Aplicación de Persistencia Políglota con Sistemas de Almacenamiento NoSQL

Estefanía Gutiérrez Pérez

Directora:

Arantza Illarramendi Echave

Proyecto Fin de Carrera

February 2014

Abstract

A pesar del amplio uso de los sistemas de bases de datos relacionales, el gran crecimiento en el número de datos digitalizados disponibles; los nuevos requisitos de procesamiento que requieren de un alta disponibilidad y una alta escalabilidad; y la necesidad creciente de manejar grandes volumenes de datos por parte de los usuarios; ha hecho que en los últimos años se popularicen los llamados sistemas de bases de datos NOSQL. Sin embargo, la especialización y las características técnicas de estas nuevas tecnologías, hacen que algunos sistemas sean más adecuados para realizar ciertas tareas que otros.

Ésto unido al hecho de que cada vez más, las aplicaciones y servicios realizan tareas más diversas sobre los datos con la intención de ofrecer a los usuarios un servicio más completo, ha propiciado la aparición de la idea de *"Aplición de persistencia políglota"* enfocada al uso de múltiples sistemas de bases de datos de diferente naturaleza para gestionar los datos de una única aplicación. Este proyecto trata de mostrar las ventajas y usos de varios sistemas NoSQL; y así mismo, de mostrar su integración sobre una única aplicación, donde se intenta que cada sistema sea utilizado para realizar aquellas tareas para las cuales esté mejor adaptado.

Despite the popularity of relational data base systems, the high growth in the number of digitalize data; the new processing requeriments that ask for high availability and scalability of data; and the growing need the users have to manage large volume of data; has promote the use of so called NoSQL databases. However, the specialization and the technical characteristics of these new tecnologies has lead to the fact that some systems are better when performing some tasks than others.

All of this coupled with the fact that more and more applications and services are performing different tasks on the data, with the idea to give users a more complete service, has contribute to the emergence of the concept of "*Poliglot Persistence Aplication*", in which the main idea is the use of multiple and diverse database systems in order to manage the data of one application. This project tries to show the advantages and uses of various NoSQL systems; as well as to show how they can be integrated on a single application, where each system will be used to perform the tasks for which it is best suited.

Agradecimientos

Gracias a Arantza por permitir que con su duro trabajo, paciencia y consejos; fuera posible completar con éxito este proyecto. Gracias también a mis compañeros por aconsejarme a lo largo del proyecto, en especial gracias a David y a Alberto.

Gracias a mi familia por aconsejarme para que finalizase la carrera y apoyarme a lo largo de estos años.

Finalmente gracias, a todas aquellas personas, que me han ayudo o creído en mi.

Contenidos

Abstract i

Acknowledgements ii

List of Figures xi

List of Tables xiii

1 Introducción 1

1.1 Objetivos . . . 1
1.2 Características Generales de la Aplicación de Persistencia Políglota . . . 2
1.2.1 Antecedentes . . . 3
1.2.2 Conceptos Básicos . . . 4
1.2.3 Arquitectura . . . 5
1.3 Características Generales de los Sistemas NoSQL . . . 9
1.3.1 Antecedentes . . . 9
1.3.2 Conceptos Básicos . . . 10
1.3.2.1 Teorema de CAP . . . 11
1.3.2.2 Tipos de bases de datos NoSQL . . . 13
1.3.3 ¿Porqué NoSQL? . . . 15
1.4 Planificación . . . 16
1.4.1 Alcance . . . 16
1.4.1.1 Alcance mínimo . . . 17
1.4.1.1.1 Aplicación de persistencia políglota . . . 17
1.4.1.1.2 Diseño del entorno de trabajo . . . 18
1.4.1.2 Ampliación del alcance . . . 18
1.4.1.3 Criterios de aceptación . . . 18
1.4.1.4 Exclusiones del alcance . . . 19
1.4.2 Entregables del proyecto . . . 19
1.4.3 Esquema de desglose de trabajo . . . 19
1.4.4 Planificación temporal . . . 20
1.4.4.1 Actividades . . . 20
1.4.4.2 Actividades de la ampliación . . . 23
1.4.4.3 Hitos . . . 23
1.4.5 Cronograma . . . 24
1.4.6 Costes . . . 24

1.4.7 Análisis de calidad . . . 26
1.4.7.1 Calidad del proyecto . . . 26
1.4.7.2 Calidad de los entregables . . . 26
1.4.8 Comunicaciones . . . 27
1.4.8.1 Política de comunicación . . . 27
1.4.8.2 Reuniones . . . 27
1.4.9 Gestión de riesgos . . . 27

2 Entorno de Trabajo bajo MongoDB y OpenLink Virtuoso 30
2.1 MongoDB . . . 30
2.1.1 Herramientas . . . 32
2.1.2 Formato de los datos . . . 33
2.1.3 ¿Porqué MongoDB? . . . 33
2.1.4 Modelado de datos . . . 34
2.1.4.1 Documentos . . . 34
2.1.4.1.1 Restricciones de los documentos en MongoDB . 36
2.1.4.2 Diseño . . . 36
2.1.4.3 Patrones de diseño . . . 37
2.1.4.4 Diseño de un esquema de pruebas . . . 38
2.1.4.4.1 Diseño 1 . . . 39
2.1.4.4.2 Diseño 2 . . . 40
2.1.5 Diseño de operaciones . . . 42
2.1.5.1 Operaciones CRUD . . . 43
2.1.5.1.1 Operaciones de lectura . . . 43
2.1.5.1.2 Operaciones de escritura . . . 51
2.1.5.1.3 Métodos para cursores . . . 61
2.1.5.2 Entorno de agregación . . . 62
2.1.5.2.1 Operadores . . . 63
2.1.5.2.2 Secuencia de fases . . . 65
2.1.6 Índices . . . 66
2.1.6.1 Tipos de índices . . . 66
2.1.6.1.1 Índices de un sólo campo . . . 67
2.1.6.1.2 Índices compuestos . . . 67
2.1.6.1.3 Índices con valor múltiple . . . 68
2.1.6.1.4 Índices "text" . . . 68
2.1.6.1.5 TTL . . . 70
2.1.6.1.6 Sparse . . . 70
2.1.6.2 Diseño de índices . . . 70
2.2 OpenLink Virtuoso . . . 73
2.2.1 Grafos . . . 74
2.2.2 Almacenamiento de las tripletas . . . 75
2.2.3 Uso del sistema . . . 75
2.2.3.1 Interfaz Web . . . 75
2.2.3.2 Línea de comandos . . . 76
2.2.4 ¿Porqué Virtuoso? . . . 76
2.2.5 Modelado de datos . . . 77
2.2.5.1 Diseño Básico de un grafo . . . 78

2.2.5.2 Diseño de predicados . . . 79
2.2.5.3 Diseño del esquema de pruebas . . . 80
2.2.6 Diseño de operaciones . . . 81
2.2.6.1 Operaciones de Lectura SPARQL . . . 81
2.2.6.1.1 SELECT . . . 81
2.2.6.1.2 GRAPH . . . 82
2.2.6.1.3 FILTER . . . 83
2.2.6.1.4 OPTIONAL . . . 84
2.2.6.1.5 UNION . . . 85
2.2.6.1.6 Modificadores de resultados . . . 85
2.2.6.1.7 Funciones de agregación . . . 86
2.2.6.1.8 CONSTRUCT . . . 86
2.2.6.1.9 ASK . . . 87
2.2.6.2 Operaciones de Escritura SPARQL . . . 87
2.2.6.2.1 DELETE . . . 87
2.2.6.2.2 INSERT . . . 87
2.2.6.2.3 MODIFY . . . 88
2.2.6.3 Operaciones sobre grafos . . . 88
2.2.6.3.1 CREATE . . . 89
2.2.6.3.2 CLEAR y DROP . . . 89
2.2.6.3.3 COPY . . . 89
2.2.6.3.4 MOVE . . . 89
2.2.6.3.5 ADD . . . 89
2.2.6.4 Formato de salida de los datos . . . 90
2.2.6.5 Funciones dentro de SPARQL . . . 90
2.2.6.5.1 Ejemplo . . . 90
2.2.6.6 Funciones Virtuoso PL . . . 91
2.2.6.6.1 Declaración de variables . . . 92
2.2.6.7 Sentencias Condicionales . . . 92
2.2.6.7.1 Sentencias de salto . . . 93
2.2.6.7.2 Llamada a otros métodos . . . 93
2.2.6.7.3 Sentencias SPARQL . . . 93
2.2.6.7.4 Uso de las funciones . . . 93
2.2.6.8 Reglas de inferencia . . . 94
2.2.6.8.1 Ejemplo . . . 95
2.2.7 Índices . . . 96
2.2.7.1 Crear índices . . . 97
2.2.7.2 Eliminar y Alterar índices . . . 97

3 Aplicación de Persistencia Políglota: Diseño **98**
3.1 Arquitectura . . . 98
3.1.1 Arquitectura general . . . 98
3.1.2 Servicios . . . 99
3.1.3 Aplicación Web . . . 104
3.2 Modelo de Casos de Uso . . . 105
3.2.0.1 Ver Contenidos . . . 106
3.2.0.1.1 Caso de uso: Buscar Artículo . . . 106

3.2.0.1.2 Caso de uso: Ver Artículo . . . 107
3.2.0.1.3 Caso de uso: Navegar . . . 108
3.2.0.2 Gestión de Cesta . . . 109
3.2.0.2.1 Caso de uso: Ver Cesta . . . 109
3.2.0.2.2 Caso de uso: Modificar Cesta . . . 110
3.2.0.2.3 Caso de uso: Añadir a Cesta . . . 111
3.2.0.3 Control de usuarios . . . 111
3.2.0.3.1 Caso de uso: Identificarse . . . 112
3.2.0.3.2 Caso de uso: Registrarse . . . 112
3.2.0.3.3 Caso de uso: Desconectarse . . . 113
3.2.0.4 Clientes . . . 113
3.2.0.4.1 Caso de uso: Realizar Pedido . . . 114
3.2.0.4.2 Caso de uso: Ver Pedido . . . 115
3.2.0.4.3 Caso de uso: Cancelar Pedido . . . 116
3.2.0.4.4 Caso de uso: Ver Compras . . . 116
3.2.0.4.5 Caso de uso: Ver Valoraciones del Usuario . . . 117
3.2.0.4.6 Caso de uso: Eliminar Valoraciones . . . 117
3.2.0.4.7 Caso de uso: Añadir Valoración . . . 117
3.2.0.4.8 Caso de uso: Añadir Like . . . 118
3.3 Modelo de Datos . . . 118
3.3.1 Modelo de datos en MongoDB . . . 119
3.3.1.1 Colección Artículos . . . 119
3.3.1.2 Colección Carrito . . . 121
3.3.1.3 Colección Pedidos . . . 122
3.3.2 Modelo de datos en Virtuoso . . . 123
3.3.3 Modelo de datos en MySQL . . . 124
3.3.4 Relación entre los datos . . . 125
3.4 Modelo de Clases . . . 125
3.4.1 Servicios . . . 125
3.4.1.1 Clase ConexVirtuoso . . . 127
3.4.1.2 Clase ConexMongoDB . . . 127
3.4.1.3 Clase ConexMySQL . . . 128
3.4.1.4 Clase ServicioUsers . . . 129
3.4.1.5 Clase ServicioValoraciones . . . 129
3.4.1.6 Clase ServicioCompras . . . 130
3.4.1.7 Clase ServicioRecomendaciones . . . 131
3.4.1.8 Clase ServicioPedidos . . . 131
3.4.1.9 Clase ServicioCarrito . . . 132
3.4.1.10 Clase ServicioDatosArticulos . . . 133
3.4.1.11 Clase ServicioInventario . . . 134
3.4.2 Clases de Objetos . . . 135
3.4.2.1 Clase filtro . . . 137
3.4.2.2 Clase Par . . . 137
3.4.2.3 Clase Compra . . . 138
3.4.2.4 Clase Reviews . . . 138
3.4.2.5 Clase PedidoArt . . . 138
3.4.2.6 Clase CarritoArt . . . 139

3.4.2.7 Clase Pedido . . . 139
3.4.2.7.1 Clase anidada Pago . . . 140
3.4.2.7.2 Clase anidada Envío . . . 140
3.4.2.8 Clase objArticle . . . 140
3.4.2.8.1 Clase anidada Precio . . . 140
3.4.2.9 Clase MovArtic . . . 141
3.4.2.9.1 Clase anidada Detalles . . . 141
3.4.2.10 Clase BookArtic . . . 141
3.4.2.10.1 Clase anidada Detalles . . . 141
3.4.3 Clases de la aplicación Web . . . 141
3.4.3.1 Clases de presentación . . . 143
3.4.3.2 Clase GestorBusqueda . . . 144
3.4.3.3 Clase GestorNavegarArt . . . 144
3.4.3.4 Clase GestorPresentacionArticulo . . . 144
3.4.3.5 Clase GestorCesta . . . 145
3.4.3.6 Clase GestorClientes . . . 145
3.4.3.7 Clase GestorRecomendaciones . . . 145
3.4.3.8 Clase GestorValoraciones . . . 146
3.4.3.9 Clase GestorPedidos . . . 146

4 Aplicación de Persistencia Políglota: Implementación 147
4.1 Desarrollo de la aplicación . . . 147
4.2 Estructura de la aplicación . . . 148
4.3 Diagramas de secuencia . . . 150
4.4 Pruebas de errores . . . 164
4.5 Tecnologías de Desarrollo . . . 166
4.5.1 Implementación central de la aplicación . . . 166
4.5.1.1 Java . . . 166
4.5.1.2 J2EE . . . 167
4.5.1.3 JSON y GSON . . . 167
4.5.2 Implementación de la página web . . . 168
4.5.2.1 HTML y CSS . . . 168
4.5.2.2 JavaScript y jQuery . . . 168
4.5.3 Pruebas . . . 168
4.5.3.1 JUnit . . . 168
4.5.3.2 Mozilla Firefox . . . 169
4.5.4 Servidor Web . . . 169
4.5.4.1 Apache Tomcat . . . 169
4.5.5 Desarrollo . . . 169
4.5.5.1 Netbeans . . . 169
4.5.6 Otros . . . 170
4.5.6.1 FreeBase . . . 170
4.5.6.2 Dropbox . . . 170

5 Aplicación con un Sistema Relacional: Comparación 171
5.1 Arquitectura . . . 171
5.2 Modelo de datos . . . 172

5.2.1 Comparación con MongoDB . . . 172
5.2.2 Comparación con Virtuoso . . . 174
5.2.3 Comparación con MySQL . . . 175
5.3 Consultas . . . 175
5.3.1 Comparación con MongoDB . . . 176
5.3.2 Comparación con Virtuoso . . . 177
5.4 Datos en la aplicación . . . 179
5.4.1 Comparación General . . . 179
5.4.2 Comparación con MongoDB . . . 180
5.5 Pruebas de rendimiento . . . 182
5.5.1 Preparación de los Sistemas . . . 182
5.5.1.1 MongoDB . . . 183
5.5.1.2 Virtuoso . . . 183
5.5.1.3 MySQL . . . 184
5.5.2 Preparación de las pruebas . . . 184
5.5.3 Resultados . . . 185
5.6 Conclusiones . . . 188

6 Extensión a un Entorno de Trabajo Distribuido 190
6.1 MongoDB . . . 190
6.1.1 Replicación . . . 190
6.1.1.1 Introducción . . . 191
6.1.1.2 Inicio de los servidores . . . 191
6.1.1.3 Preparación del Sistema de Replicación . . . 192
6.1.1.4 Elecciones . . . 193
6.1.2 Sharding . . . 197
6.1.2.1 Introducción . . . 197
6.1.2.2 Inicio de las servidores . . . 198
6.1.2.3 Preparación del sistema . . . 199
6.2 OpenLink Virtuoso . . . 200
6.2.1 Replicación . . . 201
6.2.1.1 Introducción . . . 201
6.2.1.2 Inicio de los Servidores . . . 201
6.2.1.3 Preparación del Sistema de Replicación . . . 203
6.2.1.4 Comportamiento . . . 205
6.2.2 Sharding . . . 205
6.2.2.1 Introducción . . . 205
6.2.2.2 Preparación del sistema . . . 206
6.2.2.3 Partición de los índices . . . 209

7 Conclusiones y Líneas Futuras 210
7.1 Conclusiones . . . 210
7.2 Líneas Futuras . . . 211
7.3 Valoración Personal . . . 212

A Manual de Instalación 213

A.1 Requisitos . . . 213
A.2 MongoDB . . . 213
A.2.1 Instalación de MongoDB . . . 214
A.2.2 Configuración de usuarios . . . 214
A.3 Openlink Virtuoso . . . 215
A.3.1 Instalación . . . 215
A.3.2 Configuración . . . 215
A.4 MySQL . . . 217
A.4.1 Instalación de Xampp . . . 217
A.4.2 Configurar de MySQL . . . 219
A.5 Oracle Java . . . 222
A.6 Apache Tomcat . . . 223
A.6.1 Instalación . . . 223
A.6.2 Configuración . . . 224
A.7 Desplegar la aplicación . . . 225

B Seguimiento y Control **226**
B.1 Desarrollo de los Objetivos . . . 226
B.2 Planificación temporal . . . 227
B.2.1 Actividades . . . 227
B.2.2 Actividades de la ampliación . . . 232
B.2.3 Hitos y replanificaciones . . . 233

C MongoDB: Información Adicional **236**
C.1 Instalación . . . 236
C.1.1 Instalación Básica . . . 236
C.1.2 Inicio de la instancia del servidor . . . 237
C.1.3 Configuración . . . 237
C.2 Control de usuarios . . . 238
C.2.1 Creación de un administrador . . . 238
C.2.2 Creación del resto de usuarios . . . 241
C.2.3 Modificación de usuarios . . . 242
C.3 Concurrencia . . . 242
C.3.1 Ejemplo 1 . . . 243
C.3.2 Ejemplo 2 . . . 244
C.4 Transacciones . . . 246
C.4.1 Operador $isolation . . . 247
C.4.2 Método findAndModify . . . 247
C.4.3 Modificar si es el actual . . . 248
C.5 Administración . . . 248
C.5.1 Volcado de datos . . . 248
C.5.1.1 Ejemplo . . . 249
C.5.2 Recuperación de datos . . . 250
C.6 Pruebas . . . 250
C.6.1 Lectura . . . 251
C.6.1.1 Diseño 1 . . . 251
C.6.1.2 Diseño 2 . . . 253

C.6.2 Escritura . . . 256
C.6.2.1 Diseño 1 . . . 256
C.6.2.2 Diseño 2 . . . 257
C.6.3 Comparación . . . 258

D OpenLink Virtuoso: Información Adicional 260
D.1 Instalación . . . 260
D.1.1 Instalación Básica . . . 260
D.2 Control de Usuarios . . . 261
D.2.1 Creación de usuarios . . . 261
D.2.2 Permisos para realizar consultas SPARQL . . . 262
D.2.3 Permisos sobre grafos . . . 263
D.3 Concurrencia . . . 264
D.4 Transacciones . . . 266
D.5 Administración . . . 268
D.5.1 Mejora de rendimiento . . . 268
D.5.2 Dump y load de datos RDF . . . 268
D.5.2.1 Dump . . . 269
D.5.2.2 Load . . . 269
D.6 Pruebas . . . 270
D.6.1 Lectura . . . 271
D.6.2 Escritura . . . 273

E Lista de entregables 275
E.1 Entregables relacionados con el entorno de trabajo . . . 275
E.1.0.1 Documento del entrono . . . 275
E.1.0.2 MongoDB . . . 275
E.1.0.3 Virtuoso . . . 276
E.2 Entregables de la aplicación políglota . . . 277
E.2.1 Acceso a la aplicación . . . 277
E.2.2 Ejecutable . . . 278
E.2.3 Código Fuente y Javadoc . . . 278
E.2.3.1 Servicios . . . 278
E.2.3.2 Aplicación Principal . . . 279
E.2.4 Tabla de Inventario . . . 279
E.3 Entregables de la aplicación relacional . . . 280
E.3.1 Acceso a la aplicación . . . 280
E.3.2 Ejecutable . . . 280
E.3.3 Código Fuente y Javadoc . . . 280
E.3.4 Datos MySQL . . . 281
E.4 Otros . . . 281

Bibliography 284

Lista de figuras

1.1 Ejemplo de una arquitectura con un único sistema de almacenamiento . . 5
1.2 Arquitectura para una *Aplicación de Persistencia Políglota*: Diseño I . . . 6
1.3 Arquitectura para una *Aplicación de Persistencia Políglota*: Diseño II . . 7
1.4 Arquitectura para una *Aplicación de Persistencia Políglota*: Diseño III . . 8
1.5 Datos estructurados vs no estructurados [1] 10
1.6 Sistema de Bases de Datos Basado en Documentos 13
1.7 Sistema de Bases de Datos Basado en Grafos 14
1.8 Sistema de Bases de Datos Basado en Clave-Valor 14
1.9 Sistema de Bases de Datos Basado en Familia de columnas 15
1.10 Esquema de desglose de trabajo 20
1.11 Cronograma 25

3.1 Distribución de los servicios 103
3.2 Arquitectura en Tres Capas 105
3.3 Casos de usos: Ver Contenidos 106
3.4 Casos de usos: Gestión de Cesta 109
3.5 Casos de usos: Control de usuarios 111
3.6 Casos de usos: Operaciones para Clientes 114
3.7 Modelo de datos en MongoDB:(a) artículos, (b) carritos, (c) pedidos . . . 119
3.8 Modelo de datos en Virtuoso 124
3.9 Modelo de datos en MySQL 124
3.10 Diagrama de Clases: Servicios 126
3.11 Clase: ConexVirtuoso 127
3.12 Clase: ConexMongoDB 128
3.13 Clase: ConexMySQL 128
3.14 Clase: ServiciosUsers 129
3.15 Clase: ServicioValoraciones 130
3.16 Clase: ServicioCompras 130
3.17 Clase: ServicioRecomendaciones 131
3.18 Clase: ServicioPedidos 132
3.19 Clase: ServicioCarrito 133
3.20 Clase: ServicioDatosArticulos 134
3.21 Clase: ServicioDatosArticulos 135
3.22 Modelos de Clases: Objetos 136
3.23 Clase: Par 137
3.24 Diagrama de Clases:Aplicación Central 142

4.1 Estructura de la aplicación 149

4.2 Diagrama de secuencia del caso de uso Navegar . . . 151
4.3 Diagrama de secuencia del caso de uso Buscar Artículo . . . 152
4.4 Diagrama de los casos de uso Ver Cesta . . . 152
4.5 Diagrama de secuencia del caso de uso Ver Artículo (I) . . . 153
4.6 Diagrama de secuencia del caso de uso Ver Artículo (II) . . . 154
4.7 Diagrama de secuencia del caso de uso Insertar en Cesta . . . 155
4.8 Diagrama de secuencia del caso de uso modificar Cesta: eliminar artículo . 155
4.9 Diagrama de secuencia del caso de uso modificar Cesta (II): modificar cantidad de un artículo . . . 156
4.10 Diagrama de secuencia del caso de uso Registrarse . . . 157
4.11 Diagrama de secuencia del caso de uso Login . . . 158
4.12 Diagrama de secuencia del caso de uso Desconectarse . . . 159
4.13 Diagrama de secuencia del caso de uso Ver Pedidos del usuario . . . 159
4.14 Diagrama de secuencia del caso Ver Compras del cliente . . . 159
4.15 Diagrama de secuencia del caso Ver Valoraciones de los Usuarios . . . 160
4.16 Diagrama de secuencia del caso de uso Eliminar Valoración . . . 160
4.17 Diagrama de secuencia del caso de uso Añadir Valoración . . . 160
4.18 Diagrama de secuencia del caso de uso Realizar Pedido (I) . . . 161
4.19 Diagrama de secuencia del caso de uso Realizar Pedido (II) . . . 162
4.20 Diagrama de secuencia del caso de uso Realizar Pedido (III) . . . 163
4.21 Diagrama de los casos de uso Cancelar Pedido . . . 164

5.1 Modelo de datos para MySQL . . . 173

A.1 Identificarse en la interfaz web de Virtuoso . . . 216
A.2 Crear un usuario (I) . . . 216
A.3 Crear un usuario (II) . . . 217
A.4 Instalar Xampp . . . 218
A.5 Finalizar Instalación Xampp . . . 218
A.6 Acceder a phpMyAdmin . . . 219
A.7 Crear una Base de datos en MySQL . . . 220
A.8 Configurar usuario de la base de datos en MySQL (I) . . . 220
A.9 Configurar usuario de la base de datos en MySQL (II) . . . 221
A.10 Configurar usuario de la base de datos en MySQL (III) . . . 221
A.11 Crear tabla inventario (I) . . . 222
A.12 Crear tabla inventario (II) . . . 222
A.13 Desplegar archivo .war . . . 225

Lista de Tablas

1.1 Tiempo de planificación estimado 20
1.2 Tiempo de getión estimado 21
1.3 Tiempo de formación estimado 21
1.4 Tiempo de desarrollo estimado 22
1.5 Tiempo de documentación estimado 22
1.6 Estimación del tiempo de la ampliación 23
1.7 Fecha de Hitos 24

5.1 Comparación del rendimiento entre aplicaciones 186
5.2 Comparación del rendimiento entre aplicaciones (II) 188

B.1 Tiempo de planificación estimado y real 228
B.2 Tiempo de gestión estimado y real 228
B.3 Tiempo de formación estimado y real 229
B.4 Tiempo de desarrollo estimado y real 230
B.5 Tiempo de documentación estimado y real 231
B.6 Estimación del tiempo de la ampliación y real 232
B.7 Fecha de Hitos 233
B.8 Fecha de Hitos segunda replanificación 235

Capítulo 1

Introducción

En este capítulo se va a hablar de los objetivos principales del proyecto, y de las características de las principales tecnologías involucradas en el proyecto.

1.1 Objetivos

En los últimos años se han popularizado los sistemas de bases de datos NoSQL y con ellos, recientemente, ha surgido la idea de *Aplicación de Persistencia Políglota*. Ésta sostiene que debido a la gran variedad y cantidad de datos, y los diversos servicios que pueden dar las aplicaciones hoy en día; es posible que un único tipo de sistema de almacenamiento no sea capaz de cubrir de forma eficiente **todas** las necesidades de la aplicación que use dicho sistema. Sin embargo, sostiene que las aplicaciones se pueden beneficiar del uso de varios sistemas de distinto tipo donde los datos se repartirían entre los sistemas que mejor fueran capaces de dar acceso a estos en función del tipo de datos, y de las tareas que se realizarán con ellos. Además, con esta idea también se considera que se tiene que ir más allá de los sistemas de almacenamiento relacionales y utilizar, **además**, sistemas de almacenamiento NoSQL.

El objetivo de este proyecto, por tanto, es el de crear una *Aplicación de Persistencia Políglota* que permita interactuar con varios tipos de sistemas de bases de datos, siendo la interacción con los diferentes sistemas lo más transparente posible. Al mismo tiempo, también es objetivo del proyecto aprender, manejar y diseñar un entorno de trabajo para dos sistemas de bases de datos NoSQL.

Para lograr esto, en este proyecto se han llevado a cabo dos tareas principales: (1) por un lado se ha hecho un estudio de dos sistemas de almacenamiento NOSQL de distinto

tipo - MongoDB y OpenLink Virtuoso; y (2) se ha creado una aplicación web que usa estos dos sistemas junto a un sistema relacional.

Para el primer punto, se ha escrito un documento - cuyo contenido se encuentra en los siguientes capítulos de la memoria, pero también se puede encontrar en un documento separado - donde se detallan las características y la forma en la que se usan los dos sistemas. Este documento está orientado a personas que tienen conocimientos de sistemas relacionales y se intenta explicar cómo en los sistemas no relacionales se llevarían a cabo ciertas tareas que se pueden hacer en los sistemas relacionales - si es posible hacerlas. En concreto se centra, entre otros detalles, en explicar aspectos referentes al: control de usuario, modelado de datos, índices, concurrencia y transacciones y las diferentes formas en las que se pueden realizar las consultas. Además, se han generado varios *datasets* para cada sistema, con los que seguir la documentación - parte de los datos de estos *datasets* serán reutilizados para la parte de la aplicación.

Para el segundo punto, se ha desarrollado una aplicación que gestiona una tienda electrónica la cual hace uso de varios sistemas de almacenamiento. Se presentarán los distintos problemas que surgen durante su diseño e implementación y las diferentes soluciones que se pueden adoptar.

Finalmente, en este documento se intenta: (1) dar una idea general de los sistemas de almacenamiento NoSQL y de las *Aplicaciones de Persistencia Políglota*;(2) dar información concreta del funcionamiento y configuración de los sistemas NoSQL, MongoDB y OpenLink Virtuoso; (3) mostrar las diferentes decisiones de diseño e implementación llevadas a cabo para realizar la aplicación; y (4) realizar una comparación con una aplicación que sólo utilice un sistema relacional. Para este último punto, se ha creado una aplicación idéntica - reutilizando la mayor parte del código - pero únicamente se vale del sistema relacional de almacenamiento MySQL. La comparación de estás dos implementaciones se ha basado tanto a nivel de diseño como a nivel de rendimiento. Cabe destacar que la información de diseño e implementación de esta segunda aplicación está orientada a la comparación con la aplicación original; y por tanto no se entra en los detalles de la implementación o del diseño no orientados a la comparación- que en su mayoría serán similares a los da la aplicación original.

1.2 Características Generales de la Aplicación de Persistencia Políglota

En esta sección se va a explicar en qué consiste una *"Aplicación de Persistencia Políglota"* y cuales son sus posibles beneficios frente a otras aplicaciones más tradicionales.

1.2.1 Antecedentes

La tendencia de muchas empresas al desarrollar una aplicación es la de utilizar un único sistema de almacenamiento para guardar los datos, independientemente de las operaciones que hagan con ellos [2]. En ciertos casos las aplicaciones son sencillas y realizan una operación concreta o varias muy relacionadas, con lo que el uso de un único sistema de bases de datos no sería problemático.

Sin embargo, cada vez más, se ve a las aplicaciones como un servicio que puede realizar gran variedad de tareas, bien por la naturaleza misma del servicio o porque se quiere enriquecer la experiencia del usuario. Por ejemplo, la llamadas redes sociales, donde el usuario crea contenido propios, mantiene comunicaciones con otros usuarios con los que tiene alguna relación, recibe publicidad personalizada...

Estas aplicaciones, más complejas, realizan diferentes tareas sobre distintos tipos de datos. En este caso, el uso de un único sistema de bases de datos puede evitar que se obtenga un buen rendimiento [2], puesto que cada tipo de datos y las operaciones que se quieren realizar sobre estos tienen distintas necesidades. Por ejemplo, la información sobre los artículos de un comercio no tendrá las mismas propiedades o necesidades de consistencia que la información sobre la disponibilidad de los artículos.

Para una misma aplicación es posible que se quieran tener unos ciertos datos que estén disponibles en el menor tiempo posible, independientemente de si todos los usuarios pueden ver los datos introducidos hace un segundo. Otros querrán que los datos sean siempre consistentes, otros que estén altamente relacionados entre ellos...

Todo esto resulta ser problemático al usar una única base de datos, debido a que distintos sistemas de bases de datos están diseñados para resolver distintos problemas. Ésto hace que los sistemas de almacenamiento que sean buenos para una única tarea no sean la mejor solución para otra. Debido a esto, y junto con la popularización de los sistemas NoSQL, surge la idea de utilizar varios sistemas de bases de datos de diferente tipo en una misma aplicación.

Lo que se busca al usar varios sistemas al mismo tiempo es que se dividan los datos entre los distintos sistemas en función de las necesidades de los datos y las tareas que hagan uso de dichos datos. Todo esto con la idea de que el sistema usado sea el mejor posible en cada caso para cubrir dichas necesidades.

1.2.2 Conceptos Básicos

El termino *"Aplicación de Persistencia Políglota"* describe la idea de que las aplicaciones deberían utilizar diferentes sistemas de bases de datos al mismo tiempo, pensando en que se utilice el sistema de bases de datos más adecuado para cada tarea. Esto se debe a que como se ha indicado antes, algunos tipos de bases de datos están mejor diseñadas para realizar ciertas tareas que otros.

En lugar de decidir usar un sistema que cubra las necesidades de una de las tareas y que el resto de datos y tareas se adapten al sistema usado, se busca que los sistemas se adapten a la aplicación. Es decir, lo que se busca con este tipo de aplicaciones es que primero se decida qué tareas se van a realizar, con qué tipo de datos y cuales serán sus necesidades; y después elegir el sistema que mejor se adapte a las necesidades de cada tarea. De esta forma se tendrán varios sistemas de bases de datos para cada aplicación.

Por ejemplo, suponiendo que se quiere implementar una red social básica con la información de los usuarios y, además, con información sobre sus actividades y las de sus "amigos", si se utiliza un diseño simple con un único sistema de almacenamiento, se guardará en éste la información de los usuarios, la cual puede incluir tanto los datos personales, como el contenido creado por ellos mismos. Y en el mismo sistema habría que guardar la información sobre las relaciones que tiene con otros usuarios.

Sin embargo, si se utilizan varios sistemas, se podría, por ejemplo, guardar la información personal y los datos de los contenidos en un sistema de familia de columnas o de documentos que permita representar datos complejos. Y por otro lado, las relaciones de los usuarios y sus acciones en un sistema de grafos que permite almacenar datos altamente relacionados de forma simple.

De esta forma, si la aplicación necesitase generar recomendaciones basadas en los gustos de sus "amigos" podría utilizar la base de datos de grafos. Y si se quiere visualizar la página del usuario se podría usar el otro sistema para obtener los datos de los contenidos creados.

El principal problema a la hora de crear este tipo de aplicación es que debe comunicarse con sistemas de bases de datos diferentes. Por lo que en principio ésto podría hacer que la aplicación fuera muy compleja. En el siguiente punto se muestran las diferentes opciones que se pueden seguir al diseñar este tipo de aplicaciones y como evitar, en parte, dicha complejidad.

1.2.3 Arquitectura

Uno de los puntos más interesantes en las *Aplicaciones de Persistencia Políglota* - además de los sistemas de almacenamiento en si - es la forma en la que se debería estructurar la aplicación. La presencia de varios sistemas de almacenamiento trae consigo una serie de problemas y necesidades que no existen en las aplicaciones con un único sistema de almacenamiento.

En una aplicación simple se tendría un único sistema al cual se accedería para realizar las diferentes operaciones, independientemente de qué tarea se trate o qué datos se quieran consultar. Por tanto, tal y como se muestra en la figura 1.1, la aplicación realiza operaciones que necesitan hacer diferentes consultas, pero todas ellas se hacen sobre el mismo sistema de almacenamiento.

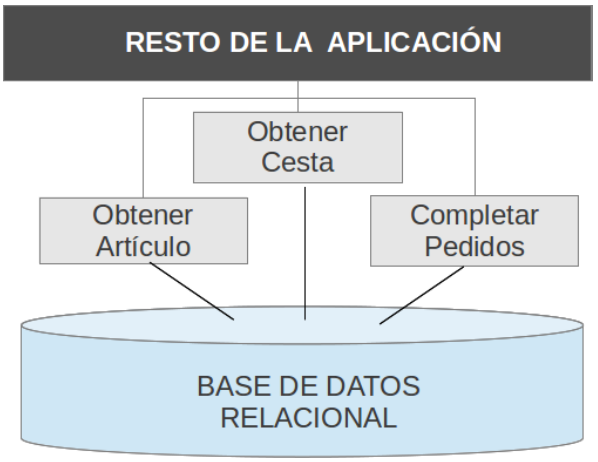

FIGURA 1.1: Ejemplo de una arquitectura con un único sistema de almacenamiento

Al introducir nuevos sistemas de almacenamiento, la aplicación deberá realizar las consultas de los datos sobre distintos sistemas. Esto genera nuevas cuestiones sobre cómo diseñar la arquitectura. ¿Es necesario que la aplicación central sepa que hay varios sistemas? ¿Es posible que el uso de varios sistemas sea transparente? ¿Qué ocurre cuando se cambia uno de los sistemas o se añade uno nuevo? Teniendo en cuenta estas cuestiones y los posibles escenarios que se pueden presentar se observa que existen diferentes opciones de diseño que se pueden tomar. En general la elección de qué diseño usar vendrá dada por las necesidades de cada aplicación.

Un primer diseño sería el más directo. En este caso, cada parte de la aplicación se comunica directamente con el sistema de almacenamiento que contiene los datos a los que se quiere acceder. Tal y como se puede ver en la figura 1.2 la aplicación realiza operaciones

sobre cada uno de los sistemas de almacenamiento según lo vea necesario. En principio este diseño genera varios problemas, pero uno de los principales es que la presencia de las tres bases de datos no es transparente para la aplicación.

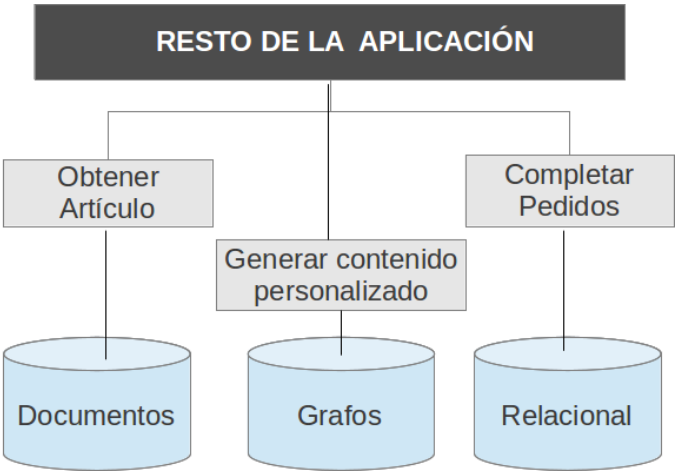

FIGURA 1.2: Arquitectura para una *Aplicación de Persistencia Políglota*: Diseño I

El segundo diseño intenta eliminar los problemas anteriores. La idea es envolver las bases de datos en servicios, de forma que la/s aplicacion/es no se comunique/n directamente con los sistemas de almacenamiento [2]. Éstas se comunicarán con los servicios. Éstos servicios serán los encargados de realizar las consultas sobre las bases de datos. Con esto, por un lado se consigue: (1) que la distribución de los datos entre diferentes sistemas de almacenamientos sea transparente para la aplicación; y (2) que los servicios puedan ser reutilizados por otras aplicaciones que necesiten acceder a esos datos. Por tanto, una arquitectura con este diseño tendría una forma similar al de la figura 1.3

Siguiendo la idea anterior del uso de servicios se puede considerar que hay dos posibles opciones de diseño. Por un lado, se podría tener un servicio por cada base de datos. Por ejemplo, si en un sistema de documentos se guardan datos sobre los artículos de un comercio y sobre la cesta de la compra, se podría tener un servicio que se encargara de realizar consultas tanto sobre los datos de los artículos como de las cestas; tal y como se muestra en la figura anterior 1.3 .

Aunque el diseño anterior es perfectamente valido existen algunas cuestiones sin resolver. Si, por ejemplo, se quisiera utilizar un nuevo sistema de almacenamiento para guardar la información de la cesta. En este caso, además de crear el nuevo servicio, habría que modificar todo el servicio que envolvía el sistema de documentos, el cual contenía a las cestas. Otro problema que surge si se ponen varios objetos sobre un mismo servicio es que acceder a los servicios desde una segunda aplicación puede resultar algo complejo. Por

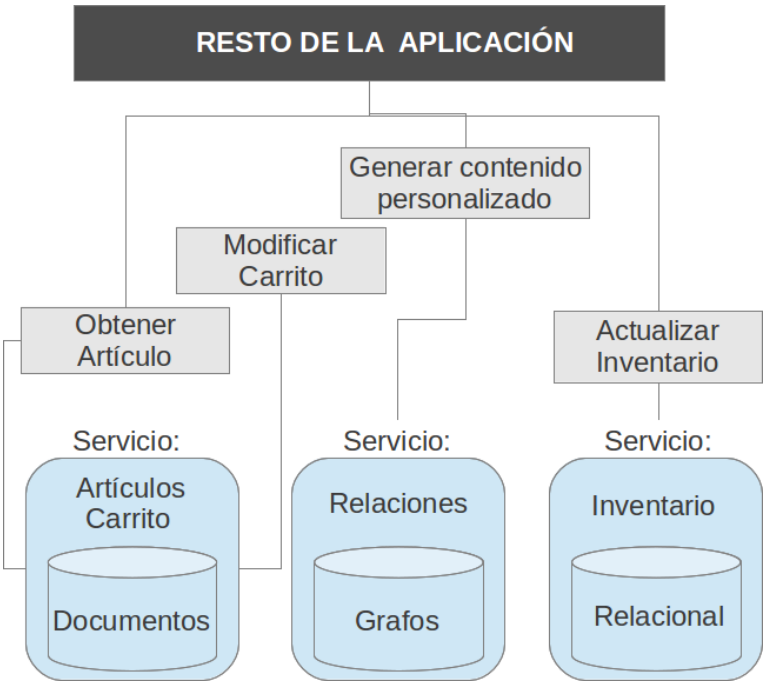

FIGURA 1.3: Arquitectura para una *Aplicación de Persistencia Políglota*: Diseño II

ejemplo, si se quisiese acceder a los datos de los artículos desde una segunda aplicación, pero no dar acceso a los datos de las cestas, sería necesario, por ejemplo, crear un servicio especial que sirviera como puente entre el servicio anterior y la aplicación.

Por tanto, una segunda opción de un diseño basado en servicios, es crear servicios en función de los datos que se guardan y no en función de los sistemas. De esta forma si se quiere utilizar un sistema de almacenamiento de tipo clave-valor para guardar la cesta, sólo se tendría que modificar el servicio encargado de la cesta y no el de los artículos. Así la arquitectura tendría la forma que se muestra en la figura 1.4; donde ahora hay dos servicios que usan un mismo sistema de almacenamiento.

Cualquiera de los diseños anteriores es perfectamente valido, todo depende de si se tiene pensado reutilizar las consultas o se van a añadir o modificar los sistemas de bases de datos en el futuro.

Finalmente, una vez definida la estructura base de la arquitectura, un punto importante a tener en cuenta es la forma en la que se repartirán los datos. Como se comento antes, la idea de crear una aplicación de estas características es que se guarden los datos en los sistemas de almacenamiento que mejor se adecuen a los datos y a los usos que se van a hacer de ellos. Cada caso es diferente y cada sistema de almacenamiento dentro de un mismo tipo tiene una serie de características que le pueden hacer más o menos

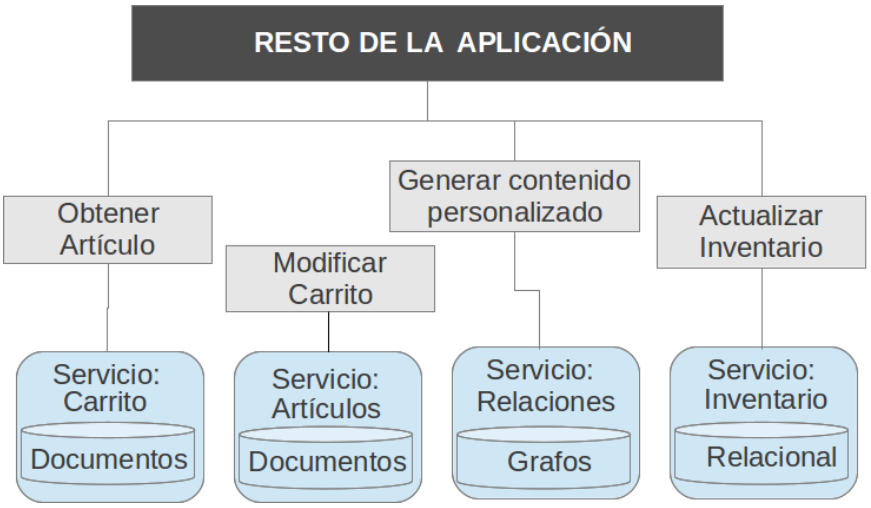

FIGURA 1.4: Arquitectura para una *Aplicación de Persistencia Políglota*: Diseño III

adecuado. Sin embargo, una primera distribución de los datos en función del tipo de almacenamiento se podría hacer teniendo en cuenta lo siguiente:

- Datos complejos: si los datos que se van a almacenar tienen una estructura compleja, de varios niveles y se quiere mantener una estructura similar para los objetos que estén en las bases de datos y los que esten en las aplicaciones. En este caso debido a su flexibilidad y al formato en que se guardan los datos, tanto los sistemas de Documentos como los de Familias de Columnas son adecuados para almacenar este tipo de datos.

- Datos altamente entrelazados: si los datos que se van a guardar tienen gran número de relaciones entre ellos, y se quiere conocer esta relación de forma precisa. En este caso los sistemas de grafos están mejor adaptados para representar datos donde prima conocer las relaciones entre ellos, y, además, son más eficientes que otros sistemas a la hora de realizar consultas en función de estas relaciones.

- Datos consistentes: si los datos necesitan ser consistentes en todo momento para todos los usuarios. En este caso algunos sistemas como los de grafos cumplen con los principios ACID para mantener la consistencia de los datos, sin embargo, otros sistemas como los relacionales presentan herramientas sencillas que permiten controlar las transacciones y los bloqueos.

- Datos modificados frecuentemente: si los datos van a ser modificados y leídos de forma frecuente, quieren ser accedidos en tiempo real; y cada acceso sólo modifica/lee una pequeña parte de los datos. Si estos no requieren de una estructura con muchos niveles de anidamiento - no más de tres -, la mejor opción son los sistemas de tipo clave-valor los cuales permiten acceder de forma muy eficiente a los datos.

1.3 Características Generales de los Sistemas NoSQL

En esta sección se va a intentar mostrar una idea básica del uso de las tecnologías NoSQL y sus posibles beneficios con respecto a otros sistemas de almacenamiento.

1.3.1 Antecedentes

Durante muchos años los sistemas de bases de datos relacionales han sido el sistema preferido para el almacenamiento de datos digitales. Sin embargo, si bien no es una tecnología nueva, en los últimos años la popularidad de los sistemas de bases de datos NoSQL ha crecido [3, 4] en gran medida.

Existen varios factores que han propiciado dicha popularidad. Uno de ellos ha sido la adopción y/o desarrollo de este tipo de sistemas por parte de grandes empresas como Google (BigTable), Amazon (SimpleDB) o Facebook (Cassandra), entre otras. Todo esto ha ayudado tanto a popularizar los sistemas de cara a los usuarios, como a inspirar el desarrollo de otros sistemas similares, generando con ello un gran número de opciones en el mercado. Otros factores han sido: el surgimiento de nuevos problemas derivados del aumento de los datos digitalizados, la variedad de los datos y del uso que se hace de ellos.

Con la constante aparición de nuevas tecnologías y el alto uso de internet, cada vez es más sencillo obtener y almacenar datos, lo cual ha provocado que aumente drásticamente la cantidad de datos disponibles [5] y es cada vez más común encontrar aplicaciones y servicios que utilicen grandes conjuntos de datos - Big Data. Ésto hace que se deban tener los recursos necesarios tanto para almacenar los datos como para poder procesarlos. Generalmente en los sistemas relacionales se tendía a mejorar el rendimiento - a través del escalonamiento - mejorando el hardware de los ordenadores; pero con un gran número de datos escalar el hardware resulta complicado y/o costoso con lo que se hace necesario repartir la carga de trabajo entre varios ordenadores. Si bien los sistemas relacionales son capaces de distribuir los datos, no están diseñados para ser distribuidos y resulta complejo realizar consultas entre tablas que estén distribuidas en varios sistemas [6].

Con el aumento del número de datos, también ha aumentado la complejidad y desestructuración de estos. Algunos datos como la información de una hoja de cálculo son simples y pueden ser representados fácilmente mediante las tablas de los esquemas relacionales. Otros datos más complejos que representan objetos que contienen objetos o datos altamente relacionados como la información sobre relaciones entre personas - social graphs - resultan complicados de representar en un sistema relacional. Si bien es posible

hacerlo el esquema resultante podría contener varias tablas relacionadas que requerirían de "JOIN"'s complejos para obtener los datos.

Además de la complejidad, cada vez más, objetos similares pueden tener características distintas y carecer de una estructura concreta, con lo que cada objeto puede tener un esquema diferente. Por esto, en ciertos casos se busca un sistema flexible que permita adaptar el esquema fácilmente a las distintas estructuras de los datos de forma dinámica.

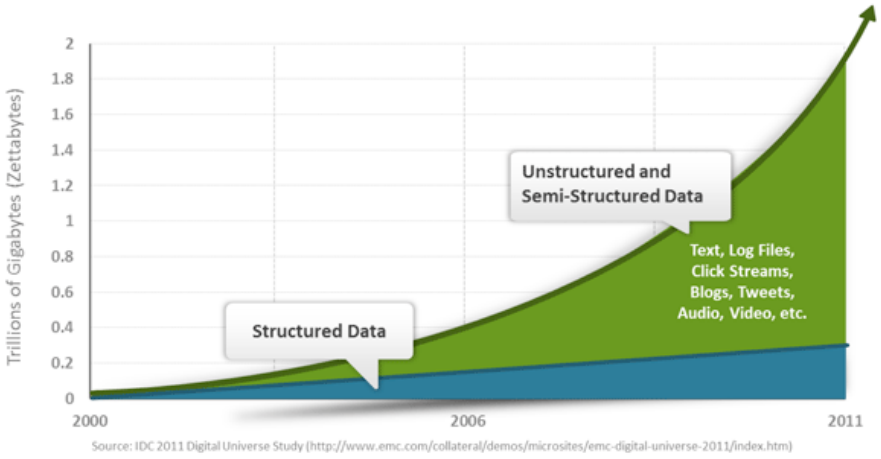

FIGURA 1.5: Datos estructurados vs no estructurados [1]

Estos problemas han hecho que se busque en los sistemas NoSQL una alternativa a los sistemas tradicionales de bases de datos relacionales los cuales no estaban específicamente adaptados a estos problemas.

1.3.2 Conceptos Básicos

Pese a su nombre las bases de datos NoSQL, no hacen referencia a bases de datos que no utilizan SQL, sino que el término se refiere a las bases de datos que no utilizan un modelos relacional para modelar los datos que estén almacenados en estas. Para los sistemas NoSQL este modelo no es único y se pueden distinguir varios tipos de sistemas dependiendo de cómo abordan este problema. Más adelante, [pag.15], se hablará de los principales tipos de sistemas.

Los sistemas de bases de datos NoSQL generalmente están diseñados pensando en que se usarán para el almacenamiento distribuido de un gran conjunto de datos; y con la idea de que sean fácilmente escalables mediante la distribución de sus datos entre distintos ordenadores o servidores - "clusters".

Una de las características principales, y ventaja frente a sistemas relacionales, es que estos sistemas manejan datos no estructurados como emails, archivos multimedia o grafos sociales de forma muy eficiente [6]. Esto se debe a que estos sistemas almacenan los datos sin esquemas - schemaless -. En teoría, según esta característica los datos almacenados no deben seguir ningún tipo de esquema prefijado. Si bien esto es cierto, en la realidad y debido en parte al uso que se hará de los datos, siempre habrá algún tipo de esquema prefijado el cual se ajuste a las necesidades de la aplicación que utiliza dichos datos. Sin embargo, es posible insertar nuevos objetos que tengan un esquema distinto al de los datos ya insertados, sin que se tengan que re-definir las estructuras - tablas, colecciones, grafos, familias de columnas... - que guarda dichos objetos.[2]

Un detalle a tener en cuenta es que, si bien en general las bases de datos no relacionales están orientadas a su uso en "clusters", algunas bases de datos como las de grafos utilizan un modelo distribuido similar al de las bases de datos relacionales con lo que se tendrían problemas similares al escalar sobre varios ordenadores o servidores. Sin embargo, el modelo de datos de estas bases de datos es mejor para manejar datos con relaciones complejas.[2] Así mismo, otros sistemas no relacionales más adaptados para su uso en "clusters" - con la intención de escalar los datos - pueden no resultar tan eficientes al tratar datos altamente relacionados [7].

Finalmente, un punto importante es que la mayoría de los sistemas no relacionales, no respetan los principios ACID - atomicidad, consistencia, aislamiento y durabilidad. Pese a que este principio parece imprescindible para llevar a cabo tareas con los datos almacenados, resultan incompatible con la alta disponibilidad y eficiencia en el uso de grandes conjuntos de datos que se espera de los sistemas NoSQL. Ésto se debe principalmente a los problemas planteados por el teorema de CAP de Brewer [8], y las soluciones adoptadas por los sistemas que dan prioridad a la disponibilidad y la eficiencia antes que a la consistencia.

Seguidamente se explica el Teorema de CAP y las soluciones planteadas a los problemas, entre ellas el principio BASE.

1.3.2.1 Teorema de CAP

En el 2000 Brewer dio una conferencia en la que planteaba los tres requerimientos principales de una aplicación distribuida y enunciaba el teorema de CAP.[8]

Estos tres requerimientos son:

1. Consistencia: todos los usuarios deben ver la misma versión de los datos aunque haya cambios en estos. Éstos cambios deben poder ser vistos por otros usuarios desde el mismo momento en el que se hagan.

2. Disponibilidad: el servicio debe estar disponible y el tiempo de respuesta del servicio debe ser bajo.

3. Tolerancia a particiones/fallos: todos los usuarios deberían ser capaces de acceder a todos los datos pese a que, en caso de tener un "cluster", alguno de los nodos quedase deshabilitado.

El teorema de CAP dice que no es posible mantener estos tres requerimientos con un gran volumen de datos y actividad. Con poca actividad o pocos datos es posible mantener la consistencia entre diferentes nodos de un sistema distribuido de forma rápida y sin que afecte al resto de comunicaciones.

En el caso de que haya gran cantidad de actividad, aumentará la cantidad de información que habrá que mantener consistente. Si esto ocurre cuando únicamente se tiene un nodo, el tiempo empleado por el sistema en mantener la consistencia será bajo y los datos estarán disponibles casi inmediatamente. Si se añaden muchos nodos para aumentar la tolerancia a fallos, con cada nuevo nodo, aumentará el tiempo empleado en mantener la consistencia, con lo que los datos tardarán más tiempo en estar disponibles.

Por tanto, no es posible tener consistencia inmediata, disponibilidad alta y gran tolerancia a fallos - un gran número de nodos con datos replicados - al mismo tiempo.

Existen varias opciones a la hora de tratar estos problemas [8] y en general las soluciones buscan limitar alguno de los tres requisitos :

- Eliminar la tolerancia a fallos: sin nodos el tiempo de espera a que los datos sean consistentes es mínimo.

- Eliminar la alta disponibilidad: permitiendo que haya retrasos en el tiempo de respuesta derivados del tiempo que se tarda en mantener la consistencia de los datos.

- Eliminar la consistencia: los datos serán consistentes en algún momento. Si algún usuario requiere algún dato, se responderá sin esperar a que los datos sean consistentes.

- BASE: es un modelo de consistencia que consta de tres principios [9]:

1. Basic Availability: este principio indica que los datos deben ser altamente disponibles en todo momento, por lo que el tiempo de respuesta debe ser bajo y el sistema debe ser tolerante a fallos.
2. Soft State: indica que el estado de los datos puede cambiar de repente debido al tercer principio.
3. Eventual Consistency: asegura que en algún momento los datos en todos los nodos serán consistentes con los cambios hechos. Sin embargo, la consistencia no tiene que ser inmediata.

1.3.2.2 Tipos de bases de datos NoSQL

Es posible dividir los tipos de bases de datos en función del modelo de datos que utilizan. Entre ellos existen cuatro modelos principales.

Documentales

Cada objeto esta representado por documentos. Estos documentos son estructuras de datos complejas formadas por un conjunto de campos y sus valores, los cuales pueden ser tanto datos simples, como subdocumentos - subestructuras de datos. Generalmente son buenas para almacenar representaciones des-normalizadas de un objeto como es el caso de los artículos de una tienda que pueden tener distintos tipos de productos.

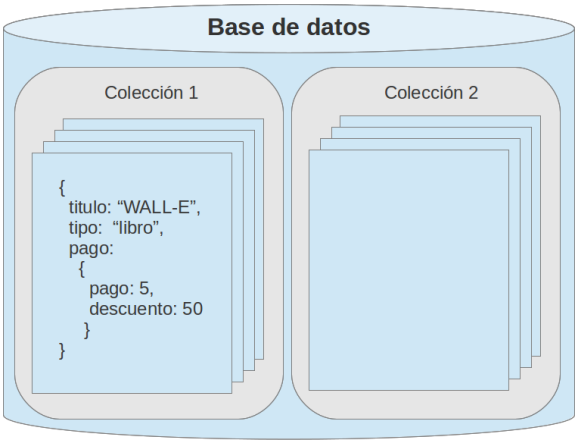

FIGURA 1.6: Sistema de Bases de Datos Basado en Documentos

Grafos

Generalmente se representan con unas estructuras de datos llamadas tripletas. Estas tripletas contienen tres valores los cuales representan a dos nodos y a la relación que existe entre ellos. Generalmente se usan para representar datos altamente relacionados como grafos sociales con información de las relaciones entre personas.

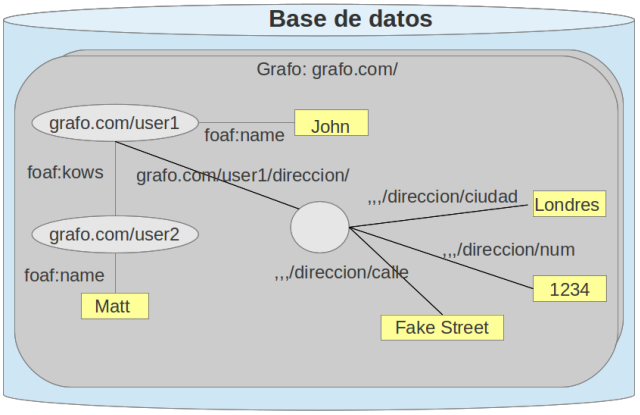

FIGURA 1.7: Sistema de Bases de Datos Basado en Grafos

Clave-Valor

Son las más simples, los elementos son guardados en la base de datos como una clave que será el nombre de un atributo y su valor. Cada objeto está representado por un conjunto de pares clave-valor. Generalmente son buenas obteniendo rápidamente el valor de una clave determinada. Por ejemplo, Amazon utiliza este tipo de base de datos para implementar la cesta de la compra.

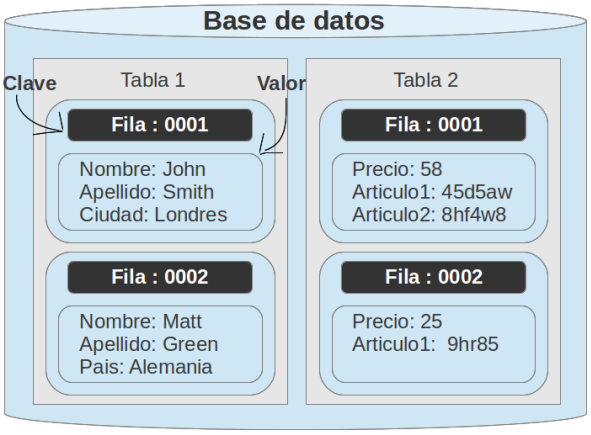

FIGURA 1.8: Sistema de Bases de Datos Basado en Clave-Valor

Familia de columnas

Estas bases de datos se pueden ver como una evolución de las bases de datos clave-valor, donde el valor puede ser un dato o un conjunto de columnas. En estas bases de datos, la estructura más básica es una columna que está formada por una clave y un valor simple. Después una estructura algo más compleja son las *supercolumnas* que contienen un conjunto de columnas. Por último, las familias de columnas son un conjunto de filas las cuales pueden estar formadas por un conjunto de columnas y/o *supercolumnas*. Estas últimas forman la estructura principal de las base datos, es decir una base de datos estará formada por un conjunto de columnas. Además como ocurre con las bases de datos de clave-valor, en general las búsquedas se realizan sobre las claves.

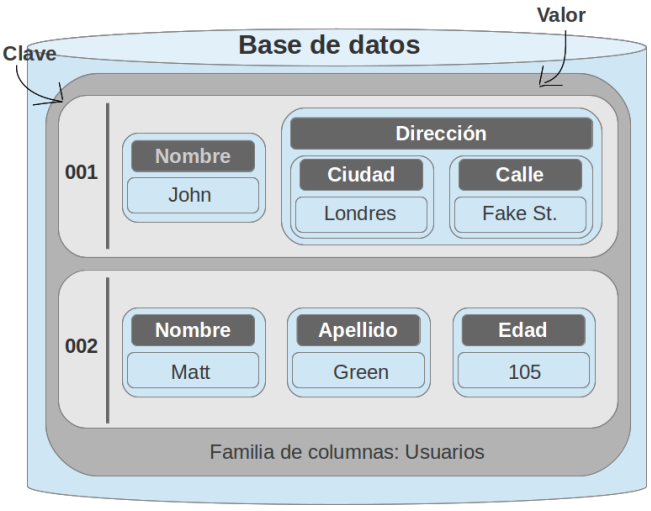

FIGURA 1.9: Sistema de Bases de Datos Basado en Familia de columnas

1.3.3 ¿Porqué NoSQL?

Como se ha comentado en los puntos anteriores, las bases de datos NoSQL tienen una cierta ventaja sobre las bases de datos relacionales cuando la cantidad de datos almacenados es grande, si los datos son muy complejos o si están altamente relacionados. Sin embargo, no todo son ventajas.

Por un lado, los sistemas no relacionales en general no cumplen el principio ACID y no permiten el uso de transacciones para controlar la concurrencia. Si bien algunos permiten ésto, éstos pueden tener problemas al escalar los datos entre varios ordenadores, tal y como ocurre con los sistemas relacionales.[6]

Por otro lado, las bases de datos no relacionales no están estandarizadas. A diferencia de los sistemas relacionales donde es posible aprender los conceptos básicos del modelo relacional y aplicarlo a los diferentes sistemas.[2]

Por tanto, a pesar de estas desventajas ¿Porqué NoSQL? La realidad es que la pregunta no debería ser porqué, sino cuándo.

Un sistema NoSQL no va a sustituir a uno relacional cuando se requiera una gran consistencia. Por ejemplo, en un banco al realizar transferencias se querrá que las operaciones sean consistentes; con lo que un sistema NoSQL podría generar problemas que uno relacional no generará. Por tanto, un sistema no relacional sería más adecuado aunque puediera resultar algo más lento.

Pero no siempre se quiere que la base de datos sea consistente. Por ejemplo, puede ser el caso de una aplicación que requiera la búsqueda de artículos en una página de noticias con gran número de noticias. En este caso es probable que se prefiera que las búsquedas sean más rápidas aunque no muestren el artículo que se creó hace un segundo. Si el número de artículos es lo suficientemente grande o si los datos son muy complejos, un sistema NoSQL sería más adecuado.

Por tanto, no es una cuestión de sistemas relacionales vs. sistemas NoSQL. Estos dos pueden coexistir, ya que cada uno resuelve distintos problemas [2] y por tanto serán adecuados para distintas tareas.

1.4 Planificación

En esta sección se explican los diferentes aspectos que se han tenido en cuenta a la hora de planificar el desarrollo del proyecto; así como el análisis de alcance mínimo y de la ampliación del alcance.

1.4.1 Alcance

En los siguientes apartados se explica el alcance mínimo que deberá tener el diseño del entorno trabajo y la aplicación a desarrollar; así como la ampliación del alcance que se plantea en caso de disponer de tiempo tras realizar el alcance mínimo y otros aspectos de la planificación relacionados con el alcance.

1.4.1.1 Alcance mínimo

En las siguientes secciones se especifica el alcance mínimo que deben cumplir cada uno de los objetivos mencionados anteriormente.

1.4.1.1.1 Aplicación de persistencia políglota

La idea de una aplicación que gestione una persistencia políglota es la de crear una API, la cual deberá permitir a los usuarios conectarse a tres sistemas de bases de datos – MongoDB, OpenLink Virtuoso, MySQL - y realizar operaciones sobre los mismos, de forma transparente, es decir, sin que tenga que preocuparse de las características específicas de cada uno de ellos.

Con la idea de acercase a un ejemplo del mundo real en el que una misma aplicación pudiera beneficiarse del uso de diferentes sistemas de bases de datos para realizar distintas tareas, se ha decidido implementar una aplicación en el dominio del comercio electrónico. La aplicación cuenta con los siguientes servicios de los que se encargarán los distintos sistemas:

- Bussiness Intelligent: este servicio permitirá mostrar anuncios personalizados a un cliente dado en función de las preferencias de otros clientes que tengan gustos similares. Para este servicio se utilizará OpenLink Virtuoso el cual usará datos rdf.
- Manejo de contenido: este servicio es el que mostrará los artículos y realizará las búsquedas solicitadas por los clientes. Para este servicio se utilizará MongoDB.
- Carro de la Compra: este servicio es el que permitirá tener guardados artículos temporalmente en la lista de la compra. Para este servicio también se utilizará MongoDB, aunque los datos solo serán guardados durante un tiempo y luego espirarán.
- Inventario: este servicio es el que se encargará de controlar la cantidad de artículos disponibles para evitar que un cliente compre algún artículo para el cual no existan unidades disponibles. Para este servicio se utilizará MySQL.

Además, puesto que se quiere hacer esto con una gran carga de datos, se creará un dataset artificial para cada sistema, cuyo modelo de datos se ajuste a las necesidades de cada uno de los servicios.

1.4.1.1.2 Diseño del entorno de trabajo

Se realizará un diseño del entorno de trabajo básico sobre dos sistemas de bases de datos NoSQL en particular: MongoDB y OpenLink Vistuoso.

Para cada sistema se deberán tratar como mínimo los siguientes aspectos:

- Instalación del sistema de base de datos.
- Modelado de datos e importación de estos.
- Alguna operacion de administración como realizar volcados y cargas de datos o configurar el sistema para mejorar el rendimiento.
- Diseño de Queries para realizar consultas en el lenguaje que utilice cada sistema, y pruebas para ver el rendimiento de los diferentes tipos de operaciones sobre cada base de datos.
- Estudio de las opciones de concurrencia que ofrece cada sistema.
- Diseño de índices para mejorar el rendimiento de las consultas.
- Gestión de seudotransacciones.

1.4.1.2 Ampliación del alcance

En caso de que fuera posible tras realizar el alcance mínimo se propone la realización de las siguientes ampliaciones:

- Diseño de un sistema distribuido en MongoDB que incluya: replicación, sharding y/o diseño de operaciones de tipo map reduce.
- Diseño de un sistema distribuido en OpenLink Virtuoso que incluya: replicación, sharding y/o diseño de operaciones de tipo mapreduce.

1.4.1.3 Criterios de aceptación

Se considerará que el proyecto cumple con los criterios de aceptación cuando el proyecto cumpla el alcance mínimo planteado anteriormente y haya sido revisado y aprobado por el director del proyecto.

1.4.1.4 Exclusiones del alcance

En lo que se refiere a las tareas de la ampliación del alcance se ha considerado aumentar el diseño del entorno para que incluya aspectos relacionados con el despliegue de bases de datos distribuidas; sin embargo, queda fuera del alcance el incluir en la aplicación que gestiona una persistencia políglota las operaciones relacionadas con dichos sistemas distribuidos. Así mismo, puesto que el objetivo principal de la aplicación a desarrollar es el estudio de las aplicaciones de persistencia políglota, se excluye del alcance el implementar servicios no relacionados con los indicados en el alcance mínimo, como pueden ser servicios de pagos reales.

1.4.2 Entregables del proyecto

La realización del proyecto incluye los siguientes entregables:

1. Memoria: documento que contenga información sobre lo realizado durante el proyecto.
2. Instancias de las base de datos sobre las que se permita realizar pruebas.
3. Documentación con el diseño del sistema, pasos que se han seguido y resultados de las distintas pruebas realizadas.
4. Intancia de la aplicación en funcionamiento.
5. Código fuente de la aplicación realizada.

1.4.3 Esquema de desglose de trabajo

Seguidamente se muestra el esquema de desglose de trabajo 1.10, en el cual se plasman los diferentes paquetes de tareas en los que se ha descompuesto el proyecto. Dicho esquema se ha dividido en cinco tareas generales – planificación, gestión, formación, desarrollo y documentación – las cuales a su vez se han dividido en tareas más concretas.

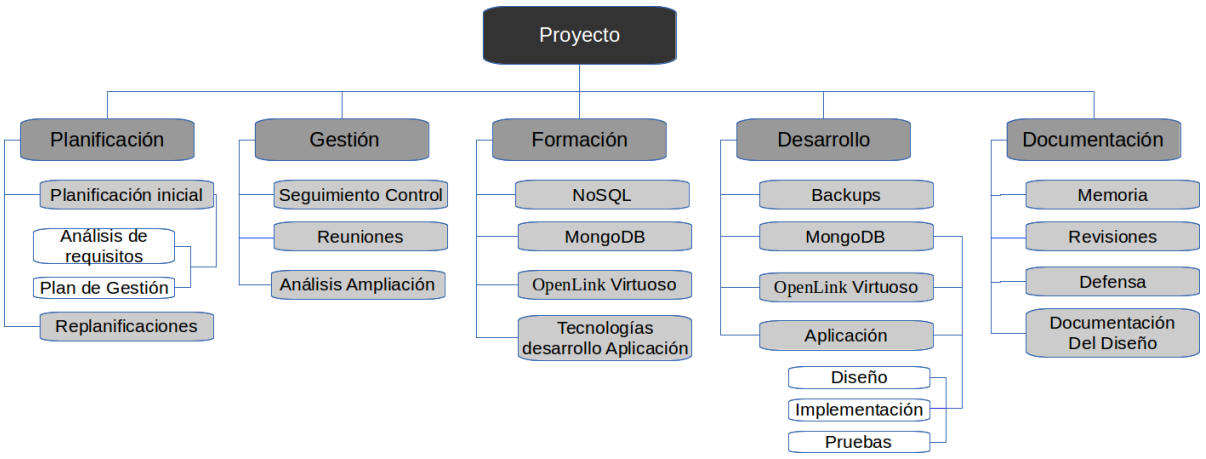

FIGURA 1.10: Esquema de desglose de trabajo

1.4.4 Planificación temporal

Seguidamente se muestra el tiempo necesario que se estima se tardará en realizar las diferentes tareas que componen el proyecto, así como la lista de hitos más importantes y las fechas en las que se prevé que se llevarán a cabo. Por último, también se incluye un cronograma con las fechas durante las cuales se planea realizar cada tarea.

1.4.4.1 Actividades

Seguidamente se muestran las tablas con la información de las horas que se planea invertir en cada una de las tareas a realizar.

La tabla 1.1 muestra la planificación del paquete de tareas "Planificación" el cual recoge tareas relacionadas con la planificación del desarrollo del proyecto.

Planificación	Tiempo estimado
Planificación inicial	16h
Análisis de requisistos	8h
Plan de gestión	8h
Replanificaciones	7h
Total	**23h**

TABLA 1.1: Tiempo de planificación estimado

La tabla 1.2, correspondiente al paquete de gestión, muestra la estimación del tiempo que se tardará en realizar las tareas relacionadas con el control del progreso del proyecto.

Gestión	Tiempo estimado
Análisis de la ampliación	3h
Seguimiento y Control	22h
Reuniones	10h
Total	**35h**

Tabla 1.2: Tiempo de getión estimado

La tabla 1.3 muestra información sobre el tiempo que se estima que el desarrollador tardará en formarse en las diferentes tecnologías a usar de las que no se tenga previo conocimiento.

Concretamente, el punto "Formación para realizar la aplicación" se refiere a la formación sobre aplicaciones de persistencia políglota; y en menor medida a la formación en tecnologías para implementar la aplicación.

Esto último es debido a que se prevé que la formación en herramientas para el desarrollo de la capa de presentación de la aplicación será mínima, puesto que se ha tenido contacto con el desarrollo de páginas web en varias asignaturas.

Por otro lado, así mismo ocurre con el lenguaje – java - elegido para desarrollar el resto de la aplicación, del cual se ha tenido un extenso contacto durante los pasados años, en especial en la asignatura de programación concurrente e ingeniería del software.

Formación	Tiempo estimado
Formación en NoSQL general	10h
Formación en MongoDB básico	30h
Formación en OpenLink Virtuoso básico	30h
Formación para realizar la Aplicación	10h
Total	**80h**

Tabla 1.3: Tiempo de formación estimado

La tabla 1.4 recoge información sobre el tiempo que se estima se tardará en desarrollar el diseño y la aplicación con las especificaciones indicadas en el alcance mínimo. Además

por cada punto, se especifica el tiempo que se tardará en diseñar, implementar y realizar las pruebas. Por otro lado, en esta tabla también se incluye el tiempo que se empleará en realizar los backups de las bases de datos para evitar la pérdida del trabajo en caso de problemas.

Desarrollo	Tiempo estimado
Backups de las BD	5h
Desarrollo MongoDB	30h
Diseño del entorno MongoDB	15h
Implementación de operaciones MongoDB	10h
Pruebas de rendimiento MongoDB	5h
Desarrollo OpenLink Virtuoso	30h
Diseño del entorno OpenLink Virtuoso	15h
Implementación de operaciones OpenLink Virtuoso	10h
Pruebas de rendimiento OpenLink Virtuoso	5h
Desarrollo de la aplicación	115h
Diseño	25h
Implementación	65h
Pruebas	15h
Total	**170h**

Tabla 1.4: Tiempo de desarrollo estimado

Finalmente, la tabla 1.5 muestra el tiempo que se estima se tardará en realizar las tareas de documentación, tanto de la memoria del proyecto, como de la documentación del diseño del entorno de trabajo sobre las diferentes bases de datos.

Documentación	Tiempo estimado
Memoria	40h
Revisiones	10h
Defensa	10h
Documentación del diseño	25h
Total	**85h**

Tabla 1.5: Tiempo de documentación estimado

El tiempo total que se estima se tardará en realizar el proyecto para el alcance mínimo es de 393 horas.

1.4.4.2 Actividades de la ampliación

Seguidamente se muestra en la tabla 1.6 una estimación de lo que se cree debería tardase en realizar la ampliación del alcance. Hay que tener en cuenta que este tiempo viene condicionado por el tiempo que se prevé que habrá disponible tras realizar el alcance mínimo. Por tanto será necesario durante el análisis de ampliación obtener más información sobre las tareas que se deben realizar y recalcular los tiempos, para ver si la ampliación es viable con el tiempo disponible real.

Ampliación	Tiempo estimado
Formación Amazon EC2	10h
MongoDB	55h
Replicación	20h
Sharding	20h
Mapreduce	10h
Documentación	5h
Openlink Virtuoso	55h
Replicación	20h
Sharding	20h
Mapreduce	10h
Documentación	5h

Tabla 1.6: Estimación del tiempo de la ampliación

1.4.4.3 Hitos

Seguidamente se muestra la tabla 1.7 con los hitos más importantes junto a la fecha en la que se deberían dar.

Hitos	Fecha del Hito
Planificación inicial	15/07/2013
Diseño MongoDB básico	29/07/2013
Análisis de la viabilidad del proyecto	29/07/2013
Diseño OpenLink Virtuoso básico	22/08/2013
Prototipo de la aplicación	16/09/2013
Finalización de la aplicación	30/09/2013
Análisis de ampliación	30/09/2013
Borrador de la memoria	18/11/2013
Finalización de los entregables	02/12/2013
Finalización del proyecto	15/01/2014

TABLA 1.7: Fecha de Hitos

1.4.5 Cronograma

Seguidamente se muestra un cronograma 1.11 el cual especifica las fechas durante las cuales se espera realizar cada una de las tareas indicadas en los apartados anteriores. Hay que destacar, que en el cronograma entre las fechas del 17 de septiembre y el 4 de noviembre únicamente se realizan tareas de seguimiento y control, y la revisión del documento de los diseños de desarrollo. Esto se debe a que se han dejado esas fechas libres, por un lado, con la intención de que sirvan de holgura en caso de que haya problemas durante la realización de las tareas del alcance mínimo; y por otro lado para que en caso de decidir realizar la ampliación del alcance, tener tiempo extra para que su realización sea posible.

1.4.6 Costes

Durante la realización del alcance mínimo se planea realizar las tareas que conforman el proyecto con herramientas de libre uso. En concreto, para los sistemas de bases de datos utilizadas se planea utilizar las versiones de libre uso y no las versiones de pago de estos sistemas. Por tanto el coste estimado del alcance mínimo es de cero euros. Por otro lado, en caso de realizar las tareas presentes en la ampliación del alcance, se ha pensado en usar servicios similares a Amazon EC2 para poder desplegar el sistema distribuido. Puesto que estos servicios son normalmente de pago, esto podría conllevar un cierto gasto. En cualquier caso, el coste de dicho servicio se calculará al realizar el análisis de ampliación para ver si la realización de las tareas de ampliación es viable.

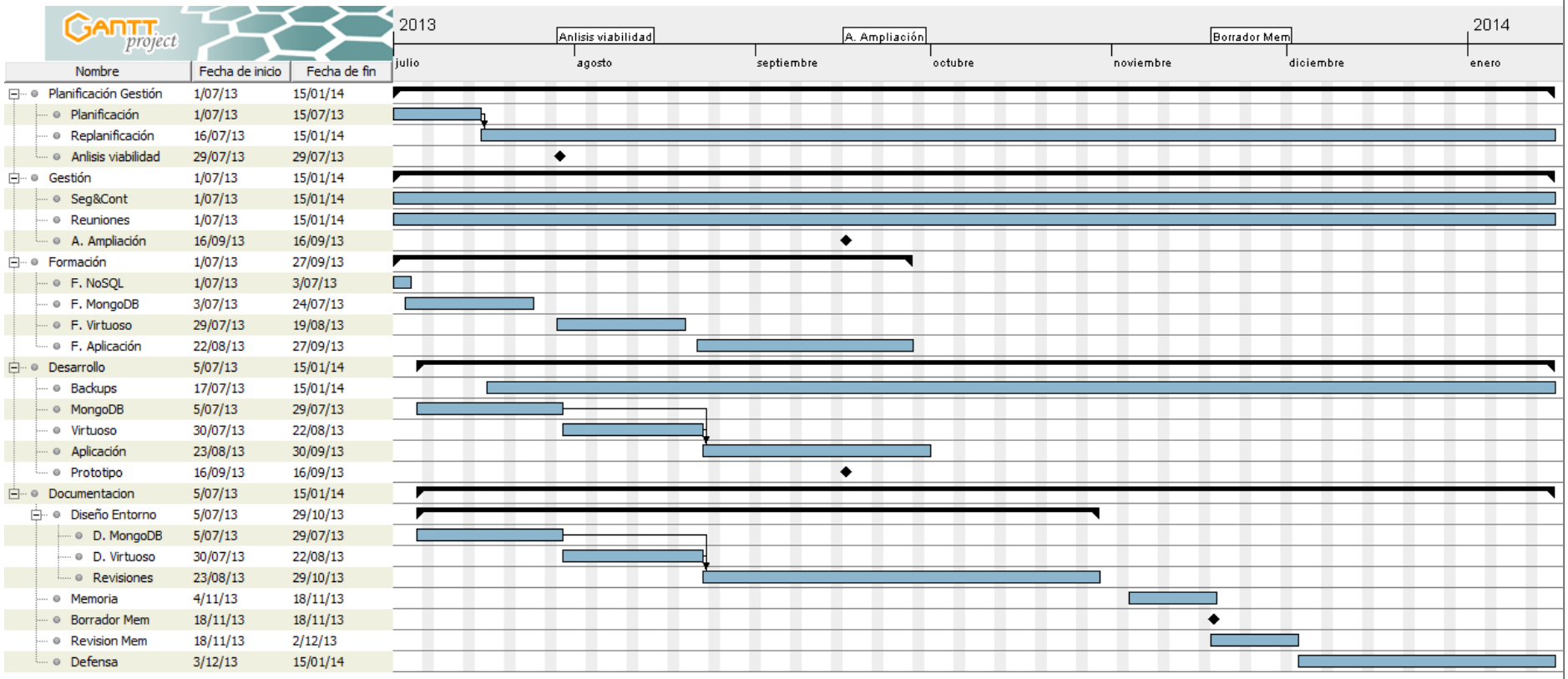

FIGURA 1.11: Cronograma

1.4.7 Análisis de calidad

Para mantener la calidad a lo largo del proyecto una de las principales tareas que se ha programado es la de mantener reuniones con el director del proyecto una vez cada dos semanas para comprobar que lo realizado durante el proyecto se adecua a lo esperado inicialmente. En las siguientes secciones se indica una planificación más concreta para mantener la calidad por un lado del proyecto y por otro la calidad del producto.

1.4.7.1 Calidad del proyecto

Para mantener la calidad en lo referente al proyecto se han planeado las siguientes actuaciones:

- Para asegurar que el proyecto se lleva a cabo según lo indicado en la planificación, habrá un día a la semana en el que se analizarán las tareas realizadas durante la semana y se replanificará si fuese necesario. Todo esto como parte de las tareas del seguimiento y control.
- Se ha establecido un política de replicación de documentos en varios sistemas de almacenamiento tanto online como offline, lo cual permitirá que en caso de perder los datos en una de las bases de datos, esto no provocará la pérdida de más de un día de trabajo en el peor de los casos.

1.4.7.2 Calidad de los entregables

En lo referente a los entregables se han planificado las siguientes medidas:

- En lo que se refiere a la documentación del diseño, se ha planificado la entrega al terminar dicho diseño y se ha planificado tiempo de holgura para poder realizar cambios en la documentación si fuera necesario.
- En lo que se refiere a la aplicación, se ha planificado la presentación al director del proyecto de un prototipo para comprobar que cumple con los requisitos solicitados.
- Para la calidad de la memoria del proyecto se ha incluido en la planificación una tarea de revisión y corrección, la cual al igual que en el caso anterior se ha planeado con un cierto margen a la entrega final de los entregables.

1.4.8 Comunicaciones

Para mantener una comunicación con los diferentes interesados del proyecto se han diseñado la política de comunicación que se muestra en los siguientes apartados en los que se detalla quienes son los interesados, el método de comunicación y reuniones a llevar a cabo.

1.4.8.1 Política de comunicación

Principalmente, los interesados de este proyecto son:

- Directora: Arantza Illarramendi Echave.
- Desarrollador: Estefanía Gutiérrez.

La comunicación entre los dos interesados anteriores se realizará mediante el envío de correos electrónicos, o en casos de urgencia mediante comunicación telefónica o en persona.

1.4.8.2 Reuniones

Para ayudar a mantener la calidad del proyecto y como parte de las tareas de seguimiento y control, se ha planeado celebrar reuniones entre los dos interesados con un intervalo de tiempo entre dos semanas y una semana y media.

1.4.9 Gestión de riesgos

A continuación se lista los riegos que se han identificado, los cuales podrían afectar al desarrollo e integridad del proyecto; así como las soluciones que se plantean en caso de que se dieran dichos riesgos.

No cumplimiento de los plazos de entrega

Gravedad: Muy alta

Probabilidad: Baja

Medidas preventivas: principalmente se han planificado con tiempo las fechas entre las que se planea realizar las tareas del alcance mínimo y la fecha de entrega de los entregables.

Medidas correctoras: no se han planificados fines de semana como tiempo de trabajo por lo que sería posible añadir dichas fechas como tiempo de trabajo si resultase necesario; así como ampliar las horas de trabajo diarias.

Retrasos

Gravedad: media

Probabilidad: alta

Medidas preventivas: se han planificado las fechas con un cierto margen de holgura.

Medidas correctoras: aumentar el número de horas diario que se estaba empleando en ese momento.

Planificación inadecuada

Gravedad: media

Probabilidad: alta

Medidas preventivas: se ha planificado un tiempo de replanificación, así como un hito durante el alcance mínimo para replanificar el proyecto en caso de que no se haya realizado adecuadamente la fase anterior.

Requisitos recogidos inadecuadamente

Gravedad: alta

Probabilidad: media-alta

Medidas preventivas: se han planificado reuniones para comprobar que lo realizado es acorde con lo establecido inicialmente por el director.

Medidas correctoras: replanificar.

Perdida de información y trabajo realizado

Gravedad: alta

Probabilidad: baja

Medidas preventivas: se ha planificado una política de backups cada cierto tiempo, así como la replicación de los documentos en varios sistemas de almacenamiento.

Pérdida de las herramientas de trabajo

Gravedad: alta

Probabilidad: baja

Medidas preventivas: se estará trabajando desde la universidad, por lo que en caso de pérdida de alguna de las herramientas se puede recurrir a alguno de los ordenadores disponibles. En lo referente a los datos que se puedan perder, al igual que en los casos anteriores este problema se solucionaría con la replicación y backup de los datos.

Enfermedad

Gravedad: baja

Probabilidad: baja

Medidas preventivas: el entorno de trabajo desde el que se realiza el proyecto permitirá realizar tareas desde el domicilio del desarrollador. Esto solo soluciona el problema en caso de enfermedad leve. Para otros casos véase la solución propuesta en el apartado de retrasos.

Capítulo 2

Entorno de Trabajo bajo MongoDB y OpenLink Virtuoso

La *Aplicación de Persistencia Políglota* de este proyecto utilizará tres sistemas de almacenamiento. En concreto se han utilizado dos sistemas NoSQL y uno relacional. En lo que se refiere a los sistemas NoSQL se han usado de dos tipos: (1) un sistema de documentos y (2) un sistema de grafos. En concreto las tecnologías de almacenamiento usadas han sido: MongoDB para documentos, Openlink Virtuoso para grafos, y MySQL como sistema relacional. En este capítulo se van a dar información sobre las características principales y el funcionamiento de los sistemas MongoDB y OpenLink Virtuoso.

2.1 MongoDB

MongoDB [13] es un sistema de bases de datos basado en documentos que ofrece alta eficiencia. Este sistema guarda los datos en documentos que internamente son objetos *BSON*, los cuales son muy similares a los objetos *JSON*. Éstos documentos se guardan dentro de colecciones formando algo similar a una tabla en un sistema relacional. A su vez un conjunto de colecciones forman una base de datos.

Una de las características presentes en los documento es que permiten diseñar esquemas muy complejos. En estos esquemas es posible tener subestructuras llamadas subdocumentos que se asignan como valor de uno de los campos de un documento. Con esto se pueden representar objetos que tengan varios niveles de anidación. Por otro lado, al igual que ocurre con otros sistemas como los relacionales, es posible guardar una gran variedad de tipos de datos como: enteros, reales, fechas... Pero destaca el hecho de que

es posible guardar listas de elementos en un sólo campo, incluyendo listas de subdocumentos. Otra característica interesante de los documentos es que los documentos de una colección puede tener diferentes esquemas unos de otros. Todo esto permite modelar los datos de forma similar a como se guardarían en los objetos de una aplicación con lo que se consigue que haya una cierta homogeneidad entre ambos.

En lo que se refiere a la estructura de este tipo de sistemas, en cierto modo se puede considerar que la estructura de las bases de datos es similar a la que se puede encontrar en Sistemas de Bases de Datos Relacionales (SBDR). El sistema esta formado por un servidor el cual gestiona las bases de datos que guardan los datos; y realiza operaciones sobre los datos en función de las consultas que los usuarios le soliciten. Los datos, al igual que ocurre con los sistemas SBDR, se organizan en bases de datos creadas por los usuarios, y en éstas se guardan los datos con el "esquema" deseado. Dentro de estas bases de datos se encuentran las colecciones que se podrían considerar son semejantes a las tablas de las bases de datos relacionales.

Dentro de estas colecciones se encuentran los llamados documentos. Estos documentos equivaldrían a las tuplas de las bases de datos, principalmente porque ambos contienen campos con valores en ellos. Sin embargo, la estructura es completamente diferente. Mientras que en las tablas los campos son fijos para todas las tuplas y estos contienen datos simples, en las colecciones puede haber un documento con campos que otro no tiene, y los campos pueden contener subdocumentos con sus propios campos. Esto hace que el esquema de un objeto complejo que puede tener distintas propiedades en función del tipo de objeto y que en una base de datos relacional requeriría múltiples tablas relacionadas entre ellas, en MongoDB se pueda solucionar con una única colección donde los documentos pueden tener distintos esquemas y se pueden ajustar a las propiedades que tenga cada objeto.

Por otro lado, como es común en muchos sistemas NoSQL, MongoDB no soporta el uso de JOINS. Sin embargo, soporta el uso de índices y el uso de instrucciones atómicas a nivel del documento. Tampoco soporta SQL ni lenguajes similares como SPARQL, en su lugar monogDB implementa una serie de métodos que permiten realizar operaciones - consultas - sobre los documentos, colecciones y las bases de datos, así como sobre el mismo servidor. Además incluye un entorno de trabajo en JavaScript el cual permite valerse de las características de dicho lenguaje para facilitar la realización de las consultas y operaciones que se hagan sobre la base de datos.

En lo que se refiere al sistema en si, las principales características que presenta son:

- Índices: es posible crear índices sobre varios campos tanto de los documentos como de los subdocumentos y, también, es posible indexar los campos de tipo array.

La única restricción es que no permite indexar dos campos de tipo array sobre el mismo índice.

- Consultas: no es posible utilizar lenguajes como *SQL* para realizar las consultas, sino que MongoDB ofrece una serie de instrucciones propias que permiten realizar las consultas sobre las bases de datos.
- Agregación: además de las instrucciones anteriores, MongoDB también ofrece una serie de instrucciones y operadores, equivalentes en cierto modo a las funciones de agregación de *SQL*, las cuales permiten realizar modificaciones sobre los resultados antes de devolverlos.
- Map-Reduce: es posible realizar una serie de consultas especiales que permiten condensar grandes cantidades de datos en resultados agregados.

Además, MongoDB ofrece una serie de herramientas entre las que se encuentra: (1) un cliente -mongo shell - desde el cual se pueden realizar las consultas y que soporta un lenguaje similar a *JavaScript* para facilitar la realización de operaciones; (2) herramientas para realizar volcados de los datos y para recuperar los datos - mongodump/mongoexport y mongorestore/mongoimport; y (3) una serie de herramientas de diagnóstico - como mongostat.

Entre las limitaciones de MongoDB hay que destacar que, aunque ofrece herramientas para realizar operaciones atómicas, MongoDB no soporta el uso de transacciones de forma nativa y no cumple con los principios ACID.

Por último, para realizar consultas a nivel de aplicación MongoDB ofrece drivers de forma oficial para una gran cantidad de lenguajes como Java, Python, C/C++... Este driver permite conectarse al sistema de almacenamiento y realizar consultas sobre ellos. Una característica de este driver es que a diferencia de los drivers *JDBC*, los datos obtenidos y enviados son guardados en unos objetos llamados *DBObject* con una estructura similar a los objetos *JSON*. Esto último permite usar librerías como *GSON* para pasar los datos entre objetos de la aplicación y de la base de datos de forma sencilla [pag.181].

2.1.1 Herramientas

La filosofía de MongoDB es la de ser una base de datos de documentos básica sin añadir herramientas que se encarguen de hacer tareas que otras herramientas hacen mejor, por ejemplo, en el caso del manejo de memoria MongoDB considera que es mejor que el sistema operativo se encargue de su manejo. Por tanto, todas las herramientas que provee este sistema están orientadas a esta tarea.

Las principales herramientas son las siguientes:

- mongod: es la instacia principal del sistema y es el que se encarga de procesar las consultas y realizar tareas sobre los datos. En general esta instancia guarda los datos en el directorio "/data/db".
- mongo: es la interfaz que MongoDB incluye en el paquete de instalación para que el usuario pueda comunicarse con la instancia del servidor. Este interfaz es una herramienta de trabajo basada en JavaScript y que permite realizar operaciones y consultas sobre los datos, así como administrar las bases de datos, utilizando para ello una serie de métodos ya implementados. Además permite al usuario valerse del lenguaje JavaScript para facilitar la realización de las diferentes operaciones.
- mongorestore/mongodump: estas dos herramientas permiten por un lado realizar volcados de los datos que hay en el sistema a nivel tanto de base de datos como de colección; y por otro recuperar los datos volcados con anterioridad.
- Driver: existen drivers oficiales para diversos lenguajes de programción como C++, Java, Python, Ruby, Node.js... Pero además existen drivers mantenidos por la comunidad de usuarios para los lenguajes no soportados de forma oficial.

2.1.2 Formato de los datos

En MongoDB, como se ha dicho anteriormente, los datos se guardan en documentos. Estos documentos son guardados en el sistema con un formato llamado BSON el cual es similar al formato JSON, y en cierto modo se puede considerar BSON como una representación binaria de un documento JSON.

En general, la estructura para guardar los datos es la mencionada anteriormente- Sin embargo, y puesto que los documentos BSON tienen un tamaño máximo de 16MB, en ocasiones se utiliza una especificación llamada GridFS [14] cuando se sabe que los documentos van a crecer más alla de los 16MB. Esta especificación consiste en que en lugar de guardar un documento en un único archivo, se divide el documento en varios trozos que se guardan como diferentes archivos. Al realizar una consulta, estos trozos son únidos por el sistema o el driver, con lo que la partición del documento es transparente para el usuario.

2.1.3 ¿Porqué MongoDB?

En cuanto a porqué utilizar un sistemas de documentos como éste en lugar de un sistema relacional tradicional; la respuesta es que en los casos en los que se utiliza este sistema

- como para representar los artículo de la tienda - los datos tienen una estructura compleja y pueden existir objetos de varios tipos que son similares pero que tienen algunas propiedades diferentes. Este tipo de sistemas están más adaptados a tratar estos problemas que un sistema relacional. Además, al mismo tiempo los datos que guardan estas estructuras no necesitan mantener una gran consistencia, con lo que no se beneficiarían de una de las principales ventaja de los sistemas relacionales.

Por otro lado, existen otros tipos de sistemas NoSQL que también pueden ser capaces de representar estructuras con un cierto grado de complejidad como es el caso de los sistemas de familias de columnas, estás están más limitadas y resulta difícil manejar estructuras de gran complejidad. Además, en muchos casos las familias de columnas sólo permiten realizar búsquedas en función de la clave y no del valor.

Finalmente, existen varios sistemas de bases de datos basadas en documentos; entre ellas se eligió MongoDB por dos motivos principales: (1) por su extensa documentación y alta popularidad [15]; y (2) porque la mayoría de los otros sistemas son relativamente recientes y podrían dar problemas de estabilidad.

2.1.4 Modelado de datos

En esta sección se va ha hablar de la forma general que tienen los llamados "documentos"; y de las diferentes formas en las que se pueden modelar los datos que se quieren almacenar en el sistema.

2.1.4.1 Documentos

Los datos en MongoDB se guardan en documentos dentro de colecciones. Estos documentos contienen campos con valores, tal y como ocurre con las tuplas de las tablas de las bases de datos relacionales. La mayor diferencia entre las tuplas y los documentos es por un lado que los documentos tienen un esquema flexible lo cual permite que documentos en una misma colección contengan una estructura diferente y que esta estructura pueda cambiar dinámicamente. Por ejemplo, si se tiene un usuario que puede tener un nombre o un alias, en una base de datos relacional habría que crear una tabla con los tres campos o dos tablas diferentes; sin embargo, con los documentos es posible tener documentos que contengan el campo "nombre" pero no el campo "alias", y otros documentos que contengan sólo el alias; todo ello en una misma colección - tabla.

En cuanto a la estructura general de los documentos en MongoDB, ésta es similar a la de una estructura de datos JSON, formada por campos que representan las propiedades de un objeto - documento - y los valores de cada objeto. Por ejemplo, un documento que

representará a un usuario con las propiedades nombre, email y edad tendría la siguiente forma:

```
{ _id: 87966468,
  nombre: "Jonh",
  email: "user1@email.com",
  edad: 47
}
```

Como se puede ver, un documento se representa con un conjunto de datos, separados entre ellos por comas y recogidos entre corchetes. Entre los campos además de las propiedades del usuario se incluye el campo "_id", esto es especifico de MongoDB, dicho campo debe aparecer en todos los documentos y su valor debe ser único. Por defecto, el valor de ese campo se genera automáticamente pero es posible asignarle un valor personalizado siempre que este sea único.

Otra características de los documentos, es que los valores de los campos pueden no sólo ser valores simples como strings o números, sino también pueden contener array, subdocumentos e incluso arrays de subdocumentos. Por ejemplo, siguiendo el ejemplo anterior, si se quisiera incluir la dirección de contacto junto con los datos de los usuarios, y que en esta dirección constara la provincia, la ciudad y la calle, se podría crear un documento con la siguiente forma:

```
{ _id: 87966468,
  nombre: "Jonh",
  email: "user1@email.com",
  edad: 47,
  direccion:
        { provincia: "Guipuzcoa",
          ciudad: "Irun",
          calle: "Andrearriaga"
        }
}
```

Como se puede ver la información que contien el campo "direccion" también tiene la forma de un documento, es decir, es un documento dentro de un documento. Si bien, este esquema es perfectamente valido, también sería posible guardar la dirección en un colección a parte y guardar en el campo "dirección" una referencia al documento.

En lo que se refiere a los array, estos se representan mediante una lista de valores - tanto simples, como subdocumentos o arrays - recogidos entre "[]". Por ejemplo, si el usuario tuviese varios emails:

```
{ _id: 87966468,
  nombre: "Jonh",
```

```
  email: ["user1@email.com","user1@gmail.com"],
  edad: 47
}
```

2.1.4.1.1 Restricciones de los documentos en MongoDB

Los documentos en MongoDB tiene una serie de restricciones:

1. Los documentos en MongoDB v.2.4 son documentos BSON y tiene un tamaño máximo de 16MB, si se va a exceder dicho tamaño – por ejemplo, para guardar archivos multimedia – entonces seria conveniente usar el sistema de archivos GridFS. Este sistema es a nivel de colección con lo que no es necesario que el resto de colecciones dentro de esa misma base de datos utilicen dicho sistema.

2. Todos los documentos deben contener el campo "_id", este debe ser único para cada elemento de la colección, y si no se especifica dicho campo al insertar un documento, entonces MongoDB añade automáticamente el campo y genera un identificador único. Dicho campo ocupa 2bytes y, como se ha comentado anteriormente, los documentos tienen un tamaño máximo, por tanto si se quiere ahorrar espacio es conveniente reutilizar el campo para guardar valores de otro campo, siempre que este sea único. Por ejemplo, si el título de los artículos fuera único, se podría guardar dicho en "_id" el título del artículo y no utilizar el campo "titulo".

3. Los nombres de los campos no pueden empezar por "$" - ya que se utiliza como identificador para los operadores como "$or" -; tampoco pueden contener puntos.

2.1.4.2 Diseño

Una de las características de las bases de datos basadas en documentos es que la forma en la que se guardan los datos no sólo depende de los datos en si, sino también de la forma en la que los datos van a ser usados por las aplicaciones. Esto hace que algunos esquemas de los documeto sean más adecuados para bases de datos que reciben muchas consultas de lectura y otros para las que la mayoría de las consultas son de escritura. Además hay que tener presente que diseños que agrupan información en un documento con subdocumentos o listas de subdocumentos, sean más apropiados cuando esta información va a ser accedida a menudo de forma conjunta.

Un ejemplo de esto último sería tener una base de datos que contenga información de libros (título, autor, editorial) e información de los autores (nombre, edad). Para dividir esta información en la base de datos se podrían crear dos colecciones una con los libros y

otra con las editores, relacionándolas con los identificadores de los autores. Sin embargo, si se hicieran consultas de forma regular en las que se solicita tanto el libro como los datos de los autores de ese libro se podrían incluir los datos de los autores en el mismo documento en el que se incluyen los datos del libro. Por ejemplo:

```
{ _id: 454,
  Titulo :nombre del libro,
  editorial: nombre de la editorial,
  autores:{
   [ {nombre: autor1, edad: 65},
         {nombre: autor2, edad: 49}
       ]}
}
```

Uno de los problemas con este tipo de esquemas en MongoDB se da porque hay un límite con el tamaño máximo de los documentos de tipo BSON (16MB) y si se ponen muchos datos en un mismo documento puede darse que éste crezca más allá del tamaño permitido. Sin embargo, MongoDB provee de una especificación para guardar documentos de mayor tamaño llamada gridFS [14].

Otros aspectos a tener en cuenta a la hora de modelar los datos es que los diferentes diseños pueden dar lugar a tamaños distintos de bases de datos. Por ejemplo, en el caso de tener muchas colecciones puede ocurrir que se necesiten varios índices por colección, y en este caso por cada índice se necesitan 18bytes. Como los índices son a nivel de colección el tamaño de la base de datos estaría aumentando en 18bytes multiplicados por el número de colecciones. Por otro lado, con colecciones de muchos documentos, debido a que todos los documentos tienen que contener el campo "_id" que tiene un tamaño de 12 bytes, el tamaño de la base de datos aumentará con cada nuevo documento. Esto se puede paliar en cierta medida reutilizando dicho campo con algún otro parámetro. Por ejemplo, se podría utilizar el título de un libro, si éste fuera único, por desgracia no es posible usar más de un campo como valor del campo "_id".

2.1.4.3 Patrones de diseño

Existen diversos patrones que intentan simplificar la creación de los diseño. Entre todos ellos [16] los que destacan son:

- Modelo de documentos embebidos uno a uno: este diseño hace referencia a los casos en los que se incluye un subdocumento en uno o varios de los campos.

- Modelo de documentos embebidos uno a muchos: este diseño hace referencia a los casos en los que se incluya una lista de subdocumentos dentro de un campo del documento.

- Modelo de documentos relacionados uno a muchos: en este caso en lugar de incluir los subdocumentos en un campo se tiene una lista de referencias a otros documento. Generalmente esto se usa cuando no se van a solicitar los datos reverenciados junto con los del documento con mucha frecuencia o bien cuando la estructura del documento reverenciado es muy compleja.

- Modelo de datos para operaciones atómicas: guardar los campos con los que se van a realizar operaciones atómicas en los documentos que contienen la información que se relaciona con este campo. Por ejemplo, si se tiene un artículo, se quería guardar la información de la disponibilidad de este documento en el mismo documento para que estén sincronizados. Esto se debe a que si no están en el mismo documento, entre el momento en el que se realiza la consulta de la disponibilidad y el momento en que se realiza la consulta para obtener los datos; la disponibilidad de los datos puede haber cambiado.

- Modelos en forma de árbol: estos modelos son usados principalmente para representar modelos de datos jerárquicos.

2.1.4.4 Diseño de un esquema de pruebas

En esta sección se explica como se han modelado los datos del dataset utilizado a lo largo de la documentación para realizar los ejemplos y las pruebas.

Este dataset contendrá información sobre "elementos" como libros o películas. Por cada artículo se guardará información en campos que serán comunes para todos los artículos como puede ser el título; y campos con información propia de cada tipo de artículo como es la lista de autores para los libros, o la lista de actores para las películas. Además cada artículo puede tener varios formatos, los cuales variarán en función de si son libros o películas.

Teniendo en cuenta las características y restricciones que pueden tener los documentos comentadas con anterioridad, se han diseñado dos esquemas diferentes.

Seguidamente se muestra cada uno de estos diseños, y más adelente en el documento se muestra el rendimiento que tiene cada uno de los documentos al realizar diferentes consultas.

2.1.4.4.1 Diseño 1

Este esquema contendrá un documento por artículo y todos ellos estarán en la misma colección. El esquema de este diseño se basa en que cada artículo está representado por un documento, considerando un artículo en formato dvd o blueray como dos artículos diferentes.

Este tipo de diseño intenta no tener un esquema muy complejo del documento evitando el uso de listas de subdocumentos, pero al mismo tiempo esto hace que en el número de documentos aumente y con ello el tamaño de la colección.

El esquema general de los documentos tendrán la siguiente forma:

- _id: identificador único del documento.
- titulo: título del libro o la película.
- tipo: tipo de artículo, película o libro.
- formato: el formato del artículo, que puede ser dvd, blueray o collection para las películas y tapa dura o tapa blanda para los libros.
- genero: una lista conteniendo todos los géneros a los que pertenece el artículo.
- detalles: un subdocumento que contiene información sobre el artículo.
- precio: un subdocumento con información sobre el precio del producto. Este subdocumento contiene los campos:
  - precio: el precio del artículo. Como se puede ver el nombre del campo de un subdocumento puede tener el mismo nombre que cualquier campo del documento que no esté dentro de ese mismo subdocumento. Ésto se debe a que internamente el campo del subdocumento es 'precio.precio' y no simplemente "precio".
  - descuento: el descuento que tiene el artículo.
  - envio: si el envío es gratis o de pago.

Las principales diferencias entre el esquema de los documentos de tipo libro y tipo película se encuentra en el subdocumento del campo detalles, aunque tienen algunos campos en común, como puede ser el campo que hace referencia al idioma del artículo, otros son únicos para cada tipo de artículos.

El esquema del subdocumento detalles para las películas es el siguiente:

- actores: lista de actores que aparecen en la película.
- productora: lista de las productoras que tomaron parte en parte en la película.
- duración: duración en minutos de la película.
- idioma: idioma de la película.
- fecha: fecha de estreno de la película.
- director: lista de los directores de la película.
- guionista: lista de los guionistas de la película.
- descripción: descripción de la trama de la película.
- datos: enlace a la información de contenidos de "FreeBase" para esa película.

En el caso del los libros el esquema de detalles será el siguiente:

- autores: lista de autores del libro.
- editorial: nombre de la editorial del libro.
- paginas: número de las páginas del libro.
- idioma: idioma de la película.
- fecha: fecha de estreno de la película.
- descripcion: descripción de la trama de la película.
- datos: enlace a la información de contenidos de "FreeBase" para esa película.

2.1.4.4.2 Diseño 2

Este diseño es similar al anterior salvo que en lugar de tener un documento por artículo, tiene un documento que contiene información de un libro o película en varios formatos diferentes. La idea de este esquema es la de agrupar información de productos similares en un mismo documento guardando los datos que pueden diferir de un artículo con uno u otro formato, en una lista de subdocumentos. Estos subdocumentos contiene información que puede variar para una misma película o libro, en función de cual sea su formato. Por ejemplo, para la película "WALL-E" que puede estar disponible en formato dvd, blueray y de coleccionista, los detalles de los artículos en este dataset son los mismos independientemente del formato. Sin embargo, otros datos como el precio variarán dependiendo del formato del artículo; por tanto se puede crear un sólo

documento para la película WALL-E el cual contenga en el campo "formato" una lista con tres subdocumentos cada uno de ellos con un campo que índica a que formato hacen referencia los datos de ese subdocumento y campos referentes por ejemplo al precio de la película con ese formato.

Por tanto, el esquema de los documentos es muy similar al anterior, únicamente varían el campo "precio" y el campo "formato". En este caso el campo "precio" desaparece y el campo "formato" se convierte en una lista de subdocumentos donde cada subdocumaneto contiene el precio que cuesta cada película o libro en función de su formato.

El esquema de cada documento tendría la siguiente forma:

- _id: identificador único del documento.
- titulo: título del libro o la película.
- tipo: tipo de artículo, película o libro.
- genero: una lista conteniendo todos los generos a los que pertenece el artículo.
- detalles: un subdocumento que contiene información sobre el artículo y al igual que en el diseño 1 los campos varían en función de si se trata de un libro o una película.
- formato: este campo contiene la lista de subdocumentos con los precios de la película o libro para cada posible formato de esa película o libro. Cada dsubdocumento en la lista contiene los campos:
  - formato: el formato del artículo, que puede ser dvd, blueray o collection para las películas y tapa dura o tapa blanda para los libros.
  - precio: que contiene un subdocumento con información sobre el precio del producto. Este subdocumento contiene los campos:
    - precio: el precio del artículo.
    - descuento: el descuento que tiene el artículo.
    - envio: si el envío es gratis o de pago.

El subdocumeto detalles es idéntico al del diseño 1 y al igual que en éste los campos variarán en función de si se trata de una película o un libro.

2.1.5 Diseño de operaciones

MongoDB provee de varios mecanismos para realizar consultas y modificaciones sobre los documentos de las bases de datos. El uso de esto mecanismos depende principalmente las tareas que se quieran realizar. Los dos más importantes son:

1. Operaciones CRUD: estas son operaciones básicas de creación, lectura, modificación y eliminación.[17]
2. Entorno de agregación: este entorno se utiliza mediante la función "aggregate()" del cliente mongo. Este tipo de operaciones agrupa varios documentos y es capaz de realizar operaciones sobre estos. Esta operación devuelve un único resultado en forma un documento que contiene una lista de subdocumentos.[18]

En los dos casos anteriores la sintaxis para llamar a las funciones de las operaciones CRUD y a la función "aggregate()" requiere que se indique qué base de datos se usa y la colección sobre la que se hace la operación. Para indicar la base de datos se le da a una variable el valor de la base de datos mediante el comando:

```
> db=db.getSiblingDB('nombre de la base de datos')
```

Esta operación cumple dos funciones, primero avisa al shell de que se va a utilizar la base de datos contenidos - por lo que sería similar a la operación "use" de MySQL - y después asigna a "db" el identificador de la base datos.

Alternativamente se puede usar el comando "use <nombre_BD>", similar al comando "USE" de MySQL.

Una vez hecho esto para realizar las llamadas a las diferentes funciones que realicen operaciones sobre los documentos de alguna colección se utiliza la sintaxis:

```
> db.coleccion.funcion()
```

Seguidamente se muestra con más detallada el uso de las distintas herramientas para realizar consultas sobre los documentos. Los ejemplos que se muestran se realizan sobre la base de datos "contenidos" la cual contiene los esquemas de los diseños propuestos en la sección anterior y cuyos nombre son "articulos" para la colección del diseño 1 y "articulosD2" para la colección del diseño 2.

2.1.5.1 Operaciones CRUD

En esta sección se habla de las diferentes operaciones disponible las cuales permiten realizar consultas sobre los documentos.

Estas operaciones forman la herramienta principal de MongoDB para realizar consultas y permiten realizar operaciones sobre los documentos de forma sencilla. Además, junto a estas operaciones se pueden utilizar las llamadas funciones sobre cursores que permiten realizar modificaciones, entre otras tareas, sobre los resultados obtenidos en las consultas.

2.1.5.1.1 Operaciones de lectura

Las operaciones de lectura buscan siempre los documentos en una colección concreta. La estructura de una consulta de lectura consta de cuatro partes, tres de ellas opcionales:

1. Selección de la colección.
2. Criterio de búsqueda del documento. Esta parte es opcional, si no se indica en la consulta se devolverán todos los documentos de la colección.
3. Proyección de los campos del documentos. Esta parte es opcional.
4. Modificación de los resultados obtenidos. Esta parte es opcional.

Por tanto las consultas de lecutura tendrá la siguiente forma:

```
db.coleccion.funcion_de_busqueda(
        {criterio de busqueda},
        {proyeccion}
   ).funcion_de_modificacion();
```

Los métodos CRUD principales para realizar lecturas sobre los datos son "find()" y "findOne()".

- find({criterio de búsqueda},{proyección}): este método devuelve todos los documentos que hay en una colección.
- findOne({criterio de búsqueda},{proyección}): este método devuelve únicamente el primer documento encontrado.

Así por ejemplo, para encontrar el primer documento en la colección "articulos" se haría la siguiente consulta - suponiendo que se ya se ha seleccionado la base de datos:

```
> db.articulos.findOne()
```

Y para encontrar todos los documentos de la colección se haría:

```
> db.articulos.find()
```

Esta última consulta equivaldría a realizar en SQL la operación "SELECT * FROM ARTICULOS", donde se obtienen todos los artículos de la tabla, siempre y cuando la información que hay en la tabla "Articulos" fuera la misma que en la colección y no fuera necesario realizar ningún "JOIN".

Un ejemplo de una consulta sencilla en la que se obtengan todos los artículos de tipo película que llevan el título "WALL-E", sería el siguiente:

```
> db.articulos.find({"titulo":"WALL-E","tipo":"movie"})
```

Además si se quisiera seleccionar únicamente el campo "precio", sería necesario añadir la proyección a la consulta como un segundo parámetro del método. Este parámetro contiene el nombre del campo que se quiere seleccionar seguido del valor 1. La consulta final tendría la siguiente forma:

```
> db.articulos.find(
    {"titulo":"WALL-E","tipo":"movie"},
    {"precio":1}
    )
```

Esta consulta devolverá por defecto el subdocumento precio junto al campo "_id" para cada documento que cumpla el criterio de búsqueda. Esta consulta debería devolver tres documentos:

```
{ "_id" : ObjectId("51efc5b744aefcc2d72ca2d7"),
  "precio" : { "precio" : 40, "descuento" : 0, "envio" : "gratis" }
}
{ "_id" : ObjectId("51efc5b744aefcc2d72ca2d8"),
  "precio" : { "precio" : 60, "descuento" : 0, "envio" : "gratis" }
}
{ "_id" : ObjectId("51efc5b744aefcc2d72ca2d9"),
  "precio" : { "precio" : 75, "descuento" : 0, "envio" : "gratis" }
}
```

Si se quisiera eliminar un campo concreto del resultado, se haría de forma similar a la proyección anterior, salvo que en lugar de darle el valor uno al campo, se le asignaría el valor cero. En general no se puede mezclar la opción de seleccionar un campo (1), con la opción de no seleccionar un campo (0); salvo en el caso del campo "_id" ya que éste

aparece siempre por defecto a no ser que se indique lo contrario. Por ejemplo, para el caso anterior si se quisiera eliminar el campo "_id" se haría de la siguiente forma:

```
> db.articulos.find(
    {"titulo":"WALL-E","tipo":"movie"},
    {"precio":1,_id:0}
  )
```

Una sentencia SQL que devolviera valores similares equivaldría a realizar la siguiente consulta:

```
    SELECT PRECIO , DESCUENTO, ENVIO
    FROM ARTICULO
    WHERE TITULO='WALL-E AND TIPO='MOVIE'
```

Por tanto, se podría considerar que para consultas simples el parámetro que hace referencia al criterio de búsqueda de los métodos find() y findOne() equivaldría a la clausula "WHERE" de las consultas SQL y el parámetro referente a la proyección equivaldría a la clausula "SELECT". Además para indicar que se quieren filtrar los documentos en función de los valores de varios campos,en lugar de usar "AND" como en SQL, en MongoDB únicamente se separan las condiciones de búsqueda por una coma.

En cierto modo se pueden considerar las siguientes equivalencias para un operación de lectura:

- SELECT : db.<coleccion>.find(...)
- SELECT _ID, PRECIO : find(... , {"precio":1})
- FROM ARTICULO : db.articulos...
- WHERE TITULO='WALL-E AND TIPO='MOVIE':
  find({"titulo":"WALL-E","tipo":"movie"})

Si se quisieran filtrar los documentos en función de si cumple una condición u otra, es decir, como si se usara "OR" con la clausula "WHERE" de SQL, se dispone del operador "$or" - en las sucesivas secciones se hablará de otros tipos de operadores disponibles -, el cual es utilizado junto a una lista de condiciones. Por ejemplo, para realizar una consulta que obtiene un artículo con título "WALL-E" y con formato "dvd" o "blueray", se haría la siguiente consulta.

```
> db.articulos.find(
    {"titulo":"WALL-E",
    $or:[{"formato":"dvd"},{"formato":"collection"}]
    })
```

En SQL una consulta similar tendría la forma:

```
SELECT *
FROM ARTICULO
WHERE TITULO='WALL-E AND
      (FORMATO='DVD' OR FROMATO = 'COLLECTION')
```

2.1.5.1.1.1 Subdocumentos

MongoDB permite modelar los datos de forma que haya subdocumentos dentro de los documentos. A la hora de realizar consultas sobre estos documentos existen dos opciones:

1. Realizar la consulta donde todo el subdocumento encontrado coincide con los datos del criterio de búsqueda. Por ejemplo:

```
>db.articulos.find(
    {"precio" :
        { "precio" : 40,
          "descuento" : 0,
          "envio" : "gratis" }
    })
```

2. Realizar la consulta sobre un campo del subdocumento. Para hacer esto hay que indicar el nombre del campo del documento seguido de un punto, seguido del nombre del campo del subdocumento sobre el que realizar la comparación; y todo ello recogido entre comillas simples. Por ejemplo, para realizar una consulta en la que se busque el título de las películas con una duración de 98 minutos:

```
> db.articulos.find(
    {'detalles.duracion':98},
    {"titulo":1}
    )
```

2.1.5.1.1.2 Listas de elementos

Además de subdocumentos, MongoDB también acepta arrays de elementos, siendo estos elementos tanto valores simples, como subdocumentos - similar al esquema utilizado en el segundo diseño del dataset de pruebas para el campo formato.

1. Como en el caso anterior se tienen varias opciones para realizar las consultas sobre campos que contienen listas de elementos.

   Por un lado, es posible hacer una consulta donde se indica que el array debe coincidir exactamente con la lista de elementos indicada al hacer la consulta. Por

ejemplo si se quisiera buscar el título de los artículos en los que la lista de guionistas es : "Alec Sokolow", "Joss Whedon", "Joel Cohen" y "Andrew Stanton" – la cual coincide con la lista de guionistas de la película Toy Story –, se debería hacer la siguiente consulta:

```
> db.articulos.find(
    {"detalles.guionista":["Alec Sokolow", "Joss Whedon",
                           "Joel Cohen", "Andrew Stanton"]},
        {"titulo":1,_id:0})
```

Dicha consulta devolverá el título de tres artículos: dvd, blueray y colecionista, para la película toy story -si se estuviese usando la colección que contiene los documentos con el esquema del segundo diseño se obtendría un único resultado.

2. Por otro lado, se puede realizar una consulta de forma que se devuelvan aquellos documentos en los que el campo array contiene un elemento dado, pero no siendo necesario que sea el único valor en el array. Por ejemplo, para preguntar el título de los artículos cuyo género, entre otros, sea comedia se haría la siguiente consulta:

```
> db.articulos.find(
    {"genero":'Comedy'},
    {"titulo":1,_id:0,"tipo":1})
```

Es importante que el valor de búsqueda vaya entre comillas simples si se trata de una cadena de caracteres.

3. También es posible indicarle en que posición del array debería encontrarse dicho valor. Por ejemplo, si se tiene ordenado el género de los artículos de mayor a menor importancia se podría preguntar por los artículos que son principalmente comedias:

```
> db.articulos.find(
    {"genero.0":'Comedy'},
    {"titulo":1,_id:0,"tipo":1})
```

Además, es posible mejorar las opciones de búsqueda mediante el uso se operadores especiales de los que se hablará más adelante.

2.1.5.1.1.3 Búsqueda mediante expresiones regulares

Además de utilizar texto exacto para realizar la búsqueda de los documentos, MongoDB permite realizar consultas con condiciones expresadas en forma de expresiones regulares. Para indicar que el valor de búsqueda es una expresión regular se rodea con el símbolo "/". Además al final de la expresión regular se pueden añadir opciones como: i para indicar que la palabra puede tener cualquier letra en mayúsculas o minúsculas, o m

para indicar que puede haber un salto de linea. Por ejemplo, si se quiere encontrar el documento cuyo título es wall-e pero no necesariamente todas las letras en minúsculas, se puede realizar la siguiente consulta:

```
> db.articulos.find({titulo:/wall-e/i})
```

Otro ejemplo sería buscar todos los documentos que contengan en su título la palabra Java:

```
> db.articulos.find({titulo:/.* Java *./})
```

Además de esta sintaxis es posible utilizar el operador "$redex". Por ejemplo, para realizar la consulta anterior en la que se busca cualquier combinación de mayúsculas o minúsculas del titulo "wall-e":

```
> db.articulos.find(
    {titulo:{$regex:'wall-e',$options: 'i'}}
    )
```

2.1.5.1.1.4 Cursor

El método "find()", no devuelve todos los resultados de una vez, sino que devuelve un número concreto de elementos – éste número se puede cambiar mediante el método "limit()". Para obtener todos los valores es necesario iterar el curso devuelto por "find()".

Puesto que el shell de mongo permite utilizar sentencias con una sintaxis similar a javascript, una manera de hacer lo anterior es asignar al cursor una variable e iterar el cursor valiéndose de los métodos ".hasNext" y ".next()" de la siguiente manera:

```
> var c = db.articulos.find(
    {formato:"dvd",
    'precio.precio':{$lt:30}
    })

> while ( c.hasNext() ) printjson( c.next() )
```

Una forma más sencilla es utilizar el método "forEach()" junto con "printjson". Este método realiza una operación dada cada vez que la consulta de lectura devuelva un documento; y al mismo tiempo itera por todos los resultados, de forma automática. De esta forma si la tarea indicada es la de imprimir los documentos - mediante la sentencia "printjson" -, por cada resultado posible de la consulta imprimirá el correspondiente documento. La consulta sería la siguiente:

```
> db.articulos.find({tipo:"movie"}).forEach(printjson)
```

También es posible pasar como parámetro al método "forEach()" una función que realice varias operaciones. Más adelante se trata este método en más profundidad y se muestran varios ejemplos de sus posibles usos.

2.1.5.1.1.5 Operadores de selección

Los operadores de selección son clausulas que permiten dar una mayor flexibilidad y exactitud a las condiciones de búsqueda de documentos.

- $all : dada una lista de valores para un campo de tipo array, este operador permite devolver todos los documentos que contienen esos valores. El campo de los documentos seleccionados pueden contener otros valores además de los buscados. Por ejemplo, para buscar artículos cuyo género es "comedia" y "fantasia":

```
> db.articulos.find({genero:{$all:["Comedy","Fantasy"]}})
```

- $gt/$gte/$lt/$lte: permiten obtener documentos donde el valor de un campo es mayor/menor – gt, lt - o mayor/menor igual – gte, lte - que el valor indicado en la consulta. Por ejemplo, para obtener artículos cuyo precio es mayor a 60:

```
> db.articulos.find({'precio.precio':{$gt:60}})
```

- $in: selecciona los documentos donde el valor del campo se encuentra entre alguno de los valores dados en la consulta. Si el campo es un array, entonces, devuelve todos los documentos donde al menos uno de los valores se encuentra en el array. Por ejemplo, obtener todos los artículos cuyo precio es 50 o 30:

```
> db.articulos.find({'precio.precio':{$in:[50,30]}})
```

- $ne: selecciona los documentos donde el valor del campo a buscar no coincide con el valor indicado.

```
> db.articulos.find({tipo:{$ne:"book"}})
```

- $nin: es similar a $in salvo que en este caso selecciona todos los documentos donde el valor del campo no contenga ninguno de los valores indicados en la consulta.

```
> db.articulos.find({'precio.precio':{$nin:[50,30]}})
```

- $not: selecciona los documentos que no coinciden con una condición dada. Por ejemplo, para obtener los artículos cuyo precio no es mayor que 30.

```
> db.articulos.find({'precio.precio':{$not:{$gt:30}}})
```

- $or/$and/$not/$nor: permite seleccionar documentos tal que dado un arry de condiciones, los documentos cumplen alguna de las condiciones, todas las condiciones o ninguna de las condiciones. Por ejemplo, para encontrar los artículos que no son de tipo "book" y cuyo precio no es mayor a 30:

```
> db.articulos.find({$nor:[{'precio.precio':{$gt:30}},
                           {tipo: "book"}]})
```

- $exist <true/false>: sólo devuelve los documentos que contienen el campo indicado, siempre y cuando el valor del operador sea true, si es false devuelve los documentos que no contienen el campo.

- $size: permite seleccionar los documentos que contienen un campo de tipo array cuyo número de elementos coincide con el número indicado en la consulta.

- $type: selecciona los documentos donde el campo de la consulta es del tipo indicado. Este valor se indica mediante un número entero que representa al tipo de datos.[19]

- $elemMatch: selecciona los documentos donde un campo de tipo array cumple las condiciones dadas. Por ejemplo, utilizando la colección "articulosD2" la cual tiene subdocumentos dentro del campo array "formato" la siguiente consulta obtendría los documentos cuyo formato es dvd y cuyo precio es menor que 30:

```
> db.articulosD2.find(
    {formato:
        {$elemMatch:
            {'precio.precio':{$lt:30},
              formato:"dvd"}
        }})
```

2.1.5.1.1.6 Operadores de proyección

- $elemMatch: es el mismo operador que en la parte de selección, sin embargo, si se incluye esta condición en la parte de proyección de la consulta, la consulta devolverá los campos correspondientes al primer elemento del array que cumpla la condición. Es decir, no devuelve toda la lista de subdocumentos, sino únicamente el primer subdocumento que cumpla la condición.

- $slice: permite seleccionar un documento donde un campo de tipo array únicamente contendrá los n primeros elementos, donde n es el valor dado al operador. Por ejemplo, para obtener el primer artículo y que el campo "genero" mostrado sólo contenga dos elementos, la consulta se haría de siguiente manera:

```
> db.articulos.findOne({},{genero:{$slice:2}})
```

- $: si se hace una consulta donde una de las condiciones de selección utiliza un campo de tipo array, se puede usar este operador en la proyección para indicar que únicamente se seleccione el elemento que ha hecho que se cumpla la condición. Por ejemplo, si se quiere seleccionar el primer documento con género "comedia" y que únicamente muestre el título y el "genero", pero solo con el valor que hace cumplir la condición. La consulta ser haría de la siguiente manera:

```
> db.articulos.findOne(
    {genero:'Comedy'},
    {"genero.$":1})
```

2.1.5.1.2 Operaciones de escritura

Las operaciones de escritura son crear, insertar, modificar – create, update, insert. Para realizar estás operaciones, en MongoDB están disponibles los métodos : "insert()", "update()", "save()", "findAndModify()" y "remove()".

2.1.5.1.2.1 Crear

MongoDB no tiene un método concreto a partir del cual se pueda crear una colección, como ocurre en sistemas relacionales, como con MySQL donde se puede usar "Create" para crear tablas antes de insertar los elementos de las tablas.

En el caso de MongoDB, las colecciones se crean automáticamente al insertar el primer documento de la colección. Por ejemplo, si se miran las colecciones presentes en la base de datos "contenidos" mediante el comando "show collections", se puede ver que las colecciones presentes son cuatro, "articulos", "articulosD2" y las dos colecciones del sistema usadas para guardar los usuarios y los índices. Si se inserta un documento en una colección inexistente de la siguiente forma:

```
> db.NombreColeccion.insert({"titulo":"SoylentGreen"})
```

Al volver a hacer "show collections" la colección "NombreColeccion" aparece entre las colecciones de la base de datos debido a que ésta se ha creado automáticamente. Para eliminar dicha colección es suficiente con hacer:

```
>db.crearColeccion.drop()
```

Cuando se crean bases de datos ocurre algo similar, si el sistema de identificación de la base de datos no está habilitado – o el usuario de dicha base de datos se identifica sobre la base de datos "admin" - entonces para crear una base de datos es suficiente con indicar que se quiere usar una base de datos de nombre X que no exista aún, de forma

que al insertar la primera colección se creará con ello la base de datos de nombre X. Por otro lado, si el sistema de identificación está habilitado y se añade un nuevo usuario en la colección "system.user" de dicha base de datos X – y no en la base de datos "admin" – entonces la base de datos será creada automáticamente al añadir un nuevo usuario.

2.1.5.1.2.2 Insertar

Como se comento en apartados anteriores es posible insertar documentos con diferentes esquemas en una misma colección, con lo que no es necesario que un nuevo documento siga la estructura del primer documento insertado.

Para realizar las inserciones se pueden utilizar varios métodos: "insert()", "update()" y "save()".

- Insert(): Esté es el método principal, permite insertar tanto un documento como una lista de documentos. Este método tiene la forma "insert(<documento>)". Por ejemplo, para insertar un nuevo artículo - con menos campos para simplificar la consulta - en la base de datos contenidos la consulta tendría la siguiente forma:

```
> db.articulos.insert(
    {"titulo":"SoylentGreen2",
     "tipo":"movie",
     "formato":"dvd",
     "detalles":{"duracion":15}
     })
```

  De esta forma al buscar el artículo con título "SoylenteGreen2" se puede ver que el nuevo documento ha sido insertado sin ningún problema, aunque el esquema del documento no coincida con el esquema de los otros documentos.

  Si se tiene que insertar gran cantidad de documentos, resulta más eficiente insertar una lista de documentos en lugar de insertarlos uno a uno, utilizando para ello algún bucle. Esto sería similar a la inserción en Batch de SQL. Por ejemplo, para insetar una película para los tres formatos disponibles, siguiendo el esquema anterior se haría lo siguiente:

```
db.articulos.insert([
 {"titulo":"SoylentGreen3", "tipo":"movie",
  "formato":"dvd","detalles":{"duracion":15}},

 {"titulo":"SoylentGreen3", "tipo":"movie",
  "formato":"blueray","detalles":{"duracion":15}},

 {"titulo":"SoylentGreen3",
```

```
  "tipo":"movie","formato":"collection","detalles":{"duracion":15}}
 ])
```

- Update():

  De forma general este método tiene dos parámetros: el criterio de búsqueda del documento a modificar y los datos del nuevo documento. Este método se utiliza para modificar documentos ya existentes; sin embargo, existe la opción de añadirle un tercer parámetro "{ upsert: true }", el cuál sirve como flag para indicar que si no se encuentra ningún documento que cumpla el criterio de búsqueda, entonces se inserte un documento con los datos dados en el segundo parámetro. Este método podría ser usada de dos maneras:

  1. Sin el operador $set:

     ```
     > db.collection.update(
         {criterio de busqueda},
         {<datos>},
         { upsert: true })
     ```

  2. Con el operador $set:

     ```
     > db.collection.update(
             {criterio de busqueda},
             {$set:{<datos>}},
             { upsert: true } )
     ```

  La diferencia entre ambos casos es que en el segundo se añadirá un documento con los datos del apartado "datos" junto con los datos presentes en el criterio de búsqueda.

  En secciones posteriores se explicará de forma más extendida el uso del método "update()", pero por ejemplo, si se quiere modificar con nuevos datos el artículo de título "SoylentGreen2" con formato "blueray" de tal forma que si el documento existe se modifique y si no se cree, la consulta se haría la siguiente forma:

  ```
  db.articulos.update(
      { "titulo":"SoylentGreen2", "formato":"blueray"},
      { "titulo":"SoylentGreen2", "tipo":"movie",
        "formato":"blueray","detalles":{"duracion":15}},
      {  upsert: true }
  )
  ```

  Al buscar el documento de título "SoylentGreen2", se puede ver que se ha insertado un nuevo documento con los datos anteriores.

  Si se utiliza el operador "$set" para realizar una operación similar a la anterior pero siendo el valor del formato "collection", la operación sería la siguiente:

```
db.articulos.update(
        {"titulo":"SoylentGreen2", "formato":"collection"},
        {$set:{"tipo":"movie","detalles": {"duracion":15}}},
        { upsert: true }
)
```

Como se puede ver al buscar el documento de título "SoylentGreen2" y formato de "collection", el esquema del documento incluye los datos de las condiciones de búsqueda:

```
{
        "_id" : ObjectId("51f5a6e297209b5d375d8aeb"),
        "detalles" : { "duracion" : 15 },
        "formato" : "collection",
        "tipo" : "movie",
        "titulo" : "SoylentGreen2"
}
```

- save(): Este método es equivalente a un "update()" con "upsert", es decir, que si encuentra un documento que cumple el criterio de búsqueda lo modifica con los datos dados y si no inserta un nuevo documento.

2.1.5.1.2.3 Modificar

En la sección anterior ya se explicó como realizar un par de modificaciones sencillas en las que se utilizaba el flag "upsert" para insertar nuevos documentos. Sin embargo, si no se quiere insertar un nuevo documento en caso de fallar la búsqueda, sería suficiente con no incluir el parámetro referente al flag.

Por ejemplo, una operación similar a las de la sección anterior pero sin el flag sería de la siguiente forma:

```
> db.articulos.update(
    {"titulo":"SoylentGreen4"},
    {"titulo":"SoylentGreen4", "tipo":"movie",
     "formato":"dvd","detalles":{"duracion":15}}
)
```

En este caso no se encontrará ningún documento con título "SoylentGreen4"; y como,además, no se ha utilizado el flag "upsert", no se debería añadir ningún documento nuevo. Por tanto, al realizar la consulta "db.articulos.findOne("titulo":"SoylentGreen4")" debería devolver null.

Uno de los aspectos a tener en cuenta al utilizar esta función es que únicamente se modifica un documento. Por ejemplo, si se quiere modificar el primer documento cuyo campo "titulo" es "SoylentGreen3" - en la sección anterior se insertaron tres documentos con estos títulos - la consulta sería:

```
> db.articulos.update(
    {"titulo":"SoylentGreen3"},
    {"titulo":"SoylentGreen3", "tipo":"book",
     "formato":"Tapa Blanda","detalles":{"paginas":100}}
)
```

Como se puede ver al buscar todos los documentos con título "SoyletGreen3", solo se ha modificado el primer documento y no el resto.

Si se quiere que la operación modifique todos los documentos que cumplen con una condición es necesario usar la opción "multi" de la siguiente forma:

```
db.articulos.update(
    {"titulo":"SoylentGreen3"},
    {$set{"titulo":"SoylentGreen3", "tipo":"book",
        "formato":"Tapa Blanda","detalles":{"paginas":100}}
        },
    {multi:true}
)
```

Hay que notar, que la opción "multi" solo funciona si la operación contiene operadores como "$set" - el cual hace que únicamente se modifiquen los campos indicados en el segundo parámetro.

Además de las operaciones anteriores, monogDB provee de una serie de operadores que permiten añadir un nuevo campo, eliminar un campo, realizar modificaciones en los elementos de un array:

- Modificación de un campo:

  Para realizar modificaciones de campos concretos dentro de un documento o en un subdocumento, se utiliza el operador "$set" ya visto en secciones anteriores. Cuando no se utiliza ningún operador MongoDB asume que se modifica por completo todo el documento de forma que se borran los campos antiguos y se añaden los datos indicados en el segundo parámetro de la operación.

  Para modificar el campo "formato" del primer artículo con título "SoylentGreen3" - documento creado durante las secciones anteriores – para mantener el resto de los campos se haría la operación de modificación de la siguiente forma:

```
 db.articulos.update(
    {"titulo":"SoylentGreen3"},
    {$set:{"formato":"Tapa Dura"}
})
```

Al buscar todos los documentos con título "SoyletGreen3",se puede ver que únicamente se ha modificado el campo "formato" y no se han eliminado el resto de campos.

Por otro lado, si lo que se quiere es modificar un campo dentro de un subdocumento se debería indicar de la siguiente manera:

```
 db.articulos.update(
        {"titulo":"SoylentGreen3"},
        {$set:{'detalles.paginas':305}
})
```

Hay que notar que si en lugar de usar " 'detalles.paginas':305 ", se usa { $set: { "detalles" : { "paginas" : 305 }}}, se estaría eliminando cualquier campo presente en el subdocumento "detalles" y se reemplazaría únicamente por el campo "paginas".

- Añadir un nuevo campo:

  Para añadir un nuevo campo al documento o uno de los documentos también se utiliza el operador "$set" y el formato sería similar al de los casos anteriores, donde lo único que hay que hacer es indicar un campo que no exista en el documento o subdocumento y este será añadido. Esto acarrea un problema y es que si se quiere modificar un campo y se escribe el nombre de dicho campo de forma errónea, no se producirá ningún error y se añadirá un nuevo campo.

- Eliminar un campo:

  MondoDB permite eliminar campos concretos dentro de un documente mediante el método "update()", y para ello es necesario utilizar el operador "$unset ".

  Por ejemplo, si quisiéramos eliminar el campo "tipo" del primer documento con titulo "SoylentGreen2" se haría lo siguiente:

```
> db.articulos.update(
    {"titulo":"SoylentGreen2"},
    {$unset:{"tipo":1}
})
```

Como se puede ver no es necesario indicar el valor del campo en el documento, sino únicamente darle el valor uno para indicar que es el campo seleccionado para eliminar.

- Modificar arrays:

  Las modificaciones básicas de elementos en los arrays se hacen también utilizando el operador "$set" e indicando la posición dentro del array donde se encuentra el elemento o bien indicando en el criterio de búsqueda el campo que hay que buscar en el array.

  Por ejemplo, para el primer caso, si se quiere modificar el primer elemento del array actores del primer artículo con título "Toy Story" y de tipo "movie" se haría la siguiente operación:

```
db.articulos.update(
    {"titulo":"Toy Story", "tipo":"movie"},
    {$set:{'detalles.actores.0':'Tommy Hanks'}
})
```

  Como se puede ver, al poner 'detalles.actores.0' se indica que se quiere modificar el elemento en la posición cero del campo actores, el cual está dentro del subdocumento detalles.

  Si por otro lado se quiere modificar un elemento sin tener que indicarle en que posición está dicho elemento se haría lo siguiente:

```
db.articulos.update(
    {"titulo":"Toy Story","tipo":"movie",
     'detalles.actores':'Tommy Hanks'},
    {$set:{'detalles.actores.$':'Tom Hanks'}
})
```

  La operación anterior sobre el array se divide en dos partes, primero busca el documento que contiene dicho valor en una cierta posición, y segundo se utiliza el símbolo "$", para indicar que la posición del array donde se encontró el elemento del criterio de búsqueda es la posición del array a modificar.

  Además de estas modificaciones, MongoDB ofrece una serie de operadores que permiten realizar diferentes modificaciones sobre los arrays como: añadir un elemento,eliminar un elemento, eliminar todos los elementos...

2.1.5.1.2.4 Eliminar

Las operaciones de eliminación permiten eliminar documentos, y para esto se utiliza la función "remove()".

Existen varias opciones a la hora de eliminar documentos:

- Eliminar todos los documentos que coinciden con la condición de búsqueda.

```
db.articulos.remove({"titulo":"SoylentGreen2"})
```

- Eliminar sólo el primer documento que coincide con la opción de búsqueda.

```
db.articulos.remove({"titulo":"SoylentGreen3"},1)
```

- Eliminar todos los documentos.

```
db.articulos.remove()
```

2.1.5.1.2.5 Operadores de modificación

Además de los operadores anteriores existen otros que permiten realizar modificaciones sobre campos.

- $inc : permite incrementar el valor actual de un campo de tipo numérico en una cantidad dada.
- $rename: permite renombrar un campo. Si el campo a modificar no existe entonces no hace nada, pero si se quiere renombrar más de un campo y solo existen algunos de ellos, entonces modifica los existentes e ignora el resto, es decir, no añade nuevos campos si no existe el campo a modificar. Otra ventaja de este operador es que permite mover campos de un subdocumento a otro dentro del mismo documento.
- $setOnInsert : se utiliza junto al flag "upsert" e indica que un campo/s dado será incluido en el documento si se produce una inserción, pero si es una modificación el campo no será ni modificado ni incluido.

  Por ejemplo, para realizar un incremento en el campo "duración" del primer documento con título "SoylentGreen2" se haría una consulta de la siguiente forma:

```
db.articulos.update( {"titulo":"SoylentGreen2"},
                           {$inc:{'detalles.duracion':60}
})
```

  Para renombrar el campo duración de ese mismo documento se podría hacer lo siguiente:

```
db.articulos.update(
        {"titulo":"SoylentGreen2"},
        {$rename:{'detalles.duracion':'detalles.minutos'}
})
```

Esto hará que el campo duración del subdocumento "detalles" sea renombrado con el nombre minutos. Por otro lado, si se hubiese querido mover el campo a otro subdocumento habría que haber indicado en el segundo parámetro lo siguiente: " $rename:{'detalles.duracion':'detalles.minutos' ". Si el subdocumento donde se mueve el campo no existe, este se creará automáticamente; pero si el subdocumento de origen queda vacío este no es eliminado automáticamente.

Además, de estos operadores existen otros que permiten realizar diferentes modificaciones sobre los arrays como: añadir un elemento, eliminar un elemento, eliminar todos los elementos...

Estos operadores son los siguientes:

- $addToSet : añade un nuevo elemento a un array, siempre y cuando no esté ya en dicho array. Por ejemplo, la siguiente consulta añadirá el nombre "Paco" a la lista de actores:

```
db.articulos.update(
        {"titulo":"Toy Story", "tipo":"movie"},
        {$addToSet :{'detalles.actores':"Paco"}
    })
```

  Si se busca el artículo de nombre "Toy Story" y tipo "movie" se ve que se ha añadido un nuevo actor – Paco - a la lista. Si se realiza de nuevo la modificación anterior, puesto que "Paco" ya está en la lista, no se volverá a insertar dicho valor en el array.

  Además, si se quisiera añadir varios elementos se podría utilizar el operador $each

```
db.articulos.update(
        {"titulo":"Toy Story", "tipo":"movie"},
        {$addToSet :
            {'detalles.actores':
                {$each:["Paco0","Paco1","Paco","Paco2"]}}
                })
```

  En este caso, se añadirán todos los elementos de la lista, excepto "Paco", el cuál había sido añadido en el paso anterior,

- $pop: elimina el primer o el último elemento de un array, dependiendo del valor que se le indique al array. Si el valor es un uno, entonces se elimina el último elemento, si es menos uno, entonces, se elimina el primer elemento.

  Por ejemplo, para eliminar al actor "Paco2" añadido al final de la lista de actores en el punto anterior la consulta a realizar sería la siguiente:

```
db.articulos.update({"titulo":"Toy Story", "tipo":"movie"},
                    {$pop :{'detalles.actores': 1}
                })
```

Si se busca el artículo de nombre "Toy Story" y tipo "movie" se ve que el elemento "Paco2" ya no está presente.

- $pullAll : elimina varios valores dados. Además, se eliminarán todos los duplicados.Por ejemplo, si se quieren eliminar todos lo valores añadidos en los puntos anteriores se haría la siguiente operación:

```
db.articulos.update(
        {"titulo":"Toy Story", "tipo":"movie"},
        {$pullAll :
            {'detalles.actores':["Paco0","Paco1",
                                 "Paco","Paco2"]}
})
```

En este caso, la operación elimina los elementos "Paco0","Paco1" y "Paco", mientras ignora el elemento "Paco2" que ya no estaba presente en la lista tras haber sido eliminado en el punto anterior. Por tanto, se puede ver que en caso de que uno de los elementos de la lista no esté presente en el array del campo, no se cancela la operación, y simplemente se ignora dichos elementos.

- $pull: es equivalente a $pullAll con la diferencia de que en este caso sólo no se le puede indicar una lista de valores a eliminar, sino que tiene que ser un valor concreto.

- $push: añade al final del array los elementos indicados, independientemente de si existe un valor similar dentro del array. En este caso si el campo no existe, es añadido al documento, pero si existe y no es un array entonces la operación devolvería un error.

  Por ejemplo, para añadir una lista de actores se puede realizar la siguiente consulta:

```
db.articulos.update(
        {"titulo":"Toy Story", "tipo":"movie"},
        {$push :
            {'detalles.actores':{$each:["Paco","Tom Hanks",
                                        "Paco","Paco2"]}}
                })
```

Como se puede ver, en este caso añade tanto los elementos que no están en el array como aquellos que están en el array "detalles.actores".

2.1.5.1.3 Métodos para cursores

Este tipo de métodos del shell de mongo, son utilizados junto con el método "find()" para modificar los resultados obtenidos por la consulta de lectura. Generalmente las consultas que incluyen estos métodos tienen la forma:

```
> db.coleccion.find({criterio de busqueda},{proyeccion}).funcionDeCursor\
    ();
```

Entre los diferentes métodos destacan:

- count(): hace que se devuelva como resultado el número de elementos que cumplen con el criterio de búsqueda.
- max()/min(): estos métodos no devuelven un documento con el valor máximo o mínimo en un campo, sino que devuelven los documentos con un valor mayor o menor a un valor dado para un campo, por lo que son equivalentes a los operadores $gt y $lt.
- limit(n): modifica el tamaño del cursor haciendo que devuelva más o menos resultados por cada iteración.
- skip(n): hace que se ignoren los n primeros documentos que cumplen el criterio de busqueda.
- sort(campo: valor/<funcion>): ordena los resultados en función de un campo del documento. Para ordenar de mayor a menor se usa como valor un -1 y si se quiere de menor a mayor un 1.
- forEach(<funcion>): permite realizar tareas por cada uno de los resultados que se obtienen de una consulta.

De los métodos anteriores forEach es el que da a los usuarios una mayor flexibilidad a la hora de realizar consultas de lecturas. Esto se debe a que con este método los usuarios con privilegios básicos de lectura y escrituras pueden realizar operaciones complejas sobre los datos, sin necesidad de utilizar una aplicación externa - otros métodos como eval() permiten ejecutar funciones pero requieren que el usuario tenga mayores privilegios. Por ejemplo, si un usuario decidiera que quiere crear una colección con los datos de los descuentos que hay en la colección "articulosD2", podría realizar una consulta que buscara todos los documentos y que por cada documento obtenido insertara un documento con los valores deseados en una colección llamada "promocionD2". La siguiente consulta realizaría dicha operación:

```
db.articulosD2.find({}).forEach(function(doc)
{
        db.promocionD2.insert({"_id":doc._id});
        var cont = 0;
        for(var i=0; i< doc.formato.length; i++){

                if(doc.formato[i].precio.descuento > 0){
                        cont++;
                        db.promocionD2.update(
                {"_id":doc._id},
                {$addToSet:{"descuento":
                                {"formato": doc.formato[i].formato ,
                                 'descuento':doc.formato[i].precio.\
    descuento}
                            }
                });
                }
        }
        if (cont ==0){
                db.promocionD2.remove({"_id":doc._id});
        }
})
```

El valor de entrada de la función (doc) corresponde con cada uno de los documentos obtenidos en la consulta.

A la hora de usar este método e implementar funciones hay que tener en cuenta que si en una colección hay documentos con distintos esquemas, si se utiliza un campo que está en unos documento pero no están en otros se puede producir un error. Ésto se debe a que cuando un campo no existe esté se considerará nulo, con lo que hacer operaciones con este valor dará errores.

Por ejemplo, suponiendo que el campo "precio" no está presente en todos los documentos; obtener el valor "doc.precio" devolvería null pero no error. Sin embargo, si se quiere obtener el valor de algún campo del subdocumento, por ejemplo "doc.precio.descuento", se produciría un error. Para solucionar esto sería suficiente con comprobar que "doc.precio" no es null antes de acceder a los campos del subdocumento.

2.1.5.2 Entorno de agregación

Además de las operaciones de escritura y lectura vistas anteriormente, MongoDB ofrece un entorno para realizar consultas con operaciones de agregación.

Para realizar este tipo de operaciones se utiliza el método "aggregate()". Este método a diferencia de "find()" no devuelve múltiples resultados sino que devuelve un único documento que contiene una lista con los documentos que cumplen la condición de búsqueda. Puesto que es un documento BSON tiene un límite de 16MB, si existen muchos documentos que cumplen con los criterios de búsqueda entonces es posible que se supere el tamaño del documento y se producirá un error.

La idea detrás de este entorno es que los documentos entran en una secuencia de fases que transforman dichos documentos en resultados agregados. Normalmente las consultas básicas tienen una primera fase de selección de documentos donde se buscan aquellos que cumplan un criterio de búsqueda y después una fase de agrupación y/o proyección donde se modifican los datos de los documentos. Siguiendo este patrón la estructura básica de una operación de agregación tendría la siguiente forma:

```
> db.coleccion.aggregate(
    {$match:{criterio de busqueda},
    {$group:{modificaciones de campos}},
    {$project:{campos que se incluyen}}
    }
)
```

Aunque éste es un ejemplo básico de una estructura que puede tomar una operación de agregación, éste no es un esquema fijo. La propia naturaleza del entorno basado en una secuencia de fases permite realizar diferentes combinaciones con dichas fases. Sería posible añadir nuevas fases como por ejemplo una fase de ordenación o repetir alguna de las fases anteriores - por ejemplo la de selección para filtrar los documentos modificados en la fase de agrupación -, o reordenar las diferentes fases para mejorar el rendimiento de la consulta.

2.1.5.2.1 Operadores

El método "aggregate()" permite el uso de los siguientes operadores para construir las distintas fases de la consulta y para realizar modificaciones sobre los documentos que se darán como resultado. Éstos operadores son los siguientes:

- $match : este operador permite realizar una fase de filtrado para seleccionar los documentos que cumplan un criterio de búsqueda. Este operador permite realizar una operación similar a la clausula "WHERE". También se puede considerar que es similar a "HAVING" puesto que se puede utilizar al mismo tiempo otros operadores para realizar comparaciones. Por ejemplo, para seleccionar los documentos con título "WALL-E": {$match{"titulo": "WALL-E"}}

- $project : este operador permite realizar una fase de proyección para seleccionar los campos que contendrán los resultados. Esta operación sería similar a la clausula "SELECT" de SQL. En esta fase también es posible renombrar campos o crear campos nuevos que contengan valores obtenidos a partir de otros campos, por ejemplo sumando los valores de dos campos de un documento. Por ejemplo, para obtener el precio de los documentos pero añadiendo cinco al precio original del artículo: {$project:{"precioTotal":{$add:["$precio.precio",5]}}}

- $group: este operador permite realizar una fase de agrupación de documentos en función de un campo y realizar operaciones de agregación - por ejemplo operaciones de suma - con los valores de los documentos de un mismo grupo. Esta operación es similar a la clausula "GROUP BY" de SQL. Por ejemplo, para agrupar artículos en función del título: {$group:{_id: '$titulo'}}

- $sort : este operador permite realizar una fase de ordenación de los documentos en función de uno o varios campos. Es posible indicar si se quiere ordenar de forma descendente o ascendente - con los valores 1 y -1 respectivamente. Esta operación es similar a la clausula "ORDER BY" de SQL. Por ejemplo, para ordenar los artículos en función del precio {$sort:{'precio.precio': 1}}

- $limit : este operador permite realizar una fase que limite el número de documentos a devolver. Esta operación es similar a la clausula "LIMIT".

- $unwind : este operador permite realizar una fase en la que se descompone un documento en varios documentos en función de los valores presentes en un campo array. Por ejemplo, tomando un documento de tipo libro del segundo diseño, si se utiliza el campo formato como parámetro de esta fase el resultado serían dos documentos donde cada documento solo tendría en el campo formato la información de uno de los elementos del array. {$unwind:"$formato"}

- $skip : este operador permite ignorar un número determinado de documentos.

En estas consultas también es posible utilizar los operadores lógicos, de comparación y aritméticos vistos en la sección sobre operaciones CRUD.

Por último, existen una serie de operadores específicos para utilizar en la fase de agrupación:

- $sum: este operador permite contar el número de elementos en un mismo grupo o para sumar los valores de un campo. Esta operación es similar a las clausulas "SUM" y "COUNT".

- $min, $max: estos operadores permiten devolver el valor mínimo o máximo de cada grupo de documentos. Esta operación es similar a las clausulas "MIN" y "MAX".
- $first, $last: estos operadores permiten devoler el primer y el último valor de cada grupo de documentos.
- $avg: este operador permite devolver la media de los valores de un campo por de cada grupo de documentos.
- $addToSet, $push : estos operadores permiten devolver un array con todos los valores de un campo para cada grupo de documentos. La diferencia entre $addToSet y $push es que en el primer caso no pueden aparecer valores repetidos pero en el segundo caso sí.

2.1.5.2.2 Secuencia de fases

Es posible organizar las fases en diferente orden y repetir fases, no solo para obtener un resultado que se acerque al deseado, sino también para mejorar el resultado de la consulta.

Por ejemplo, suponiendo que se quieren obtener las películas de la colección "articulosD2" - segundo diseño - con título "WALL-E" pero sólo en formato dvd. En este caso no es suficiente con hacer únicamente una fase de selección, ya que devolvería los artículos junto con los datos de todos los formatos. Por tanto, habría que hacer tres fases: una primera fase de selección para obtener las películas con nombre "WALL-E", una segunda fase de separación de los datos en función del formato, y una tercera fase de selección para elegir los artículos con formato "dvd". Por tanto la consulta sería la siguiente:

```
db.articulosD2.aggregate(
    {$match:{"titulo": "WALL-E"
              "tipo" : "Movie"}},
    {$unwind:"$formato"},
    {$match:{'formato.formato': "dvd"})
```

Otro ejemplo, usando la fase de agrupar sería por ejemplo, calcular el precio total de todos los artículos con precio mayor a 30, separados en función del tipo de artículos. En este caso primero se buscaría los documentos donde el precio sea mayor a 30 con una fase "$match"; y después se hace una fase de agrupación en función del "tipo", fase en la que también se suman los precios de los artículos. La consulta sería la siguiente.

```
db.articulos.aggregate(
{$match:{'precio.precio':{$gt:30}}},
{$group:{_id:"$tipo",
      total:{$sum:"$precio.precio"}}
```

```
});
```

2.1.6 Índices

En MongoDB los índices son guardados a nivel de colección y soporta cualquier campo en un documento o subdocumento. El índice guarda estos campos ordenados en función del valor del campo. Estos índices permiten limitar el número de documento que son revisados, y además como los valores están ordenados por el valor del campo, MongoDB usa los índices para devolver los documento si se se pide ordenar los resultados en función de dicho campo. Por otro lado, si los campos del documento proyectado coinciden con los campos de indexación, entonces devuelve los valores directamente desde los índices.

Por defecto, se crea un índice que hace referencia al campo "_id" el cual contienen todos los documentos en MongoDB, para poder crear otros índices se utiliza el método "ensureIndex({campos},{opciones})" donde se puede indicar el campo que se quiere utilizar y si se quiere ordenar de forma ascendente o descendente asignando al campo los valores 1 y -1 respectivamente. Por ejemplo, para crear un índice en la colección "articulos" en función del título del artículo y darle el nombre "indTitulo" al índice, se haría lo siguiente:

```
db.articulos.ensureIndex({"titulo":1},{name:"indTitulo"});
```

Darle nombre al indice, al igual que el resto de opciones, es opcional.

Por otro lado, para eliminar un índice se utiliza el método "dropIndex()". Esta función permite borrar los índices bien indicado los campos que forman el índice o indicando el nombre que se le dio al índice el crearlo. Por ejemplo, para borrar el índice anterior, es posible borrar de dos formas:

- Indicando el nombre: db.articulos.dropIndex("indTitulo");
- Indicando los campos: db.articulos.dropIndex({"titulo":1});

2.1.6.1 Tipos de índices

En esta sección se muestran los diferentes tipos de índices que se pueden utilizar en MongoDB.

2.1.6.1.1 Índices de un sólo campo

Estos índices están formados por un único campo, estos campos pueden ser tanto campos simples de un documentos - de tipo entero, string... - como campos de un subdocumentos, o campos que contienen un subdocumentos. Ya se ha visto un ejemplo del primer caso cuando se creo un índice para el campo "titulo". En los otros dos casos, la diferencia es que el primer caso sólo se utiliza el campo de un documento embebido en otro mientras que en otro caso se utiliza el documento embebido para crear el índice.

Por ejemplo, para la colección "articulos", si se quisiera crear un índice para el campo "duracion" de subdocumento que contiene el campo "detalles", se haría la siguiente operación:

```
> db.articulos.ensureIndex({"detalles.duracion":1});
```

Sin embargo, si se quisiera crear, para la misma colección, un índice sobre el subdocumento en el campo precio, este índice se crearía se haría de la siguiente forma:

```
> db.articulos.ensureIndex({"precio":1});
```

2.1.6.1.2 Índices compuestos

Estos índices son creados a partir de varios campos. El número de campos que puede haber en único índice es de 31 campos, a pesar de esto, sólo es posible incluir un único campo de tipo array por cada índice.

El orden de los campos es importante, especialmente si se van a realizar consultas que ordenen los resultados en función de alguno de esos campos, ya que el índice ordena los valores empezando por el primer campo, y después ordena los documentos que tienen el mismo valor para el primer campo a partir del segundo campo - y así sucesivamente.

Además, MongoDB es capaz de utilizar los primeros campos del índice como si el índice sólo estuviera formado por estos, es decir, si se hace una consulta con los dos primeros campos de un índice formado por tres o más campos, MongoDB puede utilizar el índice para obtener los resultados, por lo que un índice a parte con únicamente los dos primeros campos no sería necesario.

Para crear un índice compuesto se haría la misma operación que en el punto anterior, salvo que se añadirían varios campos.

2.1.6.1.3 Índices con valor múltiple

Estos índices son creados cuando el campo utilizado es un campo cuyo valor es de tipo array. Estos arrays pueden ser tanto de valores simples como de subdocumentos.

Este tipo de índices soporta índices compuestos, es decir, pueden crease junto a otros campos, sin embargo, sólo se puede usar un campo array por índice.

Para crear estos índices se utiliza el misma método y no es necesario configurar ninguna opción extra.

2.1.6.1.4 Índices "text"

Este tipo de índices permite realizar búsquedas de palabras dentro de un campo de tipo string, es decir, si por ejemplo se quieren buscar los títulos que contengan la palabra "java", además, puesto que este tipo de índices no distingue entre mayúsculas y minúsculas, se pueden realizar evitar el uso de expresiones regulares. Es posible utilizar más de un campo para crear el índice.

Para crear este tipo índices también se utiliza el método "ensureIndex({campo:valor})", salvo que en este caso en lugar de usar como valor un uno o un menos uso, se utiliza el string "text". Así por ejemplo, para crear un índice sobre el campo "titulo" y el campo "tipo", de los cuales el primer campo es sobre el que se quiere realizar la busqueda de texto, entonces se debería realizar la siguiente operación:

```
> db.articulos.ensureIndex({"titulo":"text","tipo":1});
```

Una de las peculiaridades de este tipo de índices, es que puede saltarse ciertas palabras al realizar la indexación, las cuales son palabras que se encuentran de forma repetida en dicho lenguaje, pero que rara vez son utilizadas para realizar búsquedas. Por ejemplo, en inglés una de estas palabras sería el artículo "the".

Finalmente, este tipo de índices sólo es utilizado al realizar búsquedas de texto, las cuales requieren una serie de configuraciones y son realizadas mediante el método "runcommand()".

2.1.6.1.4.1 Configuración del servidor

Para que el servidor permita realizar búsquedas de texto es necesario que la opción "textSearchEnabled" del archivo de configuración "MongoDB.conf", esté a "true" o bien si se inicia el servidor desde "mongod" ejecutar desde el terminal:

```
> mongod --setParameter textSearchEnabled=true
```

2.1.6.1.4.2 Buscar texto

Para realizar este tipo de búsquedas se utiliza el método "runCommand("text",{search: "valor"})". El valor de búsqueda es un string y la sintaxis de este hará que en la búsqueda se realicen diferentes tareas. La diferentes opciones son las siguientes:

- Buscar un única palabra: el string estará formado únicamente con la palabra.
- Buscar una u otra palabra: el string tiene la forma "palabra1 palabra2", permite buscar títulos que contenga la palabra1, la palabra2 o ambas; es decir, equivaldría a un "or" entre las palabras separadas por el espacio.
- Buscar un frase exacta: el string estará recogido entre comillas y tendrá la forma:

```
"\"esto es una frase\""
```

- Buscar una palabra pero que el texto no contenga otra: el string tendrá la forma "buscar -noBuscar".

Además, este tipo de búsqueda permite realizar proyecciones de los datos y filtrar los documentos en función de otros campos. Por ejemplo, para buscar un artículo que contenga la palabra "java", que sea de tipo libro y que solo devuelva el titulo se haría la siguiente operación:

```
> db.articulos.runCommand("text",{search: "java",
                                  filter: { tipo : "book" }
                                  project: { "titulo": 1 }})
```

2.1.6.1.4.3 Uso de pesos

Con este tipo de índices es posible dar pesos a cada campo de texto al crear el índice, para que cuando se realice la búsqueda, los resultados se ordenen de forma que aquellos documentos en los que se encuentra el valor en el campo de mayor peso, aparezcan primero.

Por ejemplo, si se quiere buscar una palabra que pueda formar parte de un título o del nombre de un autor, pero que de prioridad a los documentos en los que el valor se encuentra en el título se crearía el índice de la siguiente manera:

```
>  db.articulos.ensureIndex({titulo:"text","detalles.autores":"text"},
                            {weights: {titulo: 10, "detalles.autores":5}
                            );
```

2.1.6.1.5 TTL

Más que un índice en si, es una opción disponible al crear los índices llamada "expireAfterSeconds". Esta opción permite que el documento de una colección que contenga un cierto campo, será eliminado de la colección pasado un cierto tiempo tras la creación del documento. El campo sobre el que se crea el índice ha de ser de tipo "Date", el cual se utiliza para comprobar el tiempo que ha pasado desde que se creo el documento para ver si se ha superado el tiempo de caducidad. Este permite que si se modifica el campo con una nueva fecha, el tiempo que le quede al documento por ser eliminado se reinicie.

Por ejemplo, si se quiere crear una colección que contenga documentos con cestas de compra los cuales se borren cada cierto tiempo se debería crear un índice de la siguiente manera:

```
> db.carritos.ensureIndex( { "caducidad": 1 },
                           { expireAfterSeconds: 3600 });
```

El borrado de los índices se hace cada minuto por lo que el tiempo de borrado no es exacto. Esto hace que si bien este método es lo suficientemente exacto para casos como el del ejemplo anterior, puede que no lo sea para otros casos en los que se necesite una mayor precisión, para esos casos sería mejor hacer la eliminación a nivel de aplicación.

2.1.6.1.6 Sparse

Este tipo de índices es como los índices normales, salvo que no incluyen en el índice aquellos documentos no contienen el campo utilizado para crear el índice. La ventaja de estos índices es que reducen el espacio de los índices permitiendo así que sea más fácil que el índice entre en la memoria.

Para crear un índice de este tipo es suficiente con añadir la opción { sparse: true } .

2.1.6.2 Diseño de índices

Los índices permiten que las consultas de lectura sobre ciertos campos se realicen sobre con mayor prioridad. Sin embargo, un gran número de índices puede provocar que ciertas consultas de escritura se ralenticen debido al hecho de que cada vez que se inserta un documento o se modifica un campo que hace referencia a un índice, los índices también se deben actualizar.

Por tanto, teniendo en cuenta la restricción anterior y las características de los índices vistas en el apartado sobre índices de este documento, se plantean los siguientes índices

para cada diseño, suponiendo que las consultas realizadas son similares a las de un comercio electrónico:

- Diseño 1:
  - Búsqueda por título, actores, autores, directores, guionistas o género: sería común realizar consultas en las que se buscan los datos en función de estos campos, bien directamente sobre uno de los campos o en consultas que contienen el operador $or - las búsquedas que se realizan con este operador son en paralelo, es decir, equivaldría a realizar una consulta por cada uno de los campos y ejecutarlas en paralelo- por lo que sería recomendable crear un índice sobre cada uno de los campos.

    También es común que en estas búsquedas se quiera filtrar los artículos en función del "tipo" y/o formato del artículo. Puesto que, como se comento anteriormente, es posible utilizar los índices compuestos para realizar búsquedas que utilicen únicamente los primeros campos del índice, no sería necesario, por ejemplo, crear tres índices para título ya que un índice que contenga los campo {título,tipo,formato} permite realizar búsquedas con sólo el campo título o título y tipo.

```
db.articulos.ensureIndex({"titulo":1,"tipo":1,"formato":1});
db.articulos.ensureIndex({"genero":1,"tipo":1,"formato":1});
db.articulos.ensureIndex(
        {'detalles.autores':1,"tipo":1,"formato":1},
        {sparse: true});
db.articulos.ensureIndex(
        {'detalles.actores':1,"tipo":1,"formato":1},
        {sparse: true});
db.articulos.ensureIndex(
        {'detalles.director':1,"tipo":1,"formato":1},
        {sparse: true});
db.articulos.ensureIndex(
        {'detalles.guionista':1,"tipo":1,"formato":1},
        {sparse: true});
```

    Además, puesto que no todos los documentos contienen los campos actores, autores, directores y guionistas, lo ideal los índices para estos casos tuvieran la opción "sparse" a "true" para no incluir los documentos que no tienen estos campos en los índices.

  - Ordenar sobre el precio: generalmente se ordena tras realizar alguna de las búsquedas del caso anterior, por lo que la mejor opción sería añadir un campo extra al final al crear los índices anteriores, por ejemplo para el caso

de los actores: db.articulos.ensureIndex({'detalles.autores':1, "tipo":1, 'precio.precio':1}, {sparse: true});

- Diseño 2:
  - Búsqueda por título autores directores guionistas o género: al igual que en el caso anterior interesa crear un índice por cada uno de los campos anteriores y que además contenga el campo tipo al final. Si bien, es cierto que se puede hacer un filtrado de los artículo con ese formato, normalmente se haría después de una fase {$unwind}, o a nivel de aplicación, por lo que añadir este campo al índice únicamente tendría efecto si existen muchos casos en los que el libro o película puede estar en un formato pero no en otro. Sin embargo, este no sería el caso del dataset de prueba puesto que todos tienen los artículos de un mismo tipo, tienen los mismos formatos, por tanto no sería necesario añadir este campo al índice.

    Además, puesto que no todos los documentos contienen los campos actores, autores, directores y guionistas, lo ideal sería para estos casos los índices tuvieran la opción "sparse" a "true" para no incluir los documentos que no tienen estos campos en los índices.

```
db.articulosD2.ensureIndex({"titulo":1,"tipo":1});
db.articulosD2.ensureIndex({"genero":1,"tipo":1});
db.articulosD2.ensureIndex({'detalles.autores':1,"tipo":1},
                           {sparse: true});
db.articulosD2.ensureIndex({'detalles.actores':1,"tipo":1},
                           {sparse: true});
db.articulosD2.ensureIndex({'detalles.director':1,"tipo":1},
                           {sparse: true});
db.articulosD2.ensureIndex({'detalles.guionista':1,"tipo":1},
                           {sparse: true});
```

  - Ordenar sobre el precio: de nuevo, al igual que en el caso anterior sería conveniente añadir a los índices el campo del precio. Sin embargo, puesto que el campo que contiene el precio es un índice, únicamente se podrá utilizar con el campo título, ya que el resto de los campos - actores, autores, directores y guionistas - son de tipo array. Para solventar esto, se podría crear un índice extra para el tipo y el precio y realizar las consultas con dos fases de selección "$match".

```
db.articulosD2.ensureIndex({"titulo":1,"tipo":1,
                            'formato.precio.precio':1});
db.articulosD2.ensureIndex({"tipo":1,'formato.precio.precio\
    ':1});
```

- Índice de tipo texto: como se ha dicho antes sería común realizar una búsqueda sobre los campos título, autores, directores o guionistas, donde normalmente en el caso del título se querrán obtener artículos que no coinciden exactamente con el título dado, por lo que crear un índice de tipo texto puede ser una alternativa a los índices anteriores, reduciendo así el número de índices presentes.

2.2 OpenLink Virtuoso

OpenLink Virtuoso [20] es un sistema híbrido que integra funcionalidades de servidor Web y de ficheros; y de Bases de datos, todo ello en un *Servidor Universal*. En lo que a las bases de datos se refiere, Virtuoso ofrece dos sistemas internos de almacenamiento - relacional y rdf - y acceso a sistemas externos de bases de datos relacionales - MySQL, Oracle... - de forma transparente.

Para el manejo de bases de datos, el sistema tiene dos motores. Por un lado el motor principal el cual permite manejar un modelo de datos relacional nativo; y por otro lado el motor de bases de datos virtuales (VDB) para manejar las bases de datos ajenas al sistema [21]. Además, de lo referente a bases de datos, Virtuoso también tiene un servidor de aplicaciones el cual cumple dos funciones, por un lado dar a los usuarios acceso a herramientas y servicios para manejar y realizar consultas sobre las bases de datos; y por otro lado permitir a los usuarios usar el servidor para desplegar sus propias aplicaciones y servicios que pueden o no estar relacionados con las bases de datos.

En cualquier caso, en lo que se refiere al almacenamiento de datos RDF, Virtuoso guarda estos datos - tripletas - en el sistema de bases de datos relacional nativo, guardando - a diferencia de MongoDB - los datos directamente en tablas y no en estructuras datos específicas. Dos aspectos que distinguen el uso de Virtuoso como almacenamiento de datos RDF del almacenamiento de datos relacional son: (1) que los datos estén almacenados en una tabla es totalmente transparente para el usuario, el cual ve los datos en forma de grafos; (2) que las tablas donde se almacenan los datos están especialmente preparadas para ser utilizadas en el almacenamiento de datos RDF teniendo, por ejemplo, índices por defecto específicos para usar con este tipo de datos.

Para realizar consultas sobre los datos RDF, Virtuoso soporta el uso del lenguaje SPARQL desde las diferentes interfaces de usuario. La sintaxis de este lenguaje es muy similar a SQL y al igual que SQL, no es un lenguaje especifico de Virtuoso sino que es utilizado de forma general al tratar con este tipo de datos.

A nivel del almacenamiento de tripletas, las características principales que se pueden encontrar en el sistema son las siguientes:

- Consultas: para realizar consultas sobre el depósito de tripletas Virtuoso soporta el lenguaje *SPARQL*. Este es un lenguaje similar a *SQL* salvo que está orientado al uso con grafos. Al igual que ocurre con *SQL*, no es un lenguaje propietario de Virtuoso sino que es un lenguaje estandarizado, para el uso con grafos RDF.

- Índices: aunque permite el uso de índices sobre el deposito de tripletas, estos son algo especiales. Los índices no se crean a nivel de grafos, sino que se crean para toda la tabla que contiene las cuartetas - el grafo más las tripletas.

- Transacciones y concurrencia: el depósito de tripletas soporta el uso de transacciones y cumple con el principio ACID. Además, los bloqueos se hacen a nivel de tupla sobre la tabla de cuartetas con lo que será también a nivel de tripleta.

Por último, para realizar consultas a nivel de aplicación Virtuoso ofrece drivers para diferentes lenguajes como Java o C/C++. En el caso de Java ofrece un driver *JDBC*, pero además ofrece un proveedor para el *framework de web semántica JENA*, permitiendo usar este entorno para realizar consultas de forma más sencilla que con el driver *JDBC*. Sin embargo, hay que destacr que el uso del proveedor puede hacer que las consultas tarden más en realizarse [22].

2.2.1 Grafos

Las bases de datos RDF, guardan los datos en forma de grafos los cuales están formados por nodos y las relaciones entre los nodos. Estos nodos y sus relaciones viene representadas mediante las llamadas tripletas que tienen la forma <nodo1 relación nodo2>, en general el nombre que se les da a los campos de las tripletas es: sujeto S, predicado P, y objeto O; donde el sujeto y el objeto son los nodos y predicado la relación que existe entre ambos. Los nodos pueden ser tanto valores simples - nombres o números -, como otros subgrafos - generalmente se índica guardando como valor un URI. Las relaciones normalmente son URI's que indican como se relacionan los nodos. Por ejemplo, si se tienen dos nodos, uno con el nombre de un usuario y otro con el identificador único del usuario; una posible relación entre los dos datos es "tiene_nombre", pero como en RDF este tipo de relaciones se representa con URI's una opción seria llamar a la relación "http://grafo.com/user/name". La tripleta final de este ejemplo tendría la forma: <ID_usuario http://grafo.com/user/name nombre>. Mediante un conjunto de estas tripletas es posible representar grafos; ya que los objetos que son relacionados por P pueden ser tanto valores simples como identificadores de otros conjuntos de tripletas - subgrafos.

Al igual que ocurre con los documentos, los grafos permiten representar datos con varios niveles de anidación donde cada nivel viene dado por un subgrafo. Por otro lado, el esquema para representar a los objetos también es flexible permitiendo que, por ejemplo, los datos de una persona no tengan las mismas propiedades que las de otra persona. Sin embargo, en lo que se refiere a los tipos de datos, los valores simples pueden ser de una gran variedad de tipos - enteros, reales, conjuntos de caracteres... - pero no es posible guardar una lista de elementos en una única tripleta. En su lugar, cada uno de los elementos de la lista de elementos se representa como una nueva tripleta.

2.2.2 Almacenamiento de las tripletas

Como se ha comentado antes, Virtuoso guarda directamente los datos RDF en una serie de tablas del motor de bases de datos relacionales. En concreto estos datos son guardados en la tabla DB.DBA.RDF_QUAD, [23] , la cual contiene cuatro columnas <G,S,P,O>. Cada tripleta es respresentada por una fila de la tabla, donde no sólo se guardan las tripletas que representan a los datos, sino que también se guarda el grafo al que pertenece dicha tripleta.

Generalmente, los datos de los campos G,S,P - los cuales suelen ser URI's - no se almacenan directamente en los campos de la tabla, en su lugar en la tabla se almacena lo que se llama un IRI_ID, que es un identificador interno, a partir del cual se podrá obtener el valor real del campo. En el caso del campo O , siempre que el valor no es una URI - hace referencia a un subgrafo -, se guarda el valor directamente en la tabla.

Además, de esta tabla, también son de destacar las tablas DB.DBA.RDF_PREFIX y DB.DBA.RDF_IRI, las cuales almacenan las relaciones entre el identificador interno y el valor real.

2.2.3 Uso del sistema

Virtuoso ofrece varias opciones para acceder al sistema y realizar diferentes tareas. En las siguientes secciones se muestran varias de ella.

2.2.3.1 Interfaz Web

Esta herramienta es una interfaz gráfica para el sistema que permite realizar tareas administrativas, automatizar operaciones, crear usuarios, ejecutar comandos SQL...

A este interfaz se puede acceder desde un navegador web en la dirección http://localhost:8890/ , una vez ahí para entrar en el sistema se debe seleccionar el link "conductor" de la barra

lateral izquierda, o directamente acceder desde la dirección http://localhost:8890/conductor para entrar en la zona de administración del sistema llamada "conductor". Por otro lado, si se quiere acceder de forma remota a la instancia del servidor se deberá sustituir en la dirección anterior la palabra "localhost" por la dirección ip correspondiente al servidor.

Hasta que se creen más usuarios, para poder utilizar la interfaz es necesario identificarse como dba con la contraseña que se dio durante la instalación del sistema.

Dentro de la zona de administración, en el menú superior se muestran las diferentes opciones para manejar los diferentes aspectos del sistema. Concretamente las herramientas para el manejo del almacenamiento RDF se encuentran en "Linked Data".

2.2.3.2 Línea de comandos

Junto con la instalación del sistema, se instala la herramienta "isql", la cual permite realizar conexiones a las bases de datos y ejecutar sentencias SQL desde la línea de comandos.

Para lanzar la herramienta, desde el terminal ejecutar: isql-vt

Una vez hecho esto, al realizar una consulta se solicitará la contraseña del usuario dba si el usuario no estaba conectado. Además, también es posible lanzar la herramienta indicándole una serie de opciones como -U y -P - en mayúsculas - para indicar el usuario y la contraseña. Por ejemplo, para iniciar una sesión con el usuario dba:

```
>isql-vt -U dba -P tab053
```

Si se quiere acceder a un sistema que se encuentra en un servidor remoto se pueden usar las opciones -H y -S e indicar la dirección IP y el puerto de escucha de servidor remoto. Se puede acceder a una lista más extendida de estas opciones mediante la opción –help.

Mediante esta herramienta es posibles realizar tanto consultas SQL sobre las base de datos relacionales que haya en el sistema - incluyendo aquellas que contiene información sobre el sistema - como consultas SPARQL - siempre y cuando se tengan los permisos necesarios. Para realizar consultas SPARQL desde esta herramienta es necesario incluir la palabra reservada SPARQL antes de la consulta.

2.2.4 ¿Porqué Virtuoso?

Una de las principales razones para elegir un sistema de bases de datos orientado a grafos en lugar de un sistema relacional para guardar ciertos datos que estén muy relacionados;

es porque como se comento con anterioridad los sistemas relacionales no están adaptados a este tipo de datos y cuando se quieren obtener datos en función de sus relaciones con otros datos las consultas pueden llegar a ser muy complejas.

En cuanto al uso de un sistema de grafos en lugar de otro tipo de sistemas, esto se debe principalmente al rendimiento de este sistema. Por ejemplo, es posible utilizar MongoDB como un deposito de tripletas utilizando los documentos; sin embargo, el rendimiento de MongoDB frente a Virtuoso es bastante bajo [7].

En cuando a la elección de Virtuoso en lugar del resto de sistemas se debe principalmente al hecho de que en general obtiene unos mejores resultados en el rendimiento que el resto de sus competidores [24].

2.2.5 Modelado de datos

Las bases de datos RDF, guardan los datos en forma de grafos, donde cada grafo contiene información en forma de tripletas, estas tripletas representan información con la forma <Sujeto, Predicado, Objeto>, y dicha información representa relaciones entre los datos. Por ejemplo, para guardar las contraseñas de los usuarios en una base de datos relacional se podría usar una tabla que contenga el identificador de usuario junto con la contraseña, pero para guardar esta información en un grafo no solo se guardaría el identificador del usuario y la contraseña sino también la relación entre estos dos datos. Esta relación tendría por ejemplo la forma:

```
"user43@email.com" user:pwd "password"
```

En lo que se refiere a Virtuoso, como se ha indicado anteriormente, éste guarda los grafos en tablas con cuatro columnas - cuartetos. Sin embargo, esto no afecta al modelado de los datos ya que las operaciones que se realizan sobre los grafos se hace con "SPARQL" como si se tratarán de tripletas en grafo, es decir, la estructura que utiliza Virtuoso es totalmente transparente al usuario. La única característica que puede ser de interés es que puesto que todos los grafos se guardan en la misma tabla y debido a los índices de esta tabla - de los que ya se hablará más adelante - el tener un grafo con muchos datos o muchos grafos con pocos datos, en general, no afecta al rendimiento, crear los grafos de una forma u otra solo dependerá del uso que se haga de los datos.

Por ejemplo, si se quiere crear un blog en el que colaboran varios usuarios, una forma sencilla de organizar los datos y a los usuarios sería tener un grafo por usuario donde se guarde toda la información de los artículos que escriben. Este modelo con un grafo por usuario, permite ,por ejemplo, que se den permisos a estos usuarios sobre sus grafos para que sólo ellos puedan modificarlos.

Por otro lado, si se tiene un gran número de usuarios con poca información pero altamente relacionados, como sería una base de datos para guardar los gustos de los usuarios de un comercio electrónico, una mejor opción sería crear un único grafo que contenga toda la información. Si bien, en general, esto no supondrá una mejora en el rendimiento, permitirá simplificar las consultas sin empeorar tampoco el rendimiento.

2.2.5.1 Diseño Básico de un grafo

Como se ha dicho antes, un grafo rdf contiene una serie de tripletas. Lo que estás tripletas - S,P,O - representan son dos nodos <S,O> unidos mediante el arco P. Este arco implica una relación entre los dos nodos. En un modelo RDF los grafos y subgrafos se representan con una URI único para la base de datos, y además los predicados P que representan las relaciones también se representan con URI - aunque no es necesario que sean únicos.

En este caso los nodos pueden ser tanto valores simples como subgrafos. Por ejemplo, para representar a un usuario cuya información es su nombre, su contraseña y que conoce a otro usuario - suponiendo que se guarda a los usuarios en un mismo grafo, se puede considerar a cada usuario como un subgrafo - la representación en tripletas sería la siguiente:

```
<http://user.graph/userID> <URI/password> "123"
<http://user.graph/userID> <URI/nombre> "Mark"
<http://user.graph/userID> <URI/conoce> <http://user.graph/userID2>
```

Donde <http://user.graph/userID2> sería un subgrafo con la información del userID2.

A la hora de diseñar la relación entre grafos existen varias opciones. Por un lado, se puede dividir el grafo en varios grafos o bien agrupar los datos en un único grafo de forma que ciertos nodos objeto O sean subgrafos. La diferencia principal entre las dos opciones anteriores se dará al realizar la consulta, ya que será necesario utilizar los dos grafos. En cuanto al rendimiento, como se comentó con anterioridad debido al esquema de Virtuoso, en general, no se ve afectado por el uso de uno u otro esquema.

Por ejemplo, suponiendo que se quisiera guardar información sobre los usuarios de un sitio web y los artículos que les han gustado. En este caso habría varías formas de representar los datos.

Una primera forma sería guardar la información de los usuarios en un grafo <http://users .graph>, y otro para guardar los artículos <http://articles.graph>. Los dos grafos se podrían relacionar con una tripleta del tipo:

```
<http://users.graph/userID> URL:likes <http://articles.graph/articleID>
```

Una segunda forma es usar subgrafos, con lo que todos los datos se guardarían en un mismo grafo. En este caso una tripleta que relacionaría a los usuarios con los artículos que les gustan tendría una forma similar a la anterior, salvo que las URI de los subgrafos tendrán el mismo prefijo para los dos casos, seguidos de un identificador del subgrafo. Por ejemplo, si el grafo fuera <http://grafo.graph>, la tripleta podría ser la siguiente:

```
<http://grafo.graph/users/userID>
        URL:likes
            <http://grafo.graph/articles/articleID>
```

Donde userID y articleID son identificadores únicos de los subgrafos.

2.2.5.2 Diseño de predicados

Como se indicó al comienzo de la sección, los predicados en RDF generalmente se representan mediante una URI. A la hora de diseñar la forma de esta URI existen dos opciones, por un lado crear una URI personalizado que represente la relación que se quiera, o por otro lado utilizar las URI presentes en namespace predefinidos como foaf. Estos namespaces pretenden estandarizar las URI de ciertas relaciones que se presentan con frecuencia, como es el nombre de una persona o las relaciones de jerarquía entre objetos.

Algunos de los namespaces más comunes son:

- rdf: permite representar relaciones de jerarquía como "rdf:subclasOf" para indicar que un objeto es subclase de otro.
- foaf: permite representar propiedades de una persona como "foaf:name" para indicar que el nombre de un sujeto S es el objeto O.

Como se puede observar se utiliza la sintaxis "namespace:propiedad", esto se debe a que en RDF es posible asignar un alias a una URI y utilizarlo de la forma anterior para simplificar la sintaxis. Para asignar el alias se utiliza la clausula PREFIX.

Por ejemplo, para representar el ejemplo del punto anterior con lo visto en este punto se podría hacer lo siguiente:

```
PREFIX user: <http://user.graph/>
PREFIX datos: <http://user.graph/datos>
user:userID datos:password "123"
```

```
user:userID foaf:name "Mark"
user:userID foaf:knows user:userID2
```

Como se puede ver foaf no se ha definido con PREFIX. Esto se debe a que en Virtuoso los namespaces anteriores vienen definidos por defecto, por lo que al insertar o consultar tampoco será necesario definirlos.

2.2.5.3 Diseño del esquema de pruebas

Para utilizar como ejemplo se ha diseñado un esquema de datos y se ha creado el dataset siguiendo este esquema. Este esquema, representa a los usuarios de un comercio electrónico y a sus relaciones con los productos, las compras y las valoraciones que han realizado. Todos estos datos se guardan en un único grafo donde se han definido subgrafos para los usuarios, los articulos, las compras y las valoraciones.

El esquema de cada uno de los subrafos es el siguiente:

- Articulos art: <http://appl.com/art/> : contiene información sobre el identificador del artículo en mongodb, el título, el tipo de artículo y los formatos disponibles.

```
art:id <http://appl.com/art/article/id>   "idMongo"
art:id  <http://appl.com/art/article/title> "Titulo"
art:id  <http://appl.com/art/article/type>  "movie/book"
art:id  <http://appl.com/art/article/format> "formato"
```

- Compras buy:<http://appl.com/purchase/>: contiene información sobre el artículo comprado, la fecha, precio, y el formato.

```
buy:id <http://appl.com/purchase/buy/article>  <http://appl.com/art\
    /id>
buy:id <http://appl.com/purchase/buy/date>      "31/08/2013"
buy:id <http://appl.com/purchase/buy/price>      57
buy:id <http://appl.com/purchase/buy/formato>   "formato"
```

  Como se puede ver el nodo objeto de la primera tripleta es el subgrafo de un artículo.

- Valoraciones rev:<http://appl.com/rev/idReview>: contiene información sobre el artículo valorado, la opinión sobre el producto, la valoración y el formato.

```
rev:id <http://appl.com/rev/review/of>  <http://appl.com/art/id>
rev:id <http://appl.com/rev/review/description> "Descripcion"
rev:id <http://appl.com/rev/review/rating>       6
rev:id <http://appl.com/rev/review/formato>     "Formato"
```

Como se puede ver el nodo objeto de la primera tripleta es el subgrafo de un artículo.

- Usuarios <mailto:userID@email.com>: contiene el nombre del usuario, la contraseña y las acciones sobre los productos.

```
<mailto:userID@email.com> <http://appl.com/users/pwd>          \
    "123"
<mailto:userID@email.com> foaf:name                           "\
    Nombre"
<mailto:userID@email.com> <http://appl.com/actions/like>       art:\
    id
<mailto:userID@email.com> <http://appl.com/actions/reviewing>  rev:\
    id
<mailto:userID@email.com> <http://appl.com/actions/buying>     buy:\
    id
```

En este caso las tres últimas tripletas indican que le usuario ha hecho una acción <http://appl.com/actions/> y qué acción es la que se ha realizado - like, buying, reviewing.

2.2.6 Diseño de operaciones

Para realizar operaciones sobre los datos RDF, en Virtuoso se puede utiliza el lenguaje SPARQL. Este lenguaje tiene una sintaxis con cierto parecido a SQL, pero que está orientada a la consulta de información guardada en forma de grafos. Además, también existen funciones predeterminadas que permiten realizar ciertas operaciones.

2.2.6.1 Operaciones de Lectura SPARQL

En esta sección se van a mostrar la operaciones básicas de lectura que se pueden realizar con SPARQL en Virtuoso sobre los datos de prueba que siguen el modelo diseñado en el puento anterior.

2.2.6.1.1 SELECT

La sintaxis básica para realizar lecturas sobre grafos es la siguiente:

```
SELECT *
FROM <grafo>
WHERE {?s ?v ?p}
```

En la consulta anterior se pueden distinguir tres partes:

1. SELECT: permite , al igual que ocurren en SQL, realizar una proyección de los atributos que se quieren elegir.

2. FROM: permite elegir el grafo desde sobre el cual se desea realizar la consulta. Este es similar a la clausula FROM de SQL, salvo que en lugar de consultar sobre una tabla, se hace sobre un grafo.

3. WHERE: permite indicar criterios de búsqueda sobre datos del grafo que se quieren obtener como resultado de la consulta. En este caso las variables que comienzan por "?", indican que se quiere obtener cualquier valor para ese campo.

Por ejemplo, para obtener toda la información referente a un usuario concreto, sobre el dataset de pruebas se haría la siguiente consulta:

```
> SELECT *
  FROM <http://appl.com/>
  WHERE{ <mailto:user54@email.com> ?o ?p}
```

2.2.6.1.2 GRAPH

Para filtrar en función de los grafos, se puede utilizar "GRAPH" dentro de las condiciones de filtrado, de forma que en lugar de consultar a un grafo en concreto - como ocurría con la clausula "FROM"- se consulte en todos los grafos que cumplen ciertas condiciones.

Por ejemplo, para obtener la lista de todos los grafos disponibles es posible realizar la consulta:

```
SELECT DISTINCT ?g
WHERE
  { GRAPH ?g {?s ?p ?t}}
```

Como se puede ver, en esta consulta se utiliza GRAPH para filtrar los datos en función del grafo, y mediante el uso de la variable "?g" se indica que puede ser cualquier grafo. Como se proyectan los resultados de forma que únicamente se obtengan los valores de la variable "?g", el resultado de la consulta sólo contendrá los nombres de los grafos.

Alternativamente, para realizar esta consulta se puede utilizar la función:

```
DB.DBA.SPARQL_SELECT_KNOWN_GRAPHS();
```

2.2.6.1.3 FILTER

En los casos anteriores, los criterios de búsqueda de la consulta se limitan a indicar si puede coincidir con cualquier valor o debe coincidir con un valor en concreto. Sin embargo, es posible filtrar los datos obtenidos en la consulta mediante la opción FILTER.

La sintaxis sería de la siguinete forma:

```
SELECT *
  FROM <grafo>
  WHERE {
    ?s ?v ?p
    FILTER (<condicion de filtrado>)
  }
```

Un ejemplo más concreto sería por ejemplo intentar buscar todas las compras cuyo precio es menor a un valor concreto. Para esto, se realizaría la siguiente consulta:

```
SELECT *
FROM <http://appl.com/>
WHERE  { ?i <http://appl.com/purchase/buy/price> ?price.
         FILTER (xsd:integer(?price) < 30 )
         }
```

Además de la opción FILTER, es necesario pasar el valor del campo ?price a entero para que sea capaz de realizar la comparación correctamente. Para esto se utiliza el método ya implementado "xsd:integer()".

Las condiciones que se pueden especificar para realizar el filtrado son muy diversas [25], entre ellas podemos encontrar:

- Operadores de comparación aritmética, como en el ejemplo anterior.Pueden ser de tipo mayor, menor igual... Estás comparaciones no sólo se pueden realizar sobre valores numéricos, sino también sobre otro tipo de datos como fechas.
- Operadores de comparación sobre campos de tipo string mediante el uso de la opción "regex". Esta opción sería similar al LIKE de SQL y permite comparar strings con expresiones regulares. Por ejemplo, para seleccionar los artículos cuyo título contiene "Potter", se indicaría mediante "FILTER regex(?titulo,"Potter","i")" - donde i al igual que en MongoDB indica que no se tengan en cuenta mayúsculas ni minúsculas.
- Operadores de test como "bound()", isURI(), isBLANK, isLITERAL() los cuales permiten comprobar si el campo contiene algún valor que pueda ser una URI, vacío o un literal.

- Otros operadores de comparación como SameTerm que comprueba que dos campos contengan el mismo termino, datatype comprueba que el campo sea de tipo indicado.

Además de los operadores de comparación existen otros operadores que dan gran flexibilidad a este tipo de filtrado. Por un lado se tiene la opción de utilizar operadores lógicos de la forma && y || - and y or respectivamente - para concatenar condiciones. También es posible el uso de operadores matemáticos para realizar operaciones antes de comparar valores. Y otros operadores propios de Virtuoso como str que obtiene el literal de un campo de tipo URI; o como lang que obtiene el idioma del campo - siempre que este venga indicado en el campo con la forma valor@EN, siendo EN el idioma inglés.

Un ejemplo de consulta que utiliza varios de estos operadores sería la siguiente:

```
SELECT *
FROM <http://appl.com/>
WHERE  { ?b <http://appl.com/purchase/buy/price> ?price.
         FILTER (xsd:integer(?price) < 30 && xsd:integer(?price) > 26 )
 }
```

2.2.6.1.4 OPTIONAL

Este operador permite indicar una condición de búsqueda opcional, lo cual permite devolver valores que cumplen con esta condición opcional y con las obligatorias, o que solo cumplen con las condiciones obligatorias.

Por ejemplo, si se quiere obtener los productos comprados por un usuario y además en caso de que los haya valorado, obtener también el rating.

```
SELECT *
FROM <http://appl.com/>
WHERE {
    {?u <http://appl.com/actions/buying> ?b .
    ?b <http://appl.com/purchase/buy/article> ?a.
    }
  OPTIONAL{
        ?u <http://appl.com/actions/reviewing> ?r .
        ?r <http://appl.com/rev/review/of> ?a .
        ?r <http://appl.com/rev/review/rating> ?v .
        }
}
```

2.2.6.1.5 UNION

Permite agrupar resultados en un mismo atributo, los cuales son devueltos porque cumplen un criterio de búsqueda presente en la unión.

Por ejemplo, para obtener todos los artículos comprados y los artículos valorados por un usuario se haría la siguiente consulta:

```
SELECT ?a ?v
FROM <http://appl.com/>
WHERE{
       {<mailto:user5@email.com> <http://appl.com/actions/buying> ?b .
       ?b <http://appl.com/purchase/buy/article> ?a.
       }
 UNION{
       <mailto:user5@email.com> <http://appl.com/actions/reviewing> ?r .
       ?r <http://appl.com/rev/review/rating> ?v .
       FILTER (xsd:integer(?v)>5).
       ?r <http://appl.com/rev/review/of> ?a .
      }
 }
```

2.2.6.1.6 Modificadores de resultados

Estos operadores permiten modificar la secuencia de resultados obtenidos para obtener una secuencia con ciertas características que puedan acercarse más a las necesidades de los usuarios.

Éstos modificadores son los siguientes:

- Order By: ordena la secuencia de resultados en función del campo indicado. El orden puede ser tanto ascendente como descendente y se indica mediante los operadores ASC() y DESC().
- Distinct : elimina los resultados repetidos de la secuencia de resultados.
- Offset: elimina los x primeros elementos de la secuencia de resultados y devuelve el resto.
- Limit: limita el número de resultados que se devuelven.
- Group By: permite agrupar en una sola tripleta los valores de uno o varios campos que sean idénticos.

Estos modificadores no son excluyentes unión de otros, por lo que al igual que ocurre con SQL pueden ser utilizados al mismo tiempo.

Por ejemplo, para obtener los cinco primeros artículos con mayor valoración, sin repetir, del usuario3 se haría la siguiente consulta:

```
SELECT DISTINCT ?a ?v
FROM <http://appl.com/>
WHERE  {
    <mailto:user3@email.com> <http://appl.com/actions/reviewing> ?r .
    ?r <http://appl.com/rev/review/of> ?a .
    ?r <http://appl.com/rev/review/rating> ?v .
}
ORDER BY DESC(?v)
LIMIT 5
```

2.2.6.1.7 Funciones de agregación

Tal y como ocurre en SQL, se pueden utilizar funciones de agregación como MAX(), MIN(), SUM(), COUNT(), AVG().

Esto permite realizar cálculos sobre los resultados obtenidos en las consultas. Por ejemplo, si se quisiera obtener el coste total de todos los productos de un usuario dado se haría lo siguiente:

```
Select SUM(xsd:int(?v))
from <http://appl.com/>
WHERE{
 <mailto:user3@email.com> <http://appl.com/actions/buying> ?b .
 ?b <http://appl.com/purchase/buy/price> ?v
}
```

2.2.6.1.8 CONSTRUCT

Esta consulta es similar a SELECT, salvo que en lugar de devolver variables, CONSTRUCT es capaz de devolver grafos.

La sintaxis de este tipo de consultas es la siguiente:

```
CONSTRUCT { ?x foaf:name ?name }
FROM <http://appl.com/>
WHERE  { ?x foaf:name ?name }
```

2.2.6.1.9 ASK

Este tipo de consulta a diferencia de SELECT y CONSTRUCT, no devuelve valores guardados en los grafos, sino que únicamente devuelve un valor booleano indicando si el criterio de búsqueda se cumple o no.

Por ejemplo, un uso de este tipo de consulta sería el de comprobar que las credenciales de un usuario son validas, para esto se podría hacer la siguiente consulta sobre la base de datos de prueba:

```
ASK {
    <mailto:user3@email.com> <http://appl.com/users/pwd> "tab053"
    }
```

2.2.6.2 Operaciones de Escritura SPARQL

En esta sección se van a mostrar las operaciones de escritura que se pueden realizar mediante SPARQL una base de datos rdf en Virtuoso.

2.2.6.2.1 DELETE

DELETE permite eliminar tripletas de un grafo. En Virtuoso es posible realizar esto de varias maneras.

Por un lado, es posible eliminar una tripleta con datos concretos mediante la consulta:

```
DELETE DATA FROM <grafo> {<datos>}
```

Por otro, es posible eliminar todas las tripletas que cumplan una serie de condiciones mediante la consulta:

```
DELETE FROM <grafo> {?u ?v ?o}
WHERE{<condiciones de busqueda>}
```

Utilizando el segundo tipo de consultas, es posible hacer que se borre una única consulta con unos determinados valores, sin embargo, no es posible utilizar la primera consulta para borrar tripletas sin especificar los valores de todos los campos, es decir, los datos de la primera consulta no pueden ser variables del tipo "?p".

2.2.6.2.2 INSERT

Para realizar inserciones sobre un grafo se utiliza la siguiente consulta:

```
INSERT INTO <http://appl.com/> {<datos>}
```

Y de forma opcional se puede incluir la clausula WHERE para indicar que los valores de uno o varios campos deben cumplir una cierta condición.

Por ejemplo, si se quiere introducir un nuevo usuario que conoce a los usuarios que han comprado el artículo número 1, se haría la siguiente consulta:

```
INSERT INTO <http://appl.com/> {<mailto:userB@email.com> foaf:knows ?v}
WHERE {
    GRAPH  <http://appl.com/>
        { ?v <http://appl.com/actions/buying> ?b .
          ?b <http://appl.com/purchase/buy/article> <http://appl.com/art\
   /1>
        }
  }
```

2.2.6.2.3 MODIFY

Este tipo de operaciones no permite realizar una operación de modificación sobre los datos directamente, sino que requiere de dos tareas, primero se borra la tripleta a modificar y seguidamente se inserta una nueva tripleta con los nuevos valores. Por tanto, una consulta que permitiera modificar el nombre de usuario de todos los usuarios que conocen a alguien - en el dataset sólo existe el usuario userB@email.com insertado en el punto anterior -, tendrá la siguiente forma:

```
MODIFY GRAPH <http://appl.com/>
DELETE { ?s foaf:knows ?o }
INSERT { <mailto:userC@email.com> foaf:knows ?o }
FROM <http://appl.com/>
WHERE { ?s foaf:knows ?o }
```

2.2.6.3 Operaciones sobre grafos

Además de las operaciones anteriores que realizan operaciones sobre las tripletas de los grafos, es posible realizar operaciones sobre los grafos que afectarían a todas las tripletas que forman el grafo.

2.2.6.3.1 CREATE

Como se ha dicho en apartados anteriores, es posible crear un grafo mediante la operación de insertar, indicando como grafo el nombre del nuevo grafo. Sin embargo, también es posible crear un nuevo grafo mediante la consulta "CREATE GRAPH nombre del grafo".

2.2.6.3.2 CLEAR y DROP

Estas dos operaciones permiten eliminar los datos de un grafo dado. En el caso de "CLEAR", se borrarán todos los datos del grafo, pero el grafo no se eliminará. En el segundo caso - "DROP" - se borrarán tanto los datos como el grafo.

La sintaxis es similar para los dos casos:

```
DROP/CLEAR <grafo>
```

2.2.6.3.3 COPY

Esta claúsula permite copiar los datos de un grafo a otro grafo. Esta operación tiene como consecuencia que los datos del grafo destino son sobrescritos con los nuevos datos.

La consulta para realizar esta operación tendría la siguiente sintaxis:

```
COPY <grafo> TO <grafo>
```

2.2.6.3.4 MOVE

Esta claúsula es similar a la anterior, salvo que en este caso, los datos del grafo origen son borrados, por lo que solo habrá una copia de los datos en el grafo destino. La consulta para realizar esta operación tendría la siguiente sintaxis:

```
MOVE GRAPH <grafo> TO GRAPH <grafo>
```

2.2.6.3.5 ADD

Con esta claúsula se permite insertar los datos de un grafo en otro grafo, de forma los datos que contenía el grafo destino no son sobrescritos, sino que se mantienen los datos de los dos grafos.

```
MOVE GRAPH <grafo> TO GRAPH <grafo>
```

2.2.6.4 Formato de salida de los datos

Es posible definir el formato de salida que tendrán los datos obtenidos como resultado de una consulta mediante la claúsula "define output:format "formato"". Los formatos que se le pueden dar a los datos son : XML, N3, TURTEL y json. Para obtener los datos en los distintos formatos simplemente hay que sustituir en la claúsula anterior el formato por "RDF/XML", NT,TTL y JSON respectivamente.

2.2.6.5 Funciones dentro de SPARQL

Además de SPARQL para las bases de datos RDF y SQL para las bases de datos relacionales, Virtuoso tiene una serie de funciones que pueden ser utilizadas desde la linea de comandos ISQL. Algunas de estas funciones además pueden ser utilizadas dentro de las consultas SPARQL, lo cual permite realizar consultas más complejas.

Entre las diferentes operaciones para utilizar dentro de las consultas SPARQL, podemos destacar cuatro grupos de operaciones:

1. bif: estas funciones se usan precedidas del prefijo "bif:", el cual indica que la función es una función SQL creada en virtuoso.
2. sql: estas funciones se usan precedidas del prefijo "sql:", el cual indica que la función es una función de virtuoso del tipo DB.DBA; donde , por ejemplo, "sql:ejemplo" estaría llamando a la función DB.DBA.ejemplo.
3. xsd: estas funciones se usan precedidas del prefijo "xsd:", este tipo de funciones permiten convertir entre tipos de datos como de string a entero. Por ejemplo, para convertir un dato a entero se indicaría como: "xsd:int(dato)".
4. Funciones que no requieren prefijo: estás funciones son especificas para usar con SPARQL - similares en algunos casos a las funciones bif - y permiten realizar operaciones de modificación y comparación de números, strings, datos de tipo fecha.

2.2.6.5.1 Ejemplo

Un ejemplo del uso de las diversas funciones para realizar una consulta, es por ejemplo si se quisiera añadir una nueva compra a los datos. El primer problema que se presenta es que el URI's que identifica a la compra ha de ser único. Si este identificador fuera un valor simple, sería suficiente con obtener el valor máximo ya presente y sumarle uno - como este es un ejemplo para comprobar el uso de funciones dentro de consultas SPARQL , no se está teniendo en cuenta problemas relacionados con la atomicidad de las operaciones.

Sin embargo, dicho campo contiene una URI que hace referencia al subgrafo de la compra y por tanto no se puede hacer una simple suma. Una posible solución sería realizar la siguiente serie de operaciones sobre la URI obtenido:

1. Convertir la URI al formato string mediante la función "str()"
2. Separar el identificador del resto del URI con la función "bif:trim()". Lo que esta función hace es que dado un dato de tipo string y un prefijo también de tipo string devuelve el dato inicial menos el prefijo.
3. Convertir el identificador a tipo string mediante la función "xsd:integer", sumarle uno y volver a convertirlolo a tipo string, de nuevo con la función "str".
4. Concatenar el prefijo anterior con el nuevo identificador con la función bif:concat.
5. Convertir el dato de tipo string a iri - identificador de la URI - mediante la función ‘iri(?num)‘.

Siguiendo esos pasos la consulta tendría la siguiente forma:

```
PREFIX does:<http://appl.com/actions/>
PREFIX  art:<http://appl.com/art/>
PREFIX p:<http://appl.com/purchase/>

INSERT INTO <http://appl.com/>
{<mailto:userA@email.com> does:buying ‘iri(?num)‘}
WHERE{
{SELECT
    (bif:concat
        ("http://appl.com/purchase/",
         str(xsd:integer(bif:trim(
                    str(MAX(?b)),
                    "http://appl.com/purchase/")) + 1))
    ) as ?num
WHERE{
     ?u does:buying ?b.
    }
}
}
```

2.2.6.6 Funciones Virtuoso PL

Virtuoso permite crear métodos o funciones propios que pueden ser utilizados tanto desde la línea de comandos como desde las consultas SQL o SPARQL - los métodos no pueden

ser llamados desde SPARQL pero las funciones si-. Una vez creados, estos son compilados y después guardados en la tabla "SYS_PROCEDURES". Los métodos de esta tabla son siempre cargados al iniciarse el servidor por lo que pueden están disponibles al usuario en cualquier momento. Es posible, tanto crear métodos propios como reescribir métodos presentes previamente en el sistema.

Para crear un método se utiliza la siguiente sentencia:

```
CREATE FUNCTION/PROCEDURE Nombre ([in|out|inout] parametro tipo)
[opcional return tipo ]
{sentencias}
```

Generalmente el nombre de las funciones son del tipo DB.DBA.nombre, de forma que es posible llamarlas desde SPARQL utilizando el namespace "spl:". Hay que tener en cuenta que para ejecutar estas nuevas funciones es necesario dotar a los usuarios de los permisos necesarios para ejecutar dichas funciones. Para ejecutar desde ISQL es suficiente dar permisos de ejecución al usuario que este realizando la operación como se indica en los apartados anteriores. Para ejecutar desde la interfaz WEB es necesario otorgar los permisos de ejecución al usuario SPARQL - la sentencia para otorgar los privilegios de ejecución es: grant execute on DB.DBA.hello to "SPARQL".

Seguidamente se muestran una serie de sentencias y opciones que se pueden utilizar para crear los métodos.

2.2.6.6.1 Declaración de variables

Para declarar variables se utiliza la sentencia "Declare" seguida del nombre de la variable y del tipo. Entre otros tipos - como int - es posible indicar el tipo de variable como "any" para no tener que especificar éste.

2.2.6.7 Sentencias Condicionales

Dentro de este tipo de sentencias podemos encontrar sentencias de tipo if else, while, whenever... El formato de estas sentencias en general sigue el patrón:

```
if/while/whenever (condicion) {resto de sentencias;}
```

Además también esta disponible la sentencia "FOR" que tiene la forma:

```
for(declare i int; i:=1;X<=3;opcional[x:=x+1]){resto de sentencias;}
```

Otra variante de la sentencia anterior es la sentencia "FOR VECTOR", la cual permite iterar los valores de un array de la siguiente forma:

```
FOR VECTORED (IN i INT := array; OUT rest := r ) {r:= i+2;} return rest;
```

2.2.6.7.1 Sentencias de salto

En lugar de usar las sentencias condicionales para controlar el flujo del programa es posible realizar saltos a otro punto del programa mediante la sentencia "GOTO" y el uso de un label para indicar el lugar al que saltar. Esta sentencia puede ser usada junto a la sentencia "WHENEVER" para realizar saltos condicionados con la siguiente forma:

```
WHENEVER condicion GOTO label;
LABEL: sentencia;
```

2.2.6.7.2 Llamada a otros métodos

Es posible realizar llamadas a otros métodos mediante la sentencia "CALL", y si fuera necesario asignar el valor de salida a una variable.

2.2.6.7.3 Sentencias SPARQL

También es posible realizar consultas SPARQL - y SQL - dentro de los métodos. Esto puede ser útil tanto para realizar una tarea dentro del método que requiera consultar datos guardados en algún grafo como para crear un método que ejecuta una consulta que pueda ser utilizada con cierta frecuencia. Para realizar estas consultas dentro de los métodos es suficiente con indicar que es una consulta SPARQL antes de la consulta, tal y como se hace al utilizar la linea de comandos isql.

Por ejemplo para obtener los nombres de todos los usuarios se podría crear la siguiente función:

```
create function DB.DBA.hello(){
declare lista any array;
declare i int; i:=0;
for(sparql select ?n from <http://appl.com/> where{?u foaf:name ?n})do
  return result("n");
}
```

2.2.6.7.4 Uso de las funciones

Estas funciones y métodos, pueden ser utilizadas de varias maneras. Se pueden llamar desde la línea de comandos ejecutando "SELECT nombre_de_la_función()" , desde la

linea de comando en segundo plano " nombre_de_la_función()&", y desde las consultas SPARQL siempre que sea una función y vaya precedida del namespace correspondiente - si es de tipo DB.DBA que vaya precedida de "sql:" y si es de tipo SPARQL.SPARQL de "bif:".

Por ejemplo, para llamar a la función creada anteriormente desde SPARQL, se podría hacer de la siguiente manera:

```
SELECT sql:hello()
FROM <http://appl.com/>
where{}
```

2.2.6.8 Reglas de inferencia

En Virtuoso es posible crear reglas de inferencia las cuales permiten realizar consultas sobre tripletas que no están físicamente guardadas en el sistema de bases de datos. Esto permite que, por ejemplo, si se tiene un grafo que contiene tripletas que representa la relación "padre de", se pueda crear una regla gracias a la cual se puedan realizar consultas sobre la relación hijo de sin tener que insertar tripletas con dicha relación. Para poder construir las reglas es posible valerse del lenguaje de ontología OWL el cual ofrece propiedades y clases como owl:inverseOf o owl:TransitiveProperty; o del esquema RDF con relaciones como rdfs:subClassOf o rdfs:subPropertyOf entre otras.

Para construir y utilizar las reglas hay que realizar tres pasos:

1. Construir las reglas insertando en un grafo destinado a esta tarea, las relaciones que hay entre la propiedad guardada físicamente y la que se quiere utilizar para inferir.

2. Indicar que el grafo es utilizado para guardar reglas y asignarle un nombre mediante la función rdfs_rule_set(nombre, grafo). Esta función se debe ejecutar desde la línea de comandos.

3. Indicar cuando una consulta utilizará las reglas construidas mediante la clausula "DEFINE input:inference nombre".

Seguidamente se muestra como crear y utilizar éstas reglas con el ejemplo anterior de las relaciones familiares.

2.2.6.8.1 Ejemplo

Suponiendo que se tiene un grafo que contiene tripletas con información sobre que persona es padre de que otra persona, creado mediante una consulta como la siguiente:

```
PREFIX relacion:<http://familia.com/relacion/>
PREFIX persona:<http://familia.com/persona/>
INSERT INTO GRAPH <http://familia.com/>
{
 persona:Carlos  relacion:padre_de persona:Alex
}
```

El primer paso sería construir una regla la cual indica que si una persona es padre de otra, la segunda persona será hijo de la primera. Para esto es necesario insertar en un grafo - destinado a guardar las reglas - una tripleta que indique que si hay una relación padre de entre X e Y entonces hay una relación hijo de entre Y y X, para indicar este tipo de relación entre las dos propiedades se puede usar owl:inverseOf. Para esto se puede realizar una operación como la siguiente:

```
PREFIX relacion:<http://familia.com/relacion/>
PREFIX persona:<http://familia.com/persona/>
INSERT INTO GRAPH <familia:reglas>
  { relacion:padre_de owl:inverseOf relacion:hijo_de}
```

Una vez hecho esto hay que indicar que el grafo anterior contiene reglas de inferencia y habrá que asignarle un nombre mediante la ejecución de la sentencia "rdfs_rule_set ('rel', 'familia:reglas');", mediante la linea de comandos isql.

Por último, si por ejemplo se quiere realizar una consulta que obtenga a todas las personas y a sus padres, siempre que cumplan que se relacionan mediante "hijo_de"; debería indicarse que se quiere usar la regla de inferencia creada anteriormente mediante la clausula DEFINE input:inference "rel". Con lo que la consulta final tendría la siguiente forma:

```
DEFINE input:inference "rel"
PREFIX relacion:<http://familia.com/relacion/>
SELECT *
FROM <http://familia.com/>
WHERE
    {?hijo  relacion:hijo_de ?padre}
```

2.2.7 Índices

Como ya se ha comentado en secciones anteriores Virtuoso guarda los grafos RDF en la tabla DB.DBA.RDF_QUAD, la cual contiene cuatro columnas, tres para guardar las tripletas <S,P,O> - sujeto, predicado y objeto - y una para guardar el nombre del grafo. Esto hace que, aunque no sea posible implementar una política de índices para cada grafo, si se puedan usar índices para mejorar el rendimiento de las consultas.

Por defecto Virtuoso viene con un esquema de índices que se adapta a los diferentes tipos de consultas SPARQL que se puedan realizar sobre los grafos guardados en el sistema , [26]. Este esquema está formado por cinco índices, dos índices con todas las columnas y tres índices parciales:

- PSOG: índice construido por defecto al crear la tabla ya que contiene los campos que forman la clave primaria de la tabla.
- POGS: índice de tipo bitmap sobre todos los campos de la tabla.
- SP, OP y GS: estos son índices parciales sobre dos campos de los campos de la tabla. Se usan principalmente para cuando sólo el valor de s , o o g , respectivamente, es especificado.

Los dos primeros índices permiten que se pueda encontrar resultados a una consulta de forma eficiente si se ha especificado el valor de P y al menos el valor de S o O.

Si por el contrario sólo se especifica S, se buscarán los valores S y P en el índice SP y una vez obtenidos se usarán para acceder al índice PSOG y obtener los valores de O y/o el grafo G. En caso de que se especificará O la búsqueda sería similar, salvo que se utilizaría el índice OP para después buscar S y/o el grafo G en el índice POGS. En caso de que sólo se especifique el nombre del grafo G la búsqueda sería algo diferente, primero se buscaría en GS los valores de S para cada G, seguidamente se buscarían los valores de P para cada S y finalmente se buscarían todo el cuarteto <G,S,P,O> en el índice PSOG.

Generalmente, este esquema permite realizar consultas de forma eficiente tanto en los casos en los que hay pocos grafos con gran cantidad de tripletas, como en los que hay muchos grafos con pocas tripletas. Sin embargo, en el segundo caso, si se borran con frecuencia los grafos, el esquema de índices anterior puede hacer que el rendimiento no sea bueno. Esto es debido a que en las consultas de borrado de grafos sólo se especifica el valor de G, por lo que hay que realizar tres búsquedas: en el índice GS, en el PS y en el PSOG. Para evitor esto la mejor opción sería eliminar el índice GS y crear un nuevo índice GPSO con G, P, S y O, de esta forma sólo sería necesario realizar una búsqueda para obtener los cuartetos a borrar.

2.2.7.1 Crear índices

Para crear índices se utiliza la sentencia CREATE INDEX.

```
CREATE <opciones> INDEX <nombre del indice>
    ON RDF_QUAD (<columnas>);
```

De la lista de opciones disponibles, destacan BITMAT para indicar que el índice es un mapa de bits, DISTINCT para indicar que no se guarden repeticiones, NO PRIMARY KEY para indicar que las columnas usadas no forman una clave primaria y REF.

Opcionalmente se puede indicar al final de la consulta si se quiere que el índice se particióne.

Por ejemplo, el índice POGS bitmap se crearía de la siguente manera:

```
CREATE BITMAP INDEX RDF_QUAD_POGS
  ON RDF_QUAD (P, O, G, S)
  PARTITION (O VARCHAR (-1, 0hexffff));
```

2.2.7.2 Eliminar y Alterar índices

Para eliminar índices se utiliza la sentencia DROP INDEX <nombre del índice>.

Para alterar los índices se utiliza la sentencia ALTER INDEX <nombre del índice> ON <nombre de la tabla>. 15

Capítulo 3

Aplicación de Persistencia Políglota: Diseño

En este capítulo se va a tratar cada uno de los diferentes aspectos relacionados con el diseño de la *"Aplicación de Persistencia Políglota"*. Entre ellos se va a tratar las diferentes decisiones que se han tomado para diseñar la arquitectura de la aplicación, la distribución de los datos entre los distintos sistemas y el modelo de los datos que se ha diseñado en función del sistema en el que guarden los diferentes datos. Además, se indicaran los casos de uso y el modelo de clases.

3.1 Arquitectura

La principal característica de una "*Aplicación de Persistencia Políglota*" es que utiliza varios sistemas de bases de datos para realizar las diferentes operaciones que se requieran. En esta sección se muestra la arquitectura de la aplicación desarrollada. arquitectura que ha sido diseñada teniendo en cuenta las características mencionadas anteriormente.

3.1.1 Arquitectura general

En la sección anterior [1.2.3], se habló de las diferentes formas que podría tener la arquitectura general de este tipo de aplicaciones. Para esta aplicación se optó por utilizar una arquitectura similar a la expuesta en la figura 1.4.

La elección de este diseño en lugar de los otros se debe a dos razones principales: (1) la existencia de diferentes sistemas de bases de datos es totalmente transparente al resto de la aplicación fuera de los servicios; y (2) tener los servicios divididos en función de qué

datos se traten permite que si se utiliza un nuevo sistema para ciertos datos, el cambio sólo afectará al servicio que trata esos datos y el resto de los servicios que utilizaban el mismo servicio no se verán afectados.

Además, la modularidad de los servicios permite que si se quieren reutilizar éstos para otra aplicación, ésta sólo tendrá acceso a los datos y servicios que necesite.

Como se comento antes, además de este diseño existían otras opciones. La primera era que la aplicación se conectase directamente al sistema de bases de datos que se necesitase en cada momento. Una de las principales razones para no seguir este diseño es que perdería una de las características principales que se busca con esta aplicación y es que el uso de varios sistemas sea transparente para la mayor parte de la aplicación.

La segunda opción era que cada sistema estuviese envuelto en un servicio. Por ejemplo, si las cestas y los artículos están en un mismo sistema, entonces un mismo servicio se encargaría de gestionar las cestas y los artículos. Aunque hay un cierto nivel de transparencia, la principal razón por la que no se utilizó este diseño es una falta de modularidad que provoca, entre otros, que el cambio de ciertos datos de un sistema de datos a otro sea complejo.

3.1.2 Servicios

Para poder hacer frente a las diferentes tareas y operaciones necesarias para cumplir con las necesidades de la aplicación, es necesario tener una serie de servicios que den acceso a los diferentes datos.

Además, como se comento en secciones anteriores, para el almacenamientos de los datos se van a utilizar varios sistemas de almacenamiento, principalmente tecnologías NoSQL. En concreto, se utilizarán MongoDB y Openlink Virtuoso. Junto a los sistemas NoSQL también se utilizará el sistema relacional MySQL.

Como cada sistema tiene diferentes características y cada tipo de datos y el uso que haga de estos tienen diferentes necesidades, los servicios creados y los sistemas que se usan para cada uno de ellos se reparten de la siguiente forma:

Servicio de artículos

La tarea de este servicio es la de dar acceso a la información de los artículos que se venden en el comercio. Por tanto, los datos con los que trata este servicio son los relacionados con las características de los artículos.

En general las tareas que se realizarán sobre estos datos se limitarán a consultar los datos de uno o varios artículos, y en menor medida a modificar/añadir/eliminar artículos. Por tanto, no se van a realizar tareas extremadamente complicadas sobre estos datos.

La característica principal de estos datos, es que existen distintos tipos de artículos que aunque similares tienen ciertas propiedades que son especificas de cada tipo. Esto hace que un sistema flexible, como es el caso de los sistemas NoSQL, que permita adaptarse a los distintos tipos de artículos - actuales y futuros - sea más adecuado.

Por último, existen ciertas propiedades de los artículos cuyo valor es un conjunto de datos. Por tanto, puesto que dichas propiedades son fácilmente representables mediante arrays o subdocumentos en MongoDB, se ha elegido este sistema para almacenar los artículos. Si bien es cierto que es posible representar estas propiedades en Virtuoso mediante varias propiedades y subgrafos, esto resulta más complejo que tener un campo que contenga un array o un subdocumento.

Servicio de cestas

La tarea principal de este servicio es la de permitir a los usuarios crear cestas de la compra y acceder a los datos de estas cestas. Este tipo de cestas en general tienen información sobre el usuario que creó la cesta, una lista de artículos para comprar y la cantidad elegida de cada artículo.

En general, la estructura de estos datos siempre va a ser la misma; con lo que en este caso un sistema flexible no sería tan importante. Sin embargo, hay dos puntos importantes a tener en cuenta al elegir el sistema que almacene las cesta: (1) siempre se querrá obtener ciertas propiedades de los artículos junto con los datos de la cesta; y (2) las cestas expirarán transcurrido un tiempo.

Puesto que la información de la cesta consta de una lista de documentos, representar esta información en un sistema relacional requeriría una o varias tablas donde cada cesta está representada por varias filas.

Aunque este método es perfectamente valido, resulta más simple tener un único documento donde se guarde toda la información de una cesta. Además, MongoDB tiene un tipo de índices llamados TTL que permiten que algunos documentos de una colección se eliminen de forma automática transcurrido un cierto tiempo. Por esto, para almacenar la información de la cesta se ha utilizado MongoDB.

Servicio de pedidos

La tarea principal de este servicio es la de mantener al día el estado de los pedidos - pendiente, enviado, disponible...- y mostrar los pedidos recientes que han realizado los usuarios.

En este caso, al igual que ocurre con la cesta, se ha elegido el sistema de documentos MongoDB para guardar los datos. Esto se debe, por un lado a las ventajas que da el sistema TTL para poder poner un tiempo de caducidad a lo datos de los pedidos en un cierto estado si se quisiera, y por otro a la posibilidad de guardar todos los datos del pedido, los cuales generalmente serán accedidos de forma conjunta - en un único documento, evitando con ello consultas complejas.

Servicio de recomendaciones

Este servicio principalmente obtiene recomendaciones de artículos en función de las compras o valoraciones que han hecho otros usuarios. Por ejemplo, para un artículo dado busca los artículos comprados por otros usuarios que también han comprado este artículo.

En general, para realizar esta tarea es necesario usar información sobre las compras, valoraciones y gustos de los usuarios. Las principales necesidades a la hora de utilizar un sistema de almacenamiento, en este caso, no es la forma que tendrán los datos mostrados; sino la relación entre los datos.

Por tanto, una buena representación de los datos para realizar estás tareas sería en forma de grafos RDF, donde los datos se guardan en función de sus relaciones con otros datos. En general Virtuoso es el que está más adaptado a este diseño de datos por lo que se utilizará este para almacenar la información necesaria para obtener recomendaciones.

Si bien es cierto que es posible representar grafos en MySQL y MongoDB, no es tan adecuado que hacerlo con Virtuoso. En el primer caso, representar datos muy relacionados requerirá varias tablas y obtener las recomendaciones de artículos necesitará de consultas con varios "JOINS", aumentará la complejidad de estas consultas y en ciertos casos hará disminuir el rendimiento. En el segundo caso, es posible representar los grafos RDF con los documentos de forma sencilla. Sin embargo, el rendimiento al realizar consultas sobre datos RDF en MongoDB es peor que en Virtuoso[7].

Servicio de usuarios

La tarea principal de este servicio es la de realizar consultas sobre la información de los usuarios. Estás consultas pueden ser tanto del tipo obtener datos personales, como comprobar las credenciales de identificación o crear/eliminar/modificar un usuario.

Puesto que la información guardada para este caso será muy simple y constará del nombre, el identificador y la contraseña del usuario; al igual que en el caso anterior se utilizará Virtuoso para reducir la cantidad de datos guardados. Si la información fuera más compleja, donde por ejemplo se guardaran múltiples direcciones de envío, o se dejara personalizar aspectos del perfil de usuario - como poner un avatar - una opción más adecuada sería usar sistema de documentos como MongoDB.

Servicio de compras

La tarea principal es la de obtener información sobre las compras realizadas por los usuarios. Esta información se reducirá al artículo comprado y precio.

En este caso, la información guardada de las compras es similar a la que se necesita para el servicio de recomendaciones, por tanto se utilizará Virtuoso para guardar esta información con la idea de reducir el número de datos guardados en todos los sistemas.

Servicio de valoraciones

La tarea principal es la de obtener información sobre las valoraciones realizadas por los usuarios, tanto las valoraciones escritas como los gustos.

En este caso ocurre algo similar al caso anterior, la mayor parte de los datos necesarios para guardar la información de las valoraciones es similar a la que se necesita para consultar los artículos recomendados. Por tanto, se reutilizarán los datos guardados en Virtuoso para dar el servicio de recomendaciones.

Servicio de inventario

La tarea de este servicio es la de mostrar y modificar la cantidad de cada artículo disponible en el comercio.

Una de las principales características que deben tener estos datos no es su formato, sino la necesidad de que los datos deben ser consistentes en todo momento. Esto se debe a que los datos referentes a la cantidad de artículos disponibles serán modificados al realizar una compra, y si los datos no fueran consistentes podría ocurrir que varios

usuarios modificaran el mismo valor al mismo tiempo y el valor final guardado no fuera correcto. Un valor erróneo en la disponibilidad podría llevar a que los clientes compraran artículos que en la realidad no están disponibles.

Los sistemas relacionales que generalmente cumplen con el principio ACID, permiten mantener la consistencia de los datos mediante el uso de transacciones y cerrojos manuales. Por esto, para guardar los datos relacionados con el inventario se ha utilizado el sistema relacional MySQL.

Los otros dos sistemas aunque permiten mantener un cierto nivel de consistencia, no están tan adaptados como los sistemas relacionados. Por un lado, MongoDB no dispone de transacciones y aunque permite realizar operaciones de forma atómica no permite hacer "rollback", en caso de que alguna operación falle. Por otro lado, Virtuoso cumple el principio ACID y permite realizar transacciones a nivel de aplicación; sin embargo, para el almacenamiento de tripletas Virtuoso carece de herramientas como "SELECT FOR UPDATE" que permiten bloquear manualmente ciertas filas de forma sencilla.

En resumen, para almacenar datos que requieren una estructura compleja y que generalmente van a ser consultados al mismo tiempo se utiliza MongoDB; para datos que van a ser utilizados para tareas que requieren información de como están relacionados los datos se utiliza Virtuoso y para datos cuyo uso requiere que sean consistentes se utiliza MySQL. En la figura 3.1 se muestra que sistema es usado por cada servicio.

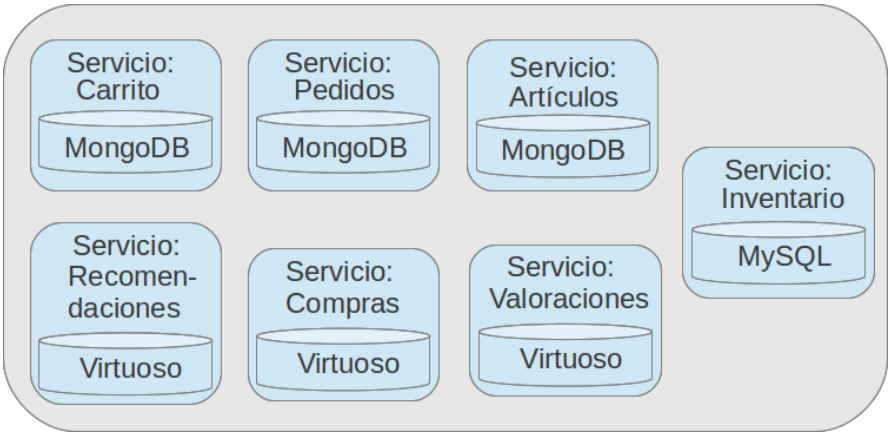

FIGURA 3.1: Distribución de los servicios

3.1.3 Aplicación Web

Además del diseño general anterior, la aplicación web en sí esta diseñada teniendo en mente el modelo en tres capas. Siguiendo este diseño, la aplicación se puede dividir en tres partes.

La primera capa es la capa de Presentación. La principal tarea de esta capa es la de presentar la aplicación a los usuarios mediante una interfaz de usuario y la de recoger las diferentes operaciones que realiza el usuario sobre la interfaz. Se comunica directamente con la segunda capa y una vez recibida la respuesta de esta, presenta los resultados a los usuarios. Para esta aplicación web, esta capa correspondería a las páginas web y a la parte de la aplicación que recoge las peticiones recibidas desde las páginas. En este caso, esta parte correspondería a las páginas web html/jsp y a los Servelet.

La segunda capa es la capa de la Lógica de Negocio. La principal tarea es la de procesar las peticiones realizadas por los usuarios. A diferencia de la capa anterior, ésta no se comunica con el usuario, sino que la capa de presentación le comunica las peticiones recibidas y ésta segunda capa realiza las operaciones necesarias para procesar las peticiones y devuelve la respuesta a la primera capa. Además, también se comunica con la tercera capa, para obtener la información necesaria que le permita realizar las diferentes operaciones. En esta aplicación, esta capa corresponde a todas aquellas clases que no son ni Servelets, ni parte de los servicios base comentados en el punto anterior.

Finalmente, la tercera capa es la capa de Datos. La principal tarea de esta capa es la de comunicarse directamente con los sistemas de almacenamiento de datos para realizar consultas sobre los datos en función de las peticiones recibidas desde la capa de Negocio. Es decir, si la capa de negocio necesita ciertos datos para procesar la peticiones de los usuarios, esta tercera capa se encarga de obtenerlos de la/s base/s de datos. En esta aplicación, esta capa correspondería con los servicios base comentados en el punto anterior; los cuales se encargan de realizar operaciones sobre los datos almacenados en los distintos sistemas de bases de datos.

En la figura 3.2 se puede observar el esquema general utilizado para diseñar la arquitectura siguiendo un diseño en tres capas. En éste, la capa de Presentación está formada principalmente por los Servelet, la capa de la Lógica de Negocio por los Gestores y la capa de datos por los Servicios Básicos vistos en el punto anterior. Como se puede observar además, la capa de presentación y la de datos no tienen comunicación directa, sino que están separadas por la capa de Negocio.

Una de las principales razones para elegir este diseño es que se adapta perfectamente al diseño general de la "*Aplicación de Persistencia Políglota* ", ya que en ambos casos se

busca separar la parte que interactúa con los sistemas de bases de datos, del resto de la aplicación. Por otro lado, si bien es posible separar la aplicación únicamente en dos partes, hacer la separación en varias capas permite que sea más sencillo realizar cambios que sólo afecten a una de las capas.

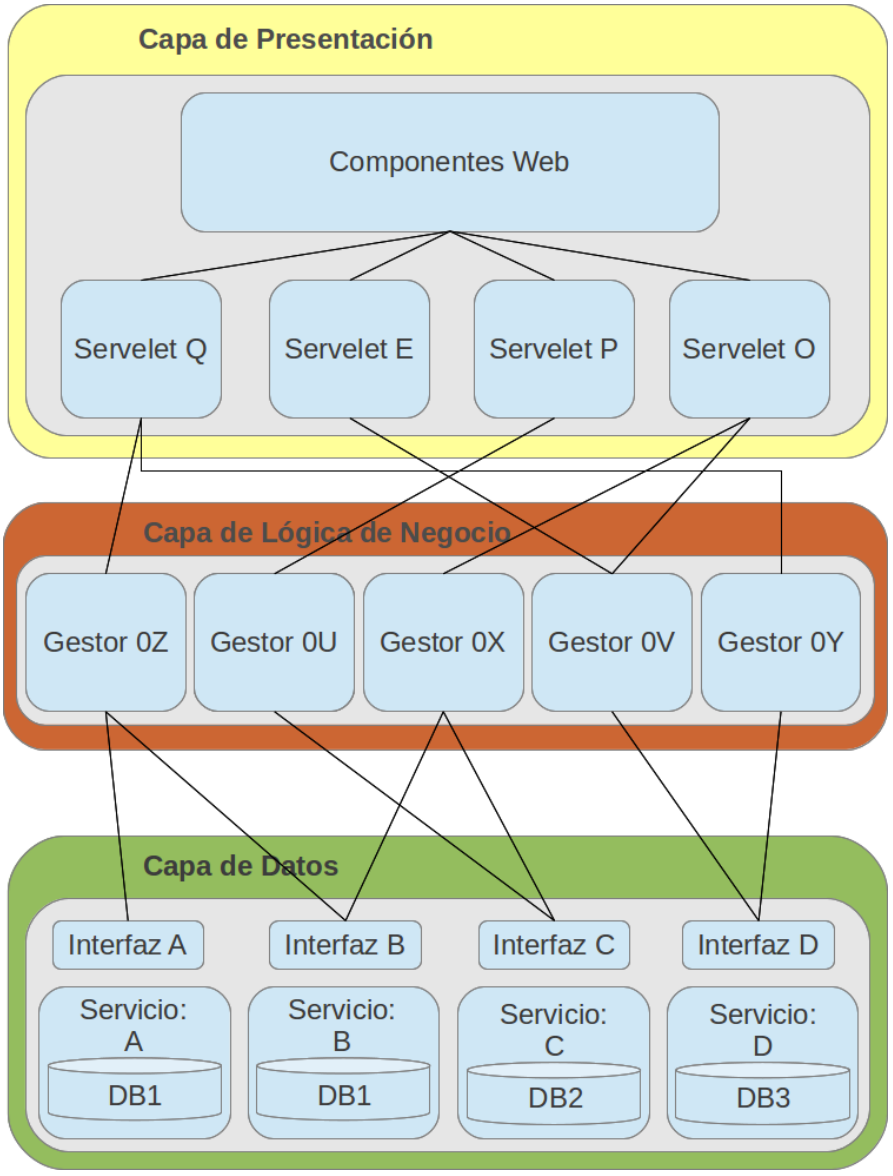

FIGURA 3.2: Arquitectura en Tres Capas

3.2 Modelo de Casos de Uso

Con la intención de plasmar y prever cual será la interacción de los distintos usuarios con la aplicación se ha diseñado un modelo de casos de usos. En este modelo intervienen dos actores principales:

1. Usuario: es un usuario básico del sistema. Podrá realizar tareas como ver los artículos o gestionar la cesta. Además, podrá identificarse o registrarse para pasar a ser un usuario "Cliente".

2. Cliente: es un usuario identificado por el sistema. Podrá realizar todas las tareas que realiza el usuario básico - excepto registrase o identificarse - y además podrá realizar pedidos y acceder a una zona única para clientes, donde podrá ver el historial de compra o los pedidos; así como cancelar estos últimos.

Seguidamente se muestran una serie de figuras con el modelo de casos de uso. Se ha dividido dicho modelo en varias figuras para mejorar la visualización.

3.2.0.1 Ver Contenidos

En la figura 3.3, se muestran los casos de uso dirigidos a la exploración de los contenidos por parte de los usuarios tanto "Usuario" como "Cliente".

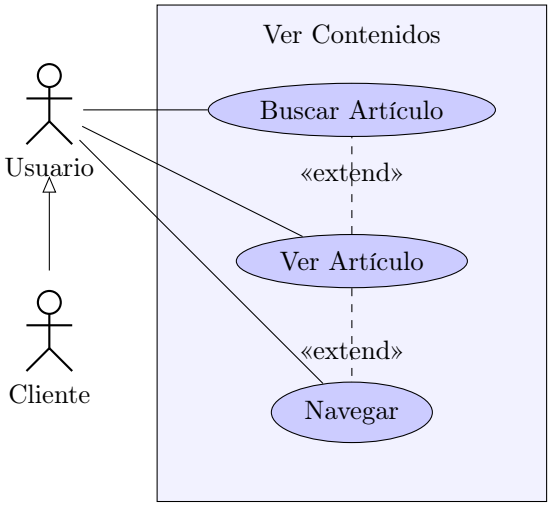

FIGURA 3.3: Casos de usos: Ver Contenidos

3.2.0.1.1 Caso de uso: Buscar Artículo

Descripción:

Los usuarios podrán utilizar un sistema de búsqueda personalizado para encontrar los artículos. Este sistema consta de una lista de selección que permitirá a los usuarios filtrar los elementos en función del tipo de artículo o formato; y un campo de texto para indicar un valor que será usado para encontrar los artículos

que contengan este valor como parte del título o donde el valor sea un autor, actor, director o guionista.

Actores:

En este caso, podrán interactuar los dos tipos de actores. Como no es necesario que el usuario este identificado, tanto el actor "Usuario" como el actor "Cliente", podrán realizar búsquedas.

Flujo principal:

Para poder encontrar los artículos deseados mediante este sistema, se realizarán los siguientes pasos:

1. El usuario visita la página principal.
2. El usuario Selecciona una opción de filtrado del menú desplegable - opcional - e introduce un termino de búsqueda.
3. El usuario presiona el botón de búsqueda.
4. El sistema devuelve una lista de los primeros elementos encontrados.
5. El usuario presiona los botones Next y Prev para solicitar los artículos siguientes o anteriores.
6. El usuario presiona el enlace ver artículo sobre el artículo deseado.
7. El sistema redirige al usuario a la página con la información del artículo.

Flujo Alternativo:

Este flujo se dará cuando el usuario introduzca un termino no válido en el campo de búsqueda.

1. El usuario visita la página principal.
2. El usuario Selecciona una opción de filtrado del menú desplegable - opcional - e introduce un termino de búsqueda.
3. El usuario presiona el botón de búsqueda.
4. El sistema devuelve un mensaje avisando que el termino de búsqueda no es válido.

3.2.0.1.2 Caso de uso: Ver Artículo

Descripción:

Los usuarios podrán visitar la página de un artículo directamente. Este caso se da tanto al ser redirigido a la página tras realizar la búsqueda de un artículo al presionar el botón "Ver Artículo"; como si se introduce directamente la dirección de una página en el navegador que , por ejemplo, se haya guardado previamente en el navegador.

Actores:

En este caso, podrán interactuar dos actores. Como no es necesario que el usuario este identificado, tanto el actor "Usuario" como el actor "Cliente", podrán visitar la página del artículo.

Flujo principal:

Al visitar la página de los artículos se realizarán los siguientes pasos:

1. El usuario es redirigido a la página o introduce la dirección en el navegador.
2. El sistema muestra los datos del artículo deseado. Entre ellos, el título, formato, precio, descripción y detalles como actores o autores.

Flujo Alternativo:

Este flujo ocurre cuando los datos al solicitar la página no son validos. Por ejemplo, si el identificador dado no corresponde a ningún artículo.

1. El usuario es redirigido a la página o introduce la dirección en el navegador.
2. El sistema muestra un mensaje indicando que el artículo solicitado no existe.

3.2.0.1.3 Caso de uso: Navegar

Descripción:

Los usuarios podrán utilizar el menú lateral de la página principal para realizar un búsqueda predefinida en función del género y del tipo de los artículos.

Actores:

En este caso, podrán interactuar dos actores. Como no es necesario que el usuario este identificado, tanto el actor "Usuario" como el actor "Cliente", podrán realizar búsquedas predeterminadas de artículos.

Flujo principal:

Para poder encontrar los artículos deseados mediante este sistema, se realizarán los siguientes pasos:

1. El usuario visita la página principal.
2. El usuario presiona uno de los enlaces de la barra lateral para indicar el género y el tipo del artículo.
3. El sistema devuelve una lista de los primeros elementos encontrados.
4. El usuario presiona los botones Next y Prev para solicitar los artículos siguientes o anteriores.
5. El usuario presiona el enlace ver artículo sobre el artículo deseado.

6. El sistema redirige al usuario a la página con la información del artículo.

Flujo Alternativo:

En este caso no hay flujo alternativo.

3.2.0.2 Gestión de Cesta

En la figura 3.4, se muestran los casos de uso dirigidos a la exploración de los contenidos por parte de los usuarios tanto "Usuario" como "Cliente".

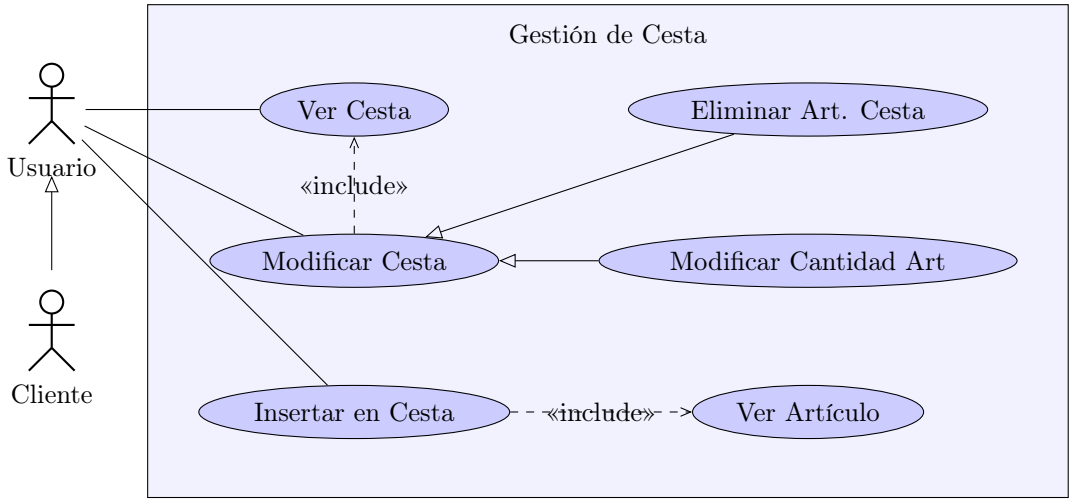

FIGURA 3.4: Casos de usos: Gestión de Cesta

3.2.0.2.1 Caso de uso: Ver Cesta

Descripción:

Los usuarios podrán ver la información que contiene el sistema sobre los artículos que hayan sido introducidos en la cesta, es decir, que hayan sido seleccionados previamente para ser comprados. Para el caso de los actores "Usuario" la información guardada en la cesta sólo será accesible durante la misma sesión. En el caso de los clientes "Cliente" será accesible siempre. En ambos casos, la información será borrada transcurrido un cierto tiempo sin que se haya actualizado la cesta.

Actores:

En este caso, podrán interactuar dos actores. Como no es necesario que el usuario este identificado, tanto el actor "Usuario" como el actor "Cliente", podrán visitar la página con la información de la Cesta.

Flujo principal:

El usuario puede ver la cesta pulsando el botón de la barra de herramientas "Cesta". Al pulsar dicho botón será redirigido a la página de la cesta y allí se le mostrará un lista con todos los artículos que han sido añadidos a la cesta.

Además, en dicha página se le mostrarán una serie de botones y enlaces para realizar diferentes operaciones como elimivar un artículo o modificar la cantidad elegida.

3.2.0.2.2 Caso de uso: Modificar Cesta

Descripción:

Los usuarios podrán modificar la cantidad elegida de cada artículo a comprar o eliminar alguno de los artículos de la cesta.

Actores:

En este caso, podrán interactuar dos actores. Como no es necesario que el usuario esté identificado, tanto el actor "Usuario" como el actor "Cliente", podrán modificar la cantidad elegida de un artículo.

Flujo:

Para modificar la cesta, los usuarios primero deben acceder a la página de la cesta para así poder ver la información de los artículos que se han seleccionado. Una vez hecho esto es posible modificar las cesta de dos maneras: (1) modificar la cantidad de cada artículo o (2) eliminar alguno de los artículos de la cesta.

Para modificar la cantidad elegida de cada artículo, junto a la información de cada artículo habrá un campo que muestre la cantidad elegida. El valor de este campo se puede cambiar y esto permite a los usuarios modificar la cantidad. Cuando un usuario modifica dicho campo y presiona el botón "modificar", el sistema modificará la información del artículo/s en la cesta siempre que la cantidad sea mayor a cero. Si la cantidad no es mayor a cero, el artículo se eliminará de la cesta. En ambos casos el tiempo de caducidad de la cesta se actualizará en el sistema.

Para eliminar uno de los artículos también se mostrará un botón para eliminar los artículos. En este caso el sistema se limitará a quitar dicho elemento de la cesta y al igual que en el caso anterior, si el número de artículos en la cesta es cero, entonces se eliminará la cesta.

Finalmente,en ambos casos, tras finalizar las operaciones, se mostrará al usuario la cesta con la información actualizada; o un mensaje si la cesta está vacía.

3.2.0.2.3 Caso de uso: Añadir a Cesta

Descripción:

Los usuarios podrán añadir nuevos artículos a la cesta.

Actores:

En este caso, podrán interactuar los dos actores. Como no es necesario que el usuario este identificado, tanto el actor "Usuario" como el actor "Cliente", podrán añadir elementos a la cesta.

Flujo:

Sólo es posible añadir artículos a la cesta desde la misma página del artículo. Por tanto, antes de realizar dicha tarea sería necesario ver el artículo.

Un vez en la página del artículo el usuario puede presionar el botón "añadir a cesta". Al hacer esto el sistema añadirá este artículo siempre que la cesta esté creada, de lo contrario no la creará. Sin embargo, si el artículo no está disponible no se añadirá.

Tras ésto, se mostrará una alerta al usuario para indicar que se ha añadido el artículo correctamente. Pero si ha habido algún error durante el proceso de inserción, se mostrará una alerta con un mensaje de error indicando que no se ha podido añadir el elemento a la cesta.

3.2.0.3 Control de usuarios

En la figura 3.5, se muestra los casos de uso dirigidos al control de usuarios.

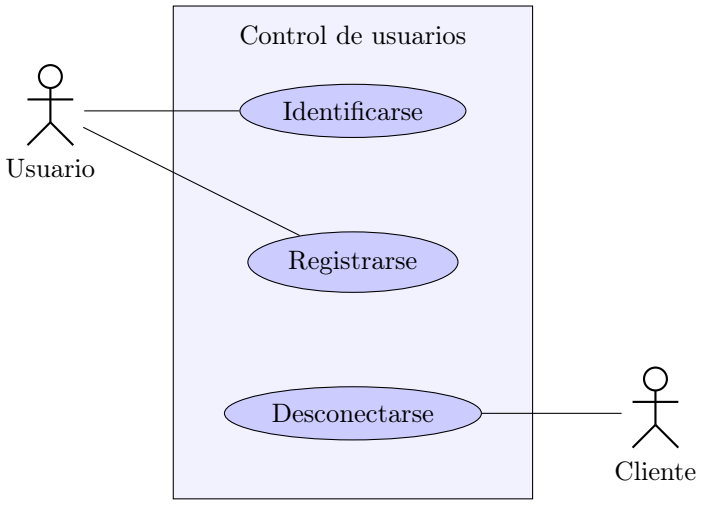

FIGURA 3.5: Casos de usos: Control de usuarios

3.2.0.3.1 Caso de uso: Identificarse

Descripción:

Los usuarios podrán identificarse como "Cliente" mediante un usuario y una contraseña que les permitirá tener acceso a la zona de Cliente y realizar compras.

Actores:

En este caso, únicamente los actores "Usuarios" podrán identificarse, ya que los usuarios "Cliente" ya están identificados.

Flujo:

Desde cualquier página el usuario tendrá la opción de acceder a la página de "Identificación" mediante el botón de la barra superior "login".

En está página habrá un formulario con dos campos que permitirán al usuario introducir el email de usuario - identificador - y la contraseña. Una vez pulsado el botón "aceptar", el sistema comprobará si las credenciales son válidas. Si lo son se mostrará un mensaje indicado que se ha identificado correctamente. Si no, mostrará un mensaje de error junto con el formulario.

Alternativamente, si se intenta realizar una compra o acceder a la zona de clientes sin estar registrados, el usuario será redirigido a la página de "Identificación".

Además, tras identificar al usuario si esté había creado una cesta y si el cliente tenía ya una cesta guardada con anterioridad, el sistema unificará ambas cestas en una única.

3.2.0.3.2 Caso de uso: Registrarse

Descripción:

Si los usuarios no tiene credenciales para identificarse como clientes; estos podrán registrarse, creando para ello un nuevo "Cliente".

Actores:

En este caso, únicamente los actores "Usuarios" podrán registrarse, ya que los usuarios "Cliente" ya están registrados.

Flujo:

Desde cualquier página el usuario tendrá la opción de acceder a la página de "Registro" mediante el botón de la barra superior "Registro".

En está página habrá un formulario con dos campos que permitirán al usuario introducir un email y un nombre de usuario, y la contraseña deseada. Una vez pulsado el botón "aceptar", el sistema comprobará que el formato de la información es correcto y que no existe un usuario con dicho email. Si todo es correcto se mostrará un mensaje indicando que se ha registrado correctamente. Si no, mostrará un mensaje de error junto con el formulario.

Alternativamente, en la página de "Identificación", también se mostrará un enlace a la página de "Registro".

3.2.0.3.3 Caso de uso: Desconectarse

Descripción:

Un usuario "Cliente" que esté identificado, tendrá la opción de desconectarse y pasar a ser un usuario "Usuario".

Actores:

En este caso, únicamente el actor "Clientes" podrán desconectarse.

Flujo:

En todas las páginas del sitio Web se mostrará el botón "Desconectar" - si el usuario está identificado. Al pulsar dicho botón el sistema borrará las credenciales de la información de sesión. Finalmente, al usuario no se le mostrará ningún mensaje, pero el botón "Desconectar" desaparecerá y en su lugar aparecerá el botón "Login".

Además, si el usuario estaba en la zona de clientes, éste será redirigido a la página principal.

3.2.0.4 Clientes

En la figura 3.6, se muestran los casos de uso relacionados con las operaciones que pueden realizar los usuarios de tipo "Cliente".

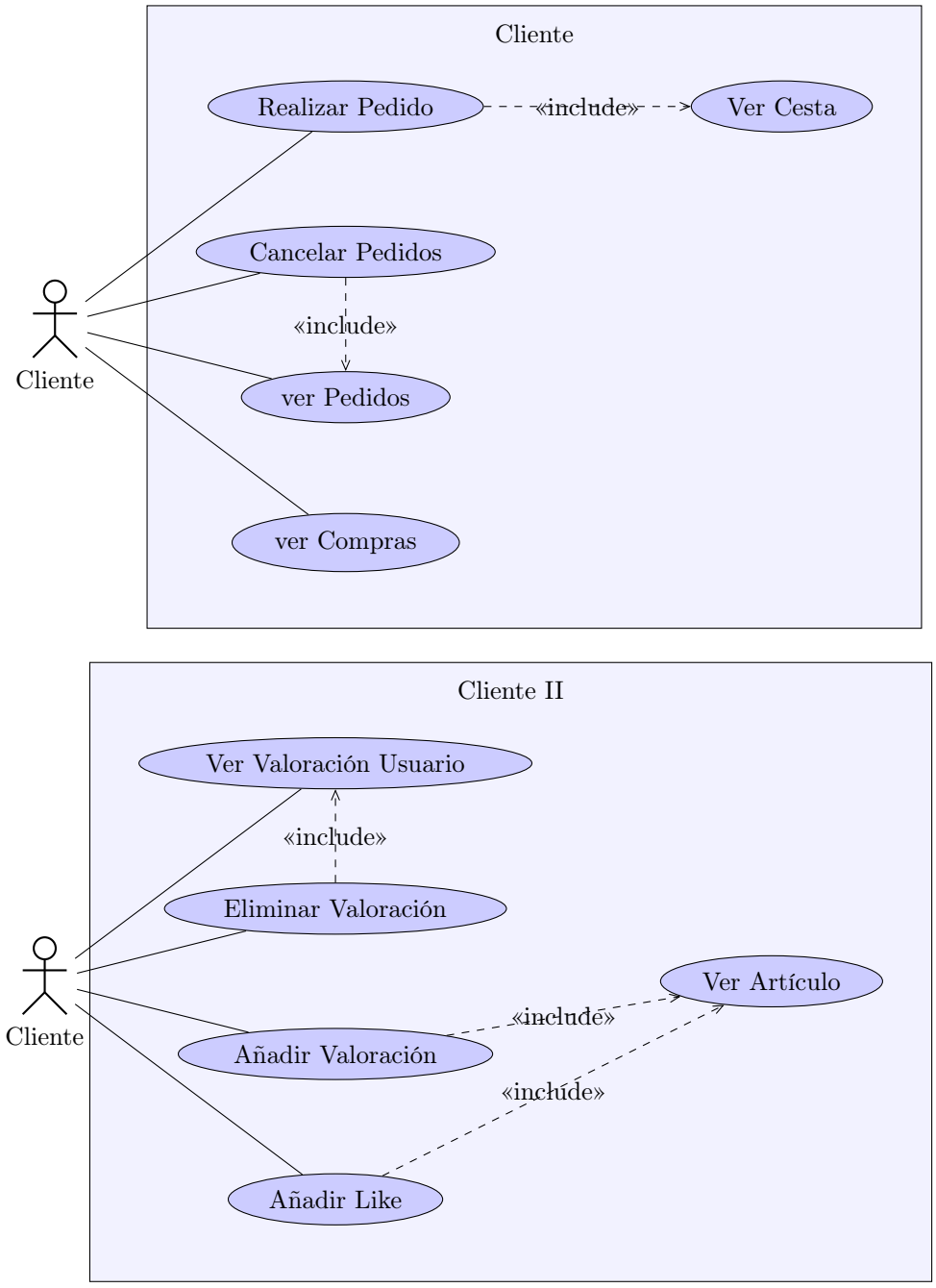

FIGURA 3.6: Casos de usos: Operaciones para Clientes

3.2.0.4.1 Caso de uso: Realizar Pedido

Descripción:

Los usuarios que estén identificados como clientes, podrán realizar un pedido con los artículos que contiene la cesta de la compra.

Actores:

Aunque todos los usuarios pueden ver y modificar la cesta, únicamente se permite a usuarios registrados de tipo "Cliente" realizar el pedido.

Flujo:

Para poder realizar la compra de los artículos que contiene la cesta, los usuarios deberán acceder primero a la página de la "Cesta". Una vez en esta página pueden pulsar el botón "Realizar Pedido", lo cual les permitirá comenzar con los tramites para realizar la compra.

Primero, al usuario se le mostrará un formulario dónde introducir los datos de envío y los datos de pago. Una vez rellenados los campos del formulario el cliente puede pulsar el botón "Comprar" para realizar la compra o "Cancelar" para cancelar el pedido.

Si pulsa "Comprar", el sistema comprobará los datos del formulario y la disponibilidad de los artículos. Si algún dato es invalido el sistema devolverá un mensaje de error junto al formulario. Si son correctos, pero algún artículo ya no está disponible mostrará una nueva página con información indicando los artículos que no estén disponibles y el precio total sin estos. Junto a esto se dará la opción de cancelar el pedido, o realizar la compra sólo con los artículos disponibles.

Finalmente, si todo es correcto se mostrará una última página de confirmación para finalizar la compra. Si el cliente pulsa el botón "Confirma", el sistema guardará el pedido como pendiente y se mostrará un mensaje de éxito. Si el cliente pulsa el botón "Cancelar" el cliente será redirigido a la página de la cesta.

Por otro lado, si un usuario "Usuario" pulsa el botón "Realizar Pedido" de la página de la cesta, será redirigido a la página de "Identificación" para que se identifique como cliente.

3.2.0.4.2 Caso de uso: Ver Pedido

Descripción:

Los usuarios que estén identificados como clientes, podrán ver pedidos realizados recientemente.

Actores:

Únicamente los actores "Clientes" pueden ver los pedidos ya que son los únicos que pueden realizar un pedido.

Flujo:

Para ver los pedidos realizados, los clientes deben acceder a la página "Zona Cliente". En ella estará disponible el botón "Pedidos" que le permitirá acceder a la información de los pedidos en curso - pendiente, enviado o recogido.

3.2.0.4.3 Caso de uso: Cancelar Pedido

Descripción:

Los usuarios que estén identificados como clientes, podrán cancelar pedidos suyos que estén pendientes.

Actores:

Únicamente los actores "Clientes" pueden cancelar un pedido ya que son los únicos que pueden realizar un pedido.

Flujo:

Para cancelar a la página "Pedidos" de la zona de clientes. El cliente podrá cancelar aquellos pedidos que estén pendientes pulsando el link "Cancelar" que aparecerá junto a este tipo de pedidos.

3.2.0.4.4 Caso de uso: Ver Compras

Descripción:

Los usuarios que estén identificados como clientes, podrán ver los artículos que hayan comprado.

Actores:

Únicamente los actores "Clientes" pueden ver los artículos comprados ya que son los únicos que pueden realizar un pedido.

Flujo:

Para ver los artículos comprados, los clientes deben acceder a la página "Zona Cliente". En ella estará disponible el botón "Artículos" que les permitirá ver una lista con información de los artículos comprados.

3.2.0.4.5 Caso de uso: Ver Valoraciones del Usuario

Descripción:

Los usuarios que estén identificados como clientes, podrán ver una lista con las valoraciones que han realizado sobre los distintos artículos.

Actores:

Ya que hay que estar identificado para realizar una valoración, sólo los usuarios registrados de tipo "Cliente" podrán ver las valoraciones en función de su nombre de usuario.

Flujo:

Para poder ver las valoraciones escritas, el cliente deberá acceder a la página "Zona Cliente". Una vez en ella en la barra lateral estará disponible el botón "Valoraciones" que al ser pulsado mostrará la lista de todas las valoraciones realizada.

3.2.0.4.6 Caso de uso: Eliminar Valoraciones

Descripción:

Los usuarios que estén identificados como clientes, podrán eliminar en cualquier momento cualquiera de las valoraciones que hayan escrito sobre un producto.

Actores:

Ya que hay que estar identificado para realizar una valoración, sólo los usuarios registrados de tipo "Cliente" podrán eliminar las valoraciones que hayan escrito.

Flujo:

Al ver la lista de valoraciones escritas, tal y como se indicaba en el caso de uso anterior, junto a cada valoración se mostrará un enlace que permitirá al usuario eliminar dicha valoración.

3.2.0.4.7 Caso de uso: Añadir Valoración

Descripción:

Los usuarios que estén identificados como clientes, podrán añadir una valoración a un artículo indicando el "rating" que le den e incluyendo una descripción o crítica del producto.

Actores:

Únicamente los usuarios registrados de tipo "Cliente" podrán realizar una valoración de un producto.

Flujo:

Para poder realizar una valoración, el cliente deberá primero visitar la página del artículo. En esta página se mostrará, junto con la lista de valoraciones del artículo, el botón "Realizar Valoración". Si el usuario lo pulsa se mostrá un campo de texto para escribir la crítica del producto y un menú seleccionable para elegir el "rating" a dar. Junto a estos campos de entrada se mostrarán los botones "Aceptar" y "Cancelar", para que el cliente guarde o cancele la valoración.

3.2.0.4.8 Caso de uso: Añadir Like

Descripción:

Los usuarios que estén identificados como clientes, podrán indicar de forma sencilla que un artículo les ha gustado.

Actores:

Únicamente los usuarios registrados de tipo "Cliente" podrán indicar que les ha gustado un artículo.

Flujo:

Para poder indicar que un producto le ha gustado, el cliente deberá primero visitar la página del producto. En esta página se mostrará el botón "Like" que permite al usuario indicar que le ha gustado el producto al pulsarlo. Al hacer esto, el sistema guardará está información la cual permitirá mostrar en el sistio web recomendaciones personalizadas.

3.3 Modelo de Datos

En una aplicación normal con un único sistema de bases de datos relacional habría que hacer un único modelo de datos para todo el sistema. Sin embargo, para esta aplicación donde hay varios sistemas, relacionales y no relacionales al mismo tiempo, habrá que hacer varios modelos.

Por tanto, una vez decididos qué datos serán guardados en cada sistema - tal y como se muestra en el punto 3.1.2 - habrá que: (1) diseñar un modelo de datos para cada sistema; y (2) diseñar como están relacionados los datos de los distintos sistemas.

En esta sección se muestran los diferentes diseños que se han realizado para modelar los datos de la aplicación.

3.3.1 Modelo de datos en MongoDB

En MongoDB se guardarán los datos de los artículos para presentar los artículos a los usuarios, los datos de las cestas de la compra de cada usuario y los datos de los pedidos recientes.

Como se observa en la figura 3.7 para representar estos tres objetos se han creado únicamente tres colecciones.

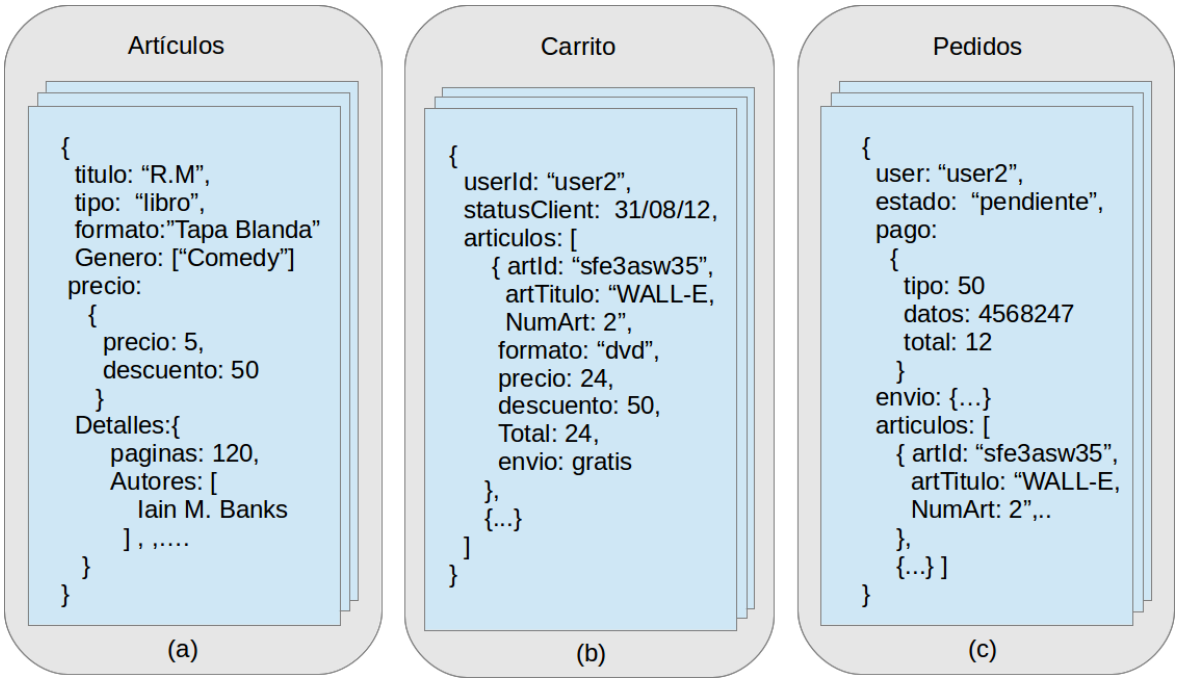

FIGURA 3.7: Modelo de datos en MongoDB:(a) artículos, (b) carritos, (c) pedidos

3.3.1.1 Colección Artículos

La colección artículos(a) contiene documentos que representan a los artículos del comercio electrónico. Estos artículos serán principalmente de dos tipos, películas y libros. La información que representa cada tipo es similar pero varía en algunos datos. A pesar de esto, debido a la flexibilidad que da el sistema de documentos es posible representar ambos tipos en la misma colección.

El esquema principal de los documentos es el siguiente:

- _id: identificador único del documento, el cual no aparece en la figura porque lo asigna automáticamente el sistema.
- titulo: título de la película o del libro.
- tipo: tipo de artículo, película o libro.
- formato: el formato del artículo, que puede ser dvd, blueray o collection para las películas y tapa dura o tapa blanda para los libros.
- genero: una lista conteniendo todos los géneros a los que pertenece el artículo.
- detalles: un subdocumento que contiene información sobre el artículo.
- precio: un subdocumento con información sobre el precio del producto. Este subdocumento contiene los campos:
  - precio: el precio del artículo. Como se puede ver el nombre del campo de un subdocumento puede tener el mismo nombre que cualquier campo del documento que no esté dentro de ese mismo subdocumento. Ésto se debe a que internamente el campo del subdocumento es "precio.precio" y no simplemente "precio".
  - descuento: el descuento que tiene el artículo.
  - envio: si el envío es gratis o de pago.

La diferencia principal entre los artículos de un tipo y otro se encuentran en el subdocumento del campo "detalles".

El esquema del subdocumento detalles para las películas es el siguiente:

- actores: lista de actores que aparecen en la película.
- productora: lista de las productoras que tomaron parte en la película.
- duración: duración en minutos de la película.
- idioma: idioma de la película.
- fecha: fecha de estreno de la película.
- director: lista de los directores de la película.
- guionista: lista de los guionistas de la película.

- descripción: descripción de la trama de la película.

En el caso del los libros el esquema de detalles será el siguiente:

- autores: lista de autores del libro.
- editorial: nombre de la editorial del libro.
- paginas: número de las páginas del libro.
- idioma: idioma del libro.
- fecha: fecha de lanzamiento del libro.
- descripción: descripción de la trama del libro.

3.3.1.2 Colección Carrito

La colección "carrito" contiene los documentos que representan las cestas de la compra la cual contiene una lista con los artículos seleccionados e información del cliente. Además, estos documentos utilizan índices TTL que harán que los documento se borren transcurrido un tiempo.

El esquema del documento para las cestas de las compras es el siguiente:

- _id: identificador único del documento.
- userId: identificador único del usuario que ha creado la cesta. Este identificador puede ser un identificador de sesión para usuarios no conectados, o el email para clientes conectados.
- status/statusClient: son utilizados por los índices TTL para calcular el tiempo de caducidad del documento. Sólo aparecerá uno de los campos en el documento, dependiendo de si la cesta pertenece a un usuario normal o a un cliente conectado, ya que la cesta de los clientes tiene un tiempo de caducidad mayor.
- articulos: contiene una lista de subdocumentos los cuales representan cada uno de los artículos elegidos.
  - artId : identificador del artículo.
  - artTitulo : título del artículo.
  - numArt : cantidad elegida de cada artículo.
  - formato : formato del artículo.

  - precio : precio.
  - descuento : descuento.
  - total : precio final del artículo aplicado el descuento.
  - envio : tipo de envío, de pago o gratis.

3.3.1.3 Colección Pedidos

La colección pedidos contiene documentos que representan los pedidos hechos por los clientes. En estos documentos se guardan datos referentes al envío, el pago, los clientes que realizan cada pedido y los artículos comprados.

El esquema del documento es el siguiente:

- _id: identificador único del documento.
- user: identificador - email - del cliente que ha realizado la compra.
- estado: indica el estado actual del pedido - procesando, cancelado, enviado o recibido.
- articulos: contiene una lista de subdocumentos los cuales representan cada uno de los artículos elegidos.
  - artId : identificador del artículo.
  - artTitulo : título del artículo.
  - NumArt : cantidad elegida de cada artículo.
  - formato : formato del artículo.
  - precio : precio.
  - descuento : descuento.
  - Total : precio final del artículo aplicado el descuento.
  - envio : tipo de envío, de pago o gratis.
- envio: contiene la información de envío del pedido.
  - nombre: nombre de la persona a la que se envía el pedido.
  - direccion: dirección a la que se envía el pedido.
- pago: contiene la información de pago del pedido.
  - tipo: el tipo de pago - tarjeta, contra reembolso, pago electrónico.
  - datos: datos del pago.
  - total: cantidad total pagada por el pedido.

3.3.2 Modelo de datos en Virtuoso

En Virtuoso se guardan los datos de los clientes así como las valoraciones y las compras que realizan éstos. Además, para representar estos datos se ha creado un único grafo que se dividirá en varios subgrafos.

El esquema de cada uno de los subrafos es el siguiente:

- Artículos art: <http://applV.com/art/> : contiene información sobre el identificador del artículo en mongodb, el título, el tipo de artículo y los formatos disponibles.

```
art:id <http://applV.com/art/article/id>   "idMongo"
art:id  <http://applV.com/art/article/title> "Titulo"
art:id  <http://applV.com/art/article/type>  "movie/book"
art:id  <http://applV.com/art/article/format> "formato"
```

- Compras buy:<http://appl.com/purchase/>: contiene información sobre el artículo comprado, la fecha, precio, y el formato.

```
buy:id <http://applV.com/purchase/buy/article>  <http://applV.com/\
    art/id>
buy:id <http://applV.com/purchase/buy/date>     "31/08/2013"
buy:id <http://applV.com/purchase/buy/price>     57
buy:id <http://applV.com/purchase/buy/formato>  "formato"
```

  Como se puede ver el nodo objeto de la primera tripleta es el subgrafo de un artículo.

- Valoraciones rev:<http://appl.com/rev/idReview>: contiene información sobre el artículo valorado, la opinión sobre el producto, la valoración y el formato.

```
rev:id <http://applV.com/rev/review/of>  <http://applV.com/art/id>
rev:id <http://applV.com/rev/review/description> "Descripcion"
rev:id <http://applV.com/rev/review/rating>       6
rev:id <http://applV.com/rev/review/formato>    "Formato"
```

  Como se puede ver el nodo objeto de la primera tripleta es el subgrafo de un artículo.

- Usuarios <mailto:userID@email.com>: contiene el nombre del usuario, la contraseña y las acciones sobre los productos.

```
<mailto:userID@email.com> <http://applV.com/users/pwd>          \
    "123"
<mailto:userID@email.com> foaf:name                            "\
    Nombre"
```

```
<mailto:userID@email.com> <http://applV.com/actions/like>       art\
    :id
<mailto:userID@email.com> <http://applV.com/actions/reviewing>  rev\
    :id
<mailto:userID@email.com> <http://applV.com/actions/buying>     buy\
    :id
```

En este caso las tres últimas tripletas indican que el usuario ha hecho una acción <http://applV.com/actions/> y qué acción es la que se ha realizado - like, buying, reviewing.

En la siguiente figura 3.8 se puede ver una representación del grafo especificado anteriormente.

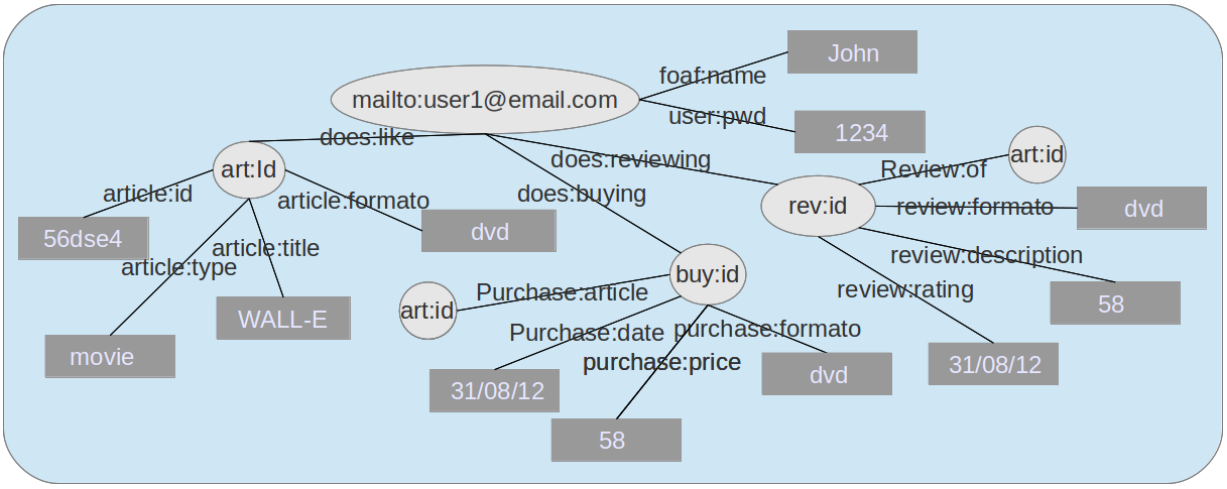

FIGURA 3.8: Modelo de datos en Virtuoso

3.3.3 Modelo de datos en MySQL

En éste sistema se guarda la información relacionada con el inventario de los artículos, es decir, la cantidad disponible de cada artículo. Por tanto, no es necesario tener un esquema muy complicado y es suficiente con tener una tabla con dos campos: (1) el identificador del artículo; y (2) la cantidad de artículos disponibles.

Por tanto el esquema será el presente en la figura 3.9

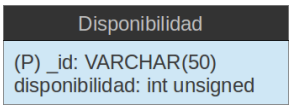

FIGURA 3.9: Modelo de datos en MySQL

3.3.4 Relación entre los datos

En lo que se refiere a la relación entre los datos, en la implementación actual los datos que se guardan en los distintos sistemas de almacenamiento no están relacionados entre ellos. Por ejemplo, la información de los usuarios sólo está presente en Virtuoso. La principal relación entre los datos se encuentra en los artículos ya que para un mismo artículo puede haber datos tanto en MongoDB como en Virtuoso o MySQL. Lo ideal sería mantener un identificador de artículo que sea consistente entre los diferentes sistemas, de forma que mediante este identificador se identificará a un mismo artículo en todos los sistemas. Por otro lado, no se intenta normalizar los datos y que no haya repetición de datos, de forma que datos como el título de un artículo pueden estar presentes en varios sistemas.

3.4 Modelo de Clases

En esta sección se explica el modelo de clases de la aplicación. Para simplificar la presentación se ha dividido el diagrama que representa el modelo de clases en tres partes: servicios, objetos, aplicación web.

3.4.1 Servicios

En el diagrama de la figura 3.10 se puede ver el diagrama de clases correspondiente a los servicios encargados de realizar las consultas sobre los distintos sistemas de bases de datos.

Como se puede observar, hay dos tipos de clases: las encargadas de realizar las conexiones a cada uno de los tres sistemas de bases de datos y los servicios que realizan consultas sobre las bases de datos.

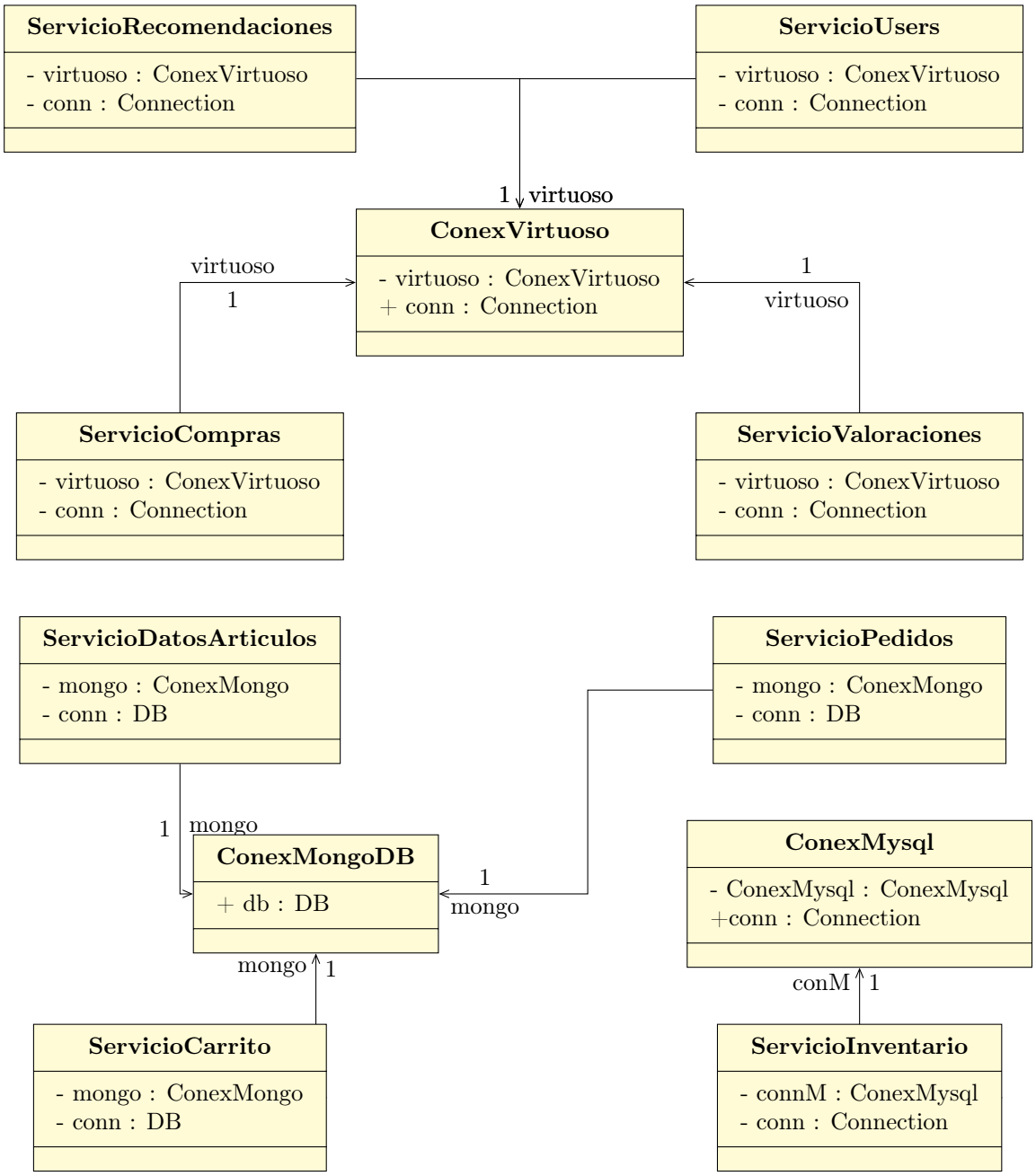

FIGURA 3.10: Diagrama de Clases: Servicios

3.4.1.1 Clase ConexVirtuoso

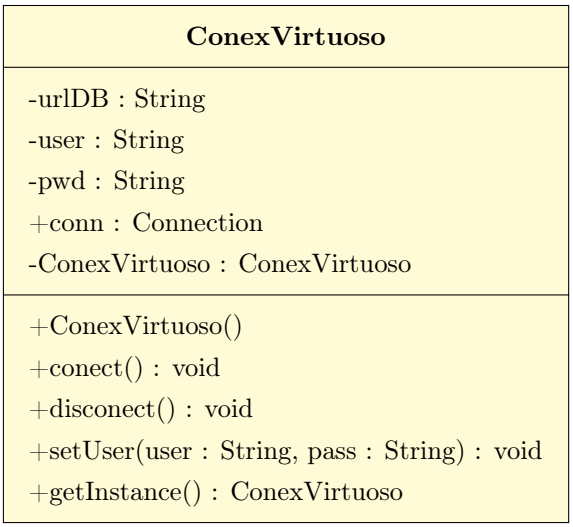

FIGURA 3.11: Clase: ConexVirtuoso

Esta clase implementa la conexión al sistema de bases de datos Virtuoso. Los atributos *user*, *pwd*, *urlDB* son privados y son utilizados por la clase para realizar la conexión a la base de datos en la dirección *usrlDB*. La aplicación usa el usuario *user* y la contraseña *pwd* como datos de identificación al realizar la conexión.

El atributo *ConexVirtuoso* representa una instancia de la clase que se crea automáticamente y que se puede obtener mediante el método *getInstance()*, de forma que no es necesario llamar al constructor cada vez que se crea un objeto.

Por último, el atributo publico *db* representa la conexión establecida a la base de datos y es usado por los servicios para realizar las consultas a través de esa conexión.

3.4.1.2 Clase ConexMongoDB

Esta clase implementa la conexión al sistema de bases de datos MongoDB. El atributo *Server* indica la dirección donde está la instancia del sistema MongoDB. Además, Los atributos *user*, *pwd* contienen las credenciales de identificación para conectarse a la base de datos mediante el método *auth()*.

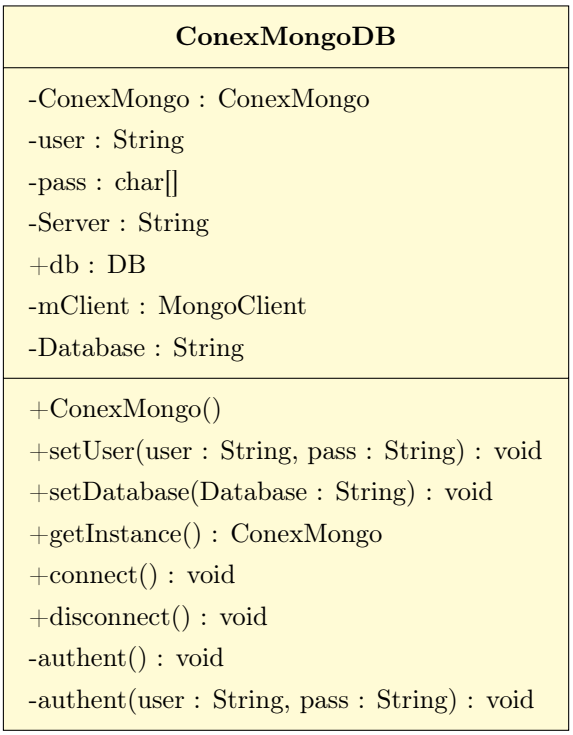

FIGURA 3.12: Clase: ConexMongoDB

Al igual que ocurría en el caso anterior está disponible un método *getInstance()*, el cuál permite obtener una instancia de la clase con una conexión ya creada.

Por último, el atributo publico *conn* representa la conexión establecida a la base de datos y es usado por los servicios para realizar las consultas a través de esa conexión.

3.4.1.3 Clase ConexMySQL

Esta clase implementa la conexión al sistema de bases de datos MySQL. Al igual que en los casos anteriores, se tiene el método *getInstance()* y el atributo *conn*, los cuales permiten acceder a la conexión establecida con el sistema MySQL.

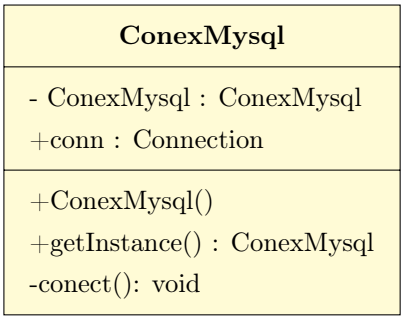

FIGURA 3.13: Clase: ConexMySQL

3.4.1.4 Clase ServicioUsers

ServicioUsers
- virtuoso : ConexVirtuoso - conn : Connection
+ServicioUsers() +addUser(user : String, name : String, pass : String) : void +login(user : String, pass : String, Context : String) : bool +usuarioRegistrado(user : String, Context : String) : boolean

FIGURA 3.14: Clase: ServiciosUsers

Esta clase implementa el servicio que se encarga de realizar las consultas relacionadas con los usuarios. Los dos atributos que contiene la clase son utilizados de forma privada y guardan información sobre la conexión en la que hacer las consultas.

Los métodos de la clase están enfocados a obtener y modificar información de los usuarios en la base de datos. En concreto estos métodos permiten: añadir/eliminar/modificar un usuario, comprobar las credenciales del usuario y comprobar si un usuario ya está presente en la base de datos.

3.4.1.5 Clase ServicioValoraciones

Esta clase implementa el servicio que se encarga de realizar las consultas relacionadas con las valoraciones. Los dos atributos que contiene la clase son utilizados de forma privada y guardan información sobre la conexión en la que hacer las consultas.

Los métodos de la clase están enfocados a obtener y modificar información de las valoraciones en la base de datos. En concreto estos métodos permiten: añadir/eliminar/ modificar una valoración, obtener todas las valoraciones de un artículos o las hechas por un usuario dado, añadir un "like" y obtener el valor total del "rating" dado a un artículo.

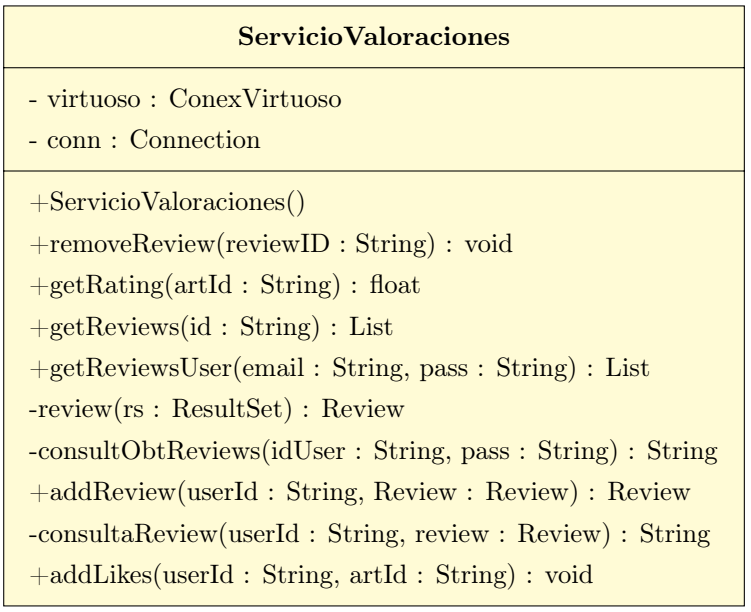

FIGURA 3.15: Clase: ServicioValoraciones

3.4.1.6 Clase ServicioCompras

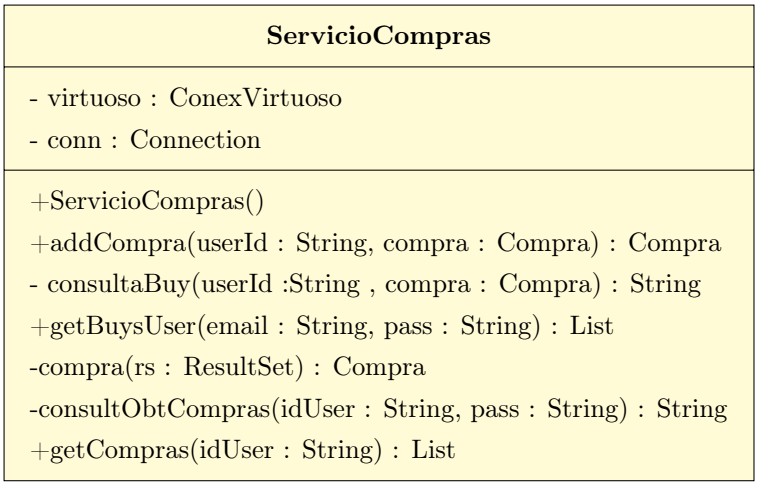

FIGURA 3.16: Clase: ServicioCompras

Esta clase implementa el servicio que se encarga de realizar las consultas relacionadas con los artículos comprados por cada usuario. Los dos atributos que contiene la clase son utilizados de forma privada y guardan información sobre la conexión en la que hacer las consultas.

Los métodos de la clase están enfocados a obtener y modificar información de los artículos comprados. En concreto estos métodos permiten: añadir/eliminar/modificar una compra

y obtener todas las compras realizadas por un usuario tanto si se le da únicamente el nombre de usuario, como si se le da el nombre de usuario y la contraseña.

3.4.1.7 Clase ServicioRecomendaciones

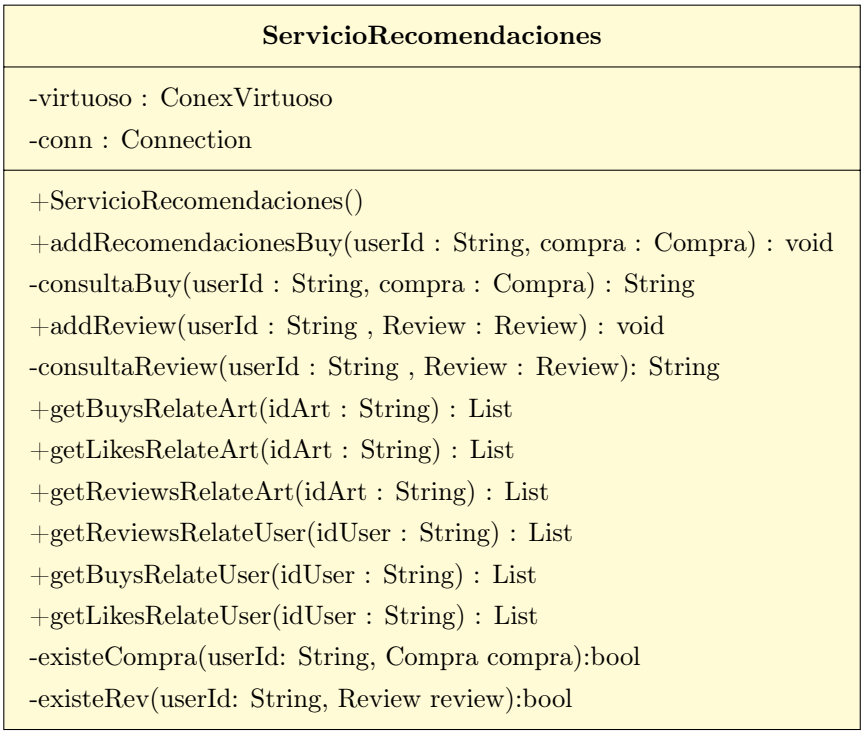

FIGURA 3.17: Clase: ServicioRecomendaciones

Esta clase implementa el servicio que se encarga de realizar las consultas que permiten dar recomendaciones sobre artículos que les pueden interesar a los usuarios. Los dos atributos que contiene la clase son utilizados de forma privada y son los que permiten realizar consultas sobre la conexión. Los métodos permiten obtener recomendaciones de artículos tanto en función de un artículo dado como en función de un usuario. En el primer caso dado un artículos se buscan a todos los usuarios los cuales hayan comprado/ valorado/gustado dicho artículos y se devuelve otros artículos que estos mismos usuarios hayan comprado/valorado/gustado. En el segundo caso, se tienen en cuenta todos los artículos que un usuario dado haya comprado/valorado/gustado y por cada artículos se hace un busqueda similar a la anterior.

3.4.1.8 Clase ServicioPedidos

Esta clase implementa el servicio que se encarga de realizar consultas sobre los pedidos hechos por los usuarios. Al igual que en los casos anteriores los atributos son utilizados

para realizar las consultas sobre la conexión al sistema. En este caso, en lugar de tener la conexión al sistema se tiene la conexión a la base datos en el atributo *db*. Por otro lado, además se tiene el atributo *CollectionName* que contiene el nombre de la colección en la base de datos *db* sobre la que se harán las consultas; y el atributo *pedidos* que contiene una instancia de la colección sobre la que se realizarán las consultas.

Los métodos presentes en esta clase insertan eliminan y modifican pedidos realizados por los clientes.

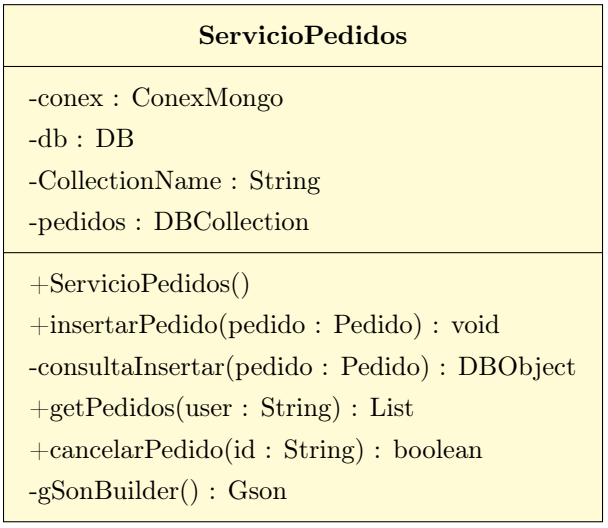

FIGURA 3.18: Clase: ServicioPedidos

3.4.1.9 Clase ServicioCarrito

Esta clase implementa el servicio que se encarga de realizar consultas sobre las cesta de la compra creadas por los usuarios. Al igual que en los casos anteriores los atributos son utilizados para realizar las consultas sobre la conexión al sistema. En este caso, en lugar de tener la conexión al sistema se tiene la conexión a la base de datos en el atributo *db*. Por otro lado, además se tiene el atributo *CollectionName* que contiene el nombre de la colección en la base de datos *db* sobre la que se harán las consultas; y el atributo *carrito* que contiene una instancia de la colección sobre la que se realizarán las consultas.

Los métodos presentes en esta clase insertan eliminan y modifican las cestas de la compra realizadas por los usuarios. Mediante los métodos de modificación es posible eliminar/añadir un articulo de la cesta, modificar la cantidad elegida de un artículo, modificar el usuario y reiniciar el tiempo de caducidad del documento. Destaca además, el método *unificarCesta* que dados dos usuarios permite unificar las cestas de dos usuarios en una sola. Ésto evita que un cliente pierda una cesta guardada por el hecho de que otro usuario

haya creado una cesta desde el mismo ordenador; y que al mismo tiempo el cliente no pierda las cestas antes de identificarse.

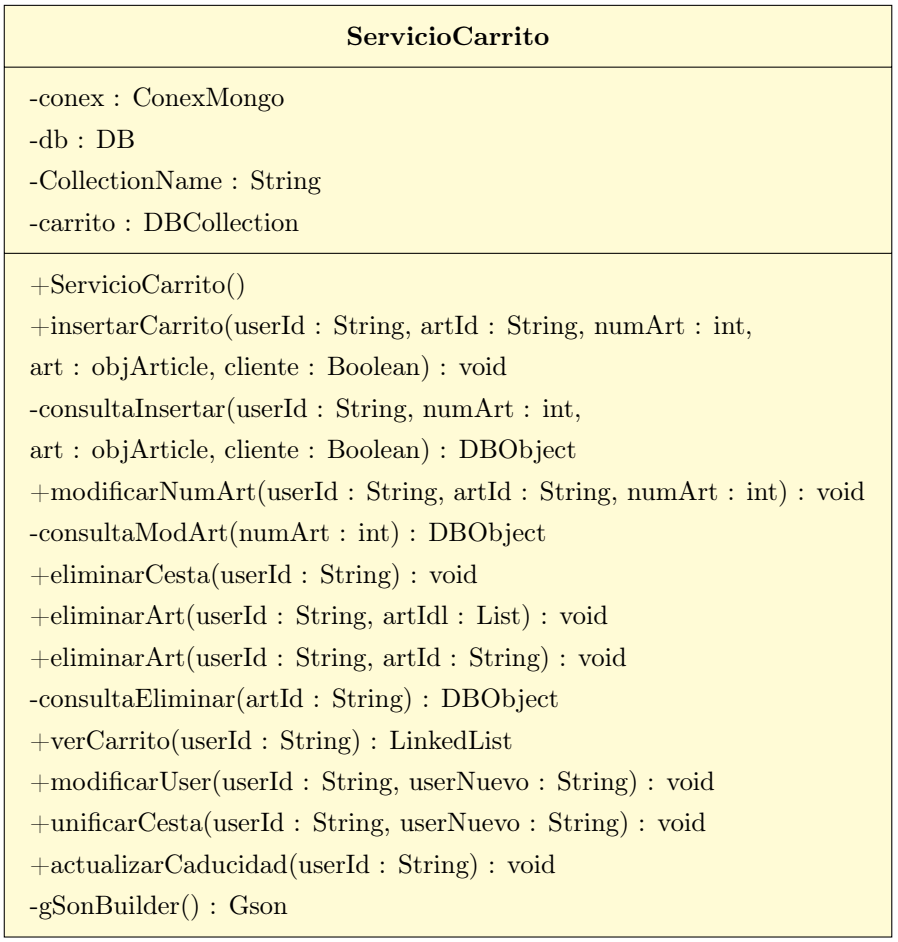

FIGURA 3.19: Clase: ServicioCarrito

3.4.1.10 Clase ServicioDatosArticulos

Esta clase implementa el servicio que se encarga de realizar consultas sobre los artículos guardados en el sistema. Al igual que en los casos anteriores los atributos son utilizados para realizar las consultas sobre la conexión al sistema. En este caso, en lugar de tener la conexión al sistema se tiene la conexión a la base de datos en el atributo *db*. Por otro lado, además se tiene el atributo *CollectionName* que contiene el nombre de la colección en la base de datos *db* sobre la que se harán las consultas; y el atributo *productos* que contiene una instancia de la colección sobre la que se realizarán las consultas.

Los métodos de este servicio permiten ver/modificar/eliminar artículos. Entre los métodos destacan los de búsqueda con los cuales, además de obtener los datos de un artículo en función de su identificador, también es posible realizar dos búsquedas, la primera busca

los artículos que coinciden con una lista de filtros, por ejemplo para buscar artículos de tipo libro y formato "Tapa Dura". La segunda permite realizar búsquedas de texto, esto permite buscar artículos que contienen una palabra dada en ciertos campos del artículo - por ejemplo libros que contienen la palabra "Potter" en su título.

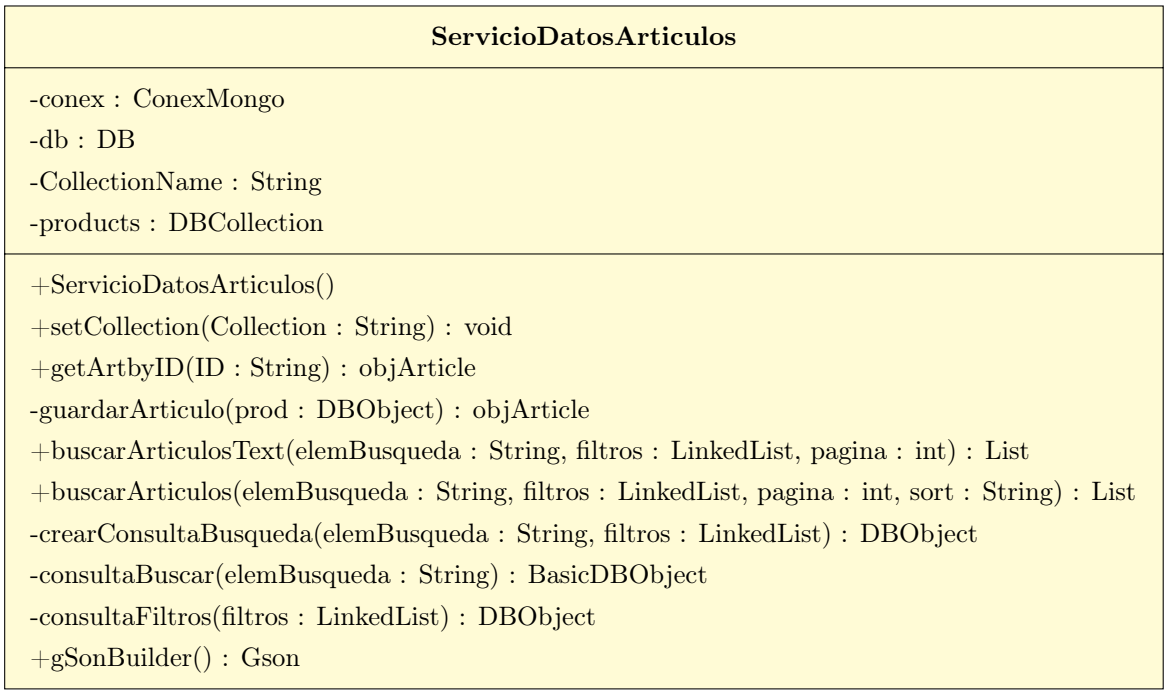

FIGURA 3.20: Clase: ServicioDatosArticulos

3.4.1.11 Clase ServicioInventario

Esta clase implementa el servicio que se encarga de modificar el inventario y consultar la disponibilidad de los artículos. Al igual que ocurre con los servicios de Virtuoso, esta clase sólo contiene dos atributos: el que contiene la instancia a la clase conexMysql y el que contiene la instancia de la conexión a MySQL.

Esta clase tiene un método que le permite obtener la disponibilidad de un artículo en función de un artículo o una lista de artículos. Sin embargo, el método que destaca es el de *modificarInventario()* el cual permite modificar la cantidad de varios artículos de forma atómica y manteniendo la consistencia entre los distintos usuarios.

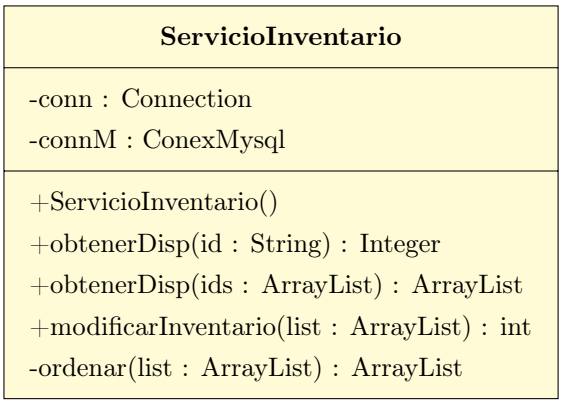

FIGURA 3.21: Clase: ServicioDatosArticulos

3.4.2 Clases de Objetos

Estas clases se utilizan para representar los diferentes objetos de los que hace uso la aplicación, tanto los que representan a objetos como artículos o pedidos como aquellos que se utilizan para facilitar la implementación de los servicios como es el caso de "par".

En la figura 3.22 se muestra el diagrama de clases que representa a los objetos. En ésta figura únicamente se muestran los atributos de las clases y no los métodos. Estos métodos en general son del tipo *getNombreAtributo* y *setNombreAtributo*, y permiten obtener y asignar valores a los atributos. Para reducir la información mostrada, se han obviado dichos métodos. Un caso a parte que se mostrará más adelante es la clase *Par*, la cual contiene otros métodos.

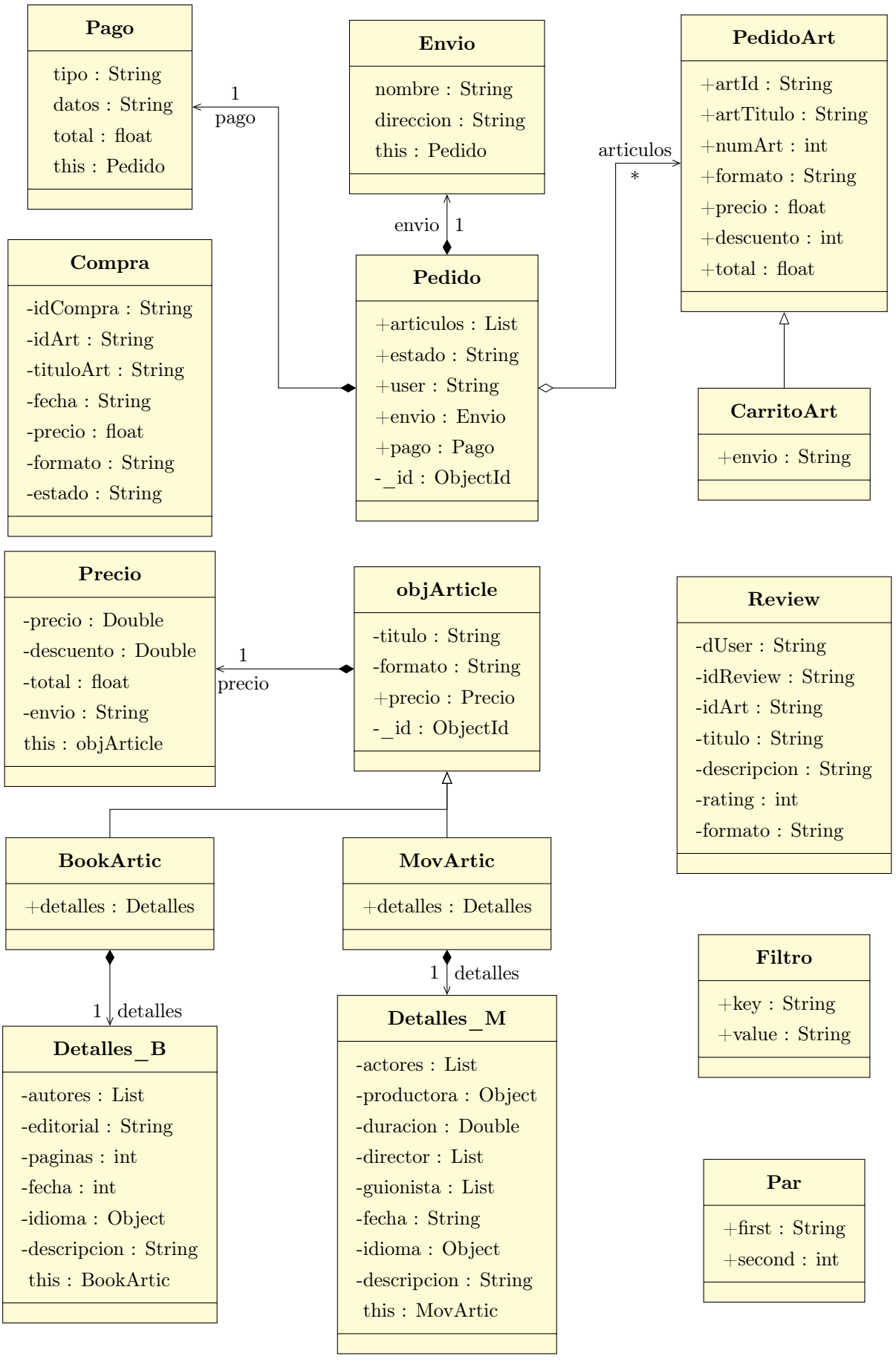

FIGURA 3.22: Modelos de Clases: Objetos

3.4.2.1 Clase filtro

Esta clase representa una condición de busqueda con la forma <"campo", "valor">. Sólo tiene dos atributos, ambos de tipo *String*:(1) el primero llamado *key* que corresponde al campo;(2) y el segundo llamado *valor* que corresponde al valor del campo. Esta clase permite implementar búsquedas de elementos genéricas, de forma que no sea necesario especificar el campo concreto sobre el que se realice la búsqueda, sino que tanto los campos y valores que forman el criterio de búsqueda se dan como parámetros del método que implemente la búsqueda.

Hay que destacar que esta clase tiene dos constructores: (1) uno que crea un objeto sin asignar valores a los atributos; y (2) otro que permite asignar los valores directamente al ponerlos como parámetros del constructor.

3.4.2.2 Clase Par

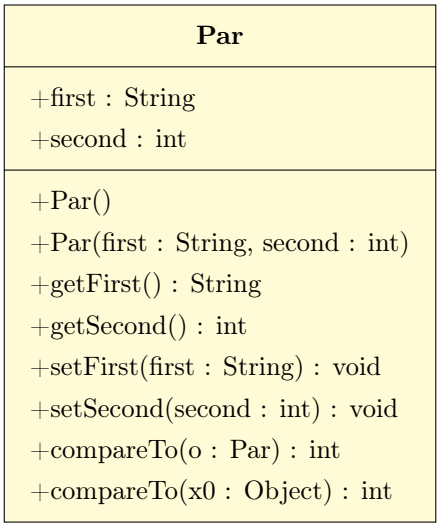

FIGURA 3.23: Clase: Par

Esta clase representa un simple objeto con dos atributos, uno *String* y otro numérico. Los métodos de esta clase en general son de tipo get y set, los cuales permiten obtener y asignar los valores de los atributos de la clase. Además, también implementa - sobrescribe - el método *compareTo()* . Este método índica sobre que atributo comparar dos objetos de tipo Par. Ésto permite que se pueda ordenar una lista de este tipo de objetos mediante el método *sort()*.

En general esta clase se utiliza para representar el par <identificador de artículo, cantidad/disponibilidad>. Sin embargo, no es específica y puede ser usada para representar otros elementos.

3.4.2.3 Clase Compra

Esta clase representa a las compras de artículos. Los atributos de esta clase se corresponden con los atributos del subgrafo que representan a las compras en el sistema de bases de datos Virtuoso - identificador de la compra, identificador del artículo, título, fecha, precio y estado.

Los métodos de esta clase son de tipo get y set, los cuales permiten obtener y asignar los valores de los atributos de la clase.

3.4.2.4 Clase Reviews

Esta clase representa a las valoraciones de los artículos. Los atributos de esta clase se corresponden con los atributos del subgrafo que representan a las valoraciones en el sistema de bases de datos Virtuoso - identificador de la valoración y del artículo, título y descripción -; y además también contiene el atributo *userId* que representa el nombre del usuario que realizó la valoración.

Los métodos de esta clase son de tipo get y set, los cuales permiten obtener y asignar los valores de los atributos de la clase.

3.4.2.5 Clase PedidoArt

Esta clase representa los datos de un artículo del que se hace un pedido. Es decir, no representa todo el pedido sino únicamente a un artículo del pedido. En MongoDB los pedidos se representaban con la información del pedido - general, precio, pago - y con una lista de subdocumentos que representan a los artículos seleccionados. Esta clase representa esos subdocumentos y los atributos coinciden con los campos de estos subdocumentos - identificador del artículo, título, cantidad seleccionada de cada artículo, formato, precio, descuento y precio total.

Los métodos de esta clase son de tipo get y set, los cuales permiten obtener y asignar los valores de los atributos de la clase.

3.4.2.6 Clase CarritoArt

Esta clase representa los datos de un artículo en una cesta. Puesto que los datos de los artículos en un pedido y en la cesta son similares, esta clase extiende la clase PedidoArt, añadiendo únicamente el atributo *envio* el cual indica qué tipo de envío tiene asignado dicho artículo.

Los métodos de esta clase son de tipo get y set, los cuales permiten obtener y asignar los valores de los atributos de la clase.

3.4.2.7 Clase Pedido

Esta clase representa los pedidos realizados por los usuarios. La estructura es similar a la del esquema de los pedidos guardados en MongoDB. Por un lado, contiene los atributos simples *estado*, *user*, *_id* correspondientes al estado en el que se encuentra el pedido, el usuario que ha realizado el pedido y el identificador del pedido.

Por otro lado, también contiene los atributos *envío* y *pago* que son instancias de las clases anidadas *Envio* y *Pago*. Tener los datos de los envíos y los pagos en clase anidadas permite tener una estructura similar a la de los documentos de los pedidos en MongoDB donde los campos *envío* y *pago* contienen subdocumentos anidados dentro del documento principal.

Finalmente, contiene el atributo *artículos* que es una lista de objetos de la clase PedidoArt y que por tanto representa una lista de artículos. De nuevo, esto es similar a la estructura de los pedidos en MongoDB donde el campo *artículos* contiene una lista de subdocumentos.

Existen varias ventajas al usar una estructura similar al de los documentos. Por un lado se mantiene un cierto nivel de homogeneidad entre los objetos de la aplicación y de la base de datos. Y por otro lado, valiéndose de librerías como GSON es posible pasar la estructura de los datos devueltos por la base de datos a la de los objetos de la aplicación de forma casi automática.

Los métodos de esta clase en general son de tipo get y set, los cuales permiten obtener y asignar los valores de los atributos de la clase.

3.4.2.7.1 Clase anidada Pago

Esta clase representa los datos de pago del pedido. Los atributos se corresponden con los campos del subdocumento guardado en el campo *pago* en los documentos que guardan los datos de los pedidos.

Los método de esta clase son de tipo get y set, los cuales permiten obtener y asignar los valores de los atributos de la clase.

3.4.2.7.2 Clase anidada Envío

Esta clase representa los datos de envío del pedido. Los atributos se corresponden con los campos del subdocumento guardado en el campo *envio* en los documentos que guardan los datos de los pedidos.

Los métodos de esta clase son de tipo get y set, los cuales permiten obtener y asignar los valores de los atributos de la clase.

3.4.2.8 Clase objArticle

Esta clase es una clase abstracta que representa a los artículos en general independientemente del tipo de artículo que se trate. Esta clase sólo contiene los atributos comunes para los dos tipos de artículos disponibles. Estos son: el título, el identificador del artículo, el formato y el precio. En el caso del precio este atributo es de la clase *Precio* que es una clase anidada la cual contiene los datos del precio del artículo. De forma similar en los documentos que contienen los datos de los artículos, el campo *precio* contiene un subdocumento - un documento anidado - con la información del precio.

Los métodos de esta clase son de tipo get y set, y éstos permiten obtener y asignar los valores de los atributos de la clase.

3.4.2.8.1 Clase anidada Precio

Esta clase anidada representa el precio de un artículo mediante los atributos que representan el precio, el descuento, el total y el tipo de envío. En este caso, a diferencia de lo que ocurre en los documentos, se incluye el dato referente al precio total calculado a partir del descuento.

Los métodos de esta clase son de tipo get y set, los cuales permiten obtener y asignar los valores de los atributos de la clase.

3.4.2.9 Clase MovArtic

Esta clase representa a los datos de los artículos de tipo película y extiende la clase abstracta *objArticle*. Contiene el atributo *detalles* que es de la clase anidada *Detalles*. Ésta clase representa los detalles específicos para los artículos de este tipo.

3.4.2.9.1 Clase anidada Detalles

Esta clase anidada representa los detalles específicos para los artículos de tipo película. Los atributos incluyen una lista con los actores, el nombre de la productora, una lista con los directores, una lista con los guionistas, la duración de la película, el idioma de la película, la fecha de creación y una descripción de la película.

Los métodos de esta clase son de tipo get y set, los cuales permiten obtener y asignar los valores de los atributos de la clase.

3.4.2.10 Clase BookArtic

Esta clase representa a los datos de los artículos de tipo libro y extiende la clase abstracta *objArticle*. Contiene el atributo *detalles* que es de la clase anidada *Detalles*. Ésta clase anidada representa los detalles específicos para los artículos de este tipo.

3.4.2.10.1 Clase anidada Detalles

Esta clase anidada representa los detalles específicos para los artículos de tipo libro. Los atributos incluyen una lista con los autores, el nombre de la editorial, el número de páginas, el idioma del libro , la fecha de creación y una descripción del libro.

Los métodos de esta clase son de tipo get y set, los cuales permiten obtener y asignar los valores de los atributos de la clase.

3.4.3 Clases de la aplicación Web

El diagrama de la figura 3.24 se puede ver el diagrama de clase correspondiente a la parte de presentación y de lógica de negocio de la aplicación. Se a simplificado la representación de la capa de datos - servicios - puesto que ya se ha hablado de ella.

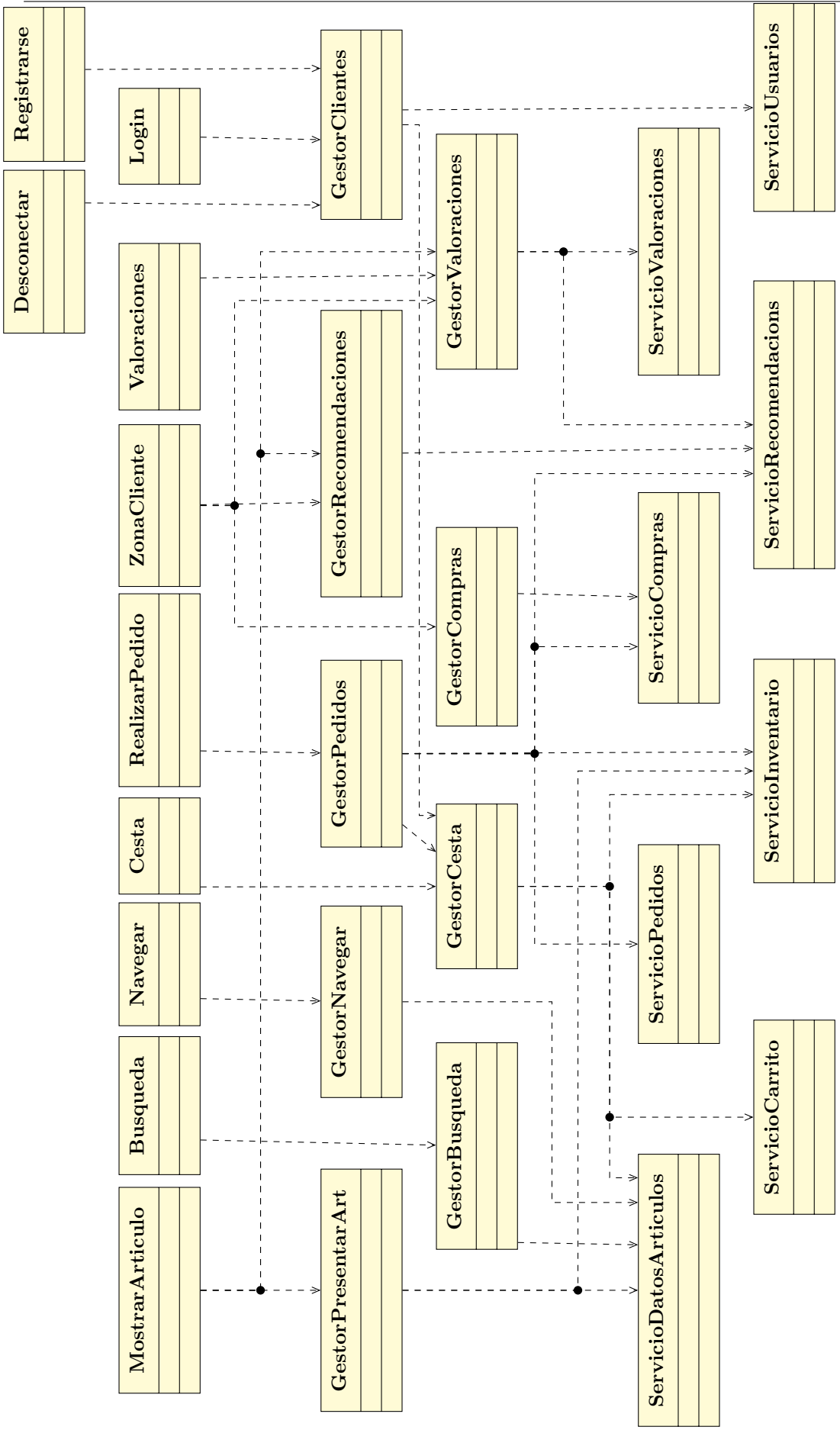

FIGURA 3.24: Diagrama de Clases:Aplicación Central

3.4.3.1 Clases de presentación

En general todas estas clases tiene una estructura cuya característica principal es que se encargan de implementar los métodos *doGet* y *doPost* para tratar las peticiones http.

En concreto las clases de presentación son las siguientes:

- Navegar: esta clase se encarga de responder a las peticiones para buscar artículos en función del tipo y del género.
- Búsqueda: esta clase se encarga de responder a las peticiones para buscar artículos en función de un valor dado, el cual puede ser el título o el nombre de un autor/actor/director/guionista.
- Login: esta clase se encarga de responder a las peticiones de identificación por parte de los usuarios. Para esto, una vez comprobadas las credenciales guardará la información de usuario en la sesión asociada al usuario.
- Desconectar: esta clase se encarga de responder a las peticiones de desconexión. En este caso se limitará a borrar las credenciales de la información de sesión.
- Registro: esta clase se encarga de responder a la petición de registro.
- Mostrar Articulo: esta clase responderá a la petición de mostrar un pedido. Entre la información que mostrará está: la información general del artículo, las recomendaciones en función de dicho artículo, las valoraciones de los usuarios para ese artículo y la disponibilidad.
- Cesta: esta clase se encarga de responder a las peticiones de visualización de la cesta de la compra, así como a las peticiones de modificación y eliminación de artículos de la cesta. La cesta que se muestra y se modifica será identificada mediante la información de sesión guardada.
- Realizar Pedido: esta clase se encarga de responder a las peticiones para llevar a cabo un pedido, se encarga tanto de mostrar el formulario para obtener los datos del pedido como de mostrar el pedido modificado en caso de falta de disponibilidad de algún artículo.
- Zona Cliente: esta clase se encarga de tratar varias operaciones que están disponibles únicamente para los clientes. Entre ellas se incluyen, ver los artículos comprados, ver los pedidos realizados y cancelarlos si todavía no han sido enviados, modificar la contraseña, ver las valoraciones realizadas con la opción de eliminarlas, y además como página de inicio se muestran recomendaciones en función de las compras y las valoraciones hechas por los usuarios.

3.4.3.2 Clase GestorBusqueda

Esta clase se encarga de gestionar las búsquedas personalizadas de artículos. Solicita al servicio *ServicioDatosArticulos* artículos en función de un valor que puede coincidir con parte de los valores de alguno de los datos de un artículo.

Como atributos tiene por un lado una instancia al servicio de los artículos, y por otro atributos relacionados con el criterio de búsqueda.

En esta clase el método principal es *buscarArt()*, este genera la lista de filtros a partir de los atributos *tipo* y *formato*, y solicita al servicio una búsqueda de texto con el atributo *elem* como criterio de búsqueda, la lista de filtros como condiciones de filtrado de los resultados y los atributos *sort* y *numPag* como criterio de ordenación y de salto.

Los métodos *next()* y prev() modifican el atributo *numPag* y llaman al método anterior para solicitar los diez siguientes/anteriores elementos.

3.4.3.3 Clase GestorNavegarArt

Este gestor es similar al anterior salvo que gestiona búsquedas predeterminadas sobre el dato *género* de los artículos.

En este caso, en lugar del atributo *elem*, se tiene el atributo *genero*.

El método *obtenerArt()* es similar al método *buscarArt()*, salvo que en lugar de solicitar una búsqueda de texto, se solicita una búsqueda sobre campos determinados de los artículos.

El método *setSort()* permite que se modifique el criterio de ordenación.

3.4.3.4 Clase GestorPresentacionArticulo

Esta clase se encarga de solicitar a varios gestores todos los datos necesarios para presentar los datos de un artículo.

Como atributos tiene las instancias del servicio *ServicioDatosArticulos* y del servicio de inventario.

Los métodos simplemente realizan solicitudes - obtener artículo por id y obtener disponibilidad - a los servicios y devuelven directamente los resultados obtenidos de la solicitud.

3.4.3.5 Clase GestorCesta

En general esta clase se encarga de gestionar las operaciones que el usuario quiere hacer sobre la cesta.

Únicamente tiene como atributos las instancias a los servicios del carrito, de los datos de los artículos y del inventario.

Los métodos que contiene esta clase se corresponden con las operaciones para ver/modificar/eliminar la cesta o para añadir un nuevo artículo. Los métodos para ver y eliminar hacen una solicitud directa al servicio de carrito y devuelven los resultados. El método para insertar un artículo, sin embargo, requiere que se soliciten los datos del artículo al servicio de datos de artículos; y el método para que se modifique la cantidad seleccionada de un artículo requiere que se compruebe la disponibilidad antes de modificar la cantidad en la cesta.

Por último, destaca el método para modificar al usuario, en este caso comprueba si el nuevo usuario tiene una cesta guardada y si la tiene solicita que se unifique la cesta actual con la guardada. Sino simplemente solicita el cambio del nombre de usuario.

3.4.3.6 Clase GestorClientes

Esta clase gestiona las operaciones sobre los datos de los clientes. En concreto, se encarga de comprobar las credenciales de los usuarios al iniciar sesión y de registrar a nuevos usuarios.

Para el primer caso se usa el método *loginClient*. Se solicitará al servicio de usuarios que compruebe si son correctas las credenciales. Si lo son solicita al gestor de cestas que modifique al dueño de la cesta.

Para el segundo caso se usa el método *Registrar*. Se solicita al servicio de usuarios que compruebe si el usuario ya está registrado, si no lo está solicita que se añada un nuevo usuario y de nuevo solicita al gestor de cestas que modifique al dueño de la cesta.

3.4.3.7 Clase GestorRecomendaciones

Esta clase se encarga de solicitar las recomendaciones que se quieren mostrar a los usuarios.

Hay dos tipos de métodos: los que solicitan las recomendaciones en función del usuario y los que solicitan las recomendaciones en función del artículo que se está viendo en ese

momento. Aún así todos los métodos realizan unos pasos similares: (1) solicitan las recomendaciones al servicio de recomendaciones; y (2) solicitan los datos del artículo recomendado al servicio de datos de artículos.

Tiene como atributos las instancias del servicio de recomendaciones y del servicio de datos de artículos.

3.4.3.8 Clase GestorValoraciones

Se encarga de gestionar las diferentes operaciones que un usuario puede hacer sobre las valoraciones. En concreto esta clase permite ver las valoraciones de un artículo, ver las valoraciones hechas por un usuario, añadir una valoración , indicar que un artículo le gusta y eliminar una valoración.

3.4.3.9 Clase GestorPedidos

Esta clase permite ver y modificar el estado de los pedidos. Sin embargo, su tarea principal es la de procesar el pedido de un usuario.

La clase contiene el método *realizarPedido* que se encarga de realizar los pasos necesarios para realizar el pedido. Para ello realiza los siguientes pasos: (1) solicita los artículos en la cesta del usuario al gestor de cestas; (2) intenta modificar el inventario mediante el servicio de inventario; (3) comprueba si se ha modificado el inventario correctamente.

Llegado a este punto con éxito se procesa el pedido lo cual incluye insertar el pedido, añadir la compra, actualizar el sistema de recomendaciones y finalmente eliminar la cesta. Si no se llega con exito, se eliminan de la cesta los artículos no disponibles.

Hay que destacar, que en este caso se hace tanto al añadir compra como al actualizar el sistema de recomendaciones. Esto se debe a que aunque están en el mismo sistema de bases de datos, la aplicación no lo sabe y para ella son tareas independientes sobre datos independientes. Por tanto, este tipo de problemas que se dan al usar un mismo sistema para varios tipos de datos se deberían tener en cuenta al diseñar los servicios. Por ejemplo, en este caso para solucionar el problema simplemente se comprueba que las compras no están ya guardadas - comprueba que el id de la compra no existe.

Capítulo 4

Aplicación de Persistencia Políglota: Implementación

A lo largo de este capítulo se va a tratar todo lo relacionado con la implementación de la aplicación. Por un lado, se va a explicar el método de desarrollo y de pruebas que se ha seguido para desarrollar la aplicación. Por otro lado, en un nivel más técnico se va a explicar las estructura de los diferentes ficheros desarrollados para implementar la aplicación y el flujo de ejecución que sigue la aplicación para llevar a cabo cada uno de los casos de uso; por último también se tratarán las diferentes tecnologías que se han usado para desarrollar la aplicación, sin entrar en los sistemas NoSQL ya comentados anteriormente.

4.1 Desarrollo de la aplicación

Debido a la naturaleza de nuestra aplicación, la cual usa varios sistemas de almacenamiento y a que se ha diseñado la aplicación siguiendo la arquitectura en tres capas, se puede decir que se ha seguido un modelo de desarrollo *bottom-up* para las primeras fases. La idea principal de este modelo es la de desarrollar partes individuales básicas y después ir uniendo los distintos componente para crear otros más complejos, y así hasta construir la aplicación.

Siguiendo este modelo, se realizaron las siguientes fases para desarrollar la aplicación:

- Fase I: en esta fase por cada sistema se desarrollaron de forma conjunta los servicios que hacen uso de un mismo servicio. Es decir, se desarrollaron por un lado los servicios que se conectan a MongoDB - artículos, pedidos y carrito -, por otro los

que se conectan a Virtuoso - usuarios, compras, recomendaciones y valoraciones - y finalmente los servicios que hacen uso de MySQL - inventario. Y además se implementaron las clases de los objetos.

- Fase II: en esta fase se unieron todos los servicios en un único proyecto.
- Fase III: en esta fase se desarrollaron los gestores y una versión básica de la parte de presentación. En este caso se desarrollaron los casos de uso uno a uno, empezando por los más básicos. Por otro lado, durante esta fase se unió parte de la aplicación con los servicios, ya que los gestores hacen uso de estos servicios. Cabe destacar que siguiendo la idea de reutilización y modularidad, en lugar de añadir los servicios como un nuevo paquete, se han añadieron como una librería *.jar*.
- Fase IV: en esta fase se mejoró la parte de presentación, principalmente la apariencia de la aplicación.

4.2 Estructura de la aplicación

Como ya se ha dicho la aplicación sigue una arquitectura de tres capas - Datos, Lógica de Negocio y Presentación. Para mantener esta estructura, la aplicación principal se ha dividido de la siguiente manera:

- Paquete *Gestores*: contiene las clases que implementan la capa de la Lógica de Negocio.
- Paquete *Servelets*: contiene las clases que junto con las páginas web implementan la capa de Presentación. En concreto estas clase se encargan de recibir y responder a las solicitados *http* realizadas por el usuario al interactuar con la página web.
- Pagina Web: esta carpeta contiene la página web - jsp, css y javascript - que con las clases del paquete Servelets implementan la capa de Presentación. En concreto está página presenta una interfaz web que permite al usuario interactuar con la aplicación.
- Librería de Servicios: esta librería implementa la capa de Datos. Permite acceder a los distintos sistemas de almacenamiento de forma transparente, es decir la aplicación no necesita saber en qué sistema está cada dato. Además también implementa una serie de objetos que representa a los datos guardados en los sistemas de almacenamiento.

En lo que se refiere a la estructura de la librería de los servicios, ésta se divide entre los objetos, las conexiones y los servicios de la siguiente manera:

- Paquete *conexiones*: este paquete contiene las clases que implementan las conexiones a los tres sistemas de almacenamiento. Se ha creado un paquete con las conexiones a parte porque hay varios servicios que se conectan al mismo sistema de almacenamiento, y de esta forma no es necesario implementar la conexión varias veces.
- Paquete *services*: este paquete contiene las clases que implementan los servicios que realizarán consultas sobre los sistemas de almacenamiento. Hay que destacar que cada servicio sólo usará un sistema de almacenamiento asociado con el tipo de datos sobre los que realiza las consultas. Si se quisiera tener un servicio que utilizase datos de varios sistemas la mejor opción sería crear un servicio que llamase a otros servicios más básicos, pero que no realizase consultas sobre los sistemas de almacenamiento.
- Paquete *objetos*: este paquete contiene las clases que implementan los objetos que se usarán para guardar los datos de los distintos elementos almacenados en los sistemas de bases de datos.

En la figura 4.1 se muestra la estructura especificada anteriormente.

Finalmente, una especificación más concreta de los contenidos de cada paquete y de sus métodos se pueden encontrar en los JavaDoc generados - Servicios [27] y Aplicación Principal [28].

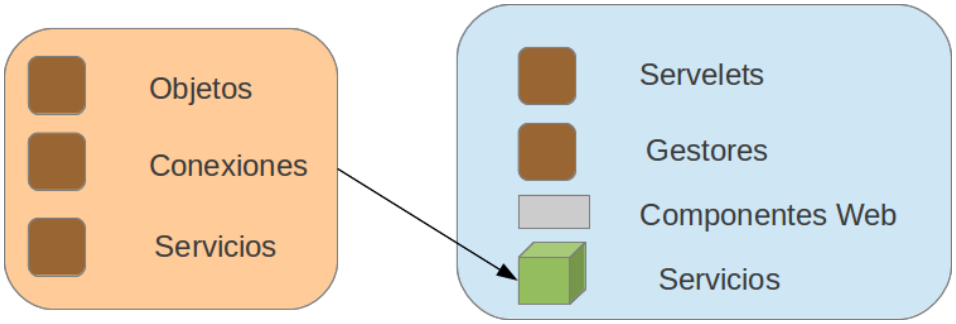

FIGURA 4.1: Estructura de la aplicación

4.3 Diagramas de secuencia

En esta sección se incluyen los diagramas de secuencia de los casos de uso vistos en el punto 3.2. Para simplificar se ha sustituido la palabra gestor/es por g, la palabra servicio/s por s y la palabra Servelets por sv.

En la figura 4.2 se muestra el diagrama de secuencia para el caso de uso Navega. Para simplificar se ha puesto como parámetro de *buscarArticulos(opciones)*, en este caso opciones se refieren a los parámetros *(String elemento de búsqueda, List<Filtro> filtros, entero numero de pagina,String campo de ordenación)* - en este caso el elemento de búsqueda será *null*.

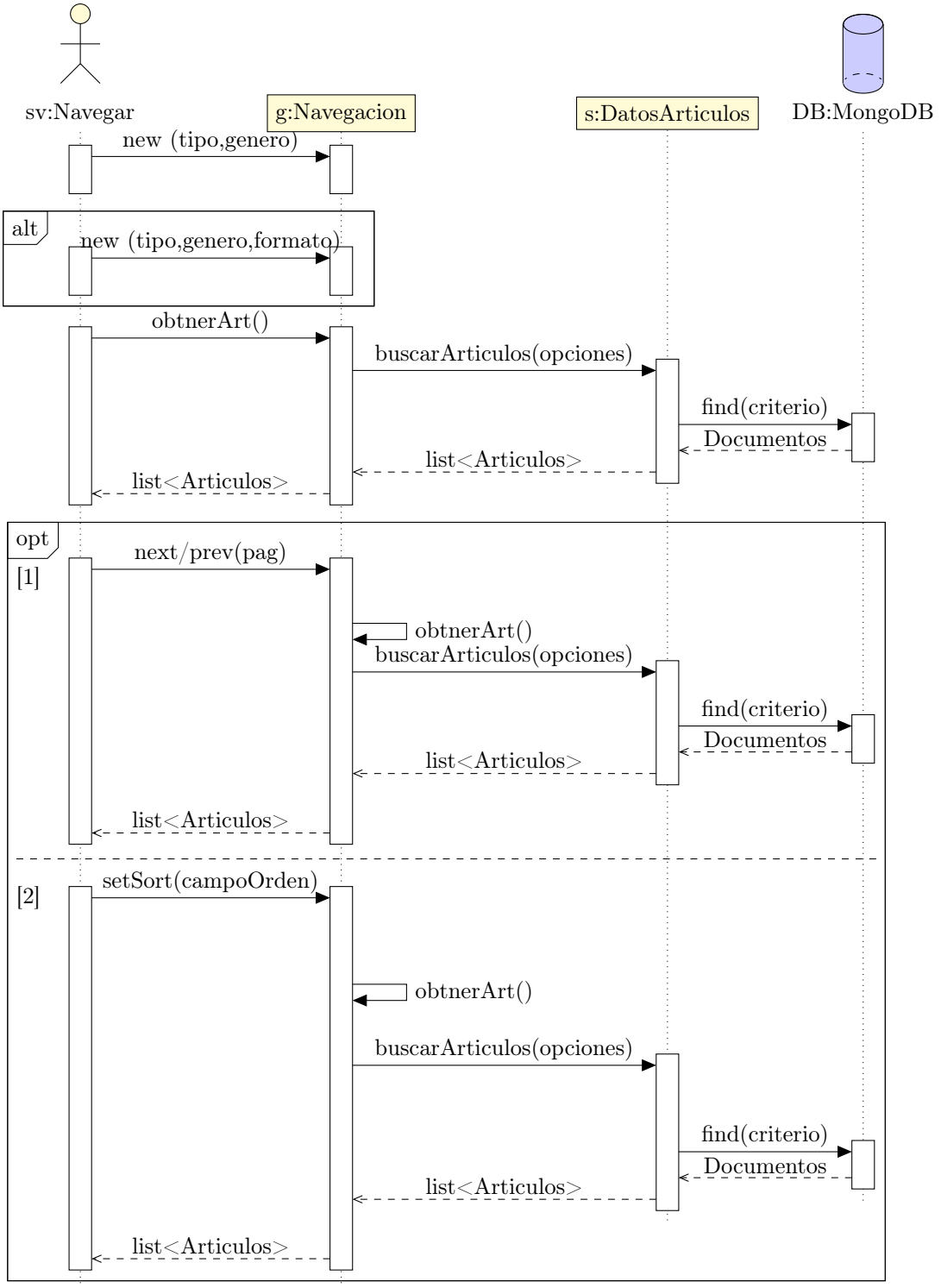

FIGURA 4.2: Diagrama de secuencia del caso de uso Navegar

El caso de uso *Búscar Artículo* es similar al anterior.

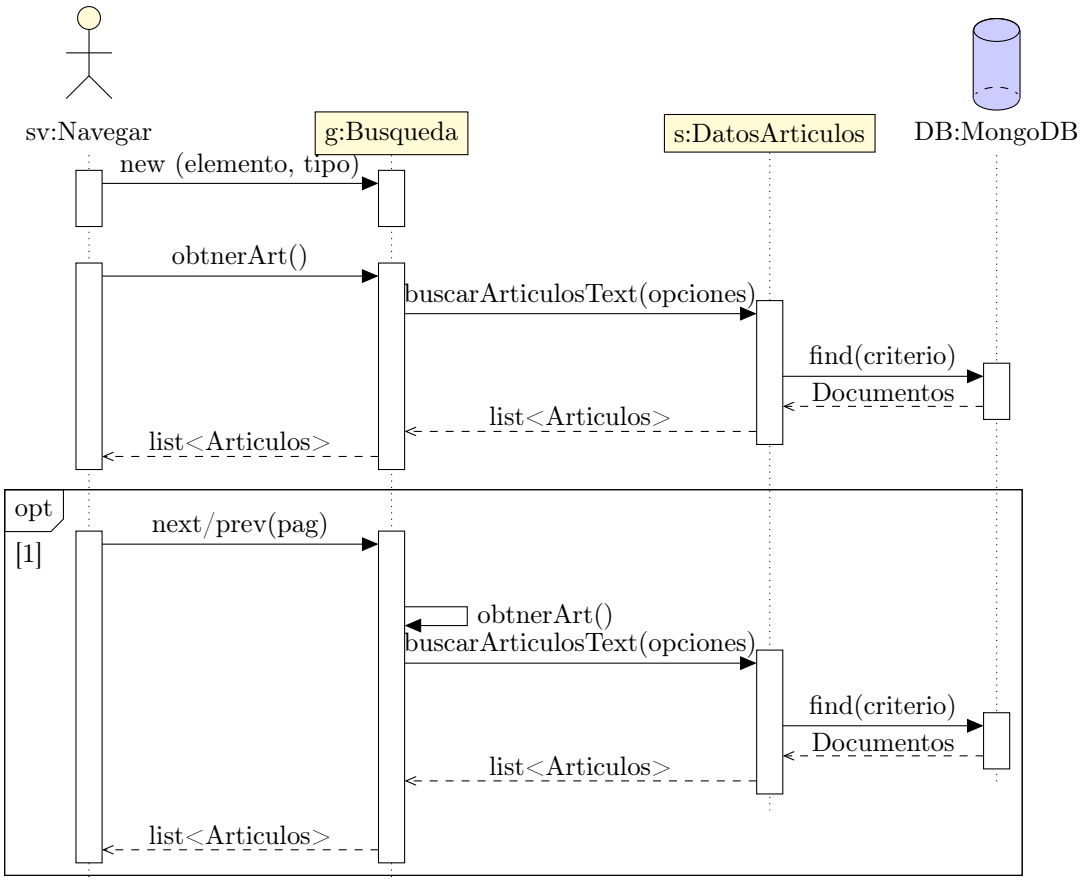

FIGURA 4.3: Diagrama de secuencia del caso de uso Buscar Artículo

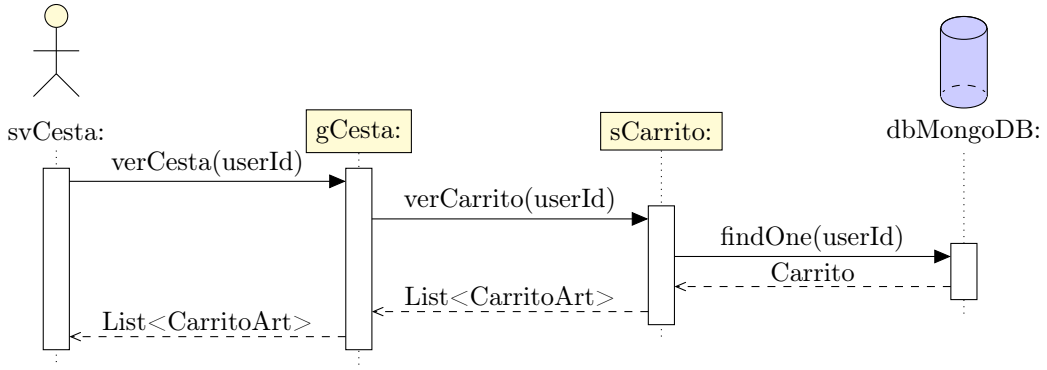

FIGURA 4.4: Diagrama de los casos de uso Ver Cesta

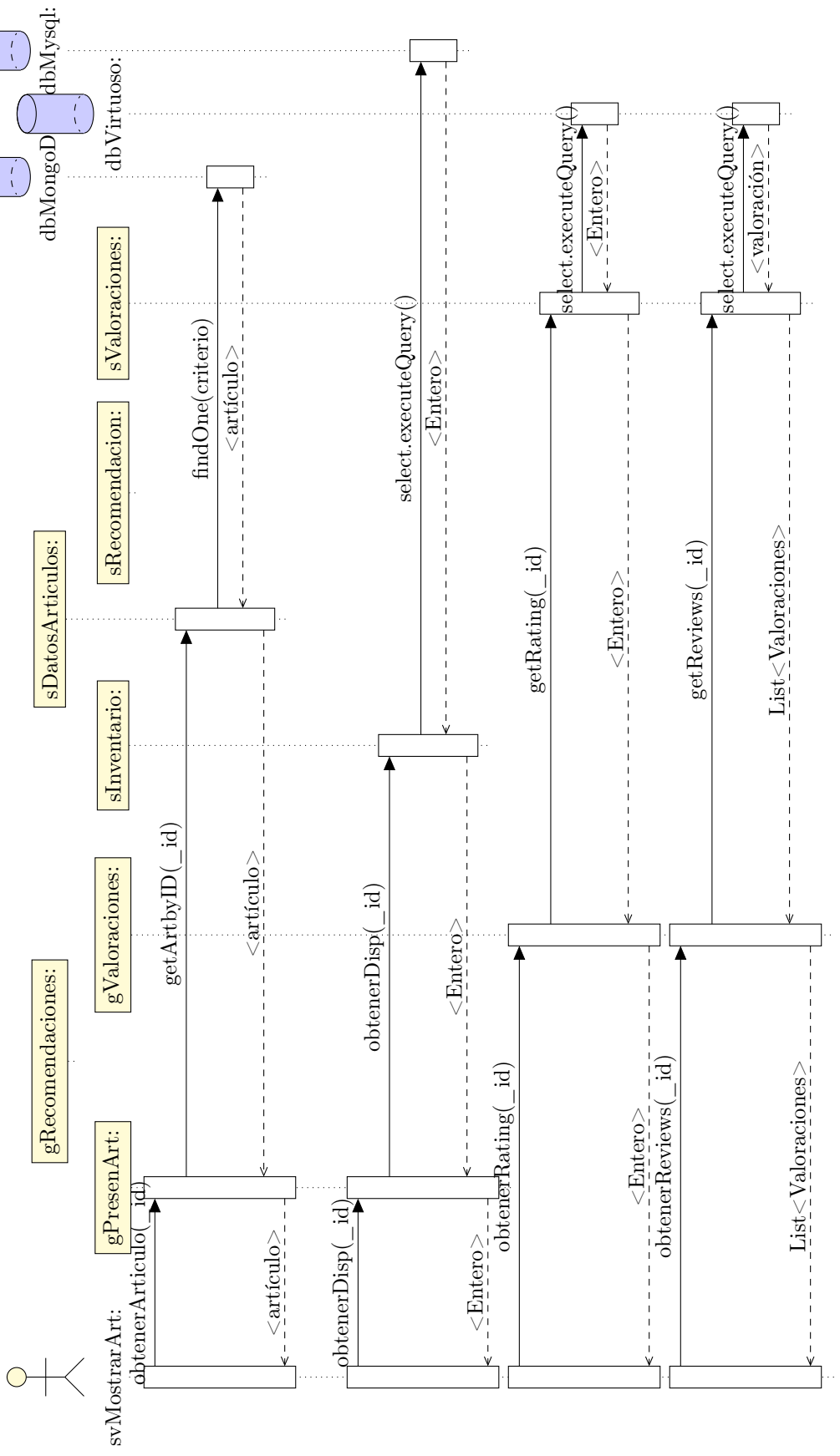

FIGURA 4.5: Diagrama de secuencia del caso de uso Ver Artículo (I)

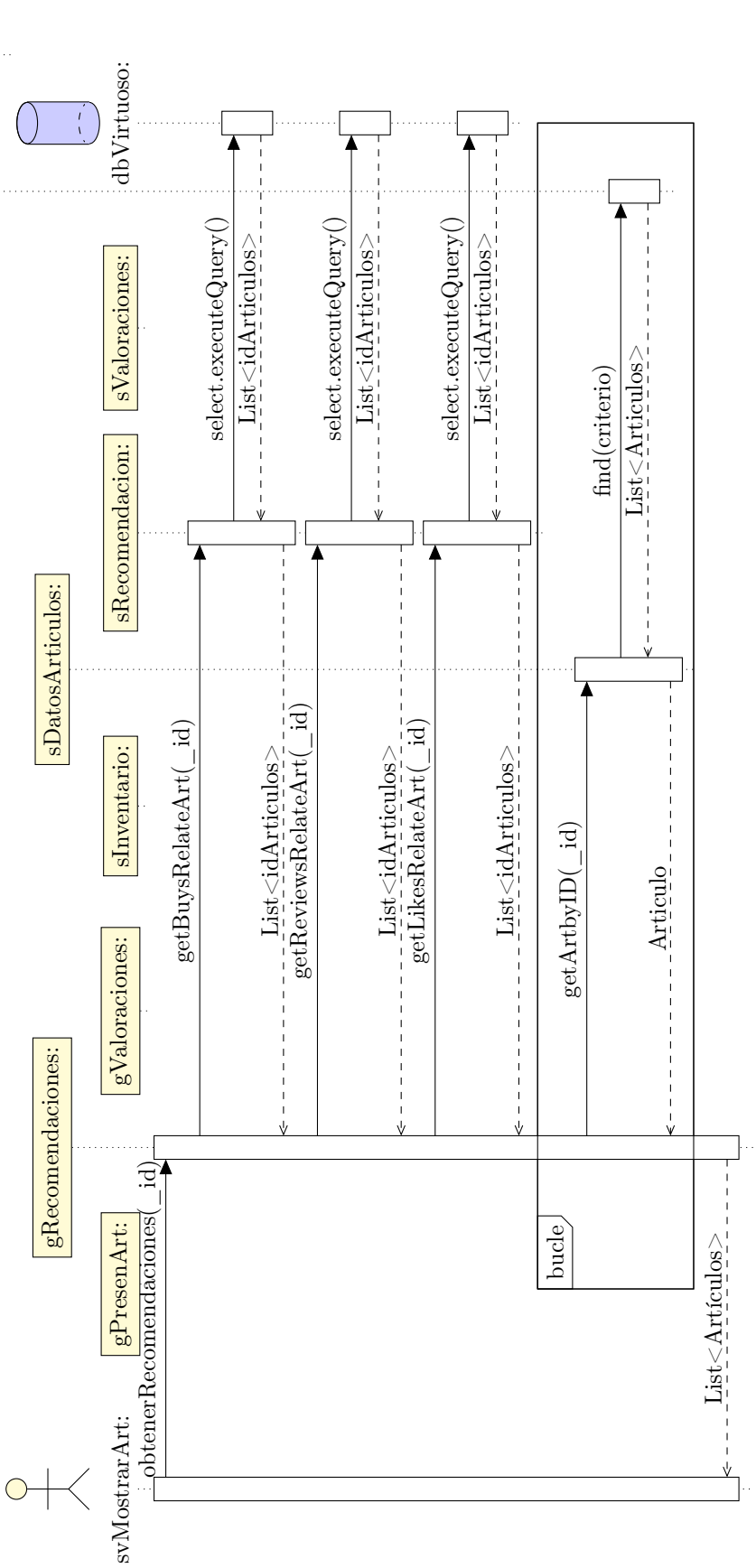

FIGURA 4.6: Diagrama de secuencia del caso de uso Ver Artículo (II)

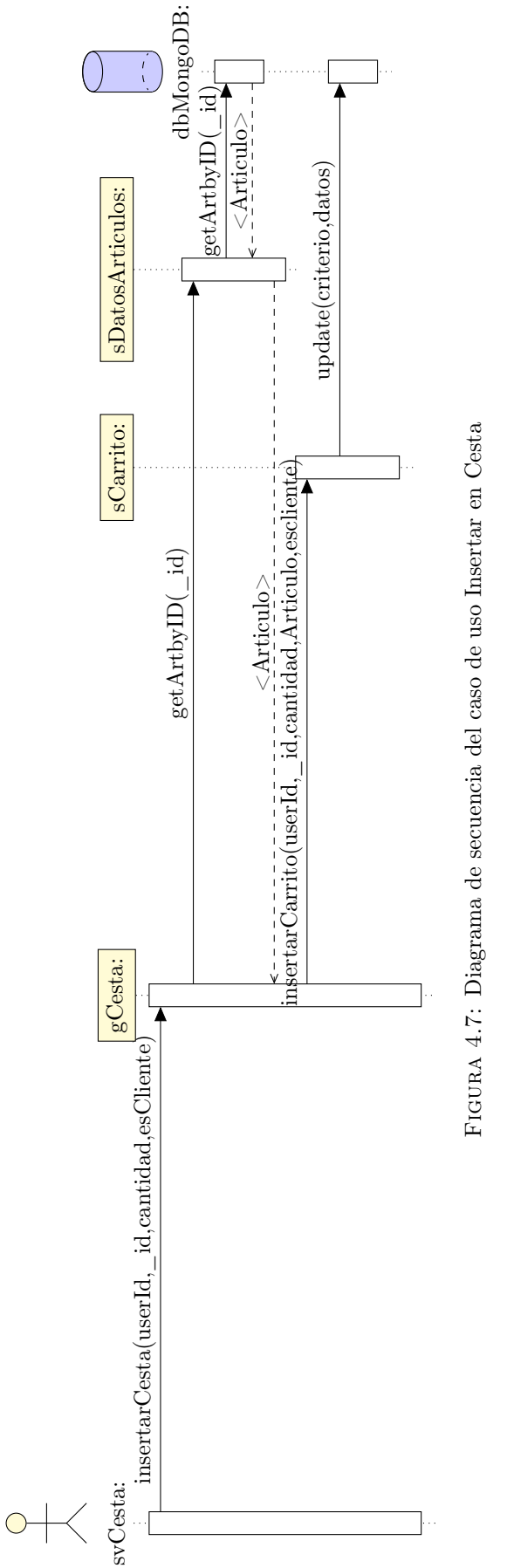

FIGURA 4.7: Diagrama de secuencia del caso de uso Insertar en Cesta

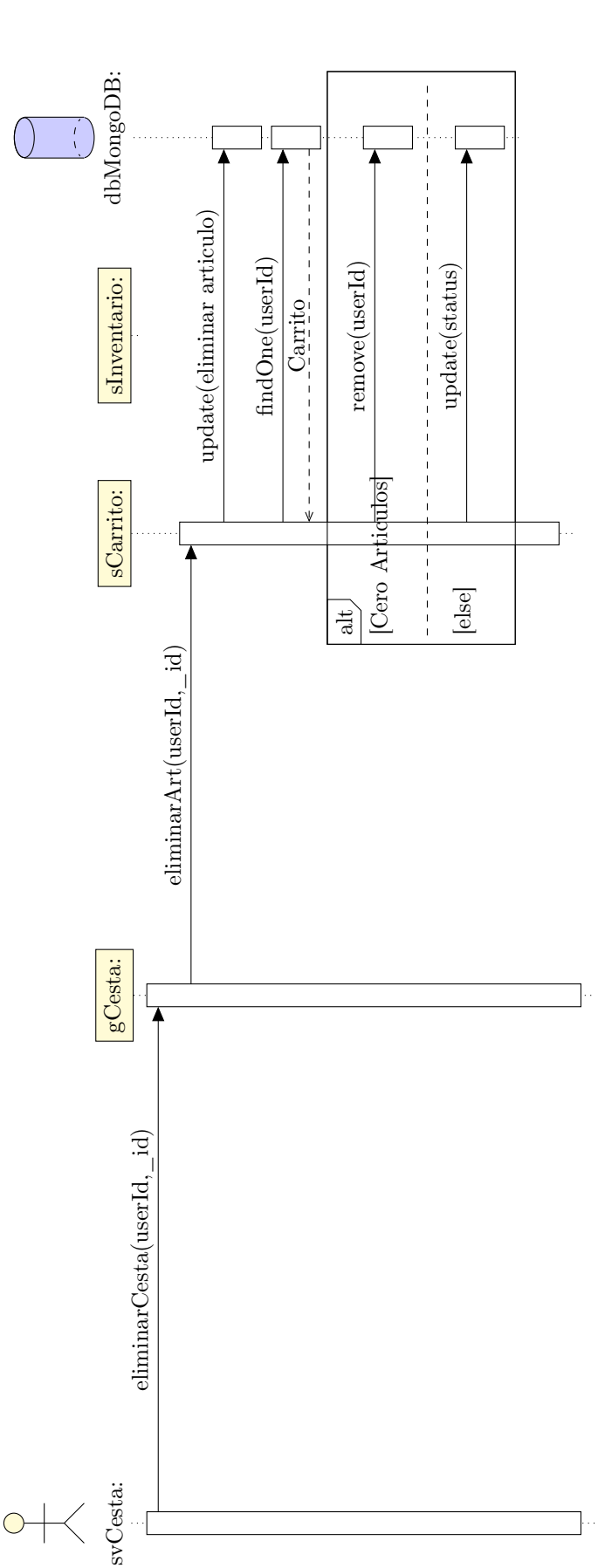

FIGURA 4.8: Diagrama de secuencia del caso de uso modificar Cesta: eliminar artículo

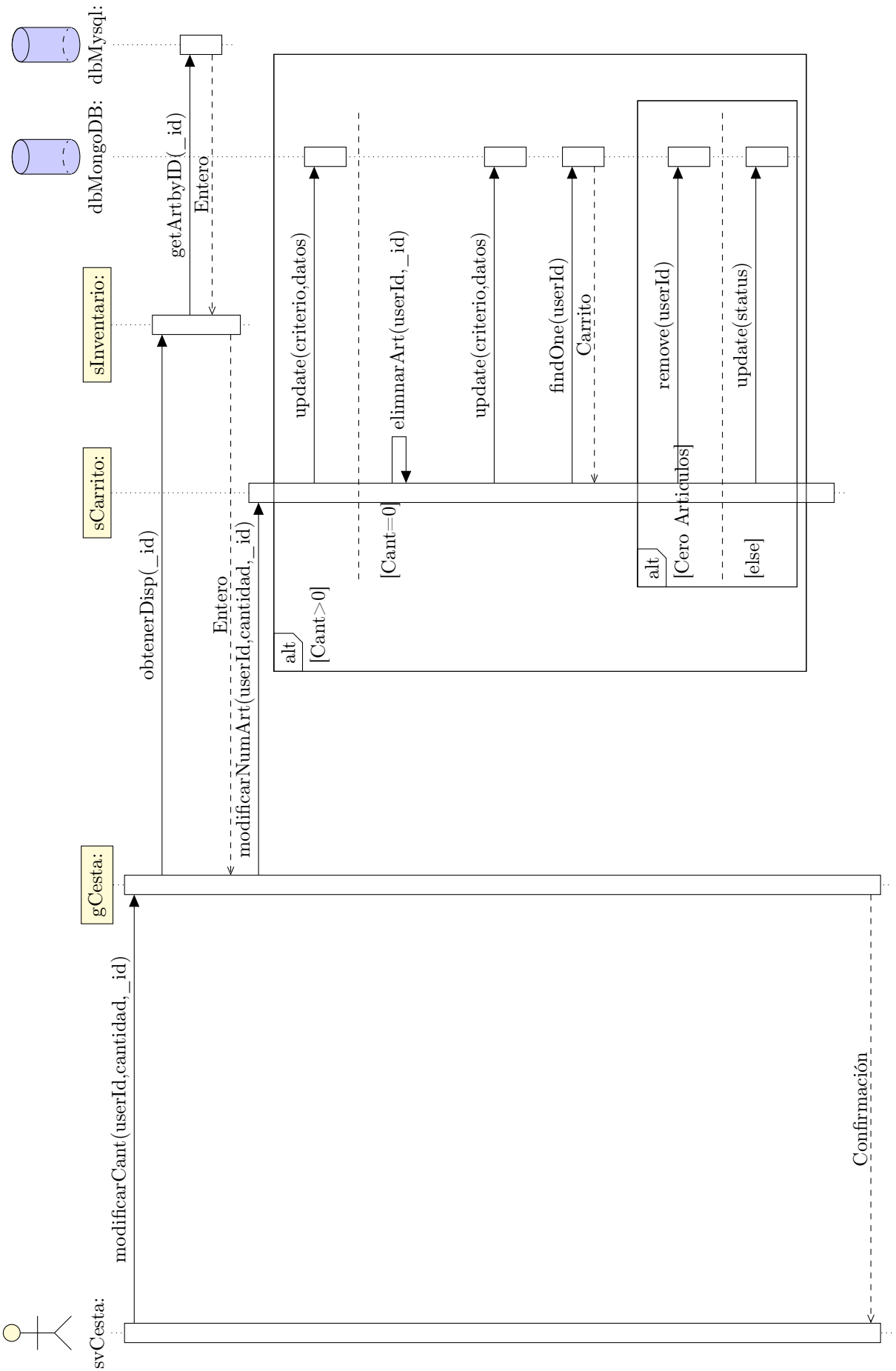

FIGURA 4.9: Diagrama de secuencia del caso de uso modificar Cesta (II): modificar cantidad de un artículo

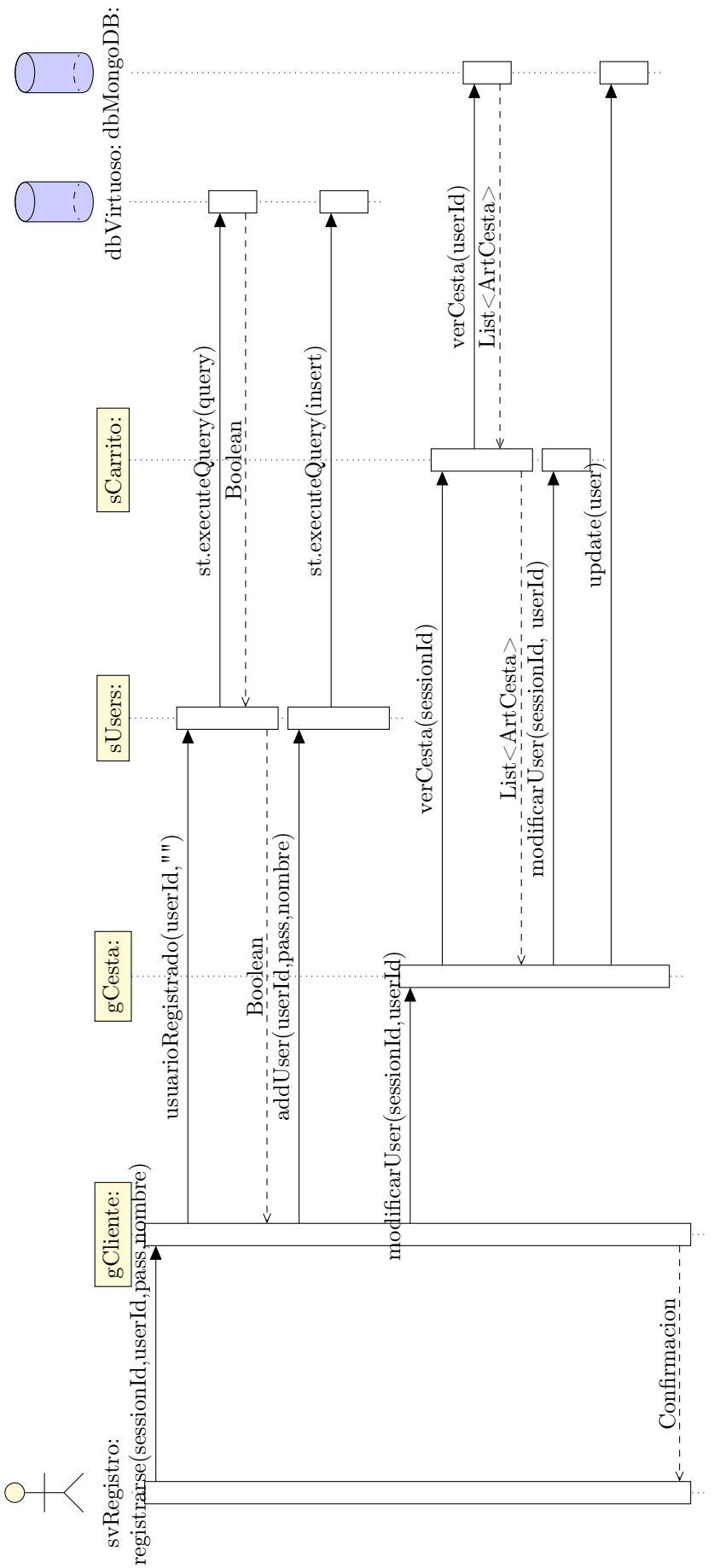

FIGURA 4.10: Diagrama de secuencia del caso de uso Registrarse

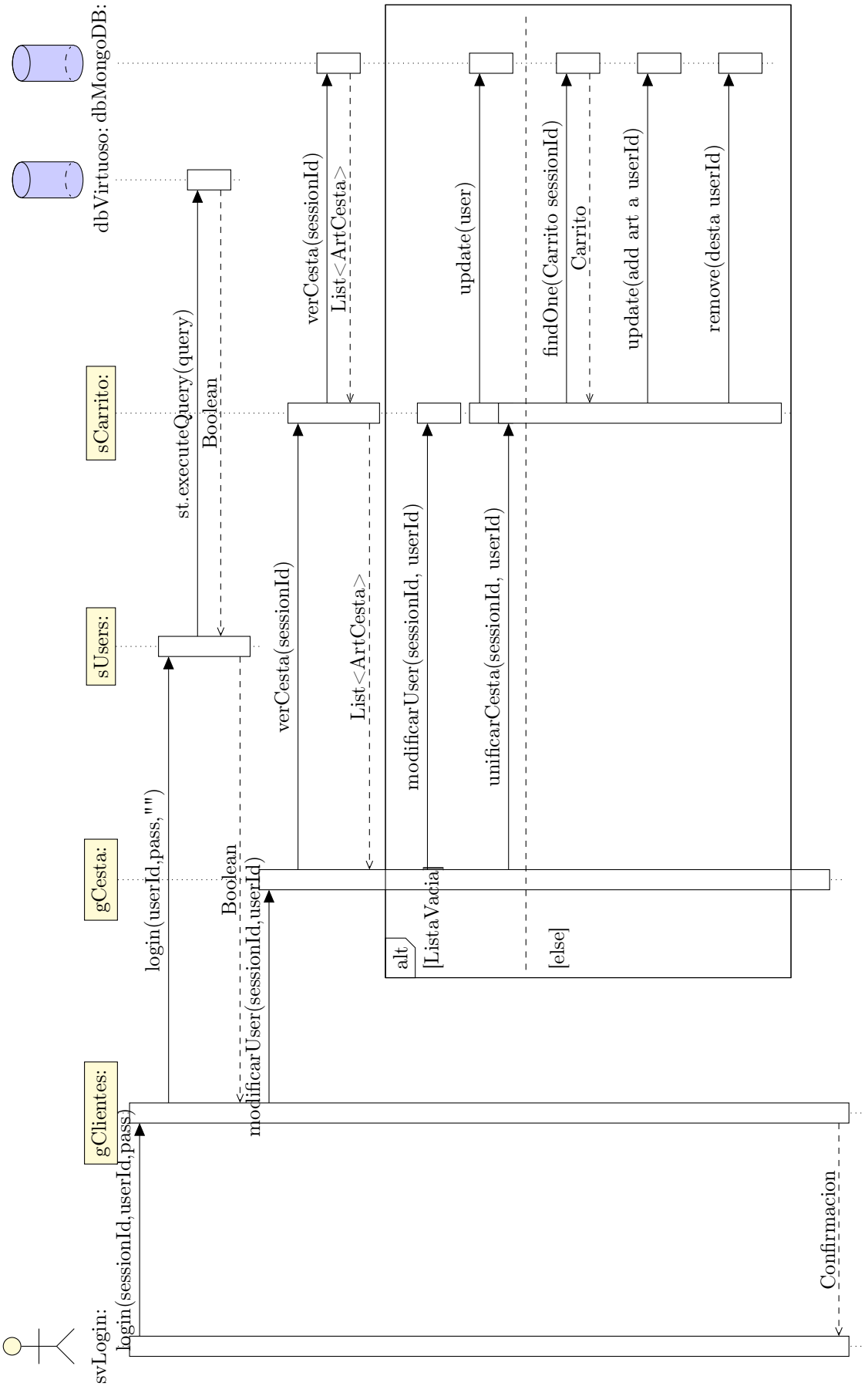

FIGURA 4.11: Diagrama de secuencia del caso de uso Login

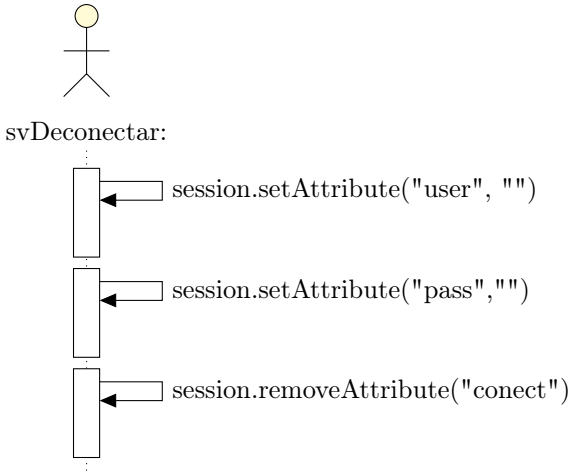

FIGURA 4.12: Diagrama de secuencia del caso de uso Desconectarse

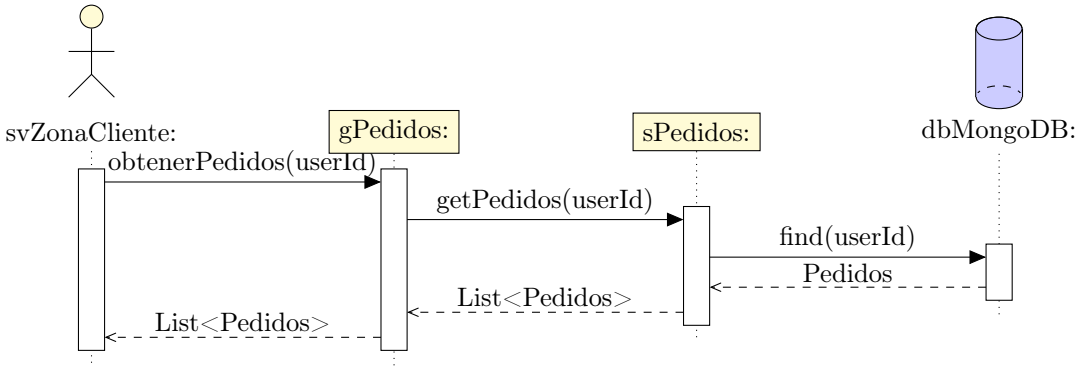

FIGURA 4.13: Diagrama de secuencia del caso de uso Ver Pedidos del usuario

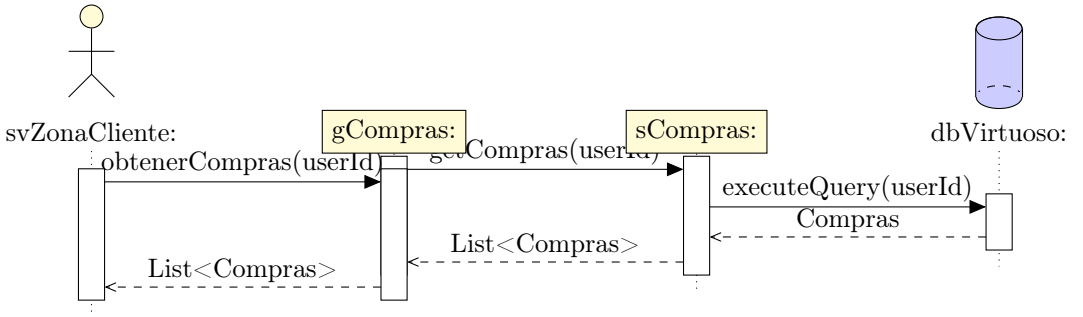

FIGURA 4.14: Diagrama de secuencia del caso Ver Compras del cliente

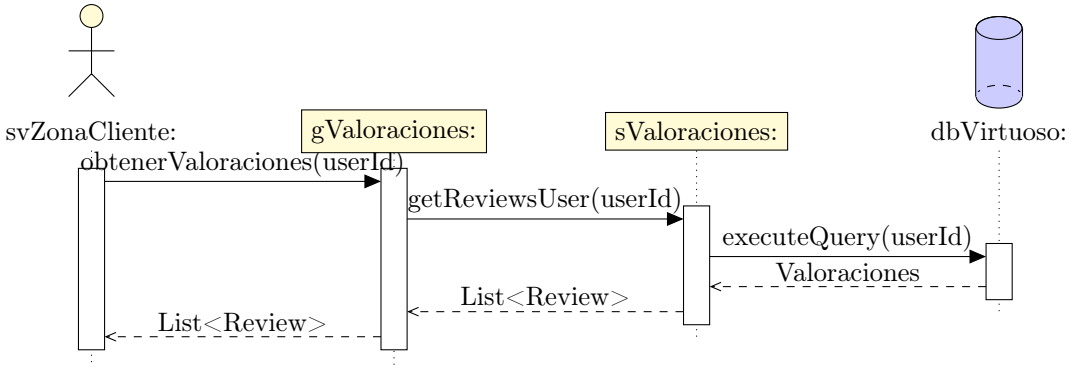

FIGURA 4.15: Diagrama de secuencia del caso Ver Valoraciones de los Usuarios

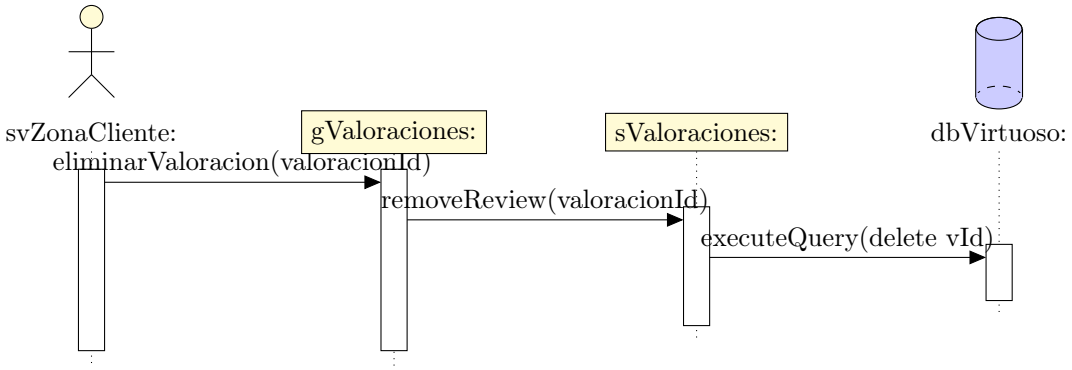

FIGURA 4.16: Diagrama de secuencia del caso de uso Eliminar Valoración

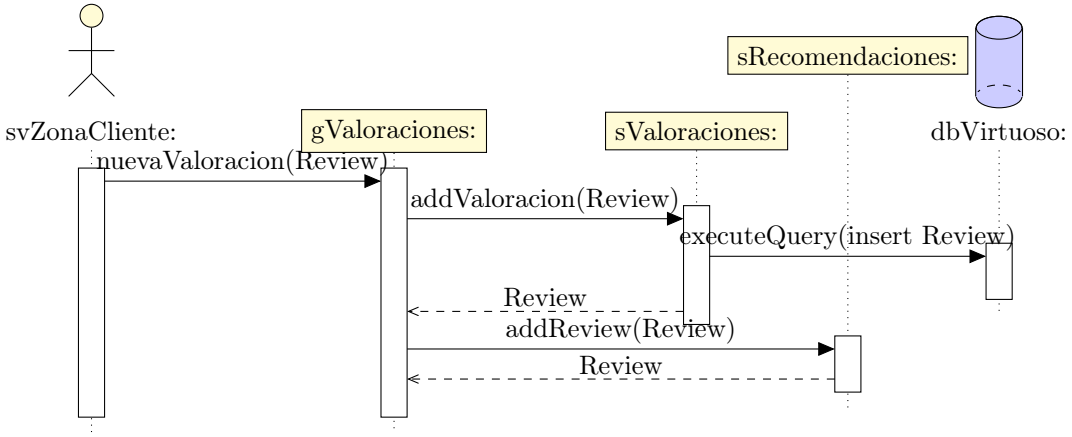

FIGURA 4.17: Diagrama de secuencia del caso de uso Añadir Valoración

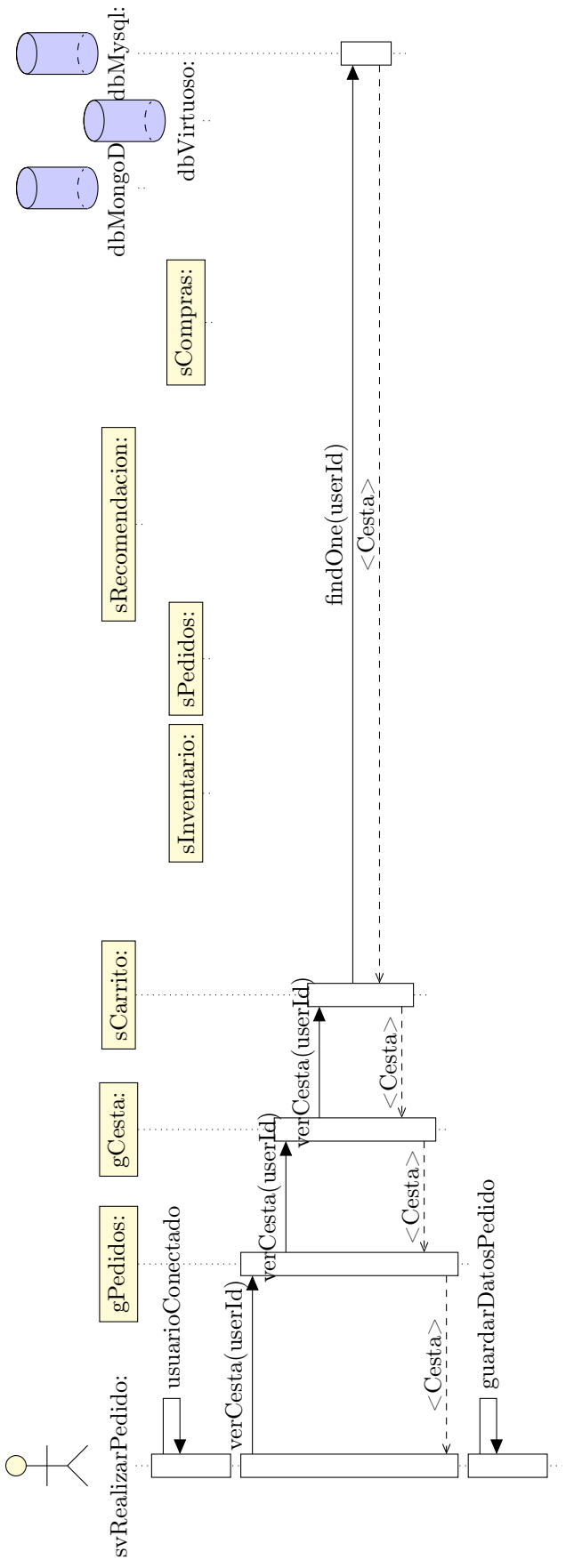

FIGURA 4.18: Diagrama de secuencia del caso de uso Realizar Pedido (I)

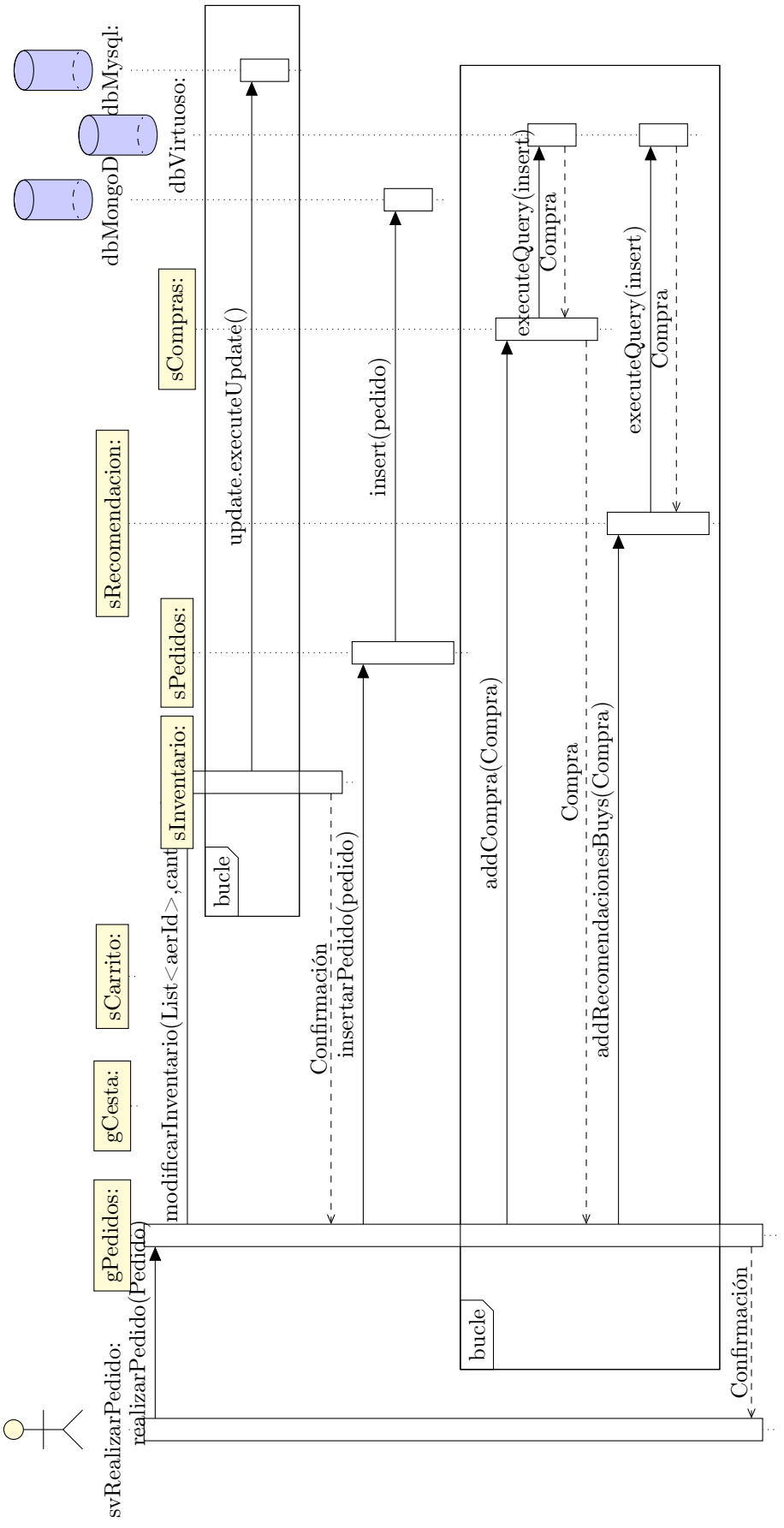

FIGURA 4.19: Diagrama de secuencia del caso de uso Realizar Pedido (II)

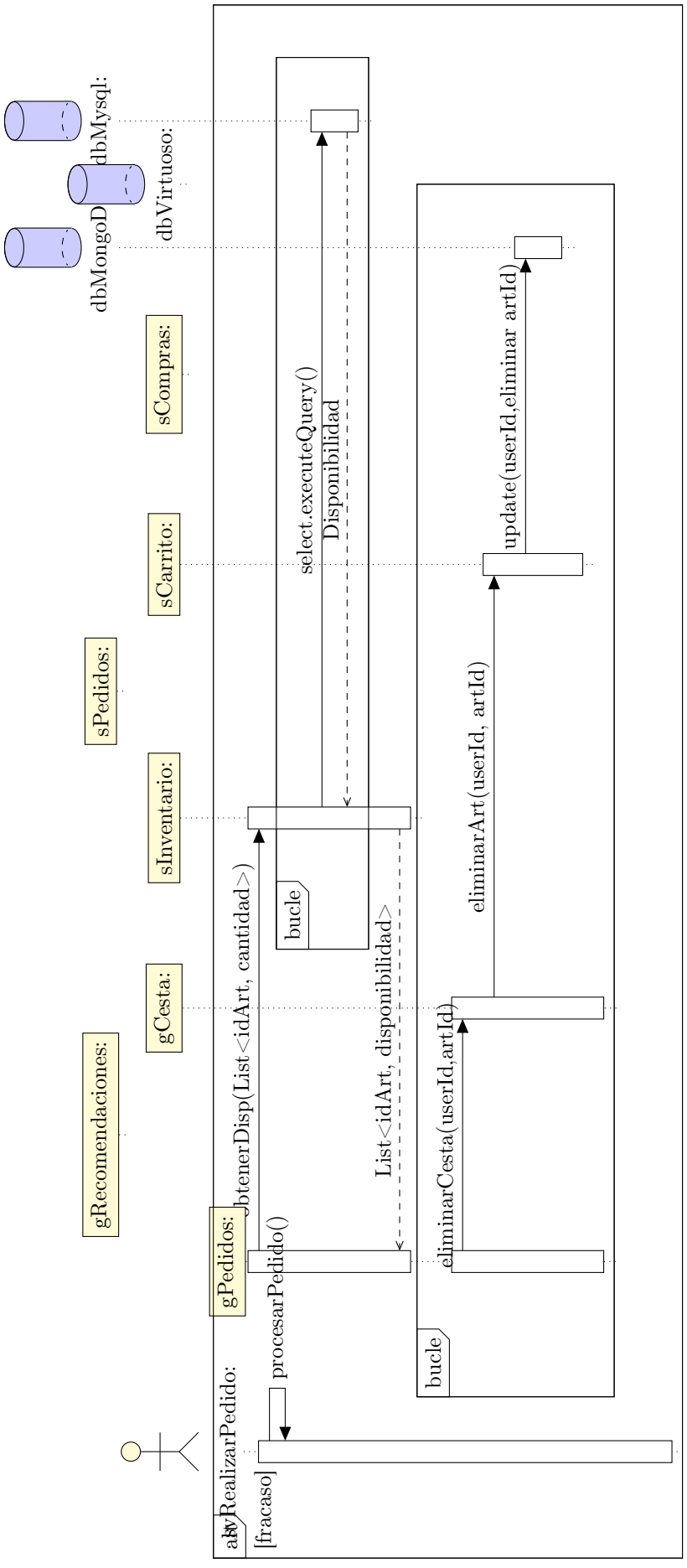

FIGURA 4.20: Diagrama de secuencia del caso de uso Realizar Pedido (III)

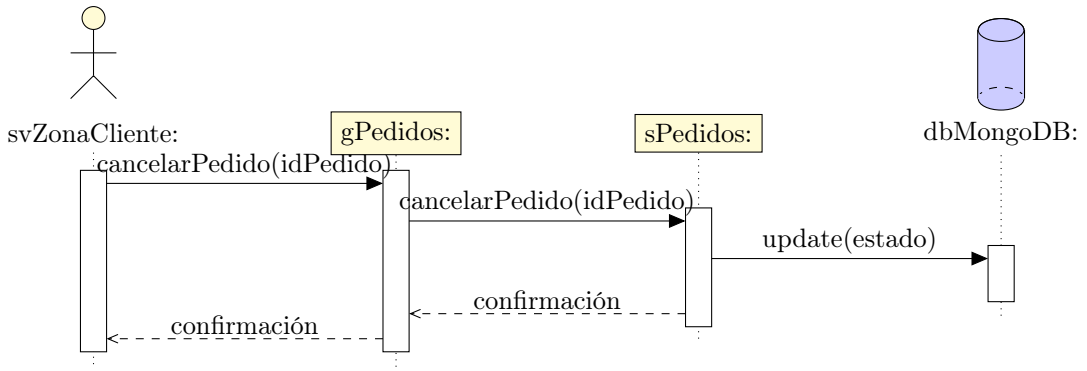

FIGURA 4.21: Diagrama de los casos de uso Cancelar Pedido

4.4 Pruebas de errores

Para que la *Aplicación de Persistencia Políglota* mantenga un grado de estabilidad y no genere errores inesperados durante el uso de la misma se han realizado una serie de pruebas.

Estas pruebas, al igual que la aplicación en si, se han dividido en tres partes:

1. Servicios:

   En una primera fase se han realizado las pruebas sobre los servicios que realizan las consultas a los distintos sistemas. Para esto se han realizado una serie de pruebas de unidad mediante el entorno *JUnit*.
   El diseño general de las pruebas ha consistido en comprobar: (1) que los datos solicitados se correspondan con los esperados y que se guardan correctamente en los objetos de la aplicación; y (2) que tras guardar datos en los sistemas de almacenamiento, al solicitar los datos de nuevo se obtengan los datos esperados.

   En cuanto a los resultados, la mayor parte de los errores obtenidos fueron fallos mínimos como guardar algún dato en un atributo erróneo.

   Si bien, merece la pena mencionar dos casos concretos de error generados al usar de MongoDB. El primero se debió a la flexibilidad que da MongoDB, ya que permite que dos documentos en una misma colección tengan campos de diferente tipo pero con el mismo nombre. En este caso, existen artículos con campos como *idioma* o *productora* que pueden contener datos que pueden ser tanto listas de *String* como *String* simples - esto es debido a que se usó un dataset externo para generar

los datos. Por tanto, al guardar los datos obtenidos desde el sistema de almacenamiento se producían errores por incompatibilidad en el tipo de datos. Para solucionar esto se cambió el tipo de datos de los atributos *idioma y productora* de la clase anidada *Detalles*. Como tipo de datos de estos atributos se usó el tipo *Object* y se modificaron los métodos *getIdioma/Productora()* y se crearon los métodos *getIdiomas* y *getProductoras()*; para que devolvieran un *String* y una lista de *String*, respectivamente.

El segundo caso, es similar, salvo que el problema se debía a como trabaja el cliente de MongoDB con los números. En principio al generar el dataset, los campos precio y el descuento de los artículos eran de tipo *float* e *integer* respectivamente. Sin embargo, tras modificar algunos de los datos desde el cliente - sumas y restas - los datos modificados se guardaron con el tipo *double*. De nuevo, esto generaba un conflicto con los tipos y, de nuevo, se solucionó cambiando el tipo de los atributos a *Object* y modificando los métodos para que devolvieran los datos con el tipo deseado.

2. Gestores:

   En esta segunda fase se realizaron las pruebas sobre los gestores para comprobar la estabilidad de las diferentes operaciones que realiza la aplicación. Al igual que en el caso anterior se utilizó el entorno *JUnit* para realizar las pruebas.

   En este caso el diseño general de las pruebas consistió en llamar a los métodos que realizan las operaciones principales y comprobar tanto que los datos recibidos eran los esperados, como que se realizaran las consultas necesarias para llevar a cabo las operaciones. En este caso no se detecto ningún error grabe.

3. Presentación:

   En esta fase se realizaron pruebas para comprobar la estabilidad de la aplicación ante la interacción de los usuarios con las páginas web. En este caso al realizar las pruebas únicamente se comprobó que en los pasos que podría hacer un usuario para los diferentes casos de usos; la aplicación funcionaba correctamente. Es decir, se mostraba la información solicitada y se realizaban las operaciones sin producirse fallos. Además, se comprobó a mano que al realizar pedidos, crear una cesta o al crear un usuario, los datos se guardaban correctamente en las bases de datos.

En cuanto a los fallos encontrados en general se limitaron a la presentación erronea de algunos datos y a algún error en el código *JavaScript* al formar algunas de las peticiones http.

Si bien, cabe destacar un error de diseño que se detectó durante la operación destinada a realizar los pedidos. En esta operación se tenía pensado que un usuario hiciera los siguientes pasos: (1) el usuario solicita comprar los artículos de la cesta, (2) el usuario rellena un formulario para los datos del pedido,(3) el sistema modifica el inventario - si todos los artículos están disponibles -; (4) el usuario confirma la compra. El problema principal estaba en el paso (4), ya que aunque el usuario tenía un botón de cancelar que permitía deshacer los cambios en el inventario, si el usuario cerraba la página estos cambios no se desharían. Por tanto, para solucionar esto se elimino el paso (4).

4.5 Tecnologías de Desarrollo

En esta sección se van a mostrar las diferentes tecnologías que se han usado para desarrollar y manterner la aplicación. Sin incluir a los sistemas de almacenamiento de los que ya se ha hablado en capítulos anteriores.

4.5.1 Implementación central de la aplicación

En este punto se van a mostrar las tecnologías usadas para la implementación de la parte central de la aplicación.

4.5.1.1 Java

Como lenguaje principal para implementar la aplicación se ha usado Java, entre otras razones por: (1) el conocimiento previo de esta tecnología, (2) la presencia de drivers oficiales en todos los sistemas de almacenamiento para este lenguaje; y (3) el hecho de que existen entornos como J2EE que facilitan la implementación de aplicaciones web de varias capas.

Este lenguaje [29] es un lenguaje de programación de alto nivel y orientado a objetos basado en clases. Generalmente los programas implementados con este lenguaje se compilan a *Bitecodes* que después son ejecutados en una máquina virtual java. Ésto permite que los programas desarrollados en este lenguaje puedan ser portados fácilmente a distintos sistemas informáticos que tengan dicha máquina virtual instalada.

4.5.1.2 J2EE

J2EE [30] es un entorno construido sobre la plataforma Java SE - que es una API la cual ofrece la funcionalidad básica del lenguaje Java.

Esta plataforma, J2EE, ofrece una serie de servicios, API's y protocolos que permiten desarrollar aplicaciones de gran escala, soportando la división de la aplicación en varias capas.

Algunas de las tecnologías que ofrece son:

- *Servelets*: son clases que procesan peticiones y construyen respuestas de forma dinámica en general orientados a construir contenido *html*.
- *JSP*: son documentos en forma de texto que son compilados a *servelets* y permiten generar contenido web dinámico.
- *EJB*: son componentes que generalmente contienen la lógica de negocio y se encargan de implementar la funcionalidad principal de la aplicación.
- *JDBC*: es una API de bajo nivel que permite acceder y realizar consultas sobre bases de datos externas a la aplicación.

4.5.1.3 JSON y GSON

JSON [31] es un formato orientado al intercambio de datos, cuyas propiedades lo hacen ideal para el intercambio de datos. La estructura en la que es construida tiene dos formas: (1) una colección de parejas de tipo clave-valor; y (2) una lista ordenada de valores. Estas son estructuras universales soportadas por gran cantidad de lenguajes de programación. Para Java existen varias librerías que permiten realizar diferentes operaciones.

En este caso se ha utilizado la librería GSON de Google, que entre sus características principales está la de permitir a partir de cadenas de caracteres que siguen el formato JSON de clave-valor; pasar los valores a una estructura de datos especifica, donde el nombre de los atributos coinciden con el de las claves.

La razón por la que se ha utilizado esta librería es que los objetos de MongoDB que representan a los documentos, tienen una estructura similar a la de los objetos JSON. Por tanto, usando la librería anterior es posible pasar de forma casi automática los datos de estos objetos a los objetos usados en la aplicación y viceversa.

4.5.2 Implementación de la página web

En este punto se habla de las tecnologías usadas para implementar las páginas web que presentan la aplicación a los usuarios.

4.5.2.1 HTML y CSS

HTML [32] es un lenguaje de marcado que permite definir la estructura con la que se mostrarán las páginas web.

CSS, por su parte, es un lenguaje de hojas de estilo, que permiten definir la apariencia con la que se presenta la estructura definida en HTML u otro lenguaje de marcado.

4.5.2.2 JavaScript y jQuery

JavaScript [33] es un lenguaje que se ejecuta mediante un interprete sin ser compilado. Si bien existen interpretes que permiten ejecutar programas a nivel del servidor; generalmente se usan en el lado del cliente y son interpretados por los navegadores. En este segundo caso, JavaScript permite modificar la página web de forma dinámica, para cambiar la apariencia o añadir funcionalidades.

jQuery [34] es una librería de JavaScript que permite mediante una API, manipular la página, manejar eventos y realizar consultas de forma sencilla.

4.5.3 Pruebas

En este punto se habla de las tecnologías utilizadas para comprobar la estabilidad de la aplicación.

4.5.3.1 JUnit

JUnit [35] es un *framework* para el lenguaje de programación Java, que permite realizar test repetibles de unidad.

Este *framework* permite ejecutar clases de manera controlada y comprobar que los resultados obtenidos son los esperados. De esta forma, si el valor obtenido es el esperado, JUnit indica que se ha pasado el test, pero si no es el valor esperado, se indicará que no se ha pasado el test.

4.5.3.2 Mozilla Firefox

Mozilla Firefox [36] es uno de los navegadores web más conocidos. Entre otras características, destacan las herramientas de desarrollo que permiten no sólo ver el código - tanto HTML y CSS como JavaScript -, sino también permiten visualizar las peticiones *GET/POST* junto con el tiempo de respuesta que tarda. Con esto se facilita que se encuentre donde se ha producido un fallo en caso de que se dé algún error al realizar operaciones sobre la página web.

4.5.4 Servidor Web

En este punto se habla de las tecnologías usadas para servir las páginas web a los clientes.

4.5.4.1 Apache Tomcat

Apache Tomcat [37] es un servidor web, no de aplicaciones. A pesar de esto, implementa especificaciones de los *servlets* y *JSP*, lo que hace que pueda funcionar como un contenedor de servelet y ejecutarlos para mostrar las páginas generadas por estos.

Una de las razones principales para elegir esta tecnología para servir las páginas web ha sido debido a que por un lado ya se había tenido contacto con esta tecnología previamente, y por otro porque es una herramienta que consume pocos recursos.

4.5.5 Desarrollo

En este punto se habla de las tecnologías utilizadas para desarrollar la aplicación.

4.5.5.1 Netbeans

NetBeans [38] es un entorno de desarrollo orientado principalmente al desarrollo en el lenguaje de programación Java. Si bien es cierto que los paquetes oficiales disponibles también incluyen herramientas de desarrollo para otros lenguajes.

Este entorno de desarrollo permite que las aplicaciones se dividan y se desarrollen de forma separada en módulos, permitiendo además que a los módulos se les pueda añadir nuevos móodulos.

Una de las principales razones para elegir este entorno es que se ha tenido contacto previo con esta tecnología y se ha visto que no sólo es sencillo de utilizar sino que también es bastante estable.

4.5.6 Otros

Además de las tecnologías anteriores también se han utilizado otra serie de servicios y tecnologías que han permitido desarrollar el proyecto.

4.5.6.1 FreeBase

Freebase [39] es una base de datos colaborativa en la que los datos son creados por los usuarios. Esta base de datos contiene millones de datos de diferentes temas como películas, libros, personas, música... Todos ellos accesibles libremente.

Además está disponible una API para, entre otros, el lenguaje Java que permite acceder a los datos desde las aplicaciones.

En este caso, tanto los datos como la API se han utilizado para generar los dataset con los que se ha probado la aplicación. Una de la razones principales por las que se ha usado esta base de datos como fuente para generar los datasets ha sido la gran cantidad de datos que hay presentes. Aunque, hay que destacar que el hecho de que los datos sean creados por cualquier usuario hace que en algunos casos se presenten campos con valores vacíos o haya campos con el mismo nombre pero que guardan datos de distinto tipo. Sin embargo, esto es una minoría de datos y en general permite acceder a una gran cantidad de datos.

4.5.6.2 Dropbox

Dropbox [40] es un servicio de almacenamiento en línea que permite guardar archivos. Estos archivos se guardan en un servidor y pueden ser accedidos remotamente. Además, el servicio permite acceder a los datos guardados tanto desde una página web como desde el mismo escritorio, proporcionando para esto una aplicación que permite sincronizar ciertas carpetas del ordenador con los archivos guardados en el servicio.

Una de las principales razones para su uso es que se ha usado, entre otras cosas, para almacenar datasets que llegan a ocupar GB's de datos. En este caso, se disponía de una cuenta que tiene el espacio suficiente para almacenar dichos archivos.

Capítulo 5

Aplicación con un Sistema Relacional: Comparación

Con la intención de ver qué diferencias pueden existir entre una aplicación con una base de datos relacional y la aplicación creada para el proyecto, se ha creado una segunda aplicación con una base de datos relacional. En concreto se ha utilizado MySQL como sistema de almacenamiento relacional, porque ya se tenía conocimiento previo de este sistema y porque es fácilmente accesible.

Por tanto, a lo largo de este capítulo se va a mostrar una comparación entre ambas aplicaciones incluyendo: tanto (1) las diferencias encontradas al diseñar la aplicación y los modelos de datos; como (2) los resultados obtenidos tras realizar una serie de pruebas de rendimiento entre las dos aplicaciones sobre un mismo sistema informático.

5.1 Arquitectura

Con la idea de reutilizar la mayor parte del código implementado para la *Aplicación de Persistencia Políglota*, se ha seguido una arquitectura similar.

Por un lado se ha seguido el diseño en tres capas. Éste no es dependiente del número de bases de datos utilizadas para almacenar los datos de la aplicación, por lo que se seguirá el mismo diseño en el que los *servelets* y las páginas web forman la capa de presentación; los gestores la Lógica de Negocio, y los servicios la capa de Datos.

Por otro lado, a la hora de diseñar la capa de Datos, como se ha dicho también se ha seguido la idea de dar acceso a los datos mediante servicios. Una de las principales razones para hacer esto ha sido la reutilización del código de la aplicación *Políglota*.

Para diseñar los servicios existen dos posibilidades:(1) crear un único gestor que realice las consultas al sistema y que implemente todos los servicios; (2) seguir el diseño de la aplicación *Políglota* e implementar cada servicio por separado. En este caso, puesto que resulta más cómodo organizar las consultas por servicios se han diseñado los servicios de forma similar a la aplicación original.

Como la parte de Presentación y de Negocio de ambas aplicaciones es similar, en este caso no hay diferencias.

Sin embargo, a la hora de diseñar la arquitectura existirían varias opciones. Sería posible haber eliminado por completo el concepto de los servicios y que la aplicación no fuera consciente de qué sistema se utilizaba para almacenar los datos. También hubiera sido posible crear el gestor de consultas que se comento antes, si la aplicación fuera sencilla y no requiriera de muchas consultas por servicio.

Por tanto, la arquitectura de este tipo de aplicaciones es algo más flexible y los diferentes diseños son menos problemáticos que los expuestos al comienzo de la documentación. Por ejemplo, en este caso puesto que la aplicación se basa en que sólo hay y habrá un único sistema de almacenamiento; si se cambiara dicho sistema se verían afectados todos los servicios independientemente de si las consultas se implementaran sobre un gestor o sobre cada servicio. Por tanto, no habría que pensar en que pasaría si se cambiara un sistema y no el resto.

5.2 Modelo de datos

El sistema que se usa para almacenar los datos, es un sistema relacional, en el que se usan tablas y relaciones para modelar los datos. Por tanto, para modelar los datos que utiliza la aplicación se ha diseñado el esquema de la figura 5.1.

Como se puede ver en este caso se han guardado los datos en una única base de datos. Si bien sería perfectamente válido guardar los datos en varias bases de datos sobre el mismo sistema, se ha usado sólo una para simplificar el diseño.

5.2.1 Comparación con MongoDB

Para la aplicación *políglota*, MongoDB se encargaba de almacenar los datos de los artículos, las cestas y los pedidos.

En el caso de los artículos, los datos son guardados en varias tablas, a diferencia de en MongoDB donde sólo se utilizaba una única colección. Por un lado, puesto que existen

FIGURA 5.1: Modelo de datos para MySQL

varios tipos de datos con atributos diferentes, sería necesario como mínimo tener una tabla por cada tipo. En este caso, puesto que la intención del modelo es normalizar los datos se tienen tres tablas principales: (1) una tabla de artículos genéricos; (2) una tabla para los detalles de los libros; y (3) una tabla para los detalles de las películas. Esto por si sólo no aumenta la complejidad del esquema en gran medida; sin embargo, por cada nuevo tipo de artículo que se quisiera tener habría que añadir una nueva tabla, aumentando con ello la complejidad del esquema.

Por otro lado, siguiendo con los artículos, en los documentos de MongoDB hay campos que son de tipo array, es decir, que el campo contiene varios valores. Sin embargo, en MySQL esto no es posible. Existen dos posibilidades para guardar campos como la lista de actores: (1) guardar la lista de elementos como texto; o (2) crear una tabla a parte con los datos de cada elemento y el identificador del artículo en el que están. En este caso, se ha optado por la segunda opción ya que querer guardar varios datos por cada elemento - por ejemplo por autor guardar además del nombre la fecha de nacimiento y muerto - los campos de texto no serían lo suficientemente flexibles. Con esto, se estaría de nuevo aumentando la complejidad del esquema - a cambio de una cierta flexibilidad.

Por tanto, sólo con los artículos, ya se tendrían siete tablas para representar los diferentes tipos de artículos. Lo cual en MongoDB requeriría una única colección.

En lo que se refiere a los carritos de la compra y a los pedidos, se usa el mismo esquema donde hay una tabla con los datos generales de los carritos/pedidos y otra tabla con los datos de los artículos que contienen los diferentes carritos/pedidos. En MongoDB cada uno de estos se guarda en una colección donde los documentos contienen campos con la información general del carrito/pedido y un campo con una lista de los artículos elegidos. Sin embargo, el uso de dos tablas no es más complejo que la estructura interna de los documentos de los carritos/pedidos. Finalmente, en MongoDB se guardaba por cada artículo presente en los carritos/pedido, ciertos datos como el título o el formato, dentro del mismo documento. Sin embargo, en este caso puesto que estos datos ya están en la tabla artículos, no se añaden a la tabla que contiene los datos de los artículos por pedido/carrito. Ésto podría ayudar a reducir el tamaño de la base de datos, si hubiese muchos carritos/pedidos o si se repitieran muchos artículos.

5.2.2 Comparación con Virtuoso

Para la aplicación *políglota*, Virtuoso se encargaba de almacenar los datos de los usuarios, las compras y las valoraciones.

En esta caso se tendrán cuatro tablas: para los usuarios, para las compras, para los artículos que les han gustado a los usuarios; y para las valoraciones. Usando Virtuoso se tenía un subgrafo por cada uno de los objetos anteriores, sin embargo, en esta nueva aplicación no se tiene una tabla nueva con los datos de los artículos, sino que se relacionan las compras y las valoraciones a la tabla de artículos comentada en el punto anterior. Otra diferencia es que ahora se tiene una tabla para los gustos, donde antes estos se indicaban como un dato más en el grafo de los usuarios.

Finalmente, una de las principales diferencias es que las relaciones entre los datos se limitan a relacionar tablas mediante un atributo común, sin dar la opción a indicar el tipo de relación.

5.2.3 Comparación con MySQL

En la aplicación original, los datos relacionados con el inventario se guardaban en una tabla a parte, en este caso simplemente se incluye un atributo en la tabla *artículos*.

5.3 Consultas

Uno de los principales problemas al usar varios sistemas de almacenamientos NoSQL, es que en cada uno la forma en la que se pueden realizar las consultas es diferente. En algunos casos se utiliza un lenguaje de consultas estandarizado como ocurre en Virtuoso con *SPARQL*; en otros se utilizan métodos propios como en MongoDB en el que se utiliza un lenguaje propio similar a *JavaScript* con funciones específicas para realizar las consultas.

En general, esto resulta ser un problema porque requiere del aprendizaje de los distintos métodos para realizar las consultas, los cuales pueden variar no solo de un tipo de sistema de almacenamiento a otro; sino entre sistemas del mismo tipo, por ejemplo en los sistemas de documentos MongoDB y CouchDB.

Este problema no se da en los sistemas relacionales, en los que prácticamente en todos se utiliza *SQL* - en ciertas ocasiones con pequeñas variaciones en la sintaxis.

Por tanto, una aplicación con un único sistema de almacenamiento relacional requerirá un menor grado de aprendizaje - aunque no se tengan conocimientos de *SQL* - y el cambio de un sistema relacional a otro no generará muchos problemas en este sentido.

Por otro lado, también hay que tener en cuenta el nivel de complejidad de las consultas, lo cual viene dado no sólo por el lenguaje, sino por otros factores como la complejidad del modelo o de las operaciones a realizar.

5.3.1 Comparación con MongoDB

Algunas operaciones simples como la de obtener un artículo en función del identificador resultan sencillas de realizar en MongoDB con un simple:

```
> db.articulos.find({_id: "5789f97e6a9e"})
```

Sin embargo, a la hora de realizar esta misma operación sobre MySQL, la consulta necesaria será más complicada. Esto se debe principalmente a que los datos de un artículo están distribuidos en varias tablas con lo que es necesario hacer varios *JOIN*'s. Además, los datos de las tablas como *actores* y *autores* podrán devolver varias filas por cada artículo, con lo que para obtener los datos habría que o bien realizar varias consultas o usar funciones de agregación como *GROUP_ CONCAT()*. Por ejemplo, usando la función de agregación para obtener el artículo sería las dos consultas siguientes:

```
> SELECT *,
  GROUP_CONCAT(Distinct Ac.nombre) As Autores
  FROM ((Articulos As A
                LEFT OUTER JOIN  bookDetalles As B ON A._id = B._id)
        LEFT OUTER JOIN autores As Ac ON A._id=Ac._id)
  WHERE A._id=2
  GROUP BY A._id

> SELECT *,
  GROUP_CONCAT(Distinct Ac.nombre) As Actores,
  GROUP_CONCAT(Distinct D.nombre) As Directores,
  GROUP_CONCAT(Distinct G.nombre) As Guionistas
  FROM (((Articulos As A
           LEFT OUTER JOIN  movieDetalles As B ON A._id = B._id)
       LEFT OUTER JOIN actores As Ac ON A._id=Ac._id)
       LEFT OUTER JOIN director As D ON A._id=D._id)
       LEFT OUTER JOIN guionista As G ON G._id=Ac._id
  WHERE A._id=1
  GROUP BY A._id
```

Como se puede observar, la necesidad de hacer uniones entre las diferentes tablas que guardan los datos hace que la complejidad de consultas sea alta.

Por otro lado, las inserciones de elementos como por ejemplo los Pedidos, es compleja en ambos casos. En el caso de MySQL es necesario hacer varias inserciones, primero habrá que guardar la información general del pedido en la tabla *Pedido* y después cada uno de los artículos del pedido en la tabla *pedidoArt*. En el caso de MongoDB al insertar el pedido habrá que crear todo el documento y después insertarlo; que dependiendo de la complejidad del esquema puede llevar a errores. Si bien, a nivel de aplicación se puede solventar este último problema como se verá más adelante.

Por último, una de las operaciones de actualización como es modificar la cantidad elegida de una cesta, en MySQL tendría la forma:

```
> UPDATE carritoArt
  SET carritoArt.cantidad = 2
  WHERE carritoArt._id = 2 and
        carritoArt.idCarrito=
                (Select C.idCarrito
                 FROM carrito as C
                 WHERE C.idUser= "user1@email.com"
            )
```

Mientras que en MongoDB la consulta sería:

```
> db.carrito.update(
        {userId:"user1@email.com",
    'articulos.artId':2},
    {$set:{articulos.$.numArt:2}})
```

En ambos casos hay un cierto grado de complejidad. En el primer caso, es necesario encontrar el identificador del carrito en función del usuario y usarlo para filtrar los artículos del carrito a modificar. En el segundo caso, es necesario encontrar la posición del array en la que se encuentra el artículo de la cesta a modificar.

5.3.2 Comparación con Virtuoso

En Virtuoso las consultas se pueden realizar mediante el leguaje de consultas *SPARQL*. Este lenguaje es similar a *SQL*, por lo que para operaciones básicas como buscar las credenciales de los usuarios, las consultas serán muy similares y tendrán una complejidad baja.

Incluso, en operaciones que requieren datos con un cierto grado de relación, no hay una gran diferencia de complejidad entre las consultas de los dos sistemas. Por ejemplo, si se

quieren obtener todas las valoraciones de un usuario incluyendo el título del artículo, en Virtuoso la consulta sería:

```
PREFIX  does:<http://applV.com/actions/>
PREFIX  article:<http://applV.com/art/article/>
PREFIX  review:<http://applV.com/rev/review/>
SELECT *
FROM <http://applV.com/>
WHERE{
       <mailto:user1@email.com> does:reviewing ?r
       ?r    review:of  ?a .
       ?rev  rev:description ?desc .
       ?rev  rev:rating ?rating .
       ?a article:title ?titulo
}
```

Mientras en MySQL, está misma operación requeriría realizar la consulta:

```
SELECT *
FROM (user AS U JOIN Review AS R ON U.userID = R.idUser)
     JOIN Articulos AS A ON R.idArt=A._id
WHERE U.userID = "user1@email.com"
```

Sin embargo, con operaciones que requieren obtener datos muy relacionados, la complejidad de las consultas en MySQL empieza a aumentar. Por ejemplo, para obtener todos los artículos que unos usuarios han comprado, siempre que estos usuarios hayan comprado un artículo concreto. Esta consulta en Virtuoso se haría de la siguiente forma:

```
PREFIX does:<http://applV.com/actions/>
PREFIX  article:<http://applV.com/art/article/>
PREFIX  buy:<http://applV.com/purchase/buy/>
SELECT ?u ?l ?id ?t
FROM <http://applV.com/>
WHERE{
       ?a  article:id "2".
       ?b  buy:article ?a .
       ?u  does:buying ?b .
       ?u  does:buying ?l .
       FILTER(?b != ?l) .
       ?l buy:article ?a2 .
       ?a2 article:id ?id .
   ?a2 article:title ?t
}
```

Mientras que esta misma operación en MySQL requeriría la siguiente consulta:

```
SELECT _id = "2"
FROM   Compra
WHERE  NOT(_id="2") AND
       idUser IN ( Select C2.idUser
                   From Compra as C2
                   Where C2._id=2
                 )
```

Si bien es cierto que la consulta para Virtuoso es más larga, la consulta para MySQL requiere anidar una consulta *SELECT* para comprobar que los usuarios han comprado el artículo dado. Mientras, en el caso de Virtuoso la mayor complejidad se encuentra en la introducción de un filtro para evitar devolver el artículo original.

5.4 Datos en la aplicación

En las aplicaciones, los datos que se reciben de las consultas serán guardados en estructuras propias. Por tanto, será necesario pasar los datos de las estructuras obtenidas tras las consultas a los sistemas de almacenamiento; a las estructuras propias de las aplicaciones.

5.4.1 Comparación General

En general la estructura utilizada para recibir los resultados, viene dada por el driver que permite interactuar con cada sistema de almacenamiento.

En el caso de MySQL y Virtuoso se utilizan dos drivers *JDBC* muy similares. En ambos casos el resultado se devuelve en una estructura de tipo *Resulset*. Esta estructura está formada por filas y columnas, donde cada fila es un resultado. Para pasar los datos de cada fila de la estructura a las estructuras de datos usadas por la aplicación, es necesario hacerlo a mano. Es decir, se obtienen los datos con los métodos *.get(Columna)* y se le asigna al atributo correspondiente de la estructura deseada.

El caso de MongoDB es algo diferente ya que los datos recibidos se guardan en unas estructuras llamadas *DBObject*, que son similares a las estructuras *JSON*. Seguidamente se muestran cuales son las diferencias al pasar los datos de una estructura otra.

5.4.2 Comparación con MongoDB

Si se quisiera, de nuevo, obtener los datos de un artículo en función del identificador del artículo, habría que pasar los datos obtenidos en el *Resulset* a un objeto de la clase *BookArtic* - suponiendo que es de tipo libro - de la siguiente manera:

```
BookArtic obj = new BookArtic();
obj.setTitulo(resulset.getString("titulo"));
obj.setFormato(resulset.getString("formato"));
obj.setId(resulset.getString("_id"));
obj.precio.setPrecio(resulset.getFloat("precio"));
obj.precio.setDescuento(resulset.getInt("descuento"));
obj.precio.setEnvio(resulset.getString("envio"));
obj.detalles.setEditorial(resulset.getString("editorial"));
obj.detalles.setPaginas (resulset.getInt("paginas"));
obj.detalles.setFecha(resulset.getString("fecha"));
obj.detalles.setIdioma(resulset.getString("idioma"));
obj.detalles.setDescripcion(resulset.getString("descripcion"));
Array a = rs.getArray("Autores");
obj.detalles.setAutores((List<String>)a.getArray());
```

Como se puede ver, pasar los datos de una estructura a otra es algo tedioso. Algo similar ocurre si se pasan los datos directamente de un objeto *DBObject* del driver de MongoDB, a la estructura *BookArtic*. Llegando incluso a ser más complejo que en el caso anterior.

```
BookArtic obj = new BookArtic();
obj.setTitulo((String)dbobject.get("titulo"));
obj.setFormato((String)dbobject.get("formato"));
obj.setId((String)dbobject.get("_id"));
obj.precio.setPrecio(
            Float.valueOf(
                ((DBObject)dbobject.get("precio")).get("precio")
           );
obj.precio.setDescuento(
            Integer.valueOf(
                ((DBObject)dbobject.get("precio")).get("precio")
           );
obj.precio.setEnvio(
            ((DBObject)dbobject.get("precio")).get("precio").toString
           );
obj.detalles.setEditorial((String)
            ((DBObject) dbobject.get("detalles")).get("editorial")
           );
obj.detalles.setPaginas ((Int)(DBObject)
             dbobject.get("detalles")).get("paginas")
```

```
            );
 obj.detalles.setFecha((String)
             (DBObject) dbobject.get("detalles")).get("fecha")
            );
 obj.detalles.setIdioma((String)
             (DBObject) dbobject.get("detalles")).get("idioma")
            );
 obj.detalles.setDescripcion((String)
             (DBObject)dbobject.get("detalles")).get("descripcion")
            );
 obj.detalles.setAutores((List<String>
             (DBObject)dbobject.get("detalles")).get("autores")
            );
```

A pesar de que pasar los datos del un objeto *DBObject* a otro de tipo *BookArtic* es tedioso y propenso a errores si se hace de la manera anterior, es posible pasar los datos de una estructura a otra de forma automática. Esto se debe en gran parte a que la estructura de datos *DBObject* es similar a la estructura *JSON*. Más concretamente, si se pasa el objeto *DBObject* a *String*, el conjunto de caracteres obtenido es idéntico al que se obtendría de un objeto *JSON*. De esta forma, es posible valerse de una API como *gson* la cual permite pasar datos de tipo *String*, que representan objetos *JSON*, a cualquier estructura que se desee. Por tanto, usando este método la operación anterior se realizaría de la siguiente manera:

```
Gson gson = new Gson();
String data;
data = prod.toString();
BookArtic obj = gson.fromJson(data, BookArtic.class);
```

La única pega es que tanto los campos de los documentos como los atributos de la clase deben tener el mismo nombre. Sin embargo, no es necesario que tengan los mismos campos/atributos. Es decir, la estructura de datos de la aplicación puede tener un atributo que no exista en el documento, y viceversa.

Otro detalle importante es que si se quiere guardar el campo *_id* de tipo *ObjectID*, es necesario sobrescribir una de la funciones de *GSON* de la siguiente manera:

```
public Gson gSonBuilder(){
  JsonDeserializer<ObjectId> des = new JsonDeserializer<ObjectId>() {
     @Override
     public ObjectId deserialize (JsonElement je, Type type,
```

```
            JsonDeserializationContext jdc) throws JsonParseException {

          return new ObjectId(je.getAsJsonObject().get("\$oid").\
    getAsString());
         }
    };
Gson gson = new GsonBuilder().registerTypeAdapter(
                                ObjectId.class, des).create();
return gson;
 }
```

Y usar *gSonBuilder()*, en lugar de *new Gson()*.

En cualquier caso, este último método es más simple que cualquiera de los otros dos, y hace que agrupar datos en estructuras de datos complejas sea más sencillo con MongoDB.

También hay que señalar que al insertar datos en MySQL, será necesario también incluir los datos en la consulta de forma manual. Sin embargo, puesto que en MongoDB al insertar un documento - a nivel de aplicación - hay que dar al método *insert()* un objeto *DBObject*, también es posible usar *GSON* para pasar los datos de la clase, por ejemplo, *Pedido* a un objeto de la clase *DBObject* de forma automática.

Todo esto hace que pasar los datos de las bases de datos a las aplicaciones sea mucho más sencillo en MongoDB, en especial si las estructuras de los datos utilizadas son muy complejas.

5.5 Pruebas de rendimiento

Para comprobar si existe una mejora en el rendimiento de la aplicación a usar varios sistemas de almacenamiento, se han realizado una serie de pruebas sobre las distintas operaciones que realizan las dos aplicaciones, tanto a nivel de los servicios como a nivel de la aplicación web. Estas operaciones no incluyen operaciones sobre la cesta o los pedidos ya que no hay un dataset creado para estos datos.

5.5.1 Preparación de los Sistemas

Para mejorar el rendimiento de todos los sistemas e intentar hacer las pruebas bajo las mejores condiciones posibles se han realizado una serie de configuraciones.

5.5.1.1 MongoDB

En este caso, la principal mejora que se ha hecho es la de añadir una serie de indices que cubren las posibles consultas que se puedan realizar.

En concreto los índices que se utilizan son:

- El índice (genero, tipo,formato,precio.precio): este índice cubre las búsquedas fijas de la aplicación para buscar artículos en función del genero y el tipo, tanto si se incluye el formato como si no. Además permite ordenar en función del precio si se incluye el formato.
- El índice (genero, tipo,precio.precio): este índice cubre las búsquedas fijas de la aplicación para buscar artículos en función del genero, tanto si se incluye el tipo como si no. Además permite ordenar en función del precio aunque no se incluya el formato.
- El índice (genero, tipo,formato,titulo): idéntico al primero salvo que permite ordenar en función del título.
- El índice (genero, tipo,titulo): idéntico al segundo salvo que permite ordenar en función del título.
- El índice de texto (titulo, detalles.autores, detalles.actores, detalles.director, detalles.guionista): este índice permite ver si alguno de los campos anteriores de un documento contiene una o varias de las palabras. Este tipo de búsquedas sería similar a realizar búsquedas con expresiones regulares sobre cada uno de los campos; sin embargo, son más eficientes ya que la búsqueda con expresiones regulares no soporta el uso de índices.

5.5.1.2 Virtuoso

En este caso los índices del almacenamiento de tripletas son a nivel de tabla y no de grafo, y están especialmente diseñados para dar el mayor rendimiento posible. En principio se decidió dejar los índices que vienen por defecto en Virtuoso; sin embargo, al realizar una pruebas iniciales con estos índices se ha visto que al comparar los resultados con los de MySQL, Virtuoso parece ser peor. Para mejorar el rendimiento se han añadido varios índices; al realizar de nuevo el "explain()" se ha podido ver que las consultas usan los índices; sin embargo, al realizar las pruebas de nuevo se ha podido ver que el rendimiento no mejora y que en ocasiones empeora por lo que se ha vuelto a los índices iniciales.

Para intentar mejorar el rendimiento del sistema más haya de los índices - además de revisar las consultas -, se han hecho una serie de modificaciones al fichero de configuración *virtuoso.ini*. En concreto el cambio que ha dado una mayor mejora ha sido la modificación de los *buffers* de memoria mediante los parámetros *NumberOfBuffers* y *MaxDirtyBuffers*. Además, se ha eliminado del almacenamiento de tripletas todos aquellos grafos que no son del sistema y se ha vuelto a cargar únicamente el grafo que usa la aplicación. Además, se han eliminado y se han vuelto a crear los índices por defecto, con la intención de borrar de los índices información de datos que ya no estén presentes en el sistema.

5.5.1.3 MySQL

En este caso, al igual que con MongoDB, se ha creado una serie de índices para aumentar el rendimiento sobre ciertas consultas. Los índices creados han sido los siguiente:

- En la tabla actores, autores, director y guionista se ha creado un índice de texto sobre el campo nombre.
- En la tabla género se ha creado un índice sobre el campo del género.
- En la tabla artículo se ha creado un índice de texto sobre el campo del título del artículo. Y un índice normal con dos campos: el tipo y el formato del artículo.
- En las tablas Review y Compra se han creado dos índices uno sobre el campo del identificador del artículo y otro sobre el campo del identificador del usuario.
- En la tabla Likes sólo se ha creado un índice sobre el campo del identificador del usuario.

5.5.2 Preparación de las pruebas

Para poder realizar las pruebas se han generado datasets con una cantidad de datos similar a la de los datos de la aplicación políglota. Para esto, a partir de los datos presentes en los sistemas NoSQL se ha generado un archivo .csv por tabla y posteriormente se ha importado al sistema relacional.

Además, para realizar las pruebas a nivel de servicios, se han implementado dos programas que realizan una serie de operaciones sobre los servicios y devuelven el tiempo de cada operación. Estos programas se han ejecutado diez veces borrando la memoria del sistema tras cada ejecución y diez veces sin borrar la memoria.

Por último, también se han hecho pruebas a nivel de la aplicación web de forma manual sobre la interfaz web del comercio electrónico. Para calcular los tiempos se han utilizado las herramientas de desarrolladores del navegador Mozilla Firefox, en concreto se ha utilizado el analizador de red que calcula el tiempo que tarda el servidor en responder las solicitudes que se le hagan.

5.5.3 Resultados

Durante las pruebas iniciales la cantidad de datos es de unos cuatro millones y medio de artículos; y unos cuatrocientos mil usuarios, los cuales han hecho unos cuatro millones de valoraciones, cuatro millones de compras y cuatro millones de *"likes"*. En lo que se refiere a los artículos hay que destacar que - aunque en el caso MongoDB todos los datos de un artículo se guardan en un documento - en el caso de MySQL, además las tablas referentes a: los detalles, los autores, los actores y el género; contienen entre un millón y cuatro millones de filas cada una. En el caso de guionistas y directores el número de elementos es menor.

En la tabla 5.1 se muestran los resultados obtenidos al realizar las diferentes operaciones.

Operación	Políglota	Relacional
Pruebas sobre servicios (MongoDB)	-	-
Buscar genero de libro (MongoDB)	86ms/2ms	502ms/48ms
Ordenar genero de libro (MongoDB)	325ms/35ms	45ms/43ms
Buscar genero de película (MongoDB)	311ms/1ms	475ms/37ms
Buscar de texto libre (MongoDB)	890ms/380ms	4120ms/1362ms
Obtener Art. por id (MongoDB)	20ms/1ms	40ms/3ms
Obtener valoración por id (Virtuoso)	512ms/15ms	73ms/1ms
Obtener valoración por user (Virtuoso)	763ms/13ms	142ms/2ms
Obtener recomendación por id (Virtuoso)	521ms/16ms	319ms/2ms
Obtener recomendación por user (Virtuoso)	630ms/7ms	405ms/103ms
Pruebas sobre la aplicación	-	-
Buscar libros de comedia (MongoDB)	300ms/32ms	2297ms/251ms
Buscar libros de terror (MongoDB)	71ms/25ms	2541ms/177ms
Buscar libros biográficos (MongoDB)	50ms/19ms	13080ms/254ms
Buscar películas de comedia (MongoDB)	326ms/39ms	28364ms/235ms
Ver Artículo	-	-
Datos (MongoDB)	2ms	14ms/3ms
Valoracines (Virtuoso)	1810ms	1305ms
Recomnedacines (Virtuoso)	656ms	250ms
Login	276ms/55ms	288ms/54ms
Recomendaciones usuario (Virt/Mong)	-	-
Según Compras	2574ms/136ms	1702ms/233ms
Según Valoraciones	3459ms/194ms	2024ms/239ms
Según Likes	2840ms/17ms	1214ms/77ms
Ver Compras (Virtuoso)	237ms/3ms	288ms/3ms
Ver Valoraciones (Virtuoso)	549ms/3ms	104ms/3ms

Tabla 5.1: Comparación del rendimiento entre aplicaciones

En el caso de MongoDB en general las pruebas muestran que MongoDB es más rápido que MySQL. Para las búsquedas de listas de artículos MongoDB es siempre más rápido que MySQL, excepto en el caso en el que se busca un libro y se ordena el libro por primera vez. Esta desviación se debe a que MongoDB carga dos índices distintos para cada una de las búsquedas. En MongoDB para los datos que se buscan en función del género y el tipo de artículo - sin ordenar - existen dos posibles índices a usar, uno que usa el campo "precio.precio" y otro que usa el campo "titulo"; puesto que en la consulta para buscar el libro en función del género sólo se índica el "genero" y el "tipo", MongoDB usa el

primer índice que encuentre de los dos anteriores que en este caso es el que contiene el campo "titulo". Mientras, MySQL usa el mismo índice para las dos consultas haciendo que la consulta con ordenación sea más rápida.

Por otro lado, una gran diferencia que se puede observar en rendimiento entre MongoDB y MySQL, es al realizar búsquedas de texto. Estas búsquedas son las que permiten encontrar un artículo en función de un valor dado que puede coincidir con parte del título o el nombre de un autor, actor, director o guionista. Para este tipo de consultas, en ambos casos - MongoDB y MySQL - en lugar de usar expresiones regulares que no permiten el uso de índices, se ha usado una opción que está disponible en ambos sistemas para realizar la búsqueda de texto usando índices especiales - *text index* - que permiten aumentar el rendimiento frente a búsquedas que usen expresiones regulares. En las pruebas se ha visto, que la búsqueda de artículos con este tipo de consultas resulta más eficiente en MongoDB, esto se debe principalmente a que en el caso de MySQL los campos sobre los que se realizan las búsquedas están repartidos entre varías tablas, y al no ser posible crear un índice para varias tablas, es necesario buscar cada en una de las tablas por separado y cargar cada uno de los índices.

Por último, hay que destacar que durante las pruebas se ha visto que la precisión de las búsquedas de texto es mejor sobre MongoDB que sobre MySQL. Esto parece deberse principalmente a dos factores: (1) que MongoDB permite el uso de pesos que favorece a ciertos campos; y (2) que en MySQL los campos están divididos entre varias tablas por lo que hay que hacer uniones entre los resultados.

En el caso de Virtuoso en general las pruebas muestran que este sistema parece ser más lento. La mayor diferencia entre ambos sistemas se da en la búsqueda de valoraciones, donde MySQL es casi siete veces mejor. Sin embargo, la diferencia entre los dos sistemas en el caso de buscar las recomendaciones - incluye buscar recomendaciones en función de compras/valoraciones/likes - la diferencia entre ambos sistemas es menor, aunque MySQL sigue siendo más rápido. Uno de los motivos que puede afectar a que la diferencia sea menor en estas consultas es el hecho de que son más complejas que las dos primeras, ya que la búsqueda de recomendaciones en MySQL requiere de varios *SELECT* anidados, siendo la búsqueda en función de los usuarios la más compleja.

Por otro lado, al realizar las pruebas sobre diferentes usuarios y artículos se encontró un caso particular que hacía que la búsqueda de recomendaciones en función de los usuarios se deteriorase ampliamente en MySQL mientras se obtenían resultados similares a los otros en Virtuoso. Este caso particular ocurría al realizar la búsqueda en función de los treinta primero usuarios - *user0-user30*. Aunque en principio se pensó que la desviación se podría deber a que eran los primeros usuarios de la tabla, se vio que un factor que podría estar afectando al rendimiento era el hecho de que durante las

pruebas de estabilidad de la aplicación se utilizaban siempre los veinte/treinta primeros usuarios para realizar compras y añadir valoraciones, haciendo que en los *datasets* hubiera más valoraciones y compras para estos usuarios que para el resto. Para comprobar el rendimiento de las aplicaciones en los casos en los que los usuarios han comprado/valorado/gustado más de diez artículos se modificaron los *datasets* añadiendo un mayor número de compras/valoraciones/likes por usuario - la cantidad de valoraciones y *likes* paso a ser de ocho millones y medio; y la de las compras a seis millones y medio -. Además, se volvieron a realizar pruebas con diferentes índices y se vio que el uso de los índices por defecto junto con un índice de tipo *bitmap* sobre los campos *PSOG* mejoraba el rendimiento en algunas de las consultas, a diferencia de con los datos originales donde el rendimiento era similar que al no usar dicho índice.

Al realizar las pruebas con estos nuevos datos se obtuvieron los resultados de la tabla 5.2 - no se muestran los resultados de las consultas que haría MongoDB ya que al no modificarse los datos que este usa, los resultados son los mismos.

Operación	Políglota	Relacional
Pruebas sobre servicios (MongoDB)	-	-
Obtener valoración por id (Virtuoso)	524ms/6ms	247ms/2ms
Obtener valoración por user (Virtuoso)	1066ms/22ms	389ms/3ms
Obtener recomendación por id (Virtuoso)	705ms/16ms	328ms/3ms
Obtener recomendación por user (Virtuoso)	770ms/8ms	3717ms/536ms

Tabla 5.2: Comparación del rendimiento entre aplicaciones (II)

En las tres primeras consultas, al igual que ocurría con los datos originales el rendimiento en MySQL sigue siendo mejor que en Virtuoso. Sin embargo, en la última consulta, el rendimiento de MySQL se vio afectado en gran medida mientras que el tiempo empleado por Virtuoso para realizar la consulta sólo aumenta ligeramente. Por tanto, en este caso cuando se da que la consulta es compleja - tres niveles de anidamiento de *SELECT* para la consulta de SQL - y que hay una gran cantidad de datos; el rendimiento de MySQL se ve ampliamente afectado, mientras que el de Virtuoso sólo crece ligeramente.

5.6 Conclusiones

Durante la realización de las comparaciones y de las pruebas de rendimiento se ha podido comprobar que existe una serie de ventajas y desventajas en el uso de varios sistemas de almacenamiento.

Por un lado, los dos sistemas NoSQL permiten realizar de forma sencilla consultas que en MySQL son complejas. Además, en el caso de MongoDB el salto en el formato de la representación de datos entre objetos de la aplicación y de la base de datos se reduce hasta ser prácticamente nulo.

Por otro lado, la división de los datos entre varios sistemas ayuda a organizar y reducir la complejidad de los esquemas. Aunque es cierto que en un sistema relacional se podría, en ciertos casos, hacer algo similar dividiendo los datos entre varias bases de datos; esto también podría llevar a consultas más complejas.

En lo que a rendimiento se refiere, con la configuración elegida del sistema relacional parece que MySQL sólo resulta ser peor frente a MongoDB. Frente a Virtuoso, el rendimiento es mejor para Virtuoso en los casos en los que la consulta requiera buscar datos de múltiples tablas, y la cantidad de datos es grande. Para evitar parte de estos problemas, una opción sería mover los datos de las valoraciones y las compras a MongoDB dejando en Virtuoso únicamente información básica de quién ha hecho qué valoración/compras/likes. De este modo el rendimiento podría mejorar debido a que MongoDB se encargaría de las consultas simples sobre valoraciones y compras; y a que la cantidad de tripletas en el almacenamiento disminuiría al eliminar información como la descripción de las valoraciones o los datos de los artículos.

Si bien, hay que destacar que se está utilizando la versión Opensource de Virtuoso y que la versión comercial podría dar resultados diferentes, llevando a mejorar éstos.

Capítulo 6

Extensión a un Entorno de Trabajo Distribuido

En este capítulo se van a tratar las diferentes herramientas y opciones disponibles en los dos sistemas de almacenamiento - MongoDB y OpenLink Virtuoso - para configurar cada uno de los sistemas deforma que sea posible replicar y/o dividir los datos entre varias instancias de los servidores.

6.1 MongoDB

En está sección se van a tratar las diferentes opciones de distribución de datos que ofrece MongoDB. En concreto se va a tratar la replicación de los datos y de la partición o "Sharding" de los datos, y también de la distribución de estos en distintas instancias de los servidores.

6.1.1 Replicación

La replicación permite mantener copias de los datos en distintos servidores - tanto locales como remotos. De esta forma si un servidor no está disponible es posible acceder a los datos a partir de las copias almacenadas en otros servidores.

En este punto se va a hablar de las herramientas que MongoDB ofrece para realizar la replicación de datos.

6.1.1.1 Introducción

MongoDB permite crear un sistema de replicación formado por un servidor primario y una serie de servidores secundarios. El servidor principal se encarga de realizar las operaciones de escritura y en general es el servidor al que acceden por defecto los usuarios – aunque es posible realizar lecturas directamente sobre los servidores secundarios. Si este servidor no está disponible los otros servidores secundarios elegirán un servidor entre todos los demás para que se convierta en principal.

Los servidores secundarios se encargan de mantener copias de datos presentes en el servidor principal, para esto los servidores secundarios se sincronizan con el servidor principal la primera vez que se conectan y a partir de ese momento, mantienen las copias realizando las mismas operaciones que se hacen sobre el servidor principal. Hay que destacar, que si existe un servidor secundario que ya está sincronizado, otros secundarios también se pueden valer de este para sincronizarse al conectarse por primera vez.

Además de los servidores primario y secundarios, MongoDB ofrece la opción de tener un tercer tipo de servidor llamado arbitro, el cual no mantiene copias de los datos pero puede votar para elegir un nuevo servidor como primario en caso de que el primario no esté disponible.

Hay que destacar que la replicación es a nivel del servidor y por tanto no es posible replicar únicamente una de las colecciones o de base de datos concretas. Es decir, se replica todo lo que se haya en el servidor.

6.1.1.2 Inicio de los servidores

MongoDB permite desplegar un sistema de replicación con servidores tanto locales como remotos. En caso de que se tengan varios servidores en una misma máquina habrá que crear una carpeta por cada instancia donde cada una de estas guardará sus datos.

Para indicar a que sistema de replicación pertenece un servidor - y no es un es servidor normal - habrá que indicar el nombre del sistema mediante la opción "replSet". Esta opción se puede incluir tanto en el archivo de configuración "mongodb.conf"; como indicar la opción al iniciar el servidor mediante "mongod". Para la primera opción es suficiente con incluir en el archivo de configuración - /etc/mongodb.conf en Ubuntu - la linea:

```
replSet = Nombre del Grupo de Replicacion
```

Una vez hecho esto es necesario iniciar la instancia del servidor, bien como servicio o mediante mongod indicando que se use el archivo de configuración mediante la opción "–config"; tal y como se vio en secciones anteriores.

Para la segunda opción es suficiente con iniciar el servidor con la siguiente opción:

```
>  mongod  --replSet Nombre del Grupo de Replicacion
```

Hay que tener en cuenta que el nombre dado a la opción "repSet" debe ser el mismo para todos los miembros de un mismo grupo.

Por otro lado, si se van a utilizar varios servidores en la misma máquina, es necesario indicar el directorio en el que ese servidor guardará los datos mediante la opción "dbpath" - siempre que no sea el directorio por defecto - ; e indicar un puerto diferente para cada servidor presente en la misma máquina mediante la opción "port". Por ejemplo si se quisieran iniciar dos servidores en la misma máquina que formen parte del mismo grupo de replicación se podría hacer lo siguiente:

```
>mongod --port 27018 --dbpath /directorio/rep0/ --replSet rep1
>mongod --port 27019 --dbpath /directorio/rep1/ --replSet rep1
```

6.1.1.3 Preparación del Sistema de Replicación

Para poder establecer la red de servidores que formarán parte del sistema de replicación será necesario indicar los servidores que formaran parte del sistema del grupo de replicación e iniciar el sistema. Por ejemplo, si se quisiera crear un sistema de tres servidores locales se podrían hacer los siguientes pasos:

1. Iniciar los servidores indicando el grupo de replicación tal y como se vio en el caso anterior:

   ```
   >mongod --port 27017 --dbpath /directorio/prin/ --replSet rep1
   >mongod --port 27018 --dbpath /directorio/rep0/ --replSet rep1
   >mongod --port 27019 --dbpath /directorio/rep1/ --replSet rep1
   ```

2. Acceder al servidor "mongod" que se quisiera poner como primario mediante el cliente "mongo", tal y como se hacía con un sólo servidor. Modificando los datos de identificación si fuera necesario.

3. Asignar a una variable los datos del grupo de replicación, indicando el nombre del grupo y las direcciones de los servidores que forman parte de él.

```
> conf= {     "_id" : "rep1",
              "version" : 4,
              "members" : [
               {"_id" : 1,"host" : "localhost:27017"},
               {"_id" : 2,"host" : "localhost:27018"},
               {"_id" : 3,"host" : "localhost:27019"}]
       }
```

4. Iniciar el sistema con los datos anteriores:

```
> rs.initiate(conf)
```

Además una vez hecho esto sería posible añadir un nuevo servidor mediante el método "rs.add(" dirección del servidor ")". Si, por otro lado se quisiera que el nuevo servidor fuera un arbitro se usará el método "rs.addArb(" dirección del servidor ")".

6.1.1.4 Elecciones

Al crear un sistema de replicación, en general el servidor en el que se realiza la configuración es el que será establecido por defecto como servidor primario - excepto si se han establecido pesos como se explicará más adelante.

Si el servidor principal se desconecta, el resto de servidores realizarán una votación para elegir a un nuevo servidor principal sobre el que se puedan hacer operaciones de escritura. Por ejemplo, para el caso anterior, suponiendo que el servidor en el puerto "27017" es el servidor principal y el resto los secundarios, el comportamiento del sistema al detener la instancia del servidor principal sería el siguiente:

1. Los dos servidores secundarios detectan que el servidor principal está desconectado:

```
[rsHealthPoll] replset info 127.0.0.1:27017 heartbeat failed, \
 retrying
[rsHealthPoll] replSet info 127.0.0.1:27017 is down (or slow to \
 respond):
[rsHealthPoll] replSet member 127.0.0.1:27017 is now in state \
 DOWN
```

2. Los servidores secundarios eligen servidores para ver quien es el nuevo servidor primario. Por ejemplo, puede ocurrir que el servidor en el puerto "27019" se elija a si mismo como candidato a servidor primario:

```
[rsMgr] replSet info electSelf 2
```

   El servidor en el puerto "27018" puede detectar la elección a candidato y decidir no elegirse a si mismo puesto que "27019" le vetaría, al tener ambos la misma prioridad.

```
[rsMgr] not electing self, localhost:27019 would veto with '\
  localhost:27018 is trying to elect itself
```

3. Los servidores votarán a un servidor entre los elegidos para pasar a ser el servidor primario. En este caso, "27018" decidiría dar su voto al otro servidor.

```
[conn51] replSet info voting yea for localhost:27019
```

   Con lo que finalmente el servidor en el puerto "27019" pasará a ser el nuevo servidor primario.

En el ejemplo anterior ambos servidores tienen las mismas posibilidades de ser elegidos como candidatos; y el hecho de que uno u otro lo sea depende principalmente de que servidor sea el más rápido en elegirse a si mismo. Sin embargo, existen varios factores que pueden hacer que un servidor sea elegido sobre otro. Por un lado, MongoDB permite asignar prioridades a los servidores. Por otro lado, en general los servidores que se encuentren en una misma red tenderán a votar a aquellos servidores de su misma rec antes que un a servidor de una red externa – siempre que la prioridad del servidor externo no sea mayor.

Asignar prioridades es especialmente importante si se quiere que un servidor vuelva a ser primario, de forma automática, al unirse a un sistema de replicación tras no haber estado disponible. Por ejemplo, en el caso anterior si el servidor que en el puerto "27017" - el servidor que inicialmente era primario – volviera a unirse al grupo, no sería elegido como servidor primario puesto que no tenía asignada una prioridad mayor que el resto. Para asignar prioridades de forma que el servidor en el puerto "27017" volviera a ser primario tras no haber estado disponible se haría lo siguiente:

1. Iniciar únicamente el servidor que se quiere que sea primario:

```
>mongod --port 27017 --dbpath /directorio/prin/ --replSet rep1
```

2. Acceder al servidor "mongod" mediante el cliente "mongo" tal y como se hace en puntos anteriores.

3. Guardar la configuración del grupo de replicación en una variable:

```
conf = rs.conf()
```

4. Asignar una prioridad de dos – por ejemplo - al servidor en el puerto "27017" . Para esto, se modifica el documento guardado en la variable anterior de la siguiente forma:

```
conf.members[0].priority = 2
```

   Suponiendo que los datos del servidor en el puerto "27017" están en la primera posición del array "members" el cual guarda la lista de los datos de los servidores que forman el grupo de replicación.

5. Reconfigurar el sistema mediante el método "reconfig(cfg)":

```
        rs.reconfig(config)
```

6. Iniciar el resto de servidores:

```
>mongod --port 27018 --dbpath /directorio/rep0/ --replSet rep1
>mongod --port 27019 --dbpath /directorio/rep1/ --replSet rep1
```

Tras asignar las prioridades, si el servidor principal se desconecta, uno de los otro dos pasará a ser el nuevo servidor principal tal y como ocurría en el caso anterior. Sin embargo, a diferencia de los que ocurría en el caso anterior, al volver a unirse al grupo, el servidor en el puerto "27017" pasará a ser de nuevo el servidor principal. Por ejemplo, el comportamiento de los servidores cuando el servidor en el puerto "27017" se conecte será el siguiente:

1. El servidor en el puerto "27017" pasa automáticamente a ser un servidor secundario y procede a sincronizarse con servidor principal.

```
        [rsSync] replSet SECONDARY
            [rsBackgroundSync] replSet syncing to: localhost:27019
```

2. El servidor en el puerto "27017" intenta ponerse como servidor principal:

```
[rsMgr] not electing self, localhost:27018 would veto with \
  '127.0.0.1:27017 is trying to elect itself but localhost:27019 \
  is already primary and more up-to-date'
```

3. El resto de servidores secundarios reciben un mensaje indicando que el servidor primario actual va a dejar de ser primario:

```
[rsMgr] stepping down localhost:27019 (priority 1), \
  127.0.0.1:27017 is priority 2 ...
```

4. El servidor principal pasa a ser secundario:

```
[conn28] replSet info stepping down as primary secs=1
[conn28] replSet relinquishing primary state
[conn28] replSet SECONDARY
```

5. Se realizan elecciones de nuevo y los dos servidores no se eligen:

```
[rsMgr] not electing self, localhost:27018 would veto with '\
  localhost:27019 has lower priority than 127.0.0.1:27017'
```

Mientras que el servidor en el puerto "27017" se elige a si mismo:

```
[rsMgr] replSet info electSelf 0
```

6. Finalmente los dos servidores de menor prioridad votan si y el servidor en el puerto "27017" es elegido servidor primario.

```
[conn27] replSet info voting yea for 127.0.0.1:27017 (0)
[rsHealthPoll] replSet member 127.0.0.1:27017 is now in state \
  PRIMARY
```

6.1.2 Sharding

El sharding consiste en repartir los datos entre varias instancias de servidores. Esto se hace cuando la cantidad de datos presentes en un sistema de almacenamiento es alta y se quiere mejorar el rendimiento repartiendo la carga de trabajo entre diferentes máquinas.

Al igual que ocurre con la replicación MongoDB ofrece una serie de herramientas que permiten repartir los datos almacenados en el sistema de forma sencilla.

6.1.2.1 Introducción

MongoDB permite realizar la división de datos a nivel de colección, es decir, que permite dividir las colecciones en trozos distribuidos entre diferentes instancias. Para realizar esto es necesario crear un "cluster" de servidores formado por: (1) un servidor router "mongos" el cual se encarga de mandar las peticiones de consulta al servidor correspondiente; (2) uno o varios servidores de configuración que mantiene información sobre que datos contiene cada servidor; y (3) una serie de "shards" - un único servidor o un grupo de replicación - que contiene las diferentes particiones.

Al hacer sharding, en general, las colecciones se dividen en varias particiones que serán distribuidas entre los distintos "shards". La partición de estas colecciones se hace en función de una clave "shard" - textitshard key - la cual generalmente corresponde a un índice o parte de un índice compuesto. La división de los datos se puede hacer de dos maneras: (1) dividiendo los datos según el rango de la clave; o (2) calculando los valores hash de la clave y dividiendo los datos en función de los valores calculados.

Una vez distribuidas las colecciones se guardan en los servidores de configuración la información sobre qué valores de la clave guarda cada "shard".

Por último, el servidor router "mongos" es el que se encarga de enviar las peticiones de consulta a los servidores correspondientes. Para esto, el servidor router solicita a los servidores de configuración la información sobre cómo se distribuyen los datos y guarda dicha información en cache. De esta forma, "mongos" enviará la solicitud al servidor/es correspondiente si un usuario solicita los datos en función del campo correspondientes a la clave "shard"; sino, la solicitud se enviará a todos los servidores.

Además, este servidor router - "mongos" - se encarga de unir los datos recibidos desde los diferentes "shards". En general devuelve los datos alternando los resultados obtenidos desde los diferentes "shrad" en el orden el que se reciben los datos; ésto siempre y cuando no se solicita que los datos estén ordenados, ni que se limite el número de resultados. Por otro lado, si se solicita que se ordenen los datos, cada "shard" ordena los resultados

antes de enviarlo a "mongos" y "mongos" hace una operación de "mergesort" sobre los datos recibidos. En el caso en el que se solicita limitar el número de resultados, tanto cada "shard" como el servidor router al recibir todos los datos, limitan el número de resultados.

La tolerancia a error del sistema viene determinada en función del servidor en el que se produce el error. Si uno de los "shard" no se encuentra disponible, el sistema sigue funcionando; sin embargo, no se podrá acceder a esa parte de los datos. Si sólo hay un servidor de configuración y este cae, el sistema sólo funcionará mientras el servidor router mantenga los datos de configuración en cache. Por último, si el servidor router cae, no será posible realizar peticiones al sistema; sin embargo, como éste no guarda ninguna información de configuración es posible desplegar otro servidor router sin problemas y que solicite la información necesaria a los servidores de configuración.

6.1.2.2 Inicio de las servidores

Para desplegar un sistema de datos distribuidos en MongoDB, como mínimo es necesario disponer de un servidor de configuración, un servidor router "mongos" y al menos dos "shards" entre los que repartir la/s colecciones de datos.

En el caso de los servidores de configuración, éstos se lanzarían como cualquier otro servidor "mongod" con la diferencia de que hay que indicar que es un servidor de configuaración mediante la opción "configsvr". Esta opción se puede indicar bien modificando el archivo "mongo.conf" o bien al lanzar el servidor indicar la opción "–configsvr". Por ejemplo, si se utiliza esta segunda opción para lanzar el servidor habría que hacer lo siguiente:

```
> mongod --configsvr
```

Además de esto, si se utiliza el directorio por defecto donde MongoDB guarda los datos es necesario crear una carpeta de nombre "configdb" dentro de dicha carpeta. Alternativamente se puede indicar un directorio diferente mediante la opción "–dbpath" tal y como se ha indicado en puntos anteriores.

Por otro lado, en el caso del servidor router no es necesario crear ningún directorio ya que este servidor no guarda ningún tipo de datos. Por tanto, para lanzar el servidor únicamente es necesario ejecutar el comando "mongos" desde el terminal - en caso de que haya otro servidor en la misma máquina también habría que indicar un puerto de escucha que esté libre.

Por último, los shard se despliegan como cualquier instancia "mongod" si cada uno esta formado por un servidor; y tal y como se explico en la sección anterior si están formados por un grupo de replicación.

6.1.2.3 Preparación del sistema

Una vez lanzados todos los servidores el siguiente paso es añadir los "shards" al sistema e indicar que colecciones se quieren dividir entre los diferentes "shards".

Es posible añadir los servidores que harán las veces de "shard" se mediante el método "sh.addShard("direccion:puerto")". Para esto primero es necesario conectarse al servidor router "mongos" mediante el cliente tal y como se haría al conectarse a un servidor normal. Por ejemplo, suponiendo que hay un servidor router en el puerto "27019", se conectaría de la siguiente forma:

```
> mongo --port 27019
```

Una vez conectado al servidor, se añadiría los "shard" mediante el método anterior. Por ejemplo, si se han desplegado dos servidores en la misma máquina sobre el puerto "27020" y "27021", dichos servidores se añadirían indicando la dirección del servidor y el puerto de la siguiente forma:

```
>sh.addShard("127.0.0.1:27021")
>sh.addShard("127.0.0.1:27021")
```

Por otro lado si lo que se quisiera añadir fuera un grupo de replicación, habría que indicar el nombre del grupo y las direcciones de cada uno de los servidores que forman parte del grupo. Por ejemplo, suponiendo que se quiere añadir como "shard" un grupo de replicación de nombre "rep1" formado por los dos servidores anteriores, en este caso se añadiría el nuevo "shard" de ls siguiente forma:

```
>sh.addShard("rep1/127.0.0.1:27021,127.0.0.1:27022")
```

Por ultimo, una vez añadidos los "shard" es necesario indicar queé colección se quiere repartir entre los diferentes "shard". Para esto hay que hacer dos pasos:

1. Habilitar la partición en la base de datos que contenga las colecciones a dividir mediante el método "sh.enableSharding("base de datos")". Por ejemplo, si uno

de los "shards" contiene una base de datos llamada productos, desde un cliente conectado al servidor router "mongos" esto se haría de la siguiente forma:

```
>sh.enableSharding("productos")
```

2. Indicar que colecciones se van a dividir mediante el método "sh.shardCollection()". Habrá que indicar tanto el nombre de la colección como en el de la base de datos que contiene dicha colección. Además, también se deberá decidir que clave se utiliza para realizar la partición de los datos. Por ejemplo, suponiendo que en la base de datos "productos" hay una copia de la colección "articulos" y se quiere usar como clave de "shard" el título y el tipo de los artículos; la partición de la colección se haría de la siguiente manera:

```
> sh.shardCollection("productos.articulos",{"titulo":1,"tipo":1})
```

Con esto, el servidor router "mongos" abilita la partición entre los "shards", calcula el número trozos "chunks" que se van a crear y comienza a mover los trozos de un "shard" a otro.

Por otro lado, si se quiere hacer una partición mediante claves "hash", en lugar de poner un "uno" al indicar la clave, habría que poner ""hashed". Por ejemplo, para el caso anterior ésto se haría de la siguiente manera:

```
> sh.shardCollection("productos.articulos {"titulo":"hashed","tipo":"\
  hashed"})
```

6.2 OpenLink Virtuoso

En esta sección se va a hablar de las herramientas que Virtuoso ofrece para realizar la replicación y partición de datos. Hay que destacar que aunque la versión comercial trae opciones de replicación y sharding; la versión OpenSource de Virtuoso no ofrece estas opciones por tanto en esta sección se va a tratar de ver como realizar esta tareas sin las herramientas de la versión comercial.

6.2.1 Replicación

La replicación permite mantener copias de los datos en distintos servidores - tanto locales como remotos. De esta forma si un servidor no está disponible es posible acceder a los datos a partir de las copias almacenadas en dichos servidores.

Puesto que Virtuoso es un sistema de almacenamiento híbrido, éste dispone de herramientas para realizar replicación tanto sobre las bases de datos relacionales como el sistema de almacenamiento RDF. En este punto, sin embargo, se va a hablar únicamente de las herramientas disponibles para realizar replicación de los grafos RDF.

6.2.1.1 Introducción

Virtuoso permite crear un sistema de replicación sobre los grafos de forma que es posible realizar copias de estos, guardarlas en servidores tanto remotos como locales y mantener los datos sincronizados con los cambios que se hagan en la copia principal.

Un sistema simple de replicación en Virtuoso consta de un servidor principal y de varios servidores secundarios. El servidor principal es el encargado de "publicar" los datos, es decir, es el servidor sobre el que los usuarios realizan las diferentes operaciones sobre los datos. Los servidores secundarios, por otro lado se "subscriben" al servidor principal para mantener una copia de los datos publicados por el servidor principal.

Hay que destacar que si el grafo a replicar contiene datos antes de ser publicado, la replicación de estos datos hay que hacerla de forma manual para todos servidores, es decir no se hace un volcado de los datos automático al publicar por primera vez un grafo que contenía datos.

Virtuoso permite, además, tener un servidor a parte que guarde una copia sincronizada de los datos del servidor principal, pero que además puede publicar los datos para que otros servidores secundarios se sincronicen con éste, en lugar de directamente con el primario.

La replicación de los datos es a nivel de grafo, con lo que no es necesario replicar todo el almacenamiento de RDF. Además, de replicar los datos de los grafos, Virtuoso también permite que otros datos como son los "logs" de replicación.

6.2.1.2 Inicio de los Servidores

En el caso de Virtuoso el inicio de los servidores requiere únicamente que se modifiquen algunos parámetros del archivo de configuración "virtuoso.ini".

En este caso habrá que realizar dos cambios:

1. Añadir tres parámetros nuevos que permiten configurar las varias opciones sobre la automatización de la replicación:

```
[Parameters]
SchedulerInterval    = 1     ; run the internal scheduler every \
 minute
CheckpointAuditTrail = 1     ; enable audit trail on transaction \
 logs
CheckpointInterval   = 60    ; perform an automated checkpoint \
 every 60 minutes
```

2. Cambiar el nombre del servidor - en el parámetro "ServerName" - para que todos los servidores que forman parte del sistema de replicación tengan un nombre diferente. Por ejemplo, dando al servidor principal en nombre "Prim".

Esto es necesario hacerlo tanto para el servidor principal como para los secundarios.

Por otro lado, también es necesario configurar el driver ODBC - o iODBC en el caso de Linux/Unix/MacOS – para configurar los datos de acceso al servidor principal. Esta configuración solo es necesario hacerla sobre los servidores que se vayan a conectar al servidor principal. En el caso de iODBC para configurar dicho driver, es necesario modificar el archivo "odbc.ini" - en Ubuntu se encuentra en el directorio /etc - e incluir las siguientes opciones junto a la configuración por defecto:

```
[ODBC Data Sources]
 Nombre DNS del Servidor = OpenLink Virtuoso

[Nombre DNS del Servidor]
 Driver = OpenLink Virtuoso
  Address = ip:puerto
```

El nombre del servidor puede ser – o no - el mismo nombre que se le dio al servidor en el archivo de configuración "virtuoso.ini".

Este último paso es necesario, ya que durante la configuración de los servidores secundarios se usa dicho driver para conectarse al servidor principal mediante el nombre que se le da al servidor.

6.2.1.3 Preparación del Sistema de Replicación

A la hora de configurar el sistema de replicación existen dos opciones: (1) configurar el sistema desde la interfaz web; o (2) configurarlo mediante las funciones predefinidas del sistema. En este caso se va a indicar como hacerlo desde la linea de comandos ya que resulta más sencillo.

Para desplegar el sistema de replicación lo primero que hay que hacer, una vez que se han realizado las configuraciones indicadas en el punto anterior, es el habilitar la replicación e indicar que grafos se quieren replicar. Para esto, habría que hacer los siguientes pasos:

1. Conectarse al servidor principal con el cliente "isql" desde el terminal o la interfaz web, tal y como se ha indicado en secciones anteriores. Por ejemplo, si el servidor principal se encuentra en la máquina local, suponiendo que nos encontramos en la carpeta que contiene el cliente "isql":

```
> ./isql -U dba -P tab053
```

2. Habilitar la replicación de datos RDF mediante la función "RDF_REPL_START()":

```
>DB.DBA.RDF_REPL_START();
```

3. Indicar que grafo se quiere replicar mediante la función "RDF_REPL_GRAPH_INS()". Por ejemplo, suponiendo que se quiere replicar un grafo de nombre "http://prueba.com":

```
>DB.DBA.RDF_REPL_GRAPH_INS ('http://prueba.com');
```

Con estos pasos, el servidor principal ya esta configurado para publicar los datos. El siguiente paso es el de configurar los servidores secundarios que mantendrán copias de los grafos. Para esto es necesario seguir los siguientes pasos para cada uno de los servidores secundarios:

1. Conectarse a un servidor secundario con el cliente "isql" desde el terminal o la interfaz web.

2. Conectar con el servidor principal mediante la función "repl_server"(). Ésta tiene tres parámetros en los que hay que indicar: (1) el nombre del servidor principal

dentro del grupo de replicación; (2) el nombre DNS que se le dio al servidor en el archivo "odbc.ini"; (3) la dirección donde se encuentra el servidor principal, indicando la ip y el puerto.

```
> repl_server ('Prim', 'Nombre DNS del Servidor', 'ip:puerto');
```

3. Suscribirse a las publicaciones del servidor principal mediante la función "repl_subscribe()". Esta función contiene seis parámetros: (1) el nombre del servidor; (2) el nombre de la publicación que por defecto es "__rdf_repl"; (3) el nombre de una cuenta con acceso a "WebDAV" del servidor secundario como es el caso de la cuenta "dav" creada durante la instalación de virtuoso; (4) el nombre de un grupo que sea dueño de "WebDAV" del servidor secundario, el grupo por defecto también tendrá el nombre "dav"; (5) el nombre de un usuario con acceso al servidor principal; (6) la contraseña del usuario con acceso al servidor principal.

```
> repl_subscribe ('Prim', '__rdf_repl', 'dav', 'dav', 'dba', '\
  tab053');
```

4. Resincronizar todo de forma manual para replicar los datos que pueda haber previamente en el grafo, mediante la función "repl_sync_all();".

```
> repl_sync_all();
```

5. Programar el sistema para que se sincronice con el sistema periódicamente. En este caso además de sincronizarse cada vez que hay una escritura en el sistema principal, también se resincroniza cada cierto tiempo. Ésto se hace mediante la función "SUB_SCHEDULE()" la cual tiene tres parámetros: (1) el nombre del servidor principal; (2) el nombre de la publicación a sincronizar; (2) el intervalo de tiempo en el que se quiere hacer la sincronización – en minutos.

```
> DB.DBA.SUB_SCHEDULE ('Nombre del servidor', '__rdf_repl', 1);
```

Por último, como se comento en un punto anterior es posible tener servidores que sean secundarios y que al mismo tiempo publiquen grafos permitiendo a otros servidores secundarios suscribirse a ellos en lugar de al servidor principal.

6.2.1.4 Comportamiento

En el caso de la replicación en Virtuoso no hay implementado por defecto un sistema de elecciones. Por tanto si un servidor principal cae no se elige ninguno de los servidores secundarios para que ocupe el lugar del servidor principal.

Sin embargo, a pesar de que el servidor principal cayera seguiría siendo perfectamente posible acceder a los datos mediante los servidores secundarios. En este caso sería posible tanto leer como escribir en el servidor secundario, sin embargo, cualquier datos que se escribiera en dicho servidor no se extendería al resto de los servidores secundarios a no ser que éste estuviera publicando el grafo y el resto de servidores secundarios estuviese subscrito a este servidor.

6.2.2 Sharding

El sharding consiste en repartir los datos entre varias instancias de servidores. Esto se hace cuando la cantidad de datos presentes en un sistema de almacenamiento es alta y se quiere mejorar el rendimiento repartiendo la carga de trabajo entre diferentes máquina.

En el caso de Virtuoso la repartición de datos entre varios nodos se hace a nivel de todo el sistema de almacenamiento, es decir, que a diferencia de lo que ocurría con la replicación no se puede hacer únicamente sobre el almacenamiento de tripletas – o cuartetos - sino se hace también sobre el sistema relacional.

6.2.2.1 Introducción

En Virtuoso el "sharding" se realiza a nivel de tabla de forma que se parten las tablas en varios trozos y estas se reparten entre varias instancias. En lo que a los datos RDF se refiere, puesto que tal y como se comento anteriormente están almacenados en varias tablas, las particiones se hacen de forma similar a como se hace con el almacenamiento relacional de Virtuoso.

En general el conjunto de servidores que forman el "cluster" no está formado por servidores especializados sino que están formados por el mismo tipo de servidor que el servidor básico del que se ha hablado en el resto de la documentación. Es decir, que a diferencia de MongodDB donde el sistema está formado por servidores de tres tipos – shard, router y configuración - en el caso de Virtuoso sólo hay un único tipo de servidor. Hay que destacar que uno de los servidores que forma parte del "cluster" deberá hacer las veces de Maestro para tratar problemas del tipo "deadlock" o bloqueo mutuo. Sin embargo, en

general el resto de las operaciones se pueden realizar sobre cualquiera de los servidores que forma el "cluster".

A la hora de realizar la partición de los datos, estás se hacen en función de los índices de las tablas; en lugar de utilizar una clave a parte, tal y como ocurría con MongoDB. A partir de los valores de los índices Virtuoso generará una clave "hash" y repartirá los datos entre los servidores en función de dicha clave. Hay que tener en cuenta, que para que una tabla sea repartida entre los diferentes servidores es necesario que todos sus índices estén partidos. Si no se utiliza un "cluster" la partición de los datos no tiene efecto.

En el caso de los datos RDF, tal y como se vio, las tablas donde se guardan los datos contienen una serie de índices en los que se ha indicado durante la creación que por defecto se pueden partir los índices. La partición se hace en función de una clave hash de forma que la distribución de las tripletas entre los diferentes servidores es equilibrada. Además, puesto que los bits de menor peso no se usan para la partición, tripletas consecutivas estarán en el mismo servidor.

No todas las tablas del sistema tienen que estar repartidas entre los distintos servidores, de hecho es posible que se den tres tipos de tablas: (1) partidas, que son las que se dividen entre los diferentes servidores, (2) replicadas, las cuales están repetidas en todos los servidores y contienen siempre los mismo datos; y (3) locales las cuales están repetidas en todos los servidores pero no contienen los mismo datos.

En lo que se refiere a la realización de operaciones, la ejecución de ciertas operaciones de administración de los servidores es a nivel de cada servidor, pero existen una series de operaciones como es el caso "cl_exec" que permite que un comando se ejecute al mismo tiempo sobre todos los servidores. En cuanto a las operaciones sobre los datos, y más concretamente las consultas "SPARQL", estás se ejecutan de forma transparente independientemente desde el servidor en el que se realicen, es decir, el usuario puede realizar una consulta sobre cualquiera de los servidores y recibirá los datos que contiene todo el "cluster".

6.2.2.2 Preparación del sistema

Los servidores que forman parte del "cluster" se inician de forma similar a como se inicia un servidor normal , puesto que no son servidores especiales tal y como ocurría en MongoDB. La principal diferencia es que en este caso es necesario configurar un archivo llamado "cluster.ini" donde se incluyen los datos de configuración.

Este fichero no existe por defecto y es necesario crearlo con la información que se índica en la documentación de Virtuoso. Dicho archivo se debe guardar en la misma carpeta en la que está contenido el archivo virtuoso.ini, que en el caso de la versión comercial de Virtuoso se encuentra en la carpeta "database" del directorio en el que se ha instalado el programa.

En lo que se refiere a la configuración del archivo, los campos más importantes es el de "Master" el cual permite configurar el servidor que actuará como servidor principal; "ThisHost" para indicar cual es la dirección de servidor que contiene dicho fichero y "Threads" que permite indicar cuantas conexiones desde otros servidores se pueden tener activas, independientemente de las conexiones que los usuarios hagan directamente a esa instancia. Por ejemplo, si se tienen dos instancias en un mismo servidor, una en el puerto 2221 y otra en el puerto 2222, la configuración del archivo en el servidor con puerto "2222" suponiendo que la otra instancia es el maestro, sería la siguiente:

```
[Cluster]
Threads = 100
ThisHost = Host2
Master = Host1
ReqBatchSize = 100
BatchesPerRPC = 4
BatchBufferBytes = 20000
LocalOnly = 2

Host1 = 127.0.0.1:2221
Host2 = 127.0.0.1:2222
```

Ahora, una vez configurados los archivos para las dos instancias es suficiente iniciar cada una de las instancias. Hay que tener en cuenta que si se utilizan dos instancias en la misma máquina es necesario modificar los archivos de configuración virtuoso.ini de cada instancia, de forma que cada una use puertos diferentes.

Una vez iniciadas las instancias hay que crea el "cluster" con los servidores que lo forman. Para esto se utiliza la sentencia "CREATE CLUSTER". Mediante esta sentencia es posible indicar si se quiere que los servidores que forman el "cluster" replique los datos de las tablas con índices partidos o si se quiere que los servidores se repartan las filas de estas tablas. Otra opción que también es posible indicar es si se quiere que el "cluster" sea elástico o no - por defecto -; es decir, que una vez creado el "cluster" sea posible añadir o eliminar servidores al/del sistema.

Por ejemplo, si se quiere crear un "cluster" no elástico, que divida las tablas y formado por las dos instancias anteriores la sentencia sería la siguiente:

```
> create cluster Shard default group ("Host1"), group ("Host2");
```

En este caso se indica que el servidor no es elástico mediante la opción "default". Por otro lado, se índica que las tablas se reparten entre las dos instancias porque el "cluster" está formado por dos grupos, donde cada uno contiene una única instancia. Si por el contrario se hubiera escrito un único grupo con los dos servidores - group ("Host1" , "Host2") – se estaría indicando que las tablas se replican, donde los dos servidores tendrán copias exactas de las tablas.

Por otro lado, también es posible combinar las dos opciones anteriores para crear un "cluster" más complejo. Por ejemplo para el caso anterior además de la partición entre las dos instancias se quiere usar un tercer servidor - "Host3" - para replicar los datas de la primera instancia; la sentencia sería la siguiente:

```
> create cluster Shard default group ("Host1","Host3"), group ("Host2\
");
```

Por último, si se quiere que el servidor sea elástico, se utiliza la opción "__elastic " en lugar de "default" e se indica el número máximo de instancias permitidas y la cantidad de segmentos inicial para cada grupo. Por ejemplo, para crear un "cluster" elástico con las dos instancias iniciales se haría lo siguiente:

```
> create cluster Shard __elastic 2048 4 group ("Host1"), group ("\
Host2");
```

Además, previamente a ejecutar la sentencia es necesario modificar el archivo "cluster.ini" para indicar los segmentos en los que se divide cada instancia, indicando el directorio donde se guardan los datos.

```
[ELASTIC]
Segment1 = 1024, ./db1/d1.db = q1
Segment2 = 1024, ./db2/d2.db = q2
Segment3 = 1024, ./db3/d3.db = q3
Segment1 = 1024, ./db4/d4.db = q4
```

6.2.2.3 Partición de los índices

A la hora de repartir la tablas entre los diferentes nodos, se utilizan los índices de dichas tablas como clave para repartir las filas. Para poder utilizar los índices para dividir las tablas es necesario indicar que se quiere partir el índice al crearlo o bien al alterarlo. Para esto, durante la sentencia "CREATE INDEX" o "ALTER INDEX" se debe indicar la opción "PARTITION" e indicar opcionalmente el nombre del "cluster" que usará el índice para partir los datos.

Tomando como ejemplo uno de los índices creados por defecto en Virtuoso para las tablas que contienen la información de las tripletas; el índice "GS" se crearía de la siguiente manera:

```
CREATE DISTINCT NO PRIMARY KEY REF BITMAP INDEX RDF_QUAD_GS
 ON RDF_QUAD (G, S)
 PARTITION (S INT (0hexffff00));
```

Como se puede ver se utiliza la opción "Partition" sin indicar el nombre de ningún "cluster" por lo que será usado por todos. Junto con esta opción se indican una serie de parámetros, en concreto se indican tres cosas: (1) la columna que se usa como clave para partir las tablas, en este caso es la columna "S"; (2) el tipo de datos de la columna, que este caso es un entero; y (3) sin es un entero una máscara para indicar que bits se usarán para crear un hash de la tabla; y si es una cadena de caracteres - varchar - un valor numérico para indicar que caracteres usar para calcular la clave .

En el caso del entero, la máscara permite que al codificar claves consecutivas con por ejemplo valores como 11 - "0hex000011" - y 12 - "0hex000012" -, la clave hash obtenida sea la misma para los dos valores y por tanto valores consecutivos irán a parar al mismo servidor. Esta máscara se puede modificar para permitir el uso de más o menos bits.

Para las cadenas de caracteres, el valor numérico puede ser negativo o positivo, de forma que si el valor es negativo coge todos los caracteres excepto los n últimos, mientras que si es positivo coge los n primeros.

Capítulo 7

Conclusiones y Líneas Futuras

En este capítulo se presentan las conclusiones a las que se ha llegado tras realizar el proyecto, así como la valoración personal al finalizar el proyecto. Por último se hace un breve comentario sobre las posibles vías que se podrían seguir en el futuro gracias a las ideas planteadas y el conocimiento adquirido a lo largo del proyecto.

7.1 Conclusiones

A lo largo del proyecto se han podido estudiar las diversas necesidades que surgen al desarrollar una aplicación de naturaleza políglota con sistemas NoSQL, tanto a nivel de diseño e implementación, como de aprendizaje - en especial para las tecnologías NoSQL.

A la hora de diseñar la aplicación una de las partes más complicadas es la distribución y modelado de datos. Hay que tener especial cuidado al repartir los datos entre los sistemas para utilizar el sistema más adecuado con lo que no es sólo necesario estudiar las características de los datos sino también de los sistemas a utilizar; y una vez hecho esto modelar los datos acorde no solo al tipo de sistema sino también a las peculiaridades de rendimiento que pueda tener un sistema concreto. En lo que se refiere al diseño de la arquitectura general se ha visto que existen diferentes opciones y que cada una de ellas tiene asociadas desventajas y beneficios, y que en general vienen determinadas por las características que se priorizan en la aplicación - por ejemplo que el uso de varias bases de datos sea transparente para la parte central de la aplicación.

En lo que se refiere a la implementación, una vez hecho un diseño correcto, la implementación de la mayor parte de la aplicación es trivial. Sin embargo, la implementación de las consultas puede ser complicada por varias razones: (1) es necesario conocer los lenguajes de consulta de cada sistema; (2) es necesario conocer las diferentes formas en

las que se puede hacer una consulta en un sistemas; y (3) es necesario conocer a un nivel de aplicación el funcionamiento de los drivers.

Como se puede ver el principal problema de este tipo de aplicaciones - y del que derivan la mayoría de los problemas - es el hecho de que es necesario tener un amplio conocimiento de todos los sistemas que van a ser utilizados y por ello es necesario emplear un cierto tiempo en el aprendizaje de todos los sistemas usados. Si bien es cierto que es posible implementar una aplicación con un conocimiento básico de los sistemas; desconocer ciertos detalles puede hacer que los beneficios de usar estos sistemas desaparezcan. Un ejemplo claro de esto es lo que se explicaba en el tema de comparación en lo referente al paso de datos de los objetos del sistema a los objetos de la aplicación.

Teniendo todo esto en cuenta la pregunta que hay que hacerse es ¿merece la pena hacer una *Aplicación de Persistencia Políglota*? La respuesta es que depende. En una aplicación simple y con poca carga de datos, la realidad es que un único sistema es más que suficiente. Sin embargo, en una aplicación con muchos datos de diferente tipo y con diferentes necesidades se puede sacar provecho del uso de varios sistemas frente a uno solo donde el rendimiento baje o la complejidad del esquema de datos desborde al desarrollador y a los administradores.

7.2 Líneas Futuras

A la hora de decidir por qué línea expandir este proyecto existen varias opciones.

Una opción sería expandir la aplicación en si, dando más funcionalidades o crear nuevas aplicaciones que hiciesen uso de la librería de servicios. Por ejemplo, sería posible crear una aplicación de administración del comercio que use la librería o bien extender los servicios para que una aplicación se conecte de forma remota.

En lo que se refiere al uso de varios sistemas, otra opción sería añadir un nuevo sistema de otro tipo - clave-valor o familia de columna - para así tener un conocimiento general de la mayoría de los tipos de bases de datos NoSQL.

Sin embargo, hay otra opción que podría ser interesante. Uno de los principales problemas que se plantean a la hora de desarrollar este tipo de aplicaciones es que en general es necesario: por un lado, conocer las características de los distintos tipos de bases de datos - documentos, grafos, clave/valor o familia de columnas -; por otro lado, además es necesario conocer las peculiaridades de los sistemas de almacenamiento concretos de los que se va a hacer uso. Si bien es cierto que esto último también se podría decir de los sistemas relacionales, la mayoría de los sistemas usan SQL para permitir hacer

consultas sobre los datos; y en general entre los sistemas relacionales hay un mayor nivel de estandarización que en los sistemas NoSQL. Un ejemplo de esto es el hecho de que sistemas de almacenamiento como MongoDB y CouchDB, donde a pesar de que ambos almacenan documentos, la forma en la que se realizan las consultas es muy diferente.

Para solucionar este problema una opción sería seguir la idea de sistema híbrido que tiene OpenLink Virtuoso, donde se implementasen varios tipos de almacenamiento sobre un único sistema. En este caso la idea sería prescindir del servidor Web y de Aplicaciones, y desarrollar un sistema que implementase tres tipos de bases de datos NoSQL - sin clave/valor - junto a una base de datos relacional básica. Si bien, un sistema como éste sería complicado de desarrollar y diseñar, una primera idea básica de cómo podría ser la aplicación sería por ejemplo: (1) desarrollar en el nivel más bajo cada uno de los sistemas por separado, (2) en un nivel intermedio desarrollar una serie de métodos que permitieran solicitar datos a cada uno de los sistemas, (3) en un nivel superior implementar una API que permitieran a los usuarios comunicarse con los diferentes sistemas mediante un lenguaje estándar. La idea de esta última API es que los métodos a disposición de usuarios fueran los más generales posibles para que aunque fuera necesario indicar el tipo de base de datos que se quisiera usar, hubiera una cierta generalización en el acceso a los datos. Sin embargo, este tipo de sistema es muy complejo y sigue habiendo otros aspectos del sistema que habría que diseñar.

7.3 Valoración Personal

En lo personal este proyecto ha resultado complejo, no solo desde un punto de vista del desarrollo de la aplicación; sino desde el punto de vista de la realización de un proyecto de este tamaño y en especial de estas características tanto en lo que se refiere a la formación de los sistemas de almacenamiento y el diseño de la aplicación; y al enfoque que se le ha dado al proyecto realizado.

Este proyecto me ha ayudado a conocer el modo en que trabajan los sistemas de bases de datos NoSQL; y a formarme ampliamente en el uso de los sistemas MongoDB y Virtuoso.

También me ha ayudado a conocer arquitecturas de aplicaciones que van más allá de las dadas durante la carrera- Además, esto me ha permitido ver que algunas de los decisiones de diseño tomadas en la aplicación de persistencia políglota pueden beneficiar al desarrollo de aplicaciones con un único sistema.

Apéndice A

Manual de Instalación

En este apéndice se explican los pasos que es necesario seguir para desplegar la *Aplicación de Persistencia Políglota* en un servidor Tomcat sobre Ubuntu 12.04LTS.

A.1 Requisitos

Para desplegar la aplicación es necesario instalar y configurar las siguientes tecnologías sobre Ubuntu 12.4LTS:

- Oracle Java (JDK) 7
- Servidor Web Apache Tomcat 7
- Sistema de Bases de Datos MongoDB 2.4
- Sistema de Bases de Datos OpenLink Virtuoso 6
- Servidor Web XAMPP 1.8.3 con el Sistema de Bases de Datos MySQL

A.2 MongoDB

En esta sección se va a explicar de forma básica como instalar y configurar MongoDB para ser usado por la aplicación creada. Una explicación más extensa de la opciones de instalación y configuración se puede encontrar en el documento *Diseño de un entorno de trabajo para MongoDB y Openlink Virtuoso*.

A.2.1 Instalación de MongoDB

Aunque es posible instalar MongoDB desde el repositorio de Ubuntu, es posible que la versión disponible sea una versión antigua del sistema. Por esto, se recomienda seguir los siguientes pasos:

1. Importar la clave GPG firmada de 10gen necesaria para descargar desde apt los paquetes del repositorio de 10gen:

```
>sudo apt-key adv --keyserver keyserver.ubuntu.com --recv 7F0CEB10
```

2. Cear una lista con la dirección del repositorio de 10gen:

```
>echo 'deb
http://downloads-distro.MongoDB.org/repo/ubuntu-upstart dist 10gen'
| sudo tee /etc/apt/sources.list.d/10gen.list
```

3. Recargar el repositorio en apt:

```
>sudo apt-get update
```

4. Instalar el paquete que instalará la última versión estable:

```
>sudo apt-get install MongoDB-10gen
```

A.2.2 Configuración de usuarios

Puesto que por defecto MongoDB no crea ningún usuario, es necesario crear un usuario de la siguiente manera:

1. Entrar en el cliente mongo:

```
> mongo
```

2. Concetarse a la base de datos *contenidos* - no es necesario crear la base de datos manualme:

```
> use contenidos
```

3. Crear el usuario que usara la aplicación de nombre *appl* y contraseña *tab053*

```
> db.addUser({user: "app",
              pwd: "tab053",
              roles:["readWrite", "dbAdmin"]
            })
```

4. Activar el control de usuarios, para esto hay que hacer dos pasos:

   (a) Modificar el archivo de configuración *mongodb.conf* - en el directorio */etc/-mongodb.conf* - y des-comentar la linea *auth = true*.

   (b) Reiniciar la instancia de MongoDB:

```
> sudo service mongodb restart
```

A.3 Openlink Virtuoso

En esta sección se va a explicar de forma básica como instalar y configurar Virtuoso para ser usado por la aplicación creada. Una explicación más extensa de la opciones de instalación y configuración se puede encontrar en el documento *Diseño de un entorno de trabajo para MongoDB y Openlink Virtuoso*.

A.3.1 Instalación

En este caso la instalación se hará directamente desde el repositorio ya que aunque la versión disponible no sea la última, la complicación de la instalación manual resulta excesiva en comparación con las mejoras que trae.

Por tanto para instalar el sistema de almacenamiento es necesario realizar lo siguiente:

```
sudo apt-get install virtuoso-opensource
```

Durante la instalación se solicitará la contraseña para el usuario *dba* y *dav*, se introduzcirá las contraseñas que se deseen.

A.3.2 Configuración

Una vez instalado el sistema es necesario crear el usuario que usará la aplicación.

La forma más sencilla es hacerlo desde la interfaz web del sistema, siguiendo los siguientes pasos:

1. Entrar en el cliente web. Desde un navegador ir a la dirección *http://localhost:8890/conductor/*.

2. En el menú lateral introducir los datos de conexión para el usuario *dba*.

FIGURA A.1: Identificarse en la interfaz web de Virtuoso

3. Desde el menú superior ir a *System Admin -> User Accounts* y hacer click en *Create New Account*.

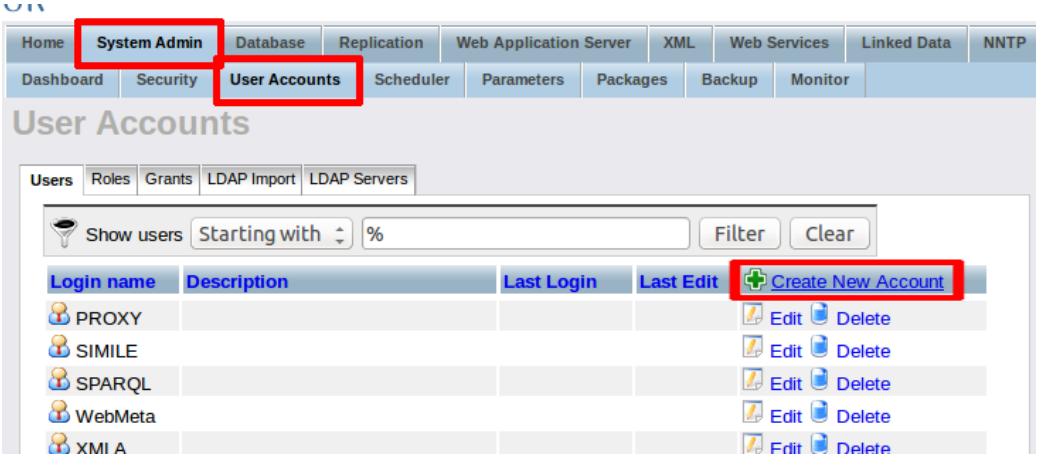

FIGURA A.2: Crear un usuario (I)

4. Rellenar el formulario con el nombre de usuario *appl* y la contraseña *tab053*; asignarle los roles relacionados *SPARQL* y pulsar *save*.

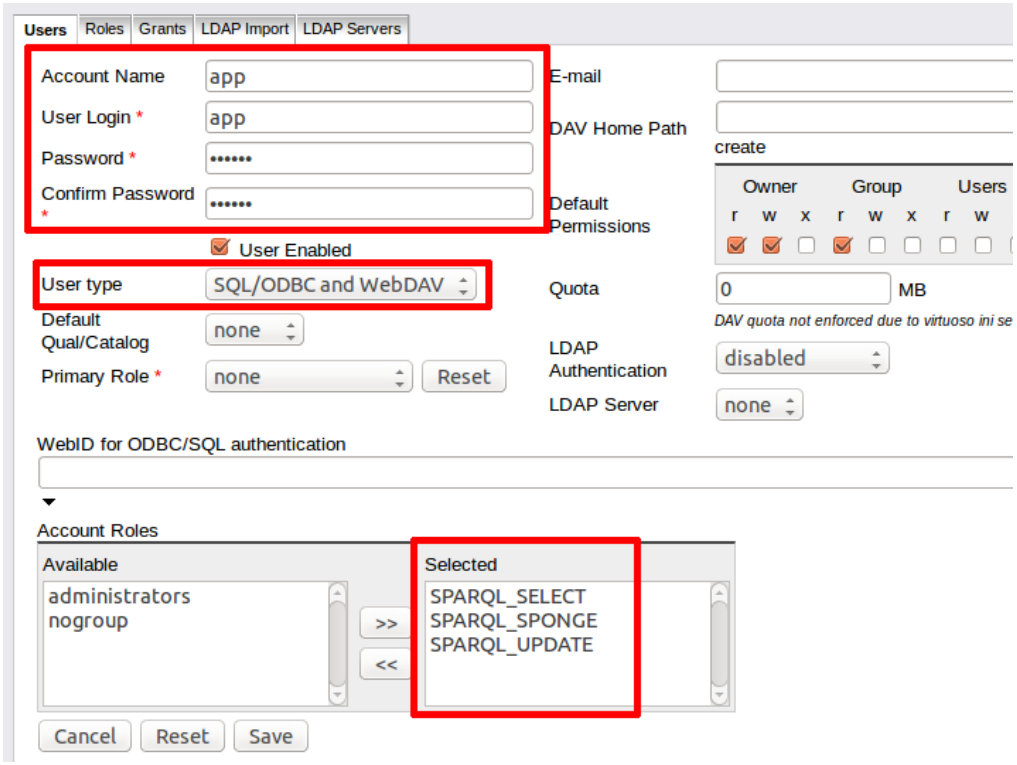

FIGURA A.3: Crear un usuario (II)

A.4 MySQL

En esta sección se va a explicar de forma básica como instalar y configurar MySQL para ser usado por la aplicación creada. Puesto que al instalar el servidor web Xampp también se instalar MySQL, se ha elegido intalar este en lugar MySQL por separado ya que además incluye la herramienta *phpMyAdmin*; que permite administrar el sistema desde una interfaz web.

A.4.1 Instalación de Xampp

Para instalar la versión 1.8.3 de 64 bits se pueden seguir los siguientes pasos:

1. Descargar la versión deseada desde el terminal:

```
> wget -O xampp-linux-x64-1.8.3-1-installer.run
  http://www.apachefriends.org/download.php?xampp-linux-x64
  -1.8.3-1-installer.run
```

2. Pasar a superusuario.

```
> sudo su
```

3. Cambiar los privilegios del archivo de ejecución *.run*.

```
> chmod 755 download.php\?xampp-linux-x64-1.8.3-1-installer.run
```

4. Ejecutar el archivo anterior:

```
> ./download.php\?xampp-linux-x64-1.8.3-1-installer.run
```

5. Pulsar el botón *next* de la interfaz de instalación hasta que comience a instalarse.

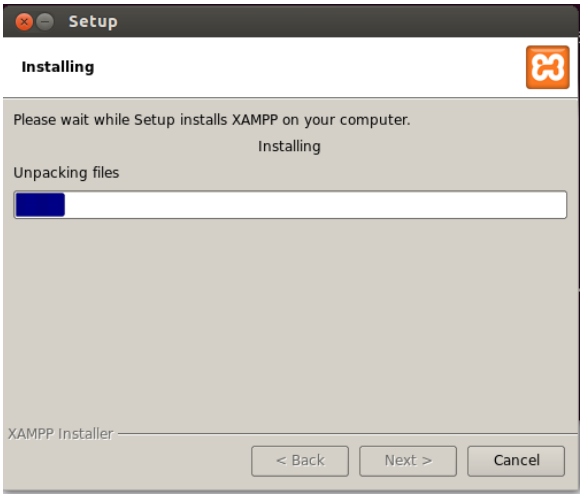

FIGURA A.4: Instalar Xampp

6. Pulsar *finalizar*.

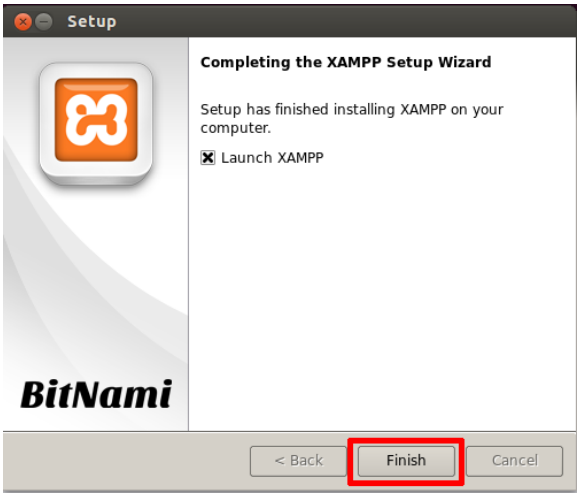

FIGURA A.5: Finalizar Instalación Xampp

7. Finalmente para inciar MySQL - si no se incia por defecto - desde la linea de comandos indicar lo siguiente - fuera del superusuario:

```
> sudo /opt/lampp/lampp start
```

A.4.2 Configurar de MySQL

A continuación se muestran los pasos necesarios para configurar MySQL para el uso con la *Aplicación de Persistencia Políglota*. En este caso además de crear un usuario será necesario crear la base de datos y una tabla para el inventario, ya que a diferencia de los sistemas anteriores MySQL no crea tablas automáticamente al querer insertar en una tabla que no existe.

1. Acceder a phpMyAdmin: ir desde un navegador web a la página principal de Xampp en la dirección *http://localhost*. Desde ahí hacer click en el enlace *phpMyAdmin* del menú lateral - si aparece un página sin menú lateral pero con varios enlaces a idiomas hacer click en español.

FIGURA A.6: Acceder a phpMyAdmin

2. Crear una base de datos: desde el menú superior seleccionar *Databases*. En la nueva ventana debajo del label *Create database*, insertar el nombre de la base de datos *appl* y pulsar el botón *create*.

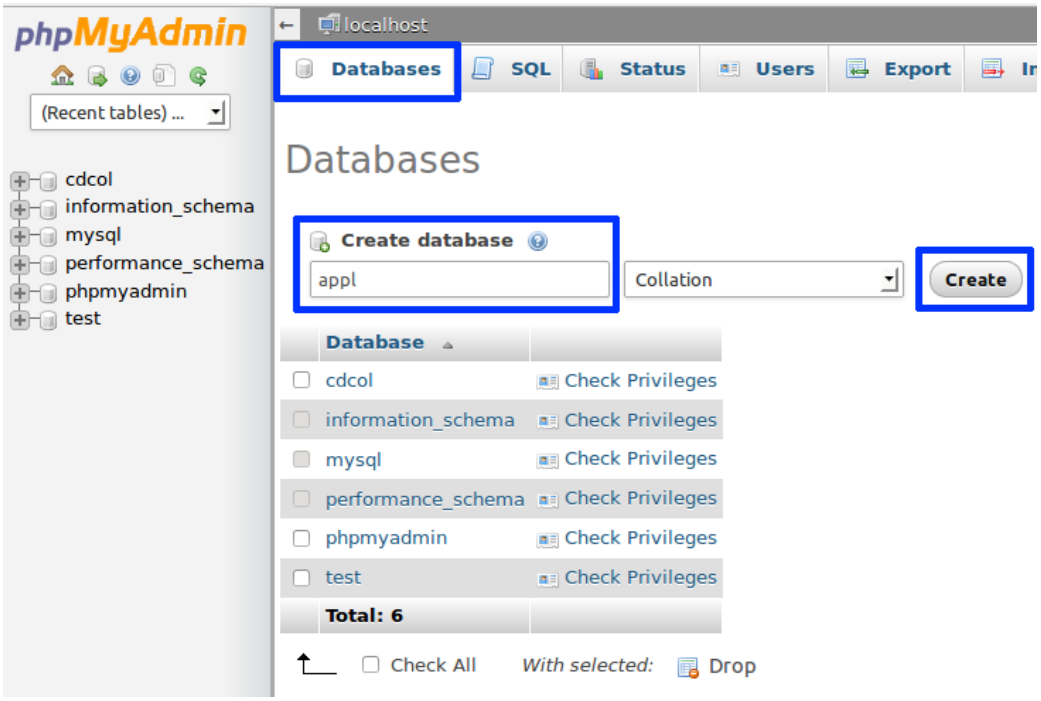

FIGURA A.7: Crear una Base de datos en MySQL

3. Configurar un usuario de la base de datos: en la lista de bases de datos, pulsar el enlace *Check Privileges* correspondiente a la base de datos recién creada.

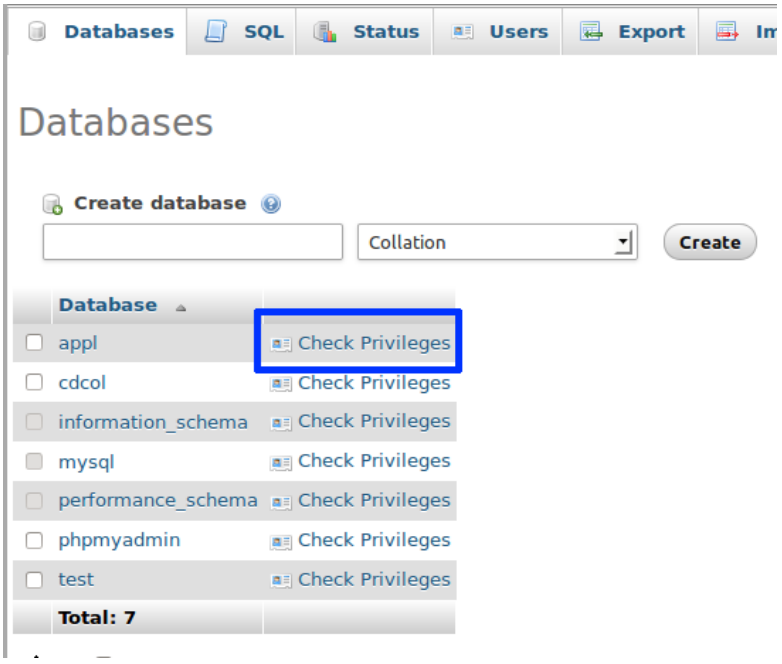

FIGURA A.8: Configurar usuario de la base de datos en MySQL (I)

En la siguiente ventana pulsar el enlace *Add user*

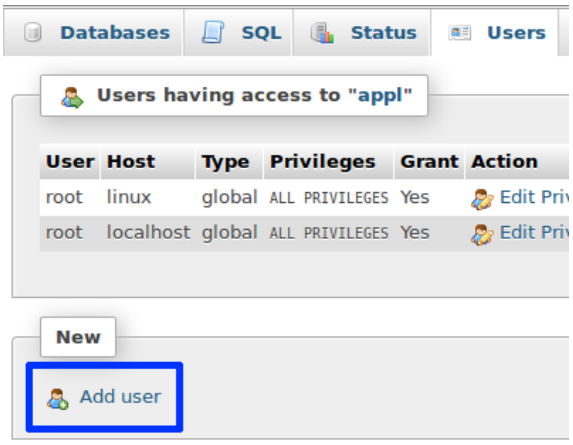

FIGURA A.9: Configurar usuario de la base de datos en MySQL (II)

En la siguiente página rellenar los datos del usuario a crear con nombre *appl* y contraseña *tab053*. En *host* elegir local a no ser que se vaya a acceder de forma remota, en ese caso indicar la dirección desde la que se va a acceder. Dejar seleccionada la opción *Grant all privileges on database "appl"* y pulsar el botón *Go* al final de la página - **no** el botón *Generate*.

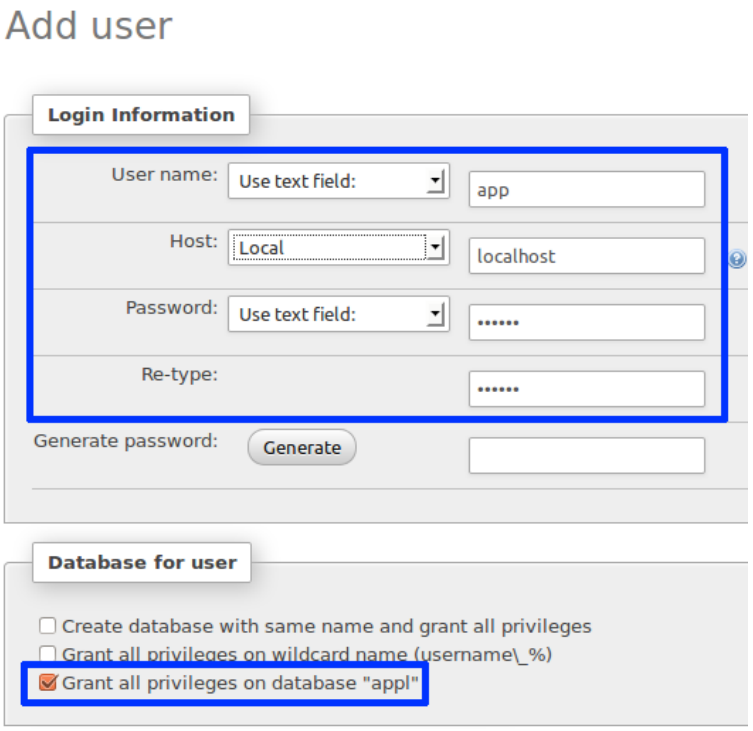

FIGURA A.10: Configurar usuario de la base de datos en MySQL (III)

4. Crear la tabla inventario: seleccionar en el menú lateral el enlace *appl*. En la nueva ventana, en la zona *Create Table* introducir los datos para crear la tabla de nombre

inventario y con un número de dos columnas. Pulsar el botón *Go*.

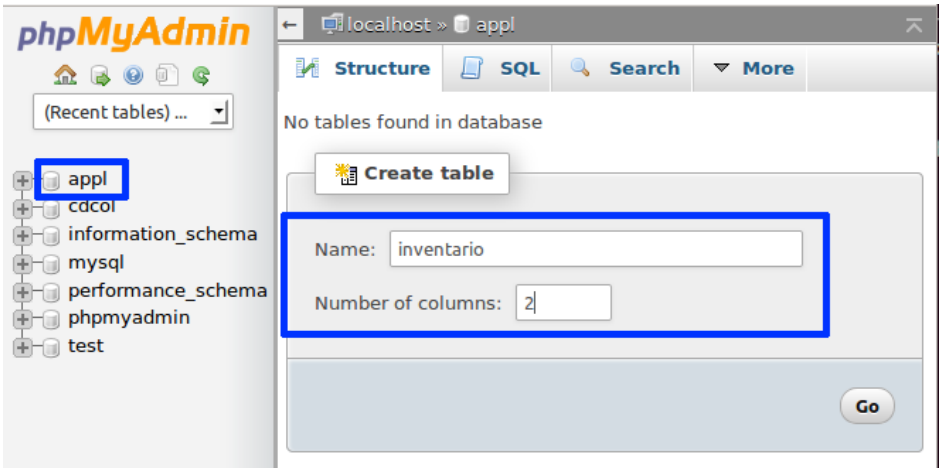

FIGURA A.11: Crear tabla inventario (I)

En la siguiente página introducir los datos de cada fila. Para la primera fila dar los paramentros *name* = *ArtID*, *Type* = *VARCHAR* , *Length* = *25* e *index* = *PRIMARY*. Para la segunda fila dar los paramentros *name* = *disponibilidad*, *Type* = *int*. Elegir como motor de almacenamiento a *InnoDB*, el cual permite realizar transacciones y cumplir los principios ACID. Finalmente pulsar el botón *Save*.

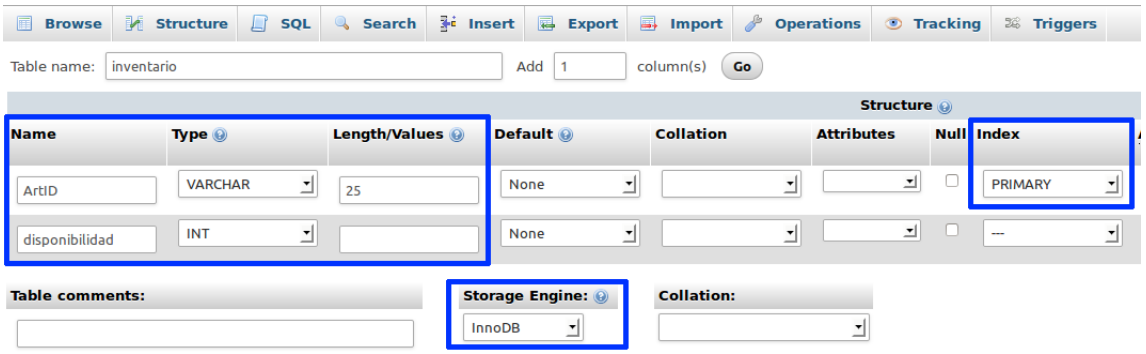

FIGURA A.12: Crear tabla inventario (II)

A.5 Oracle Java

Puesto que la aplicación se ha desarrollado en Java, es necesario instalar la máquina virtual. Para esto es necesario seguir los siguientes pasos [41]:

1. Descargar el paquete de instalación para Linux desde la página de Orcale.
2. Descomprimir el archivo:

```
> tar -xvf jdk-7u45-linux-x64.tar.gz
```

3. Crear un directorio para la máquina virtyal:

```
> sudo mkdir -p /usr/lib/jvm/jdk1.7.0
```

4. Mover los archivos descomprimidos al nuevo directorio.

```
> sudo mv jdk1.7.0_45/* /usr/lib/jvm/jdk1.7.0
```

5. Instalar las diferentes herramientas.

```
> sudo update-alternatives --install "/usr/bin/java" "java" "/usr/\
    lib/jvm/jdk1.7.0/bin/java" 1
> sudo update-alternatives --install "/usr/bin/javac" "javac" "/usr\
    /lib/jvm/jdk1.7.0/bin/javac" 1
> sudo update-alternatives --install "/usr/bin/javaws" "javaws" "/\
    usr/lib/jvm/jdk1.7.0/bin/javaws" 1
```

A.6 Apache Tomcat

En esta sección se va a explicar como instalar y configurar el servidor Apache Tomcat.

A.6.1 Instalación

La instalación del servidor se puede hacer de forma sencilla desde el repositorio. Para ello es necesario instalar dos paquetes:

1. Instalar el servidor:

```
> sudo apt-get install tomcat7
```

2. Instalar la interfaz de administración web:

```
> sudo apt-get install tomcat7-admin
```

Desde la dirección web *http://localhost:8080/* es posible acceder al administrador siempre que la instancia del servidor este activa. Sino para activarla es suficiente con hacer:

```
>  sudo /etc/init.d/tomcat7 start
```

A.6.2 Configuración

Puesto que el servidor no tiene un usuario por defecto pero requiere uno para acceder al sistema de administración web, hay que modificar el archivo *tomcat-users.xml* y añadir un usuario.

Ésto se puede hacer desde la línea de comandos siguiendo los siguientes pasos:

1. Acceder al archivo *tomcat-users.xml*:

```
> sudo nano /etc/tomcat7/tomcat-users.xml
```

2. Añadir las siguientes líneas para crear un administrador de nombre *app* con contraseña *tab053*

```
<role rolename="manager-gui"/>
<user username="app" password="tab053" roles="manager-gui"/>
```

3. Reiniciar el servidor:

```
> sudo /etc/init.d/tomcat7 restart
```

Por otro lado, es necesario configurar Tomcat para que utilice la versión de Java instalada en el punto anterior. Esto es necesario porque por defecto usa la versión OpenJDK, y puesto que el archivo *war* entregado esta compilado con la versión de Oracle anterior, se podrían producir errores al ejecutar el archivo. Por tanto para configurar esto es necesario modificar una serie de parametros en el archivo *tomcat7*, presente en el directorio */etc/default*. Para esto hay que seguir los siguientes pasos:

1. Acceder al archivo *tomcat7*:

```
> sudo nano /etc/default/tomcat7
```

2. Añadir las siguientes líneas para indicar que se quiere usar la versión de Java instalada en los pasos anteriores:

```
JAVA_HOME=/usr/lib/jvm/jdk1.7.0
```

3. Comentar o eliminar una línea similar a esta si está presente en el archivo - la versión o directorio de openjdk puede variar:

```
JAVA_HOME=/usr/lib/jvm/openjdk-6-jdk
```

4. Reiniciar el servidor:

```
> sudo /etc/init.d/tomcat7 restart
```

A.7 Desplegar la aplicación

Para desplegar la aplicación es suficiente con acceder al sistema de administrador web instalado anteriormente y subir el archivo *.war* que contiene la aplicación. Para esto se pueden seguir los siguientes pasos:

1. Con un navegador web ir a la dirección *http://localhost:8080/manager/html* para acceder al sistema de administración.

2. En la zona desplegar pulsar el botón examinar y subir el archivo *.war* que contiene la aplicación, y finalmente pulsar el botón desplegar.

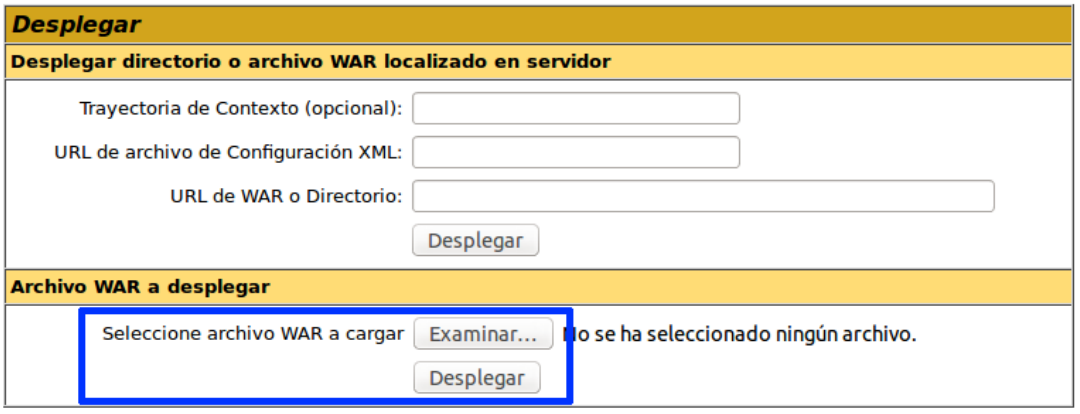

FIGURA A.13: Desplegar archivo .war

Apéndice B

Seguimiento y Control

En este apéndice se explican los diferentes acontecimientos y problemas surgidos a lo largo del proyecto. Además, se realiza una comparación con la planificación inicial del proyecto.

B.1 Desarrollo de los Objetivos

A lo largo del proyecto se marcaron como uno de los objetivos principales la familiarización con la idea de sistemas de almacenamiento NoSQL en general y con los sistemas de almacenamiento MongoDB - documentos - y OpenLink Virtuoso - grafos; y la realización de un documento con información detallada sobre el uso de estos sistemas. Además, también se marco como objetivo principal que se llevaría a cabo el desarrollo de una *Aplicación de Persistencia Políglota* la cual hiciese uso de los dos sistemas de almacenamiento anteriormente mencionados, junto al sistema de almacenamiento relacional MySQL; para llevar a cabo las diferentes tareas necesarias para el correcto funcionamiento de la aplicación.

Para la primera parte del proyecto se estudio la idea general de los sistemas de almacenamiento NoSQL y sus ventajas e inconvenientes. Seguidamente estudiaron de cada uno de los sistemas elegidos para el proyecto - y se realizó un documento explicando de forma práctica las diferentes características que se pueden encontrar cada uno de los sistemas, intentando relacionar la información - cuando se pudiese - con los sistemas relacionados con la idea de comparar las características de estos. Además, puesto que se quería mostrar ejemplos prácticos, se generaron varios *dataset's* con la idea además de ser reutilizados para su uso en la aplicación. Por último, se realizaron una serie de pruebas para mostrar el rendimiento de los distintos sistemas, valiéndose para ello de los con los *dataset's* generados.

En lo que se refiere a la segunda parte. Para realizar la *Aplicación de Persistencia Políglota*, primero se estudio en que consistía el concepto en si y las diferentes características que podría presentar una aplicación de este tipo. Seguidamente, se realizo el diseño de la aplicación comenzando con la parte relacionada con el uso de los sistemas de almacenamiento. Finalmente, siguiendo dicho diseño se implemento la aplicación y se realizaron pruebas de estabilidad de esta.

Finalmente, para la realización de la memoria, además de incluir la información típica del desarrollo de una aplicación general, se incluyo - aunque no se había planteado en los objetivos originales - un estudio de la comparación entre una aplicación simple con un único sistema relacional y la aplicación desarrollada. Para esto se realizaron tres tareas: (1) se diseño el modelo de datos para un sistema relacional que permitiese realizar las mismas tareas que la aplicación políglota, y se generó un *dataset* para dicho modelo de datos a partir de los datos almacenados en los sistemas NoSQL; (2) se desarrollo una aplicación simple para los cual la principal tarea que se realizo fue modificar todos los servicios que modifican y obtienen datos de los sistemas de almacenamiento para que realizasen las consultas sobre el sistema de almacenamiento relacional; y (3) se realizaron una serie de pruebas para comparar el rendimiento de las dos aplicaciones, configurando para ello todos los sistemas para poder obtener el mejor rendimiento posible en todos ellos.

B.2 Planificación temporal

En esta sección se va a hablar de las desviaciones - tanto del coste temporal como de las fecha de finalización de las diversas partes - que se han producido a lo largo del proyecto y se explicaran las causas que han llevado a dichas desviaciones así como las medidas tomadas en el caso que fuera necesario.

B.2.1 Actividades

Seguidamente se muestran las tablas con la información de las horas que se planeaba invertir en un primer momento junto con las horas reales invertidas en cada tarea.

La tabla B.1 muestra la planificación, junto al tiempo real empleado, del paquete de tareas "Planificación" el cual recoge tareas relacionadas con la planificación del desarrollo del proyecto.

Planificación	Tiempo estimado	Tiempo real
Planificación inicial	16h	16h
Análisis de requisitos	8h	8h
Plan de gestión	8h	5h
Replanificaciones	7h	6h
Total	**23h**	**22h**

Tabla B.1: Tiempo de planificación estimado y real

En este caso puesto que la planificación se realizo junto al documentos de objetos, el tiempo estimado de planificación - excluyendo las replanificaciones - es el mismo que el real.

La tabla B.2, correspondiente al paquete de gestión, muestra la estimación del tiempo, junto al tiempo real, que se tardará en realizar las tareas relacionadas con el control del progreso del proyecto

Gestión	Estimado	Real
Análisis de la ampliación	3h	3h
Seguimiento y Control	22h	16h
Reuniones	10h	7
Total	**35h**	**23h**

Tabla B.2: Tiempo de gestión estimado y real

En este caso la mayor desviación se produce en el caso del seguimiento y control - no incluyendo en este el tiempo de replanificación - ya que se llevó a cabo pensando en que se utilizaría una hora a la semana para revisar el tiempo invertido; sin embargo, se vio que con la mitad de tiempo por semana era suficiente. Otra desviación se dio en las reuniones, que ha sido menor al esperado porque, por un lado la mayoría de las reuniones duraron menos de los esperado; y porque, además, puesto que se realizó el proyecto en la facultad, las posibles dudas presentes se consultaban en el momento.

La tabla B.3 muestra una comparación sobre el tiempo que se estima que el desarrollador tardaría en formarse en las diferentes tecnologías a usar de las que no se tenga previo conocimiento; y el tiempo real que se tardó en realizar la tarea.

Formación	Estimado	Real
Formación en NoSQL general	10h	10h
Formación en MongoDB básico	30h	27h
Formación en OpenLink Virtuoso básico	30h	35h
Formación para realizar la Aplicación	10h	10h
Formación en Freebase	-	4h
Total	**80h**	**86h**

Tabla B.3: Tiempo de formación estimado y real

La mayor desviación presente en la formación, se dio en el caso de OpenLink Virtuoso, en este se planificó un tiempo similar al necesario para formase en MongoDB; sin embargo, debido a una documentación confusa para la versión *Opensource*, y a la falta de una comunidad tan grande como la de MongoDB, el tiempo necesario para formase resulto algo más alto.

Por otro lado, otro punto importante en la desviación global de la tarea se debió al uso de la base de datos *Freebase* como origen de los datos utilizados para general los diferentes *dataset's*. Una de las principales razones para usar ésta en lugar de otras bases de datos - o *dataset's* - es que esta base de datos contiene una gran cantidad de datos relacionados con los artículos que se querían usar para el comercio electrónico. Por otro lado, aunque era posible generar los datos mediante datos del tipo "Pelicula1", "Película2"; no se sabía si la respuesta de los sistemas de almacenamiento al usar datos tan similares podría general alguna desviación en el rendimiento real de estos. Por tanto, se decidió usar la API *Freebase* que permite obtener los datos usando el lenguaje de programación *Java*.

La tabla B.4 recoge una comparación entre el tiempo que se estimó se tardaría en desarrollar el diseño, y la aplicación con las especificaciones indicadas en el alcance mínimo; y el tiempo real. Además por cada punto, se especifica el tiempo que se tardará en diseñar, implementar y realizar las pruebas.

Desarrollo	Estimado	Real
Backups de las BD	5h	5h
Desarrollo MongoDB	30h	44h
Diseño del entorno MongoDB	15h	22h
Diseño general	10h	10h
Generación de datasets	5	12h
Implementación de operaciones MongoDB	10h	14h
Pruebas de rendimiento MongoDB	5h	5h
Desarrollo OpenLink Virtuoso	30h	32h
Diseño del entorno OpenLink Virtuoso	15h	19h
Diseño general	10h	10h
Generación de datasets	5	9h
Implementación de operaciones OpenLink Virtuoso	10h	10h
Pruebas de rendimiento OpenLink Virtuoso	5h	3h
Desarrollo de la aplicación Políglota	105h	104h
Diseño	25h	25h
Implementación	65h	68h
Pruebas	15h	10h
Desarrollo de la aplicación en MySQL	-	52
Diseño	-	15h
Diseño general	-	10h
Generación de datasets	-	5h
Implementación	-	37
Total	**170h**	**234h**

Tabla B.4: Tiempo de desarrollo estimado y real

Uno de los primeros focos de desviación se debe a la parte del diseño del entrono, la cual incluía el diseño del modelo de los datos que se iban a usar a lo largo de todo el proyecto - incluyendo parte del modelo de datos de la aplicación políglota -; diseño y configuración general del entorno de trabajo; y generación de los datos en función del modelo de datos diseñado. En el caso de MongoDB el tiempo empleado en el diseño general fue mayor que en Virtuoso puesto se diseñaron varios modelos de datos ya que se desconocía cual podría dar mejor rendimiento. En cualquier caso, el mayor problema se dio en el desarrollo de las aplicaciones para generar los *dataset's*, debido: (1) a que *Freebase* esta mantenido por la comunidad, por lo que los datos guardados no son perfectos y en ocasiones los datos obtenidos contenían valores nulos o no se obtenía la suficiente cantidad de datos por solicitar demasiados campos de datos; (2) debido a que no se tenía gran experiencia

con los drivers de los sistemas y la API de *Freebase* usados para desarrollar los programas que generan estos *datasets*; y (3) en el caso de los datos de Virtuoso hubo que desarrollar una aplicación algo compleja que generase el *dataset* no sólo a partir de los datos de la base de datos *Freebase*, sino también en función de los datos contenidos en MongoDB, ya que se necesitaba que los datos de los artículos para ambos sistemas fueran los mismos ya que iban a ser utilizados por la *Aplicación de Persistencia Políglota*.

Otro factor de desviación importante fue que se consideró que sería interesante realizar una comparación entre el tipo de aplicación del proyecto y otra aplicación similar que usará un único sistema. Debido a esto, se decidió desarrollar una aplicación con características similares para así hacer una comparación práctica de ambos tipos de aplicaciones. Para esto fue necesario: (1) rediseñar parte de la aplicación y diseñar un modelo de datos para una base de datos relacional; (2) generar un *dataset* que siguiera el modelo anterior, generando los datos a partir de los datos almacenados en los sistemas de almacenamiento iniciales; y (3) implementar la nueva aplicación, lo cual consistió principalmente en una nueva implementación de los servicios y una reutilización del resto del código.

Finalmente, la tabla B.5 muestra una comparación del tiempo real y el tiempo que se estimó se tardaría en realizar las tareas de documentación, tanto de la memoria del proyecto, como de la documentación del diseño del entorno de trabajo sobre las diferentes bases de datos.

Documentación	Estimado	Real
Memoria	40h	79h
Memoria general	40h	55h
Análisis comparativo de las aplicaciones	-	4h
Pruebas de rendimiento de las aplicaciones	-	20h
Revisiones	10h	18h
Defensa	10h	10h
Documentación MongoDB y Virtuoso	25h	40h
Total	**85h**	**147h**

Tabla B.5: Tiempo de documentación estimado y real

Tanto en el caso de la documentación de la memoria general y el documento de diseño se ha sobrepasado la estimación inicial que se había planteado debido principalmente a una estimación errónea del tamaño que se pensaba tendrían los documentos. En el caso de la memoria además se necesitaron dos tareas extras: (1) el análisis de las diferentes

diferencias y semejanzas que se pueden encontrar encontrar en las dos aplicaciones; y (2) la realización de pruebas. El tiempo de esta segunda tarea extra se debe no sólo a la realización de las pruebas en si - y a la implementación de un pequeño programa de pruebas -, sino también a la configuración que se realizó sobre los distintos sistemas. En concreto, Virtuoso daba tiempos superiores a MySQL, con lo que se intento optimizar el sistema realizando una serie de modificaciones de los índices, modificando las opciones de configuración del servidor, reinstalando los sistemas y las aplicaciones en nuevos ordenadores con mayor memoria para comprobar que los índices entraban en memoria, limpiando la base de datos para que sólo estuvieran los datos usados por la aplicación, reconstrucción de los índices para que no hubiera índices *basura*...

Finalmente, el tiempo total empleado en el desarrollo del proyecto para el alcance mínimo se supero al estimado - 393 horas -, debido principalmente a tres factores: (1) la generación de los *dataset*; (2) la documentación del proyecto; y (3) la inclusión de nuevas tareas para realizar una comparación práctica de la aplicación inicial con una aplicación típica que hiciese uso de un único sistema relacional.

B.2.2 Actividades de la ampliación

Seguidamente se muestra en la tabla B.6 una comparación del tiempo real y del tiempo estimado que se podría tardar en realizar las tareas de ampliación.

Ampliación	Estimado	Real
Formación Amazon EC2	10h	2h
MongoDB	55h	24h
Replicación	20h	10h
Sharding	20h	10h
Mapreduce	10h	-
Documentación	5h	4h
Openlink Virtuoso	55h	24h
Replicación	20h	10h
Sharding	20h	10h
Mapreduce	10h	-
Documentación	5h	4h
Total	-	**50h**

Tabla B.6: Estimación del tiempo de la ampliación y real

En la planificación inical no se calculó concretamente el tiempo que se tardaría en realizar las tareas, sino que se estimo el tiempo que se podría tardar en realizar en función del tiempo extra disponible.

En la realidad, el tiempo necesario para la realización de la ampliación fue menor debido a varios factores: (1) el tiempo estimado de la ampliación estaba basado principalmente en el tiempo extra disponible; (2) se eliminó la tarea de *Mapreduce* puesto que en Virtuoso no existe un *Framework* concreto para realizar estas operaciones sobre datos RDF; y (3) aunque se estudió el uso de Amazon, se optó por no usarlo por dos razones, la primera porque era posible mostrar el funcionamiento básico de replicación y *sharding* usando una máquina local; y la segunda porque se disponía de una cuenta en otro servicio de almacenamiento, que aunque no disponía de instancias muy potentes, servía para realizar comprobaciones sobre un entorno distribuido.

B.2.3 Hitos y replanificaciones

Seguidamente se muestra la tabla B.7 con los hitos más importantes junto a la fecha en la que se deberían dar y en la que se realizaron.

Hitos	Hito Plan	Hito Real
Planificación inicial	15/07/2013	15/07/2013
Análisis de la viabilidad del proyecto	29/07/2013	29/07/2013
Diseño MongoDB básico	29/07/2013	08/08/2013
Prototipo de la aplicación	16/09/2013	20/09/2013
Diseño MongoDB y OpenLink Virtuoso básico	22/08/2013	03/10/2013
Finalización de la aplicación	30/09/2013	11/10/2013
Finalización aplicación simple	-	13/11/2013
Borrador de la memoria	18/11/2013	22/11/2013
Análisis de ampliación	30/09/2013	02/12/2013
Finalización de los entregables	02/12/2013	03/02/2014
Finalización del proyecto	15/01/2014	28/02/2014

Tabla B.7: Fecha de Hitos

Como se puede ver ser produjeron una serie de desviaciones a lo largo del proyecto en gran parte debido a las desviaciones temporales comentadas en el punto anterior. Seguidamente se comentarán las diferencias existentes entre los hitos de los planificados y los reales.

- El diseño básico en MongoDB se terminó aproximadamente una semana y media más tarde de lo esperado debido al retraso al generar los *dataset*, y además quedaba pendiente pasar a Latex la documentación realizada. Debido a esto, durante el análisis de viabilidad se decidió realizar una replanificación y retrasar todo el proyecto dos semanas.

- El diseño de Virtuoso no se terminó completamente para la fecha de replanificación ya que quedaba terminar la documentación y realizar la revisión del diseño completo. En este caso, para no alargar el comienzo del desarrolló de la aplicación se decidió comenzar con la aplicación mientras se terminaba el documento de diseño.

- El prototipo de la aplicación se retrasó unos cinco días después de la fecha de planificación inicial, debido a los retrasos de las fases anteriores. De la misma manera la finalización de la aplicación se dio aproximadamente unas dos semanas después después de la fecha inicial planificada. Si bien es cierto que a lo largo del proyecto se realizaron modificaciones estéticas a la aplicación - y resolución de problemas debidos a estas modificaciones.

- El día de la presentación del prototipo se decidió adelantar la realización de la memoria y retrasar la realización de la ampliación del alcance mínimo. También se decidió la realización de un análisis de comparación de la aplicación políglota con una aplicación de un sólo sistema de almacenamiento relacional. Por tanto, tras la finalización de la aplicación se realizó una replanificación del proyecto. En esta replanificación se contemplaron las siguientes tareas:

  1. Realización de la primera parte de la documentación de la memoria (14/10 - 28/10).
  2. Realización de la aplicación un sólo sistema relacional incluyendo la generación de un *dataset* que contenga los mismos datos que están almacenados en los sistemas de almacenamiento NoSQL. (29/10 - 18/11)
  3. Realización del análisis de comparación y de las pruebas, entre las dos aplicaciones (19/11 - 25/11).
  4. Finalización del resto de la memoria - documentación del análisis y las pruebas y conclusiones (25/11 - 02/11).

  Debido a estas nuevas tareas los hitos quedaron tal y como se muestra en la figura B.8.

Hitos	Hito Plan
Primera Parte de la memoria	28/10/2013
Finalización de la aplicación simple	13/11/2013
Borrador de la memoria	22/11/2013
Análisis de ampliación	02/12/2013
Finalización de los entregables	03/02/2014
Finalización del proyecto	28/02/2014

Tabla B.8: Fecha de Hitos segunda replanificación

- El borrador se entrego el 22/11/2013, sin embargo, se continuaron haciendo más pruebas para intentar mejorar el rendimiento de Virtuoso, el cual no era el esperado.
- Durante la primera semana de diciembre se realizó la planificación de la ampliación de los objetivos.
- Durante el resto de diciembre se realizó la ampliación del proyecto, y también se empleo este tiempo para modificar el aspecto estético de la aplicación - se modificaron colores, dimensiones de las tablas y la forma de algunos botones.
- Durante las dos primeras semanas de enero se realizaron las correcciones de la memoria, y se hicieron unas últimas configuraciones y pruebas para intentar mejorar el rendimiento de Virtuoso.

Apéndice C

MongoDB: Información Adicional

C.1 Instalación

En esta sección se muestra como instalar MongoDB en Ubuntu y como configurar diferentes opciones de la instancia del servidor. Como sistema operativo se ha utilizado la versión 12.04 de Ubuntu de 64bits. Sobre este sistema, se ha instalado la versión 2.4 de MongoDB, también de 64bits para evitar las limitaciones que acarrea la versión de 34bits ya que la versión de 32bits únicamente permite utilizar 4GB de memoria.

C.1.1 Instalación Básica

La instalación de MongoDB en Ubuntu se puede realizar directamente desde el repositorio de Ubuntu.Sin embargo, es posible que la versión disponible en el repositorio no sea la más reciente. El uso de versiones antiguas de MongoDB es especialmente problemático, ya que al ser un sistema relativamente nuevo, con cada versión se incluyen nuevas características que pueden ser de gran importancia. Por ejemplo, hasta la versión 2.2 la granularidad de los cerrojos era a nivel de toda la instancia del servidor, pero con la 2.2 ésta paso a ser a nivel de las bases de datos - se espera que la versión 2.6 de cerrojos a nivel de colección.

Por tanto, la forma recomendada de instalar MongoDB es añadiendo los repositorios oficiales de 10gen y después instalar los paquetes necesarios. Para ésto, se pueden seguir los siguientes pasos:

1. Importar la clave GPG firmada de 10gen necesaria para descargar desde apt los paquetes del repositorio de 10gen:

```
>sudo apt-key adv --keyserver keyserver.ubuntu.com
 --recv 7F0CEB10
```

2. Cear una lista con la dirección del repositorio de 10gen:

```
>echo 'deb
http://downloads-distro.MongoDB.org/repo/ubuntu-upstart dist 10gen'
| sudo tee /etc/apt/sources.list.d/10gen.list
```

3. Recargar el repositorio en apt:

```
>sudo apt-get update
```

4. Instalar el paquete que instalará la última versión estable:

```
>sudo apt-get install MongoDB-10gen
```

C.1.2 Inicio de la instancia del servidor

Existen dos formas de iniciar el servidor:

- La primera es arrancar el servidor en modo normal mediante el comando mongod <opciones> desde la linea de comando. Algunas de las opciones que se le pueden dar son: opciones de identificación, ip's permitidas, opciones de replicación, especificar el directorio donde se encuentran las bases de datos... Alternativa también se le puede indicar que utilice la configuración presente en el fichero MongoDB.conf con la opción "-f" : monogod -f /etc/MongoDB.conf .
- La segunda opción es arrancar el servidor en modo servicio mediante el comando "sudo service MongoDB start".

C.1.3 Configuración

Si se arranca la instancia con "mongod", se utilizará el directorio - dbpath - "/data/db" para guardar los datos. Sin embargo, dicho directorio no se crea durante la instalación y se genera un error al iniciar la instancia. Para evitar esto existen dos opciones. La primera sería utilizar siempre el fichero de configuración al iniciar la instancia, el cual tiene por defecto asignado otro directorio donde guardar los datos. La segunda opción es crear el directorio, en este caso además sería necesario configurar los derechos de acceso para MongoDB de la siguiente forma: > sudo chown -R MongoDB:nogroup /data/db

Al iniciarse la instancia del servidor mediante "service MongoDB" se utiliza la información contenida en el archivo de configuración ('/etc/MongoDB.conf'), en esta configuración por defecto lo opción dbpath esta configurada para que se utilice el directorio

/var/lib/MongoDB, a diferencia de si se inicia la instancia con mongod. Si se quiere cambiar el directorio donde se guardan los datos para que coincida con el del caso anterior, se debería modificar la opción "dbpath" del fichero de configuración y asignarle el directorio "/data/db".

C.2 Control de usuarios

En esta sección se va a hablar de como configurar adecuadamente el control de usuarios del sistema. MongoDB guarda los datos de los usuarios en unas colecciones llamadas "system.users", estas colecciones se crean automáticamente en todas las bases de datos. Para identificarse los usuarios deben hacerlo sobre aquella base de datos que contengan sus credenciales. Existe, además, una base de datos llamada "admin" que es la única en la que los administradores pueden dar privilegios para que los usuarios usen sobre otras bases de datos distintas.

En lo que se refiere al nivel al que se pueden dar privilegios sobre objetos, en MongoDB sólo se pueden dar privilegios a nivel de base de datos, es decir, si bien en MySQL se pueden dar privilegios a otros usuarios sobre tablas, en MongoDB no se puede dar privilegios sólo sobre una colección, sino que es necesario darlos sobre toda la base de datos que contenga dicha colección.

En cuanto a los privilegios de los usuarios en si, MongoDB no permite dar privilegios concretos sino que hay que dar un rol con todos los privilegios que éste tenga, tampoco se pueden crear roles personalizados con lo que no se puede crear un rol personalizado con los privilegios que interese dar al usuario.

Por defecto MongoDB no tiene activado ningún sistema de control de usuarios por lo que es necesario incluir la opción "–auth" al lanzar el servidor con "mongod", o bien poner a "true" la opción "–auth" del fichero de configuración "MongoDB.conf". Sin embargo, por defecto no hay ningún usuario creado a diferencia de en otros sistemas de bases de datos como por ejemplo MySQL que durante la instalación solicitan un usuario y una contraseña para crear el usuario "root" inicial. Además, MongoDB está configurado de forma que permite sobrepasar el sistema de identificación si se está conectando al servidor de forma local.

C.2.1 Creación de un administrador

Lo primero que habría que hacer si se quiere activar el sistema de identificación es crear un usuario con un rol de tipo "userAdmin" o "userAdminAnyDatabase". Estos dos roles

otorgan a los usuarios privilegios administrativos básicos y además son los únicos que pueden crear usuarios y otorgarles privilegios. La única diferencia entre estos dos roles, es que el primer rol únicamente permite dar permisos en una determinada base de datos, mientras que con el segundo rol el usuario que lo tenga puede dar permisos en cualquier base de datos. En ninguno de los dos casos el usuario tiene privilegios de lectura o de escritura, sin embargo se podrá dar a si mismo estos permisos.

En general, para crear y dar privilegios a los usuarios normales es necesario crear un nuevo documento con los datos del usuario y añadirlos a la colección "system.users" de la base de datos en la que se quiere dar los privilegios al usuario. Sin embargo, para crear este primer usuario es necesario hacerlo sobre una base de datos del sistema llamada 'admin'. Los pasos a seguir para crear al usuario son los siguientes:

1. Lanzar el servidor sin la configuración "–auth" - o con la opción puesta a false en el archivo de configuración - si no se va a conectar de forma local - localhost - al servidor, si se conecta de forma local no es necesario poner la opción "–auth", siempre y cuando la opción de bypass esté activada (más adelante se explica como desactivarla y activarla).

2. Usar la base de datos 'admin'

```
> db=db.getSiblingDB('admin')
```

3. Crear un usuario con el role userAdminAnyDatabase:

```
> db.addUser({user: "admin",
              pwd: "tab053",
              roles:["userAdminAnyDatabase", "clusterAdmin"]
             })
```

   Lo que el método addUser hace es añadir un documento con los datos del usuario en la colección "system.users" de la base de datos admin. En este caso los datos con los que se creará el documento son el usuario ('user'), la contraseña ('pwd') y la lista de roles ('roles'). MongoDB no encripta ninguno de los datos excepto el campo de la crontraseña.

   Además de estas opciones también sería posible indicar otras opciones añadiendo los campos:

   (a) userSource: permite que se use una base de datos donde se encuentran recogidas las credenciales para identificar al usuario. Si se usa esta opción no se puede usar la opción pwd.

(b) otherDBRoles: permite asignar roles al usuario sobre otras bases de datos. Esta opción solo es válida para los casos en los que se esté usando la base de datos 'admin'.

Para que la identificación tenga efecto es necesario reiniciar el servidor con el parámetro de identificación y si es necesario, dependiendo desde donde se hagan las conexiones y los usuarios que lo utilicen, con la opción de bypass de identificación para el localhost desactivada:

```
>sudo  mongod --auth --setParameter enableLocalHostAuthBypass = 0
```

Alternativamente, se pueden modificar estas opciones en el archivo de configuración MongoDB.conf y lanzar la instancia como servicio o con el parámetro "-f /etc/MongoDB.conf".

Una vez hecho esto se puede iniciar el cliente "mongo" como se hacía anteriormente, pero el usuario no podrá realizar ninguna operación. Para identificarse existen diferentes opciones, pero siempre hay que tener en cuenta que la identificación hay que hacerla sobre la base de datos que contiene las credenciales, es decir, en el caso del usuario que se ha creado anteriormente, hay que identificarse sobre la base de datos 'admin' que contiene los datos de usuario en la colección 'system.users'.

Para identificarse se pueden usar los siguientes métodos:

1. Mientras se arranca el cliente:

   ```
   > mongo --username admin --password tab053
     --authenticationDatabase admin
   ```

   Además de los parámetros "usuario" y "contraseña" es necesario incluir el parámetro 'authenticationDatabase' para indicar sobre qué base de datos comprobar las credenciales del usuario. Alternativamente se puede usar -u y -p en lugar de –user y –password respectivamente.

2. Una vez abierto el cliente mediante el comando 'mongo' en el terminal, dentro del shell de monogDB, usar la base de datos sobre la que se tienen permisos y mediante el método "db.auth()" identificarse. Los pasos a seguir serían los siguiente:

   ```
   > db=db.getSiblingDB('admin')
   > db.auth('admin','tab053')
   ```

Por último, si bien no es necesario crear un usuario "administrador" el cual pueda crear usuarios y otorgar privilegios, resulta más cómodo tenerlo por si se da el caso de que

se vayan a tener diferentes usuarios, ya que de lo contrario cada vez que se quisiera añadir o modificar un usuario se debería reiniciar el servidor, desactivar las opciones de identificación, modificar los usuarios y volver a iniciar el servidor con las opciones de identificación activadas.

C.2.2 Creación del resto de usuarios

Para configurar un nuevo usuario primero se ha creado una base de datos llamada "contenidos". Después se ha creado un usuario llamado "app" y se le ha aplicado los privilegios suficientes para que pueda realizar tareas de escritura y lectura - roles readWrite y dbAdmin.

Para crear el usuario "app" se siguen los siguientes pasos:

1. Se inicia una sesión con el usuario administrador:

```
>mongo --user admin --password tab053 --authenticationDatabase \
    admin
```

2. Una vez iniciado el cliente, se usa la base de datos "contenidos". Aunque esta base de datos no esté creada, no es necesario crearla, una vez que se use y se añada un nuevo usuario, dicha base de datos se creará automáticamente.

```
>db = db.getSiblingDB('contenidos')
```

3. Para crear al usuario se añade un nuevo usuario con los roles "readWrite" y "adminDB" que le permitan escribir, leer y realizar tareas administrativas sobre la base de datos.

```
> db.addUser({user: "app",
              pwd: "tab053",
              roles:["readWrite", "dbAdmin"]
            })
```

   Los privilegios otorgados son únicamente sobre la base de datos lógica "contenidos".

4. Una vez hecho esto el nuevo usuario podrá iniciar sesión de forma similar a como lo hace el administrador, con la excepción de que el nuevo usuario debe identificarse sobre la base de datos "contenidos" y no sobre la base de datos "admin".

```
> mongo --username app --password tab053
        --authenticationDatabase  contenidos
```

C.2.3 Modificación de usuarios

Existen diferentes opciones para modificar la información de los usuarios dependiendo del la información que se quiera cambiar.

- db.changeUserPassword() : permite cambiar la contraseña del usuario.
- db.removeUser() : permite eliminar a un usuario dado.
- Para modificar los roles de un usuario no existe ningún método que haga esto de forma automática. Sin embargo, puesto que los roles son guardados en una lista, es posible realizar operaciones de tipo "update()", que permitan modificar listas dentro de documentos, sin tener que indicar todos los elementos que pertenecen a la lista, es decir, por ejemplo se puede añadir un elemento sin tener que especificar el resto de elementos que ya estaban en la lista.

  1. Añadir un rol a un usuario creado anteriormente :

  ```
  > db.system.users.update
      ({user: "app"},
       {$addToSet : { roles: 'read' }}
       )
  ```

  "En esta operación se utiliza el método "update()" para indicar que se quiere modificar un documento con usuario "app". Con "$addToSet" se indica que se quiere añadir un nuevo elemento "read" a la lista del campo roles. Más adelante, en el apartado de operaciones básicas, se indicarán las diferentes operaciones que se pueden hacer sobre listas.

  2. Eliminar un rol del usuario:

  ```
  > db.system.users.update(
       {user: "app"},
       { $pull: { roles: 'read' }})
  ```

  En este caso se hace una operación similar, pero en lugar de utilizar "$addToSet" para añadir un elemento, se utiliza el operador "$pull" el cual eliminará el elemento - y todas las coincidencias - de la lista de roles que coincida con "read".

C.3 Concurrencia

MongoDB permite realizar lecturas de forma concurrente; sin embargo, cuando se realizan escrituras se produce un bloqueo que afecta tanto a las operaciones de lectura

como a las operaciones de escritura. Este bloqueo es a nivel de base de datos, es decir, que cuando se realiza una operación de escritura en una colección, entonces, todos las colecciones que se encuentran en la misma base de datos se bloquean con ella.

Por otro lado, MongoDB da prioridad a las escrituras, por lo que si hay lecturas y escrituras para recibir un cerrojo, éste será dado a las operaciones de escritura.

Para ver que tipos de cerrojos están activos se puede utilizar el método "db.currentOp()" - sólo para usuarios con el rol "clusterAdmin" – el cual devuelve un documento con la lista de las operaciones que estén en ejecución en ese momento.

C.3.1 Ejemplo 1

Para comprobar el uso de los cerrojos se pueden abrir dos conexiones, una con el usuario "admin" - el cual debería tener el role "clusterAdmin" – y otra con el usuario "app".

Mientras el usuario "app" realiza la consulta: "db.articulos.find({"titulo":"Toy Story"});" el administrador ejecutará la operación "db.currentOp()". Ésta última operación debería devolver la información de la operación realizada por el usuario "app". Ésta debería ser algo similar a lo siguiente:

```
{ "inprog" : [ {     "opid" : 1458,
                        "active" : true,
                        "secs_running" : 2,
                        "op" : "query",
                        "ns" : "contenidos.articulos",
                        "query" : {
                                "titulo" : "Toy Story"
                        },
                        "client" : "127.0.0.1:37648",
                        "desc" : "conn14",
                        "threadId" : "0x7f113fe0d700",
                        "connectionId" : 14,
                        "locks" : {
                                "^" : "r",
                                "^contenidos" : "R"
                        },
                        "waitingForLock" : false,
                        "numYields" : 3,
                        "lockStats" : {
                                "timeLockedMicros" : {
                                        "r" : NumberLong(3162476),
                                        "w" : NumberLong(0)
                                },
                                "timeAcquiringMicros" : {
```

```
                                        "r" : NumberLong(1581498),
                                        "w" : NumberLong(0)  } }}]}
```

El campo "inprog" es un array de documentos, que en este caso sólo contiene un elemento, debido a que sólo hay una operación en curso.

Entre todos los campos aquellos que son más interesantes son:

- ns: indica sobre qué base de datos se está realizando el bloqueo.
- Op: indica qué tipo de operación se está realizando.
- "query": indica qué consulta es la que está realizándose en ese momento.
- Client/desc/threadId/connectionId: permite identificar quien está realizando la operación.
- Locks: en este caso, indica que hay un cerrojo de tipo lectura sobre la base de datos contenidos - ^contenidos:R. Y si contie32 0ne el campo " ^", indica que la operación actual es la dueña del cerrojo.
- WaitingForLock: que indica si dicha operación está o no esperando a que otra operación libere el cerrojo.

C.3.2 Ejemplo 2

En este ejemplo habrá que conectarse con tres usuarios, como en el caso anterior se tiene al usuario "admin" que realizará la consulta de información sobre las operaciones en curso, el usuario "app" que realizará una operación de escritura en la base de datos contenidos y un tercer usuario "lector" que realizará una operación de lectura en la misma base de datos, mientras "app" realiza la operación de escritura.

Las acciones a realizar por los usuarios son las siguientes:

1. El usuario app realiza la operación de escritura:

```
> db.articulos.update(
    {"titulo":"Toy Story", "tipo":"movie"},
    {$push :{'detalles.actores': {$each:["Paco","Paco","Paco2"]}}
    })
```

2. El usuario lector realiza la operación de lectura:

```
> db.articulos.find({"titulo":"WALL-E"})
```

3. El administrador ejecuta el método db.currentOp() para obtener la información sobre el estado de las operaciones.

El resultado obtenido es el siguiente:


```
{
   "inprog" : [
                {
                        "opid" : 1677,
                        "active" : true,
                        "secs_running" : 7,
                        "op" : "update",
                        "ns" : "contenidos.articulos",
                        "query" : {
                                "titulo" : "Toy Story",
                                "tipo" : "movie"
                        },
                        "client" : "127.0.0.1:37648",
                        "desc" : "conn14",
                        "threadId" : "0x7f113fe0d700",
                        "connectionId" : 14,
                        "locks" : {
                                "^" : "w",
                                "^contenidos" : "W"
                        },
                        "waitingForLock" : false,
                        "numYields" : 7782,
                        "lockStats" : {
                                "timeLockedMicros" : {
                                        "r" : NumberLong(0),
                                        "w" : NumberLong(12868028)
                                },
                                "timeAcquiringMicros" : {
                                        "r" : NumberLong(0),
                                        "w" : NumberLong(7910584)
                }}},
                {
                        "opid" : 1678,
                        "active" : true,
                        "secs_running" : 4,
                        "op" : "query",
                        "ns" : "contenidos.articulos",
                        "query" : {
                                "titulo" : "WALL-E"
                        },
                        "client" : "127.0.0.1:37940",
                        "desc" : "conn22",
                        "threadId" : "0x7f0946b0c700",
```

```
                        "connectionId" : 22,
                        "locks" : {
                                "^contenidos" : "R"
                        },
                        "waitingForLock" : true,
                        "numYields" : 7737,
                        "lockStats" : {
                                "timeLockedMicros" : {
                                        "r" : NumberLong(2229549),
                                        "w" : NumberLong(0)
                                },
                                "timeAcquiringMicros" : {
                                        "r" : NumberLong(4083017),
                                        "w" : NumberLong(0)
        }}}]}
```

En este caso, se puede ver que el resultado consta de dos subdocumentos dentro del array, lo cual indica que hay dos operaciones en curso.

La primera operación es la consulta "update" realizada por el usuario "app". Como se puede ver en el campo lock, se ha solicitado un cerrojo de escritura sobre la base de datos "contenidos", y además se puede ver que se le ha concedido dicho cerrojo ya que existe el campo "lock.^".

La segunda operación es la "query" que realizó el usuario lector, y en el campo lock se indica que esta operación solicitó un cerrojo de lectura sobre la base de datos contenidos, pero que a diferencia de la operación anterior, a esta operación no se le ha concedido el cerrojo puesto que el campo "lock.^" no existe. Además, el campo "waitingForLock" es igual a true, lo que significa que esta operación está esperando a que el cerrojo sobre la base de datos quede libre.

C.4 Transacciones

Uno de los principales problemas presentes en la mayoría de los sistemas de bases de datos NoSQL es que no cumplen con las propiedades ACID, por lo que no soportan transacciones.

Sin embargo, MongoDB ofrece una serie de herramientas las cuales permiten realizar semitransacciones.

C.4.1 Operador $isolation

Por lo general el nivel de aislamiento que da MongoDB es a nivel de documento, por lo que si una misma operación necesita modificar varios documentos, dicha operación no será atómica. El Operador "$isolation" permite que todas la modificaciones que se hagan durante una misma consulta - por ejemplo si con una sola consulta de modificación se modifican varios documentos -, se consideren como una operación atómica.

A pesar de que se puede utilizar el operador para que se realicen todas los modificaciones sobre distintos documentos como una sola operación, no soporta la característica "todo o nada" de las operaciones atómicas de otros sistemas, es decir, aunque alguna modificación no sea correcta no se deshacen las modificaciones ya realizadas, ni se cancela el resto.

Este operador se puede aplicar a cualquier operación de escritura, a operaciones tanto de modificación, eliminación e inserción.

Por ejemplo, para modificar la duración de todas los artículos con titulo "WALL-E", se haría lo siguiente:

```
> db.articulos.update(
    {"titulo":"WALL-E","tipo":"movie",$isolated : 1 },
     $set{"duracion":120},{multi:true})
```

C.4.2 Método findAndModify

Este método devuelve un documento que cumple una serie de condiciones y lo modifica, realizando las dos operaciones como una sola operación atómica.

Se pueden realizar las operaciones tanto "update" como "remove". En el caso de update, se puede utilizar "upsert" para indicar que en caso de que no exista ningún documento que cumpla las condiciones de la consulta, se inserte un nuevo documento con los valores indicados en el campo "update" del método.

La operación de modificación que se realice únicamente afectará al primer documento encontrado, por lo que no es posible realizar modificaciones sobre más de un documento.

El documento que devuelve esta operación es el documento original no modificado por esta operación a no ser que se le indique lo contrario.

Por ejemplo, si se quisiera modificar el número de elementos disponibles de un artículo siempre y cuando el valor de los elementos sea mayor a cero:

```
 db.articulos.findAndModify( {
    query:{"titulo":"WALL-E","formato":"dvd","disponibles":{$gt:0}},
        update:{$inc:{"disponibles":-1}}
})
```

C.4.3 Modificar si es el actual

Este patrón consiste en realizar tres operaciones a nivel de aplicación:

1. Buscar y obtener el documento.
2. Modificar el documento obtenido.
3. Realizar una operación de modificación "update()", usando como criterio de búsqueda el documento original obtenido en el punto uno, de forma que si el documento ha cambiado no se realice la modificación.

El problema con esta solución es que de nuevo no se soporta la característica todo o nada por defecto.

C.5 Administración

En esta sección se tratan temas relacionados con la administración del sistema. En concreto, se explica como realizar un volcado de la información del sistema - tanto de los datos como de los usuarios y los índices - y como restaurar los datos volcados previamente.

C.5.1 Volcado de datos

Se puede realizar volcados tanto de todas las bases de datos del servidor, de ciertas bases de datos o únicamente de ciertas colecciones.

Para realizar estas tareas se necesitan ciertos privilegios dependiendo de si se quiere realizar un volcado de toda la instancia o no:

1. read : si sólo se quiere hacer un volcado de todas las colecciones de una base datos sin las colecciones referentes a los usuarios – "system.users".
2. read y userAdmin si se quiere guardar también las colecciones system.users.

3. ReadAnyDatabase, userAdminAnyDatabase , clusterAdmin : para guardar todas las bases de datos.

Además, para poder importar datos es necesario que el usuario tenga derechos de escritura "readWrite" o "write" y "readWriteAnyDatabase" – este último es necesario si se importan todas las base de datos y las credenciales deberían estar en "admin".

Para realizar el volcado Mongodb dispone de dos herramientas "mongodump" y "mongoexport". La primera opción guarda los datos en un fichero de exportación "JSON" o "CSV"; la segunda opción guarda los datos en un fichero de exportación binario. Para recuperar los datos se utilizan las herramientas "mongorestore" y "mongoimport", dependiendo de qué sistema de exportación haya elegido al principio.

C.5.1.1 Ejemplo

Exportar con "mongodump". Desde el terminal se lanza la herramienta "mongodump" de la siguiente manera para guardar todas las bases de datos, incluida "admin":

```
> mongodump -v --username admin --password tab053
 --authenticationDatabase admin
 --out /home/user/Documentos/proyeto/dump/mongo/nombreDelFichero
```

Las opciones "—username", "–password" y "— authenticationDatabase", son análogas a las utilizadas para conectarse a la instancia mediante mongo y sirven para identificar al usuario. La opción -v hace la tarea se realice en modo "verbose" y se puedan ver las diferentes operaciones que realiza la tarea. Por último "–out" permite indicar donde se quieren guardar los datos volcados – si el directorio no existe la herramienta se encarga de crearlo.

Si por el contrarío se quiere guardar una única colección se utilizaría la opción "–collection" seguido de la opción "–db" para indicar la colección que se quiere guardar y la base de datos donde se encuentra dicha colección.

Al igual que ocurre con la herramienta "mongo", es posible conectarse a una instancia de un servidor remoto mediante las opciones "–host" y "–port" para indicar la dirección del servidor y el puerto de escucha – por defecto 27017.

C.5.2 Recuperación de datos

Importar con "mongorestore". Desde el terminal se lanza la herramienta "mongorestore" de la siguiente manera para recuperar todas las bases de datos guardadas durante el volcado de datos hecho anteriormente:

```
> mongorestore -v  --username admin --password tab053
  --authenticationDatabase admin
  /home/user/Documentos/proyeto/dump/mongo/dump/
```

El último parámetro indica el directorio donde se encuentran los datos a recuperar. Si se quiere recuperar una colección determinada se puede hacer indicando el fichero ".bson" que contiene la colección que se quiere cargar.

C.6 Pruebas

En esta sección se muestra el resultado de realizar una serie de consultas sobre los dos diseños planteados al comienzo del capítulo. Estas consultas son tanto de lectura como de escritura. Cada prueba consiste en realizar diez veces cada consulta y calcular la media total de los tiempos. No se dispone de una opción que calcule directamente los tiempos; aunque es posible activar un log que guarde esta información en un fichero, este debería se consultado cada vez que se realicen las consultas. Una alternativa simple es valerse de las sentencias javascript accesibles desde el shell de MongoDB y realzar las consultas de la siguiente manera:

```
> var a=Date.now();
    <consulta>
  var b=Date.now();b-a
```

Lo que esto hace es: primero guarda la fecha actual en una variable, después realiza la consulta, tras finalizar guarda el fecha actual en otra variable y finalmente calcula la diferencia entre las dos fechas para obtener el tiempo que se ha tardado.

Por otro lado, se repetirán las mismas pruebas dos veces, la primera vez no se utilizará ningún índice salvo el creado por defecto sobre el campo "_id"; y la segunda vez se utilizarán los índices diseñados en la sección anterior. Hay que tener en cuenta en el primer caso que las colecciones se cargarán en memoria la primera vez que se acceda, esto hace que la primera vez que cargue la colección la consulta tardará unos 4000msg para el primer diseño y uno 25000msg para el segundo diseño.En el caso de los índices además hay que tener en cuenta que los índices no se cargan automáticamente en memoria, por

lo que si dos consultas usan el mismo índice la primera tardará más que la otra, por esto, se calculará el tiempo que tarda cuando el índice no está en memoria y cuando lo está.

C.6.1 Lectura

En esta sección se muestran las instrucciones de lectura realizadas para cada uno de los diseños. Las lecturas se han realizado sobre campos simples, campos de array y campos de subdocumentos.

En las consultas en las que se utiliza el método "find()", se ha incluido el método "forEach (function())" para que itere automáticamente por todos los resultados.

C.6.1.1 Diseño 1

Las consultas de datos realizadas sobre los datos del primer diseño han sido las siguiente:

1. Buscar artículo por título:

```
> var a=Date.now();
  db.articulos.find({titulo:"WALL-E"}).forEach(function(){});
  var b=Date.now();b-a
```

2. Buscar artículo por género:

```
> var a=Date.now();
  db.articulos.find({genero:'Comedy'}).forEach(function(){});
  var b=Date.now();b-a
```

3. Buscar artículo por precio:

```
> var a=Date.now();
  db.articulos.find({tipo:"movie",
                     formato:"dvd",
                     'precio.precio':85
                     }
        ).forEach(function(){});
  var b=Date.now();b-a
```

4. Buscar artículo por autor:

```
> var a=Date.now();
  db.articulos.find(
        {'detalles.autores':'Truman Capote'}).forEach(function(){})\
    ;
  var b=Date.now();b-a
```

5. Buscar artículo por tipo:

```
> var a=Date.now();
  db.articulos.find({tipo:"movie"}).forEach(function(){});
  var b=Date.now();b-a
```

6. Buscar artículo por tipo y título:

```
> var a=Date.now();
  db.articulos.find({titulo:"WALL-E",tipo:"movie"}
        ).forEach(function(){});
  var b=Date.now();b-a
```

7. Buscar artículo por Título, tipo, formato:

```
> var a=Date.now();
  db.articulos.find({titulo:"WALL-E",tipo:"movie",formato:"dvd"}
        ).forEach(function(){});
  var b=Date.now();b-a
```

8. Buscar artículo por tipo o formato:

```
> var a=Date.now();
  db.articulosfind(
            {titulo:"WALL-E",
            $or:[{formato:"dvd"},{formato:"blueray"}]}
      ).forEach(function(){});
  var b=Date.now();b-a
```

9. Buscar artículo por Titulo o autor:

```
> var a=Date.now();
  db.articulos.find(
        {$or:[{titulo:"Truman Capote"},
        {'detalles.autores':'Truman Capote'}]}
     ).forEach(function(){});
  var b=Date.now();b-a
```

10. Buscar artículo por Formato y precio menor a algún valor:

```
> var a=Date.now();
  db.articulosfind(
        {"tipo":"movie",formato:"dvd",
        'precio.precio':{$lt:30}}
     ).forEach(function(){});
  var b=Date.now();b-a
```

11. Ordenar por precio:

```
db.articulos.find({formato:"dvd"}).sort({'precio.precio':1});
```

Esta consulta produce un error de ejecución que indica que sólo se puede realizar si hay un índice puesto que hay demasiados artículos con este formato. Una opción para evitar el error es realizar la consulta de la siguiente manera:

```
db.articulos.find({}).toArray().sort(
        function(doc1,doc2){
            return doc1.precio.precio - doc2.precio.precio})
```

Sin embargo, el tiempo que tarda en devolver los resultados es muy alto. Una consulta con menos elementos devueltos sería una en la que la condición de búsqueda es un género del artículo:

```
> var a=Date.now();
  db.articulos.find({"genero":'comedy',formato:"dvd"}
                  ).sort({'precio.precio':1}
                  ).forEach(function(){});
  var b=Date.now();b-a
```

O mediante una función de agregación:

```
> var a=Date.now();
  db.articulos.aggregate({$match:{"titulo":"WALL-E"}},
            {$sort:{'precio.precio':1}});
  var b=Date.now();b-a
```

C.6.1.2 Diseño 2

Para este diseño se ha utilizado principalmente el entorno de agregación el cual permite dividir los documentos en función de su formato, sin embargo esto no es siempre posible debido a que en ocasiones se supera el tamaño del documento BSON.Las consultas de datos realizadas sobre los datos del segundo diseño han sido las siguiente:

1. Buscar artículo por título:

```
> var a=Date.now();
  db.articulosD2.find({titulo:"WALL-E"}).forEach(function(){});
  var b=Date.now();b-a
```

```
> var a=Date.now();
  db.articulosD2.aggregate({$match:{titulo:"WALL-E"}},
                           {$unwind:"$formato"});
  var b=Date.now();b-a
```

2. Buscar artículo por género: En este caso no se puede hacer directamente con el método de agregación puesto que el resultado supera el tamaño del documento BSON.

```
> var a=Date.now();
  db.articulosD2.find({genero:'Comedy'}).forEach(function(){});
  var b=Date.now();b-a
```

3. Buscar artículo por descuento y tipo:

```
> var a=Date.now();
  db.articulosD2.aggregate(
        {$match:{tipo:"movie",
                 'formato.precio.descuento':85}});
  var b=Date.now();b-a
```

   Max:9383 AVG: 2467

4. Buscar artículo por autor:

```
> var a=Date.now();
  db.articulosD2.aggregate(
        {$match:{'detalles.autores':'Truman Capote'}},
        {$unwind:"$formato"});
  var b=Date.now();b-a
```

   AVG: 1902

5. Buscar artículo por formato:

```
> var a=Date.now();
  db.articulosD2.find({formato:"dvd"}).forEach(function(){});
  var b=Date.now();b-a
```

   Max: 14 AVG: 1467

6. Buscar artículo por tipo y título:

```
> var a=Date.now();
  db.articulosD2.aggregate(
        {$match:{tipo:"movie",titulo:"WALL-E"}},
        {$unwind:"$formato"});
  var b=Date.now();b-a
```

   Max: 1984 AVG: 1791

7. Buscar artículo por Título, tipo, formato:

```
> var a=Date.now();
  db.articulosD2.aggregate(
        {$match:{tipo:"movie",titulo:"WALL-E"}},
        {$unwind:"$formato"},
        {$match:{'formato.formato':"dvd"}});
  var b=Date.now();b-a
```

AVG: 1805

8. Buscar artículo por titulo y formato a o b:

```
> var a=Date.now();
  db.articulosD2.aggregate(
        {$match:{titulo:"WALL-E",tipo:"movie"}},
        {$unwind:"$formato"},
        {$match:{$or:[{'formato.formato':"dvd"},
                      {'formato.formato':"blueray"}]
                }});
  var b=Date.now();b-a
```

AVG: 1827

9. Buscar artículo por Titulo o autor:

```
> var a=Date.now();
   db.articulosD2.aggregate(
        {$match:
            {$or:[{titulo:"Truman Capote"},
                  {'detalles.autores':'Truman Capote'}]}},
        {$unwind:"$formato"});
  var b=Date.now();b-a
```

avg: 3173

10. Buscar artículo por Formato y precio menor a algún valor:

```
> var a=Date.now();
   db.articulosD2.aggregate(
        {$unwind:"$formato"},
        {$match:
            {'formato.formato':"dvd",
            'formato.precio.precio':{$lt:30}}});
  var b=Date.now();b-a
```

Avg:35840 Mejora:

```
> var a=Date.now();
  db.articulosD2.aggregate(
        {$match:{"tipo":"movie",
```

```
        'formato.precio.precio':{$lt:30}}},
        {$match:{genero:'Comedy'}},
        {$unwind:"$formato"},
        {$match:{formato:"dvd"}}
    );
  var b=Date.now();b-a
```

Avg:3545 db.articulosD2.ensureIndex('formato.formato':1,'formato.precio.precio':1);

11. Ordenar por precio:

```
> var a=Date.now();
  db.articulosD2.aggregate(
    {$match:{"titulo":"WALL-E"}},
    {$sort:{'formato.precio.precio':1}},
    {$unwind:"$formato"},
    {$sort:{'formato.precio.precio':1}});
 var b=Date.now();b-a
```

AVG: 1909

C.6.2 Escritura

En esta sección se muestran las instrucciones de escrituras realizadas para cada uno de los diseños. Las escrituras se han realizado sobre campos simples, campos de array y campos de subdocumentos.

En este caso, para calcular el tiempo que tarda la operación no es posible utilizar el método anterior, ya que calcula el valor de b-a mientras realiza la operación, y no al terminar. Por tanto, para calcular el tiempo hay que activar la opción de "profiling" para la base de datos contenidos mediante el método "db.setProfilingLevel(2)" y se ha comprobado el tiempo que tarda cada operación consultado la colección "db.system.profile" mediante la sentencia "show profile" tras ejecutar las consultas.

C.6.2.1 Diseño 1

Las consultas de datos realizadas sobre los datos del primer diseño han sido las siguiente:

- Añadir un documento:

```
> db.articulos.insert(
        {"titulo":"Pelicula1","formato":"dvd","tipo":"movie",
         "genero":["Comedy","Action"],
         "formato":"dvd",
```

```
        "detalles":{"duracion":34,"actores":["Tom Hanks"]},
        "precio":{"precio": 25, "descuento":50,"envio":"gratis"}
       });
```

Avg : despreciable

- Eliminar un documento:

```
> db.articulos.remove({"titulo":"Pelicula1"});
```

- Modificar el precio de los documentos donde el tipo sea pelicula:

```
> db.articulos.update(
        {"tipo":"movie","formato":"dvd"},
        { $inc : { 'precio.precio' : 1 }},
        { multi: true });
```

En este caso se ha elegido un campo que se repite a menudo para que el cálculo del tiempo sea más por la modificación que por la búsqueda.

- Modificar un documento para añadir un actor a un película de titulo "WALL-E":

```
> db.articulos.update(
        {"titulo":"WALL-E","tipo":"movie"},
        { $addToSet : { 'detalles.actores' : "Tom Hanks" }},
        { multi: true });
```

- Modificar el descuento de un artículo cuyo título sea "WALL-E" y formato "dvd":

```
> db.articulos.update(
    {"titulo":"WALL-E","formato":"dvd"},
    { $set: { 'precio.descuento' : 50 }},
    { multi: true });
```

En esta sección se muestrán las íntrucciones de escrituras realizado para cada uno de los diseños. Las escrituras se han realizado sobre campos simples, campos de array y campos de subdocumentos.

C.6.2.2 Diseño 2

Las consultas de datos realizadas sobre los datos del segundo diseño han sido las siguiente:

- Añadir un documento:

```
> db.articulosD2.insert(
        { "titulo":"Pelicula1","formato":"dvd",
          "tipo":"movie","genero":["Comedy","Action"],
          "detalles":{"duracion":34,"actores":["Tom Hanks"]},
          "formato":[
                {"precio":{"precio": 25, "descuento":50,
                           "envio":"gratis"}}
        ]});
```

- Eliminar un documento:

```
> db.articulosD2.remove({"titulo":"Pelicula1"});
```

- Modificar la duración de los documentos donde el tipo sea película:

```
> db.articulosD2.update(
        {"tipo":"movie",'formato.formato':"dvd"},
        { $inc : { "formato.$.precio.precio" : 1 }},
        { multi: true });
```

  En este caso se ha elegido un campo que se repite a menudo para que el calculo del tiempo sea más por la modificación que por la búsqueda.

- Modificar un documento para añadir un actor a un película de título "WALL-E":

```
> db.articulosD2.update(
    {"titulo":"WALL-E","tipo":"movie"},
    { $addToSet : { 'detalles.actores' : "Tom Hanks" }},
    { multi: true });
```

- Modificar el descuento de un artículo cuyo título sea "WALL-E" y formato "dvd":

```
>  db.articulos.update(
    {"titulo":"WALL-E","formato.$.formato":"dvd"},
    { $set: { "formato.$.precio.descuento" : 50 }},
    { multi: true });
```

C.6.3 Comparación

Seguidamente, se muestra una tabla con la comparación de los resultados obtenidos al realizar las pruebas:

Consulta	Tdiseño1	Tdiseño1 Index	Tdiseño2	Tdiseño2 Index
Lectura 1	2050	128/2	1764	68/2
Lectura 2	3814	5209/1092	2467	1524/388
Lectura 3	5884	361/68	3545	705/141
Lectura 4	4958	264/7	1902	576/5
Lectura 5	4164	3971	1467	1722
Lectura 6	2342	34/1	1791	23/1
Lectura 7	2397	37/2	1805	34/2
Lectura 8	2037	30/1	1827	27/2
Lectura 9	5758	615/5	3173	29/7
Lectura 10	2935	3761/331	1909	3563/386
Lectura 11	4842	132/2	2187	27/2
Escrtura 1	1	1	1	1
Escrtura 2	41976	1	25790	1
Escrtura 3	60070	70382	30436	119938
Escrtura 4	41653	1	27638	1
Escrtura 5	39643	1	45207	1

Como se puede ver el rendimiento aumenta en gran medida al crear los índices. Incluso en el caso de las escrituras, el tiempo de respuesta mejora en todos los casos excepto cuando se inserta o modifica un gran número de documento. Esto último se debe a que cuando sólo se modifica un documento la consulta se beneficia más del uso de los índices durante la búsqueda del documento, que del tiempo que se emplea en actualizar los índices con el nuevo valor. Sin embargo, cuando hay una gran cantidad de escrituras el tiempo de modificación de los índices acumulado hace que el rendimiento se vea perjudicado.

En cuanto a la diferencia de los dos diseños, el hecho de que el segundo diseño genere menos documentos le beneficia cuando no hay índices y las búsquedas se deban hacer sobre todas los documentos. Sin embargo, una vez se crean los índices esta ventaja prácticamente desaparece. Por tanto, a la hora de elegir un diseño u otro, debido a que el tamaño de los índices del esquema del primer diseño es mayor que el segundo, uno de los factores determinante será el tamaño de la memoria disponible en el sistema, ya que si estos índices no entran en memoria la mejora desaparecerá.

Apéndice D

OpenLink Virtuoso: Información Adicional

D.1 Instalación

En esta sección se muestra como instalar Virtuoso en Ubuntu y como configurar la instancia del servidor para que cumpla con nuestras necesidades. Como sistema operativo se ha utilizado la versión 12.04 de Ubuntu de 64bits. Sobre este se ha instalado la versión 6 presente en el repositorio de Ubuntu.

D.1.1 Instalación Básica

Para instalar Virtuoso opensource en Ubuntu es suficiente con obtenerlo desde el repositorio de la siguiente manera, tal y como se índica en la documentación [42]: sudo apt-get install virtuoso-opensource

Durante la instalación se solicitará la contraseña que los usuarios - dba y dav - utilizarán para identificarse.

Una vez instalado, en Ubuntu, la instancia de servidor será iniciada y se podrá acceder al sistema de varias formas, las cuales se explicarán más adelante.

Si el servidor no se iniciará automáticamente, es posible iniciarlo de la siguiente manera:

1. Acceder a la carpeta en la que se encuentra el archivo "virtuoso.ini". En el caso de este proyecto, dicha carpeta se encontraba en "/etc/virtuoso-opensource-6.1"

2. Lanzar el servidor mediante el siguiente comando:

   sudo virtuoso-t -fd

   Este comando lanza el servidor directamente en primer plano, por lo que si se cierra el terminal el servidor se detendrá. Para que se lance como un servicio en segundo plano se debe omitir la opción "-d"

D.2 Control de Usuarios

En Virtuoso existen dos niveles de seguridad a la hora de controlar el acceso de los usuarios a los datos [43]. En el primer nivel es necesario que el usuario sea miembro de alguno de los roles relacionados con SPARQL (SPARQL_SELECT, SPARQL_SPONGE or SPARQL_UPDATE) para poder realizar consultas sobre la tabla que contiene la información de los grafos. El segundo nivel permite controlar el acceso de los usuarios a cada grafo.

Seguidamente se muestra como manejar los privilegios de los usuarios para los diferentes niveles.

D.2.1 Creación de usuarios

La tarea de crear un usuario en el sistema se puede hacer tanto desde la línea de comandos como desde la interfaz web.

Desde la línea de comandos se puede realizar esta tarea mediante el comando DB.DBA. USER_CREATE(usuario, contraseña) de la siguiente manera:

```
> DB.DBA.USER_CREATE ('userA', '1234');
```

Desde la interfaz web, una vez dentro del sistema - en http://localhost:8890/conductor/ - utilizando el menú superior de la interfaz hay que navegar hasta System Admin -> User Accounts. En esta página se encuentra una tabla con la lista de usuarios presentes en el sistema. En la parte superior derecha de esta tabla se encuentra presente un link "Create New Account", el cual lleva a una página para introducir los datos del usuario a crear, así como opciones para configurar la cuenta. Por ejemplo, para crear un usuario sin privilegios como en el caso del usuario "userB", únicamente se rellenarían los campos "User Login" y "Account Name" con el nombre de usuario; y los campos "Password" y "Confirm Password" con la contraseña deseada.Finalmente se guardan los cambios haciendo click en el botón "save". El nuevo usuario debería aparecer en tabla anterior.

D.2.2 Permisos para realizar consultas SPARQL

A la hora de asignar diferentes permisos a los usuarios, existen tres opciones: asignar un rol ya creado por defecto, asignar los privilegios directamente o crear un rol con los permisos deseados.

En este caso, para permitir a los usuarios realizar consultas SPARQL, lo más sencillo es asignarles un rol como SPARQL_UPDATE el cual permite que los usuarios con dicho rol puedan realizar tanto consultas de lectura como de modificación/inserción sobre todos los grafos - si sólo se quisiera realizar consultas de lectura se podría utilizar el rol SPARQL_SELECT. De nuevo, es posible realizar dicha asignación de roles tanto mediante la línea de comandos como mediante la interfaz web.

Desde la línea de comando, para realizar la asignación del rol se haría de forma similar a sistemas RMDB como Oracle, mediante el comando GRANT:

```
> GRANT SPARQL_UPDATE to "userA";
```

Desde la interfaz web se debería navegar de nuevo hasta System Admin -> User Accounts - tal y como se haría para crear un usuario. Una vez hecho esto se seleccionaría el link "editar" correspondiente al usuario deseado. En esta página se puede editar diferente información del usuario.Por tanto, se le puede asignar un rol de dos formas:

1. En el campo "Primary Role" del formulario, seleccionar el rol SPARQL_UPDATE.
2. En la lista de seleción "Account Roles" seleccionar SPARQL_UPDATE y pulsar el botón "»" para mover el elemento a la casilla de la derecha.

Para asignar privilegios concretos que permitan utilizar las sentencias SPARQL de lectura, escritura y modificación se debería dotar a los usuarios de los siguientes privilegios:

- Inserción: "DB.DBA.SPARQL_INSERT_DICT_CONTENT"
- Eliminación: "DB.DBA.SPARQL_DELETE_DICT_CONTENT"
- Selección: "DB.DBA.SPARQL_SELECT_DICT_CONTENT"
- Modificación: "DB.DBA.SPARQL_UPDATR_DICT_CONTENT"

Conviene destacar que mientras es posible borrar tripletas de los grafos con el privilegio DELETE, no es posible realizar operaciones "DROP" y "CLEAR" para borrar todo el grafo, para esto es necesario tener el rol SPARQL_UPDATE.

Estos permisos se pueden asignar directamente a un usuario creado previamente, o pueden ser dados a un rol que más adelante será asignado al usuario.

Por ejemplo, si se quieren modificar los permisos que tiene el rol SPARQL se debería hacer lo siguiente:

```
> GRANT EXECUTE ON DB.DBA.SPARQL_INSERT_DICT_CONTENT TO "SPARQL";
```

D.2.3 Permisos sobre grafos

Una vez obtenidos los permisos para realizar consultas, el usuario podrá realizar por defecto consultas sobre cualquier grafo. Esto se debe a que existe un rol llamado "nobody" el cual tiene permisos sobre cualquier grafo y se asigna a todos los usuarios por defecto. Para limitar el acceso por defecto a los grafos se puede realizar la consulta:

```
> DB.DBA.RDF_DEFAULT_USER_PERMS_SET ('nobody', 0);
```

En está consulta el cero indica que "nobody" no tiene derechos de escritura ni de lectura sobre ningún grafo - el 1 indicaría que se tiene derechos de lectura, el dos que se tiene derechos de escritura, el tres que se tienen ambos derechos y el 1023 que se tiene todos los privilegios posibles que da el sistema.

Tras realizar esta consulta los usuarios creados no tendrán permisos sobre ningún grafo, esto hace que puedan ejecutar consultas pero que no obtendrá ningún resultado - es decir, no se produce un error. Hay que destacar, que el usuario dba seguirá teniendo permisos sobre todos los grafos si se realizan las consultas desde la linea de comandos; sin embargo, no podrá realizar consultas desde la interfaz web. Para solucionar esto se deberían dar permisos sobre los grafos al rol SPARQL.

Si en lugar de limitar los permisos de todos los usuarios, únicamente se quieren limitar los permisos de un usuario sólo sería necesario sustituir "nobody" por el nombre del usuario.

Una vez hecho esto, se puede indicar a qué grafos puede tener acceso cada usuario y que permisos tiene el usuario sobre dicho grafo. Para esto se utiliza el método DB.DBA.RDF_GRAPH_USER_PERMS_SET (grafo, usuario, permisos). Por ejemplo para indicar que el usuario "userA" tiene permisos solo de lectura sobre el grafo http://mygraph.com:

```
> DB.DBA.RDF_GRAPH_USER_PERMS_SET ('http://mygraph.com', 'userA', 1);
```

Como en el caso anterior para indicar que se le permite escribir sobre el grafo se utilizaría un 2 y para realizar escrituras y lecturas un 3.

Además de asignar permisos grafo a grafo, es posible crear grupos de grafos y asignar permisos sobre ese grupo de la siguiente forma:

1. Crear un nuevo grupo con los grafos 'http://mygraph.com' y 'http://mygraph2.com' - suponiendo que existan, si no es así es suficiente con insertar una tripleta en cada grafo.

Para ver los privilegios sobre los grafos rdf se pueden consultar las siguientes tablas del sistema:

1. DB.DBA.RDF_GRAPH_USER : esta tabla muestra el identificador del grafo, el identificador del usuario y el permiso que tiene asignado.
2. DB.DBA.SYS_USERS : en esta tabla hay información sobre todos los usuarios, en concreto el campo U_ID y U_NAME permite comprobar a que usuario corresponde el identificador de la tabla anterior.

Para conocer a qué grafo corresponde el identificador de la primera tabla se puede usar el método "id_to_iri()". Por tanto es posible conocer que permisos tiene cada usuario para cada tabla realizando la siguiente consulta SQL - dicha consulta se puede realizar desde la línea de comandos o desde la interfaz web "Database - Interactiva SQL" :

```
> SELECT id_to_iri(RGU_GRAPH_IID), RGU_USER_ID, U_NAME, RGU_PERMISSIONS
  FROM RDF_GRAPH_USER, SYS_USERS
  WHERE RDF_GRAPH_USER.RGU_USER_ID = SYS_USERS.U_ID
```

D.3 Concurrencia

El sistema principal de cerrojos de Virtuoso es a nivel de fila, tripleta, es decir, cada consulta sólo bloqueará una única fila. Sin embargo, este sistema es dinámico lo cual permite escalar el bloqueo a nivel de página - páginas de la base de datos en cache - si el sistema lo ve necesario. Dicho bloqueo se produce cuando se producen lecturas largas. El bloque de la página se hace únicamente si no existen otras consultas sobre dicha página, éstas consultas pueden ser tanto de lectura como de escritura. Este bloqueo puede además ser tanto compartido como exclusivo, es decir, que el bloqueo permite que otros hilos realicen bloqueos, siempre en modo compartido, o que únicamente el hilo que bloquea la tripleta sea capaz de acceder a esa tripleta. [44]

Es posible obtener información sobre los bloqueos que se están produciendo mediante el comando "status()". Este muestra información general sobre el estado del servidor

incluyendo un apartado sobre los cerrojos y los interbloqueos que se puedan estar produciendo. Existen seis apartados principales, (1) Database Status que da información general sobre el estado de las conexiones al servidor; (2) Lock Status que indica si existe algún interbloqueo y de que tipo; (3) Pending que indica los bloqueos que se están solicitando; (5) Client <Puerto>:ID que da información general del usuario y sobre que cerrojos a obtenido dicho client; y (6) Running Statements que indica que consultas o funciones se están ejecutando en ese momento. Por tanto, los apartados más interesantes para ver información sobre los cerrojos son Lock Status, Pending y Client.

El apartado Lock status indica el número de interbloqueos, deadlocks, que existen en ese momento. También que número de los interbloqueos son de tipo 2r1w que se refiere a los interbloqueos donde varios hilos quieren leer una misma página al mismo tiempo que un tercer hilo quiere obtener derecho de escritura sobre dicha página. Además de esto también se muestran el número de hilos en ejecución y el número de hilos en espera.

El apartado pending indica los cerrojos que los hilos están solicitando y muestra los datos de la solicitud con la forma <Id>: I[E|S][R|P] <num>, donde Id indica el identificador de cerrojo, [E|S] indica si se trata de una solicitud de bloqueo en exclusiva, E, o una solicitud de bloqueo compartido S, [R|P] indica si el bloqueo se hace sobre una fila, R, o sobre una página, P; y por último el número num indica que cliente está solicitando el cerrojo. Alternativamente a este número puede aparecer "INTERNAL" que indica que la solicitud viene la respuesta de un log.

El apartado Cliente va seguido del puerto desde el cual el cliente está conectado y número de identificación que se le ha dado al cliente para esa sesión, este es el mismo número que se utiliza para identificar a quien pertenece la solicitud de cerrojo. Este apartado contiene información sobre el estado de la transacción que está realizando el cliente, el estado puede ser PENDING para indicar que todo es correcto - esto no indica que la transacción está pendiente ya que puede aparecer como PENDING aunque el cliente no esté realizando ninguna transacción - o BLOWN OFF o DELTA ROLLED BACK para indicar que se ha cancelado la transacción bien porque ha habido un interbloqueo o porque se ha superado el tiempo de espera. Además, este apartado también incluye información sobre que cerrojos ha adquirido el cliente y se presentan con la forma <ID>: I[E|S] donde al igual que en el caso de las solicitudes E indica que el bloqueo es en exclusiva mientras que S indica que el bloqueo es compartido.

Por otro lado, también es posible ver información general sobre el número de interbloqueos y bloqueos que se están produciendo en cada tabla, mediante la información disponible en la tabla del sistema sys_l_stat. Para ver esta información se puede realizar la siguiente consulta SQL:

```
SELECT TOP 5 * FROM sys_l_stat ORDER BY DEADLOCKS , waits DESC;
```

Esta consulta devuelve, entre otros, los valores de los campos KEY_TABLE, INDEX_NAME , LOCKS, WAITS y DEADLOCKS. Estos campos indican la tabla donde se producen los bloqueos - la tabla DB.DBA.RDF_QUAD es la que contiene los grafos-, el nombre del índice de la tabla, el número de cerrojos adquiridos para cada índice de la tabla, el número de veces que un cursor tuvo que esperar por un cerrojo, el número de cerrojos. Esta información es muy general y sólo permite ver el número de cerrojos e interbloqueos sobre la tabla DB.DBA.RDF_QUAD sin concretar esta información a nivel de grafo.

Alternativamente, se puede acceder a la información de la tabla "sys_l_stat" y a parte de la información de los clientes - la que aparece al ejecutar la función status() - mediante la interfaz web. Para esto hay que ir a la zona de administración de la interfaz web - conductor - y desde allí: (1) navegar a System Admin -> Monitor -> Locks para ver la información de la tabla "sys_l_stat"; (2) navegar a System Admin -> Monitor -> Client Sessions para ver la información de los clientes.

D.4 Transacciones

Virtuoso soporta el principio ACID tanto para su sistema de bases de datos relacional como para el sistema de RDF. En lo que al nivel de aislamiento se refiere soporta los cuatro principales:

- READ COMMITTED: sólo permite leer datos que han sido confirmados en la base de datos.
- READ UNCOMMITTED: permite leer datos que no han sido confirmados en la base de datos.
- REPEATABLE READ: permite que los datos leídos al final de una transacción sean los mismo que se leyeron al comienzo de ésta.
- SERIALIZABLE: tiene todas las restricciones presentes en los casos anteriores; pero además evita que se lea valores nuevos que no existían antes de comenzar la transacción.

Algo que hay que tener en cuenta a la hora de utilizar los distintos niveles de aislamiento en un sistema RDF es que por ejemplo al realizar operaciones de modificación en realidad se están realizando operaciones de eliminación e inserción por lo que si un usuario quiere realizar lecturas con un nivel de aislamiento REPEATABLE READ, al realizar una

segunda lectura tras una modificación de otro usuario,el primer usuario no verá los datos ya que habrán sido borrados, por tanto un aislamiento más adecuado para ésta situación sería SERIALIZABLE. [45]

Por otro lado, Virtuoso, al igual que ocurre con el sistema MySQL, viene por defecto con una opción para confirmar de forma automática - autocommit - las modificaciones que se hagan en las bases de datos. Para desactivar esta opción se utiliza la función log_enable(), bien desde isql llamando a la función con el parámetro 2 - log_enable(2) - o bien dentro de las consultas SPARQL mediante la claúsula "define sql:log_enable 2" al comienzo de la consulta. En el primer caso el autocommit permanecerá desactivado hasta que no se llame a la función con el parámetro 3, pero en el segundo caso el autocommit sólo estará desactivado durante la ejecución de esa consulta.

Pese a que soporta los niveles de aislamiento anteriores para el sistema de grafos, no es posible asignar directamente los niveles desde SPARQL o utilizando la opción "set isolation := level;" desde isql - a diferencia de lo que ocurre con el sistema relacional normal de Virtuoso que permite asignar el nivel de aislamiento desde SQL.

Por tanto, para poder asignar el nivel de aislamiento existen tres opciones:

- La primera opción es establecer el nivel de aislamiento de forma global, modificando el archivo Virtuoso.ini para incluir la opción "DefaultIsolation = num ", donde num hace referencia al identificador del nivel de aislamiento que se quiere elegir; siendo estos 1, 2, 3, 4 para referirse a READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ y SERIALIZABLE respectivamente. Este método requiere que se reinicie el servidor cada vez que se quiera modificar el nivel de aislamiento y debe ser el mismo para todas las transacciones.

- La segunda opción es asignar el nivel de aislamiento a nivel de la aplicación, es decir que si se están realizando consultas mediante una aplicación externa sea esta aplicación la que se encargue de indicar el nivel de aislamiento. Por ejemplo, si una aplicación se está conectando a Virtuoso mediante el driver JDBC se puede usar la sentencia "conn.setTransactionIsolation (java.sql.Connection.TRANSACTION_SERIALIZABLE)" para indicar al servidor que se quiere utilizar un nivel de aislamiento SERIALIZABLE.

- La tercera opción sería crear un procedimiento o una función en Virtuoso que incluyeran las consultas en SPARQL que se quisieran realizar e indicar el nivel de aislamiento con la sentencia "set isolation := level;" y utilizar las sentencias "commit work" y "rollback work" para confirmar los cambios o cancelarlos.

D.5 Administración

En esta sección se tratan temas relacionados con la administración del sistema. En concreto, se explica como mejorar el rendimiento y como realizar un volcado de la información del sistema - tanto de los datos como de los usuarios y los índices - y como restaurar los datos volcados previamente.

D.5.1 Mejora de rendimiento

Para mejorar el rendimiento del sistema, en especial con el uso de datasets de gran tamaño se pueden realizar una serie de configuraciones:

- Configuración del número de buffers: lo recomendable es configuarar el número de buffers de forma que los procesos usen entre dos tercios y tres quintos de la memoría disponible. Para configurar esta opción es necesario modificar en el fichero virtuodo.ini - el cual se encuentra en - los campos NumberOfBuffers y MaxDirtyBuffers para que contengan los valores más adecuados. Por ejemplo, para un sistema de 4G se podría configurar los campos anteriores con 340000 y 250000; o si se tuviera una memoria de 8G se podrían configurar con 680000 y 500000.
- Eliminación de grafors con gran número de tripletas: la operación de eliminación de una tripleta forma parte de una transacción y por tanto es guardada en memoria, si se elimina un grafo conteniendo gran cantidad de tripletas se puede consumir toda la memoria haciendo que no sea posible eliminar el grafo. Para evitar esto se debería cambiar modo log a autocommit mediante el comando log_enable(3,1) antes de realizar la eliminación del grafo - o la eliminación de un gran número de tripletas. [46]

D.5.2 Dump y load de datos RDF

En Virtuoso es posible realizar backups de todo el sistema mediante la función "backup_online() " o simplemente copiando el archivo database y su archivo log asociado. Sin embargo, es más probable que se quiera realizar un volcado y carga de uno o varios grafos que contengan los datos que nos interesan. Seguidamente se muestran los métodos que se pueden seguir para realizar dicha tarea.

D.5.2.1 Dump

No existe ninguna función o método por defecto en Virtuoso para realizar dicha tarea. Sin embargo, se proporcionan funciones como DB.DBA.RDF_TRIPLES_TO_TTL(inout tripleta any , inout ses any) y DB.DBA.RDF_TRIPLES_TO_RDF_XML_TEXT(inout triples any , in print_top_level any , inout ses any) las cuales permiten transformar las tripletas de entrada a una lista de datos en formato Turtle o RDF/XML respectivamente; y aunque estas funciones no permiten guardar todas las tripletas de un grafo directamente a un fichero, sería posible crear una función o un método que realizase una consulta de lectura sobre el grafo que se quisiera guardar y al tiempo que obtuviera las tripletas las convirtiera utilizando las funciones anteriores al formato deseado y finalmente fuera guardando los stream de datos en un fichero.

En la documentación oficial se muestra como crear una función llamada dump_one_graph() [47], la cual sigue el método anterior para guardar las tripletas de un grafo en formato Turtle, salvo que en lugar de utilizar las funciones anteriores, utiliza la función http_ttl_triple (in env , in s ,in p, in o, ses). Esta función realiza una tarea similar a las comentadas anteriormente.

Por tanto, para realizar el volcado de un grafo a un archivo se podrían seguir los siguientes pasos:

1. Crear la función dump_one_graph() - si no lo está ya - copiando el script presente en el anexo o en la referencia bibliográfica, en la linea de comandos isql, y ejecutándolo.
2. Lamar a la función indicando qué grafo se quiere guardar, el directorio donde se guardarán los archivos y el tamaño máximo del fichero. Por ejemplo para guardar el grafo "http://appl.com/" se llamaría a la función de la siguiente manera: dump_one_graph ('http://appl.com/','./data_', 1000000000);
3. Comprobar que se han creado los ficheros con nombre data_XXXXX.ttl. Puesto que se ha indicado el directorio actual - "./" - los archivos deberían estar en el mismo directorio que el archivo de la base de datos y su log - virtuoso.db y virtuoso.log. En el caso de Ubuntu dicho directorio es /var/lib/virtuoso-opensource-6.1/db.

D.5.2.2 Load

Para cargar los ficheros con los datos generados durante un volcado se pueden utilizar funciones como DB.DBA.TTLP/_MT(in fichero, in nombre del grafo) y DB.DBA.RDF_LOAD_RDFXML/_MT(in fichero, in nombre del grafo) para cargar ficheros en fromato

TURTEL y RDF/XML respectivamente - la diferencia entre las funciones que acaban en _MT y las que no es que las primeras realizan la insección con múltiples hilos.

Por ejemplo, para cargar un grafo generado con el método de volcado del apartado anterior sería suficiente con ejecutar la función anterior desde isql de la siguiente forma :

```
DB.DBA.TTLP_MT(
    file_to_string_output(
        '/var/lib/virtuoso-opensource-6.1/db/data_000001.ttl'),
    'http://familia.com/');
```

Sin embargo, si el grafo que se ha volcado es muy largo, lo más probable es que haya generado más un fichero con lo que será necesario repetir la operación anterior con todos los ficheros.

De nuevo, en la documentación oficial, [48] , se muestra un script que utiliza DB.DBA.TTLP_MT para cargar todos los ficheros generados por la función dump_one_graph(), partiendo únicamente del directorio donde se encuentran dichos ficheros.

Para realizar la carga del grafo generado en la sección anterior, se podrían seguir los siguientes pasos:

1. Crear la función load_graph() - si no lo está ya - copiando el script presente en el anexo, en la línea de comandos isql, y ejecutándolo.
2. Llamar a la función indicando el directorio en el que se encuentran los ficheros a cargar. Para cargar los ficheros generados en la sección anterior se llamaría a la función de la siguiente forma: load_graphs('/var/lib/virtuoso-opensource-6.1/db/'); No es necesario indicar el nombre del grafo ya que dump_one_graph() genera un fichero llamado data_XXX.ttl.graph que contiene el nombre del grafo.

D.6 Pruebas

En esta sección se muestra el rendimiento del sistema al realizar una serie de consultas tanto de lectura como de escritura. Cada prueba consiste en realizar diez veces cada consulta y calcular la media de los resultados obtenidos. Desde la interfaz no es posible calcular el tiempo que tarda en realizarse una consulta. Sin embargo, desde la línea de comandos isql, el tiempo que el sistema tarda en completar una operación, siempre aparece por defecto.

Puesto que los índices por defecto son bastante eficientes y no se van a borrar grafos con pocos datos de forma repetida, no se modificarán estos índice y las pruebas se realizarán únicamente una vez con los índices por defecto.

D.6.1 Lectura

En esta sección se muestran las consultas de lecturas realizadas sobre el grafo <http://appl.com7>.

Estas consultas incluyen una serie de prefijos que se muestran a continuación:

```
PREFIX does:<http://appl.com/actions/>
PREFIX  article:<http://appl.com/art/article/>
PREFIX  art:<http://appl.com/art/>
PREFIX  rev:<http://appl.com/rev/>
PREFIX  buy:<http://appl.com/purchase/buy/>
PREFIX  review:<http://appl.com/rev/review/>
```

A continuación se muestran las consultas realizadas para las pruebas, pero sin los prefijos anteriores para simplificar la documentación de estas consultas:

- Buscar las compras que realiza un usuario junto a la información del articulo:

```
SELECT *
FROM <http://appl.com/>
WHERE{
    <mailto:user1@email.com> does:buying ?o .
     ?o ?p ?o2 .
     ?o2 ?d ?q}
```

  Avg nuevo usuario: 605msg Avg: 40msg

- Buscar las valoraciones de un artículo, junto a la información del usuario:

```
SELECT *
FROM <http://appl.com/>
WHERE{
?a article:id "51f880ed44aefebc04debd9f" .
 ?r review:of ?a.
 ?r ?p ?o .
 ?u does:reviewing ?r .
 ?u ?v ?w}
```

  Avg nuevo articulo: 436msg Avg: 130msg

- Busca artículos que otros usuarios han valorado siempre que los usuarios hayan valorado un artículo dado:

```
SELECT *
FROM <http://appl.com/>
WHERE{
 ?a article:id "51f880ed44aefebc04debd9f" .
 ?r review:of ?a.
 ?r ?p ?o .
 ?u does:reviewing ?r .
 ?u ?v ?w}
```

Avg con un nuevo artículo: 350msg Avg: 4msg

- Busca articulos valorados por otros usuarios que han valorado los artículos de un usuario dado:

```
SELECT *
FROM <http://appl.com/>
WHERE{
    <mailto:user134@email.com> does:reviewing ?r .
    ?r review:of ?a .
     ?b  review:of ?a .
    FILTER(?r != ?b) .
    ?b review:rating ?rating .
    FILTER(xsd:int(?rating) >6) .
     ?u  does:reviewing ?b .
    FILTER(?u != <mailto:user134@email.com>).
    ?u  does:reviewing ?l .
    FILTER(?b != ?l) .
    ?l review:rating ?rating2 .
    FILTER(xsd:int(?rating2) >8) .
    ?l review:of ?a2 .
    ?a2 article:id ?id .
}ORDER BY DESC(xsd:int(?rating))
```

Avg con un nuevo artículo: 622msg Avg: 11msg

- Contar el número totales de elementos en el grafo:

```
select COUNT(*)
from <http://appl.com/>
where {?s ?p ?o}
```

Avg: 130589 msg

D.6.2 Escritura

En esta sección se muestran las consultas de escritura realizadas sobre el grafo <http://appl2.com> - se utiliza un grafo secundario para no modificar el dataset original. Estas consultas incluyen una serie de prefijos que se muestran a continuación:

```
PREFIX does:<http://appl.com/actions/>
PREFIX  article:<http://appl.com/art/article/>
PREFIX  art:<http://appl.com/art/>
PREFIX  rev:<http://appl.com/rev/>
PREFIX  buy:<http://appl.com/purchase/buy/>
PREFIX  review:<http://appl.com/rev/review/>
```

A continuación se muestran las consultas realizadas para las pruebas, pero sin los prefijos anteriores para simplificar la documentación de estas consultas:

- Insertar una valoración nueva:

```
INSERT INTO <http://appl2.com/> {
  rev:aaaa review:of ?a.
  rev:aaaa review:rating 7.
  rev:aaaa review:descripcion "not bad".
  rev:aaaa review:formato "dvd".
   <mailto:userA@email.com> does:buying rev:aaa}
WHERE {
  GRAPH  <http://appl.com/>
    { ?a article:id "51f880ed44aefebc04debd9f"
     }
 }
```

  Avg nuevo usuario: 605msg Avg: 40msg

- Crea la relación conoce entre el usuario B y los usuarios que hayan comprado el articulo 2123:

```
INSERT INTO <http://appl2.com/> {<mailto:userB@email.com> foaf:\
    knows ?v}
WHERE {
    GRAPH  <http://appl.com/>
        { ?v <http://appl.com/actions/buying> ?b .
          ?b <http://appl.com/purchase/buy/article>
                            <http://appl.com/art/2123>
        }
  }
```

  Avg: 29msg

- Cambiar el nombre de los usuarios que conocen al usuario user124@email.com:

```
MODIFY GRAPH <http://appl2.com/>
DELETE { <mailto:userB@email.com>
            foaf:knows  <mailto:userC@email.com> }
INSERT { <mailto:userB@email.com>
            foaf:knows ?o }
FROM <http://appl.com/>
WHERE { ?s foaf:knows ?o }
```

Avg: 79msg

- Borrar al usuario C:

```
DELETE FROM <http://appl2.com/>
{<mailto:userB@email.com> ?v ?o}
WHERE
{<mailto:userB@email.com> ?v ?o}
```

Avg: 15302msg

Apéndice E

Lista de entregables

En éste apéndice se incluye la lista de todos los entregables junto con un link para su descarga.

E.1 Entregables relacionados con el entorno de trabajo

Los entregables relacionados con el entorno, se refiere a todos aquellos entregables que tienen que ver con la parte de la documentación que hace referencia al funcionamiento de los sistemas NoSQL, como es el documento del diseño por separado, los *dataset* utilizados en el diseño e instancias a los dos sistemas NoSQL utilizados.

E.1.0.1 Documento del entrono

El documento del diseño del entorno de trabajo se puede encontrar por separado en el siguiente enlace:

https://www.dropbox.com/s/4qs24f253pjaj8d/dise%C3%B1oEntorno.pdf

E.1.0.2 MongoDB

Por un lado, se ofrece una instancia al sistema MongoDB a la que se puede acceder desde un cliente "mongo" de forma remota - desde la red de la universidad - con el nombre de usuario "app" y la contraseña "tab053" de la siguiente forma:

```
> sudo mongo -u app -p tab053 --authenticationDatabase contenidos
--host 158.227.115.72
```

El cliente "mongo" se puede descargar desde la página de MongoDB junto con el resto del sistema, este se puede usar directamente al descomprimir la carpeta sin instalar el sistema. También se instala automáticamente al instalar el sistema completo, tal y como se muestra en el apéndice C.1.

En lo referente al dataset, éste se puede encontrar en el siguiente enlace:

https://www.dropbox.com/s/lpeuznq1svr8htj/dumpMongo.zip

E.1.0.3 Virtuoso

Por un lado se ofrece una instancia al sistema OpenLink Virtuoso a la que se puede acceder de dos formas, tal y como se indica en la sección del funcionamiento de Virtuoso 2.2.

La primera forma es acceder al sistema mediante el interfaz web, identificándose con el nombre de usuario "dba" y la contraseña "tab053" en la siguiente dirección:

http://158.227.115.72:8891/conductor/

Como altenartiva, también es posible acceder a un servidor externo en la siguiente dirección:

http://162.243.43.244:8890/conductor/

Desde aquí se tiene acceso a todo el sistema y para realizar consultas sobre el almacenamiento de tripletas es posible usar la herramienta "Interactive SQL" situada en la barra lateral de interfaz - para las consultas SPARQL es necesario indicar la palabra clave "SPARQL" al comienzo de la consulta -; o desde las pestañas de la barra de navegación ir a "Linked Data" - no es necesario indicar la palabra clave "SPARQL" al comienzo de la consulta -.

Finalmente es posible acceder a un segundo sistema con los datos de identificación "appl" y la contraseña "tab053" desde la página:

http://158.227.115.72:8890/conductor/

Sin embargo, en este caso sólo se tienen permisos para realizar consultas sobre el grafo "http://appl.com/".

Para acceder desde la línea de comandos es necesario tener instalado el cliente "isql" que viene en el paquete de instalación de Virtuoso. Para acceder al sistema mediante este cliente es suficiente con ejecutar:

```
>  /usr/bin/isql-vt -H 158.227.115.72 -S 1112 -U dba -P tab053
```

O en el servidor externo:

```
>  /usr/bin/isql-vt -H 162.243.43.244 -S 1111 -U dba -P tab053
```

Para acceder a la instancia alternativa en 158.227.115.72 es suficiente con cambiar el valor de la opción "-S" por "1111" e indicar el nombre de usuario "appl" y contraseña "tab053".

En lo que se refiere a los dataset, se puede encontrar: una copia de los datos del grafo usado en los ejemplos de funcionamiento del entorno "http://appl.com/"; una copia del grafo usado en la aplicación "http://applV.com/" - este es idéntico al anterior salvo que cambia el nombre del grafo y por tanto de los subgrafos-; y una copia de todo el sistema.

1. Grafo appl:
   https://www.dropbox.com/s/bo8rtc611kshpqx/virtuosoDumpDise%C3%B1o.zip
2. Grafo applV:
   https://www.dropbox.com/s/zwo31rqk1alnv7f/dumpVirtuosoApplV.zip
3. Copia del sistema:
   https://www.dropbox.com/s/i0l4lrurqc4k4xa/virtuoso.db.tar.xz

E.2 Entregables de la aplicación políglota

Los entregables de la aplicación políglota hacen referencia principalmente a una dirección para acceder a la aplicación en funcionamiento, el archivo de ejecución listo para ser desplegado, el código fuente de la aplicación y en este caso la documentación de las clases en forma de "Javadoc". Además, se incluye el dataset de la tabla de inventario en MySQL.

E.2.1 Acceso a la aplicación

Para acceder a la aplicación es suficiente acceder - desde la red de la universidad - con un navegador a la página:

http://158.227.115.72:8787/PoliglotWebapp/

La aplicación puede ser accedida desde navegadores como *Mozilla Firefox* y *Google Chrome*; sin embargo, se desaconseja su uso con *Interner Explorer*.

Por otro lado, como usuario por defecto se pueden usar los usuarios ya creados con nombre "userX@email.com" donde X es un número entre 1 a 450000 - si bien existe la posibilidad de que algunos de los usuario entre user1 y user450000 no exista - y con la contraseña "tab053".

E.2.2 Ejecutable

El archivo listo para ser desplegado se encuentra en el siguiente enlace:

https://www.dropbox.com/s/71w6oyc6cedz49y/PoliglotWebapp.war

E.2.3 Código Fuente y Javadoc

Puesto que el desarrollo de la aplicación se ha dividido en dos partes - Servicios y Aplicación Principal - se entrega el código fuente y *Javadoc* para ambas partes.

E.2.3.1 Servicios

Por un lado, el código fuente de la implementación de los servicios se puede encontrar en el siguiente enlace:

https://www.dropbox.com/s/6nv5q9yu0te613m/Services.zip

Mientras que la documentación de las clases en forma de *Javadoc* se puede encontrar en el siguiente enlace:

https://www.dropbox.com/s/96epy3htdm8knnn/ServicesJavaDoc.zip

Hay que destacar que los servicios tiene como dependencias las librerías:

- Driver de MySQL:
  http://dev.mysql.com/downloads/file.php?id=450948

- Driver de Virtuoso :
  http://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/VOSDownload/virtjdbc4.jar
- Driver de MongoDB :
  http://central.maven.org/maven2/org/mongodb/mongo-java-driver/2.9.3/
- Librería gson de Google :
  https://code.google.com/p/google-gson/downloads/detail?name=google-gson-2.2.4-release.zip&can=2&q=

E.2.3.2 Aplicación Principal

Por un lado, el código fuente de la implementación de la aplicación principal se puede encontrar en el siguiente enlace:

https://www.dropbox.com/s/xnnksppdie6k4u9/AplicacionPrinc.zip

Mientras que la documentación de las clases en forma de *Javadoc* se puede encontrar en el siguiente enlace:

https://www.dropbox.com/s/e5c0f038x2v6s83/AplicacionPrinJavaDoc.zip

Como dependencias, además de las librerías mencionadas anteriormente, también utiliza la librería *json-simple*: http://code.google.com/p/json-simple/downloads/detail?name=json-simple-1.1.1.jar&can=2&q=

E.2.4 Tabla de Inventario

En el siguiente enlace se puede descargar el dataset que permite crear la tabla con los datos del inventario sobre la base de datos que usa la aplicación - en este caso - appl:

https://www.dropbox.com/s/gc2cn572ynalsxk/inventario.sql

Para importar el esquema es suficiente con ir a "phpMyAdmin" - tal y como se hacía en el manual de configuración - y allí ir a la pestaña "Importar" de la barra de navegación. Para importar el archivo es suficiente con seleccionar el botón "examinar" y elegir el archivo dado. Una vez pulsado "continuar" se generará el esquema.

E.3 Entregables de la aplicación relacional

Los entregables de la aplicación relacional hacen referencia principalmente a una dirección para acceder a la aplicación en funcionamiento, el archivo de ejecución listo para ser desplegado y el código fuente de los servicios. Además, también se entregan los dataset de las tablas que forman la base de datos usada por la aplicación.

E.3.1 Acceso a la aplicación

Para acceder a la aplicación es suficiente acceder - desde la red de la universidad - con un navegador a la página:

http://158.227.115.72:8787/MySqlWebapp_1/

La aplicación puede ser accedida desde navegadores como *Mozilla Firefox* y *Google Chrome*; sin embargo, se desaconseja su uso con *Interner Explorer*.

Por otro lado, como usuario por defecto se pueden usar los usuarios ya creados con nombre "userX@email.com" donde X es un número entre 1 a 450000 - si bien existe la posibilidad de que algunos de los usuario entre user1 y user450000 no exista- y con la contraseña "tab053".

E.3.2 Ejecutable

El archivo listo para ser desplegado se encuentra en el siguiente enlace:

https://www.dropbox.com/s/sth4vk676nkux2v/MySqlWebapp_1.war

E.3.3 Código Fuente y Javadoc

En este caso puesto que la mayor parte de la aplicación principal es similar a la de la aplicación políglota sólo se da el código fuente de los servicios. Este puede ser encontrado en el siguiente enlace:

https://www.dropbox.com/s/xx41bzk5rhuxixv/ServiciosRelacional.zip

E.3.4 Datos MySQL

Por un lado se entrega un archivo para crear el esquema de la base de datos "applMysql" que usará la aplicación.

https://www.dropbox.com/s/90ly04z0n8sj3vu/applMysql.sql

Para importar el esquema es suficiente con ir a "phpMyAdmin" - tal y como se hacía en el manual de configuración - y ahí ir a la pestaña "Importar" de la barra de navegación. Para importar el archivo es suficiente con seleccionar el botón "examinar" y elegir el archivo dado. Una vez pulsado "continuar" se generará el esquema.

Por otro lado se entregan los dataset de cada una de las tablas:

https://www.dropbox.com/s/cnlk5hb4pl6yrcc/Mysqldump.zip

Para importar los archivos primero hay que acceder a Mysql de la siguiente forma:

```
> mysql -h localhost -u root -p --local-infile
```

Y una vez identificado, por cada archivo añadir los datos a su tabla correspondiente de la siguiente forma:

```
> LOAD DATA LOCAL INFILE '/home/user/Escritorio/nombreArchivo.csv'
  INTO TABLE nombreTabla FIELDS TERMINATED BY ',';
```

El archivo comprimido anterior también contiene el archivo *MYsqlDumpLOad.txt* que incluye información de como volcar los datos. Además de las tablas del esquema, se incluye la tabla *codId* que contiene los datos de correlación entre el identificador del artículo en la aplicación políglota y el identificador del mismo artículo en la aplicación relacional - sólo se usa para generar las preferencias de los usuarios en archivos .CSV -.

E.4 Otros

Además de los entregables anteriores también se entrega el código fuente de las aplicaciones usadas para general los datos de las dos aplicaciones.

- Código fuente de la aplicación para generar los documentos de los artículos en MongoDB:
  https://www.dropbox.com/s/lvm3t19nh2bne5d/generarArticulosMongo.zip

- Código fuente de la aplicación para generar el grafo de los usuarios y sus preferencias en Virtuoso:
  https://www.dropbox.com/s/rrmroyj8a7nw1kg/generarUsersVirtuoso.zip

- Código fuente de la aplicación para generar una tabla con la información de la disponibilidad de cada artículo:
  https://www.dropbox.com/s/931u7vacdmk9aky/generalInventarioMy.zip

- Código fuente de la aplicación para guardar la información de los artículo en un archivo CSV:
  https://www.dropbox.com/s/wrvwo5pmtb4er92/generarArticulosMySQL.zip

- Código fuente de la aplicación para guardar la información de usuarios y sus preferencias en un archivo CSV:
  https://www.dropbox.com/s/wh0kop8nharr6j4/generarUserPrefMysql.zip

  Esta aplicación hace uso de la información de correlación entre identificadores, por lo que debe haber una base de datos en MySQL llamada "cod" con una tabla llamada "tabla" que contenga dos campos llamados "_id" que es el identificador dado por MongoDB a un artículo; y "numId" que es el identificador dado a ese artículo en MySQL. Los datos de esta tabla son los generados en el archivo *codId* durante la generación de los archivos CSV de los artículos.

Ademas de los drivers de cada sistema, las aplicaciones que usan *Freebase* también usan una serie de librerías:

- Librerias de google-api-java-client como *google-api-client*, *google-oauth-client-java*, *httpclient*, *httpcore*, y *google-api-services-freebase*:
  http://repo1.maven.org/maven2/com/google/apis/google-api-services-freebase/v1-rev39-1.15.0-rc/google-api-services-freebase-v1-rev39-1.15.0-rc.jar

- Librería *commons-lang*:
  http://ftp.cixug.es/apache//commons/lang/binaries/commons-lang-2.6-bin.zip

- Librería *json-path*:
  http://mirrors.ibiblio.org/maven2/com/jayway/jsonpath/json-path/0.8.1/json-path-0.8.1-javadoc.jar

- Librería *json-smart*:
  http://code.google.com/p/json-smart/downloads/detail?name=json-smart-2.0-RC3-bundle.zip&can=2&q=

- Librería *json-simple*:
  http://code.google.com/p/json-simple/downloads/detail?name=json-simple-1.1.1.jar&can=2&q=

Bibliography

[1] Couchbase.com. Why nosql? URL http://www.couchbase.com/why-nosql/nosql-database.

[2] Pramod J. Sadalage Martin Fowler. *NoSQL Distilled: A brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley, 2013.

[3] Indeed job trends: crecimiento en trabajos que solicitan conocimientos en nosql, November 2013. URL http://www.indeed.com/jobtrends?q=NoSQL&l=&relative=1.

[4] Google trends: crecimiento del número de búsquedas del termino nosql, November 2013. URL http://www.google.com/trends/explore?q=Nosql#q=Nosql&date=1%2F2009%2060m&cmpt=q.

[5] Forbes: Improving decision making in the world of big data, November 2013. URL http://www.forbes.com/sites/christopherfrank/2012/03/25/improving-decision-making-in-the-world-of-big-data/.

[6] Neal Leavitt. Will nosql databases live up to their promise? *Computer*, 43(2): 12–14, February 2010. URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5410700&sortType%3Dasc_p_Sequence%26filter%3DAND%28p_IS_Number%3A5410692%29.

[7] Marcus Nitzschke. Mongodb as rdf triple store based on node.js. URL http://www.kendix.org/blog/mongodb-as-rdf-triple-store.

[8] Julian Browne. Brewer's cap theorem. 2009. URL http://www.julianbrowne.com/article/viewer/brewers-cap-theorem.

[9] Nosql base. URL http://www.w3resource.com/mongodb/nosql.php.

[10] *MonogoDB Manual*, . URL http://docs.mongodb.org/master/MongoDB-manual.pdf/.

[11] *Virtuoso:RDF Data Access and Data Management*, . URL http://docs.openlinksw.com/virtuoso/rdfandsparql.html.

[12] *Virtuoso: Server Administration*, . URL http://docs.openlinksw.com/virtuoso/dbadm.html.

[13] Mongodb. URL http://docs.mongodb.org/manual/core/introduction/.

[14] *MonogoDB Administration Concepts: GridFS*. URL http://docs.mongodb.org/manual/core/gridfs/.

[15] Db-engines ranking. URL http://db-engines.com/en/ranking.

[16] *MonogoDB: Patrones de diseño*. URL http://docs.mongodb.org/manual/data-modeling/.

[17] *MonogoDB: Operaciones Crud*. URL http://docs.mongodb.org/manual/crud/.

[18] *MonogoDB: Framework de agregación*. URL http://docs.mongodb.org/manual/aggregation/.

[19] *MonogoDB: relación entre tipos y su representación numérica en mongoDB*. URL http://docs.mongodb.org/manual/reference/operator/type/#op._S_type.

[20] Openlink virtuoso. URL http://docs.openlinksw.com/virtuoso/overview.html.

[21] *Virtuoso: conceptos básicos sobre el funcionamiento del sistema*. URL http://docs.openlinksw.com/virtuoso/concepts.html.

[22] Rendimiento del uso de jena con virtuoso. URL http://boards.openlinksw.com/phpBB3/viewtopic.php?f=12&t=842.

[23] *Virtuoso: representación de los datos RDF en el sistema*. URL http://docs.openlinksw.com/virtuoso/rdfdatarepresentation.html.

[24] The berlin sparql benchmark. URL http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/results/V7/.

[25] *Sparql: filtros*. URL http://www.w3.org/TR/rdf-sparql-query/#scopeFilters.

[26] *Virtuoso: RDF Index Scheme*. URL http://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/VirtRDFPerformanceTuning#RDF%20Index%20Scheme.

[27] Javadoc para los servicios. URL https://www.dropbox.com/s/96epy3htdm8knnn/ServicesJavaDoc.zip.

[28] Javadoc para la aplicación web. URL https://www.dropbox.com/s/e5c0f038x2v6s83/AplicacionPrinJavaDoc.zip.

[29] James Gosling Bill Joy Guy Steele Gilad Bracha Alex Buckley. The java language specification java se 7 edition. URL http://docs.oracle.com/javase/specs/jls/se7/html/jls-1.html.

[30] Wikipedia: J2ee. URL http://www.oracle.com/technetwork/java/javaee/overview/index.html.

[31] Json. URL http://www.json.org/.

[32] Wikipedia: Html. URL http://es.wikipedia.org/wiki/HTML.

[33] Wikipedia: Javasrcipt. URL http://es.wikipedia.org/wiki/JavaScript.

[34] Jquery. URL http://jquery.com/.

[35] Wikipedia: Junit. URL http://es.wikipedia.org/wiki/JUnit.

[36] Página de inicio del proyecto mozilla firefox. URL http://www.mozilla.org/es-ES/.

[37] Wikipedia: Tomcat. URL http://es.wikipedia.org/wiki/Tomcat.

[38] Wikipedia: Netbeans. URL http://es.wikipedia.org/wiki/NetBeans.

[39] Freebase. URL https://developers.google.com/freebase/index.

[40] Dropbox. URL https://www.dropbox.com/.

[41] Instalar oracle java 7 en ubuntu 12.04. URL http://www.ubuntu-guia.com/2012/04/instalar-oracle-java-7-en-ubuntu-1204.html.

[42] *Virtuoso: instalación de Virtuso en Ubuntu.* URL http://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/VOSUbuntuNotes.

[43] *Virtuoso: seguridad de los grafos.* URL http://docs.openlinksw.com/virtuoso/rdfgraphsecurity.html.

[44] *Virtuoso: Updates and Transactions.* URL http://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/VOSArticleVirtuosoAHybridRDBMSGraphColumnStore.

[45] *Virtuoso: Transaction Semantics in RDF and Relational Models.* URL http://www.openlinksw.com/weblog/oerling/?id=1691.

[46] *Virtuoso: mejorar el rendimieto al borrar muchos datos.* URL http://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/VirtTipsAndTricksGuideDeleteLargeGraphs.

[47] *Virtuoso: RDF Database Dump Script*. URL http://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/VirtRDFDatasetDump.

[48] *Virtuoso: Bulk RDF Loader Script*. URL http://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/VirtBulkRDFLoaderScript.