación y unas técnicas mejores de procesamiento y optimización. Eventualmente, las bases de datos relacionales se convirtieron en el tipo de sistema de bases de datos predominante para las aplicaciones de bases de datos tradicionales [2].

Debilidades de los SABD Relacionales:

- **Mal soporte de las transacciones de larga duración:** Suelen ser mucho más comunes para objetos más complejos.
- **Estructura de datos que no permite heterogeneidad:** Este modelo admite homogeneidad vertical y horizontal. Cada tupla posee los mismos atributos. Los valores de una columna tienen que pertenecer al mismo dominio. La intersección de fila y columna debe ser un valor atómico.
- **Pocas facilidades para navegar por los datos:** Acceso asociativo basado en contenido y no basado en el movimiento entre registros individuales.
- **Operaciones restringidas:** Se tienen operaciones orientadas a tuplas, operaciones que proporciona SQL, pero lamentablemente SQL no tolera definir nuevas operaciones.
- **Representación pobre de entidades consideradas del “mundo real”:** La fragmentación de una entidad del “mundo real” en varias relaciones es ineficiente ya que lleva a muchos “joins” en el procesamiento de consultas.
- **Manejo complicado de consultas recursivas:** Es complicado manejar consultas sobre relaciones que una relación tiene consigo misma (directa

o indirectamente).

#### ESTUDIANTE

Nombre	NumEstudiante	Clase	Especialidad
Luis	17	1	CS
Carlos	8	2	CS

#### CURSO

NombreCurso	NumCurso	Horas	Departamento
Introducción a la computación	CC1310	4	CC
Estructuras de datos	CC3320	4	CC
Matemáticas discretas	MAT2410	3	MAT
Bases de datos	CC3380	3	CC

#### SECCIÓN

IDSección	NumCurso	Semestre	Año	Profesor
85	MAT2410	Otoño	04	Pedro
92	CC1310	Otoño	04	Ana
102	CC3320	Primavera	05	Elisa
112	MAT2410	Otoño	05	Antonio
119	CC1310	Otoño	05	Juan
135	CC3380	Otoño	05	Enrique

#### INFORME\_CALIF

NumEstudiante	IDSección	Nota
17	112	B
17	119	C
8	85	A
8	92	A
8	102	B
8	135	A

#### PRERREQUISITO

NumCurso	NumPrerrequisito
CC3380	CC3320
CC3380	MAT2410
CC3320	CC1310

Figura 2.5. Base de datos que almacena la información de estudiantes y cursos [2]

### 2.1.6. Bases de datos orientadas al objeto

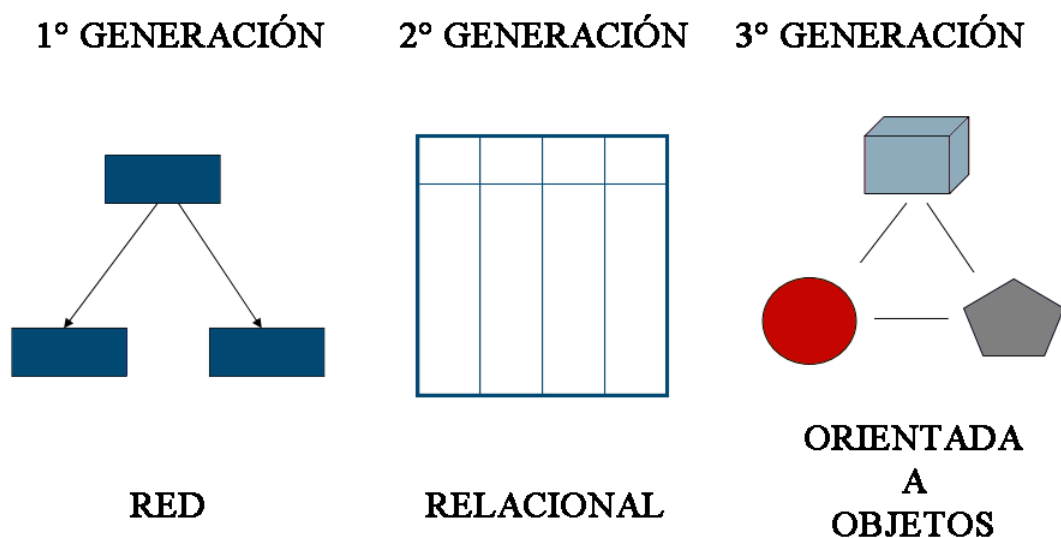


Figura 2.6. Tercera generación de bases de datos

Una base de datos orientada a objetos pertenece a la tercera generación de bases de datos, ver figura 2.6, la cual proporciona a los datos una estructura tri-dimensional, donde cualquier ítem en una base de datos puede ser recuperado desde cualquier punto de una forma rápida. Considerando que el modelo relacional se presta para la recuperación de grupos de registros en dos dimensiones, el modelo orientado al objeto es eficiente para encontrar ítems únicos. Por consiguiente, este modelo se desempeña pobremente cuando se desea recuperar más de un ítem, en lo que el modelo relacional es muy competente.

Este tipo de base de datos resuelve algunas de las complejidades más ocultas de las bases de datos relacionales. La figura 2.7 muestra un ejemplo de la estructura de un modelo orientado a objetos y su equivalencia en el relacional se encuentra en la figura 2.4. La asignación de tareas a los empleados fue manejada utilizando la inclusión de colecciones (listas) en manager, empleado y las especializaciones

de la clase empleado. Además se debe notar que los diferentes tipos de empleados fueron manejados utilizando especialización.

Otro beneficio de este modelo es la habilidad inherente de gestionar y atender aplicaciones y modelos de bases de datos extremadamente complejos. Esto es debido a un principio básico de la metodología de la orientación a objetos, por lo que los elementos altamente complejos pueden descomponerse en sus partes más elementales, lo que permite su acceso explícito, así como también su ejecución a través de esas partes básicas. En otras palabras, si se tiene claro cómo funcionan las pequeñas piezas individualmente, se puede visualizar el problema completo, que es una combinación de un pequeño número de piezas, piezas constituyentes mucho más simples [3].

Una discusión recurrente, dice que uno de los puntos de fricción entre las aplicaciones orientadas al objeto y las bases de datos relacionales es el rendimiento del proceso de mapeo entre los tipos estructurales: Objeto y relacional. La estructura de ambos es completamente diferente.

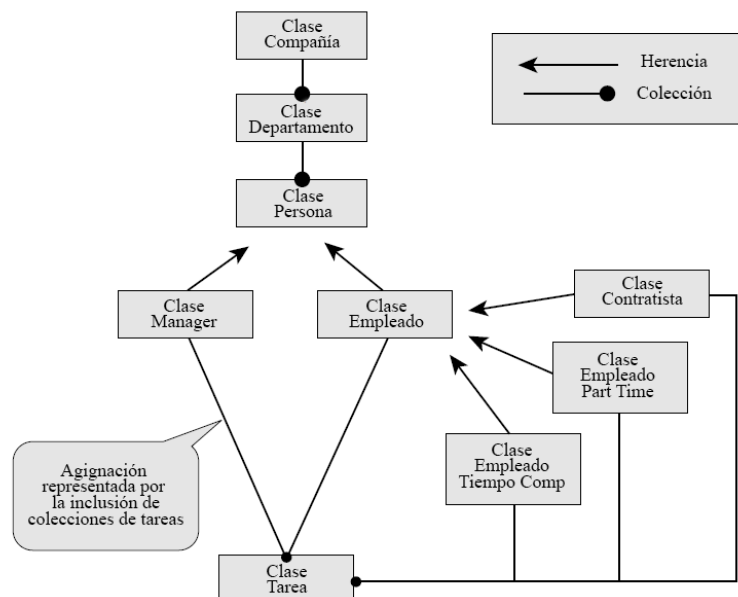


Figura 2.7. Modelo de bases de datos orientado a objetos [3]

## Objeto

Es una entidad del mundo real percibida en el sistema. En palabras técnicas se describe como por sus propiedades, mejor conocidas como *atributos* y los *métodos* que puede facilitar. El estado de un objeto se determina por los valores que posean sus atributos, valores que siempre han de satisfacer las restricciones implantadas sobre ellos. En la figura 2.8 se muestra un ejemplo de objeto. Los métodos definen el comportamiento del objeto.

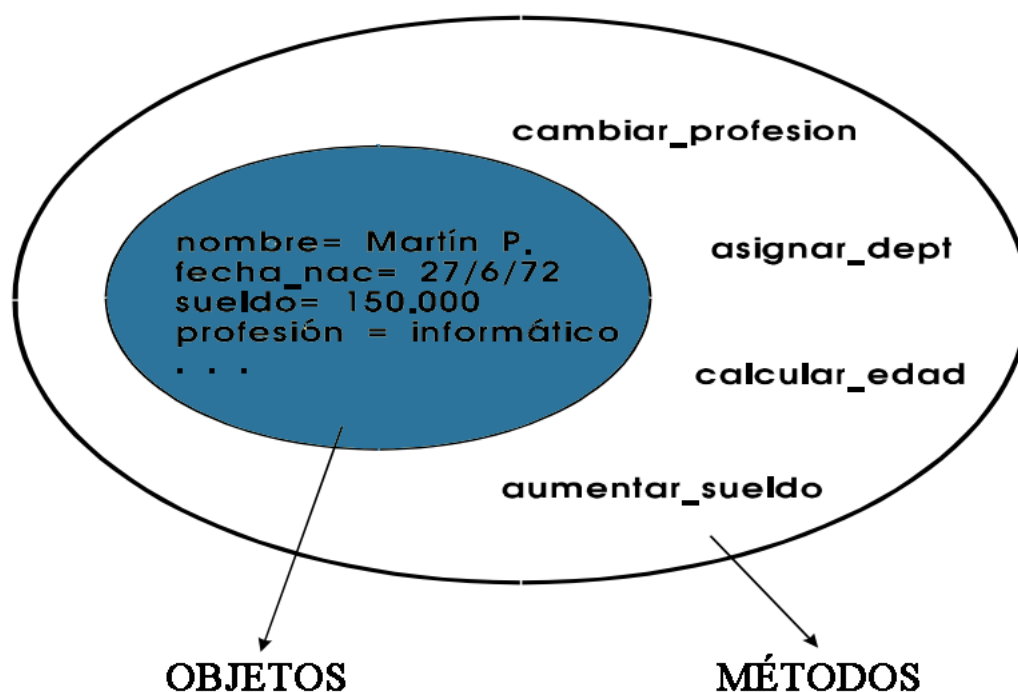


Figura 2.8. Ejemplo de objeto

## Abstracción

Es el proceso que identifica los aspectos primordiales pero ignora el resto. Conlleva centrarse en qué hace y qué es un objeto antes de pensar en cómo implementarlo. Posee dos aspectos elementales:



- **Encapsulación:** Un objeto contiene la estructura de los datos y su conjunto de operaciones las que pueden ser utilizadas para su manipulación.
- **Ocultamiento:** Los aspectos internos de un objeto se encuentran ocultos al exterior. Suministra una forma de independencia de datos.

### 2.1.7. Bases de datos objeto relacional

El modelo de bases de datos objeto relacional fue creado como respuesta a las capacidades conflictivas de los modelos relacionales y orientado a objetos. Esencialmente las capacidades que posee el modelo orientado a objetos ya se encuentran incluidos en las bases de datos relacionales, sin embargo las capacidades de una base de datos relacional no están incorporadas en las bases de datos orientadas al objeto. El modelo objeto relacional intenta incluir los aspectos más eficientes del modelo orientado a objetos en la estructura del modelo relacional, con variados grados de éxito [3].

Algunos sistemas de administración de base de datos como Oracle, añadieron algunas capacidades orientadas a objetos a sus productos, resultando en bases de datos híbrida objeto-relacionales. El modelo objeto-relacional extiende al modelo relacional agregando algunos tipos de datos complejos y métodos. En vez de atomicidad y valores simples en los atributos que requiere el modelo relacional, este modelo permite atributos estructurados y tienen conjuntos de arreglos de valores. También permite métodos y herencia. El lenguaje SQL fue extendido en 1999 para crear y manipular los tipos de datos más complejos que soporta este modelo, como la especificación de tipos de datos abstractos (denominados TDA o tipos definidos por el usuario) [2]. Cabe destacar que este modelo de base de datos posee mucho más semántica que el modelo relacional, por lo que hay menos fragmentación de datos.

## Características

### ■ Ventajas:

- Reutilización y compartición.
- Conservar los conocimientos relevantes y experiencias alcanzadas con las bases de datos relacionales.
- Resuelven muchas de las debilidades de las bases de datos relacionales ya conocidas.
- Mejora significativa de la productividad.

### ■ Inconvenientes:

- Mayores costos.
- Mayor complejidad.
- Todo el esfuerzo relacionado con la extensión objeto relacional puede que sólo sea útil, al final, para un porcentaje muy pequeño de las aplicaciones.

## Aspectos de Objetos en SQL

Las ganancias primordiales de la extensión objeto relacional:

- **Poder de expresión:** Capacidad de soporte para objetos y relaciones complejas.
- **Integración:** de los modelos relacional y objetos en un solo lenguaje.
- **Extensibilidad:** Capacidad de extender el sistema de tipos para entregar soporte a las nuevas necesidades de las aplicaciones.
- **Nuevas consultas más potentes:** Recursivas, multimedia, entre otros.

- **Reusabilidad:** Capacidad de compartir librerías de tipos existentes.

### **Extensiones de objeto añadidas a SQL**

- Objetos grandes (Large Objects, LOBs)
  - Binarios (BLOB).
  - Carácter (CLOB).
- Constructores de tipos
  - Tipos de filas (Row types).
  - Tipos por referencia (Reference types).
- Identidad de objetos (Identity column type)
- Tipos definidos por el usuario (User-Defined types, UDTs)
  - Tipos distintos (Distinct types).
  - Tipos estructurados (Structured types).
- Funciones, procedimientos y métodos definidos por el usuario.
  - Funciones (functions).
  - Procedimientos (procedures).
- Tipos colección (Collections)
  - Vectores (Arrays).
  - Multiconjuntos (Multisets).
- Jerarquías de tablas y de vistas
  - Sub/Supertables.

- Sub/Superviews.

Es relevante destacar que existen métodos y herramientas que realizan mapeos objetos relacionales, los cuales son detallados en el Anexo D.

## 2.2. Datos semi estructurados

En palabras simples son datos sin esquema o auto descriptibles, la información sobre la estructura está junto con los datos. En la actualidad es una necesidad integrar los datos muy estructurados con los poco estructurados. La principal diferencia entre ambos es la forma de manejar los constructores del esquema (nombres de atributos, relaciones y tipos de entidades, entre otros.). En el caso de los poco estructurados, la información del esquema se mezcla con los valores de los datos ya que un objeto de datos puede tener diferentes atributos no conocidos por adelantado. Por eso, estos tipos de datos se conocen como datos auto descriptivos.

La mayor parte de los modelos de base de datos requiere que los tipos de entidades (u objetos o registros, dependiendo del modelo) posean la misma estructura. La estructura se define en el esquema y permanece sin modificaciones a menos que el administrador de la base de datos cambie dicho esquema. A diferencia, el modelo semi estructurado posibilita una colección de nodos, cada uno conteniendo datos, posiblemente con diferentes esquemas. El nodo tiene información sobre la estructura de su contenido.

Algunas características de los datos semi estructurados son:

- No existe necesariamente una diferencia entre un identificador de un campo y el valor mismo de este.
- Un atributo de un registro puede ser otro registro.

- Un registro no necesariamente tiene que tener todos sus atributos definidos. Mientras por ejemplo en una base de datos relacional un campo debe establecerse como NULL cuando no se tiene, en un ambiente de datos semi estructurados basta con omitir dicho atributo.

A pesar de poder representarse de distintas formas, como por ejemplo documentos SGML (Standard Generalized Markup Language) y XML, actualmente la mejor manera de hacerlo es a través del lenguaje XML.

### **XML** (eXtensible Markup Language)

El lenguaje XML es el estándar para estructurar e intercambiar datos por la Web, el cual se puede utilizar para suministrar información sobre la estructura y el significado de ciertos componentes de los datos visualizados en una página web, en vez de especificar cómo se debe visualizar la información, trabajo que realiza el lenguaje HTML. El formateo de la visualización se puede especificar por separado, por ejemplo mediante XSL (Lenguaje de hojas de estilo extensible, Extensible Stylesheet Language). Recientemente, también se ha propuesto a XML como un posible modelo para el almacenamiento y la recuperación de datos, aunque hasta el momento sólo se han desarrollado unos pocos sistemas de bases de datos experimentales basados en XML.

Posee las siguientes características principales:

- **Sencillo:** Fácil de aprender y de usar.
- **Versátil:** Separa contenido, estructura y presentación.
- **Abierto:** Independiente de plataformas, empresas, lenguajes de programación o entornos de desarrollo.
- **Extensible:** Se pueden definir nuevas etiquetas.

- **Validable:** Cada documento se puede validar frente a un esquema o definición de tipo de documento (**DTD**), o se puede declarar bien formado.
- **Estructurado:** Se pueden modelar datos a cualquier nivel de complejidad.

El objeto básico en XML es el documento XML. En la construcción de un documento XML se utilizan dos conceptos de estructuración principales: elementos y atributos. Los atributos en XML proporcionan información adicional que describe a los elementos [2].

Es posible distinguir tres tipos principales de documentos XML:

- **Documentos XML centrados en los datos:** Estos documentos tienen muchos elementos de datos pequeños que respetan una estructura específica y, por tanto, pueden extraerse de una base de datos estructurada. Se formatean como documentos XML para intercambiarlos o visualizarlos por la Web.
- **Documentos XML centrados en el documento:** Son documentos con grandes cantidades de texto, como los artículos y los libros. En estos documentos hay pocos o ningún elemento de datos estructurado.
- **Documentos XML híbridos:** Estos documentos pueden tener partes que contienen datos estructurados y otras partes que son principalmente textuales o no estructuradas.

XML fue desarrollado para transmitir datos, un ejemplo relevante es los datos de un registro de un listado de libros de una base de datos tradicional. Una consulta SQL compleja podría retornar datos en el siguiente formato:

*History of Interviews, Juan, Smith, 99999-99999, Oracle Press, 2003.*

Si XML es utilizado como salida, este registro ahora posee un contexto adicional por cada elemento de datos, tal y como se observa en el código II.1

```
<book>
  <title>History of Interviews</title>
  <author>
    <firstname>Juan</firstname>
    <lastname>Smith</lastname>
  </author>
  <ISBN>99999-99999</ISBN>
  <publisher>Oracle Press</publisher>
  <publishyear>2003</publishyear>
  <price type="US">10.00</price>
</book>
```

Código II.1: Ejemplo de documento XML

El archivo tiene simetría, y cada pieza de datos tiene su contexto de término de la forma `<contexto>...</contexto>`. Cada conjunto de etiquetas (inicio / termino) y el dato que se encuentra en su interior es llamado elemento. Esta relación puede ser similar a una columna en una base de datos en donde el texto de la etiqueta es la nombre de la columna y el texto dentro de las etiquetas es el dato de una fila en esa columna. En el ejemplo del código II.1, el título puede ser el nombre de la columna y 'History of Interviews' puede ser el dato en esa fila.

También se puede notar que varias etiquetas contienen etiquetas en lugar de datos. Esto es una característica significativa de XML, lo que permite anidar datos para definir mejores relaciones. Ahora regresando a la metáfora de base de datos, la etiqueta `<author>` podría ser modelada como una tabla cuyas columnas fueran `<firstname>` y `<lastname>`. En terminología XML, estas columnas etiquetas son referenciadas como hijos de la etiqueta padre `<author>`.

Ahora si se presta atención en la etiqueta `<price>` se puede ver que incluye texto con el formato `nombre="valor"`, a esto se le llama atributo, y uno o muchos de estos pueden ser incluidos en el inicio de la etiqueta de cualquier elemento. Los atributos, sin embargo, no son permitidos en la etiqueta de término del elemento

(por ejemplo, `</tag nombre="foo">`). Los valores de los atributos deben ir entre comillas (simples o dobles, siempre y cuando la etiqueta de inicio y término sean iguales) como se especifica por SGML.

El ejemplo completo del código II.1 está entre las etiquetas `<book>...</book>`. Estas etiquetas son definidas como la raíz del documento, y quizás sólo exista una en el documento. Los documentos XML que siguen la regla de tener solamente una raíz y que todas sus etiquetas estén con su respectiva etiqueta de término son considerados documentos bien formados.

Los documentos XML poseen una estructura física y una lógica. La estructura física simplemente se refiere al archivo XML y a los otros archivos que quizás importen, mientras que la estructura lógica se refiere al prólogo y al cuerpo del documento. El XML del ejemplo del código II.1 representa el cuerpo en un documento XML, pero falta información importante que ayuda a identificar su naturaleza. Esta información es el prólogo.

## Prólogo

El prólogo consiste en la declaración del XML (el número de versión), una posible pista del lenguaje de codificación, otros atributos, y una gramática opcional o un modelo de datos especificado por la definición de esquema XML (XML Schema Definition XSD) o por la definición de tipo de documento (Documento Type Definition DTD) referenciado a una URL. El prólogo quizás además contenga el XSD o DTD real. Un ejemplo con referencia a un DTD externo se encuentra en el código II.2

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE book SYSTEM "book.dtd">
```

Código II.2: Ejemplo de referencia externa a DTD



La línea que contiene `<?...?>` del código II.2 es un ejemplo de una instrucción de procesamiento (PI) XML. En el ejemplo del código recién mencionado XML es el nombre de XML PI. Además, la codificación del conjunto de caracteres soportados en el ejemplo es una versión comprimida de Unicode llamada UTF-8. Aunque los procesadores de XML por lo general detectan la codificación de los 3 primeros bytes en el archivo, esta declaración puede ser utilizada como una pista para indicar la codificación esperada. Por último, el atributo `standalone` se refiere a si el procesador necesita incluir o importar otros archivos externos.

La segunda línea de este prólogo se refiere a *DOCTYPE*. Aquí es donde se hace la declaración de la gramática o del modelo de datos para este documento XML. En algunas aplicaciones, puede ser suficiente para procesar XML no tener conocimiento si la información está presente o no, pero la mayoría de las veces, una aplicación desea validar el documento XML que recibe para confirmar que todo está allí. Para ello, la aplicación debe conocer qué elementos se requieren, cuáles pueden tener hijos, cuáles pueden tener atributos, entre otros. En términos XML, la gramática o el modelo de datos en el ejemplo es referenciado como DTD. Este DTD puede residir en el propio archivo XML o simplemente se hace la referencia de manera que el procesador pueda localizarlo, como se muestra en el código II.2 de este ejemplo. Por otro lado el ejemplo anterior con una declaración de esquema XML se ve como esta descrito en el código II.3.

```
<? xml version="1.0" ?>
<xsd:schema xmlns:xsd=http://www.w3.org/2001/XMLSchema
xmlns:bk="http://www.mypublishsite.com/books">
```

Código II.3: Ejemplo de referencia externa a DTD

Para comenzar, se puede notar que la declaración del esquema XML tiene un prefijo **xsd:**, el cual está asociado con el nombre de espacio (namespace) del esquema XML a través de la declaración:

**xmlns:xsd="http://www.w3.org/2001/XMLSchema".**

Este prefijo es utilizado en los nombre de los tipos de datos definidos en el XSD referenciado para diferenciarlos de otros que utilicen el mismo nombre. La declaración **xsd:schema** denota el comienzo de este esquema XML incorporado en este documento XML, junto con otra declaración:

**xmlns:bk="http://www.mypublishsite.com/books"**

que define el espacio de nombres del prefijo **bk:** con el fin de identificar estos tipos tal como es definido por el autor de este modelo de datos. También se debe tener en cuenta que la declaración de esquema está dentro de la etiqueta **<book>** en lugar de en el prólogo. Esta es una diferencia distinta entre XSDs y DTDs. Por lo tanto, la declaración del esquema XML es un atributo del elemento raíz del documento y es parte del cuerpo.

## El cuerpo

El elemento raíz, que contiene el resto del documento XML, seguido del prólogo es llamado el *cuerpo* del documento XML. Esta parte está compuesta por elementos, procesando instrucciones, contenido, atributos, comentarios, referencia a entidades, entre otros. Como se mencionó anteriormente, los elementos deben comenzar y corresponder las etiquetas de término anidadas en el orden correcto; de lo contrario, el documento XML no está bien formado, y quizás los analizadores XML señalen errores por este motivo. Los elementos también pueden tener atributos, o valores, tales como **<author firstname="Juan" lastname="Smith">**. También existen atributos construidos definidos por la especificación de XML 1.0, tales como **xml:space="preserve"** para indicar que los espacios en blanco entre los elementos son considerados como datos y lo tanto son conservados.

La referencia a entidades, definida solamente en DTDs, son similares a macros

en que las entidades son definidas una vez, y referenciadas hacia ellas, tales como **&nameofentity**, se puede utilizar en lugar de la totalidad de sus definiciones. Por ejemplo, en un DTD, **<!ENTITY Copyright “Copyright 2000 por Smith, Jones, y Doe - Todos los derechos reservados”>** puede ser declarado y luego **&Copyright** podría ser utilizado como un acceso directo a través del documento XML. En analizador XML debe reconocer las entidades definidas en el DTD, incluso si el comprobador de validez pueda estar apagado y se especifica un esquema XML adicional. Finalmente, el cuerpo puede contener secciones *character data (CDATA)* para delimitar bloques de texto que de otra manera serían considerados como código, comentarios, referencias a entidades, instrucciones de procesamiento, entre otros. La sintaxis *CDATA* es la descrita en el código II.4. Estas secciones son simplemente omitidas por el analizador XML. Así el cuerpo de documento XML contiene un elemento raíz con su declaración de esquema, nodos hijos y hermanos, elementos, atributos, nodos de texto que representan el contexto textual de un elemento o atributo, y secciones CDATA.

```
<![CDATA[ characters including <, >, /, ?, & not legal anywhere else]]>
```

Código II.4: Sintaxis CDATA

## Documentos XML bien formados

Un documento XML está bien formado sólo si existe una raíz y todas las etiquetas de comienzo poseen su correspondiente etiqueta de cerrado, con su correspondiente anidación [5]. Por ejemplo, el Código II.5 del documento no está bien formado. Es posible trabajar con documentos no asociados a ninguna DTD y que en consecuencia jamás podrán ser válidos. El Código II.6 representa un documento que se encuentra bien formado.

```

<bookcatalog>
<book>
<title>History of Interviews</ti>
<author>
<firstname>Juan</firstname>
<lastname>Smith</author></lastname>
<ISBN>99999-99999</ISBN>
<publisher>Oracle Press</publisher>
<publishyear>2003</publishyear>
<price type="US">10.00</price>
</book>
</bookcatalog>
<bookcatalog2>...</bookcatalog2>

```

Código II.5: Código de documento XML que no se encuentra bien formado

Las razones por las que no es bien formado son:

- Existen dos raíces, 'bookcatalog' y 'bookcatalog2'.
- La etiqueta <title> no posee su etiqueta de término correspondiente, la que sería </title>.
- La etiqueta de término </author> no se encuentra correctamente anidada, ya que la etiqueta de término </lastname> está posicionada después de la </author>, en vez de ubicarse antes de ella. Los analizadores XML rechazarán este documento sin procesamiento adicional.

```

<bookcatalog>
  <book>
    <title>History of Interviews</title>
    <author>
      <firstname>Juan</firstname>
      <lastname>Smith</author></lastname>
    </author>
    <ISBN>99999-99999</ISBN>
    <publisher>Oracle Press</publisher>
    <publishyear>2003</publishyear>
    <price type="US">10.00</price>
  </book>
</bookcatalog>

```

Código II.6: Código de documento XML bien formado

## Documentos XML Válidos

Un documento XML válido es el que está conformado por un esquema XML o definición de tipo de documento (**DTD**), esto significa que los elementos, atributos, relacionales estructurales y secuencias en el documento XML son los mismos que los previamente especificados en el esquema XML o definición de tipo de documento (**DTD**) [5]. Por ejemplo, el código II.7 documento XML es válido con respecto al DTD.

En el código II.8 se representa el DTD, donde la declaración DOCTYPE especifica el elemento raíz, en este caso el elemento `<bookcatalog>`. Un elemento simple se compone de una etiqueta de inicio, por ejemplo, `<title>`; todo el texto “History of Interviews” entre medio de la etiqueta de inicio y la término `</title>`. Sin embargo, sólo un elemento raíz existe en un documento XML. El elemento raíz marca el inicio del documento y es considerado el padre de todos los otros elementos, los cuales se encuentran anidados con sus respectivas etiquetas de inicio y término. Para que los documentos XML sean considerados válidos con respecto a su DTD, el elemento raíz **bookcatalog** debe ser el primer elemento para empezar el cuerpo del documento XML.

Así, un analizador XML de validación, al analizar el documento XML de acuerdo con las reglas especificadas en el DTD, intenta determinar si el documento se ajusta a la DTD (si es válido), lo que significa que todos los elementos requeridos, atributos, relaciones estructurales y secuencias están según lo declarado.

```

<bookcatalog>
<book>
<title>History of Interviews</title>
<author>
<firstname>Juan</firstname>
<lastname>Smith</lastname>
</author>
<ISBN>99999-99999</ISBN>
<publisher>Oracle Press</publisher>
<publishyear>2003</publishyear>
<price type="US">10.00</price>
</book>
</bookcatalog>

```

Código II.7: Código de documento XML validamente formado

```

<!-- DTD bookcatalog may have a number of book entries -->
<!DOCTYPE bookcatalog [
<!ELEMENT bookcatalog (book)*>
<!-- Each book element has a title, 1 or more authors, etc. -->
<!ELEMENT book (title, author+, ISBN, publisher, publishyear, price)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (firstname, lastname)>
<!ELEMENT firstname (#PCDATA)>
<!ELEMENT lastname (#PCDATA)>
<!ELEMENT ISBN (#PCDATA)>
<!ELEMENT publisher (#PCDATA)>
<!ELEMENT publishyear (#PCDATA)>
<!ELEMENT price (#PCDATA)>
<!-- ATTLIST price type (US | CAN | UK | EURO) #REQUIRED -->
]>

```

Código II.8: Código DTD

## Espacios de nombre XML

El estándar W3C para XML introduce los siguientes términos con respecto a espacios de nombre XML:

- **Nombre local:** Representa el nombre de los elementos o atributos sin prefijo. En el ejemplo del código II.7, **book**, **title**, **author**, **ISBN**, entre otros son considerados nombres locales. Estos son utilizados cada vez que no existe la preocupación de que la etiqueta o nombres de atributos estén duplicados.

Los nombres locales también son empleados para referenciar a la parte del nombre de un nombre calificado.

- **Nombre calificado:** Representa el nombre completo del prefijo. Por ejemplo, como continuación de los ejemplos anteriores, **bk:title,bk:book**, entre otros, son considerados nombres calificados. Los nombres calificados son utilizados más seguido porque el esquema XML define tipos estándar, tales como dirección, cliente, orden de compra, y así sucesivamente, por lo que hay una necesidad de diferenciar la semántica.
- **Prefijo de nombre de espacio:** Representa el prefijo de un espacio de nombre declarado utilizando un prefijo espacial, **xmlns**. En el ejemplo anterior se definió un prefijo de espacio de nombre: **bk**. Los prefijos tienen su alcance y por lo tanto deben ser únicos dentro de los hijos del elemento padre que declara el espacio de nombres, pero los prefijos pueden ser reemplazados por una nueva declaración sobre un elemento descendiente o atributo.
- **Nombre expandido:** Representa el resultado de aplicación que espacios de nombre definidos por el prefijo de un espacio de nombre al nombre calificado. Por ejemplo, **bk:booklist** podría ser expandido a:

**<http://www.mypublishsite.com/books:booklist>**.

El nombre expandido nunca es visto en el documento XML en sí, pero es conceptualmente importante.

Existen dos tipos de atributos de espacios de nombre: prefijado y por defecto. Un atributo de espacio de nombre prefijado es de la forma **nsprefix:attr**, donde **nsprefix** es el prefijo de espacio de nombre previamente declarado. Una vez que el prefijo ha sido declarado, puede ser utilizado para especificar un espacio de nombre para cualquier elemento o atributo en el alcance del elemento donde fue

declarado. Por otro lado se podría tener la necesidad de declarar prefijos globales (prefijos que se puedan utilizar en cualquier parte del documento) como atributos del elemento raíz.

El atributo de espacio de nombre por defecto es **xmlns.xmlns**, tiene el efecto de especificar un espacio de nombre por defecto para todo el alcance de un elemento (incluyendo el elemento en si). Esto no aplica para atributos en sub árboles. Se puede observar un ejemplo en el código II.9, en donde la declaración del elemento raíz tiene el efecto de especificar que todos los elementos dentro de **booklist** (**book**,**title**,**author**) están en el espacio de nombre **http://www.osborne.com/books**. Sin embargo, el atributo **isbn** no lo está. Los espacios de nombre por defecto pueden ser especificados en cualquier nivel del documento y tienen el efecto de reemplazar las declaraciones anteriores. Poner **xmlns=** tiene el efecto de remover la declaración del espacio de nombre por defecto para un documento en particular de un subárbol [5].

```
<booklist xmlns="http://www.osborne.com/books">
  <book isbn="1234-5678-1234">
    <title>Oracle XML Handbook</title>
    <author>Oracle XML Team</author>
  </book>
  <book isbn="24345-564478-1344234">
    <title>The C programming language</title>
    <author>Kernighan and Ritchie</author>
  </book>
</booklist>
```

Código II.9: Declaración de espacios de nombre

## XML SCHEMA

XML Schema aparece debido a ciertas limitaciones en los DTD:

- Los DTD están escritos en una sintaxis diferente al XML.
- No tienen soporte para los espacios de nombres (namespaces).



- Sólo ofrecen tipo de datos limitados.
- Es complejo utilizar DTDs para especificar conjuntos de subelementos.

La validación de documentos XML Schemas es a través de restricciones, en donde existen dos tipos de restricciones:

- Restricciones de contenido: Describe el orden y la secuencia de elementos.
- Restricciones de tipo de datos: Describen unidades válidas de datos.

Las ventajas de utilizar XML Schema:

- Incremento en tipos de datos (double, date, etc).
- Tipos de datos definidos por el usuario.
- Restricción de tipos especializados, por ejemplo restringiendo valores máximos y mínimos.
- Agrupación de atributos y tipos de datos refinables, o “herencia”.
- Soporte para espacios de nombre (namespaces).

Se pueden añadir atributos en los elementos. Además se puede especificar qué atributo es clave del elemento y por último se pueden definir claves externas.

## **SQL/XML**

SQL/XML incorpora XML dentro del lenguaje SQL de bases de datos objeto relaciones. Permite almacenar documentos XML en sus bases de datos objeto relaciones, para luego realizar una búsqueda de dichos documentos mediante *XPath*

y *XQuery* y así publicar sus datos SQL en el formato de documento XML.

### **XPath**

XPath es un lenguaje que permite construir expresiones con la finalidad de recorrer un documento XML y entregar los nodos del documento que contienen la información que se desea. Trata partes de los documentos XML mediante expresiones de rutas de acceso, por ejemplo: “/curriculum/estudios/estudio/nombre\_establecimiento”, para acceder a los atributos basta con agregar “/curriculum/estudios/estudio/nombre\_establecimiento/text()”. Los predicados de selección, se escriben entre corchetes.

### **XQuery**

Es el lenguaje consultivo propio de XML[6], el cual utiliza la estructura de árbol de XML para expresar consultas a todas estas fuentes de datos, ya sea que estén físicamente almacenados en XML o bien sean vistos como XML vía un middleware. Por otro lado cuenta con una extensa implementación, es potente y más fácil de aprender y mantener que otros lenguajes alternativos.

## **2.3. Algunos motores con soporte objeto relacional y XML**

### **2.3.1. PostgreSQL**

Fue el pionero en muchos de los conceptos existentes en el sistema objeto relacional actual, incluido, más tarde en otros sistemas de administración comerciales. PostgreSQL es un sistema de administración de base de datos objeto relacional, que es capaz de manejar complejas rutinas y reglas. Ejemplos de

su avanzada funcionalidad son consultas SQL declarativas, control de concurrencia multi-versión, soporte multi-usuario, transacciones, optimización de consultas, herencia y arrays.

- **Herencia:** Las tablas pueden ser configuradas para heredar características de una tabla padre. Los datos son compartidos entre las tablas padre e hija(s). Las tuplas insertadas o eliminadas en la tabla hija serán insertadas o eliminadas en la tabla padre respectivamente.
- **Altamente extensible:** Soporta operadores funcionales, métodos de acceso y tipos de datos definidos por el usuario.

### Creación de tipos

Se registra un nuevo tipo de dato para ser utilizado en la base de datos actual mediante la sentencia **CREATE TYPE**. El usuario que define el nuevo tipo se convierte en su dueño. Si se detalla un nombre de esquema en la creación, entonces el tipo es creado en el esquema especificado. De lo contrario será creado en el esquema actual. El nombre del tipo debe ser distinto de cualquier nombre de tipo o dominio existente en el mismo esquema. (Porque las tablas tienen asociadas tipos de datos, el nombre del tipo también debe ser diferente del nombre de cualquier tabla existente en el mismo esquema.)

Existen cinco formas de la sentencia **CREATE TYPE**, como se muestra en el anexo A.1.1. Estas formas crean respectivamente un *tipo compuesto*, un *tipo enum*, *tipo range*, *tipo base*, o un *tipo shell*. Un *tipo shell* es simplemente un placeholder para un tipo que será definido después; es creado utilizando **CREATE TYPE** sin parámetros exceptuando el nombre del tipo. Los tipos shell son necesarios para enviar referencias cuando se crean tipos range y tipos base.

## Tipos compuestos

La primera forma de **CREATE TYPE** crea tipos compuestos. Un tipo compuesto representa la estructura de una fila o registro; es esencialmente solamente una lista de nombre de campos y sus tipos de datos. PostgreSQL permite que los tipos compuestos sean utilizados de la misma forma que un tipo simple. Por ejemplo, una columna de una tabla puede ser declarada para ser un tipo compuesto. El tipo compuesto es especificado por una lista de nombres de atributos y tipos de datos. Una recopilación de atributos puede ser especificada también, si sus tipos de datos son recopilados. Un tipo compuesto es esencialmente lo mismo que el tipo de fila de una tabla, pero utilizando **CREATE TYPE** se evade la necesidad de crear una tabla real, cuando todo lo que se quiere es definir un tipo. Un tipo compuesto es útil, por ejemplo, como argumento o como tipo de retorno en una función, incluso una columna de una tabla puede ser declarada como un tipo compuesto.

## Tipos enumerados

La segunda forma de **CREATE TYPE** crea un tipo enumerado (enum). Los tipos Enum toman una lista de una o más etiquetas citadas, cada de las cuales deben ser menor que *NAMEDATALEN* (tamaño del nombre del tipo de dato) bytes de longitud.

## Tipos range

El tercer tipo de **CREATE TYPE** crea un tipo range. El subtipo del tipo range puede ser cualquier tipo con una clase de operadores de árbol B asociado (para determinar el ordenamiento de los valores para el tipo range).

## Tipos base

La cuarta forma de **CREATE TYPE** crea un nuevo tipo base (tipo escalar). Para crear un nuevo tipo base, el usuario debe ser súper usuario (Esta restricción existe porque una definición de tipo errónea podría confundir o incluso dañar el servidor). Los parámetros pueden aparecer en cualquier orden, no solamente como aparece en la sintaxis presentada en el anexo A.1.1 y la mayoría son opcionales. Se debe registrar dos o más funciones de aplicación para operar en el tipo en un lenguaje de bajo nivel, generalmente C. Un tipo base siempre debe tener funciones de entrada y salida. Estas funciones determinan como el tipo aparece en cadenas (para entradas por el usuario y la salida para el usuario) y como el tipo es organizado en la memoria. La función de entrada toma un carácter terminado en null como su argumento y retorna la representación interna del tipo (en memoria). La función de salida toma la representación interna del tipo como argumento y retorna un carácter terminado en null. Si se desea hacer algo más con el tipo además de simplemente almacenarlo, se debe proporcionar funciones adicionales para implementar las operaciones que se desean tener para el tipo.

[7]

Se supondrá que se desea definir un tipo *complejo* que representa números complejos. Una forma natural de representar un número complejo en memoria sería la estructura representada en el código II.10.

```
typedef struct Complejo {  
    double    x;  
    double    y;  
} Complejo;
```

Código II.10: Struct del tipo complejo en C, utilizado en PostgreSQL para definir tipos

Se deberá hacer que este tipo sea de paso por referencia, ya que es demasiado grande para caber en un valor *Datum* (Es un tipo de dato universal, cualquier tipo de dato puede ser accedido a través de un valor *Datum* [8]).

Como representación externa del tipo, se elige una cadena de forma  $(x,y)$ . Las funciones de entrada y salida habitualmente no son difíciles de escribir, especialmente la función de salida como se puede apreciar en los códigos II.11 y II.12. Pero cuando se define la representación de la cadena externa del tipo, se debe recordar que se eventualmente se tendrá que escribir un parser completo y robusto para la representación como la función de entrada.

```
PG_FUNCTION_INFO_V1(complex_in);
Datum
complex_in(PG_FUNCTION_ARGS){
    char      *str = PG_GETARG_CSTRING(0);
    double    x,y;
    Complex   *result;

    if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2)
        ereport(ERROR,
                (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION),
                 errmsg("invalid input syntax for complex: \"%s\"",str)));
    result = (Complex *) palloc(sizeof(Complex));
    result->x = x;
    result->y = y;
    PG_RETURN_POINTER(result);
}
```

Código II.11: Función de entrada del tipo Complejo

```

PG_FUNCTION_INFO_V1(complex_out);

Datum
complex_out(PG_FUNCTION_ARGS)
{
    Complex    *complex = (Complex *) PG_GETARG_POINTER(0);
    char        *result ;

    result = (char *) palloc(100);
    snprintf ( result , 100, "( %g, %g)", complex->x, complex->y);
    PG_RETURN_CSTRING(result);}

```

Código II.12: Función de salida del tipo Complejo

## Tipos Arrays

Siempre que se crea un tipo definido por el usuario, PostgreSQL automáticamente crea un tipo array asociado, cuyo nombre se compone del nombre del tipo de elemento precedido con un guión bajo, y truncado si es necesario para mantenerlo de longitud menor que *NAMEDATALEN* bytes. (Si el nombre generado colisiona con un nombre de tipo existente, el proceso es repetido hasta que encuentra un nombre sin colisiones.). Este tipo array creado implícitamente es de longitud variable y utiliza las funciones de entrada / salida *array\_in* y *array\_out*.

## XML: Tipos y funciones

El tipo de datos XML puede ser utilizado para almacenar datos XML. Su ventaja sobre el almacenamiento de datos XML en un campo de texto es que comprueba los valores de entrada para que estén bien formados, y hay funciones soportadas para desempeñar operaciones del tipo seguras.

El tipo XML puede almacenar documentos bien formados, como está definido en los estándares XML, también fragmentos de contenido, los cuales son definidos por la producción de contenido XMLDecl en el estándar XML. Esto significa que los fragmentos de contenido pueden tener más que un elemento de nivel superior o un nodo carácter. La expresión *xmlvalue* es un documento que puede ser utilizado para evaluar si un valor XML en particular es un documento completo o sólo un fragmento de contenido.

### Creando valores XML

Para generar un valor del tipo XML de datos de caracteres, se debe utilizar la función *xmlparse*, cuya sintaxis es detallada en el anexo A.1.2. La sintaxis específica de PostgreSQL para convertir caracteres a cadenas en valores XML de acuerdo a los estándares SQL, es la descrita en el anexo A.1.2.

El tipo XML no valida los valores de entrada del DTD, incluso cuando el valor de entrada especifica un DTD. Tampoco existe actualmente un soporte para validar otros lenguajes de esquema tales como el esquema XML.

La operación inversa para convertir valor carácter desde XML, utiliza la función *xmlserialize* que se especifica en el Anexo A.1, en donde **type** puede ser *character*, *character varying*, o *text* (o un alias para uno de esos). Nuevamente de acuerdo a los estándares SQL, esta es la única forma de conversión entre el tipo XML y los tipos carácter, pero PostgreSQL también permite simplemente castear el valor. Cuando un valor carácter es casteado a tipo XML o desde tipo XML sin pasar a través de *XMLPARSE* o *XMLSERIALIZE*, respectivamente, la elección del documento contra el contenido es determinado por el parámetro de configuración de sesión “XML option”, en el anexo A.1.2 se especifica su sintaxis.

Un conjunto de funciones y expresiones están disponibles para crear contenido XML desde datos SQL. Como tal, son particularmente apropiados para forma-



tear resultados de consultas en documentos XML procesados en aplicaciones clientes.

La función *xmlcomment* creará un valor XML conteniendo un comentario XML con un texto específico como contenido. El texto no puede contener “–” o terminar con un “-” con el fin de que el constructor resultante sea un comentario XML válido. Si el argumento es nulo, el resultado también lo es. La sintaxis de esta función se puede apreciar en el anexo A.1.2.

### 2.3.2. SQLServer

#### Tipos

Un tipo definido por el usuario es implementado a través de una clase de un ensamblado en CLR (common language runtime) del framework Microsoft .NET. Para unir el tipo definido por el usuario con su implementación, el ensamblado CLR que contiene la implementación del tipo primero debe ser registrado en SQL Server utilizando la sentencia **CREATE ASSEMBLY** [9]. La sintaxis para crear un tipo definido por el usuario es detallada en el anexo A.2.1., mediante la sentencia **CREATE TYPE** al igual que en PostgreSQL.

En el código II.13 se muestra un ejemplo de cómo crear un alias de tipo basado en un tipo de dato varchar. Otro ejemplo es el especificado en el código II.14, donde se crea un tipo de dato definido por el usuario. Para crear tipo tabla definida por el usuario se debe utilizar el código II.15.

```
CREATE TYPE SSN  
FROM varchar(11) NOT NULL;
```

Código II.13: Ejemplo de creación de alias de tipo basado en el tipo de dato varchar SQLServer

```
CREATE ASSEMBLY utf8string
FROM '\\PC\utf8string\utf8string . dll ' ;
GO
CREATE TYPE Utf8String
EXTERNAL NAME utf8string.[Microsoft.Samples.SqlServer.utf8string];
GO
```

Código II.14: Ejemplo de creación tipo definido por el usuario referenciando al ensamblador SQLServer

```
CREATE TYPE LocationTableType AS TABLE
( LocationName VARCHAR(50),
  CostRate INT)
GO
```

Código II.15: Ejemplo de creación de un tipo tabla definida por el usuario SQLServer

### Valores y funciones XML

SQL Server proporciona una plataforma eficaz con el fin de programar aplicaciones completas para la administración de datos semi estructurados. La compatibilidad con XML se encuentra integrada en todos los componentes de SQL Server e incluye lo siguiente:

- El tipo de datos XML. Los valores XML se pueden almacenar de forma nativa en una columna de tipo de datos XML cuyo tipo se puede asignar de acuerdo con una colección de esquemas XML o que puede dejarse sin tipo. Es posible indizar la columna XML.
- La capacidad de especificar una consulta XQuery con datos XML almacenados en columnas y variables de tipo XML.

- Mejoras en OPENROWSET que permiten la carga masiva de datos XML.
- La cláusula FOR XML, para recuperar los datos relacionales en formato XML.
- La función OPENXML, para recuperar datos XML en formato relacional.

SQL Server posee las siguientes opciones de almacenamiento de XML:

- Almacenamiento nativo como tipo de datos XML: Los datos son almacenados en una representación interna que mantiene el contenido XML de los mismos. Dicha representación interna incluye información acerca del orden de los documentos, la jerarquía de inclusión y los valores de los elementos y atributos.
- Asignación de almacenamiento entre XML y relacional: El esquema anotado (AXSD) permite descomponer el código XML en columnas en una o más tablas. Así, se preserva la confiabilidad de los datos en el nivel relacional. Como resultado, la estructura jerárquica se mantiene aunque el orden entre los elementos sea omitido. El esquema no puede ser recursivo.
- Almacenamiento de objetos grandes, **[n]varchar(max)** y **varbinary(max)**: Se almacena una copia idéntica de los datos. Esto resulta de utilidad en el caso de aplicaciones para fines específicos, tales como documentos legales. La mayor parte de las aplicaciones no requieren una copia exacta y les basta con el contenido XML.

## FOR XML

Se pueden recuperar resultados formales de una consulta SQL como XML especificando la cláusula FOR XML en la consulta. La cláusula FOR XML puede ser utilizada en consultas de nivel superior y en subconsultas. La cláusula FOR XML

de nivel superior solamente puede emplearse en la instrucción SELECT. En el caso de las subconsultas, FOR XML puede usarse en las instrucciones INSERT, UPDATE y DELETE. También puede utilizarse en instrucciones de asignación [9]. En una cláusula FOR XML se especifica uno de estos modos:

- RAW: Este modo genera un único elemento <row> por cada fila del conjunto de filas retornado por la instrucción SELECT. Para generar una jerarquía XML se pueden escribir consultas FOR XML anidadas.
- AUTO: Genera anidamiento en el XML resultante, empleando una heurística basada en la forma en que se especifica la instrucción SELECT. El control sobre la forma del XML generado es mínimo. Es posible escribir consultas FOR XML anidadas para generar una jerarquía XML más allá de la forma del XML generado mediante la heurística del modo AUTO.
- EXPLICIT: Concede un mayor control de la forma del XML. Es posible mezclar atributos y elementos con total libertad para decidir la forma del XML. Se requiere de un formato específico para el conjunto de filas resultantes generado por la ejecución de la consulta. Después, el formato del conjunto de filas se asigna a una forma de XML. La eficacia del modo EXPLICIT reside en que se pueden mezclar atributos y elementos con total libertad, crear contenedores y propiedades complejas anidadas, crear valores separados por espacios y contenido mezclado.
- PATH: Junto con la característica de las consultas anidadas FOR XML, proporciona la flexibilidad del modo EXPLICIT de una manera más sencilla.

## **OPENXML**

Proporciona una vista de un conjunto de filas en un documento XML. Puesto que OPENXML es un proveedor de conjuntos de filas, puede ser utilizado en

instrucciones Transact-SQL en las que pueden aparecer tablas, vistas o la función OPENROWSET.

En el anexo A.2.2 se detalla mediante ejemplos creación de tablas utilizando columna XML, declaración de variables, asignación directa de datos a una variable, usando FOR XML, entre otros.

### 2.3.3. DB2

Es una base de datos propietaria de IBM, la cual posee capacidades objeto relacional ya que cuenta con sentencias SQL como *CREATE TYPE* y tipos de datos complejos, también trabaja directamente con XML.

DB2 da soporte a una serie de tipos de datos incorporados. También proporciona soporte para los tipos de datos definidos por el usuario. La Figura 2.9 ilustra los tipos de datos internos a los que este motor da soporte [10].

La sentencia *CREATE TYPE* define un tipo de datos definido por el usuario en el servidor actual.

#### **Tipos definidos por el usuario**

Existen seis tipos de tipos de datos definidos por el usuario:

**Tipo diferenciado:** Este tipo comparte su representación interna con un tipo existente (su tipo de “origen”), sin embargo, es considerado un tipo independiente y a la vez incompatible con la mayoría de operaciones. El tipo diferenciado siempre tiene su fuente en los tipos de datos incorporados.

La ejecución satisfactoria de la sentencia también genera funciones para la conversión entre el tipo diferenciado y su tipo de fuente y, opcionalmente, genera el soporte para utilizar los operadores de comparación (=, <>, <, <=, > y >=) con el

tipo diferenciado. En los casos en que el tipo de fuente es un tipo con parámetros, la función para convertir el tipo diferenciado al tipo de fuente tendrá como nombre de función el nombre del tipo de fuente sin los parámetros. El tipo del valor de retorno de esta función incluirá los parámetros dados en la sentencia CREATE TYPE (diferenciado). La función para convertir del tipo de fuente al tipo diferenciado tendrá un parámetro de entrada cuyo tipo es el tipo de fuente incluyendo sus parámetros. En el ejemplo del código II.16 se visualiza la creación de un tipo diferenciado llamado MILLAS que está basado en un tipo de datos DOUBLE, al estar con la opción **WITH COMPARISONS** da como resultado la creación de operadores de comparación mencionados previamente y su función de conversión retorna un DOUBLE ver código II.17.

```
CREATE TYPE T_MILES AS DOUBLE  
WITH COMPARISONS
```

Código II.16: Ejemplo CREATE TYPE Diferenciado DB2

```
FUNCTION DOUBLE (T_MILES) RETURNS DOUBLE  
  
FUNCTION T_MILES (DOUBLE) RETURNS T_MILES
```

Código II.17: Ejemplo función para tipo diferenciado DB2

**Tipo estructurado:** El tipo estructurado es un tipo definido por el usuario con una estructura definida en la base de datos. Incluye una secuencia de atributos con nombre, cada uno de los cuales tiene un tipo de datos. Un tipo estructurado además contiene una conjunto de definiciones de método. Este tipo puede ser utilizado como tipo de una tabla, de una vista o columna. Al momento de ser utilizado para una tabla o vista, esa tabla o vista se denomina tabla con tipo o vista con tipo.

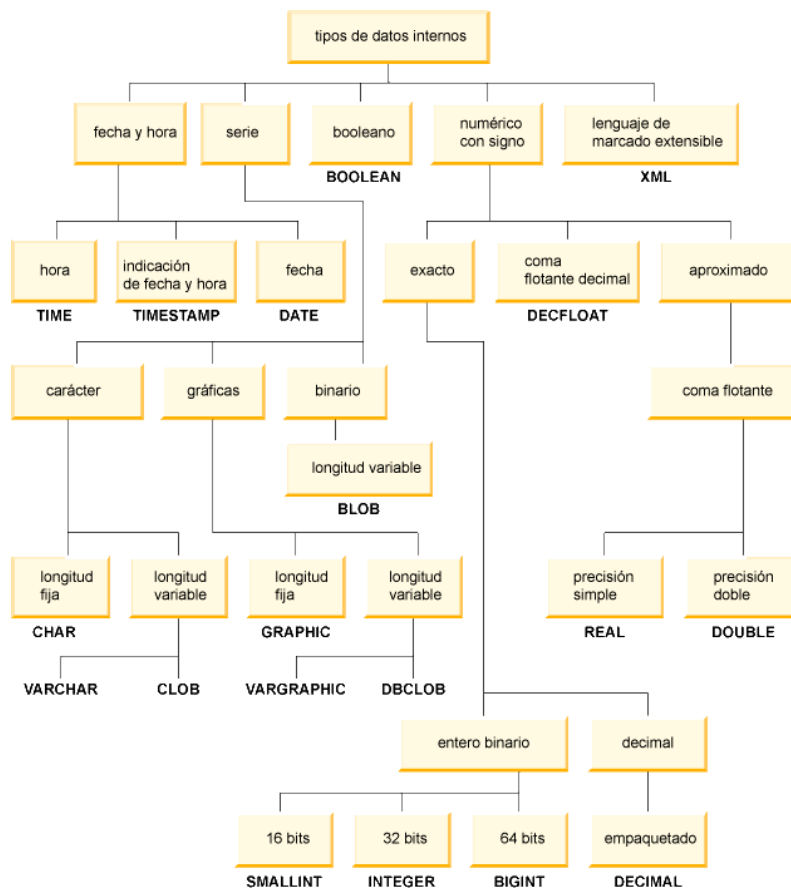


Figura 2.9. Tipos de datos internos de DB2 soportados

**Tipo de referencia:** Es un tipo compañero de un tipo estructurado. De forma semejante a un tipo diferenciado, un tipo de referencia es un tipo escalar que comparte una representación común con uno de los tipos de datos internos. La representación de este tipo se define cuando se crea el tipo raíz de una jerarquía de tipos. Cuando un tipo de referencia es utilizado, se especifica un tipo estructurado como parámetro del tipo. Este parámetro se denomina el tipo de destino de la referencia.

**Tipo de matriz:** Un tipo de matriz definido por el usuario es un tipo de datos que se define como matriz con elementos de otro tipo de datos. Cada tipo de matriz común tiene un índice con el tipo de datos INTEGER y tiene definida una

cardinalidad máxima. Cada matriz asociativa tiene un índice con el tipo de datos INTEGER o VARCHAR y no tiene definida una cardinalidad máxima.

**Tipo de fila:** Un tipo de fila es un tipo de datos que se describe como una secuencia ordenada de campos con nombre, cada uno con un tipo de datos asociado, que representa naturalmente una fila. Un tipo de fila puede ser utilizada como tipo de datos para variables y parámetros en SQL PL con el fin de suministrar una manipulación sencilla de una fila de datos.

Un tipo de fila incluye uno o más campos con tipos de datos asociados que conforman una fila de datos.

**Tipo de cursor:** Es un tipo de datos definido por el usuario con la palabra clave CURSOR y opcionalmente con un tipo de fila asociado. Un tipo de cursor definido por el usuario con un tipo de fila asociado es un tipo de cursor de tipo firme (la tipificación firme, requiere tipos de datos coincidentes, lo que significa que es necesario convertir explícitamente uno o ambos tipos de datos en un tipo de datos común antes de realizar comparaciones y asignaciones.); de lo contrario, es un tipo de cursor de tipo no firme. El valor de un tipo de cursor definido por el usuario representa una referencia a un cursor subyacente.

Algunos ejemplos de utilización de definición de tipos de usuario en DB2 son expuestos en el anexo A.3.1

### **Valores y funciones XML**

DB2 proporciona los tipos de datos XML para almacenar documentos XML bien formados. Un valor XML representa el XML con el formato que corresponde en forma de documento XML, contenido XML o secuencia de nodos XML. Un valor XML que se encuentra almacenado en una tabla como valor de una columna definida con el tipo de datos XML tiene que ser un documento XML con el formato correcto. Los valores XML se procesan en una representación interna que no puede



ser comparada con ningún valor de serie. Un valor XML puede transformarse en un valor de cadena de caracteres serializada, representante del documento XML que utiliza la función *XMLSERIALIZE*. De igual manera, para almacenar datos XML en una columna de tipo de datos XML, los datos deben ser transformados usando la función *XMLPARSE* [10].

A las expresiones que tienen como resultado un valor de tipo de datos XML se les aplican restricciones especiales, estas expresiones y columnas no se encuentran permitidas:

- Una lista SELECT precedida por la cláusula DISTINCT.
- Una cláusula ORDER BY.
- Un predicado BETWEEN, IN o LIKE básico y cuantificado.
- Una cláusula GROUP BY.
- Una subselección de un operador de conjunto que no sea UNION ALL.
- Una función agregada con DISTINCT.

### **Consultas SQL/XML**

Tal y como su nombre lo indica, SQL/XML sirve como puente entre los mundos SQL y XML. Evolucionó como parte del estándar SQL que ahora incluye especificaciones para incrustar expresiones XQuery o XPath dentro de instrucciones SQL. XPath es un lenguaje que permite navegar por un documento XML para realizar búsquedas de diferentes elementos y atributos. XQuery es compatible con XPath. Es relevante destacar que las expresiones XQuery (y XPath) distinguen mayúsculas de minúsculas. Por ejemplo, una expresión XQuery que referencia al elemento XML “rar” no se aplicará a los elementos XML denominados “RAR” o “Rar”.

DB2 suministra muchas otras funciones integradas para manipular tipos de datos XML. A partir de la versión 9, DB2 admite una nueva Tecnología llamada XML pure, cuyas características son:

- XML es un tipo de dato.
- XML se indexa para apresurar búsquedas y recuperación de datos.
- XML es gestionado y almacenado en un contenedor separado.
- XML se valida con esquemas en la base de datos.
- Incorpora procedimientos para generar documentos XML en tablas.
- Incluye funciones para transformar XML en SQL y viceversa.

Las principales funciones XML con sus respectivos ejemplos son especificadas en el anexo A.3.2.

## **2.4. Oracle**

Una base de datos Oracle es una colección de datos en uno o más archivos. La base de datos contiene estructuras físicas y lógicas. En el proceso de crear una aplicación, se crean estructuras como tablas e índices para almacenar filas y acelerar su recuperación. Se pueden crear sinónimos por los nombres de objetos, ver objetos en diferentes bases de datos (a través de enlaces a base de datos), y se puede restringir el acceso a los objetos. Incluso se puede utilizar tablas externas para acceder archivos a fuera de la base de datos como si las filas en los archivos fueran filas en tablas.

Debido a los requerimientos de las nuevas aplicaciones, desde su versión 8, Oracle ha sido significativamente extendido con conceptos del modelo de bases de

datos orientadas a objetos. De esta forma, aunque las estructuras de datos utilizadas para almacenar la información siguen siendo tablas, los usuarios pueden utilizar muchos de los mecanismos de orientación a objetos para definir y acceder a los datos. Por este motivo, se dice que se trata de un modelo de datos objeto relacional.

Este motor suministra mecanismos para que el usuario pueda definir sus propios tipos de datos, cuya estructura puede ser compleja, y que pueden ser aplicados para asignar un tipo a una columna de una tabla. Además admite el concepto de objetos, por lo que un objeto tiene un tipo, se almacena en cierta fila de cierta tabla y posee un identificador único (OID). Dichos identificadores pueden ser utilizados para referenciar a otros objetos y de esta forma representar relaciones de asociación y de agregación. También se proporciona mecanismos para asociar métodos a tipos, y constructores para diseñar tipos de datos multivaluados (colecciones) y tablas anidadas. La mayor deficiencia de este sistema es la imposibilidad de definir jerarquías de especialización y herencia, lo cual es una importante desventaja con respecto a las bases de datos orientadas a objetos.

Ya que Oracle es el motor de base de datos escogido para la implementación y pruebas del presente estudio, como fue mencionado en los objetivos del proyecto, se profundizará en detalle sus funciones, tipos de datos, objetos, entre otros.

#### **2.4.1. Tipos de datos colección**

Para poder implementar relaciones 1: N, Oracle permite definir el tipo colección. Un dato de tipo colección está formado por un número indefinido de elementos, todos del mismo tipo. De esta manera, es posible almacenar en un atributo un conjunto de tuplas en forma de array (VARRAY), o en forma de tabla anidada. Los tipo colección también tienen por defecto funciones constructoras de colecciones cuyo nombre coincide con el del tipo. Los argumentos de entrada de estas

funciones son el conjunto de elementos que forman la colección separados por comas y entre paréntesis, y el resultado es un valor del tipo colección. En Oracle es posible diferenciar entre un valor nulo y una colección vacía. Para construir una colección sin elementos se puede utilizar la función constructora del tipo seguida por dos paréntesis sin elementos dentro.

### **El tipo VARRAY**

Un array es un conjunto ordenado de elementos del mismo tipo. Cada elemento tiene asociado un índice que indica su posición dentro del array. Oracle permite que los VARRAY sean de longitud variable, aunque es necesario especificar un tamaño máximo cuando este es declarado. Se puede utilizar el tipo VARRAY para:

- Definir el tipo de dato de una columna de una tabla relacional.
- Definir el tipo de dato de un atributo de un tipo de objeto.
- Para definir una variable PL/SQL, un parámetro, o el tipo que devuelve una función.

Cuando se declara un tipo VARRAY no se produce ninguna reserva de espacio. Si el espacio que requiere lo permite, se almacena junto con el resto de columnas de su tabla, pero si es demasiado largo (más de 4000 bytes) se almacena aparte de la tabla como un BLOB. La principal limitación del tipo VARRAY es que en las consultas es imposible poner condiciones sobre los elementos almacenados dentro. Desde una consulta SQL, los valores de un VARRAY solamente pueden ser accedidos y recuperados como un bloque. Es decir, no se puede acceder individualmente a los elementos de un VARRAY. Sin embargo, desde un programa PL/SQL si que es posible definir un bucle que itere sobre los elementos de un VARRAY.

## Tablas anidadas

Una tabla anidada es un conjunto de elementos del mismo tipo sin ningún orden predefinido. Estas tablas solamente pueden tener una columna que puede ser de un tipo de datos básico de Oracle, o de un tipo de objeto definido por el usuario. En este último caso, la tabla anidada también puede ser considerada como una tabla con tantas columnas como atributos tenga el tipo de objeto. Para relacionar las tuplas de una tabla anidada con la tupla a la que pertenecen, se utiliza una columna oculta que aparece en la tabla anidada por defecto. Todas las tuplas de una tabla anidada que pertenecen a la misma tupla tienen el mismo valor en esta columna (**NESTED\_TABLE\_ID**). A diferencia de los **VARRAY**, los elementos de las tablas anidadas (**NESTED\_TABLE**) sí pueden ser accedidos individualmente, y es posible poner condiciones de recuperación sobre ellos. En el anexo A.2.1 se muestra una forma conveniente de acceder individualmente a los elementos de una tabla anidada mediante un cursor anidado. Además, las tablas anidadas pueden estar indexadas.

En la tabla II-1 se muestra una comparación de ambos tipos de colección.

	<b>Varray</b>	<b>Nested table</b>
Tamaño máximo	Si	No
Borrado elementos individual	No	Si
Almacenamiento datos	In-line	Out-of-line
Mantenimiento del orden	Si	No

Tabla II-1: Tipos colección: Varray versus Nested Table

### 2.4.2. Tipos de objetos

El modelo relacional fue diseñado para representar datos como una serie de tablas con sus respectivas columnas y atributos. Oracle Database 11g es una base de datos objeto relacional, por lo que incorpora las tecnologías orientadas al ob-

jeto. Gracias a esto, permite construir tipos de objetos complejos, tales como:

- Definir objetos dentro de objetos.
- Encapsular o asociar métodos con dichos objetos.

### **Estructura de un tipo de objeto**

Un tipo objeto se compone de dos partes: especificación y cuerpo. La especificación organiza la interface a las aplicaciones; es donde se declaran las estructuras de datos (grupo de atributos) y las operaciones (métodos) que son necesarios para la manipulación de los datos. Por otro lado, el cuerpo es lugar donde se definen los métodos, por lo que es quien implementa la especificación. La estructura de un tipo de objeto es representada de forma gráfica en la figura 2.10.

En la especificación se encuentra toda la información necesaria para utilizar los métodos. Es apropiado pensar que el cuerpo es como una caja negra y la especificación como la interface operacional. Y por esto mismo es posible realizar cambios, optimizar o depurar el cuerpo sin la necesidad de modificar la especificación y con esto no afectar las aplicaciones cliente.

En la especificación de tipo objeto los atributos se deben declarar antes que cualquiera de los métodos. Por lo que si la especificación solamente declara atributos, el cuerpo no es necesario. No es posible hacer declaraciones de atributos en el cuerpo del tipo objeto.

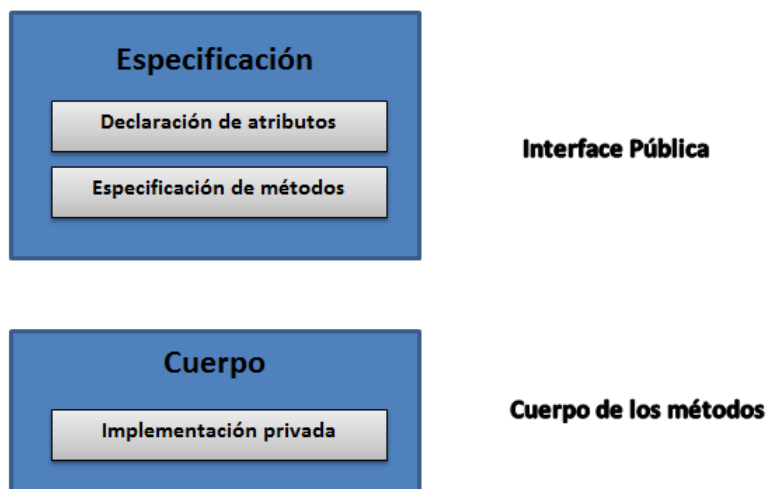


Figura 2.10. Estructura de un tipo de objeto [12]

Todas las declaraciones en la especificación del tipo son públicas, por tanto, son visibles fuera del tipo objeto. No obstante, el cuerpo puede tener declaraciones privadas, que definan métodos necesarios para el funcionamiento interno del objeto. El entorno de las declaraciones privadas es local al cuerpo del objeto. En el código II.18 se presenta un ejemplo para comprender mejor esta estructura, en donde se define un tipo objeto para manipular números complejos con algunas operaciones, en el código II.19 se encuentra la creación del cuerpo del objeto y la definición de cada función previamente declarada en la especificación.

```

CREATE TYPE Complejo AS OBJECT (
  parte_real  REAL, parte_imaginaria REAL,
  MEMBER FUNCTION mas (x Complejo) RETURN Complejo,
  MEMBER FUNCTION menos (x Complejo) RETURN Complejo,
  MEMBER FUNCTION veces (x Complejo) RETURN Complejo,
  MEMBER FUNCTION div_por (x Complejo) RETURN Complejo);
  
```

Código II.18: Ejemplo creación especificación de tipo objeto ORACLE

```

CREATE TYPE BODY Complejo AS MEMBER FUNCTION mas(x Complejo)
    RETURN Complejo IS
BEGIN
    RETURN Complejo(parte_real + x.parte_real, parte_imaginaria +
        x.parte_imaginaria);
END mas;

MEMBER FUNCTION menos (x Complejo) RETURN Complejo IS
BEGIN
    RETURN Complejo(parte_real – x.parte_real, parte_imaginaria –
        x.parte_imaginaria);
END menos;

MEMBER FUNCTION veces (x Complejo) RETURN Complejo IS
BEGIN
    RETURN Complejo(parte_real * x.parte_real – parte_imaginaria *
        x.parte_imaginaria, parte_real * x.parte_imaginaria + parte_imaginaria *
        x.parte_real);
END veces;

MEMBER FUNCTION div_por (x Complejo) RETURN Complejo IS
    z REAL := x.parte_real**2 + x.parte_imaginaria**2;
BEGIN
    RETURN Complejo((parte_real * x.parte_real + parte_imaginaria *
        x.parte_imaginaria) / z, (parte_imaginaria * x.parte_real – parte_real *
        x.parte_imaginaria) / z); END div_por;

```



## Componentes de un tipo objeto

Un tipo objeto encapsula datos y operaciones, por este motivo en la especificación solamente se puede declarar atributos y métodos, pero no excepciones, constantes, tipos o cursores. Es requerido al menos un atributo, pero los métodos son opcionales.

Así como las variables, un atributo debe ser declarado mediante un nombre y un tipo. El nombre tiene que ser único dentro del tipo objeto (pero puede ser reutilizado en otros objetos) y el tipo puede ser cualquier tipo excepto los siguientes:

- LONG y LONG RAW.
- NCHAR, NCLOB y NVARCHAR2.
- MLSLABEL y ROWID.
- Los tipos específicos de **PL/SQL**: BINARY\_INTEGER (y todos sus subtipos), BOOLEAN, PLS\_INTEGER, RECORD, REF CURSOR, %TYPE y %ROWTYPE.
- Los tipos definidos en paquetes **PL/SQL**.

No se puede inicializar un atributo en la declaración utilizando el operador de asignación o cláusula DEFAULT, de igual forma que no se permite la restricción NOT NULL. Sin embargo, los objetos pueden ser almacenados en tablas de la base de datos en que las que sí es posible imponer este tipo de restricciones. Se puede llegar a crear estructuras de datos muy complejas, por ejemplo, el tipo de un atributo puede ser otro tipo de objeto (lo que sería un tipo de objeto anidado).

Una definición simple para el concepto método sería que es un subprograma declarado en una especificación de tipo mediante la palabra clave *MEMBER*. Un método no puede tener el mismo nombre que el tipo de objeto ni el de ninguno

de sus atributos. Muchos métodos se componen de dos partes: especificación y cuerpo. La especificación consta de nombre del método, una lista opcional de parámetros y en el caso de funciones de un tipo de retorno. El cuerpo es el código que ejecuta las operaciones especificadas. Para cada especificación de método de una especificación de tipo tiene que existir el correspondiente cuerpo del método.

### **Parámetro SELF**

Todos los métodos de un tipo de objeto reciben como primer parámetro una instancia predefinida del mismo tipo denominada SELF. Indistintamente de que SELF sea declarado explícita o implícitamente, siempre es el primer parámetro pasado a un método.

El modo de acceso de SELF cuando no se declara explícitamente es:

- En procedimientos, si SELF no es declarado, su modo por omisión es IN OUT.
- En funciones miembro el acceso de SELF es IN.

En el cuerpo de un método, SELF señala al objeto a partir del cual se invocó el método. Los métodos pueden hacer referencia a los atributos de SELF sin necesidad de utilizar un cualificador.

### **Sobrecarga**

Los métodos del mismo tipo (funciones y procedimientos) pueden ser sobrecargados, es posible utilizar el mismo nombre para métodos distintos si sus parámetros formales difieren en número, orden o tipo de datos. Cuando uno de los métodos

es invocado, PL/SQL encuentra el cuerpo adecuado comparando la lista de parámetros actuales con cada una de las listas de parámetros formales. Sin embargo, la sobrecarga no es posible en el caso de las siguientes situaciones:

- Si los parámetros formales difieren sólo en el modo.
- Si las funciones solamente difieren en el tipo de retorno.

## **Constructores**

Cada tipo de objeto tiene un constructor que es una función definida por el sistema con el mismo nombre que el objeto. El constructor es utilizado para inicializar y retornar una instancia de ese tipo de objeto. Oracle genera un constructor por defecto para cada tipo de objeto. Los parámetros del constructor coinciden con los atributos del tipo de objeto, es decir, los parámetros y atributos son declarados en el mismo orden y poseen el mismo nombre y tipo. Por otro lado, PL/SQL jamás invoca al constructor implícitamente, por lo que el usuario debe llamarlo explícitamente.

### **2.4.3. Declaración e inicialización de objetos**

Una vez que un tipo de objeto ha sido definido y se ha instalado en el esquema de la base de datos, este puede ser utilizado en cualquier bloque PL/SQL. Las instancias de los objetos son creadas en tiempo de ejecución. En un bloque o subprograma, los objetos locales son instanciados cuando se entra al bloque o subprograma y estos dejan de existir cuando sale del bloque o subprograma. Por otro lado en un paquete, los objetos se instancian cuando se referencia por primera vez al paquete y dejan de existir cuando finaliza la sesión. Los tipos de objetos se declaran del mismo modo que cualquier tipo interno.

Además es posible declarar objetos como parámetros formales de funciones y procedimientos, por lo que es posible pasar objetos a los subprogramas almacenados y de un subprograma a otro.

Hasta que se inicializa un objeto, invocando al constructor para ese tipo de objeto, el objeto es atómicamente nulo. Esto significa que el objeto es nulo, no sólo sus atributos. Un objeto nulo siempre es diferente a cualquier otro objeto. De hecho, la comparación de un objeto nulo con otro objeto siempre resulta NULL. Del mismo modo, si se asigna un objeto con otro objeto atómicamente nulo, el primero se convierte a su vez en un objeto atómicamente nulo (y para poder utilizarlo debe ser reinicializado). En resumen, si se asigna el no-valor NULL a un objeto, éste se convierte en atómicamente nulo, una buena práctica de programación consiste en inicializar los objetos en su declaración.

PL/SQL se comporta del siguiente modo cuando accede a objetos sin inicializar:

- La operación de comparación IS NULL siempre produce TRUE cuando se aplica a un objeto no inicializado o a cualquiera de sus atributos.
- Los atributos de un objeto no inicializado se evalúan en cualquier expresión como NULL.
- Intentar asignar valores a los atributos de un objeto sin inicializar provoca la excepción predefinida ACCESS\_INTO\_NULL.

La invocación de los métodos de un objeto no inicializado está permitida, pero en este caso:

- SELF toma el valor NULL.
- Cuando los atributos de un objeto no inicializado son pasados como parámetros OUT o IN OUT, se produce una excepción si se intenta asignarles un valor.

- Cuando los atributos de un objeto no inicializado se pasan como parámetros IN, se evalúan como NULL.

Para poder acceder o cambiar los valores de un atributo se utiliza la notación punto ('.'). Los nombres de los atributos pueden encadenarse, lo que permite acceder a los atributos de un tipo de objeto anidado.

#### **2.4.4. Invocación de constructores y métodos**

La invocación de un constructor está permitida en cualquier punto en donde se puede invocar una función. Como las funciones, un constructor se invoca como parte de una expresión.

La especificación de un método se hace junto a la creación de su tipo y debe llevar siempre asociada una directiva de compilación (PRAGMA RESTRICT\_REFERENCES), para evitar que los métodos manipulen la base de datos o las variables del paquete PL/SQL. Se tienen las siguientes directivas y su significado:

- WNDS: no se permite al método modificar las tablas de la base de datos
- WNPS: no se permite al método modificar las variables del paquete PL/SQL
- RNDS: no se permite al método leer las tablas de la base de datos
- RNPS: no se permite al método leer las variables del paquete PL/SQL

#### **Paso de parámetros a un constructor**

Cuando los parámetros son enviados a un constructor la invocación asigna valores iniciales a los atributos del objeto que se está instanciando. Es necesario proveer parámetros para cada uno de los atributos ya que, a diferencia de las constantes y variables, los atributos no poseen la cláusula DEFAULT. Además es

posible invocar al constructor utilizando la notación con nombre en lugar de la notación posicional.

### **Invocación de métodos**

Como los subprogramas de un paquete, los métodos son invocados utilizando la notación punto.

Es posible encadenar los llamados a los métodos, en donde la ejecución se realiza de izquierda a derecha.

En las sentencias SQL el llamado de un métodos sin parámetros requiere la lista vacía de parámetros: '()'. En sentencias de procedimiento, la lista vacía de parámetros es opcional, excepto cuando se encadenan llamados, en dicho caso es obligatoria para todas las llamados excepto la última. No es posible encadenar llamados a métodos adicionales a la derecha del llamado de un procedimiento, puesto que los procedimientos no se invocan como parte de una expresión. De la misma manera, cuando se encadenan dos llamados a una función, el resultado de la primera función debe ser un objeto que puede ser pasado a la segunda función.

#### **2.4.5. Compartición de objetos**

La mayoría de los objetos del mundo real son considerablemente más grandes y complejos que el tipo Relacional. Por ejemplo, considerar los tipos de objeto del código II.20, en donde los objetos tipo *Dirección* poseen más del doble de atributos que los del tipo *Relacional* y los objetos del tipo *Persona* todavía tiene más atributos, incluyendo uno de tipo *Dirección*. Cuando objetos grandes son utilizados, resulta ineficiente enviar copias de él entre subprogramas. En dichas circunstancias es más adecuado compartir el objeto. Esto es posible si el objeto

cuenta con un identificador de objeto.

```
CREATE TYPE Direccion AS OBJECT (
    direccion_calle VARCHAR2(35) , ciudad VARCHAR2(15) ,
    estado CHAR( 2 ) , cod_postal INTEGER)
/

CREATE TYPE Persona AS OBJECT (
    nombre VARCHAR2(15) , apellido VARCHAR2(15) ,
    fecha_nac DATE, direccion_casa Direccion, //Objeto anidado
    telefono VARCHAR2(15) , numero_seg_soc INTEGER) ;
```

Código II.20: Ejemplo de objetos anidados en ORACLE

Para compartir objetos se utilizan referencias. Una referencia es un puntero al objeto. La compartición de objetos proporciona dos ventajas importantes:

- Cuando un objeto compartido es actualizado, el cambio se produce sólo en un lugar y cualquier referencia al objeto puede recuperar los valores actualizados inmediatamente.
- La información no se duplica innecesariamente.

En el código II.21 se visualizan las ventajas de compartir objetos, definiendo el tipo de objeto *Hogar* y creando una tabla que almacena las instancias de ese tipo.

```
CREATE TYPE Hogar AS OBJECT (
    direccion VARCHAR2(35), dueño VARCHAR2(25),
    estilo VARCHAR(15), precio REAL(9,2)) /

CREATE TABLE hogares OF Hogar;
```

Código II.21: Ejemplo tabla que almacena objetos en ORACLE

## Utilización de referencias

Con el objeto de tipo *Persona* del ejemplo del código II.20, se puede diseñar una comunidad que pueda compartir la misma casa (Hogar). Para ello se puede utilizar el modificador de tipo REF, ver código II.22, el cual declara una referencia (almacena un puntero al objeto). Es relevante destacar cómo las referencias entre personas y Hogares y entre personas entre sí definen relaciones que existen en el mundo real.

```
CREATE TYPE Persona AS OBJECT (
  nombre VARCHAR2(15) ,
  apellido VARCHAR2(15) ,
  fecha_nac DATE,
  direccion_casa REF Home, //Compartido con la familia
  telefono VARCHAR2(15) ,
  numero_seg_soc INTEGER
  madre REF Persona , // Miembros de la familia
  padre REF Persona ,
  ...) ;
/
```

Código II.22: Ejemplo de modificador REF en objetos en ORACLE

Es posible declarar referencias como variables, parámetros, campos o atributos. También, se pueden utilizar referencias como parámetros IN y OUT en funciones y procedimientos. Sin embargo, no es posible navegar a través de referencias. En el ejemplo del código II.23 se muestra un intento ilegal de navegar a través de una referencia a un objeto. Para poder realizar esta operación es preciso utilizar el operador Deref, a través del cual se puede acceder al objeto.



```
DECLARE
  persona_ref REF Persona;
  num_telefono VARCHAR2(15) ;
BEGIN
  ...
  num_telefono = persona_ref.telefono; // es ilegal !
```

Código II.23: Ejemplo de utilización errónea de REF en objetos ORACLE

### Limitaciones en la definición de tipos

En la creación de un tipo solamente es posible hacer referencia a objetos que existen en el esquema de objetos. Para poder solucionar este problema se emplea una sentencia *CREATE TYPE* especial denominada *definición previa* de tipo, la cual permite la creación de tipos de objetos mutuamente dependientes. Para resolver el problema mencionado previamente, basta con crear el objeto antes de hacer la referencia. El tipo creado a través de una definición previa de tipo tiene como nombre *tipo de objeto incompleto*, ya que carece de atributos y métodos hasta que se defina por completo.

Un tipo incompleto impuro posee atributos, pero compila con errores semánticos (no sintácticos) al hacer referencia a un tipo indefinido.

### 2.4.6. Manipulación de objetos

Es posible utilizar un tipo de objeto en la sentencia *CREATE TABLE* para especificar el tipo de una columna. Cuando la tabla se ha creado, se pueden utilizar las sentencias SQL para insertar un objeto, seleccionar sus atributos, invocar los métodos definidos y actualizar su estado. Ese preciso utilizar un alias de la tabla

cuando se hace referencia a un atributo o método. Cuando se crea un objeto de esta forma, este carece de identidad fuera de la tabla de la base de datos. Sin embargo, el tipo de objeto existe independientemente de cualquier tabla y puede ser utilizado para crear objetos mediante otros métodos.

En el código II.24 se puede observar como se crea una tabla que almacena en sus filas objetos del tipo *Relacional*. Este tipo de tablas, en las que sus filas contienen un tipo de objetos, son llamadas tablas de objetos, donde cada columna en una fila corresponde con un atributo del tipo de objeto. Cada fila en una tabla de objetos cuenta con un identificador de objeto, que identifica de forma única al objeto almacenado en esa fila y sirve como una referencia al objeto.

```
CREATE TABLE numeros_relacionales OF Racional ;
```

Código II.24: Ejemplo creación de tabla de objetos ORACLE

### Selección de objetos

En el código II.25 se crea un tipo de objeto denominado *Persona* y una tabla de objetos Personas junto con algunos valores. Por otro lado la subconsulta del código II.26 produce como resultado un conjunto de filas que tienen solamente atributos de los objetos *Persona*.

```
CREATE TYPE Persona AS OBJECT (  
    nombre VARCHAR2(15), apellido VARCHAR2(15),  
    fecha_nac DATE, direccion_casa Direccion, telefono VARCHAR2(15)) ; /  
  
CREATE TABLE Personas OF Persona; /
```

Código II.25: Ejemplo tabla de objetos ORACLE

```
INSERT INTO Empleados //Otra tabla de objetos de tipo Persona  
SELECT * FROM Personas per WHERE per.apellido LIKE ' % Tarantino';
```

Código II.26: Ejemplo de selección de objetos ORACLE

### El operador VALUE

Como su nombre lo dice, este operador devuelve el valor de un objeto. VALUE requiere como argumento una variable de correlación (para este contexto, sería una fila o alias de tabla asociado a una fila en una tabla de objetos). Para obtener un conjunto de objetos se puede utilizar el comando VALUES. Las reglas de integridad, de clave primaria, y el resto de propiedades que se definan sobre una tabla, sólo afectan a los objetos de esa tabla, es decir no se refieren a todos los objetos del tipo asignado a la tabla.

### El operador REF

Los identificadores únicos asignados por Oracle a los objetos que se almacenan en una tabla, permiten que éstos puedan ser referenciados desde los atributos de otros objetos o desde las columnas de tablas. El tipo de datos proporcionado por Oracle para soportar esta facilidad se denomina REF. Un atributo de tipo REF almacena una referencia a un objeto del tipo definido, e implementa una relación de asociación entre los dos tipos de objetos. Estas referencias se pueden utilizar para acceder a los objetos referenciados y para modificarlos; sin embargo, no es posible operar sobre ellas directamente. Para asignar o actualizar una referencia se debe utilizar siempre REF o NULL.

Cuando se define una columna de un tipo a REF, es posible restringir su dominio a los objetos que se almacenen en cierta tabla. Si la referencia no se asocia a

una tabla sino que sólo se restringe a un tipo de objeto, se podrá actualizar a una referencia a un objeto del tipo adecuado con independencia de la tabla donde se almacene. En este caso su almacenamiento requerirá más espacio y su acceso será menos eficiente.

### **Referencias colgadas (*dangling refs*)**

Cuando se borran objetos de la base de datos, puede ocurrir que otros objetos que referencien a los borrados queden en estado inconsistente. Estas referencias se denominan dangling references, y Oracle proporciona el predicado llamado IS DANGLING que permite comprobar cuándo sucede esto.

### **El operador Deref**

No es posible navegar a través de referencias en procedimientos SQL. Debido a esto se hace necesario emplear el operador Deref (derreferenciar un puntero es obtener el valor al cual este apunta). Este operador tiene como argumento una referencia a un objeto y retorna el valor de dicho objeto. Si la referencia está pendiente, Deref devuelve el valor NULL. En esta situación no es obligatorio especificar una tabla de objetos ni un criterio de búsqueda, ya que cada objeto almacenado en una tabla de objetos cuenta con un identificador de objeto único y estático que es parte de cada referencia a un objeto.

Además es posible la utilización del operador Deref en sentencias SQL sucesivas para derreferencias referencias.

En procedimientos SQL la utilización del operador Deref es ilegal. En sentencias SQL se puede utilizar la notación punto para navegar a través de referencias.

### **Inserción de objetos**

Para almacenar objetos en una tabla de objetos se utiliza el comando UPDATE. Por otro lado, además es posible utilizar la cláusula RETURNING, la cual almacena una referencia a un objeto en una variable local. Es necesario recalcar como esta cláusula simula una sentencia SELECT. La cláusula RETURNING puede ser utilizada también en sentencias UPDATE y DELETE. Para ingresar objetos en una tabla de objetos se puede emplear una consulta que retorne un objeto del mismo tipo.

### **Actualización de objetos**

Para poder modificar los atributos de un objeto en una tabla de objetos se debe utilizar la sentencia UPDATE.

### **Eliminación de objetos**

Para la eliminación de objetos (filas) en una tabla de objetos se utiliza la sentencia SQL DELETE. Para eliminar objetos selectivamente se utiliza la cláusula WHERE.

## **2.4.7. Herencia**

Desde la versión 9i de Oracle se incorpora la herencia simple de tipos, sin embargo aún no se soporta la herencia de tablas, lo cual si posee soporte en el standard SQL 2003. El tipo raíz de una jerarquía se crea empleando la sentencia CREATE TYPE y debe ser declarado como NOT FINAL. La opción por defecto es FINAL (indicando así, que pueden derivarse subtipos de él). En la figura 2.11 se muestra un ejemplo típico de una persona que se especializa en estudiante y empleado, su implementación se encuentra detallada en el código II.27.

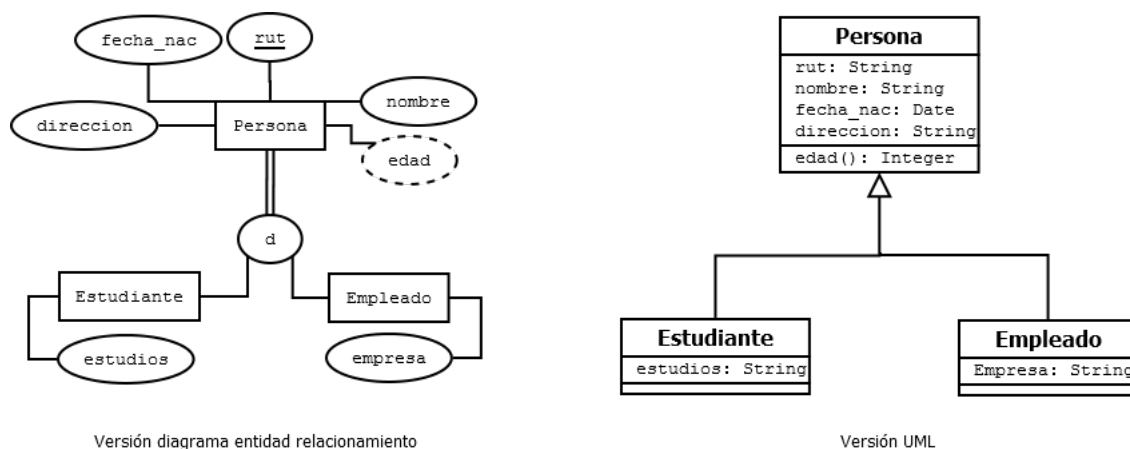


Figura 2.11. Ejemplo de herencia

```

CREATE TYPE tPersona AS OBJECT (
    rut   VARCHAR(9),
    nombre VARCHAR(25),
    fecha_nac DATE,
    direccion VARCHAR(1000),
    MEMBER FUNCTION edad RETURN NUMBER)
NOT INSTANTIABLE NOT FINAL /
CREATE TYPE tEstudiante UNDER tPersona (estudios VARCHAR(50));/
CREATE TYPE tEmpleado UNDER tPersona (empresa VARCHAR(50)); /

```

Código II.27: Ejemplo de implementación de herencia de objetos en ORACLE

La herencia simple de tipos que contempla Oracle implica que cualquier subtipo herede de su padre los métodos y atributos que este posea. Es así como para el ejemplo presentado en el código II.27, los subtipos tEstudiante y tEmpleado poseen la función edad(). Por lo tanto, esta función podrá ser llamada sobre cualquier instancia de estos tipos, es decir, cuando se defina una tabla de cualquiera de los dos subtipos, se podrá invocar a la función edad() sobre cualquiera de las filas de la tabla.

Es sumamente importante tener en cuenta que Oracle no soporta la herencia de tablas; es decir, la definición de jerarquías de tablas sobre tipos que están integrados en una jerarquía de tipos. Oracle sólo permite asegurar que los atributos y métodos del supertipo de la tabla padre se heredarán en las tablas definidas sobre los subtipos. Sin embargo, las restricciones, disparadores, etc. definidos para una tabla no podrán ser heredados por otras tablas, aunque sus tipos subyacentes pertenezcan a la misma jerarquía. En el código II.28 se muestran las sentencias para crear las tablas del tipo padre y de los dos subtipos, como se observa es necesario definir en cada tabla sus propias restricciones, ya que éstas no se propagan.

```
CREATE TABLE Persona OF tPersona
  (PRIMARY KEY (rut),
   CHECK(direccion like ( %Tocopilla %)));

CREATE TABLE Estudiante OF tEstudiante
  (PRIMARY KEY (nombre));

CREATE TABLE Empleado OF tEmpleado
  (PRIMARY KEY (rut));
```

Código II.28: Ejemplo de creación de tabla del tipo padre y de subtipos ORACLE

Las tablas Estudiante y Empleado definidas sobre subtipos del tipo tPersona no deben cumplir en absoluto las restricciones impuestas para la tabla Persona definida sobre el tipo padre tPersona. Así, por ejemplo, se puede insertar en la tabla Empleado una fila en la que el valor del atributo dirección no contenga la cadena “Tocopilla”, mientras que en la tabla Persona eso sería imposible. Del mismo modo, se puede definir el campo nombre como clave primaria en la tabla Estudiante,

aunque en la tabla padre Persona la clave primaria se definió sobre el campo rut. Pero, además de que las tablas no hereden las restricciones de la tabla definida para el supertipo, existe un problema aun mayor y este se presenta en el ejemplo del código II.28, ya que no se puede rescatar el hecho implícito que existe en toda jerarquía, de que todo estudiante es persona y de que todo empleado es persona. Esto se debe a que no existe relación alguna entre las tablas, no hay nada que le indique al SABD que un empleado o un alumno es además una persona. Por ello, al realizar una consulta a persona, se obtendría sólo aquellas personas que no fueran ni estudiantes ni empleados.

De esta forma, para la implementación de una jerarquía en Oracle, aunque es posible apoyarse en ocasiones en la utilización de la herencia de tipos, se necesita además recurrir a los clásicos mecanismos empleados en las bases de datos relacionales (claves foráneas, o referencias, entre las tablas, restricciones, vistas, entre otros.).

#### **2.4.8. Valores y funciones XML**

La base de datos XML de Oracle, referencia a una colección de tecnologías XML nativas en el servidor de base de datos. Sin embargo, el soporte nativo XML es sólo una parte de la infraestructura XML en Oracle. La infraestructura XML general proporciona soporte XML nativo de alto rendimiento y una plataforma extensible en el que los usuarios pueden crear y desplegar sus propias soluciones. En el motor nativo XML, las tablas y vistas XMLType ofrecen el almacenamiento de datos XML. El repositorio de base de datos XML proporciona un repositorio de documentos XML que se ha optimizado para el manejo de documentos XML. PL/SQL y las funciones SQL/XML permiten operaciones de XML en datos SQL y contenido XML.



## **Tipo de dato XML**

Este mapeado de la estructura XML a la estructura de la base de datos en Oracle se realiza con el tipo XMLType, que es un tipo abstracto. El tipo XMLType se almacena en un tipo CLOB, aunque puede asociarse a un esquema XML para la definición de su estructura lo que obliga que cualquier documento sea validado con este esquema. En este segundo caso el esquema del documento se modela en la estructura objeto relacional de la base de datos. La ventaja de hacerlo de la primera manera es que todo tipo de documentos XML pueden almacenarse en ese elemento XMLType. La segunda obliga a que el elemento sea válido frente al esquema asociado, aunque su mapeado en la estructura objeto relacional permite tratar el documento de manera más eficiente y flexible.

## **Mapeado de XMLType dado un esquema XML**

Los elementos del esquema XML son mapeados como objetos en los que cada elemento anidado de tipo simple es representado por un atributo de un tipo nativo lo más acorde posible con el tipo del esquema por ejemplo si es un número con NUMBER, si es texto con VARCHAR, etc. Aun así es posible forzar la representación del elemento a un tipo de Oracle mediante el atributo SQLType utilizado en el elemento del esquema. Cuando un elemento contiene un elemento complejo, este es modelado con un objeto y el elemento padre establece una referencia a él con tipos referencia. Es posible forzar que el mapeado de los tipos complejos se realice en CLOB, NCLOB o VARCHAR (sin ser representados en el modelo objeto relacional) mediante el atributo SQLType (=CLOB) utilizado en el elemento del esquema. Cuando la ocurrencia de un elemento, ya sea simple o complejo, es mayor que uno el elemento es representado en el objeto padre con un array variable si el número de ocurrencias máximas es finito o con un tabla anidada si es infinito.

## Crear tables y/o columnas XMLType

Para crear columnas o tables XMLType se hace de la misma forma que al definir columnas o tables de objetos. Para definir una columna o tabla XMLType asociada a un esquema se debe registrar primero el esquema en la base de datos. Esto se realiza mediante la librería DBMS\_XMLSCHEMA que posee dos funciones: registerSchema para registrar el esquema y deleteSchema para eliminar el registro. Una vez registrado el esquema se puede crear columnas y tables XMLType asociadas empleando el comando XMLSCHEMA.

## Insertar documentos XML

Si se tratan a los tipos XMLType como objetos se puede utilizar el constructor de dichos objetos para instanciar nuevos elementos XMLType, tomando como parámetro la cadena que representa al documento XML. En el código II.29 se presenta un ejemplo de utilización.

### **INSERT INTO** warehouses **VALUES**

```
( 100, XMLType(  
  '<Warehouse whNo="100">  
    <Building>Owned</Building>  
  </Warehouse>'), 'Tower Records', 1003);
```

### **UPDATE** warehouses **SET** warehouse\_spec = XMLType

```
(' <Warehouse whono="200">  
  <Building>Leased</Building>  
</Warehouse>');
```

Código II.29: Ejemplo de inserción de datos utilizando XMLTYPE en ORACLE

## Consultar documentos XML

Es posible rescatar un documento en forma de CLOB, VARCHAR o NUMBER mediante los métodos de XMLType: getClobVal, getStringVal, getNumberVal. Con estas funciones simplemente se obtiene el documento XML convertido en un tipo nativo. Las consultas no sólo son de recuperación de documentos completos, es posible además recuperar partes del documento y efectuar predicados de selección en partes del documento. Estas partes están basadas en la estructura DOM de XML y se señalan haciendo uso de XPath. Las funciones incluidas con este propósito son extract y existsNode: el primero devuelve el nodo del documento XML (de la estructura DOM) solicitado y el segundo devuelve verdadero (1) cuando existe el nodo solicitado.

El comando extract siempre devuelve el nodo en un tipo XMLType, si se desea recuperar el valor del nodo de texto de ese nodo se puede utilizar getNumberVal o getStringVal sobre el elemento XMLType retornado. También se puede utilizar extractValue que tiene una sintaxis idéntica a extract pero que devuelve el valor del nodo de texto y no el elemento XMLType. Estas funciones sólo son válidas para nodos que tengan un solo y único nodo de texto. El comando updateXML permite actualizar el valor de algunos nodos señalados del documento XML, para evitar de esa manera modificar todo el documento cuando sólo varía parte de él. Sus parámetros son parejas de rutas XPath y valores, donde la ruta señala el nodo a modificar y los valores sustituirán a los antiguos de ese nodo.

El comando XMLTransform toma como parámetros dos instancias de XMLType siendo la primera el documento de origen y la segunda un documento XSLT de transformaciones XML y devuelve el documento resultante de la transformación XML. Un sinónimo de este comando es el método XMLTransform de la clase XMLType. Es posible validar documentos XML frente a esquemas XML mediante el comando XMLIsValid y el método de XMLType isSchemaValidated. Ambos

devuelven verdadero (1) si el documento se valida correctamente.

## **SQLX**

Al igual que en el modelado objeto relacional, en el que no es necesario convertir los datos en el modelo plano relacional al modelo objeto relacional para trabajar con ellos en este último modelo, es posible mediante comandos y vistas representar datos del modelo relacional u objeto relacional como documentos XML sin la necesidad de modificarlos.

Oracle soporta cinco comandos del estándar SQLX (SQL to XML, SQL/XML) para la representación de datos relacionales con XML: XMLElement, XMLForest, XMLConcat, XMLAttributes y XMLAgg. También soporta XMLColAttVal como comando SQLX propio, pero aún no aceptado en el estándar. Estos comandos permiten representar datos como un documento XML cuya estructura de ese documento es definida por los desarrolladores.

Oracle, también, soporta las funciones SYS\_XMLGEN, SYS\_XMLAGG, XMLSEQUENCE y XMLFormat con el mismo propósito que las anteriores pero sin ser parte del estándar SQLX o de su propuesta. Es posible además crear vistas del tipo XMLType para representar tablas y vistas relacionales como documentos XML de forma transparente para la consulta, como si de una consulta a un XMLType se tratase.

XMLElement es una función que devuelve un tipo XMLType dados como parámetros el nombre del elemento XML, una serie de atributos y el contenido del nodo. El XMLType retornado es un nodo con el nombre del primer parámetro, los atributos del segundo y el contenido de los últimos parámetros. El contenido puede ser un valor o un nuevo elemento XMLType para poder formar la estructura anidada de los documentos XML.

Los atributos se definen mediante la función XMLAttributes que toman como método el listado de atributos a asignar al elemento XML. Si no se especifica la cláusula AS en cada atributo se deja como nombre de atributo el inferido de la estructura relacional, si se utiliza AS se deja el indicado.

La función XMLForest crea un árbol XML de los parámetros que toma. Un árbol XML son nodos situados a la misma altura, esto significa nodos que partirían del mismo nodo raíz, salvo que no se haya definido este nodo raíz. Cuando el parámetro es acompañado de la cláusula AS, se éste es utilizado como nombre de elemento XML, cuando no se infiere de la estructura de los datos.

La función XMLConcat, concatena los parámetros dados uno tras otro en el orden en que aparecen como parámetros, estos pueden ser una secuencia de elementos XMLTYPE o datos tipo XMLType. Mientras que en XMLForest los parámetros son datos relacionales, en XMLConcat son tipos XMLType.

La función XMLAgg es una función de agregado que produce un bosque de elementos XML dada una colección de elementos. Se usa normalmente con consultas con cláusulas de agrupación como GROUP BY.

La función XMLColAttVal crea un árbol de XML donde cada elemento es de tipo column y posee un atributo tipo name con el nombre del elemento, especificado por AS en los parámetros o inferido de los datos.

La función SYS\_XMLAGG engloba todos los documentos XML o fragmentos de una expresión en un solo documento XML. La etiqueta que engloba es por defecto ROWSET, pero puede ser definida con XMLFormat.

XMLSEQUENCE devuelve una secuencia (array variable) de XMLType dado un XMLType. En consecuencia, toma los nodos hijo directos del XMLType y devuelve un nodo XMLType por cada uno de ellos en un objeto XMLSequenceType.

La función SYS\_XMLGEN toma un tipo nativo, un tipo abstracto o un tipo XMLTy-

pe y genera con él un documento XML. Si es un tipo nativo forma una etiqueta con el valor dentro, si es un tipo abstracto mapea los atributos del tipo abstracto a un documento XML y si es un XMLType engloba a este elemento en otro elemento de nombre por defecto ROW. Es posible indicar el nombre de la etiqueta principal del documento XML generado mediante la función XMLFormat.

El objeto XMLFormat es un parámetro de SYS\_XMLGEN y SYS\_XMLAGG. Este objeto define las características del documento generado por estas dos funciones mediante sus atributos. Si se desea cambiar el formato del documento XML generado tan solo se tendrá que dar el valor adecuado al correspondiente atributo de XMLFormat.

### **Vistas XMLType**

Las vistas XMLType permiten tomar elementos relacionales u objeto relacionales de la base de datos, sin modificar ni los datos ni su estructura para poder mostrarlos como si fuesen documentos XML. Mediante la creación de vistas habitual se crea una vista indicando que es de tipo XMLType (OF XMLTYPE), la cláusula OBJECT ID indica que columna será el identificador único de cada elemento y que el tipo XMLType se almacenará en la columna sys\_nc\_rowinfo\$.

Además es posible crear vistas XMLType mapeando los datos relacionales mediante un esquema y no con el comando SELECT de la definición de la vista. El esquema define el mapeado de cada elemento a la columna de datos mediante el atributo xdb:SQLName en el elemento del esquema, de tal manera que el elemento contendrá el valor de la columna indicada en ese atributo.

En el anexo A.4.2 se presentan ejemplos de cada sección vista de XML.

## Diseñando la base de datos XML

Cuando se comienza a diseñar aplicaciones con Oracle XML DB, se necesita tomar varias decisiones, incluyendo como almacenar los datos XML en la base de datos, cuál es la estrategia para recuperar o generar los datos XML, y cómo crear índices apropiados para buscar el contenido en documentos XML.

### Cómo almacenar datos XML

Existen tres caminos diferentes para almacenar documentos dentro Oracle, y cada uno de ellos ofrece compensaciones en rendimiento y funcionalidad. En la figura 2.12 se presenta un diagrama de flujo que sirve de ayuda para simplificar la selección. En el anexo A.4.3 se detalla en profundidad los pasos mencionados en dicho diagrama.

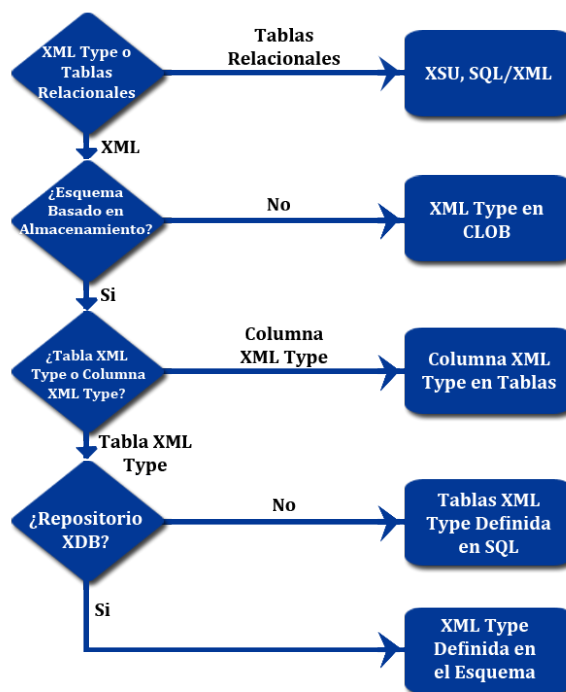


Figura 2.12. Diagrama de flujo de decisión para almacenar XML en Oracle [5]

# **CAPÍTULO III**

## **CASO DE ESTUDIO**

En este capítulo se presenta un caso de estudio que permite implementar la teoría en los diferentes modelos. Así mismo, gracias a este caso de estudio se pretende demostrar la importancia de la utilización del modelo objeto relacional y XML, ya que, mientras el modelo relacional es muy sencillo de implementar por ser el más utilizado, el objeto relacional posee una mayor semántica que se hace muy familiar si se tiene conocimientos del paradigma orientación al objeto.

### **3.1. Descripción del Caso de Estudio**

Este caso de estudio, es un caso real al cual la memorista se vió enfrentada, no obstante su implementación fue llevada a cabo en una base de datos relacional pura.

Se debe desarrollar e integrar un nuevo módulo para un sistema de gestión escolar, dicho módulo consiste en implementar las planificaciones curriculares de los profesores según curso y asignatura, a modo de automatizar el proceso actual que se realiza en planillas en papel.

Un nivel de enseñanza es el nivel al que pertenece un curso, por ejemplo 1º Básico, 2º Básico, etc. Un nivel de enseñanza posee un identificador y un nombre.



Como es sabido un curso (8°A) tiene un profesor jefe, año (el 8°A del 2013), asignaturas y alumnos. Por ejemplo, el 8°A tiene como profesor jefe a “Nelson Núñez”, cabe mencionar que todos los octavos tienen las mismas asignaturas. En el módulo sólo es necesario tener conocimiento sobre profesores y asignaturas.

Una asignatura posee un id y nombre. Considere que cada asignatura tiene un id distinto para cada nivel por ejemplo: Lenguaje y Comunicación del nivel de enseñanza 7° Básico tiene un código distinto que para los 8° Básicos. Una asignatura es dictada en todos los cursos del mismo nivel de enseñanza (8°A, 8°B, 8°C, etc.), además por cada curso asignatura existe un profesor. Por ejemplo, en el 8°A la asignatura “Matemáticas” es dictada por John Pérez y Educación Física por Robert Rodríguez y matemáticas para el 8°B es dictada por Juan Soto. Del profesor interesa el rut, nombres, apellido paterno, apellido materno, fecha de nacimiento, cantidad de cursos en los que hace clase.

Por otro lado existe el concepto de “tipo de programa” el cual tiene su identificador, nombre, y abreviatura. Existen 5 tipos de programas base:

- Unidades educativas.
- Objetivos fundamentales verticales.
- Objetivos fundamentales transversales.
- Aprendizajes esperados.
- Contenidos mínimos obligatorios.

Un tipo de programa posee muchos detalles de programa (curricular), el cual tiene un identificador, descripción, nombre y asignatura asociada. Por ejemplo, para los 8° básicos la asignatura “Matemáticas” tiene 2 unidades educativas (recuerde que una unidad educativa es uno de los 5 programas tipo), abreviadas con la letra U, donde cada una de ellas tiene un nombre y una descripción:

- Unidad educativa 1: geometría
- Unidad educativa 2: álgebra.

Cabe mencionar que c/u de ellas, geometría y álgebra, corresponde a un detalle del programa, c/u con un código y con un programa tipo asociado (unidad educativa).

Siguiendo con el ejemplo del nivel 8° básico, la asignatura “Educación Física” tiene 2 contenidos mínimos obligatorios, abreviados con la sigla CMO, donde cada uno de ellos tiene un nombre y descripción:

- Contenido mínimo obligatorio 1: Voleibol.
- Contenido mínimo obligatorio 2: Fútbol.

Cada profesor debe hacer su planificación por cada curso. De una planificación es importante conocer: id, fecha de creación, última modificación y tipo de planificación. Una planificación puede ser del tipo:

- Planificación anual.
- Planificación por unidad.
- Planificación clase a clase.

En una planificación del tipo anual se pueden crear muchos periodos, un periodo posee mes inicio, mes de fin, duración (en horas). Por ejemplo para el periodo 1 se empezará en abril y se terminará en junio y tendrá una duración de 50 hrs, para el periodo 2 se empezará en agosto y se terminará en octubre y este periodo tendrá una duración de 40 hrs.

En una planificación del tipo por unidad se pueden crear muchas unidades y una unidad posee fecha de inicio y fecha de término. Por ejemplo la unidad 1 empezará el 1 de abril y terminará el 30 de abril del 2013, la unidad 2 empezará el 2 de Mayo y terminará el 31 de mayo del 2013.

Por último, de una planificación del tipo clase a clase se pueden crear muchas clases y estas poseen fecha, horario (día de la semana, hora de inicio, hora de termino y duración).

En cada periodo, unidad o clase un alumno debe poseer ciertos conocimientos los cuales son llamados “Aprendizajes previos”, por ejemplo en 8° A para la asignatura Matemáticas, la unidad 1 que empieza el 2 de Mayo del 2013 y finaliza el 30 de Junio del 2013 posee los siguientes aprendizajes previos:

- Ángulos en polígonos.
- Construcción de polígonos.
- Áreas en triángulos y cuadriláteros.
- Características de conos, cilindros y pirámides.

En cada clase existe la posibilidad de especificar las actividades a desarrollar en dicha clase, una actividad posee un id, nombre, descripción y duración. Una actividad puede utilizar recursos de los cuales importa su id, nombre y descripción. Y por último una actividad emplea una metodología en particular, de una metodología se necesita conocer su id, nombre y descripción. Por ejemplo:

En 8° B para la asignatura Matemáticas, una de las actividades a desarrollar en la clase del 30 de Mayo del 2013 será explicar las diferencias entre círculo y circunferencia, utilizando el concepto de lugar geométrico. Esta actividad utiliza como recurso cartulina y balones de fútbol. Se emplea metodología activa.

Por otro lado, tanto para un periodo como para una unidad se deben especificar las metodologías y los recursos a utilizar.

Además cada periodo, unidad y clase puede tener muchos detalles de programas (curriculares) asociados, el profesor selecciona todos los programas que correspondan. Ejemplo:

Una planificación anual del curso 8°C para la asignatura "Matemáticas", tiene asociado el aprendizaje esperado AE1 y AE3 para dicha asignatura, los cuales tienen la descripción "Caracterizar la circunferencia y el círculo como lugares geométricos" y "Calcular el área del círculo y de sectores de él"

Una planificación posee un último estado, aunque la planificación puede pasar por varios estados. Los estados pueden ser: "borrador", "pendiente", "rechazado" y "validado", se debe saber en qué fecha cambió de estado. De un estado se debe saber id y nombre. Una planificación cuando es creada queda automáticamente en estado borrador y el profesor una vez finalizada la creación la debe enviar a validar, cuando la envía a validar queda en un estado "pendiente". La persona encargada de revisar cada una de las planificaciones que los profesores envían es un No docente. De un No docente se debe saber su rut, nombres, apellido paterno, apellido materno, fecha de nacimiento y cargo. Un cargo tiene id, nombre y descripción. Cada vez que un No docente acepta o rechaza una planificación puede escribir un comentario y debe quedar registrada la fecha en que se realizó la acción. Cuando la planificación es rechazada el profesor tiene la posibilidad de realizar los cambios solicitados y enviarla nuevamente a validación, y esta puede ser validada por otro No docente.

## 3.2. Diagrama entidad relacionamiento

En la figura 3.1 se presenta el diagrama entidad relacionamiento, en donde solamente se muestran las entidades y relacionamientos, el detalle de los atributos y su respectiva documentación se encuentra en el Anexo B.1.

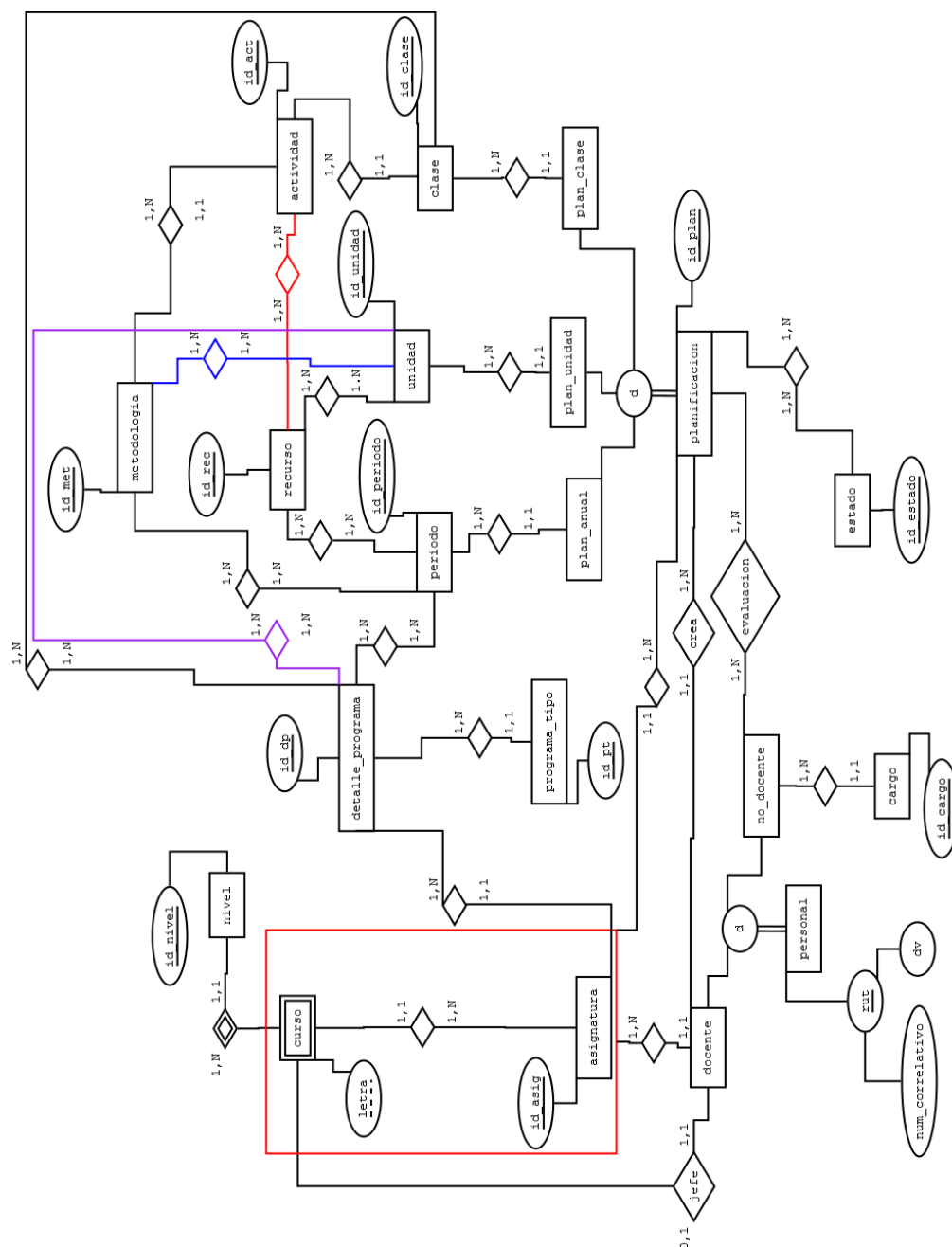


Figura 3.1. Diagrama entidad relación

### 3.3. Modelo relacional

A continuación se presenta el modelo relacional del caso de estudio. Para realizar la transformación del modelo entidad-relacionamiento al modelo relacional se utilizó la opción 8 a mencionada en [2] para las especializaciones y/o generalizaciones.

Los códigos SQL DDL (Data Definition Language, por sus siglas en inglés) de la implementación y la documentación de tablas se encuentran detallados en el Anexo B.3 y Anexo B.2 respectivamente.

**nivel**(id\_nivel, nom\_nivel)

**curso**(id\_nivel, letra, año, num\_correlativo, dv)

**programa\_tipo**(id\_pt, nom\_pt, abbrev)

**detalle\_programa**(id\_dp, nom\_dp, desc\_dp, id\_pt, id\_asig)

**personal**(num\_correlativo, dv, nombres, apaterno, amaterno, fnac, calle, num, cod\_postal, ciudad, region)

**personalmail**(num\_correlativo, dv, mail)

**personalcelular**(num\_correlativo, dv, celular)

**docente**(num\_correlativo, dv)

**no\_docente**(num\_correlativo, dv, id\_cargo)

**asignatura**(id\_asig, nom\_asig, id\_nivel, letra, num\_correlativo, dv)

**cargo**(id\_cargo, nom\_c, desc\_c)

**planificacion**(id\_plan, ult\_estado, fcreacion, ult\_modif, tipo\_plan, num\_correlativo, dv, id\_asig)

**plan\_anual**(id\_plan)

**plan\_unidad**(id\_plan)

**plan\_clase**(id\_plan)

**periodo**(id\_periodo, duracion, mes\_ini, mes\_ter, id\_plan\_anual)

**periodo\_aprevios**(id\_periodo, aprevio)

**unidad**(id\_unidad, fechai, fechate, id\_plan\_unidad)

**unidad\_aprevios**(id\_unidad, aprevio)

**clase**(id\_clase, duracion, dia, hi, ht, fecha, id\_plan\_clase)

**clase\_aprevios**(id\_clase, aprevio)

**estado**(id\_estado, nom\_estado)

**planificacion\_estado**(id\_plan, id\_estado)

**planificacion\_estado\_fcambio**(id\_plan, id\_estado, fcambio)

**evaluacion**(num\_correlativo, dv, id\_plan)

**evaluacion\_detalle**(num\_correlativo, dv, id\_plan, fecha\_eva, comentario)

**metodologia**(id\_met, nom\_met, desc\_met)

**metodologia\_unidad**(id\_met, id\_unidad)

**metodologia\_periodo**(id\_met, id\_periodo)

**recurso**(id\_rec, nom\_rec, desc\_rec)

**recurso\_unidad**(id\_rec, id\_unidad)

**recurso\_periodo**(id\_rec, id\_periodo)

**recurso\_actividad**(id\_rec, id\_act)

**actividad**(id\_act, desc\_act, duracion, id\_clase, id\_met)

**detalle\_programa\_unidad**(id\_dp,id\_unidad)

**detalle\_programa\_periodo**(id\_dp,id\_periodo)

**detalle\_programa\_clase**(id\_dp,id\_clase)



### 3.4. Notación UML para el modelo entidad relacionamiento

Para un mayor entendimiento de cómo se debe implementar el modelo objeto relacional es importante tener una visión del caso de estudio en notación UML, ya que es más sencillo de visualizar y de transformar. Pero primero se especificará la representación propuesta por [2] de entidad relacionamiento a UML.

En los diagramas de clase UML, una clase (equivalente a un tipo de entidad en ER) se muestra como un cuadro (véase la figura 3.2) que incluye tres secciones: la sección superior ofrece el nombre de la clase; la sección intermedia incluye los atributos de los objetos individuales de la clase; y la última sección incluye las operaciones que se pueden aplicar a esos objetos. En los diagramas ER no se especifican las operaciones. Un atributo compuesto se modela como un dominio estructurado. Un atributo multivaluado generalmente se modelará como una clase separada o como un atributo del tipo arreglo que contenga los valores.

En la tecnología UML, los tipos de relación se denominan asociaciones y las instancias de relación, vínculos. Una asociación binaria (tipo de relación binaria) se representa como una línea que conecta las clases participantes (tipos de entidad) y, opcionalmente, puede tener un nombre. Un atributo de relación, denominado atributo de vínculo, se coloca en un recuadro conectado con la línea de la asociación mediante una línea discontinua. La notación (mín, máx) se utiliza para especificar las restricciones de relación, que en terminología UML se denominan multiplicidades. Las multiplicidades se especifican como mín .. máx, y un asterisco (\*) indica que no hay un límite máximo en la participación. No obstante, las multiplicidades se colocan en los extremos opuestos de la relación. En UML, un asterisco indica una multiplicidad de 0 ..\*, y un 1 indica una multiplicidad de 1..1.

En UML, hay dos tipos de relaciones: asociación y agregación. La agregación es-

tá pensada para representar una relación entre un objeto completo y sus partes constitutivas, y tiene una notación diagramática distinta. No obstante la agregación y la asociación no tienen propiedades estructurales diferentes y la elección del tipo de relación que hay que utilizar es algo subjetivo. En el modelo ER, las dos se representan como relaciones.

UML también distingue entre asociaciones (o agregaciones) unidireccionales y bidireccionales. En el caso unidireccional, la línea que conecta las clases se muestra con una flecha para indicar que sólo se necesita una dirección para acceder a los objetos relacionados. Si no aparece una flecha, se asume la cualidad bidireccional, que es lo predeterminado. Por otro lado, para representar una entidad débil se utiliza una composición ya que esta posee una dependencia existencial con respecto a la entidad fuerte. En la figura 3.2 se muestra una versión simplificada UML del caso de estudio.

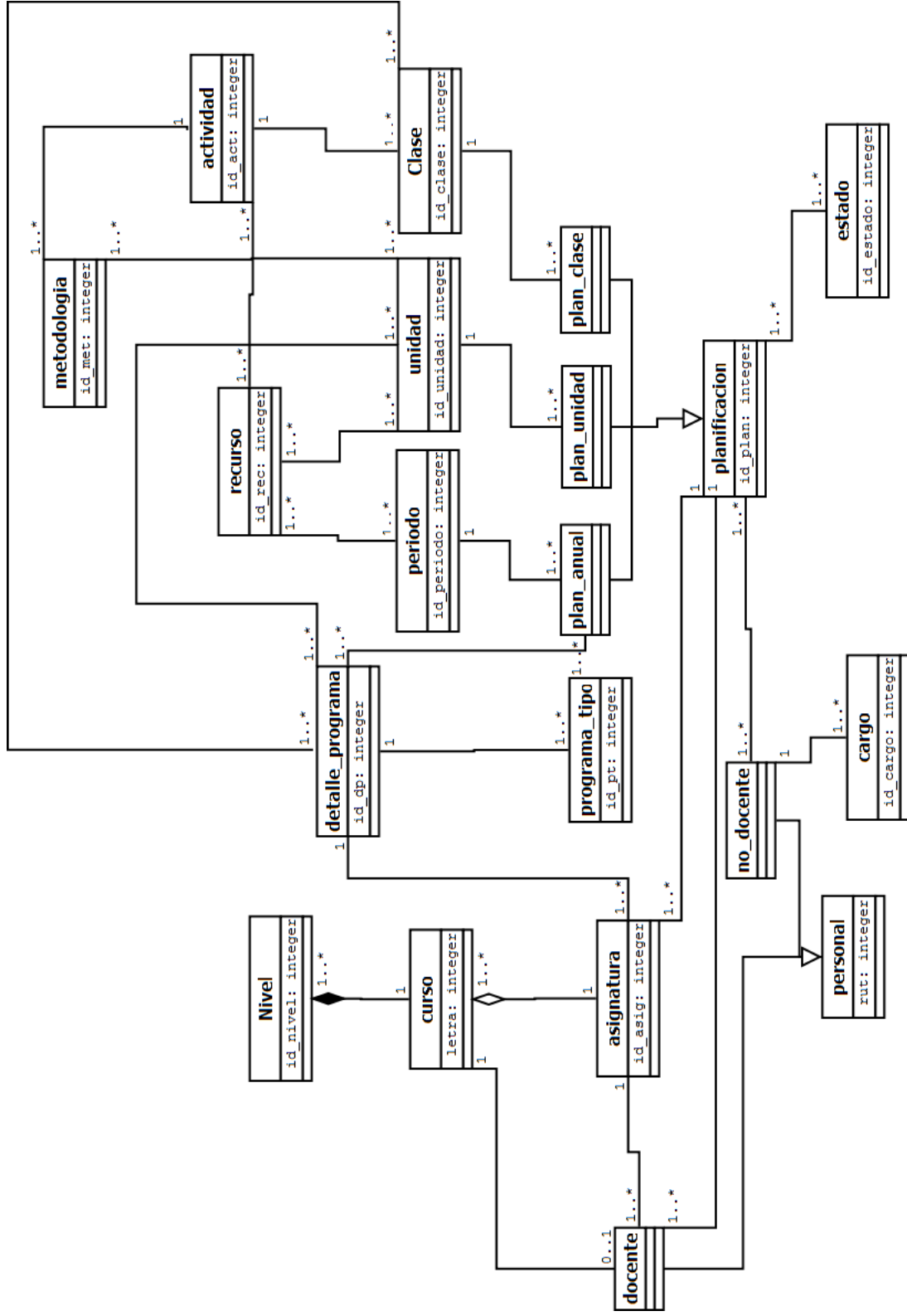


Figura 3.2. Diagrama UML del modelo relacional

## 3.5. Implementación modelo objeto relacional

### 3.5.1. Diseño de una base de datos objeto relacional utilizando la transformación de ER a objeto relacional

En la siguiente sección se presentará una descripción de un algoritmo propuesto que puede transformar un esquema ER en el esquema de una base de datos objeto relacional.

#### **Algoritmo propuesto para la transformación de ER a objeto relacional en Oracle 11G**

A continuación, se describirán los pasos de un algoritmo para la transformación de ER en objeto relacional. Se empleará la base de datos del caso de estudio. El esquema ER de esta base de datos se muestra en la figura 3.1, mientras que en la figura 3.2 se muestra el esquema de la base de datos objeto relacional correspondiente para ilustrar los pasos del mapeado.

#### **Paso 1: Mapeado de atributos compuestos**

Por cada atributo compuesto **C** del esquema ER, se crea un tipo estructurado **et** que contenga todos los atributos simples de **C**. Los tipos estructurados son útiles para representar atributos compuestos cuando dicho atributo debe ser utilizado en múltiples tablas o definiciones de tipos. Por ejemplo, el atributo dirección que se ilustra en la figura 3.3 se definirá como tipo estructurado. En el código III.1 se ve el resultado de la transformación.

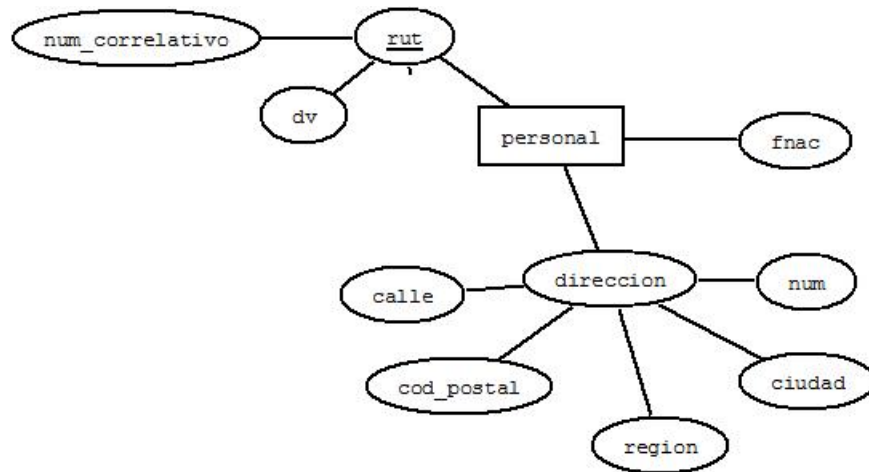


Figura 3.3. Entidad Personal con atributo compuesto

```

CREATE TYPE tDireccion AS OBJECT
(  calle varchar2(255), num varchar2(10),
  cod_postal number(10),
  ciudad varchar2(150),
  region varchar2(100));

CREATE TYPE tRut AS OBJECT (
num_correlativo number(10), dv char(1));

CREATE TYPE tPersonal AS OBJECT
(  rut tRut,
  fnac date, direccion tDireccion);

CREATE TABLE Personal OF tPersonal
(  primary key(rut.num_correlativo, rut.dv));

```

Código III.1: Definición de tipo estructurado para atributo compuesto

Es preciso definir los dos casos posibles de implementación, tanto para atributos multivaluados, como para relacionamientos de 1 : N y relacionamientos de N : M.

- En caso de que se conozca el tamaño del relacionamiento o del atributo multivaluado, por ejemplo:
  - Un persona tiene a lo más 2 teléfonos fijos.
  - Un alumno puede pertenecer como máximo a 3 carreras distintas dentro de la univesidad.
- En el caso de no conocer el tamaño máximo.

### **¿Qué utilizar dependiendo del caso?**

- Cuando el valor máximo es conocido, se propone utilizar un varray para almacenar referencias de los tipos de objetos.
- Cuando el valor no se conoce, se utiliza tablas anidadas, donde se almacenan referencias del tipo de objeto.
- Existe la posibilidad de utilizar un varray aunque se desconozca el valor máximo, en este caso se crea un varray con un tamaño máximo de tope a criterio del programador. Para realizar la manipulación de datos, es necesario crear procedimientos almacenados o métodos dentro del objeto que sean capaces de verificar si el varray tiene espacio disponible para insertar nuevos datos.

Sin embargo, la propuesta anterior que en teoría es la solución correcta para cualquier caso, si el problema es muy complejo y las entidades poseen una gran cantidad de relacionamientos Oracle, no permite su correcto funcionamiento y a

la vez mantener la navegabilidad bidireccional. Por este motivo para los relacionamientos 1 a N se utilizará la opción de crear un varray con un tamaño máximo a criterio. En el caso de un relacionamiento de N a N se propone crear un nuevo tipo objeto intersección, el cual posee una referencia a cada objeto con el que se tiene un relacionamiento, para luego crear un listado (varray) de referencias a ese nuevo objeto intersección y es dicho listado el que se agrega a cada objeto del relacionamiento. En la figura 3.4 y figura 3.5 se muestra un ejemplo del tipo intersección y el listado de referencias.

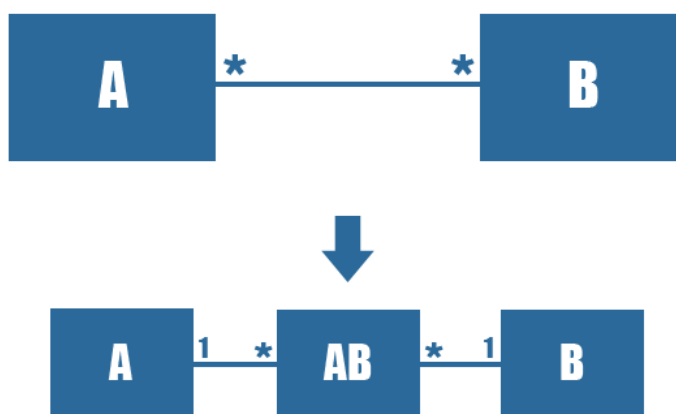


Figura 3.4. Ejemplo objetos relacionamiento N : N

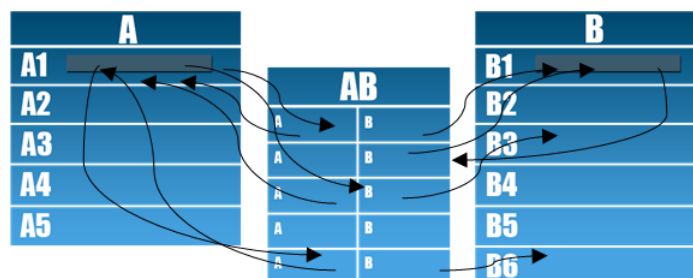


Figura 3.5. Ejemplo objeto intersección relacionamiento N : N

Para el caso de estudio será implementado con navegabilidad bidireccional por requerimiento del profesor guía, ya que cualquier objeto debe conocer con quien se encuentra asociado.

Cabe destacar, que desde el estándar SQL 2003 se tiene soporte a MULTISSET para reemplazar a los arreglos y es la forma en cómo se implementan los relacionamientos en dicho estándar [14]. En la tabla III-1 se presenta la diferencia entre Oracle y el estándar 2003.

Modelo conceptual	Estándar SQL:2003	Oracle
Clase	Tipo estructurado	Tipo Objeto
Extensión de clase	Tabla Tipada	Tabla de Tipo Objeto
Atributo Multivaluado	Array / Multiset	Varray/Nested Table
Atributo Compuesto	Row / Columna de Tipo Estructurado	Columna de Tipo Objeto
Atributo Derivado	Trigger/Método	Trigger/Método
Asociación 1 : 1	Ref / Ref	Ref / Ref
Asociación 1 : N	Ref / Multiset / Array	Ref / Nested Table / Varray
Asociación N : M	Multiset / Multiset o Array / Array	Nested Table / Nested Table o Varray / Varray
Asociación Agregación	Multiset / Array	Nested Table / Varray de Referencias
Asociación Composición	Multiset / Array	Nested Table / Varray de Objetos
Asociación Generalización	Tipos / Tablas Tipadas	Tipos / Tablas de Tipo Objeto.

Tabla III-1: Reglas de transformación (Modelo conceptual - Modelo Objeto relacional) [14]

## Paso 2: Mapeado de atributos multivaluados

Por cada atributo multivaluado del esquema ER, existen dos métodos para realizar el mapeado: sí el tamaño del atributo multivaluado es conocido *MV* y sí se desconoce el tamaño del atributo *MT*. Para *MV* se puede representar directamente como un atributo de un tipo objeto **et** para la tabla de tipos **c**, en donde se



define un varray **V** de referencias de un tipo de datos específico. Si es un atributo multivaluado compuesto, crear un **V** que contenga un tipo estructurado con los atributos simples de *MV*. Por ejemplo en el atributo celular que se ilustra en la figura 3.6, definirá como tipo varray, se supondrá que a lo más se almacenarán 5 números, en el código III.2 se ve el resultado de la transformación.

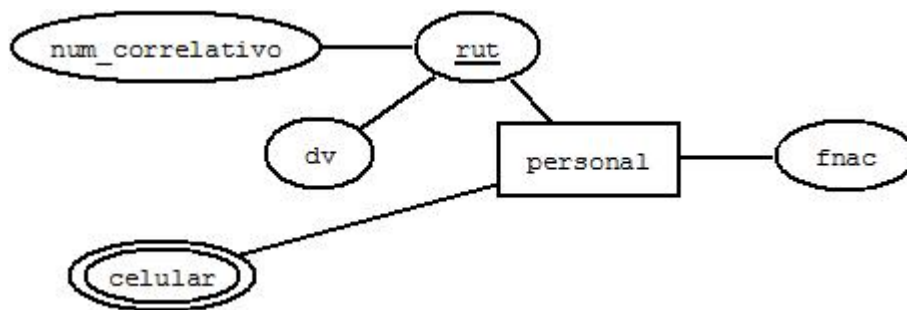


Figura 3.6. Entidad Personal con atributo multivaluado

```
CREATE TYPE tCelular AS OBJECT ( num_celular number(20));
CREATE TYPE Celulares AS VARRAY(5) OF REF tCelular;
CREATE TYPE tRut AS OBJECT ( num_correlativo number(10), dv char(1));

CREATE TYPE tPersonal AS OBJECT
( rut tRut,
  fnac date,
  celular Celulares);

CREATE TABLE Personal OF tPersonal
( primary key(rut.num_correlativo,rut.dv));
```

Código III.2: Definición de tipo varray para atributo multivaluado

### Paso 3: Mapeado de atributos derivados

Por cada atributo derivado *D* del esquema ER, se define como un método en el

tipo estructurado **et** que realice el cálculo requerido y devuelva su valor. Por ejemplo en el atributo edad que se ilustra en la figura 3.7, se definirá como método. En el código III.3 se ve el resultado de la transformación.

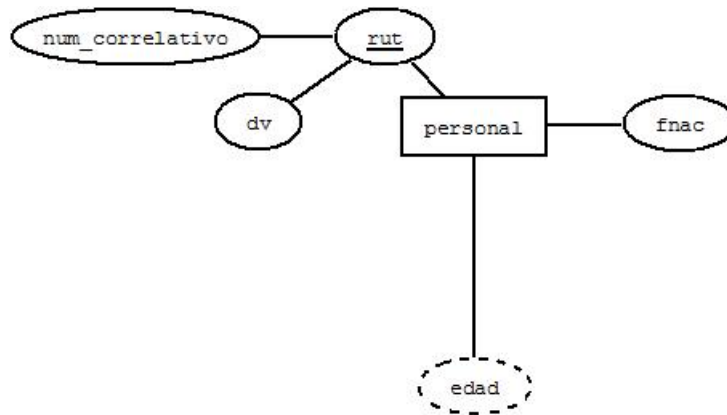


Figura 3.7. Entidad Personal con atributo derivado

```

CREATE TYPE tRut AS OBJECT (num_correlativo number(10), dv char(1));
CREATE TYPE tPersonal AS OBJECT
(  rut   tRut,  fnac date,
  MEMBER FUNCTION edad RETURN NUMBER)
/
CREATE TYPE BODY tPersonal IS
  MEMBER function edad RETURN NUMBER IS
    BEGIN
      return(trunc(months_between(Sysdate, self.fnac) / 12));
    END;
  END; /

CREATE TABLE Personal OF tPersonal
(  primary key(rut.num_correlativo, rut.dv));
  
```

Código III.3: Definición de método para atributo derivado

#### Paso 4: Mapeado de los tipos de relación 1:1 binaria

Por cada tipo de relación 1:1 binaria **R** sin atributos del esquema ER, identifique las relaciones **S** y **T** que corresponden a los tipos de entidad que participan en **R**. Se define una tabla de tipos **S1** que posee un tipo estructurado **s1t** con un atributo de tipo referencia (t1t). Además definir una tabla de tipos **T1** que posee un tipo estructurado **t1t** con un atributo de tipo referencia (s1t).

#### Paso 5: Mapeado de tipos de relaciones 1:N binarias

Por cada relación 1:N binaria regular **R** sin atributos, identifique la relación **S** que representa el tipo de entidad participante en el lado N del tipo de relación. Al igual que en el mapeado de los tipos de relación 1:1 sin atributos, se define una tabla de tipos **b** que posee un tipo objeto **bt** con un atributo del tipo Varray que contiene referencias del tipo objeto del lado N. Por ejemplo en el relacionamiento que se ilustra en la figura 3.8, se definirán los tipos tActividad y tMetodologia, en donde tActividad tiene una referencia a la metodología que utiliza, en cambio para representar las actividades que han utilizado cierta metodología se emplea un Varray que almacena referencias de dichas actividades. En el código III.4 se ve el resultado de la transformación.

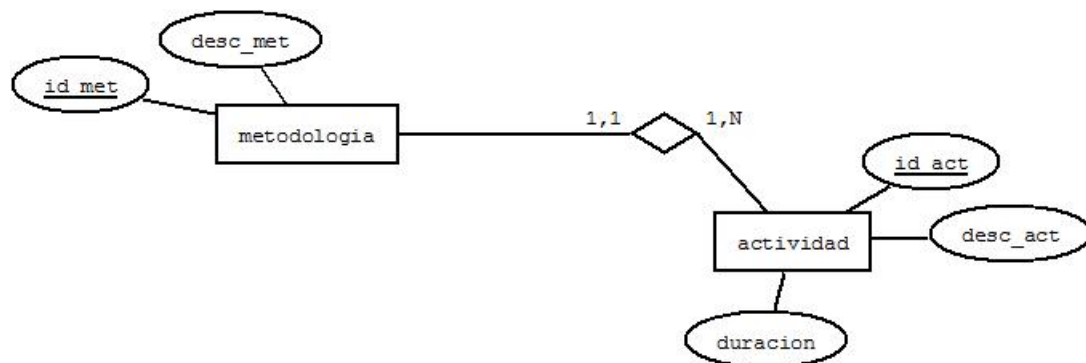


Figura 3.8. Relacionamiento 1 a N

```

CREATE TYPE tActividad AS OBJECT (
id_act   varchar2(10),
desc_act varchar2(255),
duracion integer,
met_usada REF metodologia);

CREATE TYPE ListadoActividades AS VARRAY(50) OF REF tActividades;

CREATE TABLE Actividad OF tActividad (PRIMARY KEY (id_act));

CREATE TYPE tMetodologia AS OBJECT
(id_met   varchar2(10),
desc_met varchar2(255),
actividades ListadoActividades);

CREATE TABLE Metodologia OF tMetodologia
(PRIMARY KEY (id_met));

```

Código III.4: Definición de tipos para relacionamiento 1 a N

### Paso 6: Mapeado de tipos de relaciones M:N binarias

Por cada tipo de relación M:N binaria **R** sin atributos, se debe definir un atributo del tipo objeto **to** que contiene referencias del tipo objeto M y del tipo objeto N. Se le debe agregar a cada tipo objeto un varray de referencias del tipo **to**, para luego crear una nueva tabla **tt** de tipos **to**.

En caso de que **R** tenga atributos se debe crear una nueva tabla de tipo **S** para representar a **R**. La tabla de relación **S** incluye un tipo objeto **to** que contiene referencias de cada tipo objeto de la tabla de tipo correspondiente, que participa

en la relación y además los atributos que posea la relación, al igual que en el relacionamiento sin atributos, a cada objeto se le debe agregar un atributo varray con referencias al tipo objeto **to**.

Por ejemplo en el relacionamiento que se ilustra en la figura 3.9, se definirán los tipos tActividad y tRecurso, en donde se crea el tipo objeto y luego una tabla que almacena referencias de actividades y recursos. En el código III.5 se ve el resultado de la transformación. Otro ejemplo se visualiza en la figura 3.10, donde existe un relacionamiento M:N con atributos, en el código III.6 se ve el resultado de su implementación.

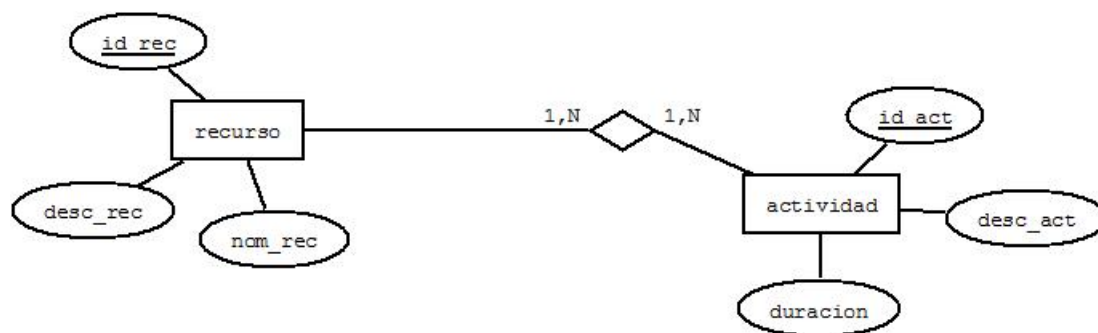


Figura 3.9. Relacionamiento N a N

```

CREATE TYPE tActividad AS OBJECT (
  id_act    varchar2(10), desc_act varchar2(255),
  duracion integer,);

CREATE TYPE tRecurso AS OBJECT (
  id_rec    varchar2(10), nom_rec varchar2(100),
  desc_rec varchar2(255));

CREATE TYPE Trecurso_actividad AS OBJECT(
  actividad  REF tActividad, recurso  REF tRecurso);

CREATE TYPE listado_recurso_actividad AS VARRAY(50) OF REF
  Trecurso_actividad;

ALTER TYPE tRecurso ADD ATTRIBUTE (listado_actividades
listado_recurso_actividad) CASCADE;

ALTER TYPE tActividad ADD ATTRIBUTE (listado_recursos
listado_recurso_actividad) CASCADE;

CREATE TABLE Actividad OF tActividad ( PRIMARY KEY (id_act)); /

CREATE TABLE Recurso OF tRecurso ( PRIMARY KEY (id_rec)); /

CREATE TABLE Recurso_Actividad OF Trecurso_actividad (
PRIMARY KEY (actividad.id_act,recurso.id_rec));

```

Código III.5: Definición de tipos para relacionamiento N a N

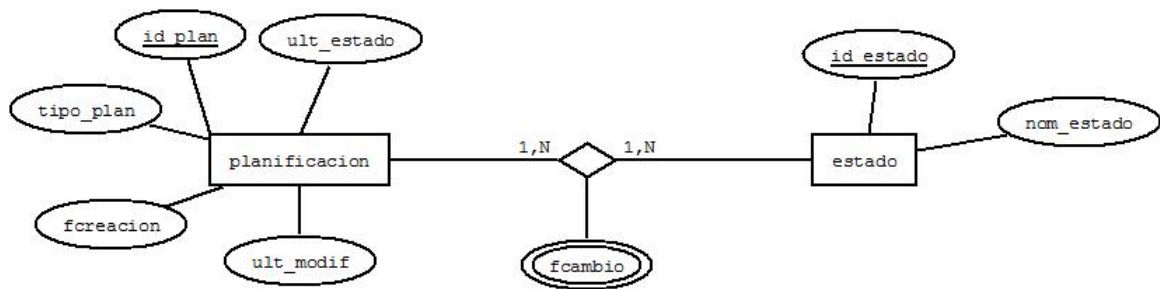


Figura 3.10. Relacionamiento N a N con atributos

```
CREATE TYPE tPlanificacion AS OBJECT (
```

```
id_plan varchar2(10),
```

```
tipo_plan varchar2(10),
```

```
fcreacion date,
```

```
ult_estado varchar2(10),
```

```
ult_modif varchar2(10));
```

```
CREATE TYPE tEstado AS OBJECT (
```

```
id_estado varchar2(10),
```

```
nom_estado varchar(50));
```

```
CREATE TYPE fcambio AS OBJECT (
```

```
fecha_cambio date);
```

```
CREATE TYPE fecha_cambio AS VARRAY(10) OF fcambio;
```

```
CREATE TYPE tPlanificacion_Estado_fcambio AS OBJECT (
```

```
planificacion ref tPlanificacion ,
```

```
estado ref tEstado,
```

```
fecha_c fecha_cambio);
```

```
CREATE TYPE listado_planificacion_estado AS VARRAY(50) OF REF
tPlanificacion_Estado_fcambio;
```

```
ALTER TYPE tPlanificacion ADD ATTRIBUTE (listado_estados
listado_planificacion_estado) CASCADE;
```

```
ALTER TYPE tEstado ADD ATTRIBUTE (listado_planificaciones
listado_planificacion_estado) CASCADE;
```

```
CREATE TABLE Planificacion OF tPlanificacion
(PRIMARY KEY (id_plan));
```

```
CREATE TABLE Estado OF tEstado
(PRIMARY KEY (id_estado));
```

```
CREATE TABLE Planificacion_Estado OF
tPlanificacion_Estado_fcambio (PRIMARY KEY
( planificacion .id_plan,estado.id_estado));/
```

Código III.6: Definición de tipos para relacionamiento N a N con atributos

### **Paso 7: Mapeado de tipos de relaciones N arias**

Por cada tipo de relación N aaria **R**, se debe crear una nueva tabla de tipo **S** para representar a **R**. La tabla de relación **S** existen N atributos con tipo referencia definidos, uno para el tipo objeto de cada tabla de tipo que participa en la relación.



### Paso 8: Mapeado de tipos de relaciones recursivas

Se debe tratar como cualquier relacionamiento.

### Paso 9: Mapeado de los tipos de entidad regulares

Por cada entidad (fuerte) regular E del esquema ER, se define una tabla de tipos **S1** que posee un tipo objeto **s1t** que incluya todos los atributos simples, compuestos, multivalor y derivados de E. Un ejemplo se puede visualizar en la figura 3.3 y su implementación en el código III.1.

### Paso 10: Mapeado de los tipos de entidad débiles

Por cada tipo de entidad débil W del esquema ER con el tipo de entidad propietario E, crear una tabla de tipo objeto para W con una clave compuesta que incluya la clave de la tabla tipo objeto de su propietario E. El tipo objeto para la tabla de tipo de W puede además incluir un atributo del tipo referencia que contiene una fila de la tabla de tipo que representa a E. Para el ejemplo de la figura 3.11 se definirán los tipos estructurados nivel\_ob y curso\_ob, luego se define el varray de tipos con referencias a curso que contiene los N cursos que posee el nivel. En el código III.7 se detalla la implementación completa.

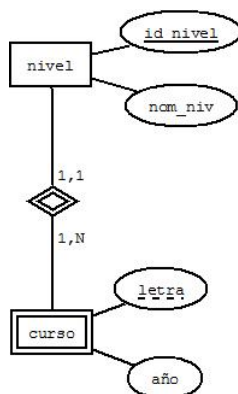


Figura 3.11. Entidad débil

```

CREATE TYPE nivel_ob AS OBJECT
( id_nivel  NUMBER(10),
  nom_nivel varchar2(255));

CREATE TYPE curso_ob AS OBJECT
( refnivel  REF nivel_ob,
  id_nivel  NUMBER(10),
  letra  CHAR(1),
  year_ DATE );

CREATE TYPE Listado_Cursos AS VARRAY(50) OF REF curso_ob;

ALTER TYPE nivel_ob ADD ATTRIBUTE (cursos Listado_Cursos) CASCADE;

CREATE TABLE nivel OF nivel_ob (
PRIMARY KEY (ID_NIVEL));

CREATE TABLE curso OF curso_ob (
id_nivel  SCOPE IS NIVEL,
PRIMARY KEY (id_nivel,letra));

```

Código III.7: Definición de tipos para definición de entidad debil

### Paso 11: Mapeado de la especialización o generalización

Para mapear una cierta cantidad de subclases que juntas forman una especialización (o, alternativamente, que están generalizadas en una subclase), como las subclases PLAN\_ANUAL, PLAN\_UNIDAD, PLAN\_CLASE de PLANIFICACION de la figura 3.12. Se debe convertir cada especialización con **M** subclases {S1,

$S_2, \dots, S_m$  y la superclase (generalizada) **C**, en donde se debe definir un tipo objeto **Ct** para la superclase **C** y luego definir **M** tipos objetos **S<sub>t</sub>** de las subclases indicando que se heredan de la superclase **Ct**. En el código III.8 se detalla el proceso de definición y se visualiza el resultado de la transformación.

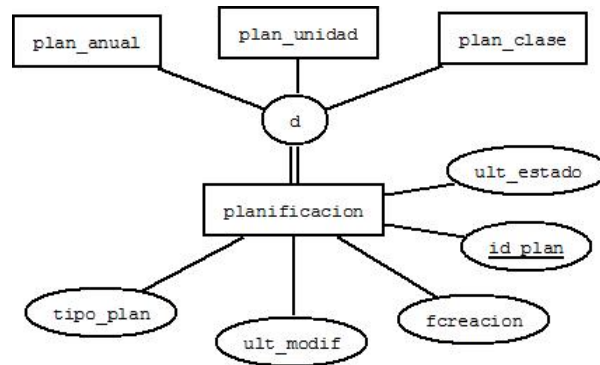


Figura 3.12. Generalización de entidades

```
CREATE TYPE tPlanificacion AS OBJECT
( id_plan  varchar2(10),
  tipo_plan varchar2(10),
  fcreacion date,
  ult_estado varchar2(10),
  ult_modif varchar2(10))
NOT FINAL
/

CREATE TABLE Planificacion OF tPlanificacion
(PRIMARY KEY (id_plan))
/

CREATE TYPE tPlan_anual UNDER tPlanificacion(); /
CREATE TYPE tPlan_unidad UNDER tPlanificacion(); /
CREATE TYPE tPlan_clase UNDER tPlanificacion(); /
```

Código III.8: Definición de tipos para generalización

### Paso 12: Mapeado de agregaciones

Para mapear una agregación dependerá netamente de las entidades y relacionamientos en su interior. Si por ejemplo la agregación es como la de figura 3.13 (la cual se cambió del problema original, para hacer más sencilla la implementación del ejemplo) se debe tratar como un relacionamiento 1 : N y su implementación se detalla en el código III.9. En el caso de que el relacionamiento de la figura 3.13 sea M : N o 1 : 1 ver los códigos III.10 y III.11 respectivamente.

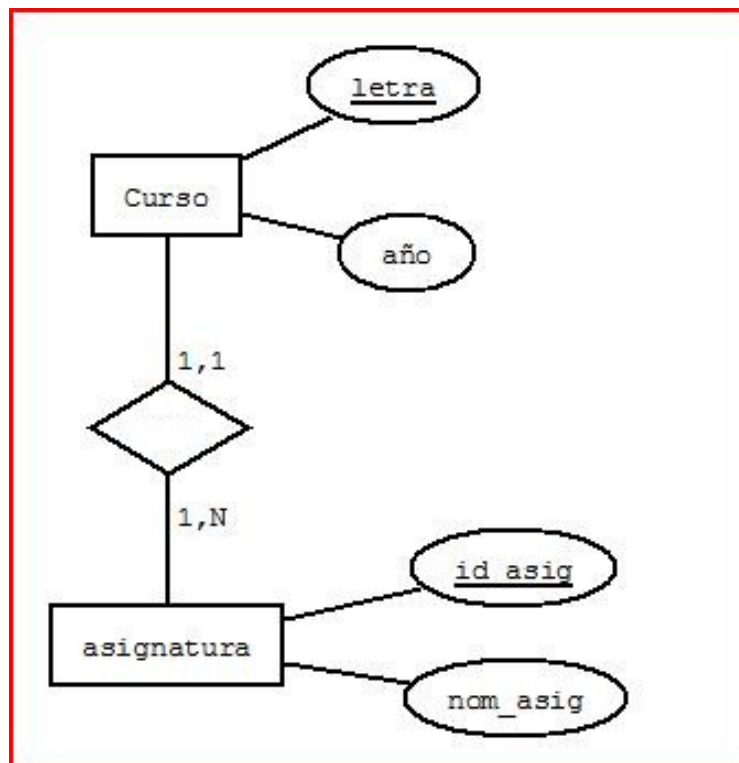


Figura 3.13. Agregaciones

```

CREATE TYPE curso_ob AS OBJECT (letra CHAR(1), year_ DATE ));

CREATE TYPE asignatura_ob AS OBJECT (id_asig number(10),
    nom_asig varchar2(255), curso_dicta ref curso_ob);

CREATE TYPE listado_asignaturas AS VARRAY(50) OF REF asignatura_ob;

ALTER TYPE curso_ob ADD ATTRIBUTE (asignaturas listado_asignaturas)
CASCADE;

CREATE TABLE curso OF curso_ob ( PRIMARY KEY (letra));

CREATE TABLE asignatura OF asignatura_ob (PRIMARY KEY (id_asig));

```

Código III.9: Definición de tipos Agregaciones con relacionamiento 1 : N

```

CREATE TYPE curso_ob AS OBJECT (letra CHAR(1), year_ DATE )) /

CREATE TYPE asignatura_ob AS OBJECT (
id_asig number(10), nom_asig varchar2(255));

CREATE TYPE curso_asignatura AS OBJECT (
curso ref curso_ob, asignatura ref asignatura_ob);

CREATE TYPE listado_cursos_asignaturas AS VARRAY(100)
OF curso_asignatura;

ALTER TYPE curso_ob ADD ATTRIBUTE (
asignaturas listado_cursos_asignaturas) CASCADE;

```

```
ALTER TYPE asignatura_ob ADD ATTRIBUTE (
cursos listado_cursos_asignaturas) CASCADE;
```

```
CREATE TABLE curso OF curso_ob ( PRIMARY KEY (letra));
```

```
CREATE TABLE asignatura OF asignatura_ob (PRIMARY KEY (id_asig));
```

Código III.10: Definición de tipos Agregaciones con relacionamiento M : N

```
CREATE TYPE curso_ob AS OBJECT
```

```
( letra CHAR(1),
  year_ DATE );
```

```
CREATE TYPE asignatura_ob AS OBJECT
```

```
( id_asig number(10),
  nom_asig varchar2(255),
  curso_dicta ref curso_ob);
```

```
ALTER TYPE curso_ob ADD ATTRIBUTE (asignatura REF asignatura_ob)
CASCADE;
```

```
CREATE TABLE curso OF curso_ob ( PRIMARY KEY (letra));
```

```
CREATE TABLE asignatura OF asignatura_ob (PRIMARY KEY (id_asig));
```

Código III.11: Definición de tipos Agregaciones con relacionamiento 1 : 1

### **3.5.2. Restricciones del problema para su implementación**

Se definirán algunas restricciones para presentar las distintas versiones de implementar el caso de estudio en el modelo objeto relacional. Ya que al conocer el tamaño exacto de cada relacionamiento se hace más sencilla su implementación.

Las restricciones son las siguientes:

- Cada personal a lo más tendrá 5 celulares y mails asociados.
- Cada periodo, unidad y clase a lo más tendrá 10 aprendizajes previos asociados.
- Cada actividad de una clase utiliza a lo más 3 metodologías educativas.
- Cada clase a lo más puede realizar 5 actividades.

Los códigos SQL DDL (Data Definition Language) de los tipos y tablas de tipos se encuentran detallados en el Anexo B.4 y B.5 respectivamente.

## **3.6. Implementación de modelo de datos semi estructurado**

Primero se extrajo una parte del caso de estudio, para luego modificarlo o agregarle un nuevo requerimiento con el propósito de lograr realizar una implementación apropiada del problema utilizando datos semi estructurados.

En esta sección se pretende realzar la utilidad de emplear los datos semi estructurados en conjunto con el modelo objeto relacional en Oracle, para esto se presentarán algunos ejemplos representados en el caso de estudio, para así ver la real ganancia en problemas del mundo real.

### 3.6.1. Nuevo requerimiento

Para el caso de estudio descrito con anterioridad, es necesario almacenar la información educacional (establecimientos donde cada persona ha estudiado), idiomas que este domina y cursos de perfeccionamiento que haya realizado cada personal del establecimiento, con el fin de disponer de mayor información curricular de cada empleado. Dado a lo anterior, se agregará un atributo currículum a cada personal, con el propósito de almacenar dicha información.

Como la información curricular de cada persona varía, ya sea por la profesión o la cantidad de estudios que este posea, se sugiere utilizar las capacidades de XML dentro de Oracle como la solución más viable para almacenar dicha información semi estructurada. Para lograr este objeto, el nuevo atributo currículum deberá ser del tipo XMLType.

La estructura de cada documento XML ingresado en dicha columna debe ser validado previamente. En el anexo B.6 se presenta en detalle el proceso de validación y registro de esquema en Oracle.

En el código III.12 se muestra el esquema XML del currículum de una persona, el cual ha sido registrado en Oracle mediante la función DBMS\_XMLSCHEMA.

En el código III.13 se muestra cómo utilizar una columna del tipo XML que contiene todos los datos del currículum de una persona dentro de un tipo objeto. Luego en el código III.14 se realiza una inserción en la tabla `personal_ob2_xml`, con el fin de efectuar futuras pruebas de selección, actualización y eliminación en tablas con columnas XMLType.



```

<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="curriculum">
    <xs:complexType><xs:sequence>
      <xs:element name="estudios"><xs:complexType><xs:sequence>
        <xs:element name="estudio" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType><xs:sequence>
            <xs:element type="xs:string" name="nombre_establecimiento"/>
            <xs:element type="xs:string" name="carrera" minOccurs="0"/>
            <xs:element type="xs:string" name="ciudad"/>
            <xs:element type="xs:short" name="fecha_egreso"/>
          </xs:sequence>
          <xs:attribute type="xs:string" name="nivel" use="optional"/>
        </xs:complexType></xs:element>
      </xs:sequence></xs:complexType></xs:element>
      <xs:element name="cursos"><xs:complexType><xs:sequence>
        <xs:element name="curso" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType><xs:sequence>
            <xs:element type="xs:string" name="tipo"/>
            <xs:element type="xs:string" name="nombre"/>
            <xs:element type="xs:byte" name="horas"/>
          </xs:sequence>
          <xs:attribute type="xs:string" name="area" use="optional"/>
        </xs:complexType></xs:element>
      </xs:sequence></xs:complexType></xs:element>
      <xs:element name="idiomas"><xs:complexType><xs:sequence>
        <xs:element name="idioma" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType><xs:sequence>
            <xs:element type="xs:string" name="nivel"/>
          </xs:sequence>
          <xs:attribute type="xs:string" name="nombre" use="optional"/>
        </xs:complexType></xs:element>
      </xs:sequence></xs:complexType></xs:element>
    </xs:sequence></xs:complexType></xs:element></xs:schema>

```

Código III.12: Ejemplo de estructura del currículum

```

CREATE TYPE tDireccion AS OBJECT
(  calle varchar2(255), num varchar2(10),
  cod_postal number(10), ciudad varchar2(150),
  region varchar2(100));

CREATE TYPE tRut AS OBJECT (num_correlativo number(10), dv char(1));

CREATE TYPE nombre_completo_persona_ob AS OBJECT
(
  nombres varchar2(255), apaterno varchar2(255),
  amaterno varchar2(255) );

CREATE TYPE PERSONAL_OB2_XML AS OBJECT
(  rut rut_persona_ob,
  nombres nombre_completo_persona_ob,
  fnac date, direccion direccion_persona_ob,
  curriculum xmltype);

CREATE TABLE Personal OF PERSONAL_OB2_XML
(  primary key(rut.num_correlativo,rut.dv));

```

Código III.13: Columna XMLType en caso de estudio

Para verificar que efectivamente los datos insertados en la columna XMLType conformen un documento XML válido, se sugiere crear un trigger (cuya implementación se encuentra en el Anexo B.6) que se gatille cada vez que se realice una inserción o actualización en dicha columna. El tipo XMLType posee un método llamado *schemaValidate()*, el cual permite asegurar que todas las instancias almacenadas en la columna son validadas con respecto al esquema xml.

**INSERT INTO personal\_ob2\_xml values**

```
(rut_persona_ob(12345678,'4'),nombre_completo_persona_ob('ingeborg',
munoz','carnot'), '13-04-1984',direccion_persona_ob('ohiggins','22','3','
tocopilla','antofagasta'),
    curriculum(XMLTYPE('<curriculum><estudios><estudio nivel="
    Enseñanza Basica">
        <nombre_establecimiento>Carlos Condell </nombre_establecimiento>
        <ciudad>Tocopilla</ciudad>
        <fecha_egreso>1997</fecha_egreso>
    </estudio>
    <estudio nivel="Enseñanza Media">
        <nombre_establecimiento>Liceo Domingo Latrille
        </nombre_establecimiento>
        <ciudad>Tocopilla</ciudad><fecha_egreso>2001</fecha_egreso>
    </estudio>
    <estudio nivel="Enseñanza Superior">
        <nombre_establecimiento>Universidad Católica del norte
        </nombre_establecimiento>
        <carrera>Ingeniería de Ejecución en Computación e Informática</carrera>
        <ciudad>Antofagasta</ciudad><fecha_egreso>2011</fecha_egreso>
    </estudio>
</estudios>
< cursos>
    <curso area="Informática"><tipo>Diplomado</tipo>
        <nombre>Redes de datos</nombre><horas>50</horas>
    </curso>
    <curso area="Idiomas"><tipo>Capacitación</tipo>
```

```

        <nombre>Inglés intermedio</nombre><horas>100</horas>
    </curso>
</cursos>
<idiomas>
    <idioma nombre="Español"><nivel>Nativo</nivel>
</idioma>
    <idioma nombre="Ingles"><nivel>Medio</nivel>
</idioma>
</idiomas></curriculum>));

```

Código III.14: Inserción en tabla con columnas del tipo objeto y XMLTYPE

Para poder extraer datos desde la columna XMLTYPE y presentarlo en una consulta SQL, se debe especificar el nombre del XPath, de esta forma se retorna un fragmento XML. Por ejemplo en código III.15 se despliega el nombre completo y el nombre de todos los establecimientos en que estudió una persona y su resultado puede ser visualizado en la figura 3.14.

```

select p.nombres.nombres,
        p.nombres.apaterno,
        extract(curriculum, ' /curriculum/estudios/estudio/nombre_establecimiento')
        .getStringVal() "estudios"
from personalxml p where p.curriculum is not null;

```

Código III.15: Columna XMLType en caso de estudio

	NOMBRES.NOMBRES	NOMBRES.APATERNO	estudios
1	phaxiqhana	huanca	<nombre_establecimiento>Colegio alcazar</nombre_establecimiento><nombr...
2	ingeborg	munoz	<nombre_establecimiento>Carlos Condell </nombre_establecimiento><nombr...

Figura 3.14. Resultado de consulta a columna XMLTYPE (con tags)

Por otro lado, en el código III.16 se presenta la consulta en caso de que se desee listar cada lugar (como tupla) donde estudió la persona. Se puede observar su resultado en la figura 3.15. En el caso de que se quiera listar todos los nombres de establecimientos sin repetirlos, se debe incluir el comando **DISTINCT**.

```
select extractValue(x.column_value, 'estudio/nombre_establecimiento') as
      nombre_establecimiento
from personalxml p,
table(XMLSequence(extract(p.curriculum,'curriculum/estudios/estudio')) x;
```

Código III.16: Columna XMLType en caso de estudio

NOMBRE_ESTABLECIMIENTO
1 Colegio alcazar
2 Liceo Isaura dinator de Guzmán ex liceo 4
3 Universidad Católica del norte
4 Universidad Católica del norte
5 Carlos Condell
6 Liceo Domingo Latrille
7 Universidad Católica del norte

Figura 3.15. Resultado de consulta a columna XMLTYPE (Tupla por cada nodo)

Para actualizar datos dentro de una columna XMLTYPE se debe tener claro conocimiento de qué se desea actualizar exactamente, el documento completo o sólo algunos nodos según alguna restricción. Por ejemplo en código III.17 se actualiza la columna XMLTYPE con un documento nuevo.

```
UPDATE personalxml p SET p.curriculum = XMLTYPE('<curriculum><estudios>
<estudio nivel="Enseñanza Superior">
      <nombre_establecimiento>Universidad Católica del norte</
      nombre_establecimiento>
      <carrera>Ingeniería Civil Plan Común</carrera>
```

```

        <ciudad>Antofagasta</ciudad>
    </estudio>
    <estudio nivel="Enseñanza Superior">
        <nombre_establecimiento>Universidad Católica del norte</
            nombre_establecimiento>
        <carrera>Ingeniería de Ejecución en Computación e Informática</carrera>
        <ciudad>Antofagasta</ciudad><fecha_egreso>2011</fecha_egreso>
    </estudio>
</estudios>
<cursos>
    <curso area="Informática"><tipo>Diplomado</tipo>
        <nombre>Redes de datos</nombre><horas>50</horas>
    </curso>
    <curso area="Informática"><tipo>Capacitación</tipo>
        <nombre>6to WORKSHOP Iniciativa CLGRID UCN</nombre>
        <horas>30</horas></curso>
    <curso area="Informática"><tipo>Capacitación</tipo>
        <nombre>Tutorial on Distributed High Performance Computing</nombre>
        <horas>10</horas></curso>
</cursos>
<idiomas>
    <idioma nombre="Español"><nivel>Nativo</nivel></idioma>
    <idioma nombre="Ingles"><nivel>Medio</nivel></idioma>
</idiomas>
</curriculum>') WHERE p.rut.num_correlativo=12345678;

```

Código III.17: Actualización de Columna XMLType en caso de estudio

Los cambios realizados pueden ser comprobados en la figura 3.16

	NOMBRES.NOMBRES	NOMBRES.APATERNO	NOMBRE_ESTABLECIMIENTO
1	phaxiqhana	huanca	Colegio alcazar
2	phaxiqhana	huanca	Liceo Isaura dinator de Guzmán ex liceo 4
3	phaxiqhana	huanca	Universidad Católica del norte
4	phaxiqhana	huanca	Universidad Católica del norte
5	ingeborg	munoz	Universidad Católica del norte
6	ingeborg	munoz	Universidad Católica del norte

Figura 3.16. Resultado de consulta a columna XMLTYPE después de un UPDATE

Si se desea actualizar un nodo en particular dentro del documento XML, es necesario tener algún identificador. Según el esquema del documento del caso de estudio no se consideró ningún tipo de identificador, pero es posible filtrar por algún elemento. En el ejemplo del código III.18 se realiza la actualización de los nodos idioma, en la figura 3.17 se muestra el resultado de la consulta antes de la actualización y en la figura 3.18 después de la actualización.

	NOMBRE_IDIOMA
1	<idioma nombre="Español"><nivel>Nativo</nivel></idioma>
2	<idioma nombre="Ingles"><nivel>Medio</nivel></idioma>

Figura 3.17. Resultado de consulta a columna XMLTYPE antes de UPDATE de nodos

```
UPDATE personalxml p SET curriculum = updateXML
(curriculum, '/curriculum/idiomas/idioma/nivel/text()', 'Avanzado')
where existsNode(curriculum, '/curriculum/idiomas/idioma')=1
and p.rut.num_correlativo=12345678;
```

Código III.18: Actualización nodos de Columna XMLType en caso de estudio

	NOMBRE_IDIOMA
1	<idioma nombre="Español"><nivel>Avanzado</nivel></idioma>
2	<idioma nombre="Ingles"><nivel>Avanzado</nivel></idioma>

Figura 3.18. Resultado de consulta a columna XMLTYPE después de un UPDATE de nodos

Finalmente la eliminación de contenido de una columna XMLType, se trata igual que la eliminación de otro tipo de datos. Por ejemplo, en el código III.19 se eliminan todas las filas de idioma que sean del nivel “Medio”, en la figura 3.17 se muestra el resultado de la consulta antes de la eliminación y en la figura 3.19 después de la misma.

```
UPDATE personalxml p set
p.curriculum=deletexml(p.curriculum,'/curriculum/idiomas/idioma[nivel="Medio"]'
)
where p.rut.num_correlativo=12345678;
```

Código III.19: Actualización nodos de Columna XMLType en caso de estudio

	NOMBRE_IDIOMA
1	<idioma nombre="Español"><nivel>Nativo</nivel></idioma>

Figura 3.19. Resultado de consulta a columna XMLTYPE después de un DELETE de nodos

### 3.7. Comparación de resultados

Los criterios a evaluar en cada implementación, son los siguientes:

- Facilidad de modelamiento



- Complejidad de aprendizaje por parte del desarrollador y/o DBA

El modelamiento de los datos es claramente más sencillo en el modelo objeto relacional, ya que se utilizan las cualidades de la orientación al objeto, lo que permite al desarrollador codificar su aplicación sin preocuparse de las inserciones fragmentadas y redundancia de los datos (en el caso de atributos multivaluados en relacionamientos N a N). La creación de objetos complejos dentro del SQL, facilita enormemente el modelamiento, ya que existe mayor semántica y cada tipo de objeto es consistente con el mundo real, a la vez reemplazar los atributos derivados con métodos que calculen los datos necesarios, como por ejemplo la edad de una persona o la cantidad de ventas realizadas por un vendedor, entre otros. Por otro lado, al permitir capacidades tan básicas como los subtipos, posibilita la implementación de herencia de tipos, que al igual que en la orientación al objeto, se heredan los atributos y métodos.

Si se desea tener un listado de datos dentro de un objeto, estos se almacenan en un arreglo, sin la necesidad de generar una tabla exclusiva para almacenar dichos datos, además esta información puede ser desplegada con facilidad a través de una consulta, dejando atrás los joins y registros duplicados. En el código III.20 se presenta una consulta sencilla del caso de estudio en donde se despliegan todos los celulares de un No Docente (en donde los celulares son un varray del tipo celular\_persona\_ob), el resultado de dicha consulta se encuentra en la figura 3.20.

```
SELECT C.* FROM NO_DOCENTE P, TABLE (P.NUM_CELULARES) C  
WHERE P.RUT.NUM_CORRELATIVO=1234567;
```

Código III.20: Ejemplo de consulta a listado de celulares de un No Docente

	NUMERO_CELULAR
1	123131
2	454545
3	66432
4	553434

Figura 3.20. Resultado de consulta a listado de celulares de un No Docente

La gran diferencia recae en la sintaxis de cada implementación, esto es debido a la naturaleza de ambos modelos, es decir, para una consulta en el modelo relacional se hace necesario realizar el join de tablas explícitamente en las cláusulas FROM y WHERE de la sentencia; sin embargo, cuando se aprovecha la utilización de punteros en el modelo objeto relacional, es posible “navegar” hacia las demás tablas involucradas en la consulta, por lo que no es necesario especificar todas las tablas en la cláusula FROM de la sentencia, sino que basta solamente con utilizar la tabla a consultar en la cláusula FROM de la sentencia.

Para ejemplificar lo antes descrito, se presenta una consulta que se encarga de desplegar el nombre del tipo de programa, su abreviatura, nombre de la asignatura y nombre de nivel de enseñanza de dicha asignatura, de cada detalle de programa existente en la base de datos. En los códigos III.21 y III.22 se puede visualizar la consulta realizada en el modelo relacional y en el modelo objeto relacional respectivamente.

```
SELECT PT.NOM_PT, PT.ABREV, A.NOM_ASIG, N.NOM_NIVEL
FROM PROGRAMA_TIPO PT, ASIGNATURA A, NIVEL N,
      DETALLE_PROGRAMA DP
WHERE PT.ID_PT=DP.ID_PT AND DP.ID_ASIG=A.ID_ASIG
AND A.ID_NIVEL=N.ID_NIVEL;
```

Código III.21: Ejemplo consulta modelo relacional en caso de estudio

```
SELECT DP.PROGRAMA_TIPO.NOM_PT, DP.PROGRAMA_TIPO.ABREV, DP  
    .ASIGNATURA.NOM_ASIG,  
    DP.ASIGNATURA.CURSO_DICTA.REFNIVEL.NOM_NIVEL  
FROM DETALLE_PROGRAMA DP;
```

Código III.22: Ejemplo consulta modelo objeto relacional en caso de estudio

Se propone utilizar notación UML para el modelo objeto relacional, ya que trabaja directamente con objetos, atributos y sus respectivos métodos. Se sugiere utilizar el algoritmo de transformación de modelo relacional a objeto relacional detallado en la sección 3.5.1 del presente capítulo, puesto que el resultado final será acorde al modelo realizado en UML. Además según las pruebas realizadas, esta es la transformación adecuada si se desea desarrollar una aplicación que posea navegabilidad bidireccional.

Se requiere de un tiempo y dedicación por parte del desarrollador para obtener un nivel de aprendizaje adecuado, ya que puede resultar confuso y complejo en un principio, puesto que existen diferencias con el SQL tradicional el cual utiliza el modelo relacional. Pero si se tiene una base en la orientación al objeto, no debería requerir de tanto tiempo de capacitación y sólo sería necesario estudiar los conceptos y la nueva sintaxis que viene de la mano del modelo objeto relacional.

Por otro lado, el modelo relacional permite un modelado sencillo con consultas potentes. Si no se posee conocimientos en bases de datos relacionales, el tiempo de aprendizaje por parte del desarrollador es claramente inferior al del objeto relacional. En cambio sí se está familiarizado con este modelo resulta sumamente fácil implementar cualquier problema medianamente complejo del mundo real. Por el contrario, si el problema es muy grande la cantidad de tablas y relacionamientos aumentan, esto provoca que las consultas sean más complejas de escribir, incluso pueden llegar a ser confusas cuando se requiere de muchos joins

(por ejemplo entre más de 10 tablas).

Otro gran inconveniente es la creación de tablas innecesarias, como se mencionó anteriormente un atributo multivaluado en el modelo relacional, automáticamente genera una tabla, produciendo una redundancia de datos en dicha tabla, cuando solamente debería bastar con guarda un listado dentro de la misma tabla, por ejemplo el listado notas de un alumno en una asignatura en particular.

Además cabe destacar la cantidad de tablas que se generan en la implementación del caso de estudio en cada modelo, claramente en el modelo objeto relacional se disminuye la cantidad de tablas drásticamente, ya que se eliminan todas las tablas generadas por los atributos multivaluados y los relacionamientos N a N. En las figuras 3.21 y 3.22 se visualiza la cantidad de tablas del modelo relacional y el modelo objeto relacional respectivamente.

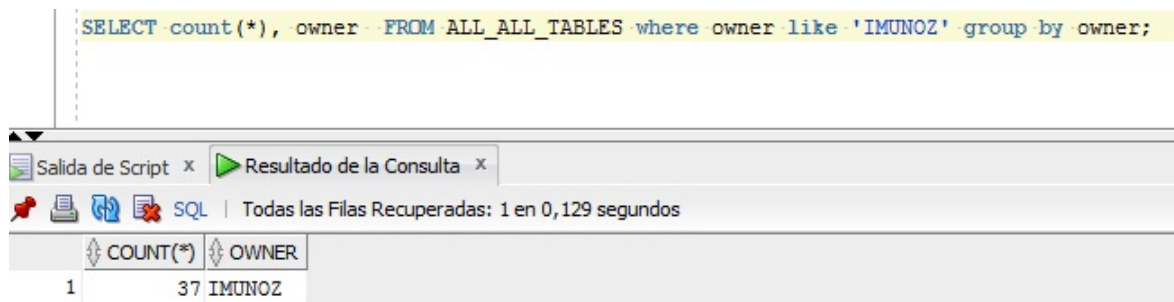
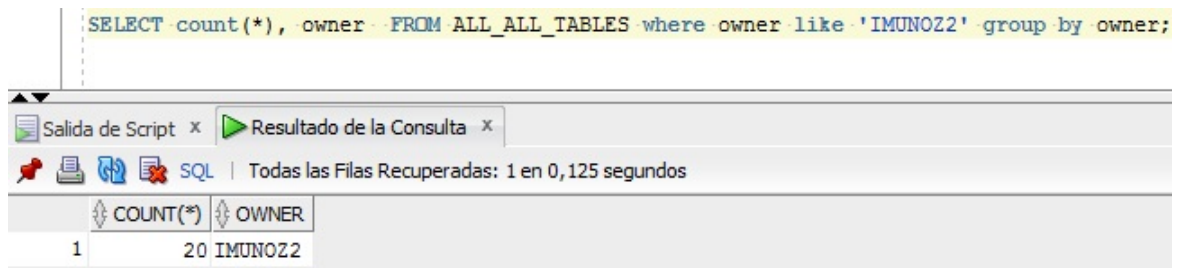


Figura 3.21. Cantidad tablas modelo relacional



The screenshot shows a database query window with the following SQL statement: `SELECT count(*), owner FROM ALL_ALL_TABLES where owner like 'IMUNOZ2' group by owner;`. The result is displayed in a table with two columns: `COUNT(*)` and `OWNER`. The first row shows a count of 20 for the owner 'IMUNOZ2'.

COUNT(*)	OWNER
20	IMUNOZ2

Figura 3.22. Cantidad tablas modelo objeto relacional

La implementación de columnas del tipo XML en una base de datos objeto relacional, es relativamente sencilla, sólo requiere de la creación del esquema del documento XML y validar con dicho esquema cada tupla que se desea insertar, con el propósito de verificar que los datos insertados efectivamente cumplen las restricciones detalladas en el esquema. En cuanto a manejo de datos sólo se debe tener claro cómo funciona XPath y obtener cierto nivel de expertiz en su funcionamiento, puesto que las funciones de extracción de datos en XML son distintas a las que se manejan en el SQL tradicional. Luego resulta simple la ejecución de consultas ya sea de selección, actualización o eliminación. Existen algunas funciones características de XML que facilitan bastante dicha labor, por lo que el tiempo de aprendizaje por parte del desarrollado es mínimo.

Aunque en este trabajo de titulación no se desarrollaron experiencias para ver tiempos de respuestas en el desempeño de las consultas, existen diversas investigaciones que indican significativas mejoras en los tiempos de respuestas, esto se ve corroborado por experiencias que se han realizado en [13] y [20]. En el anexo C se presenta una parte de los resultados.

En la figura 3.23 se presenta un resumen con el comportamiento de los criterios de cada modelo.

Criterio	Modelo relacional	Modelo objeto relacional	Datos semi estructurados
Facilidad de modelamiento	Poca semántica y por ende dificultad al modelar objetos del mundo real	Mayor semántica, facilidad al modelar e implementar objetos del mundo real	
Complejidad de aprendizaje por parte del desarrollador y/o DBA	<ul style="list-style-type: none"> <li>• Menor complejidad de aprendizaje.</li> <li>• Sencillez de implementación.</li> </ul>	<ul style="list-style-type: none"> <li>• Mayor complejidad de aprendizaje.</li> <li>• Complejidad al implementar relacionamientos bidireccionales.</li> </ul>	<ul style="list-style-type: none"> <li>• Menor complejidad de aprendizaje.</li> <li>• Sencillez de manipulación de datos y de integración con bases de datos relacionales y objeto relacionales.</li> <li>• Requiere poseer una expertiz en la utilización de las funciones ligadas a la extracción de información en XML, estas funciones son distintas a las que se manejan en SQL.</li> </ul>

Figura 3.23. Tabla resumen de criterios

En el desarrollo del caso de estudio, se resume las características del modelo relacional, modelo objeto relacional y datos semi estructurados, mostrados en las tablas III-2, III-3 y III-4 respectivamente.

Modelo Relacional	
Ventajas	Desventajas
Lenguaje de consultas potente	No soporta la navegabilidad, se debe acceder a los relacionamientos a través de Joins
Simple y puro	Poca semántica
Fácil de aprender	Fragmentación de datos y redundancia
Muy rápido de implementar	Dificultad al modelar objetos del mundo real
Sencillez al implementar relacionamientos	Tipo de datos sencillos, no permite objetos complejos

Tabla III-2: Ventajas y desventajas del modelo relacional

<b>Modelo Objeto Relacional</b>	
<b>Ventajas</b>	<b>Desventajas</b>
Semántica en el modelado	Mayor complejidad de aprendizaje
Existe navegabilidad	Mayores costos (en tiempos de implementación)
Eliminación de Joins	Complicado al momento implementar relacionamientos bidireccionales
Soporte de objetos complejos y herencia	Para implementar relacionamiento bidireccionales cuando los objetos poseen muchos relacionamientos con otros objetos, se debe generar un tipo y tabla intermediario, ya que los varray y nested tables no permiten su implementación correctamente
Reutilización y compartición	
Mejora significativamente la productividad (las consultas son más legibles)	
Preserva el cuerpo de conocimiento y experiencia alcanzado con las bases de datos relacionales	
El tamaño de las consultas disminuye al momento de consultar relacionamientos, gracias a las referencias y eliminación de joins	

Tabla III-3: Ventajas y desventajas del modelo objeto relacional

<b>Datos semi estructurados</b>	
<b>Ventajas</b>	<b>Desventajas</b>
Intercambio flexible de datos relacionales utilizando XML	Requiere comprender cómo describir la estructura y restricciones del documento xml para lograr crear esquemas XML correctamente.
Publicar datos relacionales como XML	Requiere poseer una expertiz en la utilización de las funciones ligadas a la extracción de información en XML, estas funciones son distintas a las que se manejan en SQL
Descomponer XML en datos relacionales	
Fiabilidad en la gestión de datos XML	
Manipulación, búsqueda, almacenamiento	
Integración con datos relacionales	
Permite almacenar datos XML de forma nativa en la base de datos	
Fácil de aprender	
Provee una gran flexibilidad en cuanto a la estructura sin deteriorar el desempeño	

Tabla III-4: Ventajas y desventajas de XML/SQL



## **CAPÍTULO IV**

### **CONCLUSIONES**

En este capítulo se exponen las conclusiones de este trabajo de titulación. Para ello, primero se menciona una serie de recomendaciones donde se detalla qué tipo de base de datos utilizar según la aplicación a desarrollar, luego se analiza el cumplimiento de los objetivos mencionados en el capítulo I y se describen las propuestas que se pueden llevar a cabo en el futuro a partir del trabajo desarrollado en este trabajo de titulación.

#### **4.1. Recomendaciones en la elección del tipo de base de datos para una aplicación particular**

Si se desea crear un sistema desde cero, considerando que el desarrollo será bajo el paradigma de la orientación al objeto, es muy recomendable implementar la base de datos en el modelo objeto relacional. Es posible que esto tome mayor tiempo de implementación, ya que el desarrollador primero deberá interiorizarse con el modelo, cómo se crean los tipos de objetos, los relacionamientos, referencias, tablas de objetos, entre otros. Pero obtendrá una sintaxis más limpia al momento de realizar las consultas a la base de datos. Y por consecuencia, al momento de realizar mantenimiento de dicha aplicación, resultará sumamente

sencillo de llevar a cabo porque existe una gran base de conocimiento en dicho paradigma.

Por otro lado, si se desea migrar un sistema actual que se encuentra completamente implementado en el modelo relacional, es probable que sea muy complicado, puesto que habría que realizar una transformación tanto del código desarrollado como de la base de datos. Es posible que tome un tiempo elevado, pero con buenos resultados, ya que luego de la migración el código del sistema debería ser más sencillo de mantener gracias a la semántica del modelo implementado.

Si se desea crear aplicaciones web donde existe una comunicación con otras aplicaciones codificadas en otros lenguajes de programación, se propone implementar con datos semi estructurados XML para comunicar el lenguaje de la aplicación web con las otras aplicaciones. Resulta provechoso almacenar los datos en XML, ya que es fácil de mantener y utilizar. Sólo se debe tener un cierto conocimiento en la extracción de datos XML; el desarrollo y puesta en marcha tomaría menos tiempo.

En el caso de que se desee crear aplicaciones o sistemas que no están bajo el paradigma orientación al objeto, no tendría mucho sentido utilizar el modelo objeto relacional, ya que el desarrollo sería más estructural y bastaría con utilizar el modelo relacional que para ese tipo de sistema resulta ser el más apropiado.

Si es una aplicación transaccional pequeña, como control de bodega o carrito de compras, lo más sencillo sería utilizar el modelo relacional, ya que generalmente ese tipo de aplicaciones son fáciles de modelar debido a su similitud con los objetos del mundo real, por lo que resultaría innecesaria la creación de objetos complejos.

Para problemas muy complejos, con una gran cantidad de tablas y/o relacionamientos y que deban ser implementados con urgencia en un tiempo muy corto, se recomienda emplear una base de datos relacional, puesto que su creación es

mucho más rápida que la objeto relacional y no requiere aprendizaje extra.

Para cualquier tipo de aplicación en donde se requiera de muchas consultas donde exista una gran cantidad de agrupamientos con un inmenso número de datos, es recomendable el uso del modelo objeto relacional, según los datos obtenidos en el anexo C y la sección 3.7 este modelo es mucho más eficiente que el relacional y sencillo de escribir, ya que se utilizan punteros y no existen joins.

## 4.2. Cumplimiento de los objetivos

Al comienzo de este trabajo de titulación, se propusieron una serie de objetivos específicos que eran necesarios para alcanzar el objetivo general: “Evaluar y comparar la implementación de una base de datos relacional con respecto a una base de datos objeto relacional y datos semi estructurados en el sistema de administración de base de datos Oracle, utilizando las funciones disponibles en el servidor del SABD, con el fin de generar una guía que en base a criterios claros permita determinar cuál es el tipo de base de datos más adecuada a utilizar bajo ciertas características”.

A continuación, se analizarán en qué medida se ha conseguido alcanzar cada uno de ellos:

- **Objetivo 1:** Estudiar, seleccionar y utilizar funciones del sistema de administración de base de datos relacional Oracle 11G, que permitan la implementación de bases de datos objeto relacional y XML.

Para satisfacer este objetivo, se ha llevado a cabo una labor de investigación y documentación sobre el SABD Oracle y cómo se implementan los objetos, métodos y columnas del tipo XML dentro de una base de datos objeto relacional.

- **Objetivo 2:** Comprender los múltiples métodos de implementación de los modelos mencionados con anterioridad para ser capaz de seleccionar adecuadamente entre ellos según la aplicación a desarrollar.

Sin duda esta tarea fue la más compleja, ya que las restricciones del motor hicieron que la implementación del modelo objeto relacional no fuera como se planteaba en la teoría. Sin embargo, en la sección 4.1 del presente capítulo se recomienda qué tipo de aplicaciones se deben o pueden desarrollar según cada modelo.

- **Objetivo 3:** Diseñar el caso de estudio para luego implementarlo con Oracle 11G en una base de datos relacional, objeto relacional y datos semi estructurados como XML.

- El caso de estudio fue un problema en el que la memorista se vió enfrentada en la vida real y quiso ver las múltiples ganancias de implementarlo en el modelo objeto relacional o utilizar datos semi estructurados. Pero, este resultó ser muy grande y complejo lo que repercutió en toda la fase de implementación, su gran cantidad de relacionamientos N a N, sirvió para detectar algunas falencias que existen en Oracle para implementar navegabilidad bidireccional en objetos con muchos relacionamientos N a N.
- Claramente al tener mayor familiaridad con el modelo relacional, se hace muy simple implementar el caso de estudio según el algoritmo de transformación de entidad relacionamiento a relacional [2], sin duda es un modelo muy probado y confiable, pero a la vez se queda corto en muchas funcionalidades que el objeto relacional si soluciona, la redundancia de datos y la creación de tablas innecesarias que fragmenta aún más la información, le resta puntos al momento de elegir un modelo.
- De la teoría a la implementación del modelo objeto relacional existieron

muchos obstáculos y dificultades, como por ejemplo la utilización de tablas anidadas, en la teoría esa era la opción correcta y más viable para los relacionamientos de 1 a N y N a N, sin embargo, para el caso de estudio que era altamente complejo fue imposible lograr utilizar dicha estructura de datos y mantener la navegabilidad bidireccional, ya que si la referencia de un objeto es almacenada en una tabla anidada y su estructura cambia luego de ser almacenada, la tabla anidada quedará en un estado inválido. En cambio, si la base de datos es implementada con navegabilidad unidireccional tal como se presenta en [19], se puede llevar a cabo sin inconvenientes.

- La mejor solución para dicho problema fue utilizar un varray para los relacionamientos, considerando un tamaño máximo razonable e implementando un método dentro del objeto, que verifique en cada inserción que el varray no ha llegado al tope. Esta solución es mucho más sencilla de implementar y fácil de entender, ya que su sintaxis es más limpia y menos compleja que una tabla anidada. Sin embargo, se presentó el mismo problema descrito con anterioridad al momento de mantener la navegabilidad bidireccional en relacionamientos de N a N. Para solucionar este inconveniente se creó un nuevo tipo intersección, el cual posee una referencia a cada objeto con el que se tiene un relacionamiento, para luego crear un listado (varray) de referencias a ese nuevo objeto intersección y es dicho listado el que se agrega a cada objeto del relacionamiento.
- No existieron mayores inconvenientes al implementar columnas del tipo XML, aunque se requiere poseer una expertiz en la utilización de las funciones ligadas a la extracción de información en XML, ya que estas funciones son distintas a las que se manejan en SQL tradicional.

- **Objetivo 4:** Definir criterios de comparación para las implementaciones tales como: Desempeño en consultas, facilidad de modelamiento, complejidad de aprendizaje por parte del desarrollador y/o DBA, entre otros.

En la sección 3.7 fueron presentados los criterios de comparación que se utilizaron para realizar el análisis de resultados.

- **Objetivo 5:** Realizar un análisis de resultados, en donde se compare cada una de las implementaciones, con el fin de estructurar una guía con los beneficios y desventajas dependiendo de la aplicación a desarrollar, según los criterios de comparación.

- Al igual que el ítem anterior en la sección 3.7 fueron presentados el análisis e interpretación de los resultados obtenidos según los criterios de comparación.
- Se ha podido establecer que las bases de datos objeto relacional son más sencillas de modelar, lo cual queda demostrado en la versión UML del caso de estudio. Una vez que se entiende el funcionamiento y sintaxis resulta muy fácil de utilizar. Sin embargo el aprendizaje por parte del programador establece la necesidad de tener un tiempo de capacitación, como elemento fundamental para una implementación satisfactoria, en caso contrario el tiempo de puesta en marcha se ve drásticamente aumentado.
- El mayor beneficio de utilizar el modelo objeto relacional fue simplificar enormemente las consultas, al trabajar con punteros a los objetos ya no se hace necesario hacer joins para filtrar datos. Una de las principales desventajas del modelo relacional es la utilización de joins, ya que afecta tanto su rendimiento al entregar los resultados, a mayor número de joins menos óptima es la consulta, además cuando estos son

demasiados se dificulta inmensamente la lectura y entendimiento de la consulta.

- En el caso de los datos semi estructurados, el utilizar XML se hace bastante provechoso mezclando un tipo de dato XMLTYPE dentro de una base objeto relacional, ya que en el mundo real existe información que contiene datos que no tienen una estructura predefinida, tal como fue el caso del requerimiento XML del caso de estudio, por lo que se hace necesario implementar y complementar ambas tecnologías. La manipulación de datos es bastante simple de entender y de realizar.

### **4.3. Contratiempos en la planificación**

A continuación se mencionan las dificultades encontradas en la investigación, las cuales implicaron un mayor tiempo al planificado:

- El segundo inconveniente fue implementar los relacionamientos N a N en el modelo objeto relacional. Tomó más tiempo de lo esperado lograr realizar este proceso y mantener la navegabilidad bidireccional. Se hicieron múltiples pruebas hasta llegar a la solución que finalmente cumpliera con los requerimientos.

### **4.4. Experiencia personal**

Fue una gran experiencia, descubrir que existe un mundo más allá del SQL clásico que se enseña en la universidad, en donde se puede modelar problemas del mundo real con mucha más semántica, en el cual realmente se tiene una similitud con los objetos reales. Poder aplicar los conocimientos previamente adquiridos del paradigma de la orientación al objeto y complementarlos con el SQL,

para poder generar aplicaciones potentes y sencillas de mantener.

La propuesta de transformación de modelo relacional a objeto relacional, sin duda, es uno de los mayores logros de este trabajo de titulación, ya que se encontró la mejor solución para cada caso y manteniendo la navegabilidad bidireccional para modelos complejos, a diferencia de toda la documentación citada en donde sólo mostraban soluciones para navegabilidad unidireccional o bidireccional con modelo demasiado simples.

## **4.5. Trabajo futuro**

Como trabajo futuro se pretende proponer a una empresa grande de la región, migrar uno de sus sistemas al modelo objeto relacional y XML, en primera instancia para demostrar las ganancias y beneficios que aportaría una implementación de ese estilo en sistemas grandes. Por otro lado se podría evaluar cómo cambia la teoría a la realidad y se comporta la programación orientada a objeto al utilizar este modelo, verificar la necesidad de hacer modificaciones a la capa persistencia de datos en el desarrollo de software.

Investigar y verificar cambios del modelo objeto relacional y XML en el motor Oracle en su última versión 12c y cómo esto evoluciona y se diferencia con lo presentado en esta investigación.



# Bibliografía

- [1] Oracle database 11g [en línea]  
<<http://www.oracle.com/technetwork/es/database/enterprise-edition/documentation/index.html>> [consulta: 25 marzo 2013]
- [2] Elmasri, Navathe, Fundamentos de Sistemas de Bases de Datos, Addison Wesley, 5º edición, 2007
- [3] Gavin Powell, Beginning Database Design, Wiley Publishing Inc., 2006.
- [4] Kevin Loney, Oracle database 10g: The Complete Reference, McGraw-Hill/Osborne, 2004
- [5] Mark V. Scardina, Ben Chang & Jinyu Wang, Oracle Database 10g XML & SQL Design, Build & Manage XML Applications in Java, C, C++ & PL/SQL, McGraw-Hill/Osborne, 2004
- [6] XQuery [en línea] <<http://www.w3.org/TR/xquery/>> [consulta: 25 junio 2013]
- [7] Postgresql 9.2.4 [en línea] <<http://www.postgresql.org/docs/9.2/static/index.html>> [consulta: 10 junio 2013]
- [8] Douglas, Douglas, PostgreSQL: A comprehensive guide to building, programming, and administering PostgreSQL databases, Sams Publishing, 2nd Edition, 2005

- [9] MSDN Microsoft [en línea] <[http://msdn.microsoft.com/es-es/library/bb545450\(v=msdn.10%20\).aspx](http://msdn.microsoft.com/es-es/library/bb545450(v=msdn.10%20).aspx)> [consulta: 12 junio 2013]
- [10] IBM [en línea] <<http://pic.dhe.ibm.com/infocenter/db2luw/v9r7/index.jsp>> [consulta: 12 junio 2013]
- [11] Impedance Mismatch. Kazimierz Subieta. Enero 2008. [en línea] <<http://www.sbql.pl/Topics/ImpedanceMismatch.html>> [consulta: 30 junio 2013]
- [12] Oracle® Database Object-Relational Developer's Guide 11g Release 1 (11.1) [en línea] <[http://docs.oracle.com/cd/B28359\\_01/appdev.111/b28371/toc.htm](http://docs.oracle.com/cd/B28359_01/appdev.111/b28371/toc.htm)> [consulta: 24 julio 2013]
- [13] Francisco Moreno, Guillermo Ospina, Rafael Larios. The performance of relational and object-relational SQL queries when using Oracle *Revista ingeniería e investigación Vol 25. No.3 2005*, p. 4-12, Bogotá, Diciembre 2005
- [14] Juan M. Vara, Belén Vela, José M<sup>a</sup> Cavero y Esperanza Marcos. Transformación de Modelos para el Desarrollo de Bases de Datos Objeto-Relacionales *IEEE LATIN AMERICA TRANSACTIONS, VOL. 5, NO. 4*, p. 251 - 258, Julio 2007
- [15] Migani, Silvina; Vera, Cristina; Correa, Carlos; Romera, Liliana. Almacenamiento de datos XML en Oracle 11g *XV Workshop de Investigadores en Ciencias de la Computación (WICC) 2013*, p. 152-156, Paraná, Abril 2013
- [16] Oracle® XML DB Developer's Guide 11g [en línea] <[http://docs.oracle.com/cd/B28359\\_01/appdev.111/b28369/toc.htm](http://docs.oracle.com/cd/B28359_01/appdev.111/b28369/toc.htm)> [consulta: 08 Agosto 2013]

- [17] Manejando XMLType en Oracle Database 11gR2 a través del componente XDB [en línea] <<http://www.oracle.com/technetwork/es/articles/sql/xmltype-en-database11g-a-traves-xdb-1931103-esa.html>> [consulta: 08 Agosto 2013]
- [18] Oracle Corporation, Oracle DB Performance Tuning Guide, 2003, pp.303-307, 411-544.
- [19] Suzanne W. Dietrich and Susan D. Urban, Fundamentals of Object Databases: Object-Oriented and Object-Relational Design ,Morgan & Claypool , 2011
- [20] Michael J. Carey, David J. DeWitt, Jeffrey F. Naughton, Mohammad Asgarian, Paul Brown, Johannes E. Gehrke, Dhaval N. Shah. The BUCKY Object-Relational Benchmark

# **Anexos**

## Anexo A

# Detalles, sintaxis y ejemplos de algunos motores de base de datos con soporte objeto relacional

### A.1. PostgreSQL

#### A.1.1. Sintaxis y ejemplos O-R

##### Sintaxis de CREATE TYPE

Existen diferentes formas de crear un tipo en PostgreSQL, en el código A.1 se muestra la sintaxis para crear un tipo simple con sólo un atributo, un tipo como ENUM y un tipo RANGE. En el código A.2 se visualiza una sintaxis mucho más compleja utilizando declarando funciones de entrada y salida.

```
CREATE TYPE name AS
    ( [ attribute_name data_type [ COLLATE collation ] [, ... ] ] )

CREATE TYPE name AS ENUM ( [ 'label' [, ... ] ] )

CREATE TYPE name AS RANGE (
    SUBTYPE = subtype
    [ , SUBTYPE_OPCLASS = subtype_operator_class ]
    [ , COLLATION = collation ] [ , CANONICAL = canonical_function ]
    [ , SUBTYPE_DIFF = subtype_diff_function ] )
```

Código A.1: Sintaxis CREATE TYPE en PostgreSQL

```

CREATE TYPE name (
    INPUT = input_function,
    OUTPUT = output_function
    [ , RECEIVE = receive_function ]
    [ , SEND = send_function ]
    [ , TYPMOD_IN = type_modifier_input_function ]
    [ , TYPMOD_OUT = type_modifier_output_function ]
    [ , ANALYZE = analyze_function ]
    [ , INTERNALLENGTH = { internallength | VARIABLE } ]
    [ , PASSEDBYVALUE ]
    [ , ALIGNMENT = alignment ]
    [ , STORAGE = storage ]
    [ , LIKE = like_type ]
    [ , CATEGORY = category ]
    [ , PREFERRED = preferred ]
    [ , DEFAULT = default ]
    [ , ELEMENT = element ]
    [ , DELIMITER = delimiter ]
    [ , COLLATABLE = collatable ]
)

CREATE TYPE name

```

Código A.2: Sintaxis CREATE TYPE en PostgreSQL

### Parámetros

- **name:** El nombre (opcionalmente calificado por el esquema) de un tipo a ser creado.
- **attribute\_name:** El nombre de un atributo (columna) para el tipo compuesto.
- **data\_type:** El nombre de un tipo de dato existente para convertirse en una columna del tipo compuesto.
- **collation:** El nombre de una recopilación existente asociada con una columna de un tipo compuesto, o con tipo range.
- **Label:** Una cadena literal que representa la etiqueta textual asociada a un valor de un tipo enum.
- **subtype:** El nombre de un tipo elemento que el tipo range representa.

- **subtype\_operator\_class:** El nombre de una clase de operadores árbol B para el subtipo.
- **canonical\_function:** El nombre de la función de canonización para el tipo range.
- **subtype\_diff\_function:** El nombre de una función de diferencia para el subtipo.
- **input\_function:** Nombre de la función de entrada del nuevo tipo. Esta función debe ya estar definida mediante **CREATE FUNCTION**, y debe actuar para convertir los datos de la forma de tipos externos a la forma de tipos internos.
- **output\_function:** Nombre de la función de salida del nuevo tipo. Esta función debe convertir los datos de la forma de tipos internos en su forma visualizable.
- **receive\_function:** El nombre de una función que convierte los datos del tipo de forma binaria externa a su forma interna.
- **send\_function:** El nombre de una función que convierte los datos del tipo de la forma interna a su forma binaria externa.
- **type\_modifier\_input\_function:** El nombre de una función que convierte del modificador de un array por el tipo de forma interna.
- **type\_modifier\_output\_function:** El nombre de una función que convierte la forma interna del modificador del tipo a la forma textual externa.
- **analyze\_function:** El nombre de una función que realiza análisis estadísticos para el tipo de dato.
- **internallength:** Longitud interna del nuevo tipo, en bytes.
- **alignment:** El requisito de alineación de almacenamiento del tipo de datos. Si se especifica, debe ser chan, int2, int4 o double; el predeterminado es int4.
- **storage:** La estrategia de almacenamiento para el tipo de datos.
- **like\_type:** El nombre de un tipo de datos existente que del cual el nuevo tipo tendrá la misma representación. Los valores de *internallength*, *passedbyvalue*, *alignment* y *storage* son copiados desde ese tipo, a menos que exista una especificación explícita del comando **CREATE TYPE** en otros lugares.

- **category:** El código de categoría para este tipo (un sólo carácter ASCII). El predeterminado es 'U' para "tipos definidos por usuarios".
- **externallength:** Longitud externa opcional del nuevo tipo.
- **element:** El tipo de dato de elementos individuales de un array.
- **delimiter:** El carácter delimitador a ser utilizado entre los valores en un array hechas de este tipo.
- **default:** El valor por defecto para el tipo de dato. Si esto es omitido, el valor por defecto es null.

### Ejemplos

En el código A.3 se define un tipo compuesto, en donde el tipo complex posee dos atributos y el tipo inventory\_item tres.

```
CREATE TYPE complex AS (
    r      double precision,
    i      double precision
);

CREATE TYPE inventory_item AS (
    name          text,
    supplier_id   integer,
    price         numeric
);
```

Código A.3: Ejemplo de definición de tipos compuestos en PostgreSQL

La sintaxis es comparable a la sentencia **CREATE TABLE**, excepto que solamente pueden ser especificados nombre de campos y tipos; sin restricciones que se puedan incluir (tales como *NOT NULL*). Una vez definidos los tipos, estos pueden ser utilizados para crear tablas, tal como se visualiza en el ejemplo del código A.4.

```
CREATE TABLE on_hand (
    item          inventory_item, count integer);

INSERT INTO on_hand VALUES (ROW('fuzzy dice', 42, 1.99), 1000);
```

Código A.4: Ejemplo de creación de tablas utilizando tipos compuestos en PostgreSQL



En el código A.5 se aprecia como se utilizan los tipos compuestos en funciones.

```
CREATE FUNCTION price_extension(inventory_item, integer) RETURNS
numeric
AS 'SELECT $1.price * $2' LANGUAGE SQL;

SELECT price_extension(item, 10) FROM on_hand;
```

Código A.5: Ejemplo de utilización de tipos compuestos con funciones en PostgreSQL

En el código A.6 se crea un tipo *ENUM*, que posee los estados de error de algún error y luego se crea una tabla *error* la cual tiene una columna del tipo *estado\_error*.

```
CREATE TYPE estado_error AS ENUM ('nuevo', 'abierto', 'cerrado');

CREATE TABLE error (
    id serial,
    descripcion text,
    estado estado_error
);
```

Código A.6: Ejemplo utilización de un tipo como columna de tabla en PostgreSQL

Una vez que las funciones de entrada / salida se hayan escrito y compilado en una librería compartida, se puede definir el tipo *Complejo* en SQL mediante la sentencia **CREATE TYPE** como se ilustra en el código A.7, para luego crear sus funciones ver A.8. Por otro lado en el código A.9 se realizó una definición del tipo *Complejo* en donde se declaran las funciones creadas en el código A.8.

```
CREATE TYPE Complejo;
```

Código A.7: Ejemplo declaración del tipo shell en PostgreSQL

```

CREATE FUNCTION complejo_in(cstring)
  RETURNS complejo
  AS 'filename'
  LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complejo_out(complejo)
  RETURNS cstring
  AS 'filename'
  LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complejo_recv(internal)
  RETURNS complejo
  AS 'filename'
  LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complejo_send(complejo)
  RETURNS bytea
  AS 'filename'
  LANGUAGE C IMMUTABLE STRICT;

```

Código A.8: Ejemplo declaración de funciones en PostgreSQL

```

CREATE TYPE Complejo ( internallength = 16,
  input = complejo_in,
  output = complejo_out,
  receive = complejo_recv,
  send = complejo_send,
  alignment = double
);

```

Código A.9: Ejemplo declaración de funciones de un Tipo en PostgreSQL

### A.1.2. Sintaxis y ejemplos XML

En el código A.10 se muestra la sintaxis para generar un valor del tipo XML.

```
XMLPARSE ( { DOCUMENT | CONTENT } value)
```

Código A.10: Sintaxis XMLPARSE en PostgreSQL

Un ejemplo de utilización se visualiza en el código A.11.

```
XMLPARSE (DOCUMENT '<?xml version="1.0"?><book><title>Manual</title><chapter>...</chapter></book>')
XMLPARSE (CONTENT 'abc<foo>bar</foo><bar>foo</bar>')
```

Código A.11: Sintaxis XMLPARSE en PostgreSQL

En el código A.12 describe como convertir caracteres a cadena en valores XML.

```
xml '<foo>bar</foo>'
'<foo>bar</foo> '::xml
```

Código A.12: Sintaxis conversión carácter a cadena en PostgreSQL

Para convertir un valor carácter desde XML se debe utilizar la función *xmlserialize*, ver código A.13.

```
XMLSERIALIZE ( { DOCUMENT | CONTENT }
value AS type )
```

Código A.13: Sintaxis conversión de un carácter desde XML en PostgreSQL

La sintaxis estándar de configuración de sesión XML OPTION, se detalla en el código A.14, esto sirve al momento de castear desde o un tipo XML sin tener que utilizar XMLPARSE o XMLSERIALIZE.

```
SET XML OPTION { DOCUMENT | CONTENT };
```

Código A.14: Sintaxis estándar configuración XML option

También se puede utilizar la sintaxis de PostgreSQL descrita en el código A.15.

```
SET xmloption TO { DOCUMENT | CONTENT };
```

Código A.15: Sintaxis PostgreSQL configuración XML option

La sintaxis del código A.16 sirve para crear un valor XML conteniendo un comentario XML con un texto específico como contenido. En el ejemplo del código A.17 se agrega un comentario con el contenido “hello”.

```
xmlcomment(text)
```

Código A.16: Sintaxis xmlcomment en PostgreSQL

```
SELECT xmlcomment('hello');
```

Código A.17: Ejemplo xmlcomment en PostgreSQL

## A.2. SQLServer

### A.2.1. Sintaxis y ejemplos O-R

#### Sintaxis CREATE TYPE

En el código A.18 se visualiza la sintaxis detallada para crear un tipo definido por el usuario.

```
CREATE TYPE [ schema_name. ] type_name
{
  FROM base_type
    [ ( precision [ , scale ] ) ] [ NULL | NOT NULL ]
  | EXTERNAL NAME assembly_name [ .class_name ]
  | AS TABLE ( { <column_definition> | <computed_column_definition> }
    [ <table_constraint> ] [ ,... n ] ) } [ ; ]
<column_definition> ::=
column_name <data_type>
  [ COLLATE collation_name ] [ NULL | NOT NULL ]
  [ DEFAULT constant_expression ] [ IDENTITY [ ( seed ,increment ) ] ]
  [ ROWGUIDCOL ] [ <column_constraint> [ ...n ] ]
<data type> ::= [ type_schema_name . ] type_name
  [ ( precision [ , scale ] | max | [ { CONTENT | DOCUMENT } ]
    xml_schema_collection ) ]
<column_constraint> ::=
{
  { PRIMARY KEY | UNIQUE }
    [ CLUSTERED | NONCLUSTERED ]
    [ WITH ( <index_option> [ ,... n ] ) ] | CHECK ( logical_expression )
}
<computed_column_definition> ::=
column_name AS computed_column_expression [ PERSISTED [ NOT NULL ] ]
[
  { PRIMARY KEY | UNIQUE }
    [ CLUSTERED | NONCLUSTERED ]
    [ WITH ( <index_option> [ ,... n ] ) ] | CHECK ( logical_expression ) ]
<table_constraint> ::=
{
  { PRIMARY KEY | UNIQUE }
    [ CLUSTERED | NONCLUSTERED ]
      ( column [ ASC | DESC ] [ ,...n ] )
    [ WITH ( <index_option> [ ,... n ] ) ] | CHECK ( logical_expression ) }
<index_option> ::=
{
  IGNORE_DUP_KEY = { ON | OFF } }
```

Código A.18: Sintaxis CREATE TYPE en SQLServer

Los argumentos que recibe son:

- **Nombre del esquema:** Es el nombre del esquema al cual pertenece el alias de tipo de datos o tipo definido por el usuario.
- **Nombre de tipo:** Es el nombre del alias de tipo de datos o el tipo definido por el usuario. Los nombres de tipo deben cumplir con las reglas de los identificadores.
- **Tipo base:** Es el tipo de datos suministrado por SQL Server en el que se basa el alias de tipo de dato. Los tipos de datos pueden ser los presentados en la tabla A-1.

bigint	binary( n )	bit	char( n )
date	datetime	datetime2	datetimeoffset
decimal	float	image	int
money	nchar( n )	ntext	numeric
nvarchar( n   max)	real	smalldatetime	smallint
smallmoney	sql_variant	text	time
tinyint	uniqueidentifier	varbinary( n   max)	varchar( n   max)

Tabla A-1: Valores que puede tener el tipo base en SQLServer

- **Precisión:** Indica el total máximo de números de dígitos decimales que pueden ser almacenados.
- **Escala:** Indica el número de dígitos decimales máximos que pueden ser almacenados a la derecha del punto decimal, y debe ser menor o igual a la precisión.
- **NULL | NOT NULL:** Especifica si el tipo puede contener un valor nulo. Si no se especifica, el valor predeterminado es NULL.
- **Nombre ensamblado:** Especifica el ensamblado de SQL SERVER que referencia la implementación del tipo definido por el usuario en el CLR. Debería coincidir con un ensamblado existente en la base de datos actual.
- **[.class\_name]:** Especifica la clase del ensamblado que implementa el tipo definido por el usuario.
- **<column\_definition>:** Define las columnas del tipo tabla definida por el usuario.

- **<data\_type>**: Define el tipo de dato en una columna del tipo tabla definida por el usuario.
- **<column\_constraint>**: Define las restricciones de la columna del tipo tabla definida por el usuario. Las restricciones soportadas incluyen PRIMARY KEY, UNIQUE, y CHECK.
- **<computed\_column\_definition>**: Define una expresión de columna calculada como columna del tipo tabla definido por el usuario.
- **<table\_constraint>**: Define las restricciones de una tabla en un tipo tabla definida por el usuario. Las restricciones soportadas incluyen PRIMARY KEY, UNIQUE, y CHECK.
- **<index\_option>**: Especifica el error de respuesta para valores de claves duplicadas en una operación de inserción de múltiples filas.

### A.2.2. Sintaxis y ejemplos XML

Para crear una tabla que posee una columna del tipo xml, se debe seguir la sintaxis del código A.19.

```
CREATE TABLE XML Ejemplo1 (
  Columna1 int NOT NULL PRIMARY KEY IDENTITY(1,1),
  Columna2 varchar(20) NULL,
  Columna3 xml NULL)
GO
```

Código A.19: Ejemplo creación una tabla con una columna XML en SQLServer

También como se muestra en el código A.20 se puede especificar un valor por defecto utilizando un CAST.

```
CREATE TABLE XML Ejemplo2 ( Columna1 int NOT NULL PRIMARY KEY
IDENTITY(1,1),
  Columna2 varchar(20) NULL,
  Columna3 xml NULL DEFAULT CAST(" AS xml))
GO
```

Código A.20: Ejemplo creación tabla con una columna XML y un valor por defecto usando un CAST a XML en SQLServer

Para simplemente declarar una variable del tipo XML, se debe realizar como aparece en el código A.21.

```
DECLARE @documento xml
SELECT @documento = ''
GO
```

Código A.21: Ejemplo declaración de una variable usando el tipo de datos XML en SQLServer

En el código A.22 se muestra un ejemplo de cómo asignar directamente valores a un tipo XML.

```
DECLARE @directorXML xml
SET @directorXML = CAST('1234567890Ridley, Scott' + 'Director de ciencia
    ficción' AS xml)
SELECT @directorXML AS DatosDirectores
```

Código A.22: Ejemplo asignación directa en SQLServer

En el código A.23 se visualiza como utilizar FOR XML y recuperar datos como XML.

```
DECLARE @empleadoXML xml
SET @empleadoXML = (SELECT TOP 3 IdCliente,
    Nombre,
    Pais, Departamento
    FROM Clientes
    FOR XML AUTO, ELEMENTS, ROOT ('Clientes'))
SELECT @empleadoXML AS DatosClientes
```

Código A.23: Ejemplo utilización FOR XML en SQLServer

En el ejemplo del código A.24 detalla como cargar datos desde un archivo XML mediante OPENROWSET.

```
DECLARE @productoXML xml
SET @productoXML = (SELECT *
    FROM OPENROWSET( BULK 'C:\SQLServer\consolas.xml',
    SINGLE_BLOB) AS xmlDatosProductos)
SELECT @productoXML AS DatosProductos
```

Código A.24: Ejemplo cargando datos utilizando OPENROWSET en SQLServer



En el código A.25 se crea una tabla en donde se almacenará un archivo XML y luego se inserta el contenido de dicho archivo.

```
CREATE TABLE XMLBicicletas (
    Codigo int NOT NULL PRIMARY KEY IDENTITY(1,1), BiciXML xml NULL)
GO

INSERT INTO XMLBicicletas(BiciXML)
SELECT * FROM OPENROWSET(
    BULK 'C:\SQLServer\Bicicletas.xml', SINGLE_BLOB) AS xmlProdData
GO
```

Código A.25: Ejemplo cargando datos utilizando OPENROWSET en SQLServer

En el código A.26 se muestra la estructura del archivo XML "Bicicletas.xml".

```
SELECT * FROM XMLBicicletas
<Bicycles><Product>
    <ProductID>791</ProductID>
    <Name>Road–250 Red, 52</Name>
    <ProductNumber>BK–R89R–52</ProductNumber>
    <MakeFlag>1</MakeFlag>
    <FinishedGoodsFlag>1</FinishedGoodsFlag>
    <Color>Red</Color>
    <SafetyStockLevel>100</SafetyStockLevel>
    <ReorderPoint>75</ReorderPoint>
    <StandardCost>1518.7864</StandardCost>
    <ListPrice>2443.3500</ListPrice><Size>52</Size>
    <SizeUnitMeasureCode>CM </SizeUnitMeasureCode>
    <WeightUnitMeasureCode>LB </WeightUnitMeasureCode>
    <Weight>15.42</Weight>
    <DaysToManufacture>4</DaysToManufacture>
    <ProductLine>R </ProductLine><Class>H</Class><Style>U</Style>
    <ProductSubcategoryID>2</ProductSubcategoryID>
    <ProductModelID>26</ProductModelID>
    <SellStartDate>2002–07–01T00:00:00</SellStartDate>
    <SellEndDate>2003–06–30T00:00:00</SellEndDate>
    <rowguid>C9FD1DF4–9512–420A–B379–067108033B75</rowguid>
    <ModifiedDate>2004–03–11T10:01:36.827</ModifiedDate>
</Product> </Bicycles>
```

Código A.26: Estructura de archivo xml Bicicletas SQLServer

OPENXML entrega una vista de un conjunto de filas de un documento XML, en el código A.27 se detalla su sintaxis.

```
OPENXML( idoc int [ in ] , rowpattern nvarchar [ in ] , [ flags byte [ in ] ] )
[ WITH ( SchemaDeclaration | TableName ) ]
```

Código A.27: Sintaxis OPENXML en SQLServer

OPENXML tiene como argumentos:

- **idoc**: Es el identificador del documento de la representación interna de un documento XML. La representación interna de un documento XML se crea llamando a la función `sp_xml_preparedocument` (Transact-SQL).
- **rowpattern**: Es el patrón XPath utilizado para identificar los nodos (en el documento XML cuyo identificador se pasa en el parámetro `idoc`) que se van a procesar como filas.
- **flags**: Señala la asignación que debe utilizarse entre los datos XML y el conjunto de filas relacional, y cómo debe llenarse la columna de desbordamiento; `flags` es un parámetro de entrada opcional y puede tomar uno de los valores que se mencionan en la tabla

Valor del byte	Descripción
0	Establece como valor predeterminado la asignación attribute-centric.
1	Usa la asignación attribute-centric. Se puede combinar con XML_ELEMENTS. En ese caso, primero se aplica la asignación attribute-centric y, a continuación, la asignación element-centric en todas las columnas que todavía no se han visto afectadas.
2	Usa la asignación element-centric. Se puede combinar con XML_ATTRIBUTES. En ese caso, primero se aplica la asignación attribute-centric y, a continuación, la asignación element-centric en todas las columnas que todavía no se han visto afectadas.
8	Puede combinarse (OR lógico) con XML_ATTRIBUTES o XML_ELEMENTS. Si se trata de una recuperación, esta marca informa de que los datos consumidos no se deberían copiar a la propiedad de desbordamiento <code>@mp:xmltext</code> .

Tabla A-2: Valores que puede tener el parámetro `flags` de OPENXML en SQLServer

- **SchemaDeclaration:** Es la definición de esquema de la forma: ColName-ColType "[ColPattern | MetaProperty] [,ColNameColType [ColPattern | MetaProperty]...]"
- **TableName:** Es el nombre de tabla que puede proporcionarse (en lugar de SchemaDeclaration) si ya existe una tabla con el esquema deseado y no se requiere patrones de columna.

Un ejemplo de su utilización es presentado en el código A.28, donde se crea una representación interna de la imagen XML utilizando `sp_xml_preparedocument`. A continuación se ejecuta una instrucción `SELECT` que usa un proveedor del conjunto de filas OPENXML contra la representación interna del documento XML. El proveedor del conjunto de filas OPENXML crea un conjunto de filas de 2 columnas (CustomerID y ContactName) desde el que la instrucción `SELECT` recupera las columnas necesarias (en este caso, todas las columnas).

```
DECLARE @idoc int, @doc varchar(1000);
SET @doc =
<ROOT>
<Customer CustomerID="VINET" ContactName="Paul Henriot">
  <Order CustomerID="VINET" EmployeeID="5" OrderDate="1996-07-04T00:00:00">
    <OrderDetail OrderID="10248" ProductID="11" Quantity="12"/>
    <OrderDetail OrderID="10248" ProductID="42" Quantity="10"/>
  </Order>
</Customer>
<Customer CustomerID="LILAS" ContactName="Carlos Gonzlez">
  <Order CustomerID="LILAS" EmployeeID="3" OrderDate="1996-08-16T00:00:00">
    <OrderDetail OrderID="10283" ProductID="72" Quantity="3"/>
  </Order>
</Customer>
</ROOT>';
// Create an internal representation of the XML document.
EXEC sp_xml_preparedocument @idoc OUTPUT, @doc;
// Execute a SELECT statement that uses the OPENXML rowset provider.
SELECT *
FROM OPENXML (@idoc, '/ROOT/Customer',1)
WITH (CustomerID varchar(10),
      ContactName varchar(20));
```

Código A.28: Ejemplo OPENXML en SQLServer

## A.3. DB2

### A.3.1. Sintaxis y ejemplos O-R

En el ejemplo del código A.29 se crea un tipo para DEPARTAMENTO, el código A.30 se crea una jerarquía de tipos compuesta de un tipo para empleados y de un subtipo directores.

```
CREATE TYPE DEPT AS
  (DEPT NAME VARCHAR(20),
   MAX_EMPS INT)
  REF USING INT
  MODE DB2SQL
```

Código A.29: Ejemplo CREATE TYPE estructurado en DB2

```
CREATE TYPE EMP AS
  (NAME VARCHAR(32),
   SERIALNUM INT,
   DEPT REF(DEPT),
   SALARY DECIMAL(10,2))
  MODE DB2SQL

CREATE TYPE MGR UNDER EMP AS
  (BONUS DECIMAL(10,2))
  MODE DB2SQL
```

Código A.30: Ejemplo CREATE TYPE estructurado jerarquía compuesta en DB2

En el ejemplo del código A.30 además, se puede observar que el empleado hace referencia al departamento que pertenece.

Un ejemplo de creación de tipo matriz se describe en el código A.31.

```
CREATE TYPE
  PHONENUMBERS AS DECIMAL(10,0)
  ARRAY[50]
```

Código A.31: Ejemplo CREATE TYPE MATRIX en DB2

En el código A.32 se crea un tipo fila según las columnas de la tabla existen DEPARTAMENTO.

```
CREATE TYPE DEPTROW AS ROW (DEPTNO VARCHAR(3),
                             DEPTNAME VARCHAR(29),
                             MGRNO CHAR(6),
                             ADMRDEPT CHAR(3),
                             LOCATION CHAR(16))
```

Código A.32: Ejemplo CREATE TYPE FILA en DB2

En el código A.33 se aprecia un ejemplo de creación de cursores.

```
CREATE TYPE EMPCURSOR AS CURSOR
```

Código A.33: Ejemplo CREATE TYPE CURSOR en DB2

### A.3.2. Sintaxis y ejemplos XML

Algunas de las funciones que provee DB2 para XML son:

- **XMLATTRIBUTES**: construye atributos XML a partir de los argumentos. Esta función puede ser utilizada como argumento de la función XMLEMENT. El resultado es una secuencia XML que contiene un nodo de atributo XQuery para cada valor de entrada que no sea nulo.
- **XMLCOMMENT**: entrega un valor XML con un único nodo de comentario XQuery con el argumento de entrada como contenido.
- **XMLCONCAT**: devuelve una secuencia que contiene la concatenación de un número variable de argumentos de entrada de XML.
- **XMLDOCUMENT**: retorna un valor XML con un único nodo de documento XQuery que puede tener o no nodos hijo.
- **XMLEMENT**: entrega un valor XML que es un nodo de elemento XQuery.
- **XMLFOREST**: devuelve un valor XML que es una secuencia de nodos de elemento XQuery. Además esta función es capaz de tomar un conjunto opcional de declaraciones de espacios de nombres y uno o varios argumentos que forman el contenido de nombre y elemento para uno o varios nodos de elementos.

- **XMLNAMESPACES**: construye declaraciones de espacios de nombres mediante los argumentos. Esta declaración únicamente puede emplearse como argumento de funciones específicas como XMLELEMENT, XMLFOREST y XMLTABLE. El resultado es una o varias declaraciones de espacios de nombres XML que contienen espacios de nombres con ámbito para cada uno de los valores de entrada no nulos
- **XMLPARSE**: examina el argumento como un documento XML y devuelve un valor XML.
- **XMLPI**: entrega un valor XML con un nodo de instrucción de proceso XQuery.
- **XMLQUERY**: devuelve un valor XML mediante la evaluación de una expresión XQuery utilizando posiblemente los argumentos de entrada especificados como variables XQuery.
- **XMLROW**: retorna un valor XML con un único nodo de documento XQuery que contiene un nodo de elemento de nivel superior.
- **XMLSERIALIZE**: entrega un valor XML serializado de los tipos de datos especificados, generados a partir del argumento expresión-XML.
- **XMLTEXT**: devuelve un valor XML que posee un único nodo de texto XQuery cuyo contenido es el argumento de entrada.
- **XMLVALIDATE**: retorna una copia del valor de entrada XML aumentado con la información conseguida mediante la validación del esquema XML, incluyendo los valores por omisión.

## Ejemplos

En el ejemplo del código A.34 se genera un elemento con atributos, cuya consulta da como resultado el código A.35.

```
SELECT E.rut, XMLELEMENT(
  NOMBRE "Emp",
  XMLATTRIBUTES(
    E.rut, E.nombre || ' ' || E.apellido AS "nombre"
  )
)
AS "Resultado"
FROM Empleado E WHERE E.edlevel = 14
```

Código A.34: Ejemplo XMLATTRIBUTES en DB2

```

RUT      Resultado
19990909 <Emp RUT="19990909" nombre="Claire Redfield"></Emp>
18928283 <Emp RUT="18928283" nombre="Regina George"></Emp>
14033230 <Emp RUT="14033230" nombre="Anakin Skywalker"></Emp>

```

Código A.35: Resultado de utilización de XMLATTRIBUTES en DB2

En el código A.36 se presenta un ejemplo de utilización de la función XMLPARSE.

```

INSERT INTO PRODUCTO VALUES ('111-111-11','Reproductor
Bluray',30.000,NULL,NULL,NULL,XMLPARSE ( DOCUMENT
'<producto xmlns="http://ejemplos.info" pid="111-111-11">
  <descripcion>
    <nombre>Reproductor Bluray</nombre>
    <detalles>
      Reproductor de bluray LG:
      - tecnología 3D
      - reproducción de archivos MKV vía USB
      - Wifi incorporado
    </detalles>
    <precio>30.000</precio>
    <peso>1.5 kg</peso>
  </descripcion>
</producto>' PRESERVE WHITESPACE ));

```

Código A.36: Ejemplo de utilización de XMLPARSE en DB2

Primero en la tabla A-3 se muestran la estructura con datos de la tabla empleado, para luego en el código A.37 utilizar la función XMLROW, el resultado obtenido se visualiza en el código A.38.

rut	nombre	apellido	edad
14123230	Milla	Jovovich	37
12334355	Robert	Rodriguez	42
10349529	Billy	Corgan	

Tabla A-3: Tabla empleado con datos de ejemplo

```
SELECT XMLROW(rut,apellido,edad) FROM empleado
```

Código A.37: Ejemplo de utilización de XMLROW en DB2

```
<row><rut>14123230</rut><apellido>Jovovich</apellido><edad>37</edad></row>
<row><rut>12334355</rut><apellido>Rodriguez</apellido><edad>42</edad></row>
<row><rut>10349529</rut><apellido>Corgan</apellido></row>
```

Código A.38: Resultado de utilización de XMLROW en DB2

## A.4. Oracle

### A.4.1. Sintaxis y ejemplos O-R

#### Ejemplos de tipo colección

En el ejemplo del código A.39 se presenta la definición orientada a objetos y en el código A.40 su definición en ORACLE, se define un tipo de datos para almacenar una lista ordenada de teléfonos: el tipo list (ya que en el tipo set no existe orden). Este tipo se utiliza después para asignárselo a un atributo del tipo de objeto cliente\_t.

```
define type Lista_Tel_T:
    list ( string );

define class Cliente_T:
    tuple [clinum: integer, clinomb:string,
           direccion:Direccion_T,
           lista_tel : Lista_Tel_T];
```

Código A.39: Definición orientada a objetos en C de ejemplo para Varray

```
CREATE TYPE lista_tel_t AS VARRAY(10) OF VARCHAR2(20);

CREATE TYPE cliente_t AS OBJECT (clinum NUMBER, clinomb VARCHAR2
(200), direccion direccion_t, lista_tel lista_tel_t );
```

Código A.40: Ejemplo de definición VARRAY



El ejemplo del código A.41 y A.42 en C y SQL respectivamente, se declara una tabla que después será anidada en el tipo ordenes\_t.

```
define type Linea_T:
  tuple [linum:integer,
        item:string,
        cantidad:integer,
        descuento:real];
```

Código A.41: Definición orientada a objetos en C de ejemplo para Nested Table 1

```
CREATE TYPE linea_t AS OBJECT (
  linum NUMBER,
  item VARCHAR2(30),
  cantidad NUMBER,
  descuento NUMBER(6,2));
```

Código A.42: Ejemplo 1 de tabla para utilizar Nested Table

En el código A.42 se define el tipo de objeto linea\_t para las filas de la tabla anidada. Se define el tipo colección tabla lineas\_pedido\_t para después anidarla como se visualiza en el código A.43.

```
CREATE TYPE lineas_pedido_t AS TABLE OF linea_t ;
```

Código A.43: Ejemplo de Nested Table 1

Esta definición permite utilizar el tipo colección lineas\_pedido\_t para:

- Definir el tipo de dato de una columna de una tabla relacional.
- Definir el tipo de dato de un atributo de un tipo de objetos.
- Para definir una variable PL/SQL, un parámetro, o el tipo que devuelve una función.

Otro ejemplo es presentado en el código A.44 y A.45 respectivamente, en donde se define el tipo objeto ordenes\_t y su atributo pedido almacena una tabla anidada del tipo lineas\_pedido\_t.

```
define class Ordenes_T:
  tuple [ordnum:integer,
        cliente :Clientes_T,
        fechpedido:date,
        fechentrega:date,
        pedido:set (Linea_T),
        direcentrega:Direccion_T];
```

Código A.44: Definición orientada a objetos en C de ejemplo para Nested Table 2

```
CREATE TYPE ordenes_t AS OBJECT (
  ordnum NUMBER,
  cliente REF cliente_t,
  fechpedido DATE,
  fechentrega DATE,
  pedido lineas_pedido_t,
  direcentrega direccion_t) ;
```

Código A.45: Ejemplo 2 de tabla para utilizar Nested Table

Luego en el código A.46 se define la tabla de objetos ordenes\_tab y se especifica la tabla anidada del tipo lineas\_pedido\_t. Es necesario realizarlo de esta forma porque la declaración de una tabla anidada no reserva ningún espacio para su almacenamiento. Lo que se hace es indicar en qué tabla (pedidos\_tab) se deben almacenar todas las líneas de pedido que se representen en el atributo pedido de cualquier objeto de la tabla ordenes\_tab. Es decir, todas las líneas de pedido de todas las órdenes se almacenan externamente a la tabla de órdenes, en otra tabla especial.

```
CREATE TABLE ordenes_tab OF ordenes_t
  (ordnum PRIMARY KEY,
  SCOPE FOR (cliente) IS clientes_tab)
NESTED TABLE pedido STORE AS pedidos_tab) ;
```

Código A.46: Ejemplo de Nested Table 2

En los códigos A.47 y A.48 se visualiza que en un tipo de objeto, los métodos pueden hacer referencia a los atributos y a los otros métodos.

```
CREATE TYPE Pila AS OBJECT (
  top INTEGER,
  MEMBER FUNCTION llena RETURN BOOLEAN,
  MEMBER FUNCTION insertar (n IN INTEGER)
);
```

Código A.47: Ejemplo objeto para referencia de atributos en ORACLE

```
CREATE TYPE BODY
Pila AS ....
  MEMBER FUNCTION insertar (n IN INTEGER)IS
  BEGIN
    IF NOT llena THEN
      top := top + 1;
      ...
    END insertar;
END;
```

Código A.48: Ejemplo referencia de atributos de la especificación en el cuerpo de método en ORACLE

En el código A.49 se observa un ejemplo del método *transformar* que declara SELF como un parámetro IN OUT.

```
CREATE TYPE Complejo AS OBJECT (
  MEMBER FUNCTION transformar (SELF IN OUT Complejo ) . . .
END;
```

Código A.49: Ejemplo parámetro SELF en ORACLE

Como muestra el ejemplo del código A.50, los métodos pueden hacer referencia a los atributos de SELF sin necesidad de utilizar un cualificador. Este método encuentra el máximo común divisor de x e y.

```

CREATE FUNCTION gcd ( x INTEGER, y INTEGER) RETURN INTEGER AS
ans INTEGER;
BEGIN
IF x < y THEN ans := gcd (x,y) ;
ELSE ans := gcd (y,x MOD y) ;
ENDIF; RETURN ans;
END; /

CREATE TYPE Rational AS num INTEGER, den INTEGER,
MEMBER PROCEDURE normalize ,...) ; /
CREATE TYPE BODY Rational AS
MEMBER PROCEDURE normalize IS g INTEGER;
BEGIN
    // Estas dos sentencias son equivalentes
    g:= gcd (SELF.num,SELF.den);
    g:= gcd (num,den);
    num:= num / g ;
    den:= den / g ;
    END normalize ;
    ...
END;

```

Código A.50: Ejemplo referencia a parámetro SELF desde método en ORACLE

En el bloque del código A.51 se declara un objeto r de tipo Racional y se invoca al constructor para asignar su valor. El llamado asigna los valores 6 y 8 a los atributos num y den respectivamente.

```

DECLARE
    r Racional ;
BEGIN
    r := Racional(6,8) ;
    DBMS_OUTPUT.PUT_LINE(r.num); // muestra 6

```

Código A.51: Ejemplo declaración e inicialización de objeto en ORACLE

En el ejemplo del código A.52, se emplea un objeto de tipo Cuenta para especificar el tipo de dato de un parámetro formal. Y en el código A.53 se declara una función que retorna un objeto de tipo Cuenta.

```
DECLARE
```

```
...
```

```
PROCEDURE cuenta_abierta ( nueva_cuenta IN OUT Cuenta ) IS ...
```

Código A.52: Ejemplo especificación de parámetro del tipo objeto 'Cuenta' en ORACLE

```
DECLARE
```

```
...
```

```
FUNCTION get_cuenta ( id_cuenta IN INTEGER ) RETURN Cuenta IS ...
```

Código A.53: Ejemplo declaración de función que retorna un objeto del tipo 'Cuenta' en ORACLE

Como se ilustra en el código A.54 si se asigna el no-valor NULL a un objeto, éste se convierte en atómicamente nulo. Por otro lado una buena práctica de programación se observa en el código A.55 en donde se inicializan los objetos en su declaración.

```
DECLARE
```

```
  r Racional ;
```

```
BEGIN
```

```
  r Racional := Racional (1,2) ; // r = 1/2
```

```
  r := NULL; // r atómicamente nulo
```

```
  IF r IS NULL THEN ... // la condicion resulta TRUE
```

Código A.54: Ejemplo inicialización de objeto - atómicamente nulo en ORACLE

```
DECLARE
```

```
  r Racional := Racional (2,3) ; // r = 2/3
```

Código A.55: Ejemplo inicialización de objetos ORACLE

Existe una pequeña diferencia entre objetos nulos y objetos con atributos nulos. El código A.56 ejemplifica dicha diferencia.

```

DECLARE
  r Relacional ; // es atómicamente nulo
BEGIN
  IF r IS NULL THEN ... //TRUE
  IF r.num IS NULL THEN ... // TRUE
  r := Racional (NULL, NULL); // Inicializar
  r.num = 4; // Exito : r ya no es atómicamente nulo aunque sus atributos son
    nulos
  r := NULL; // r es de nuevo atómicamente nulo
  r.num := 4; // Provoca la excepción ACCESS_INTO_NULL
EXCEPTION
  WHEN ACCESS_INTO_NULL THEN
    ...
END;
/

```

Código A.56: Ejemplo diferencia de objetos nulos y atributos nulos en ORACLE

El ejemplo del código A.57 ilustra la notación punto ('.'), que sirve para poder acceder o cambiar los valores de un atributo.

```

DECLARE
  r Racional := Racional (NULL, NULL) ;
  numerador INTEGER;
  denominador INTEGER;
BEGIN
  ...
  denominador := r.den;
  r.num := numerados;
END;

```

Código A.57: Ejemplo de acceso a los atributos de un objeto en ORACLE

Otro ejemplo es presentado en el código A.58, en donde se definen los tipos de objeto Dirección y Estudiante, como se observa `cod_postal` es un atributo de tipo Dirección y Dirección es el tipo de dato del atributo `direccion_casa` del tipo de objeto Estudiante. Si este es un objeto Estudiante, para poder acceder al valor de su `cod_postal` se utiliza la notación *est.direccion\_casa.cod\_postal*.

```

OBJECT ( calle VARCHAR2(30) ,
        ciudad VARCHAR2(20) ,
        estado CHAR( 2 ) ,
        cod_postal VARCHAR2(5)
) ;
/

CREATE TYPE Estudiante AS OBJECT (
    nombre VARCHAR2(20) ,
    direccion_casa Direccion ,
    telefono VARCHAR2(10) ,
    estatus VARCHAR2(10) ,
    nombre_consejero VARCHAR2(20) ,
    ...
) ;
/

```

Código A.58: Ejemplo de utilización de objetos como tipo de datos de atributos en ORACLE

En el código A.59, se muestra como se permite invocar al constructor en cualquier punto en donde se puede invocar una función.

```

DECLARE
    r1 Racional := Racional (2,3) ;
    FUNCTION promedio ( x Racional , y Racional ) RETURN Racional IS
    BEGIN
        ...
    END;
BEGIN
    r1 := promedio (Racional(3,4) , Racional (7,11)) ;
    IF ( Racional (5,8) > r1 ) THEN
        ...
    END IF ;
END;
/

```

Código A.59: Ejemplo invocación de constructor en ORACLE

En el ejemplo del código A.60 se muestra como es posible invocar al constructor empleando la notación con nombre en vez de la posición.

```
BEGIN
  r := Racional( den => 6 ,num => 5); //asigna num = 5 y den = 6
```

Código A.60: Ejemplo de llamado al constructor por nombre de parámetros en ORACLE

En el ejemplo del código A.61, pragma impide al método MAP convert leer el estado de la base de datos (rnds), modificar el estado de la base de datos (wnds), leer el estado de un paquete o módulo (rnps) y modificar el estado de un paquete (wnps).

```
CREATE TYPE Relacional AS OBJECT (numero INTEGER, den INTEGER,
  MAP MEMBER FUNCTION convert RETURN REAL,
  ....
  PRAGMA RESTRICT_REFERENCES (convert, RNDS, WNDS, RNPS,
    WNPS));
```

Código A.61: Ejemplo de PRAGMA REFERENCES en ORACLE

En el ejemplo del código A.62 se invoca al método normaliza, el cual divide los atributos num y den por el mayor común divisor.

```
DECLARE
  r Racional;
BEGIN
  r := Racional (6,8);
  r.normaliza;
  DBMS_OUTPUT.PUT_LINE( r.num); // muestra 3
```

Código A.62: Ejemplo de llamado de métodos de un objeto en ORACLE

En el código A.63, se muestra como encadenar los llamados a los métodos, primero se llama a la función *reciproco* y luego a la función *normaliza*.

```
DECLARE
  r Racional := Racional (6,8);
BEGIN
  r.reciproco( ).normaliza;
  DBMS_OUTPUT.PUT_LINE(r.num); // muestra 4
END;
```

Código A.63: Ejemplo de llamado encadenado de métodos de un objeto en ORACLE



Por ejemplo, la sentencia del código A.64 es ilegal ya que o es posible encadenar llamados a métodos adicionales a la derecha del llamado de un procedimiento.

```
r.normaliza( ).reciproco; // ilegal
```

Código A.64: Ejemplo de llamado encadenado erróneo en ORACLE

En el ejemplo del código A.65, la primera sentencia *CREATE TYPE* es ilegal ya que hace referencia al objeto de tipo Departamento, el cual todavía no existe.

```
CREATE TYPE Empleado AS OBJECT (
  nombre VARCHAR2(20) ,
  depto REF Departamento, // ilegal!
...) ; /
CREATE TYPE Departamento AS OBJECT (
  numero INTEGER ,
  gerente REF Empleado ,
...) ; /
```

Código A.65: Ejemplo de utilización errónea de REF en objetos en ORACLE

En el ejemplo del código A.65, cambiar el orden de las sentencias *CREATE TYPE* no soluciona en nada el problema, ya que ambos tipos son dependientes mutuamente, en otras palabras dependen el uno del otro por medio de una referencia. Para resolver el problema mencionado previamente, basta con incluir la sentencia del código A.66.

```
CREATE TYPE Departamento; // Definición previa de tipo, aquí Departamento
    es un
tipo de objeto incompleto.
```

Código A.66: Ejemplo de utilización errónea de REF en objetos en ORACLE

Por ejemplo, la sentencia *CREATE TYPE* del código A.67 compila con errores debido a que el tipo de objeto *Direccion* todavía no se encuentra definido.

```
CREATE TYPE Cliente AS OBJECT (
  id NUMBER, nombre VARCHAR2(20),
  dir Direccion, //todavía indefinido
telefono VARCHAR2(15)) ; /
```

Código A.67: Ejemplo de utilización errónea de REF en objetos en ORACLE

En el código A.68 la sentencia INSERT llama al constructor del tipo *Racional* para insertar su valor. La sentencia SELECT rescata el valor del atributo *numero* y la sentencia UPDATE llama al método *reciproco*, el cual retorna un valor *Relacional* después de invertir los valores de *numero* y *denominador*.

```
CREATE TABLE numeros(rn Racional,...);
INSERT INTO numeros (rn)VALUES(Racional(3,62));
SELECT nu.rn.num INTO mi_numero FROM numeros nu WHERE ...
UPDATE numeros nu SET nu.rn = nu.rn.reciproco WHERE ...
```

Código A.68: Ejemplo de utilización errónea de REF en objetos en ORACLE

Por ejemplo, para obtener un conjunto de objetos *Persona* se puede utilizar el comando VALUES como se detalla en el código A.69.

```
BEGIN
  INSERT INTO Empleados SELECT VALUE(per) FROM Personas per WHERE
    per.apellido
  LIKE '% Tarantino'
```

Código A.69: Ejemplo de utilización de operador VALUE en sentencia INSERT INTO en ORACLE

Por ejemplo en el código A.70, se puede ejecutar una de las dos instrucciones siguientes. En la primera instrucción, la tabla clientes\_año\_tab se considera como una tabla con varias columnas cuyos valores son los especificados. En el segundo caso, se la considera como con una tabla de objetos que en cada fila almacena un objeto. En esta instrucción la cláusula VALUE permite visualizar el valor de un objeto.

```
INSERT INTO clientes_año_tab VALUES(
  2347,
  Juan Pérez Ruíz,
  direccion_t( Calle Castalia , Onda , Castellón , 34568 ),
  696-779789,
  12/12/1981);
SELECT VALUE(c) FROM clientes_año_tab c
WHERE c.clinomb = Juan Pérez Ruíz;
```

Código A.70: Ejemplo de utilización de operador VALUE en sentencia SELECT en ORACLE

En el ejemplo del código A.71, primero se recuperan una o más referencias a objetos de tipo Persona y luego se ingresan en la tabla *persona\_refs*.

```
BEGIN
  INSERT INTO persona_refs SELECT REF(per) FROM Personas per WHERE
    per.apellido LIKE '% Scorsese';
```

Código A.71: Ejemplo de inserción de objetos en una tabla de objetos utilizando operador REF en ORACLE

El ejemplo del código A.72 define un atributo de tipo REF y restringe su dominio a los objetos de cierta tabla.

```
CREATE TABLE clientes_tab OF cliente_t;

CREATE TYPE ordenes_t AS OBJECT (
  ordnum NUMBER, cliente REF clientes_t, fechpedido DATE, direcentrega
  direccion_t);

CREATE TABLE ordenes_tab OF ordenes_t (
  PRIMARY KEY (ordnum),
  SCOPE FOR (cliente) IS clientes_tab);
```

Código A.72: Ejemplo de definición de tipo REF y restricción de dominio a objetos de una tabla

Por ejemplo, suponer que la columna gerente en la tabla relacional departamento contiene referencias a objetos Employee almacenados en una tabla de objetos. Para convertir todas las referencias colgadas en nulos, podemos utilizar la siguiente sentencia UPDATE del código A.73.

```
UPDATE departamento SET gerente = NULL
WHERE gerente IS DANGLING;
```

Código A.73: Ejemplo de DANGLING REF en ORACLE

En el ejemplo que se visualiza en el código A.74 se derreferencia una referencia a un objeto Persona de la tabla DUAL2.

```
per1 Persona;
per_ref REF Persona;
nombre VARCHAR2(15) ;
BEGIN
...
// Suponer que per_ref contiene una referencia valida a un objeto almacenado
// en una tabla de objetos

SELECT Deref( per_ref ) INTO per1 FROM DUAL;
nombre:= per1.apellido;
```

Código A.74: Ejemplo de derreferencia de objetos en ORACLE

En el código A.75 se muestra la utilización del operador Deref en sentencias SQL sucesivas para derreferencias referencias.

```
CREATE TYPE PersonaRef AS OBJECT (p_ref REF Persona) ;
/
DECLARE
nombre VARCHAR2(15) ;
pr_ref REF PersonaRef ;
pr PersonaRef ;
p Persona ;
BEGIN
...
// Suponer que pr_ref contiene una referencia valida
SELECT Deref( pr_r e f ) INTO pr FROM DUAL;
SELECT Deref( pr . p_ref ) INTO p FROM DUAL;
nombre:= p.apellido;
...
END;
/
```

Código A.75: Ejemplo utilización de operador Deref en ORACLE

Por ejemplo, la sentencia del código A.76 es legal, cn sentencias SQL se puede utilizar la notación punto para navegar a través de referencias.

```
alias_tabla.columna_objeto.ref_atributo.atributo alias_tabla.ref_columna.
atributo
```

Código A.76: Ejemplo de navegación entre referencias en ORACLE

El código A.77 crea los tipos de objeto Dirección y Persona y la tabla de objetos Personas, en donde el atributo *direccion\_casa* es una referencia a una columna en la tabla de objetos Personas, que a la vez contiene referencias a objetos *Direccion* que se encuentran almacenados en otra tabla indeterminada. Tras introducir algunos elementos en la tabla, es posible obtener una dirección particular derreferenciando su referencia, como se muestra en el código A.78.

```
CREATE TYPE Direccion AS OBJECT (
  direccion_calle VARCHAR2(35) ,
  ciudad VARCHAR2(15) ,
  estado CHAR( 2 ) ,
  cod_postal INTEGER) ;
/
CREATE TYPE Persona AS OBJECT (
  nombre VARCHAR2(15) ,
  apellido VARCHAR2(15) ,
  fecha_nac DATE,
  direccion_casa Direccion,
  telefono VARCHAR2(15));
/
CREATE TABLE Personas OF Persona;
```

Código A.77: Ejemplo de creación de objetos con referencia a otro objeto y tabla que lo contiene en ORACLE

```
dir1 Direccion,
dir2 Direccion,
...
BEGIN
  SELECT Deref(direccion_casa) INTO dir1 FROM Personas p WHERE p.
    apellido = 'Lee';
```

Código A.78: Ejemplo obtención de dato de columna derreferenciando una referencia en ORACLE

En el caso del ejemplo del código A.79, se navega a través de la columna de referencias *direccion\_casa* hasta el atributo *direccion\_calle*, para este caso en

particular se requiere de un alias a la tabla (per).

```
DECLARE
  mi_calle VARCHAR2(25),
  ...
BEGIN
  SELECT per.direccion_casa.direccion_calle INTO mi_calle FROM Personas
    per WHERE per.apellido = 'kubrick';
```

Código A.79: Ejemplo de acceso a atributo mediante columna de un objeto en ORACLE

Por ejemplo, para insertar un objeto Persona en la tabla de objetos personas se emplea el código A.80. Además, es posible utilizar el constructor para el objeto de tipo Persona tal y como se observa en el código A.81.

```
BEGIN INSERT INTO Personas VALUES ('George' , 'Lucas',...);
```

Código A.80: Ejemplo de inserción de objetos en una tabla de objetos en ORACLE

```
BEGIN
  INSERT INTO Personas VALUES (Persona('Darren','Aronofsky' ,...));
```

Código A.81: Ejemplo de inserción de objetos en una tabla de objetos utilizando constructor de objeto en ORACLE

En el código A.82 se grafica un ejemplo de la cláusula RETURNING, la cual almacena una referencia a Persona en una variable local. En el código A.83 se muestra como ingresar objetos a una tabla de objetos, las filas copiadas a la tabla de objetos *Personas2* cuentan con identificadores de objeto nuevos, ya que los identificadores de objeto son únicos.

```
DECLARE
  per1_ref REF Persona, per2_ref REF Persona, ...
BEGIN
  INSERT INTO Personas pers VALUES (Persona('Christopher','Nolan',...))
    RETURNING REF(pers) INTO per1_ref;
  INSERT INTO Personas pers VALUES (Persona('Luc' , 'Besson',...))
    RETURNING
    REF(pers) INTO per2_ref;
```

Código A.82: Ejemplo de inserción de objetos en una tabla de objetos utilizando cláusula RETURNING en ORACLE

```
BEGIN
  INSERT INTO Personas2
    SELECT VALUE(pers) FROM Personas pers WHERE pers.apellido LIKE '
    % Potter';
```

Código A.83: Ejemplo de inserción de objetos en una tabla de objetos copiando filas desde objeto en ORACLE

Otro ejemplo puede ser apreciado en el código A.84, donde se crea una tabla relacional *Departamento*, la cual cuenta con una columna del tipo *Persona*; luego se inserta una fila en la tabla. Cabe destacar cómo el constructor *Persona()* suministra un valor para la columna *Gerente*. El nuevo objeto *Persona* almacenado en la columna gerente no se puede referenciar, porque al estar almacenada en una columna (y no en una fila) carece de identificador de objeto.

```
CREATE TABLE Departamento (
  depto_nombre VARCHAR2(20),
  gerente Persona,
  ubicacion VARCHAR2(20));
/
INSERT INTO Departamento VALUES ('Ventas',Persona('Ridley','Scott',...),'Los
  Angeles');
```

Código A.84: Ejemplo tabla relacional con columna de un tipo de objeto en ORACLE

Como se ilustra en el código A.85 para poder modificar los atributos de un objeto en una tabla de objetos se debe utilizar la sentencia UPDATE.

```
BEGIN
  UPDATE Personas pers SET pers.direccion_casa = 'Prat 1313' WHERE pers.
    apellido= 'Skywalker'; ...
  UPDATE Personas pers SET pers = Persona ('James','Cameron', ...) WHERE
    pers.apellido = 'Spielberg'; ...
END
```

Código A.85: Ejemplo de actualización de objetos en ORACLE

Como se muestra en el ejemplo del código A.86, para eliminar objetos selectivamente se utiliza la cláusula WHERE.

```
BEGIN
  DELETE FROM Personas pers WHERE pers.direccion_casa = '21 de Mayo
    2121';
  ...
END;
```

Código A.86: Ejemplo de eliminación de objetos en ORACLE



### A.4.2. Sintaxis y ejemplos XML

Primero se creará un esquema de validación de documentos XML (XSD) con la finalidad de que todo documento XML ingresado cumpla cierta estructura y condiciones. Cualquier documento XML debe cumplir con dicha especificación y será validado al momento de ser insertado en la tabla RESERVA que se creará más adelante.

En la figura A.1 se presenta el documento XSD que se registrará la base de datos. Luego en el código A.87 se registra el documento XSD en la base de datos, esto se realiza a través del procedimiento DBMS\_XMLSCHEMA.REGISTERSCHEMA.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="pedidos">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" name="pedido">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="fecha" type="xs:string" />
              <xs:element name="pu" type="xs:decimal" />
              <xs:element name="cantidad" default="1">
                <xs:simpleType>
                  <xs:restriction base="xs:unsignedByte">
                    <xs:minInclusive value="0"/>
                    <xs:maxInclusive value="50"/>
                  </xs:restriction>
                </xs:simpleType>
              </xs:element>
              <xs:element name="descripcion" type="xs:string"/>
              <xs:element name="tipo">
                <xs:simpleType>
                  <xs:restriction base="xs:string">
                    <xs:enumeration value="A"/>
                    <xs:enumeration value="B"/>
                    <xs:enumeration value="C"/>
                  </xs:restriction>
                </xs:simpleType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="cod" type="xs:integer" use="required"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Figura A.1. Ejemplo estructura de esquema XML (Documento XSD) [17]

```

begin
  DBMS_XMLSCHEMA.REGISTERSCHEMA(SCHEMAURL=>'pedidos.xsd',
    SCHEMADOC=>'<?xml version="1.0" encoding="utf-8"?>
    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:element name="pedidos">
        <xs:complexType>
          <xs:sequence>
            <xs:element maxOccurs="unbounded" name="pedido">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="fecha" type="xs:string" />
                  <xs:element name="pu" type="xs:decimal" />
                  <xs:element name="cantidad" default="1">
                    <xs:simpleType>
                      <xs:restriction base="xs:unsignedByte">
                        <xs:minInclusive value="0"/>
                        <xs:maxInclusive value="50"/>
                      </xs:restriction >
                    </xs:simpleType> </xs:element>
                  <xs:element name="descripcion" type="xs:string"/>
                  <xs:element name="tipo">
                    <xs:simpleType>
                      <xs:restriction base="xs:string">
                        <xs:enumeration value="A"/>
                        <xs:enumeration value="B"/>
                        <xs:enumeration value="C"/>
                      </xs:restriction >
                    </xs:simpleType></xs:element>
                  </xs:sequence>
                <xs:attribute name="cod" type="xs:integer" use="required"/>
              </xs:complexType></xs:element>
            </xs:sequence>
          </xs:complexType> </xs:element>
        </xs:schema>', LOCAL=>true, GENTYPES=>false, GENBEAN=>false,
        GENTABLES=>false,
        FORCE=>false, OPTIONS=>DBMS_XMLSCHEMA.
        REGISTER_BINARYXML, OWNER=>USER);
  commit;
end;/

```

Código A.87: Ejemplo de registro de esquema XML en ORACLE [17]

En el código A.88 se muestra la creación de una tabla con una columna XMLTYPE. En donde se crea la tabla Reserva que se compone de un campo llamado pedido de tipo XMLTYPE al cual se le está especificando que será almacenado como Binary XML. También se indica que el campo pedido será validado por el esquema pedidos.xsd que previamente lo ha sido creado. Por otro lado en el código A.89 se inserta una tupla en la tabla reserva.

```
CREATE TABLE FRICCIO.RESERVA(id number, pedido xmltype)
XMLTYPE COLUMN pedido STORE AS BINARY XML
XMLSCHEMA "http://xmlns.oracle.com/xdbschemas/FRICCIO/pedidos.xsd"
ELEMENT "pedidos";
```

Código A.88: Ejemplo de creación de tabla con columna XMLType en ORACLE [17]

```
INSERT INTO friccio.reserva values (1,'<?xml version="1.0"?>
<pedidos>
  <pedido cod="1">
    <fecha>01-01-2013</fecha>
    <pu>45.5</pu>
    <cantidad>2</cantidad>
    <descripcion>Botella de Vino</descripcion>
    <tipo>C</tipo>
  </pedido>
  <pedido cod="2">
    <fecha>31-12-2012</fecha>
    <pu>25</pu>
    <cantidad>1</cantidad>
    <descripcion>Menu Ejecutivo</descripcion>
    <tipo>A</tipo>
  </pedido>
</pedidos>');
```

Código A.89: Ejemplo de inserción en tabla con columna XMLType en ORACLE [17]

Es posible insertar un documento a partir de un archivo XML existente en el Sistema Operativo o en el XML DB Repository, un ejemplo se describe en el código A.90.

```
INSERT INTO <tabla> VALUES
(XMLType(bfilename('<DIR>','<archivo.xml>'),nls_charset_id('AL32UTF8')));
```

Código A.90: Ejemplo de inserción en tabla con columna XMLType desde archivo existente [17]

Existen algunas funciones que permiten dar mantenimiento a los elementos de un documento XML ya registrado. Se hará una demostración de tres de ellos. Para Agregar un nuevo elemento pedido sobre el documento XML se debe utilizar la función `appendchildxml()`, un ejemplo de uso en el código A.91. En el código A.92 se modifica el `pu` (precio unitario) del nuevo elemento pedido ingresado del valor de 30 a 20 y por último en el código A.93 se elimina el último elemento ingresado.

```
UPDATE reserva set pedido=appendchildxml(pedido,'/pedidos',
'<pedido cod="3">
  <fecha>31-12-2012</fecha>
  <pu>30</pu>
  <cantidad>1</cantidad>
  <descripcion>xxx</descripcion>
  <tipo>B</tipo>
</pedido>')
where id=1;
```

Código A.91: Ejemplo de función AppendChildxml en ORACLE [17]

```
update reserva set pedido=updatexml(pedido,'/pedidos[1]/pedido[3]/pu/text()'
,20)
where id=1;
```

Código A.92: Ejemplo de UPDATE a columna XMLTYPE utilizando posiciones en ORACLE [17]

```
UPDATE reserva set pedido=deletexml(pedido,'/pedidos[1]/pedido[3]');
```

Código A.93: Ejemplo de Actualización eliminando nodo en columna XMLTYPE ORACLE [17]

En el código A.94 se obtiene el documento XML como String, en cambio en el código A.95 como CLOB y en el código A.96 se crea un String o CLOB a partir de un contenido.

```
select id, r.PEDIDO.getStringVal() from reserva r;
```

Código A.94: Ejemplo de obtención de documento XML como String en ORACLE [17]

```
select id, r.PEDIDO. getClobVal() from reserva r;
```

Código A.95: Ejemplo de obtención de documento XML como Clob en ORACLE [17]

```
select xmlserialize (DOCUMENT|CONTENT r.PEDIDO as CLOB|VARCHAR|  
VARCHAR2) from reserva r;
```

Código A.96: Ejemplo de creación de String o Clob desde un contenido en ORACLE [17]

Para obtener nodos específicos de un documento se debe utilizar XPath. En el ejemplo de la figura A.2 se obtienen todos los pu (precios unitarios) de la reserva con id=1. En la figura A.3 se muestra un ejemplo de cómo obtener aquellos pedidos que han superado un precio unitario de 48 de tipo A. Y por último en el ejemplo descrito en la figura A.4 obtiene aquellos pedidos cuyo atributo cod sea diferente del valor de 3.

```
SQL> select extract(pedido, '/pedidos/pedido/pu')
       from reserva r where id=1;

EXTRACT(PEDIDO, '/PEDIDOS/PEDIDO/PU')
-----
<pu>45.5</pu>
<pu>25</pu>
```

Figura A.2. Ejemplo XPath 1 [17]

```
SQL> select id, r.pedido
       from reserva r
       where xmlexists('/pedidos/pedido[pu>30 and tipo="A"]/descripcion' passing pedido);

no rows selected
```

Figura A.3. Ejemplo XPath 2 [17]

```
SQL> select id,r.pedido.extract('/pedidos/pedido[@cod!=3]/descripcion')
      from reserva r;
```

ID

```
R.PEDIDO.EXTRACT(' /PEDIDOS/PEDIDO [@COD!=3]/DESCRIPCION' )
```

```
<descripcion>Botella de Vino</descripcion>
<descripcion>Menu Ejecutivo</descripc
```

Figura A.4. Ejemplo XPath 3 [17]

### A.4.3. ¿Como almacenar XML en Oracle?

#### ¿XMLType o tablas relacionales?

Esta decisión inicial puede generalmente estar basada en el formato de los datos XML y la fidelidad en los requerimientos DOM (Document Object Model) en el contenido XML. Para el formato, los documentos XML generalmente pueden ser categorizados como documentos “centrado en datos” o “centrados en documentos”.

Los documentos centrados en datos son categorizados por la estructura regular de los datos, en donde la unidad más pequeña es un elemento XML con contenido simple o un atributo XML. En dicho documento XML, hay poca o ninguna mezcla de contenido (por ejemplo, etiquetas dentro del contenido de un elemento XML cadena). Además, la fidelidad DOM del documento no es requerida de conservar. El código A.97 es un ejemplo de un documento XML centrado en datos.

```
<?xml version="1.0"?>
<purchaseOrder orderDate="1999-10-20">
  <shipTo country="US"></shipTo>
  <billTo country="US"></billTo>
  <items>
    <item partNum="872-AA">
      <productName>Lawnmower</productName>
      <quantity>1</quantity>
      <USPrice>148.95</USPrice>
      <comment>Confirm this is electric</comment>
    </item>
  </items>
</purchaseOrder>
```

Código A.97: Ejemplo de documentos XML centrado en datos

Todos los datos presentados en el código A.97 son representados como elementos XML o como atributos para elementos XML.

La mayoría de los documentos XML centrados en datos son creados basados en datos relacionales para intercambiar y compartir datos entre aplicaciones.

Del mismo modo, para documentos centrados en datos, no existe una necesidad de mantener datos en XML una vez que estos son almacenados en la base de datos. Si elementos y atributos XML son almacenados correctamente en columnas de las tablas relacionales, se tiene el almacenamiento optimizado para su posterior procesamiento, lo que evita la sobrecarga de mantener y gestionar la estructura XML en la base de datos. Siempre que se necesite, los datos en XML para el intercambio de datos y la publicación en la Web, se puede construir Vistas XMLType para envolver los datos en XML.

### **Modelos de almacenamiento XMLType**

Este tipo de dato abstracto permite almacenar datos en formato XML, brindando la posibilidad de utilizar esquemas XML, XPath, XQuery, XSLT, indexación y particionamiento de documentos XML. En las versiones anteriores de Oracle Database 9i y 10g los documentos XML se almacenan como CLOB (Character Large Object) o como objetos. A partir de la versión Oracle Database 11g se agregó una nueva posibilidad, almacenarlos en formato binario (Binary XML), que desde la versión 11.2.0.2. es el formato por defecto [15].

#### **■ Almacenamiento no estructurado**

Los datos XMLType son almacenados como un tipo CLOB. Los documentos XML son almacenados preservando el documento original, inclusive los espacios en blanco. De esta forma, se mantiene fidelidad de contenido o de documento. Además, Oracle brinda la posibilidad de asociar un esquema a dicha forma de almacenamiento. Entrega un muy bueno desempeño en las operaciones de inserción y recuperación de documentos completos. Es una opción que proporciona una gran flexibilidad, pero por otro lado, requiere una sobrecarga en el procesamiento cuando se necesita consultar su contenido, como por ejemplo cuando se utilizan funciones como XMLType.Extract y XMLType.ExistsNode. Evaluar estas funciones requiere construir el árbol XML DOM en memoria y sobre este evaluar las expresiones XPath. También, toda operación de actualización se debe realizar a nivel de documento [15].

### ■ Almacenamiento estructurado

Los datos XMLType son almacenados como un conjunto de objetos. Los documentos XML mantienen fidelidad DOM. Esta alternativa también es mencionada como almacenamiento Objeto Relacional (OR) [16]. Es común y conveniente almacenarlos con un esquema asociado, puesto que permite acelerar las consultas y las actualizaciones de granularidad fina. Esto provoca que las aplicaciones que lo utilizan deban ajustarse a un esquema de datos bien estructurado y rígido. Por otro lado, presenta varias ventajas en comparación con el almacenamiento no estructurado. Se logra una optimización del manejo de memoria, se reducen los requerimientos de almacenamiento, y se pueden hacer actualizaciones a nivel de detalle. Es necesario mencionar que las mejoras de esta alternativa tiene su contrapartida, el incremento de la sobrecarga durante la inserción y la recuperación de los datos, además de la reducción de la flexibilidad en términos de estructura [15].

### ■ Almacenamiento binario

Esta forma de almacenamiento provista por la versión 11g es introducida con la intención de compensar ventajas y desventajas de las dos alternativas anteriores. Así, provee una gran flexibilidad en cuanto a la estructura sin deteriorar el desempeño. Concretamente, mantiene un buen rendimiento en la actualización, indexación y extracción de fragmentos; así como en las consultas, puesto que parsea el documento antes de almacenarlo. La tabla A-4 presenta una comparación muy ilustrativa entre las tres opciones de almacenamiento descriptas en Oracle 11g [15].

### Elección de la estructura de almacenamiento

	<b>CLOB</b>	<b>Objeto Relacional</b>	<b>XML Binario</b>
Consulta	pobre	excelente	bueno / excelente
DML	pobre	bueno / excelente	excelente
Recuperación de documentos	excelente	bueno / excelente	excelente
Flexibilidad de esquema	bueno	pobre	excelente
Fidelidad de documento	excelente	pobre	bueno / excelente

Tabla A-4: Comparación de los diferentes modelos de almacenamiento [15]

Cada modelo de almacenamiento posee un conjunto de características que lo hacen más o menos apropiado según el tipo de datos a manipular. Sin embargo,



aunque la naturaleza de los datos es un aspecto fundamental a considerar, el uso que se les dará a esos datos también constituye un factor decisivo.

- **Datos Estructurados**

Corresponde a aquellos datos que tienen una estructura regular y granularidad fina. Los datos contenidos en facturas de ventas y en cuentas de un banco son ejemplos de esta categoría.

- **Datos Semiestructurados o No Estructurados**

Se caracterizan por tener una estructura menos regular, una granularidad más gruesa y con contenidos mezclados; tales como los datos contenidos en correos electrónicos, libros, currículos y advertencias. En general, los datos están organizados para ser usados por personas. Por otra parte, como se comentó con anterioridad, las aplicaciones que usan los datos también pueden ser de diferente naturaleza:

- **Aplicaciones centradas en datos**

Estas aplicaciones necesitan conocer la estructura de los datos. En general, los datos son altamente estructurados. Luego, la aplicación puede tomar ventaja en este sentido. Es común que los datos se ajustan a un esquema XML.

- **Aplicaciones centrada en documentos**

En este caso, es común que las aplicaciones necesiten mantener una copia exacta de los documentos, como sucede por ejemplo en el ámbito judicial, médico, etc. Luego, bajo la hipótesis de contar con datos estructurados, por un lado cabe la situación que las aplicaciones no necesiten conocer esa estructura, y por otro, que sí lo demanden. En el primer caso, el modelo de almacenamiento como CLOB será la alternativa más aconsejable, mientras que en el segundo, el modelo apropiado sería el binario.

## Anexo B

### Documentación caso de estudio

#### B.1. Documentación DER

##### Documentación de entidades

<b>Nombre:</b>	<b>personal</b>	
<b>Descripción:</b>	Personal del colegio	
<b>Tipo:</b>	Normal	
<b>Atributos:</b>	<u>rut</u> :	rut del personal (atributo compuesto número correlativo y dígito verificador)
	nombre_completo:	Nombre completo del personal (atributo compuesto consta de nombres, apellido paterno y materno)
	fnac:	Fecha de nacimiento del personal
	edad:	Edad del personal (atributo derivado)
	direccion:	Dirección del personal (atributo compuesto consta: calle, número, código postal, ciudad y región)
	celular:	Números de celular personal (atributo multivaluado)
	mail:	Mails del personal (atributo multivaluado)

<b>Nombre:</b>	<b>docente</b>	
<b>Descripción:</b>	Personal docente del colegio	
<b>Tipo:</b>	Especialización de personal	
<b>Atributos:</b>	cant_cursos:	Cantidad de cursos en que hace clases (atributo derivado)

<b>Nombre:</b>	<b>no_docente</b>
Descripción:	Personal no docente del colegio
Tipo:	Especialización de personal
Atributos:	No tiene

<b>Nombre:</b>	<b>cargo</b>
Descripción:	Cargo que posee el personal no docente
Tipo:	Normal
Atributos:	<u>id_cargo</u> : Código identificador del cargo nom_c: Nombre del cargo desc_c: Descripción del cargo

<b>Nombre:</b>	<b>nivel</b>
Descripción:	Nivel de enseñanza de un curso por ejemplo 1ero básico, 2do medio, etc.
Tipo:	Normal
Atributos:	<u>id_nivel</u> : Código identificador del nivel nom_niv: Nombre del nivel

<b>Nombre:</b>	<b>curso</b>
Descripción:	Curso del colegio por ejemplo 1ero básico A, 2do medio B, etc.
Tipo:	Débil con respecto a nivel, agregación compuesta por curso, asignatura y relacionamiento curso_asignatura
Atributos:	<u>letra</u> : Letra identificadora del curso año: Año del curso por ejemplo 1ero básico A 2013, 2do medio B 2011

<b>Nombre:</b>	<b>asignatura</b>
Descripción:	Asignatura dictada en un curso en particular
Tipo:	Normal, agregación compuesta por curso, asignatura y relacionamiento curso_asignatura
Atributos:	<u>id_asig</u> : Código identificador de la asignatura nom_asig: Nombre de la asignatura

<b>Nombre:</b>	<b>programa_tipo</b>
Descripción:	Tipo de programa educacional ejemplo: Unidad, Aprendizaje esperado, etc.
Tipo:	Normal
Atributos:	<u>id_tp</u> : Código identificador del tipo de programa <u>nom_pt</u> : Nombre del tipo de programa <u>abrev</u> : Abreviatura del tipo de programa Ej: U, AE, OFT, etc.

<b>Nombre:</b>	<b>detalle_programa</b>
Descripción:	Detalle del programa educacional de una asignatura ejemplo: Sumas y restas (unidad, 1ero básico).
Tipo:	Normal
Atributos:	<u>id_dp</u> : Código identificador del detalle de programa <u>nomdp</u> : Nombre del detalle de programa, ejemplo unidad1 <u>desc_dp</u> : Descripción por ejemplo "Sumas y restas"

<b>Nombre:</b>	<b>planificacion</b>
Descripción:	Planificación realizada por un docente para una asignatura de un curso en particular
Tipo:	Normal
Atributos:	<u>id_plan</u> : Código identificador de la planificación <u>tipo_plan</u> : Tipo de planificación: Anual, por unidad o por clase <u>fcreacion</u> : Fecha de creación de la planificación <u>ult_estado</u> : Último estado por el que paso la planificación <u>ult_modif</u> : Fecha de la última modificación de la planificación

<b>Nombre:</b>	<b>estado</b>
Descripción:	Estados por los que pasa una planificación
Tipo:	Normal
Atributos:	<u>id_estado</u> : Código identificador del estado <u>nom_estado</u> : Nombre del estado por ejemplo: Vigente, rechazado, borrador

<b>Nombre:</b>	<b>plan_anual</b>
Descripción:	Planificación del tipo anual realizada por un docente para una asignatura de un curso en particular
Tipo:	Especialización de planificación
Atributos:	No tiene

<b>Nombre:</b>	<b>plan_unidad</b>
Descripción:	Planificación del tipo por unidad realizada por un docente para una asignatura de un curso en particular
Tipo:	Especialización de planificación
Atributos:	No tiene

<b>Nombre:</b>	<b>plan_clase</b>
Descripción:	Planificación del tipo clase a clase realizada por un docente para una asignatura de un curso en particular
Tipo:	Especialización de planificación
Atributos:	No tiene

<b>Nombre:</b>	<b>periodo</b>
Descripción:	periodos (de tiempo) de una planificación anual
Tipo:	Normal
Atributos:	<u>id_periodo</u> : Código identificador del periodo mes_ini: Mes de inicio del periodo mes_ter: Mes de término del periodo duracion: Duración del periodo en horas aprevios: Aprendizajes previos que debe poseer el curso (atributo multivaluado)

<b>Nombre:</b>	<b>unidad</b>
Descripción:	unidades (a pasar) en una planificación por unidad
Tipo:	Normal
Atributos:	<u>id_unidad</u> : Código identificador de la unidad fechai: Fecha de inicio de la unidad fechat: Fecha de término de la unidad duracion: Duración del periodo en horas aprevios: Aprendizajes previos que debe poseer el curso (atributo multivaluado)

<b>Nombre:</b>	<b>clase</b>
Descripción:	clases de una planificación clase a clase
Tipo:	Normal
Atributos:	<u>id_clase</u> : Código identificador de la clase <u>fecha</u> : Fecha en que se realizará la clase <u>horario</u> : horario en que se hará la clase (atributo compuesto consta de día, hora inicio, hora término y duración) <u>aprevios</u> : Aprendizajes previos que debe poseer el curso (atributo multivaluado)

<b>Nombre:</b>	<b>metodologia</b>
Descripción:	metodología a emplear en un periodo, unidad o actividad
Tipo:	Normal
Atributos:	<u>id_met</u> : Código identificador de la metodología <u>nom_met</u> : Nombre de la metodología <u>desc_met</u> : Descripción de la metodología

<b>Nombre:</b>	<b>recurso</b>
Descripción:	recurso a utilizar en un periodo, unidad o actividad
Tipo:	Normal
Atributos:	<u>id_rec</u> : Código identificador del recurso <u>nom_rec</u> : Nombre del recurso <u>desc_rec</u> : Descripción del recurso

<b>Nombre:</b>	<b>actividad</b>
Descripción:	actividad a realizar en una clase
Tipo:	Normal
Atributos:	<u>id_act</u> : Código identificador de la actividad <u>desc_act</u> : Descripción de la actividad <u>duracion</u> : Duración de la actividad

### Documentación de los relacionamientos

<b>Nombre:</b>	<b>curso_asignatura</b>
Tipo:	Normal
Entidades que participan:	curso asignatura
Cardinalidad:	1 curso tiene muchas asignaturas 1 asignatura pertenece sólo a 1 curso
Atributos:	No tiene

<b>Nombre:</b>	<b>jefe</b>
Descripción:	Profesor jefe del curso
Tipo:	Normal
Entidades que participan:	docente curso
Cardinalidad:	1 curso tiene sólo un profesor jefe 1 profesor puede ser profesor jefe de 0 o 1 curso
Atributos:	No tiene

<b>Nombre:</b>	<b>nivel_curso</b>
Descripción:	Nivel de enseñanza al que pertenece el curso
Tipo:	Débil
Entidades que participan:	nivel curso
Cardinalidad:	1 curso tiene sólo 1 nivel 1 nivel puede tener muchos cursos
Atributos:	No tiene

<b>Nombre:</b>	<b>no_docente_cargo</b>
Descripción:	Cargo que posee el no docente
Tipo:	Normal
Entidades que participan:	no_docente cargo
Cardinalidad:	1 no docente tiene sólo un cargo 1 cargo puede tener asociado 1 o más no docentes
Atributos:	No tiene

<b>Nombre:</b>	<b>detalle_programa_asignatura</b>
Descripción:	Asignatura a la que pertenece el detalle de programa
Tipo:	Normal
Entidades que participan:	detalle_programa asignatura
Cardinalidad:	1 detalle de programa tiene sólo una asignatura 1 asignatura puede tener asociado 1 o más detalles de programas
Atributos:	No tiene

<b>Nombre:</b>	<b>detalle_programa_programa_tipo</b>
Descripción:	Tipo de programa al que pertenece el detalle de programa
Tipo:	Normal
Entidades que participan:	detalle_programa programa_tipo
Cardinalidad:	1 detalle de programa tiene sólo un tipo de programa 1 tipo de programa puede tener asociado 1 o más detalles de programa
Atributos:	No tiene

<b>Nombre:</b>	<b>crea</b>
Descripción:	Un docente crea una planificación
Tipo:	Normal
Entidades que participan:	planificacion docente
Cardinalidad:	1 planificación tiene sólo un docente 1 docente puede crear 1 o más planificaciones
Atributos:	No tiene

<b>Nombre:</b>	<b>planificacion_estado</b>
Descripción:	Estados por los que pasa una planificación
Tipo:	Normal
Entidades que participan:	planificacion estado
Cardinalidad:	1 planificación puede pasar por muchos estados 1 estado puede estar asociado a 1 o más planificaciones
Atributos:	fcambio: Fecha cambio de estado (atributo multivaluado)



<b>Nombre:</b>	<b>evaluacion</b>
Descripción:	Un no docente evalúa si una planificación si esta correcta
Tipo:	Normal
Entidades que participan:	planificacion no_docente
Cardinalidad:	1 planificación puede ser evaluada muchas veces por un no docente 1 no docente puede evaluar muchas veces una planificación
Atributos:	EvaDetalle: Detalle de la evaluación (atributo multivaluado compuesto, consta de fecha de evaluación y un comentario)

<b>Nombre:</b>	<b>plan_anual_periodo</b>
Descripción:	Una planificación anual posee periodos
Tipo:	Normal
Entidades que participan:	plan_anual periodo
Cardinalidad:	1 periodo pertenece sólo a una planificación anual 1 planificación anual puede tener 1 o más periodos
Atributos:	No tiene

<b>Nombre:</b>	<b>plan_unidad_unidad</b>
Descripción:	Una planificación por unidad posee unidades
Tipo:	Normal
Entidades que participan:	plan_unidad unidad
Cardinalidad:	1 unidad pertenece sólo a una planificación por unidad 1 planificación por unidad puede tener 1 o más unidades
Atributos:	No tiene

<b>Nombre:</b>	<b>plan_clase_clase</b>
Descripción:	Una planificación clase a clase posee clases
Tipo:	Normal
Entidades que participan:	plan_clase clase
Cardinalidad:	1 clase pertenece sólo a una planificación clase a clase 1 planificación clase a clase puede tener 1 o más clases
Atributos:	No tiene

<b>Nombre:</b>	<b>clase_actividad</b>
Descripción:	Actividades a realizar en una clase
Tipo:	Normal
Entidades que participan:	clase actividad
Cardinalidad:	1 actividad pertenece sólo a una clase 1 clase puede realizar 1 o más actividades
Atributos:	No tiene

<b>Nombre:</b>	<b>clase_detalle_programa</b>
Descripción:	En una clase se enseñarán detalles de programas educativos
Tipo:	Normal
Entidades que participan:	clase detalle_programa
Cardinalidad:	En 1 clase se pueden enseñar muchos detalles de programa 1 detalle de programa puede ser enseñado en 1 o más clases
Atributos:	No tiene

<b>Nombre:</b>	<b>periodo_detalle_programa</b>
Descripción:	En un periodo se enseñarán detalles de programas educativos
Tipo:	Normal
Entidades que participan:	periodo detalle_programa
Cardinalidad:	En 1 periodo se pueden enseñar muchos detalles de programa 1 detalle de programa puede ser enseñado en 1 o más periodos
Atributos:	No tiene

<b>Nombre:</b>	<b>unidad_detalle_programa</b>
Descripción:	En una unidad se enseñarán detalles de programas educativos
Tipo:	Normal
Entidades que participan:	unidad detalle_programa
Cardinalidad:	En 1 unidad se pueden enseñar muchos detalles de programa 1 detalle de programa puede ser enseñado en 1 o más unidades
Atributos:	No tiene

<b>Nombre:</b>	<b>unidad_metodologia</b>
Descripción:	En una unidad se aplicarán metodologías de enseñanza
Tipo:	Normal
Entidades que participan:	unidad metodologia
Cardinalidad:	En 1 unidad se pueden emplear muchas metodologías 1 metodología puede ser aplicada en 1 o más unidades
Atributos:	No tiene

<b>Nombre:</b>	<b>periodo_metodologia</b>
Descripción:	En un periodo se aplicarán metodologías de enseñanza
Tipo:	Normal
Entidades que participan:	periodo
Cardinalidad:	En 1 periodo se pueden emplear muchas metodologías 1 metodología puede ser aplicada en 1 o más periodos
Atributos:	No tiene

<b>Nombre:</b>	<b>unidad_recurso</b>
Descripción:	En una unidad se puede utilizar recursos educativos
Tipo:	Normal
Entidades que participan:	unidad recurso
Cardinalidad:	En 1 unidad se pueden utilizar muchos recursos educativos 1 recurso puede ser utilizado en 1 o más unidades
Atributos:	No tiene

<b>Nombre:</b>	<b>periodo_recurso</b>
Descripción:	En un periodo se puede utilizar recursos educativos
Tipo:	Normal
Entidades que participan:	periodo recurso
Cardinalidad:	En 1 periodo se pueden utilizar muchos recursos educativos 1 recurso puede ser utilizado en 1 o más periodos
Atributos:	No tiene

<b>Nombre:</b>	<b>actividad_recurso</b>
Descripción:	En una actividad se puede utilizar recursos educativos
Tipo:	Normal
Entidades que participan:	actividad recurso
Cardinalidad:	En 1 actividad se pueden utilizar muchos recursos educativos 1 recurso puede ser utilizado en 1 o más actividades
Atributos:	No tiene

<b>Nombre:</b>	<b>actividad_metodologia</b>
Descripción:	En una actividad se aplicarán metodologías de enseñanza
Tipo:	Normal
Entidades que participan:	actividad metodologia
Cardinalidad:	En 1 actividad se pueden emplear muchas metodologías 1 metodología puede ser aplicada en 1 o más actividades
Atributos:	No tiene

<b>Nombre:</b>	<b>curso_asignatura_docente</b>
Descripción:	Un docente dicta una asignatura en un curso específico
Tipo:	Normal
Entidades que participan:	curso asignatura docente
Cardinalidad:	En un 1 curso hay muchas asignaturas que son dictadas por 1 docente Muchas asignaturas pertenecen a 1 curso y es dictada por 1 docente 1 docente puede dictar muchas asignaturas de 1 curso
Atributos:	No tiene

<b>Nombre:</b>	<b>curso_asignatura_planificacion</b>
Descripción:	Se crean planificaciones para una asignatura de un curso en específico
Tipo:	Normal
Entidades que participan:	curso asignatura planificacion
Cardinalidad:	En un 1 curso hay muchas asignaturas de las que se pueden crear muchas planificaciones 1 asignatura de 1 curso en particular puede tener muchas planificaciones asociadas 1 planificación es creada para 1 asignatura de 1 curso en particular
Atributos:	No tiene

## B.2. Documentación modelo relacional

### Documentación de tablas

#### Nombre Tabla: nivel

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
id_nivel	number	PK	NN,U	Identificador de nivel de enseñanza
nom_nivel	varchar(30)		NN	Nombre del nivel de enseñanza

#### Nombre Tabla: programa\_tipo

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
id_pt	<b>number</b>	PK	NN,U	Identificador de tipo de programa
nom_pt	varchar2(30)		NN	Nombre del tipo de programa
abrev	varchar2(10)		NN	Abreviatura de tipo de programa

**Nombre Tabla: curso**

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
id_nivel	number	PK FK1	NN,U	Identificador de nivel de enseñanza del curso
letra	char(1)	PK	NN	Letra del curso
year	number		NN	Año del curso
num_correlativo	number	FK2	NN,U	número de rut del profesor jefe
dv	char(1)	FK2	NN	dígito verificador del profesor jefe

**Nombre Tabla: personal**

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
num_correlativo	number	PK	NN,U	número de rut
dv	char(1)	PK	NN	dígito verificador
nombres	varchar2(50)		NN	Nombres de la persona
apaterno	varchar2(30)		NN	Apellido parterno de la persona
amaterno	varchar2(30)		NN	Apellido materno de la persona
fnac	date		NN	Fecha de nacimiento de la persona
calle	varchar2(30)		NN	Calle de la dirección de la persona
num	varchar2(10)		NN	Número de la dirección de la persona
cod_postal	varchar2(10)		NN	Código postal de la dirección de la persona
ciudad	varchar2(30)		NN	Ciudad donde vive la persona
region	varchar2(30)		NN	Región donde vive la persona

**Nombre Tabla: personalmail**

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
num_correlativo	number	PK FK	NN,U	número de rut
dv	char(1)	PK, FK	NN	dígito verificador
mail	varchar2(100)	PK	NN	Correo de la persona

**Nombre Tabla: personalcelular**

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
num_correlativo	number	PK, FK	NN,U	número de rut
dv	char(1)	PK FK	NN	dígito verificador
celular	number	PK	NN	Celular de la persona

**Nombre Tabla: detalle\_programa**

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
id_dp	number	PK	NN,U	Identificador de detalle de programa
nom_dp	varchar2(50)		NN	Nombre del detalle de programa
desc_dp	varchar2(1000)		NN	Descripción del detalle de programa
id_pt	number	FK1	NN,U	Identificador de tipo de programa
id_asig	number	FK2	NN,U	Identificador de la asignatura del detalle de programa



**Nombre Tabla: docente**

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
personal_num_correlativo	number	PK FK	NN,U	número de rut
personal_dv	char(1)	PK, FK	NN	dígito verificador

**Nombre Tabla: no\_docente**

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
personal_num_correlativo	number	PK FK1	NN,U	número de rut
personal_dv	char(1)	PK FK1	NN	dígito verificador
id_cargo	number	FK2	NN	Identificador del cargo del no docente

**Nombre Tabla: cargo**

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
id_cargo	number	PK	NN,U	Identificador del cargo
nom_c	varchar2(30)		NN	Nombre del cargo
desc_c	varchar2(500)		NN	Descripción del cargo

**Nombre Tabla: estado**

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
id_estado	number	PK	NN,U	Identificador del estado por el que pasa una planificación
nom_estado	varchar2(30)		NN,U	Nombre del estado por el que pasa una planificación

**Nombre Tabla: asignatura**

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
id_asig	number	PK	NN,U	Identificador de la asignatura
nom_asig	varchar2(30)		NN	Nombre de la asignatura
id_nivel	number	FK1 FK2	NN	Identificador del nivel al que pertenece la asignatura
letra	char(1)	FK2	NN	Letra del curso al que pertenece la asignatura
num_correlativo	number	FK3	NN	Número de rut del profesor que dicta asignatura
dv	char(1)	FK3	NN	Dígito verificador del rut de profesor que dicta asignatura

**Nombre Tabla: metodologia**

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
id_met	number	PK	NN,U	Identificador de la metodología
nom_met	varchar2(100)		NN,U	Nombre de la metodología
desc_met	varchar2(500)		NN,U	Descripción de la metodología

**Nombre Tabla: planificacion**

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
id_plan	number	PK	NN,U	Identificador de una planificación
tipo_plan	varchar2(10)		NN,U	Tipo de planificación anual, unidad o clase
utl_estado	varchar2(30)		NN,U	Estado actual de la planificación
fcreacion	date		NN,U	Fecha de creación de la planificación
ult_modif	date		NN,U	Fecha de la última vez que se modificó la planificación
num_correlativo	number	FK1	NN,U	Número de rut del profesor que crea la planificación
dv	char(1)	FK1	NN,U	Dígito verificador del profesor que crea la planificación
id_asig	number	FK2	NN,U	Identificador de la asignatura a la que pertenece la planificación

**Nombre Tabla: plan\_anual**

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
id_plan	number	PK FK	NN,U	Identificador de la planificación anual

**Nombre Tabla: plan\_unidad**

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
id_plan	number	PK FK	NN,U	Identificador de la planificación por unidad

**Nombre Tabla: plan\_clase**

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
id_plan	number	PK, FK	NN,U	Identificador de la planificación clase a clase

**Nombre Tabla: periodo**

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
id_periodo	number	PK	NN,U	Identificador del periodo de la planificación anual
duracion	number		NN,U	Duración del periodo
mes_ini	date		NN,U	Mes de inicio del periodo
mes_ter	date		NN,U	Mes de término del periodo
id_plan	number	FK1 FK2	NN,U	Identificador de la planificación anual

**Nombre Tabla: periodo\_aprevious**

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
id_periodo	number	PK FK	NN,U	Identificador del periodo de la planificación anual
aprevio	varchar2 (1000)	PK	NN,U	Aprendizaje previo

**Nombre Tabla: unidad**

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
id_unidad	number	PK	NN,U	Identificador de la unidad de la planificación por unidad
fechai	date		NN,U	Fecha de inicio de la unidad
fechat	date		NN,U	Fecha de término de la unidad
id_plan	number	FK1 FK2	NN,U	Identificador de la planificación por unidad

**Nombre Tabla: unidad\_aprevios**

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
id_unidad	number	PK FK	NN,U	Identificador de la unidad de la planificación por unidad
aprevio	varchar2 (1000)	PK	NN,U	Aprendizaje previo

**Nombre Tabla: clase\_aprevios**

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
id_clase	number	PK FK	NN,U	Identificador de la clase de la planificación clase a clase
aprevio	varchar2 (1000)	PK	NN,U	Aprendizaje previo

**Nombre Tabla: clase**

<b>Nombre atributo</b>	<b>Tipo dato</b>	<b>PK FK</b>	<b>Nulo Único</b>	<b>Documentación</b>
id_clase	number	PK	NN,U	Identificador de la clase de la planificación clase a clase
duracion	number		NN,U	Duración de la clase
dia	number		NN,U	Día en que se realizará la clase
hi	date		NN,U	Hora de inicio de la clase
ht	date		NN,U	Hora de término de la clase
fecha	date		NN,U	Fecha de la clase
id_plan	number	FK1 FK2	NN,U	Identificador de la planificación clase a clase

**Nombre Tabla: planificacion\_estado**

<b>Nombre atributo</b>	<b>Tipo dato</b>	<b>PK FK</b>	<b>Nulo Único</b>	<b>Documentación</b>
id_plan	number	PK FK1	NN,U	Identificador de la planificación anual
id_estado	number	PK FK2	NN,U	Identificador del estado por el que pasa una planificación

**Nombre Tabla: metodologia\_unidad**

<b>Nombre atributo</b>	<b>Tipo dato</b>	<b>PK FK</b>	<b>Nulo Único</b>	<b>Documentación</b>
unidad_id_unidad	number	PK FK1	NN,U	Identificador de la unidad
metodologia_id_met	number	PK FK2	NN,U	Identificador de la metodología

**Nombre Tabla: planificacion\_estado\_fcambio**

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
id_plan	number	PK FK1 FK2	NN,U	Identificador de la planificación anual
id_estado	number	PK FK1 FK3	NN,U	Identificador del estado por el que pasa una planificación
fcambio	date	PK	NN,U	Fecha en la que se realizó el cambio de estado en una planificación

**Nombre Tabla: evaluacion**

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
num_correlativo	number	PK FK1 FK2	NN,U	Número de rut del no docente que validó la evaluación
dv	char(1)	PK FK1 FK2	NN,U	Dígito verificador del no docente que validó la planificación
planificacion_id_plan	number	PK FK3	NN,U	Identificador de la planificación validada

**Nombre Tabla: metodologia\_periodo**

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
periodo_id_periodo	number	PK FK1	NN,U	Identificador del periodo
metodologia_id_met	number	PK FK2	NN,U	Identificador de la metodología

**Nombre Tabla: evaluacion\_detalle**

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
num_correlativo	number	PK FK1 FK2 FK3	NN,U	Número de rut del no docente que validó la evaluación
dv	char(1)	PK FK1 FK2 FK3	NN,U	Dígito verificador del no docente que validó la planificación
id_plan	number	PK FK3 FK4	NN,U	Identificador de la planificación validada
fecha_eva	date	PK	NN,U	Fecha en la que se validó la planificación
comentario	varchar2(255)		NN,U	Comentarios sobre validación de la planificación

**Nombre Tabla: recurso**

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
id_rec	number	PK	NN,U	Identificador del recurso educativo
nom_rec	varchar2(100)		NN,U	Nombre del recurso educativo
desc_rec	varchar2(500)		NN,U	Descripción del recurso educativo

**Nombre Tabla: recurso\_unidad**

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
unidad_id_unidad	number	PK FK1	NN,U	Identificador de la unidad de una planificación
id_rec	number	PK FK2	NN,U	Identificador del recurso educativo



**Nombre Tabla: recurso\_periodo**

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
periodo_id_periodo	number	PK FK1	NN,U	Identificador del periodo de una planificación
id_rec	number	PK FK2	NN,U	Identificador del recurso educativo

**Nombre Tabla: actividad**

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
id_act	number	PK	NN,U	Identificador de una actividad
desc_act	varchar2(500)		NN,U	Descripción de una actividad
duracion	number		NN,U	Duración de una actividad
id_clase	number	FK1	NN,U	Identificador de una clase de una planificación clase a clase
id_met	number	FK2	NN,U	Identificador de una metodología

**Nombre Tabla: recurso\_actividad**

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
id_rec	number	PK FK1	NN,U	Identificador del recurso educativo
actividad_id_act	number	PK FK2	NN,U	Identificador de una actividad

**Nombre Tabla: detalle\_programa\_unidad**

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
detalle_programa_id_dp	number	PK FK1	NN,U	Identificador de detalle de programa
unidad_id_unidad	number	PK FK2	NN	Identificador de una unidad de una planificación por unidad

**Nombre Tabla: detalle\_programa\_periodo**

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
detalle_programa_id_dp	number	PK FK1	NN,U	Identificador de detalle de programa
periodo_id_periodo	number	PK FK2	NN	Identificador de un periodo de una planificación anual

**Nombre Tabla: detalle\_programa\_clase**

Nombre atributo	Tipo dato	PK FK	Nulo Único	Documentación
detalle_programa_id_dp	number	PK FK1	NN,U	Identificador de detalle de programa
clase_id_clase	number	PK FK2	NN	Identificador de una clase de una planificación clase a clase

### B.3. Código SQL de creación de tablas del modelo relacional

El modelo relacional fue implementado en la tablespace IMUNOZ, del servidor oracle del DISC.

```
CREATE TABLE "NIVEL"
("ID_NIVEL" NUMBER(*,0) NOT NULL ENABLE,
"NOM_NIVEL" NUMBER(*,0),
CONSTRAINT "NIVEL_PK" PRIMARY KEY ("ID_NIVEL");
```

Código B.1: DDL Tabla nivel

```
CREATE TABLE "PERSONAL"
("NUM_CORRELATIVO" NUMBER(*,0) NOT NULL ENABLE,
"DV" CHAR(1) NOT NULL ENABLE,
"NOMBRES" VARCHAR2(255),
"APATERNO" VARCHAR2(255),
"AMATERNO" VARCHAR2(255),
"FNAC" DATE,
"CALLE" VARCHAR2(500),
"NUM" VARCHAR2(10),
"COD_POSTAL" VARCHAR2(10),
"CIUDAD" VARCHAR2(255),
"REGION" VARCHAR2(255),
CONSTRAINT "PERSONAL_PK" PRIMARY KEY ("NUM_CORRELATIVO", "DV");
```

Código B.2: DDL Tabla personal

```
CREATE TABLE "PERSONALCELULAR"
("CELULAR" NUMBER(*,0) NOT NULL ENABLE,
"NUM_CORRELATIVO" NUMBER(*,0) NOT NULL ENABLE,
"DV" CHAR(1) NOT NULL ENABLE,
CONSTRAINT "PERSONALCELULAR_PK" PRIMARY KEY ("CELULAR", "NUM_CORRELATIVO", "DV"),
CONSTRAINT "PERSONALCELULAR_PERSONAL_FK" FOREIGN KEY ("NUM_CORRELATIVO", "DV")
REFERENCES "PERSONAL" ("NUM_CORRELATIVO", "DV");
```

Código B.3: DDL Tabla personalcelular

```

CREATE TABLE "PERSONALMAIL"
(  "MAIL" VARCHAR2(500) NOT NULL ENABLE,
  "NUM_CORRELATIVO" NUMBER(*,0) NOT NULL ENABLE,
  "DV" CHAR(1) NOT NULL ENABLE,
  CONSTRAINT "PERSONALMAIL_PK" PRIMARY KEY ("MAIL", "
    NUM_CORRELATIVO", "DV"),
  CONSTRAINT "PERSONALMAIL_PERSONAL_FK" FOREIGN KEY ("
    NUM_CORRELATIVO", "DV") REFERENCES "PERSONAL" ("
    NUM_CORRELATIVO", "DV"));

```

Código B.4: DDL Tabla personalemail

```

CREATE TABLE "DOCENTE"
(
  "NUM_CORRELATIVO" NUMBER(*,0) NOT NULL ENABLE,
  "DV" CHAR(1) NOT NULL ENABLE,
  CONSTRAINT "FK_ASS_2" FOREIGN KEY ("
    PERSONAL_NUM_CORRELATIVO", "PERSONAL_DV")
  REFERENCES "PERSONAL" ("NUM_CORRELATIVO", "DV"));

```

Código B.5: DDL Tabla docente

```

CREATE TABLE "CARGO"
(  "ID_CARGO" NUMBER(*,0) NOT NULL ENABLE,
  "NOM_C" VARCHAR2(255),
  "DESC_C" VARCHAR2(500),
  CONSTRAINT "CARGO_PK" PRIMARY KEY ("ID_CARGO"));

```

Código B.6: DDL Tabla cargo

```

CREATE TABLE "NO_DOCENTE"
(
  "NUM_CORRELATIVO" NUMBER(*,0) NOT NULL ENABLE,
  "DV" CHAR(1) NOT NULL ENABLE,
  CONSTRAINT "NO_DOCENTE_PK" PRIMARY KEY ("
    PERSONAL_NUM_CORRELATIVO", "PERSONAL_DV"),
  CONSTRAINT "FK_ASS_1" FOREIGN KEY ("
    PERSONAL_NUM_CORRELATIVO", "PERSONAL_DV") REFERENCES "
    PERSONAL" ("NUM_CORRELATIVO", "DV"),
  CONSTRAINT "NO_DOCENTE_CARGO_FK" FOREIGN KEY ("ID_CARGO")
  REFERENCES "CARGO" ("ID_CARGO"));

```

Código B.7: DDL Tabla no\_docente

```

CREATE TABLE "CURSO"
(
  "LETRA" CHAR(1) NOT NULL ENABLE,
  "YEAR" NUMBER(*,0),
  "ID_NIVEL" NUMBER(*,0) NOT NULL ENABLE,
  "NUM_CORRELATIVO" NUMBER(*,0),
  "DV" CHAR(1),
  CONSTRAINT "CURSO_PK" PRIMARY KEY ("LETRA", "ID_NIVEL"),
  CONSTRAINT "CURSO_DOCENTE_FK" FOREIGN KEY ("
    NUM_CORRELATIVO", "DV") REFERENCES "DOCENTE" ("
    PERSONAL_NUM_CORRELATIVO", "PERSONAL_DV"),
  CONSTRAINT "CURSO_NIVEL_FK" FOREIGN KEY ("ID_NIVEL")
    REFERENCES "NIVEL" ("ID_NIVEL"));

```

Código B.8: DDL Tabla curso

```

CREATE TABLE "ASIGNATURA"
(
  "ID_ASIG" NUMBER(*,0) NOT NULL ENABLE,
  "NOM_ASIG" VARCHAR2(255),
  "LETRA" CHAR(1),
  "ID_NIVEL" NUMBER(*,0),
  "NUM_CORRELATIVO" NUMBER(*,0),
  "DV" CHAR(1),
  CONSTRAINT "ASIGNATURA_PK" PRIMARY KEY ("ID_ASIG"),
  CONSTRAINT "ASIGNATURA_CURSO_FK" FOREIGN KEY ("LETRA", "
    ID_NIVEL") REFERENCES "CURSO" ("LETRA", "ID_NIVEL"),
  CONSTRAINT "ASIGNATURA_DOCENTE_FK" FOREIGN KEY ("
    NUM_CORRELATIVO", "DV") REFERENCES "DOCENTE" ("
    PERSONAL_NUM_CORRELATIVO", "PERSONAL_DV"));

```

Código B.9: DDL Tabla asignatura

```

CREATE TABLE "PROGRAMA_TIPO"
(
  "ID_PT" NUMBER(*,0) NOT NULL ENABLE,
  "NOM_PT" VARCHAR2(255),
  "ABREV" VARCHAR2(10),
  CONSTRAINT "PROGRAMA_TIPO_PK" PRIMARY KEY ("ID_PT"));

```

Código B.10: DDL Tabla programa\_tipo

```

CREATE TABLE "DETALLE_PROGRAMA"
  (  "ID_DP" NUMBER(*,0) NOT NULL ENABLE,
    "NOM_DP" VARCHAR2(255),
    "DESC_DP" VARCHAR2(1000),
    "ID_PT" NUMBER(*,0),
    "ID_ASIG" NUMBER(*,0),
    CONSTRAINT "DETALLE_PROGRAMA_PK" PRIMARY KEY ("ID_DP"),
    CONSTRAINT "DETALLE_PROGRAMA_ASIGNATURA_FK"
FOREIGN KEY ("ID_ASIG") REFERENCES "ASIGNATURA" ("ID_ASIG"),
    CONSTRAINT "DETALLE_PROGRAMA_PROG_TIPO_FK"
FOREIGN KEY ("ID_PT") REFERENCES "PROGRAMA_TIPO" ("ID_PT"));

```

Código B.11: DDL Tabla detalle\_programa

```

CREATE TABLE "PLANIFICACION"
  (  "ID_PLAN" NUMBER(*,0) NOT NULL ENABLE,
    "TIPO_PLAN" VARCHAR2(10),
    "ULT_ESTADO" VARCHAR2(255),
    "FCREACION" DATE,
    "ULT_MODIF" DATE,
    "NUM_CORRELATIVO" NUMBER(*,0),
    "DV" CHAR(1),
    "ID_ASIG" NUMBER(*,0),
    CONSTRAINT "PLANIFICACION_PK" PRIMARY KEY ("ID_PLAN"),
    CONSTRAINT "PLANIFICACION_ASIGNATURA_FK" FOREIGN KEY ("
      ID_ASIG")
    REFERENCES "ASIGNATURA" ("ID_ASIG"),
    CONSTRAINT "PLANIFICACION_DOCENTE_FK" FOREIGN KEY ("
      NUM_CORRELATIVO", "DV")
    REFERENCES "DOCENTE" ("PERSONAL_NUM_CORRELATIVO", "
      PERSONAL_DV"));

```

Código B.12: DDL Tabla planificacion

```

CREATE TABLE "PLAN_ANUAL"
  (  "PLANIFICACION_ID_PLAN" NUMBER(*,0) NOT NULL ENABLE,
    CONSTRAINT "PLAN_ANUAL_PK" PRIMARY KEY ("
      PLANIFICACION_ID_PLAN"),
    CONSTRAINT "FK_ASS_11" FOREIGN KEY ("PLANIFICACION_ID_PLAN")
    REFERENCES "PLANIFICACION" ("ID_PLAN")
  );

```

Código B.13: DDL Tabla plan\_anual

```
CREATE TABLE "PLAN_UNIDAD"
("PLANIFICACION_ID_PLAN" NUMBER(*,0) NOT NULL ENABLE,
CONSTRAINT "PLAN_UNIDAD_PK" PRIMARY KEY ("
PLANIFICACION_ID_PLAN"),
CONSTRAINT "FK_ASS_12" FOREIGN KEY ("PLANIFICACION_ID_PLAN")
REFERENCES "PLANIFICACION" ("ID_PLAN"));
```

Código B.14: DDL Tabla plan\_unidad

```
CREATE TABLE "PLAN_CLASE"
("PLANIFICACION_ID_PLAN" NUMBER(*,0) NOT NULL ENABLE,
CONSTRAINT "PLAN_CLASE_PK" PRIMARY KEY ("
PLANIFICACION_ID_PLAN"),
CONSTRAINT "FK_ASS_13" FOREIGN KEY ("PLANIFICACION_ID_PLAN")
REFERENCES "PLANIFICACION" ("ID_PLAN"));
```

Código B.15: DDL Tabla plan\_clase

```
CREATE TABLE "ESTADO"
("ID_ESTADO" NUMBER(*,0) NOT NULL ENABLE,
"NOM_ESTADO" VARCHAR2(255),
CONSTRAINT "ESTADO_PK" PRIMARY KEY ("ID_ESTADO"));
```

Código B.16: DDL Tabla estado

```
CREATE TABLE "PLANIFICACION_ESTADO"
("ID_PLAN" NUMBER(*,0) NOT NULL ENABLE,
"ESTADO_ID_ESTADO" NUMBER(*,0) NOT NULL ENABLE,
CONSTRAINT "PLANIFICACION_ESTADO_PK" PRIMARY KEY ("ID_PLAN",
"ESTADO_ID_ESTADO")
CONSTRAINT "FK_ASS_19" FOREIGN KEY ("ESTADO_ID_ESTADO")
REFERENCES "ESTADO" ("ID_ESTADO"),
CONSTRAINT "PLANIFICACION_ESTADO_PLAN_FK1" FOREIGN KEY ("
ID_PLAN)
REFERENCES "PLANIFICACION" ("ID_PLAN"));
```

Código B.17: DDL Tabla planificacion\_estado

```

CREATE TABLE "PLANIFICACION_ESTADO_FCAMBIO"
(  "FCAMBIO" DATE NOT NULL ENABLE,
  "ID_ESTADO" NUMBER(*,0) NOT NULL ENABLE,
  "ID_PLAN" NUMBER(*,0) NOT NULL ENABLE,
  CONSTRAINT "PLAN_ESTADO_FC_PK"
PRIMARY KEY ("FCAMBIO","ID_ESTADO","ID_PLAN"),
  CONSTRAINT "PE_FCAMBIO_ESTADO_FK"
FOREIGN KEY ("ID_ESTADO")
REFERENCES "ESTADO" ("ID_ESTADO"),
  CONSTRAINT "PE_FCAMBIO_PLANIFICACION_FK"
FOREIGN KEY ("ID_PLAN") REFERENCES "PLANIFICACION" ("ID_PLAN"))
;

```

Código B.18: DDL Tabla planificacion\_estado\_fcambio

```

CREATE TABLE "EVALUACION"
(  "NUM_CORRELATIVO" NUMBER(*,0) NOT NULL ENABLE,
  "DV" CHAR(1 BYTE) NOT NULL ENABLE,
  "PLANIFICACION_ID_PLAN" NUMBER(*,0) NOT NULL ENABLE,
  CONSTRAINT "EVALUACION_PK"
PRIMARY KEY ("NUM_CORRELATIVO", "DV","PLANIFICACION_ID_PLAN"),
  CONSTRAINT "FK_ASS_43"
FOREIGN KEY ("PLANIFICACION_ID_PLAN")
REFERENCES "PLANIFICACION" ("ID_PLAN"),
  CONSTRAINT "EVALUACION_NO_DOCENTE_FK1"
FOREIGN KEY ("NUM_CORRELATIVO", "DV")
REFERENCES "NO_DOCENTE" ("PERSONAL_NUM_CORRELATIVO", "
  PERSONAL_DV"));

```

Código B.19: DDL Tabla evaluacion

```

CREATE TABLE "PERIODO"
(  "ID_PERIODO" NUMBER(*,0) NOT NULL ENABLE,
  "DURACION" NUMBER(*,0),
  "MES_INI" DATE,
  "MER_TER" DATE,
  "ID_PLAN" NUMBER(*,0),
  CONSTRAINT "PERIODO_PK" PRIMARY KEY ("ID_PERIODO"),
  CONSTRAINT "PERIODO_PLAN_ANUAL_FK" FOREIGN KEY ("ID_PLAN")
REFERENCES "PLAN_ANUAL" ("PLANIFICACION_ID_PLAN"));

```

Código B.20: DDL Tabla periodo



```

CREATE TABLE "EVALUACION_DETALLE"
  (  "FECHA_EVA" DATE NOT NULL ENABLE,
    "COMENTARIO" VARCHAR2(255 BYTE),
    "NUM_CORRELATIVO" NUMBER(*,0) NOT NULL ENABLE,
    "DV" CHAR(1 BYTE) NOT NULL ENABLE,
    "ID_PLAN" NUMBER(*,0) NOT NULL ENABLE,
    CONSTRAINT "EVALUACION_DETALLE_PK" PRIMARY KEY ("FECHA_EVA",
    "NUM_CORRELATIVO", "DV", "ID_PLAN"),
    CONSTRAINT "ED_NO_DOCENTE_FK" FOREIGN KEY ("NUM_CORRELATIVO", "DV")
    REFERENCES "NO_DOCENTE" ("PERSONAL_NUM_CORRELATIVO", "PERSONAL_DV"),
    CONSTRAINT "ED_PLANIFICACION_FK" FOREIGN KEY ("ID_PLAN")
    REFERENCES "PLANIFICACION" ("ID_PLAN"));

```

Código B.21: DDL Tabla evaluacion\_detalle

```

CREATE TABLE "UNIDAD"
  (  "ID_UNIDAD" NUMBER(*,0) NOT NULL ENABLE,
    "FECHA_I" DATE,
    "FECHA_T" DATE,
    "ID_PLAN" NUMBER(*,0),
    CONSTRAINT "UNIDAD_PK" PRIMARY KEY ("ID_UNIDAD"),
    CONSTRAINT "UNIDAD_PLAN_UNIDAD_FK" FOREIGN KEY ("ID_PLAN")
    REFERENCES "PLAN_UNIDAD" ("PLANIFICACION_ID_PLAN"));

```

Código B.22: DDL Tabla unidad

```

CREATE TABLE "CLASE"
  (  "ID_CLASE" NUMBER(*,0) NOT NULL ENABLE,
    "FECHA" DATE,
    "DURACION" NUMBER(*,0),
    "DIA" NUMBER(*,0),
    "HI" DATE,
    "HT" DATE,
    "ID_PLAN" NUMBER(*,0),
    CONSTRAINT "CLASE_PK" PRIMARY KEY ("ID_CLASE"),
    CONSTRAINT "CLASE_PLAN_CLASE_FK" FOREIGN KEY ("ID_PLAN")
    REFERENCES "PLAN_CLASE" ("PLANIFICACION_ID_PLAN"));

```

Código B.23: DDL Tabla clase

```
CREATE TABLE "PERIODO_APREVIOS"
  (  "APREVIOS" VARCHAR2(1000 BYTE) NOT NULL ENABLE,
    "ID_PERIODO" NUMBER(*,0) NOT NULL ENABLE,
    CONSTRAINT "PERIODO_APREVIOS_PK" PRIMARY KEY ("APREVIOS", "
      ID_PERIODO"),
    CONSTRAINT "PERIODO_APREVIOS_PERIODO_FK" FOREIGN KEY ("
      ID_PERIODO")
    REFERENCES "PERIODO" ("ID_PERIODO"));
```

Código B.24: DDL Tabla periodo\_aprevios

```
CREATE TABLE "UNIDAD_APREVIOS"
  (  "APREVIOS" VARCHAR2(1000 BYTE) NOT NULL ENABLE,
    "ID_UNIDAD" NUMBER(*,0) NOT NULL ENABLE,
    CONSTRAINT "UNIDAD_APREVIOS_PK" PRIMARY KEY ("ID_UNIDAD", "
      APREVIOS"),
    CONSTRAINT "UNIDAD_APREVIOS_UNIDAD_FK"
    FOREIGN KEY ("ID_UNIDAD") REFERENCES "UNIDAD" ("ID_UNIDAD"));
```

Código B.25: DDL Tabla unidad\_aprevios

```
CREATE TABLE "CLASE_APREVIOS"
  (  "APREVIOS" VARCHAR2(1000 BYTE) NOT NULL ENABLE,
    "ID_CLASE" NUMBER(*,0) NOT NULL ENABLE,
    CONSTRAINT "CLASE_APREVIOS_PK" PRIMARY KEY ("APREVIOS", "
      ID_CLASE"),
    CONSTRAINT "CLASE_APREVIOS_CLASE_FK" FOREIGN KEY ("
      ID_CLASE")
    REFERENCES "CLASE" ("ID_CLASE"));
```

Código B.26: DDL Tabla clase\_aprevios

```
CREATE TABLE "METODOLOGIA"
  (  "ID_MET" NUMBER(*,0) NOT NULL ENABLE,
    "NOM_MET" VARCHAR2(255 BYTE),
    "DESC_MET" VARCHAR2(500 BYTE),
    CONSTRAINT "METODOLOGIA_PK" PRIMARY KEY ("ID_MET"));
```

Código B.27: DDL Tabla metodologia

```
CREATE TABLE "METODOLOGIA_PERIODO"
  (  "PERIODO_ID_PERIODO" NUMBER(*,0) NOT NULL ENABLE,
    "METODOLOGIA_ID_MET" NUMBER(*,0) NOT NULL ENABLE,
    CONSTRAINT "METODOLOGIA_PERIODO__IDX"
    PRIMARY KEY ("PERIODO_ID_PERIODO","METODOLOGIA_ID_MET"),
    CONSTRAINT "FK_ASS_44"
    FOREIGN KEY ("PERIODO_ID_PERIODO")
    REFERENCES "PERIODO" ("ID_PERIODO"),
    CONSTRAINT "FK_ASS_45"
    FOREIGN KEY ("METODOLOGIA_ID_MET")
    REFERENCES "METODOLOGIA" ("ID_MET"));
```

Código B.28: DDL Tabla metodologia\_periodo

```
CREATE TABLE "METODOLOGIA_UNIDAD"
  (  "UNIDAD_ID_UNIDAD" NUMBER(*,0) NOT NULL ENABLE,
    "METODOLOGIA_ID_MET" NUMBER(*,0) NOT NULL ENABLE,
    CONSTRAINT "METODOLOGIA_UNIDAD__IDX"
    PRIMARY KEY ("UNIDAD_ID_UNIDAD",
    "METODOLOGIA_ID_MET"),
    CONSTRAINT "FK_ASS_46"
    FOREIGN KEY ("UNIDAD_ID_UNIDAD")
    REFERENCES "UNIDAD" ("ID_UNIDAD"),
    CONSTRAINT "FK_ASS_47"
    FOREIGN KEY ("METODOLOGIA_ID_MET")
    REFERENCES "METODOLOGIA" ("ID_MET"));
```

Código B.29: DDL Tabla metodologia\_unidad

```
CREATE TABLE "RECURSO"
  (  "ID_RECURSO" NUMBER(*,0) NOT NULL ENABLE,
    "NOM_REC" VARCHAR2(255 BYTE),
    "DESC_REC" VARCHAR2(500 BYTE),
    CONSTRAINT "RECURSO_PK"
    PRIMARY KEY ("ID_RECURSO"));
```

Código B.30: DDL Tabla recurso

```

CREATE TABLE "ACTIVIDAD"
  (  "ID_ACT" NUMBER(*,0) NOT NULL ENABLE,
    "DESC_ACT" VARCHAR2(500 BYTE),
    "DURACION" NUMBER(*,0),
    "ID_CLASE" NUMBER(*,0),
    "ID_MET" NUMBER(*,0),
    CONSTRAINT "ACTIVIDAD_PK" PRIMARY KEY ("ID_ACT"),
    CONSTRAINT "ACTIVIDAD_CLASE_FK" FOREIGN KEY ("ID_CLASE")
    REFERENCES "CLASE" ("ID_CLASE"),
    CONSTRAINT "ACTIVIDAD_METODOLOGIA_FK" FOREIGN KEY ("ID_MET"
    )
    REFERENCES "METODOLOGIA" ("ID_MET"));

```

Código B.31: DDL Tabla actividad

```

CREATE TABLE "RECURSO_ACTIVIDAD"
  (  "RECURSO_ID_RECURSO" NUMBER(*,0) NOT NULL ENABLE,
    "ACTIVIDAD_ID_ACT" NUMBER(*,0) NOT NULL ENABLE,
    CONSTRAINT "RECURSO_ACTIVIDAD__IDX" PRIMARY KEY ("
    RECURSO_ID_RECURSO",
    "ACTIVIDAD_ID_ACT"),
    CONSTRAINT "FK_ASS_50" FOREIGN KEY ("RECURSO_ID_RECURSO")
    REFERENCES "RECURSO" ("ID_RECURSO"),
    CONSTRAINT "FK_ASS_51" FOREIGN KEY ("ACTIVIDAD_ID_ACT")
    REFERENCES "ACTIVIDAD" ("ID_ACT"));

```

Código B.32: DDL Tabla recurso\_actividad

```

CREATE TABLE "RECURSO_PERIODO"
  (  "PERIODO_ID_PERIODO" NUMBER(*,0) NOT NULL ENABLE,
    "RECURSO_ID_RECURSO" NUMBER(*,0) NOT NULL ENABLE,
    CONSTRAINT "RECURSO_PERIODO__IDX" PRIMARY KEY ("
    PERIODO_ID_PERIODO",
    "RECURSO_ID_RECURSO"),
    CONSTRAINT "FK_ASS_52" FOREIGN KEY ("PERIODO_ID_PERIODO")
    REFERENCES "PERIODO" ("ID_PERIODO"),
    CONSTRAINT "FK_ASS_53" FOREIGN KEY ("RECURSO_ID_RECURSO")
    REFERENCES "RECURSO" ("ID_RECURSO"));

```

Código B.33: DDL Tabla recurso\_periodo

```
CREATE TABLE "RECURSO_UNIDAD"
  (  "UNIDAD_ID_UNIDAD" NUMBER(*,0) NOT NULL ENABLE,
    "RECURSO_ID_RECURSO" NUMBER(*,0) NOT NULL ENABLE,
    CONSTRAINT "RECURSO_UNIDAD__IDX" PRIMARY KEY ("
      UNIDAD_ID_UNIDAD",
      "RECURSO_ID_RECURSO"),
    CONSTRAINT "FK_ASS_54" FOREIGN KEY ("UNIDAD_ID_UNIDAD")
    REFERENCES "UNIDAD" ("ID_UNIDAD"),
    CONSTRAINT "FK_ASS_55" FOREIGN KEY ("RECURSO_ID_RECURSO")
    REFERENCES "RECURSO" ("ID_RECURSO"));
```

Código B.34: DDL Tabla recurso\_unidad

```
CREATE TABLE "DETALLE_PROGRAMA_CLASE"
  (  "DETALLE_PROGRAMA_ID_DP" NUMBER(*,0) NOT NULL ENABLE,
    "CLASE_ID_CLASE" NUMBER(*,0) NOT NULL ENABLE,
    CONSTRAINT "DETALLE_PROGRAMA_CLASE__IDX" PRIMARY KEY
    ("DETALLE_PROGRAMA_ID_DP", "CLASE_ID_CLASE"),
    CONSTRAINT "FK_ASS_37" FOREIGN KEY ("CLASE_ID_CLASE")
    REFERENCES "CLASE" ("ID_CLASE"),
    CONSTRAINT "DETALLE_PROGRAMA_CLASE_DE_FK1" FOREIGN KEY ("
      DETALLE_PROGRAMA_ID_DP")
    REFERENCES "DETALLE_PROGRAMA" ("ID_DP"));
```

Código B.35: DDL Tabla detalle\_programa\_clase

```
CREATE TABLE "DETALLE_PROGRAMA_PERIODO"
  (  "DETALLE_PROGRAMA_ID_DP" NUMBER(*,0) NOT NULL ENABLE,
    "PERIODO_ID_PERIODO" NUMBER(*,0) NOT NULL ENABLE,
    CONSTRAINT "DETALLE_PROGRAMA_PERIODO__IDX" PRIMARY KEY
    ("DETALLE_PROGRAMA_ID_DP", "PERIODO_ID_PERIODO"),
    CONSTRAINT "FK_ASS_39" FOREIGN KEY ("PERIODO_ID_PERIODO")
    REFERENCES "PERIODO" ("ID_PERIODO"),
    CONSTRAINT "DETALLE_PROGRAMA_PERIODO__FK1" FOREIGN KEY ("
      DETALLE_PROGRAMA_ID_DP")
    REFERENCES "DETALLE_PROGRAMA" ("ID_DP"));
```

Código B.36: DDL Tabla detalle\_programa\_periodes

```
CREATE TABLE "DETALLE_PROGRAMA_UNIDAD"
  (  "DETALLE_PROGRAMA_ID_DP" NUMBER(*,0) NOT NULL ENABLE,
    "UNIDAD_ID_UNIDAD" NUMBER(*,0) NOT NULL ENABLE,
    CONSTRAINT "DETALLE_PROGRAMA_UNIDAD__IDX" PRIMARY KEY
    ("DETALLE_PROGRAMA_ID_DP", "UNIDAD_ID_UNIDAD"),
    CONSTRAINT "FK_ASS_41" FOREIGN KEY ("UNIDAD_ID_UNIDAD")
    REFERENCES "UNIDAD" ("ID_UNIDAD"),
    CONSTRAINT "DETALLE_PROGRAMA_UNIDAD_D_FK1" FOREIGN KEY ("
      DETALLE_PROGRAMA_ID_DP")
    REFERENCES "DETALLE_PROGRAMA" ("ID_DP"));
```

Código B.37: DDL Tabla detalle\_programa\_unidad

## **B.4. Mapeo de caso de estudio de entidad relacionamiento / UML a la capa objeto relacional**

El modelo objeto relacional fue implementado en otro tablespace del servidor Oracle del DISC, esto con el propósito de evitar confusiones y futuros problemas. El nombre del tablespace es IMUNOZ2.

```
CREATE OR REPLACE TYPE NIVEL_OB AS OBJECT (
  ID_NIVEL NUMBER(10), NOM_NIVEL VARCHAR2(255));
```

Código B.38: DDL Tipo Nivel

```
CREATE OR REPLACE TYPE CURSO_OB AS OBJECT (
  REFNIVEL REF NIVEL_OB,
  ID_NIVEL NUMBER(10),
  LETRA CHAR(1), YEAR_ DATE);
```

Código B.39: DDL Tipo Curso

```
CREATE OR REPLACE TYPE CELULAR_PERSONA_OB AS OBJECT(
  NUMERO_CELULAR NUMBER(10));

CREATE OR REPLACE TYPE CELULARES AS VARRAY(5) OF
  CELULAR_PERSONA_OB;
```

Código B.40: DDL Tipo celular personal y varray con su listado

```
CREATE OR REPLACE TYPE EMAIL_PERSONA_OB AS OBJECT (
CORREO VARCHAR2(500));
```

```
CREATE OR REPLACE TYPE MAILS AS VARRAY(5) OF
EMAIL_PERSONA_OB;
```

Código B.41: DDL Tipo email personal y varray de con listado

```
CREATE OR REPLACE TYPE RUT_PERSONA_OB AS OBJECT
( NUM_CORRELATIVO NUMBER(8),
DV CHAR(1));
```

Código B.42: DDL Tipo Rut personal

```
CREATE OR REPLACE TYPE NOMBRE_COMPLETO_PERSONA_OB AS
OBJECT (
NOMBRES VARCHAR2(255),
APATERNO VARCHAR2(255),
AMATERNO VARCHAR2(255));
```

Código B.43: DDL Tipo Nombre completo del personal

```
CREATE OR REPLACE TYPE DIRECCION_PERSONA_OB AS OBJECT
( CALLE VARCHAR2(500),
NUM VARCHAR2(10),
COD_POSTAL VARCHAR2(10),
CIUDAD VARCHAR2(255),
REGION VARCHAR2(255));
```

Código B.44: DDL Tipo Dirección del personal

```
CREATE OR REPLACE TYPE CARGO_OB AS OBJECT(
ID_CARGO NUMBER(10),
NOM_C VARCHAR2(255),
DESC_C VARCHAR2(255));
```

Código B.45: DDL Tipo Cargo

```

CREATE OR REPLACE TYPE PERSONAL_OB AS OBJECT(
  RUT RUT_PERSONA_OB,
  NOMBRE_COMPLETO NOMBRE_COMPLETO_PERSONA_OB,
  FNAC DATE,
  DIRECCION DIRECCION_PERSONA_OB,
  NUM_CELULARES CELULARES,
  CORREOS MAILS,
  MEMBER FUNCTION EDAD RETURN NUMBER)
NOT FINAL;

CREATE OR REPLACE TYPE BODY PERSONAL_OB IS
  MEMBER FUNCTION EDAD RETURN NUMBER IS
  BEGIN
    RETURN(TRUNC(MONTHS_BETWEEN(SYSDATE, SELF.FNAC) / 12));
  END;
END;

```

Código B.46: DDL Tipo Personal

```

CREATE OR REPLACE TYPE DOCENTE_OB UNDER PERSONAL_OB(
  CURSO_JEFE REF CURSO_OB);

CREATE OR REPLACE TYPE NO_DOCENTE_OB UNDER PERSONAL_OB(
  CARGO REF CARGO_OB);

```

Código B.47: DDL Tipo Docente y No Docente

```

ALTER TYPE CURSO_OB ADD ATTRIBUTE
  (PROF_JEFE REF DOCENTE_OB) CASCADE;

```

Código B.48: Agregando el profesor jefe al curso

```

CREATE OR REPLACE TYPE ASIGNATURA_OB AS OBJECT(
  ID_ASIG NUMBER(10),
  NOM_ASIG VARCHAR2(255),
  CURSO_DICTA REF CURSO_OB,
  PROFESOR_DICTA REF DOCENTE_OB);

```

Código B.49: DDL Tipo Asignatura



```
CREATE OR REPLACE TYPE PROGRAMA_TIPO_OB AS OBJECT(
  ID_PT NUMBER(10),
  NOM_PT VARCHAR2(255),
  ABREV VARCHAR2(10));
```

Código B.50: DDL Tipo Programa tipo

```
CREATE OR REPLACE TYPE DETALLE_PROGRAMA_OB AS OBJECT(
  ID_DP NUMBER(10), NOM_DP VARCHAR2(255),
  DESC_DP VARCHAR2(500),
  ASIGNATURA REF ASIGNATURA_OB,
  PROGRAMA_TIPO REF PROGRAMA_TIPO_OB);
```

Código B.51: DDL Tipo Detalle Programa

```
CREATE OR REPLACE TYPE PLANIFICACION_OB AS OBJECT(
  ID_PLAN NUMBER(10), ULT_MODIF DATE,
  ULT_ESTADO VARCHAR2(30),
  FCREACION DATE,
  TIPO_PLAN VARCHAR2(10),
  PROF_CREA REF DOCENTE_OB,
  ASIGNATURA REF ASIGNATURA_OB) NOT FINAL;
```

Código B.52: DDL Tipo Planificación

```
CREATE OR REPLACE TYPE PLAN_ANUAL_OB UNDER
  PLANIFICACION_OB() NOT FINAL;
```

```
CREATE OR REPLACE TYPE PLAN_UNIDAD_OB UNDER
  PLANIFICACION_OB() NOT FINAL;
```

```
CREATE OR REPLACE TYPE PLAN_CLASE_OB UNDER
  PLANIFICACION_OB() NOT FINAL;
```

Código B.53: DDL Tipo Planificación anual, unidad y clase

```
CREATE OR REPLACE TYPE APREVIOS_OB AS OBJECT (
  APRENDIZAJE_PREV VARCHAR2(1000));
```

```
CREATE OR REPLACE TYPE LISTADO_APREVIOS AS VARRAY(10) OF REF
  APREVIOS_OB;
```

Código B.54: DDL Tipo aprevios y varray con su listado

```
CREATE OR REPLACE TYPE PERIODO_OB AS OBJECT(
  ID_PERIODO NUMBER(10),
  DURACION NUMBER(3),
  MES_INI DATE,
  MES_TER DATE,
  PLAN_ANUAL_PERTENECE REF PLAN_ANUAL_OB,
  APREVIOS LISTADO_APREVIOS);
```

Código B.55: DDL Tipo periodo

```
CREATE OR REPLACE TYPE UNIDAD_OB AS OBJECT(
  ID_UNIDAD NUMBER(10),
  FECHA1 DATE,
  FECHA2 DATE,
  PLAN_UNIDAD_PERTENECE REF PLAN_UNIDAD_OB,
  APREVIOS LISTADO_APREVIOS);
```

Código B.56: DDL Tipo unidad

```
CREATE OR REPLACE TYPE CLASE_OB AS OBJECT(
  ID_CLASE NUMBER(10),
  HORARIO HORARIO_CLASE_OB,
  FECHA DATE,
  PLAN_CLASE_PERTENECE REF PLAN_CLASE_OB,
  APREVIOS LISTADO_APREVIOS);
```

Código B.57: DDL Tipo clase

```
CREATE OR REPLACE TYPE RECURSO_OB AS OBJECT(
  ID_REC NUMBER(10),
  NOM_REC VARCHAR2(100),
  DESC_REC VARCHAR2(500));
```

Código B.58: DDL Tipo recurso

```
CREATE OR REPLACE TYPE ACTIVIDAD_OB AS OBJECT(
  ID_ACT NUMBER(10),
  DESC_ACT VARCHAR2(500),
  DURACION NUMBER(3),
  CLASE_HECHA REF CLASE_OB);
```

Código B.59: DDL Tipo actividad

```
CREATE OR REPLACE TYPE DETALLE_PROGRAMA_PERIODO_OB AS
  OBJECT( DETALLE_PROGRAMA REF DETALLE_PROGRAMA_OB,
    PERIODO REF PERIODO_OB);
```

Código B.60: DDL Tipo detalle programa periodo - tipo intersección para relacionamiento N : N

```
CREATE OR REPLACE TYPE DETALLE_PROGRAMA_UNIDAD_OB AS
  OBJECT( DETALLE_PROGRAMA REF DETALLE_PROGRAMA_OB,
    UNIDAD REF UNIDAD_OB);
```

Código B.61: DDL Tipo detalle programa unidad - tipo intersección para relacionamiento N : N

```
CREATE OR REPLACE TYPE DETALLE_PROGRAMA_CLASE_OB AS
  OBJECT(DETALLE_PROGRAMA REF DETALLE_PROGRAMA_OB,
    CLASE REF CLASE_OB);
```

Código B.62: DDL Tipo detalle programa clase - tipo intersección para relacionamiento N : N

```
CREATE OR REPLACE TYPE RECURSO_ACTIVIDAD_OB AS
  OBJECT(RECURSO REF RECURSO_OB, ACTIVIDAD REF ACTIVIDAD_OB)
  ;
```

Código B.63: DDL Tipo recurso actividad tipo intersección para relacionamiento N : N

```
CREATE OR REPLACE TYPE RECURSO_PERIODO_OB AS
  OBJECT(PERIODO REF PERIODO_OB, RECURSO REF RECURSO_OB);
```

Código B.64: DDL Tipo recurso periodo - tipo intersección para relacionamiento N : N

```
CREATE OR REPLACE TYPE RECURSO_UNIDAD_OB AS
  OBJECT(UNIDAD REF UNIDAD_OB,
    RECURSO REF RECURSO_OB);
```

Código B.65: DDL Tipo recurso unidad - tipo intersección para relacionamiento N : N

```
CREATE OR REPLACE TYPE METODOLOGIA_PERIODO_OB AS
  OBJECT(PERIODO REF PERIODO_OB,
    METODOLOGIA REF METODOLOGIA_OB);
```

Código B.66: DDL Tipo metodología periodo - tipo intersección para relacionamiento N : N

```
CREATE OR REPLACE TYPE METODOLOGIA_UNIDAD_OB AS OBJECT(
  UNIDAD REF UNIDAD_OB, METODOLOGIA REF METODOLOGIA_OB);
```

Código B.67: DDL Tipo metodología unidad - tipo intersección para relacionamiento N : N

```
CREATE OR REPLACE TYPE ACTIVIDADES_CLASE_OB AS VARRAY(5) OF
  REF ACTIVIDAD_OB;
```

```
ALTER TYPE CLASE_OB ADD ATTRIBUTE (ACTIVIDADES
  ACTIVIDADES_CLASE_OB) CASCADE;
```

```
ALTER TYPE METODOLOGIA_OB ADD ATTRIBUTE (ACTIVIDADES
  ACTIVIDADES_CLASE_OB) CASCADE;
```

Código B.68: DDL Tipo varray actividades y agrando el listado a la clase y metodología

```
CREATE OR REPLACE TYPE DETALLES_PERIODOS AS VARRAY(15)
  OF REF DETALLE_PROGRAMA_PERIODO_OB;
```

```
CREATE OR REPLACE TYPE DETALLES_UNIDADES AS VARRAY(15) OF
  REF DETALLE_PROGRAMA_UNIDAD_OB;
```

```
CREATE OR REPLACE TYPE DETALLES_CLASES AS VARRAY(15) OF REF
  DETALLE_PROGRAMA_CLASE_OB;
```

```
CREATE OR REPLACE TYPE RECURSOS_ACTIVIDADES AS VARRAY(15)
  OF REF RECURSO_ACTIVIDAD_OB;
```

```
CREATE OR REPLACE TYPE RECURSOS_PERIODOS AS VARRAY(15) OF
  REF RECURSO_PERIODO_OB;
```

```
CREATE OR REPLACE TYPE RECURSOS_UNIDADES AS VARRAY(15) OF
  REF RECURSO_UNIDAD_OB;
```

```
CREATE OR REPLACE TYPE METODOLOGIAS_PERIODOS AS VARRAY(15)
  OF REF METODOLOGIA_PERIODO_OB;

CREATE OR REPLACE TYPE METODOLOGIAS_UNIDADES AS VARRAY(15)
  OF REF METODOLOGIA_UNIDAD_OB;
```

Código B.69: DDL tipo varray de los tipo intersección

```
ALTER TYPE DETALLE_PROGRAMA_OB ADD ATTRIBUTE (PERIODOS
  DETALLES_PERIODOS)CASCADE;

ALTER TYPE DETALLE_PROGRAMA_OB ADD ATTRIBUTE (UNIDADES
  DETALLES_UNIDADES)CASCADE;

ALTER TYPE DETALLE_PROGRAMA_OB ADD ATTRIBUTE (CLASES
  DETALLES_CLASES)CASCADE;

ALTER TYPE PERIODO_OB ADD ATTRIBUTE (DETALLES
  DETALLES_PERIODOS)CASCADE;

ALTER TYPE UNIDAD_OB ADD ATTRIBUTE (DETALLES
  DETALLES_UNIDADES)CASCADE;

ALTER TYPE CLASE_OB ADD ATTRIBUTE (DETALLES DETALLES_CLASES)
CASCADE;

ALTER TYPE PERIODO_OB ADD ATTRIBUTE (RECURSOS
  RECURSOS_PERIODOS)CASCADE;

ALTER TYPE RECURSO_OB ADD ATTRIBUTE (PERIODOS
  RECURSOS_PERIODOS)CASCADE;

ALTER TYPE UNIDAD_OB ADD ATTRIBUTE (RECURSOS
  RECURSOS_UNIDADES)CASCADE;

ALTER TYPE RECURSO_OB ADD ATTRIBUTE (UNIDADES
  RECURSOS_UNIDADES)CASCADE;

ALTER TYPE RECURSO_OB ADD ATTRIBUTE (ACTIVIDADES
  RECURSOS_ACTIVIDADES)CASCADE;
```

```
ALTER TYPE ACTIVIDAD_OB ADD ATTRIBUTE (RECURSOS
RECURSOS_ACTIVIDADES)CASCADE;
```

Código B.70: Agregando listado de tipos intersección en los objetos involucrados

```
CREATE OR REPLACE TYPE LISTADO_PERIODOS AS VARRAY(50) OF REF
PERIODO_OB;
```

```
CREATE OR REPLACE TYPE LISTADO_UNIDADES AS VARRAY(50) OF REF
UNIDAD_OB;
```

```
CREATE OR REPLACE TYPE LISTADO_CLASES AS VARRAY(50) OF REF
CLASE_OB;
```

Código B.71: DDL tipo varray listado periodos, unidades y clases

```
ALTER TYPE PLAN_ANUAL_OB ADD ATTRIBUTE (PERIODOS
LISTADO_PERIODOS) CASCADE;
```

```
ALTER TYPE PLAN_UNIDAD_OB ADD ATTRIBUTE (UNIDADES
LISTADO_UNIDADES) CASCADE;
```

```
ALTER TYPE PLAN_CLASE_OB ADD ATTRIBUTE (CLASES
LISTADO_CLASES) CASCADE;
```

Código B.72: Agregando los listados de periodos, unidades y clases

```
CREATE OR REPLACE TYPE ESTADO_OB AS OBJECT (
ID_ESTADO NUMBER(10), NOM_ESTADO VARCHAR(255));
```

Código B.73: DDL tipo estado

```
CREATE OR REPLACE TYPE
FCAMBIO_ESTADO_PLAN AS OBJECT( FECHA_CAMBIO DATE);
```

```
CREATE OR REPLACE TYPE LISTADO_FCAMBIO_ESTADO AS VARRAY(20)
OF REF FCAMBIO_ESTADO_PLAN;
```

Código B.74: DDL tipo fecha cambio estado planificación y varray con su listado

```
CREATE OR REPLACE TYPE EVALUACION_DETALLE_OB AS OBJECT(
    FECHA_EVA DATE,
    COMENTARIO VARCHAR2(500));

CREATE OR REPLACE TYPE LISTADO_EVALUACION_DETALLE AS VARRAY
(20) OF REF EVALUACION_DETALLE_OB;
```

Código B.75: DDL tipo evaluación detalles y varray con su listado

```
CREATE OR REPLACE TYPE PLANIFICACION_ESTADO_OB AS OBJECT (
    PLANIFICACION REF PLANIFICACION_OB, ESTADO REF ESTADO_OB,
    LISTADO_FECHA_CAMBIOS LISTADO_FCAMBIO_ESTADO);

CREATE OR REPLACE TYPE LISTADO_PLAN_ESTADOS AS VARRAY(50) OF
    REF PLANIFICACION_ESTADO_OB;
```

Código B.76: DDL Tipo no docente planificacion - tipo intersección para relacionamiento N : N y varray con su listado

```
ALTER TYPE NO_DOCENTE_OB ADD
    ATTRIBUTE (PLANIFICACIONES LISTADO_EVALUACIONES) CASCADE;

ALTER TYPE PLANIFICACION_OB ADD
    ATTRIBUTE (EVALUADORES LISTADO_EVALUACIONES) CASCADE;

ALTER TYPE PLANIFICACION_OB ADD
    ATTRIBUTE (ESTADOS LISTADO_PLAN_ESTADOS) CASCADE;

ALTER TYPE ESTADO_OB ADD
    ATTRIBUTE (PLANIFICACIONES LISTADO_PLAN_ESTADOS) CASCADE;
```

Código B.77: Agregando listado de tipos intersección en los objetos involucrados

```
CREATE OR REPLACE TYPE LISTADO_NO_DOCENTE AS
    VARRAY(10) OF REF NO_DOCENTE_OB;

ALTER TYPE CARGO_OB ADD ATTRIBUTE (NO_DOCENTES
    LISTADO_NO_DOCENTE) CASCADE;
```

Código B.78: DDL tipo varray de no docentes y agregando el listado en cargo

```
CREATE OR REPLACE TYPE LISTADO_PLANIFICACIONES AS VARRAY
(100) OF REF PLANIFICACION_OB;
```

```
ALTER TYPE DOCENTE_OB ADD ATTRIBUTE (PLANIFICACIONES
LISTADO_PLAN_ESTADOS) CASCADE;
```

```
ALTER TYPE ASIGNATURA_OB ADD ATTRIBUTE (PLANIFICACIONES
LISTADO_PLANIFICACIONES) CASCADE;
```

Código B.79: DDL tipo varray de planificaciones y agregando el listado en docente y asignatura

```
CREATE OR REPLACE TYPE LISTADO_DETALLE_PROG AS
VARRAY(50) OF REF DETALLE_PROGRAMA_OB;
```

```
ALTER TYPE ASIGNATURA_OB ADD ATTRIBUTE (DETALLES
LISTADO_DETALLE_PROG) CASCADE;
```

Código B.80: DDL tipo varray con listado de detalle programa y agregando el listado en asignatura

```
CREATE OR REPLACE TYPE LISTADO_ASIGNATURAS AS
VARRAY(100) OF REF ASIGNATURA_OB;
```

```
ALTER TYPE CURSO_OB ADD ATTRIBUTE (ASIGNATURAS
LISTADO_ASIGNATURAS) CASCADE;
```

```
ALTER TYPE DOCENTE_OB ADD ATTRIBUTE (ASIGNATURAS
LISTADO_ASIGNATURAS) CASCADE;
```

Código B.81: DDL tipo varray con listado de asignaturas y agregando el listado en curso y docente

```
CREATE OR REPLACE TYPE LISTADO_CURSOS AS
VARRAY(100) OF REF CURSO_OB;
```

```
ALTER TYPE NIVEL_OB ADD ATTRIBUTE (CURSOS LISTADO_CURSOS)
CASCADE;
```

Código B.82: DDL tipo varray con listado de cursos y agregando el listado en nivel



## B.5. Mapeo de caso de estudio de la capa objeto relacional a la capa de persistencia

```
CREATE TABLE NIVEL OF NIVEL_OB(  
PRIMARY KEY(ID_NIVEL));  
  
CREATE TABLE CURSO OF CURSO_OB (  
ID_NIVEL SCOPE IS REFNIVEL,  
PRIMARY KEY(ID_NIVEL,LETRA));  
  
CREATE TABLE ASIGNATURA OF ASIGNATURA_OB(  
PRIMARY KEY(ID_ASIG));  
  
CREATE TABLE PERSONAL OF PERSONAL_OB(  
PRIMARY KEY(RUT.NUM_CORRELATIVO,RUT.DV));  
  
CREATE TABLE DOCENTE OF DOCENTE_OB(  
PRIMARY KEY(RUT.NUM_CORRELATIVO,RUT.DV));  
  
CREATE TABLE CARGO OF CARGO_OB(  
PRIMARY KEY(ID_CARGO));  
  
CREATE TABLE NO_DOCENTE OF DOCENTE_OB(  
CARGO SCOPE IS CARGO,  
PRIMARY KEY(RUT.NUM_CORRELATIVO,RUT.DV));  
  
CREATE TABLE ESTADO OF ESTADO_OB (  
PRIMARY KEY(ID_ESTADO));  
  
CREATE TABLE PLANIFICACION OF PLANIFICACION_OB(  
PRIMARY KEY(ID_PLAN));  
  
CREATE TABLE PLAN_ANUAL OF PLAN_ANUAL_OB(  
PRIMARY KEY(ID_PLAN));  
  
CREATE TABLE PLAN_UNIDAD OF PLAN_UNIDAD_OB(  
PRIMARY KEY(ID_PLAN));
```

Código B.83: DDL tabla nivel, curso, asignatura, personal, docente, cargo, no\_docente, estado, planificacion, plan\_anual y plan\_unidad

```

CREATE TABLE PLAN_CLASE OF PLAN_CLASE_OB(
PRIMARY KEY(ID_PLAN));

CREATE TABLE PERIODO OF PERIODO_OB(
PLAN_ANUAL_PERTENECE SCOPE IS PLAN_ANUAL,
PRIMARY KEY(ID_PERIODO));

CREATE TABLE UNIDAD OF UNIDAD_OB(
PLAN_UNIDAD_PERTENECE SCOPE IS PLAN_UNIDAD,
PRIMARY KEY(ID_UNIDAD));

CREATE TABLE CLASE OF CLASE_OB(
PLAN_CLASE_PERTENECE SCOPE IS PLAN_CLASE,
PRIMARY KEY(ID_CLASE));

```

Código B.84: DDL tabla plan\_clase, periodo, unidad y clase

```

CREATE TABLE ACTIVIDAD OF ACTIVIDAD_OB (
PRIMARY KEY(ID_ASIG));

CREATE TABLE RECURSO OF RECURSO_OB(
PRIMARY KEY(ID_REC));

CREATE TABLE METODOLOGIA OF METODOLOGIA_OB(
PRIMARY KEY(ID_MET));

CREATE TABLE PROGRAMA_TIPO OF PROGRAMA_TIPO_OB(
PRIMARY KEY(ID_PT));

CREATE TABLE DETALLE_PROGRAMA OF DETALLE_PROGRAMA_OB(
PROGRAMA_TIPO SCOPE IS PROGRAMA_TIPO,
PRIMARY KEY(ID_DP));

```

Código B.85: DDL tabla actividad, recurso, metodologia, programa\_tipo y detalle\_programa

## B.6. Detalles de implementación de nuevo requerimiento utilizando XML

### B.6.1. Registro de esquema en servidor

```
DBMS_XMLSCHEMA.REGISTERSCHEMA(
  schemaurl => 'http://webserver.disc.ucn.cl/~imuno/schemav.xsd',
  schemadoc => '<xs:schema attributeFormDefault="unqualified"
    elementFormDefault="qualified"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="curriculum">
    <xs:complexType>
    <xs:sequence>
    <xs:element name="estudios">
    <xs:complexType><xs:sequence>
    <xs:element name="estudio" maxOccurs="unbounded"
      minOccurs="0">
    <xs:complexType><xs:sequence>
    <xs:element type="xs:string" name="
      nombre_establecimiento"/>
    <xs:element type="xs:string" name="carrera"
      minOccurs="0"/>
    <xs:element type="xs:string" name="ciudad"/>
    <xs:element type="xs:short" name="fecha_egreso"
      />
    </xs:sequence>
    <xs:attribute type="xs:string" name="nivel" use="
      optional"/>
    </xs:complexType> </xs:element>
    </xs:sequence></xs:complexType></xs:element>
    <xs:element name="cursos">
    <xs:complexType><xs:sequence>
    <xs:element name="curso" maxOccurs="unbounded"
      minOccurs="0">
    <xs:complexType><xs:sequence>
    <xs:element type="xs:string" name="tipo"/>
    <xs:element type="xs:string" name="nombre"/>
    <xs:element type="xs:byte" name="horas"/>
    </xs:sequence>
    <xs:attribute type="xs:string" name="area" use="
      optional"/>
    </xs:complexType></xs:element>
    </xs:sequence></xs:complexType></xs:element>
  </xs:schema>'
```

```

        </xs:complexType></xs:element>
    </xs:sequence></xs:complexType></xs:element>
<xs:element name="idiomas">
    <xs:complexType><xs:sequence>
        <xs:element name="idioma" maxOccurs="unbounded"
            minOccurs="0">
            <xs:complexType><xs:sequence>
                <xs:element type="xs:string" name="nivel"/>
            </xs:sequence>
            <xs:attribute type="xs:string" name="nombre" use="
                optional"/>
            </xs:complexType></xs:element></xs:sequence>
        </xs:complexType> </xs:element>
    </xs:sequence></xs:complexType></xs:element>
</xs:schema>', local => FALSE, gentypes => FALSE,
genbean => FALSE, gentables => FALSE,
force => FALSE, owner => 'IMUNOZ2',
OPTIONS => DBMS_XMLSCHEMA.REGISTER_BINARYXML);
END;

```

Código B.86: Registro esquema XML en servidor

### B.6.2. Verificación de validez de documentos XML

Antes de realizar una inserción o actualización en la tabla que contiene la columna XML, se debe verificar si el documento XML a insertar o actualizar es válido con respecto al esquema registrado en el servidor. En el código B.87 se crea un trigger que se encarga de verificar si los datos insertados o actualizados en la columna xmltype son validos según el esquema registrado.

```

CREATE TRIGGER trigger_xml BEFORE INSERT OR UPDATE ON
    personalxml FOR EACH ROW
DECLARE
    newxml XMLType;
BEGIN
    newxml := :new.curriculum;
    XMLTYPE.schemavalidate(newxml);
END; /

```

Código B.87: Trigger que verifica si el documento xml es válido al insertar y actualizar

## Anexo C

# Desempeño de consultas SQL relacionales y objeto-relacionales en Oracle

Se presenta el comportamiento del modelo relacional y objeto relacional en consultas específicas, y se muestran así los resultados de medir y a la vez se comparar la eficiencia (tiempo, utilización de recursos del sistema) de operaciones que involucraron cláusulas GROUP BY, subconsultas (con las cláusulas IN y EXISTS) y joins.

El desempeño de cada consulta varía según el set de datos y el tipo de consulta, en algunos casos el modelo relacional es superior al modelo objeto relacional y viceversa. Algunas de las tasas utilizadas en [13] para medir el desempeño de una consulta son:

- Bloques leídos ( $f+g$ ) sobre filas procesadas ( $h$ ). Esta tasa indica de una manera general el costo relativo de la consulta. Mientras más bloques tienen que ser accedidos en relación en las filas retornadas, la fila traída será mucho más costosa. Una relación similar se puede deducir sobre la tasa de bloques leídos sobre ejecuciones  $(f+g)/e$ . El valor que se debe procurar para esta tasa debe ser menor a 10, sin embargo tasas con valores de 10 a 20 son aceptables. Tasas por encima de 20 pueden indicar alguna posibilidad de optimización en este campo.
- Lecturas de Disco ( $k$ ) sobre lecturas lógicas ( $f+g$ ). Esta es una tasa de error (miss rate) dentro del buffer de datos en la zona de caché, es decir, es una tasa que muestra el porcentaje de ocasiones en el que SGBD no ha encontrado las filas solicitadas en el buffer de la zona de caché y por lo

tanto ha tenido que recurrir a traer los bloques desde disco. Generalmente se busca que esta tasa no represente más de un 10 %.

Las pruebas se realizaron con tablas con muchos y pocos datos para ambos modelos, en consultas del tipo group by sencillo, se puede observar en la figura C.1, que el modelo objeto relacional tiene un rendimiento inferior con respecto al modelo relacional, tanto con muchos como con pocos datos (aunque a medida que aumentan los datos, el rendimiento del modelo objeto relacional mejora). En la figura C.2, se puede visualizar que el modelo objeto relacional es superior al modelo relacional en la tasa de lecturas de disco a lecturas lógicas, cuando se efectúan sobre muchos datos, sin embargo, al disminuir su cantidad, el modelo relacional supera al modelo objeto relacional.

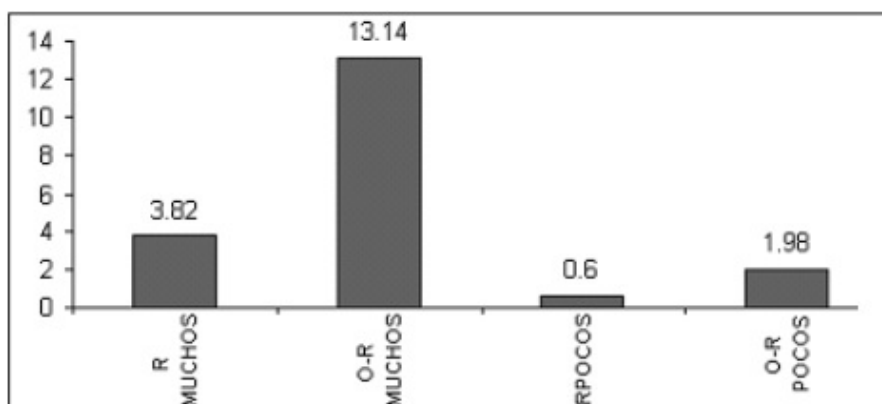


Figura C.1. Tasa de bloques leídos a filas procesadas para GROUP BY sencillo [13]

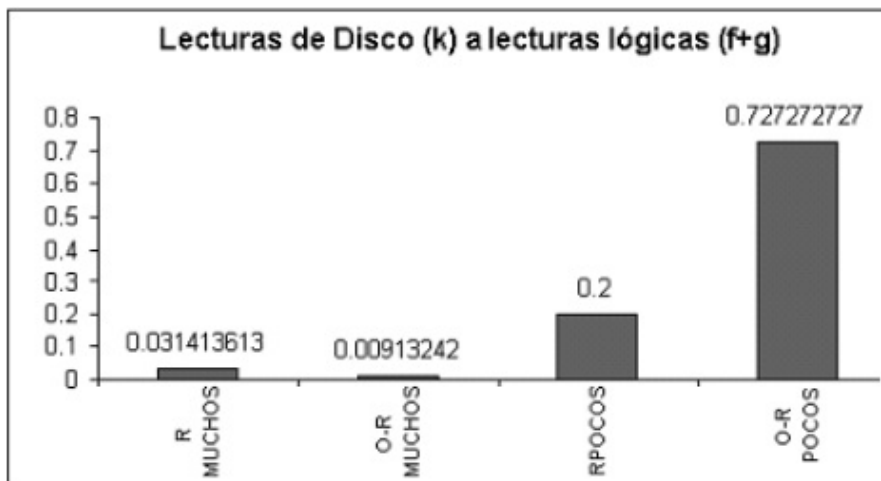


Figura C.2. Tasa de lecturas de disco a lecturas lógicas para GROUP BY sencillo [13]

En consultas del tipo GROUP BY con las cláusulas HAVING y ORDER BY, en la tasa de bloques leídos a filas procesadas, se observa en la figura C.3 nuevamente la ventaja que posee el modelo relacional con respecto al modelo objeto relacional especialmente cuando se ejecuta sobre muchos datos. De forma contraria, en la tasa de lecturas de disco a lecturas lógicas, se observa en la figura C.4 un comportamiento pobre en el modelo relacional, y muestra ser un modelo excesivamente costoso (para este caso) comparado con el modelo objeto relacional, el cual presenta valores bastante eficientes.

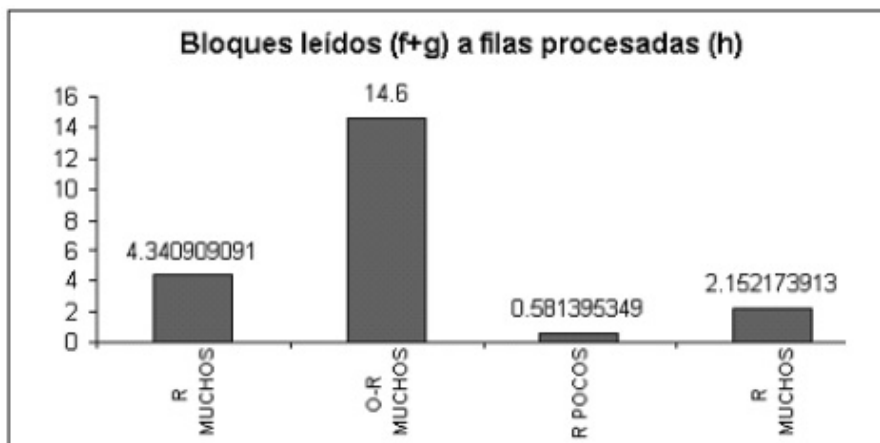


Figura C.3. Tasa de bloques leídos a filas procesadas para GROUP BY con las cláusulas HAVING y ORDER BY [13]

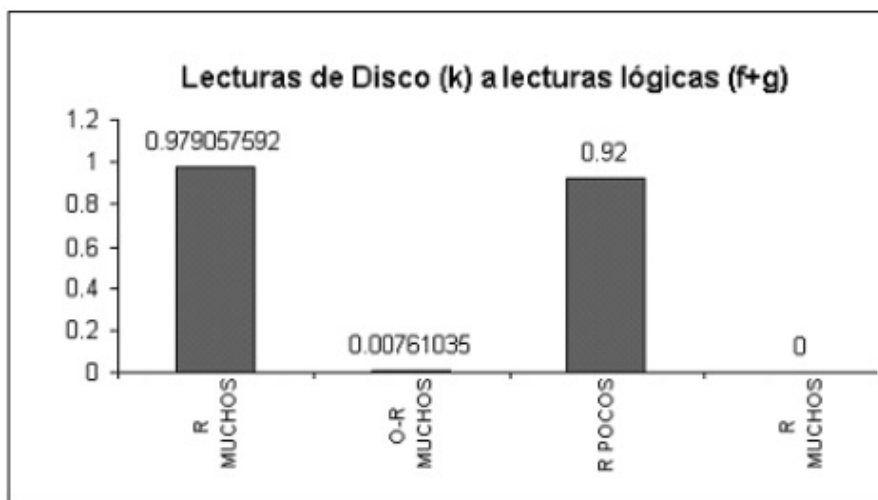


Figura C.4. Tasa de lecturas de disco a lecturas lógicas para GROUP BY con las cláusulas HAVING y ORDER BY [13]

Para las consultas que utilizan subconsultas, fueron probadas aquellas que se realizan con las cláusulas IN y EXISTS. Primero se mostrarán los resultados con las subconsultas cuando se utiliza la cláusula EXISTS, y luego los resultados cuando se utiliza la cláusula IN. En el primer caso se puede observar en la figura C.5 que en la tasa de bloques leídos a filas procesadas, existe una diferencia bastante significativa entre ambos modelos donde se visualiza una ganancia bastante visible del modelo relacional sobre el modelo objeto relacional, que en este caso muestra una eficiencia baja. De manera análoga a las pruebas realizadas sobre las subconsultas cuando se utiliza la cláusula EXISTS, las pruebas con la cláusula IN cuyos resultados son presentados en la figura C.6, muestran que en la tasa de bloques leídos a filas procesadas el modelo relacional ofrece de nuevo un rendimiento superior con respecto al modelo objeto relacional. Además de acuerdo con los resultados que, aunque la cláusula EXISTS debería ser más eficiente que la cláusula IN [18], en este caso particular los dos tipos de subconsultas ofrecen un rendimiento exactamente igual. En las demás tasas, en ambos casos, los dos modelos se comportaron de manera eficiente.



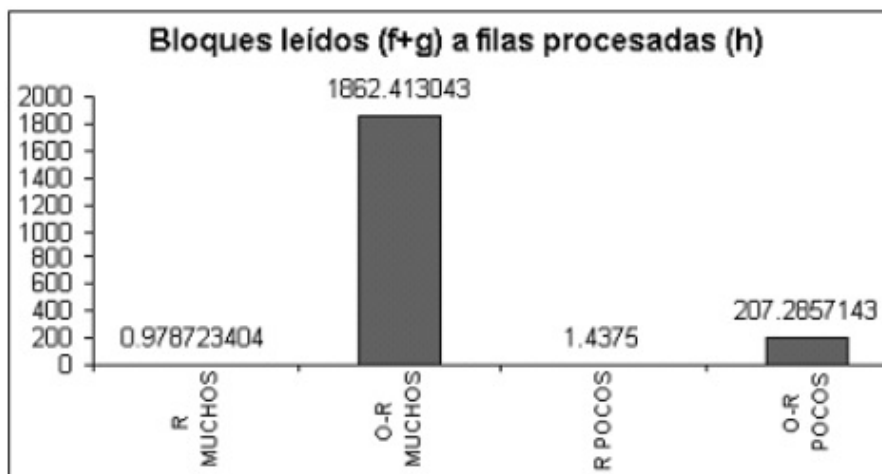


Figura C.5. Tasa de bloques leídos a filas procesadas para las subconsultas cuando se utiliza la cláusula EXISTS [13]

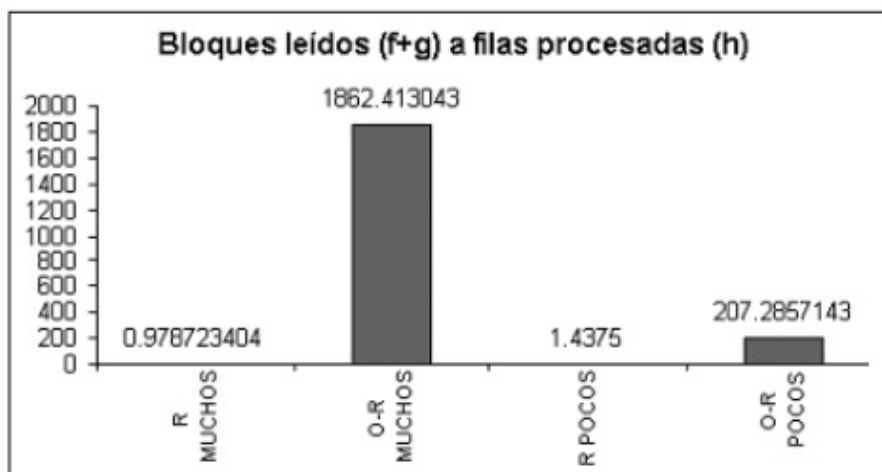


Figura C.6. Tasa de bloques leídos a filas procesadas para las subconsultas cuando se utiliza la cláusula IN [13]

## Anexo D

### Mapeo objeto relacional (ORM)

Para poder acceder a una base de datos relacional desde una aplicación que haya sido desarrollada basada en los principios de la programación orientada a los objetos, se requiere de una interfaz que permita traducir representaciones de datos de los sistemas de bases de datos relacionales, a representaciones de objetos, a esta interfaz se le denomina mapeo objeto-relacional (ORM, Object-Relational Mapping). Como los sistemas de administración de base de datos relacionales no poseen la flexibilidad para representar datos no escalares, como lo son arreglos y sólo soportan tipos de datos simples, haciéndose preciso la conversión de los objetos en un conjunto de valores simples, y viceversa, de manera que el objeto, sus propiedades y relaciones puedan ser recuperados.

La existencia de un ORM es primordial para el desarrollo de sistemas de software robustos y escalables. Típicamente los desarrolladores son quienes debían escribir el código asociado a la persistencia de la información, de forma que dicha información pudiera ser almacenada en las tablas de la base de datos.

Un principio universal en la mayoría de las aplicaciones es la independencia de los datos, teniendo en cuenta el gran número de ventajas que esto proporciona. Este principio propone que una base de datos debe ser administrada y mantenida de manera independiente de cualquier programa que haga uso de ella. No obstante, en el proceso de construcción de software esta independencia de datos ocasiona ciertos efectos negativos para los desarrolladores. Esto es debido a la mezcla de dos lenguajes incompatibles, como lo son el lenguaje de consulta para almacenar y recuperar los datos, y además el lenguaje de programación que deja a los usuarios interactuar con las bases de datos desde la aplicación. Entre estos dos modelos existe una brecha denominada desajuste por impedancia [11], dada por las diferencias entre uno y otro. Una de las principales diferencias se debe a que en los sistemas de bases de datos relacionales, los datos siempre son manejados en forma de tablas, constituidas por un conjunto de filas o tuplas;

mientras que en los entornos orientados a objetos los datos son manipulados como objetos, formados a su vez por objetos y tipos elementales.

Para disminuir los efectos del desajuste de impedancia entre ambos modelos, existen prácticas y técnicas como:

- Objetos de acceso a datos (Data Acces Objects o DAOs).
- Frameworks de persistencia (Persistence Frameworks).
- Mapeadores objeto relacionales (Object relational mappers u ORM).
- Consultas nativas (Native queries).
- Lenguajes integrados como: PL/SQL (Oracle) y T-SQL(SQLServer).
- Mediadores.
- Repositorios virtuales.
- Bases de datos orientadas al objeto.

A continuación se dará un mayor detalle en los tipos de mapeadores y los frameworks de persistencia, ya que son los conceptos más relevantes para el presente estudio.

## **D.1. Tipos de mapeadores**

Se presentarán los tipos básicos de mapeadores por diseño de originado por entidades de negocio. Se comenzará por un enfoque simple que es utilizado en aplicaciones muy simples tales como un libro de visitas en páginas de internet.

### **D.1.1. Enfoque a tuplas**

Cuando este enfoque es utilizado, es una pregunta válida si es que realmente se está hablando sobre mapeo objeto relacional, ya que las aplicaciones simplemente leen registros desde el SADB a un record set y luego trabajan con dichos record sets. Los datos no son leídos como objetos (entidades de negocio). Este enfoque no es apropiado para el desarrollo de sistemas de gran envergadura, porque la lógica de negocios se encuentra estrechamente unida con el formato de los datos en el almacén de datos. Cuando por ejemplo un atributo se mueve de una tabla a otra (por refactorización), todos los lugares en el código fuente de la aplicación, que trabajen con este atributo deben ser cambiados.

### D.1.2. Enfoque a entidades

Como en el caso anterior, la aplicación lee datos a un record set en primera instancia. Luego de leer a un record set, las entidades de negocios son llenadas con los datos del record set. Estas entidades de negocio tienen forma de estructuras de datos, las cuales son muy similares a las tablas de una base de datos relacional. Por ejemplo en ADO.NET se utilizan *DataSets* o *DataTables*. De hecho una *DataTable* de ADO.NET es una estructura tabular, la cual puede tener restricciones definidas como PK o FK. Por otro lado un *DataSet* es un conjunto de *DataTables*, entre los cuales se pueden definir relaciones, además posee soporte para actualización o eliminación en cascada. La desventaja de este enfoque se centra en que las entidades de negocios no poseen métodos, no tienen ningún comportamiento. Sólo se pueden observar métodos simples asociados con el procesamiento de bajo nivel de datos. La lógica de negocio está ubicada en lo que es llamado clases de gestión. Este tipo de diseño preserva el pensamiento relacional al nivel de objetos.

### D.1.3. Enfoque a modelo de dominio

Este enfoque es similar al enfoque a entidades, la diferencia está en las entidades de negocio. En este enfoque las entidades de negocios son objetos reales con comportamiento. Los objetos no sólo son contenedores para imágenes de tablas de bases de datos. Se puede utilizar herencia, poliformismo y otros conceptos de la orientación al objeto. La lógica de negocio está ubicada en los métodos de las entidades de negocio. No existe necesidad de clases de gestión, tal como se mencionó anteriormente.

## D.2. Frameworks de persistencia

La mayoría de las herramientas emplea un mapeo entre el mundo relacional y el mundo orientado a objetos. En este mapeo se declara como la herramienta que debe materializar o serializar el objeto desde o hacia la base de datos. Las herramientas normalmente también implementan un pseudo lenguaje para hacer consultas sobre los datos, este lenguaje es normalmente basado en objetos en vez de tablas y sus relaciones, la herramienta traduce después al SQL nativo de la base de datos. Muchas herramientas ORM también manejan conceptos como cache, connection pooling, entre otros. Las herramientas ORM tratan de esconder el mundo relacional al desarrollador para que el pueda poner sus esfuerzos en la lógica de la aplicación.

Utilizando herramientas ORM se acelera el proceso de desarrollo de software. Además se obtiene una independencia del motor de base de datos ya que el desarrollador no escribe SQL directamente sino que un pseudo lenguaje que es traducido al SQL específico de la base de datos configurada. Por otro lado la traducción adicional es un paso extra que toma más tiempo y por lo cual en algunos casos, especialmente reportes, el redimiendo de las consultas puede disminuir. Muchas bases de datos optimizan sus consultas y pueden mejorar los tiempos de respuesta al emplease procedimientos almacenados, al generar SQL en ejecución se pierde esta oportunidad de optimización. Algunas herramientas ORM no optimizan la consulta para la base de datos utilizada, sino que generan un SQL genérico lo cual resulta en una muy mala

Existen diversas herramientas que realizan la transformación según el lenguaje de programación y el motor de base de datos, en la tabla D-1 se muestran los productos de software ORM más destacados de la actualidad.

<b>Software</b>	<b>Plataforma</b>	<b>Disponibilidad</b>
Dapper	.NET 4.0	Open source
ECO	.NET 4.0	Comercial
EntitySpaces	.NET 4.0	Open source
EclipseLink	Java Virtual Machine	Open source
Hibernate	Java Virtual Machine	Open source
iBATIS	Multiplataforma	Open source
LLBLGen_Pro	.NET 4.0	Comercial
Microsoft ADO.NET Entity Framework	.NET 4.5	Parte de .NET 4.5
nHibernate	.NET 4.0	Open source
ODB	Multiplataforma C++	Licencia dual
SQLAlchemy	Python	Open source
SQLObject	Python	
Storm	Python	Open source
SubSonic	.NET 2.0	Open source
TopLink	Java Virtual Machine	Comercial
WebORB Integration Server	.NET, Java, PHP	Open source & comercial

Tabla D-1: Herramientas ORM populares en la actualidad