

CAPÍTULO 5

BASES DE DATOS NoSQL

OBJETIVOS

El alumno al término de la unidad debe ser capaz de:

Instalar y utilizar bases de datos NoSQL (eXist y Mongo DB).

Realizar consultas a documentos y colecciones utilizando XPath y XQuery.

Instalar y utilizar la BD MongoDB.

Realizar consultas a bases de datos y sus colecciones en mongoDB.

Desarrollar aplicaciones Java con bases de datos NoSQL, accediendo a su información para consultarla y manipularla.

CONTENIDOS

Bases de datos NoSQL. Características.

Base de datos nativa XML, eXist.

Instalación eXist.

Lenguajes de consultas.

Lenguaje Java y eXist.

Estructuras JSON.

Instalación de MongoDB.

Consultas y manipulación de datos en MongoDb.

Java y MongoDB.

RESUMEN

En este capítulo se estudian bases de datos NoSQL, bases de datos que no utilizan el lenguaje SQL para hacer sus consultas, y no utilizan estructuras fijas de almacenamiento. Aprenderemos a consultar las informaciones, y a crear, eliminar y modificar datos utilizando nuevos lenguajes de consulta y nuevos modelos de almacenamiento de datos basados en documentos como son el XML y JSON.

5.1. INTRODUCCIÓN

Las bases de datos NoSQL son aquellas que no siguen el modelo clásico del sistema de gestión de bases de datos relacionales (RDBMS). Se caracterizan porque no usan el SQL como lenguaje principal de consultas, y además, en el almacenamiento de los datos no se utilizan estructuras fijas de almacenamiento.

El término NoSQL surge con la llegada de la web 2.0, ya que hasta ese momento solo subían contenidos a la red aquellas empresas que tenían un portal, pero con la llegada de aplicaciones como Facebook, Twitter o Youtube, en las que el usuario interactúa en la web, cualquier usuario podía subir contenido, provocando así un crecimiento exponencial de los datos.

Surgen así los problemas para gestionar y acceder a toda esa información almacenada en bases de datos relacionales. Una de las soluciones, propuestas por las empresas para solucionar estos problemas de accesibilidad, fue la de utilizar más máquinas, sin embargo, era una solución cara y no terminaba con el problema.

La otra solución fue la de crear nuevos sistemas gestores de datos pensados para un uso específico, que con el paso del tiempo han dado lugar a soluciones robustas, apareciendo así el movimiento NoSQL.

Así pues, hablar de bases de datos NoSQL es hablar de estructuras que nos permiten almacenar información en aquellas situaciones en las que las bases de datos relacionales generan ciertos problemas, debido principalmente a problemas de escalabilidad y rendimiento de las bases de datos relacionales, donde se dan cita miles de usuarios concurrentes y con millones de consultas diarias. Por tanto, las bases de datos NoSQL intentan resolver problemas de almacenamiento masivo, alto desempeño, procesamiento masivo de transacciones (sitios con alto tránsito) y, en términos generales, ser alternativas NoSQL a problemas de persistencia y almacenamiento masivo (voluminoso) de información para las organizaciones.

SQL vs NoSQL

Las bases de datos relacionales focalizan su interés en la fiabilidad de las transacciones bajo el conocido principio **ACID**, acrónimo de ***Atomicity, Consistency, Isolation and Durability*** (Atomicidad, Consistencia, Aislamiento y Durabilidad en español):

PRINCIPIO ACID – Bases de datos relacionales	
Atomicity	Asegurar de que la transacción se complete o no, sin quedarse a medias ante fallos
Consistency	Asegurar el estado de validez de los datos en todo momento
Isolation	Asegurar independencia entre transacciones
Durability	Asegurar la persistencia de la transacción ante cualquier fallo

El principio ACID aporta una robustez que colisiona con el rendimiento y operatividad a medida que los volúmenes de datos crecen.

Cuando la magnitud y el dinamismo de los datos cobran importancia, el principio ACID de los modelos relacionales queda en segundo plano frente al rendimiento, disponibilidad y escalabilidad, las características más propias de las bases de datos NoSQL. Hoy en día, los modernos sistemas de datos en internet se ajustan más al también conocido principio **BASE**,

acrónimo de ***Basic Availability*** (disponibilidad como prioridad) ***Soft state*** (la consistencia de datos se delega a gestión externa al motor de la base de datos) ***Eventually consistency*** (intentar lograr la convergencia hacia un estado consistente)

PRINCIPIO BASE – Bases de datos NoSQL	
Basic Availability	Prioridad de la disponibilidad de los datos
Soft state	Se prioriza la propagación de datos, delegando el control de inconsistencias a elementos externos
Eventually consistency	Se asume que inconsistencias temporales progresen a un estado final estable

Estas informaciones han sido consultadas en las siguientes URLs:

<https://www.certsi.es/blog/bases-de-datos-nosql> y

<http://www.acens.com/wp-content/images/2014/02/bbdd-nosql-wp-acens.pdf>

5.2. VENTAJAS DE LOS SISTEMAS NoSQL

La gran diferencia de estas bases de datos es cómo almacenan los datos. Por ejemplo, una factura en el modelo relacional termina guardándose en 4 tablas (con 3 o 4 claves ajena) y en NoSQL simplemente guarda la factura y no se diseña ninguna estructura por adelantado, se almacena y ya está, por ejemplo, una clave (numero de la factura) y el Objeto (la factura).

La forma de almacenamiento de información en este tipo de bases de datos ofrece ciertas ventajas sobre los modelos relacionales, a destacar las siguientes:

- ***Se ejecutan en máquinas con pocos recursos:*** estos sistemas no requieren mucha programación, por lo que se pueden instalar en máquinas de un coste más reducido.
- ***Escalabilidad horizontal:*** para mejorar el rendimiento de estos sistemas simplemente se consigue añadiendo más nodos, con la única operación de indicar al sistema cuáles son los nodos que están disponibles.
- ***Pueden manejar gran cantidad de datos:*** esto es debido a que utiliza una estructura distribuida, en muchos casos mediante tablas Hash.
- ***No genera cuellos de botella:*** el principal problema de los sistemas SQL es que necesitan transcribir cada sentencia para poder ser ejecutada, y cada sentencia compleja requiere, además, de un nivel de ejecución aún más complejo, lo que constituye un punto de entrada en común, que ante muchas peticiones puede ralentizar el sistema.

5.3. DIFERENCIAS CON LAS BASES DE DATOS SQL

Estas son las diferencias más importantes entre los sistemas NoSQL y los sistemas SQL:

- ***No utilizan SQL como lenguaje de consultas.*** La mayoría de las bases de datos NoSQL evitan utilizar este tipo de lenguaje o lo utilizan como un lenguaje de apoyo. Por poner algunos ejemplos, Cassandra utiliza el lenguaje CQL, MongoDB utiliza JSON o BigTable hace uso de GQL.

- **No utilizan estructuras fijas como tablas para el almacenamiento de los datos.** Permiten hacer uso de otros tipos de modelos de almacenamiento de información como sistemas de clave–valor, objetos o grafos.
- **No suelen permitir operaciones JOIN.** Al disponer de un volumen de datos tan extremadamente grande suele resultar deseable evitar los JOIN. Esto se debe a que, cuando la operación no es la búsqueda de una clave, la sobrecarga puede llegar a ser muy costosa. Las soluciones más directas consisten en desnormalizar los datos, o bien, realizar el JOIN mediante software en la capa de aplicación.
- **Arquitectura distribuida.** Las bases de datos relacionales suelen estar centralizadas en una única máquina o bien en una estructura máster–esclavo, sin embargo en los casos NoSQL la información puede estar compartida en varias máquinas mediante mecanismos de tablas Hash distribuidas.

5.4. TIPOS DE BASES DE DATOS NOSQL

Según el tipo o modelo escogido para almacenar los datos, las bases de datos NoSQL se agrupan en cuatro categorías principales:

- **Clave/Valor.** Los datos son almacenados y se localizan e identifican usando una clave única y un valor (un dato o puntero a los datos). Ejemplos de este tipo son: *DynamoDB*, *Riak*, o *Redis*. Amazon y Best Buy entre otros utilizan esta implementación. Se caracterizan por ser muy eficientes tanto para las lecturas como para las escrituras.
- **Columnas.** Parecido al modelo clave/valor, pero la clave se basa en una combinación de columna, fila y marca de tiempo que se utiliza para referenciar conjuntos de columnas (familias). Es la implementación más parecida a bases de datos relacionales. Ejemplos: *Cassandra*, *BigTable*, *Hadoop/HBase*. Compañías como Twitter o Adobe hacen uso de este modelo.
- **Documentos.** Los datos se almacenan en documentos que encapsulan la información en formato XML, YAML o JSON. Los documentos tienen nombres de campos auto contenidos en el propio documento. La información se indexa utilizando esos nombres de campos. Este tipo de implementación permite, además de realizar búsquedas por clave–valor, realizar consultas más avanzadas sobre el contenido del documento. Ejemplos: *MongoDB*, *CouchDB*, *eXist*. Un caso de uso de esta tecnología lo tenemos con Netflix, empresa que proporciona contenidos audiovisuales online.
- **Grafos.** Se sigue un modelo de grafos que se extiende entre múltiples máquinas. En este tipo de bases de datos, la información se representa como nodos de un grafo y sus relaciones con las aristas del mismo, de manera que se puede hacer uso de la teoría de grafos para recorrerla. Es un modelo apropiado para datos cuyas relaciones se ajustan a este modelo, como, por ejemplo, redes de transporte, mapas, etc. Ejemplos: *Neo4J*, *GraphBase* o *Virtuoso*.

En esta unidad estudiaremos dos bases de datos NoSQL, por un lado la base de datos *eXist* que almacena documentos XML, y por otro lado, la base de datos *MongoDB* que almacena estructuras JSON.

5.5. BASES DE DATOS NATIVAS XML

A diferencia de las bases de datos relacionales (centradas en los datos), las bases de datos nativas XML, no poseen campos, ni almacena datos, lo que almacenan son documentos XML, son bases de datos centradas en documentos y la unidad mínima de almacenamiento es el documento XML. Podemos definir una base de datos nativa XML (o XML nativa), como un sistema de gestión de información que cumple con lo siguiente:

- Define un modelo lógico para un documento XML (y no para los datos que contiene el documento), y almacena y recupera los documentos según este modelo.
- Tiene una relación transparente con el mecanismo de almacenamiento, que debe incorporar las características ACID de cualquier SGBD.
- Incluye un número arbitrario de niveles de datos y complejidad.
- Permite las tecnologías de consulta y transformación propias de XML, XQuery, XPath, XSLT, etc., como vehículo principal de acceso y tratamiento.

Ventajas de las bases de datos XML:

- Ofrecen un acceso y almacenamiento de información ya en formato XML, sin necesidad de incorporar código adicional.
- La mayoría incorpora un motor de búsqueda de alto rendimiento.
- Es muy sencillo añadir nuevos documentos XML al repositorio.
- Se pueden almacenar datos heterogéneos.

Por el contrario, se pueden considerar como desventajas de las bases de datos XML:

- Puede resultar difícil indexar documentos para realizar búsquedas.
- No suelen ofrecer funciones para la agregación (crucial para el procesamiento de transacciones en línea OLTP -Online Transaction Processing) en muchos casos hay que reintroducir todo el documento para modificar una sola línea.
- Se suele almacenar la información en XML como un documento o como un conjunto de nodos, por lo que su síntesis para formar nuevas estructuras sobre la marcha puede resultar complicada y lenta.

En la actualidad la mayoría de SGBD incorporan mecanismos para extraer y almacenar datos en formato XML. A continuación se muestra un ejemplo de soporte de datos XML en Oracle y en MySQL:

- ***La siguiente select de ORACLE devuelve las filas de la tabla EMPLE en formato XML:***

```
SELECT XMLEMENT ("EMP_ROW",
    XMLFOREST(EMP_NO ,APELIDO,OFICIO, DIR, FECHA_ALT ,
    SALARIO , COMISION ,DEPT_NO )) FILA_EN_XML
FROM EMPLE;
```

- ***Creación de una tabla que almacena datos del tipo XML (el tipo de dato XMLTYPE permite almacenar y consultar datos XML):***

```
CREATE TABLE TABLA_XML_PRUEBA (COD NUMBER, DATOS XMLTYPE) ;
```

- ***Insertar filas en formato XML:***

```

INSERT INTO TABLA_XML_PRUEBA VALUES (1,
    XMLTYPE('<FILA_EMP><EMP_NO>123</EMP_NO>
    <APELLIDO>RAMOS MARTÍN</APELLIDO>
    <OFICIO>PROFESORA</OFICIO>
    <SALARIO>1500</SALARIO></FILA_EMP>')) ;

INSERT INTO TABLA_XML_PRUEBA VALUES (1,
    XMLTYPE('<FILA_EMP><EMP_NO>124</EMP_NO>
    <APELLIDO>GARCÍA SALADO</APELLIDO>
    <OFICIO>FONTANERO</OFICIO>
    <SALARIO>1700</SALARIO></FILA_EMP>')) ;

```

- **Extracción de datos de la tabla, se utilizan expresiones XPath:**

Visualiza los apellidos de los empleados:

```

SELECT EXTRACTVALUE(DATOS, '/FILA_EMP/APELLIDO')
    FROM TABLA_XML_PRUEBA;

```

Visualiza el nombre del empleado con número de empleado = 123

```

SELECT EXTRACTVALUE(DATOS, '/FILA_EMP/APELLIDO')
    FROM TABLA_XML_PRUEBA
WHERE EXISTSNODE(DATOS, '/FILA_EMP[EMP_NO=123]') = 1;

```

- En el siguiente ejemplo se muestra cómo MySQL devuelve los datos de una tabla en formato XML. Por ejemplo, disponemos de la BD en MySQL ejemplo, y se desea obtener los datos de la tabla departamentos en formato XML. El usuario de la BD y su clave se llama también ejemplo.

Desde la línea de comando y en la carpeta donde se encuentra instalado MySQL (por ejemplo desde la carpeta D:\xampp\mysql\bin>) escribiremos la siguiente orden para obtener los datos de la tabla departamentos:

```
mysql --xml -u ejemplo -p -e "select * from departamentos;" ejemplo
```

Donde **--xml**, se utiliza para producir una salida en XML.

-u, a continuación se pone el nombre de usuario de la BD *ejemplo*.

-p, al pulsar en la tecla [Intro] para ejecutar la orden nos pide la clave del usuario (*ejemplo*)

-e, ejecuta el comando y sale de MySQL.

El resultado es:

```

<?xml version="1.0"?>
<resultset statement="select * from departamentos"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <row>
        <field name="dept_no">10</field>
        <field name="dnombre">CONTABILIDAD</field>
        <field name="loc">SEVILLA</field>
    </row>
    <row>
        <field name="dept_no">20</field>
        <field name="dnombre">INVESTIGACIÓN</field>
        <field name="loc">MADRID</field>
    </row>

```

```

</row>
<row>
    <field name="dept_no">30</field>
    <field name="dnombre">VENTAS</field>
    <field name="loc">BARCELONA</field>
</row>
<row>
    <field name="dept_no">40</field>
    <field name="dnombre">PRODUCCIÓN</field>
    <field name="loc">BILBAO</field>
</row>
</resultset>

```

Si deseamos redireccionar la salida al archivo **departamentos.xml** escribiremos en la misma línea:

```
mysql --xml -u ejemplo -p -e "select * from departamentos;" ejemplo
>departamentos.xml
```

5.5.1. Base de Datos eXist

eXist es SGBD libre de código abierto que almacena datos XML de acuerdo a un modelo de datos XML. El motor de base de datos está completamente escrito en Java, soporta los estándares de consulta XPath, XQuery y XSLT, además de indexación de documentos y soporte para la actualización de los datos y para multitud de protocolos como SOAP, XML-RPC, WebDav y REST. Con el SGBD se dan aplicaciones que permiten ejecutar consultas directamente sobre la BD.

Los documentos XML se almacenan en colecciones, las cuales pueden estar anidadas; desde un punto de vista práctico el almacén de datos funciona como un sistema de ficheros. Cada documento está en una colección, las colecciones serían como las carpetas. No es necesario que los documentos tengan una DTD o un XML Schema asociado (XSD), y dentro de una colección pueden almacenarse documentos de cualquier tipo.

En la carpeta **eXist\webapp\WEB-INF\data** es donde se guardan los archivos más importantes de la BD, entre ellos están:

- **dom.dbx**: el almacén central nativo de datos; es un fichero paginado donde se almacenan todos los nodos del documento de acuerdo al modelo DOM del W3C.
- **collections.dbx**, se almacena la jerarquía de colecciones y relaciona esta con los documentos que contiene; se asigna un identificador único a cada documento de la colección que es almacenado también junto al índice.

5.5.1.1. Instalación de eXist

eXist puede funcionar de distintos modos:

- Funcionando como servidor autónomo (ofreciendo servicios de llamada remota a procedimientos que funciona sobre Internet como XML-RPC, WebDAV y REST).
- Insertado dentro de una aplicación Java.
- En un servidor J2EE, ofreciendo servicios XML-RPC, SOAP y WebDAV.

En el sitio <http://exist-db.org/exist/apps/homepage/index.html> podremos descargar la última versión de la BD. En este caso se ha instalado la versión *eXist 3.0.RC1.jar*, es necesario tener instalada la versión Java 8. Se instala utilizando el fichero JAR proporcionado en la web, que ha de invocarse desde la línea de comandos (*java -jar eXist 3.0.RC1.jar*), o bien, haciendo doble clic sobre el ícono correspondiente una vez descargado, y siguiendo el asistente.

El proceso de instalación es bastante intuitivo, simplemente seguir el asistente y estar atento a los pasos 2, donde se indica el directorio para la instalación, y 4 donde se indica la password del administrador, en nuestro caso pondremos **admin**, para acordarnos. Véanse las Figuras 5.1 y 5.2.

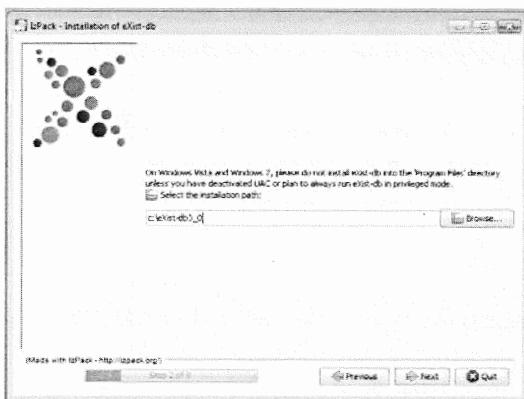


Figura 5.1. Paso 2 de la instalación.

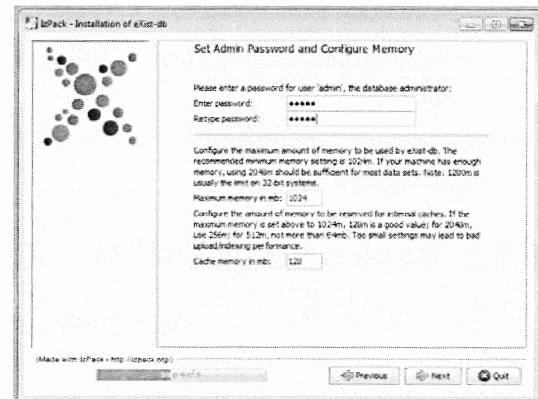


Figura 5.2. Paso 4 de la instalación.

Una vez instalada, para arrancar la base de datos se puede hacer desde el acceso directo que se crea en el escritorio (si así se ha decidido en la instalación), o también desde el menú de la aplicación seleccionando (véase la Figura 5.3) *eXist-db XML Database*. También se puede lanzar el fichero *start.jar* que se encuentra en la carpeta de eXist.

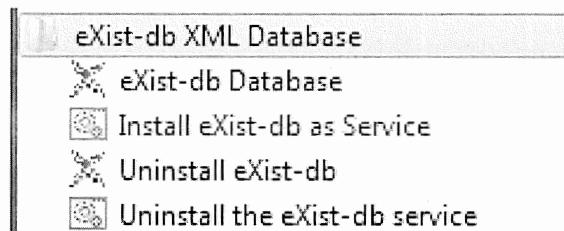


Figura 5.3. Menú de eXist.

Al lanzar la BD puede ocurrir que esta no se inicie y nos visualice una pantalla como la de la Figura 5.4. El problema ocurrirá cuando al intentar conectar con la BD, el puerto que utiliza eXist esté ocupado por otra aplicación. Exist se instala en un servidor web (*jetty*) y ocupa el puerto **8080**, como la mayoría de servidores web (Apache, Tomcat, Lampp...)

```

may 2016 13:24:52.950 [Thread-5] INFO  (JettyStart.java [run]:121) - Configuring eXist from /etc/exist-db/jetty.xml
may 2016 13:24:52.953 [Thread-5] INFO  (JettyStart.java [run]:123) - Running with Java 1.8.0_40 Oracle Corporation (Java HotSpot(TM) 64-Bit Server VM in C:\Program Files\Java\jre1.8.0_40)
may 2016 13:24:52.955 [Thread-5] INFO  (JettyStart.java [run]:132) - [eXist Version : 3.0.0.M1]
may 2016 13:24:52.955 [Thread-5] INFO  (JettyStart.java [run]:134) - (eXist Build : 20150707)
may 2016 13:24:52.957 [Thread-5] INFO  (JettyStart.java [run]:134) - (eXist Name : omnion)
may 2016 13:24:52.957 [Thread-5] INFO  (JettyStart.java [run]:134) - (eXist Home : omnion)
may 2016 13:24:52.959 [Thread-5] INFO  (JettyStart.java [run]:134) - (Operating System: Windows 7 6.1 amd64)
may 2016 13:24:52.959 [Thread-5] INFO  (JettyStart.java [run]:146) - [jetty.home : C:\exist-db_3.0.0.M1\exist-db_3.0.0.M1]
may 2016 13:24:52.969 [Thread-5] INFO  (JettyStart.java [run]:148) - [log4j.configurationFile : file:/C:/exist-db_3.0.0.M1/log4j2.xml]
may 2016 13:24:52.971 [Thread-5] INFO  (JettyStart.java [run]:149) - [log4j.configurationFile : log4j2.xml]
or deleting dir: c:\exist-db_3.0\log4j2.xml
or deleting dir: c:\exist-db_3.0\log4j2.xml\log4j2.xml
or deleting dir: c:\exist-db_3.0\log4j2.xml\log4j2.xml\log4j2.xml
or deleting dir: c:\exist-db_3.0\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml
or deleting dir: c:\exist-db_3.0\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml
or deleting dir: c:\exist-db_3.0\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml
or deleting dir: c:\exist-db_3.0\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml
or deleting dir: c:\exist-db_3.0\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml
or deleting dir: c:\exist-db_3.0\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml
or deleting dir: c:\exist-db_3.0\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml
or deleting dir: c:\exist-db_3.0\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml
or deleting dir: c:\exist-db_3.0\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml
or deleting dir: c:\exist-db_3.0\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml
or deleting dir: c:\exist-db_3.0\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml
or deleting dir: c:\exist-db_3.0\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml\log4j2.xml
utilizing RemoteConsole...
may 2016 13:25:05.228 [Thread-5] INFO  (JettyStart.java [run]:833) - [eXist] Could not bind to port because address already in use: bind
may 2016 13:25:05.229 [Thread-5] INFO  (JettyStart.java [run]:835) - [eXist] bindToPortOptimistic: Address already in use: bind
may 2016 13:25:05.229 [Thread-5] INFO  (JettyStart.java [run]:839) -

```

Figura 5.4. Error de arranque de la BD.

Para resolver el problema, cambiamos el puerto. Para ello se edita el fichero de configuración: **%HOME_EXIST%/tools/jetty/etc/jetty.xml**, en la etiqueta **<Call name="addConnector">** cambiaremos el puerto, y en la propiedad **SystemProperty** de la etiqueta **<Set name="port">** escribimos un nuevo valor, por ejemplo 8083:

```

<SystemProperty name="jetty.port" default="8080"/>
por
<SystemProperty name="jetty.port" default="8083"/>

```

Al lanzar la BD también se creará un ícono asociado a eXist en la barra de tareas (Figura 5.5), desde allí podremos iniciar y parar el servidor, y abrir las distintas herramientas de trabajo con esta base de datos (*Dashboard*, *eXide* y *Java Admin Client*).

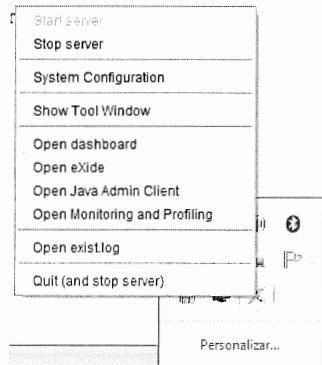


Figura 5.5. Opciones de eXist desde a barra de tareas.

5.5.1.2. Primeros pasos con eXist

La base de datos, una vez arrancada, también se puede abrir desde el navegador escribiendo: <http://localhost:8083/exist/>.

Para hacer consultas y trabajar con la base de datos lo podemos hacer desde varios sitios:

- Desde el **eXide (Open eXide)**: el eXide (véase Figura 5.6) es una de las herramientas para realizar consultas a documentos de la bd, cargar documentos externos a la BD, crear y borrar colecciones, entre otras cosas.

La URL del exide es: <http://localhost:8083/exist/apps/eXide/index.html>. Desde la pestaña **directory**, se podrá navegar por las carpetas y documentos de la BD. Dentro de **/db/apps/demo/data/** se encuentran algunos documentos XML como *hamlet.xml*, o *macbeth.xml*. Para realizar operaciones se utilizarán los iconos de la barra de herramientas.

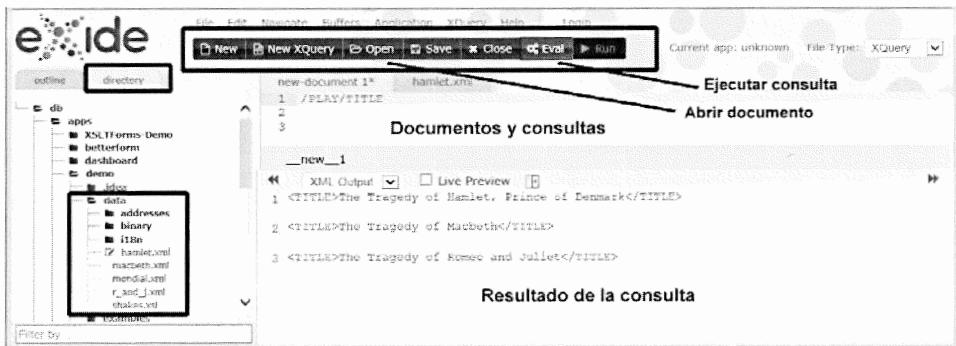


Figura 5.6. El eXide.

- Desde el dashboard (**Open dashboard**): el salpicadero o cuadro de instrumentos es el administrador de aplicaciones de la BD (véase Figura 5.7). Soporta aplicaciones y plugins. Las aplicaciones proporcionan su propia interfaz gráfica de usuario web, mientras que los "plugins" se ejecutan dentro del *dashboard* como los diálogos de una sola pantalla. Ejemplos de las aplicaciones son la documentación *eXist-bd Documentation*, el entorno de consultas *eXide - XQuery IDE*, o la aplicación de demostración *eXist-bd Demo Apps*. Ejemplos de plugins son el gestor de colecciones *Collections*, o el gestor de Backups *Backup*, o el gestor de usuarios *User Manager*, o el gestor de paquetes *Package Manager*. Los plugins son más adecuados para las funciones administrativas.

Más información en <http://exist-db.org/exist/apps/doc/dashboard.xml>

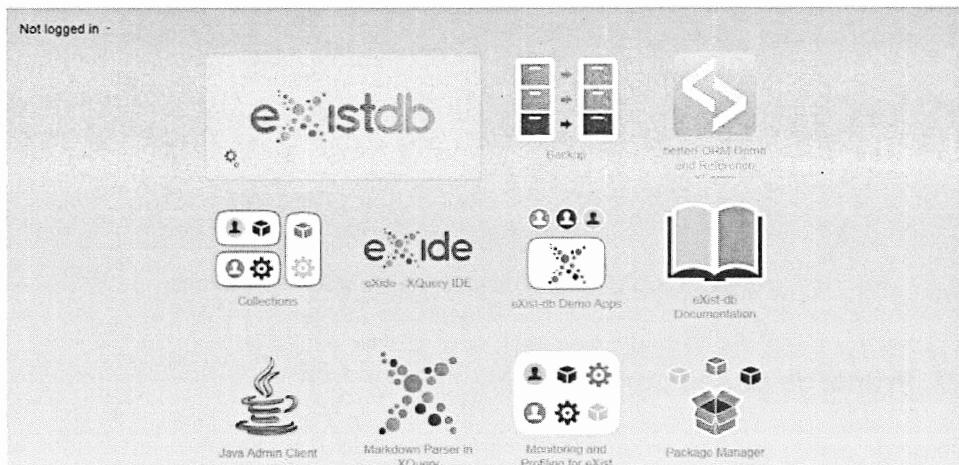


Figura 5.7. El dashboard.

- Desde el cliente java (**Open Java Admin Client**): el cliente es la herramienta que utilizaremos lo largo del tema para hacer las consultas. El cliente nos pedirá conexión, con el usuario y la contraseña (utilizaremos admin/admin) y hay que asegurarse de poner correctamente el puerto de la URL: `xmlrpc:exist://localhost:8083/exist/xmlrpc`.

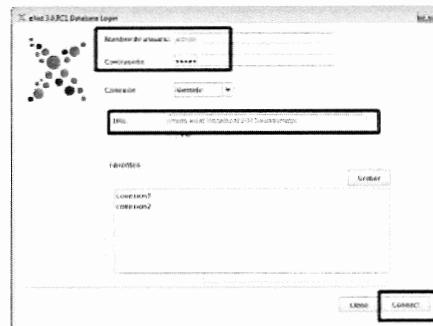


Figura 5.8. El Cliente Java.

5.5.1.3. El cliente de administración de eXist

Una vez conectado se muestra la ventana del *Cliente de Administración eXist* (Figura 5.9), desde aquí se podrán realizar todo tipo de operaciones sobre la BD, crear y borrar colecciones, añadir y eliminar documentos a las colecciones, modificar los documentos, crear copias de seguridad y restaurarlas, administrar usuarios y realizar consultas XPath, entre otras operaciones. Además, podremos navegar por las colecciones (las carpetas) y elegir un contexto a partir de donde se ejecutarán las consultas. Igualmente si se hace doble clic en un documento este se abrirá, y también se podrán hacer cambios en los mismos.

Si pulsamos al botón **(Consultar la BD usando XPath)**, aparece la ventana de consultas **Diálogo de consulta**, desde aquí se puede elegir también, el contexto sobre el que ejecutaremos las consultas, se pueden también guardar las consultas y los resultados de las consultas en ficheros externos. En la parte superior escribimos la consulta, pulsamos el botón *Ejecutar*, y en la inferior se muestra el resultado, también se puede ver la traza de ejecución seguida en la ejecución de la consulta. En la Figura 5.10 se muestra la ventana de consultas.

Figura 5.9. Cliente de Administración de eXist.

Figura 5.10. Ventana de consultas *Dialogo de Consultas* de eXist.

SUBIR UNA COLECCIÓN A LA BASE DE DATOS

Lo primero que haremos es subir una colección a la base de datos desde el cliente. Nos conectamos con *Admin*, de momento es el único usuario con el que se va a trabajar. En la ventana

del *Cliente* pulsamos al botón *Almacena uno o más ficheros en la base de datos*, y en la ventana que aparece seleccionamos la carpeta a subir y se pulsa al botón *Seleccionar los ficheros o directorios*, véase la Figura 5.11.

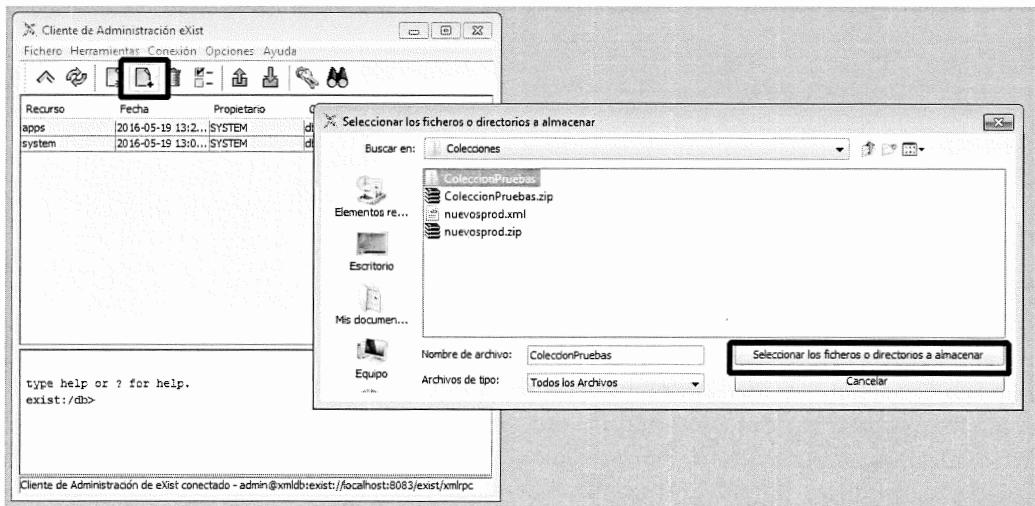


Figura 5.11. Subir una colección a eXist.

Una vez pulsado, aparece una ventana de transferencia de datos, y al cerrarla, la colección con los documentos se creará en la base de datos, y se creará dentro de la colección (o carpeta) desde donde se llamó a la subida de documentos. En principio se subirán dentro del contexto `exist:/db>`.

Para hacer consultas sobre esa colección, hacemos doble clic sobre ella para seleccionarla, y una vez dentro pulsamos al botón *Consultar la base de datos usando XPath*. Así nos aseguramos de hacer las consultas dentro de nuestra colección, ese será el contexto a utilizar. Desde el diálogo de consulta también se podrán guardar las consultas y los resultados en ficheros, Compilar y ejecutar las consultas. Y desde el historial obtendremos las consultas ya realizadas. Véase la Figura 5.12. Si la consulta se realiza mal, aparecerá una ventana Java informando del error.

The screenshot shows the 'eXist Administration Client' window. On the left, the 'Recurso' list includes 'sucursalales.xml', 'universidad.xml', and 'zonas.xml'. The 'colecciónPruebas' folder is selected and highlighted with a red box. The status bar at the bottom indicates: 'Cliente de Administración de eXist conectado - admin@xmldb:exist://localhost:8083/exist...'.

On the right, a 'Diálogo de consulta' window is open. It has tabs for 'Consultar', 'Guardar la consulta', 'Ejecutar', and 'Historial de consultas'. The 'Ejecutar' tab is active, showing the query results. The results pane displays the following XML output:

```

<departamento telefono="112233" tipo="A">
  <codigo>IFC1</codigo>
  <nombre>Informática</nombre>
  <empleo>salario="2000"</empleo>
  <puesto>Asociado</puesto>
  <nombre>Juan Parral</nombre>
</departamento>
<departamento telefono="234567" tipo="B">
  <codigo>IFC2</codigo>
  <nombre>Ingeniería de Software</nombre>
  <empleo>salario="2300"</empleo>
  <puesto>Profesor</puesto>
  <nombre>Alicia Martín</nombre>
</departamento>

```

The status bar at the bottom of the client window indicates: 'Hallados 3 resultados. Compilación: 29ms, Ejecución: 55ms Line: 1 Column: 26'.

Figura 5.12. Haciendo consultas en eXist.

5.5.2. Lenguajes de consultas XPath y XQuery

Ambos son estándares para acceder y obtener datos desde documentos XML, estos lenguajes tienen en cuenta que la información en los documentos está semiestructurada o jerarquizada como árbol.

XPath, es el lenguaje de rutas de XML, se utiliza para navegar dentro de la estructura jerárquica de un XML.

XQuery es a XML lo mismo que SQL es a las bases de datos relacionales, es decir, es un lenguaje de consulta diseñado para consultar documentos XML. Abarca desde archivos XML hasta bases de datos relacionales con funciones de conversión de registros a XML. XQuery contiene a XPath, toda expresión de consulta en XPath es válida en XQuery, pero XQuery permite mucho más.

5.5.2.1. Expresiones XPath

XPath es un lenguaje que permite seleccionar nodos de un documento XML y calcular valores a partir de su contenido. Existen varias versiones de XPath aprobadas por el W3C, aunque la versión más utilizada sigue siendo la versión 1.

La forma en que XPath selecciona partes del documento XML se basa en la representación arbórea que se genera del documento. A la hora de recorrer un árbol XML podemos encontrarnos con los siguientes tipos de nodos:

- **nodo raíz**, es la raíz del árbol, se representa por /.
- **nodos elemento**, cualquier elemento de un documento XML, son las etiquetas del árbol.
- **nodos texto**, los caracteres que están entre las etiquetas.
- **nodos atributo**, son como propiedades añadidas a los nodos elemento, se representan con @.
- **nodos comentario**, las etiquetas de comentario.
- **nodos espacio de nombres**, contienen espacios de nombres.
- **nodos instrucción de proceso**, contienen instrucciones de proceso, van entre las etiquetas <? ?>.

Por ejemplo. Dado este documento XML:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<universidad>
<espacio xmlns="http://www.misitio.com"
          xmlns:prueba="http://www.misitio.com/pruebas" />
<!-- DEPARTAMENTO 1 -->
<departamento telefono="112233" tipo="A">
  <codigo>IFC1</codigo>
  <nombre>Informática</nombre>
</departamento>
<!-- DEPARTAMENTO 2 -->
<departamento telefono="990033" tipo="A">
  <codigo>MAT1</codigo>
  <nombre>Matemáticas</nombre>
</departamento>
<!-- DEPARTAMENTO 3 -->
<departamento telefono="880833" tipo="B">
  <codigo>MAT2</codigo>
  <nombre>Análisis</nombre>
```

```
</departamento>
</universidad>
```

Nos encontramos los siguientes tipos de nodos:

TIPO NODO	
Elemento	<universidad>, <departamento>, <codigo>, <nombre>
Texto	IFC1, Informática, MAT1, Matemáticas, MAT2, Análisis
Atributo	telefono="112233" tipo="A" , telefono="990033" tipo="A", telefono="880833" tipo="B"
Comentario	<!-- DEPARTAMENTO 1 -->, <!-- DEPARTAMENTO 2 --> <!-- DEPARTAMENTO 3 -->
Espacio de nombres	<espacio xmlns="http://www.misitio.com" xmlns:prueba="http://www.misitio.com/pruebas" />
Instrucción de proceso	<?xml version="1.0" encoding="ISO-8859-1"?>

Los test sobre los tipos de nodos pueden ser:

- Nombre del nodo, para seleccionar un nodo concreto, ej.: /universidad
- **prefix:***, para seleccionar nodos con un espacio de nombres determinado.
- **text()**, selecciona el contenido del elemento, es decir, el texto, ej.: //nombre/text().
- **node()**, selecciona todos los nodos, los elementos y el texto, ej.: /universidad/node().
- **processing-instruction()**, selecciona nodos que son instrucciones de proceso.
- **comment()**, selecciona los nodos de tipo comentario, /universidad/comment().

La sintaxis básica de XPath es similar a la del direccionamiento de ficheros. Utiliza **descriptores de ruta o de camino** que sirven para seleccionar los nodos o elementos que se encuentran en cierta ruta en el documento. Cada descriptor de ruta o paso de búsqueda puede a su vez dividirse en tres partes:

- **eje**: indica el nodo o los nodos en los que se realiza la búsqueda.
- **nodo de comprobación**: especifica el nodo o los nodos seleccionados dentro del eje.
- **predicado**: permite restringir los nodos de comprobación. Los predicados se escriben entre corchetes.

Las expresiones XPath se pueden escribir utilizando una sintaxis abreviada, fácil de leer, o una sintaxis más completa en la que aparecen los nombres de los ejes (AXIS), más compleja. Por ejemplo, estas dos expresiones devuelven los departamentos con más de 3 empleados, la primera es la forma abreviada y la segunda es la completa:

```
/universidad/departamento [count ( empleado ) >3]
/child::universidad/child::departamento [count ( child::empleado ) >3]
```

En este tema estudiaremos la **sintaxis abreviada**, así pues, los descriptores se forman simplemente nombrando la etiqueta separada por / (hay que poner el nombre de la etiqueta tal cual está en el documento XML, recuerda que hace distinción entre mayúsculas y minúsculas).

Si el descriptor comienza con / se supone que es una **ruta desde la raíz**. Para seguir una ruta indicaremos los distintos nodos de paso: /paso1/paso2/paso3... Si las rutas comienzan con / son **rutas absolutas**, en caso contrario serán relativas.

Si el descriptor comienza con // se supone que la ruta descrita puede comenzar en cualquier parte de la colección.

EJEMPLOS XPATH UTILIZANDO UNA SINTAXIS ABREVIADA:

A partir de la colección **ColecciónPruebas**, que contiene los documentos *departamentos.xml* y *empleados.xml*, cuyas estructuras se muestran en la Figura 5.13. Realiza desde el diálogo de consultas las siguientes consultas:

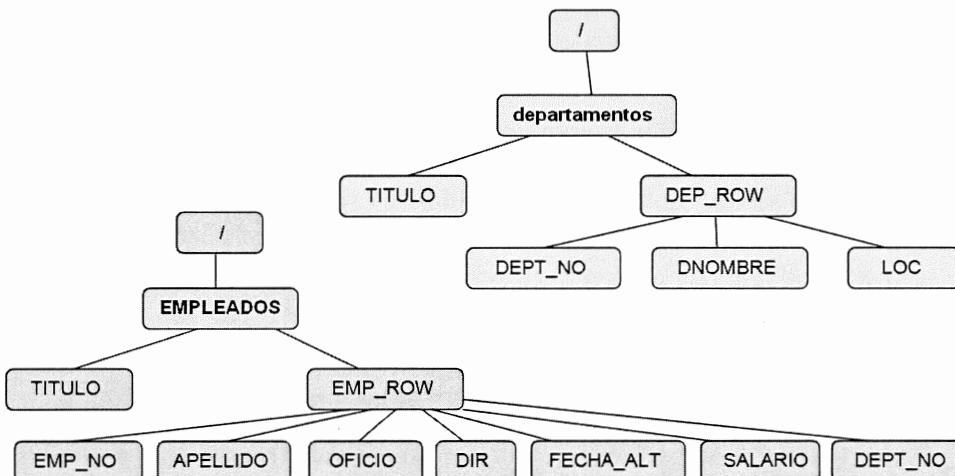


Figura 5.13. Estructuras de *departamentos.xml* y *empleados.xml*.

- /, si ejecutamos esta orden en el (*Diálogo de Consulta*) y se está dentro del contexto /db/ColecciónPruebas devuelve todos los datos de los departamentos y los empleados, es decir, incluye todas las etiquetas que cuelgan del nodo *departamentos* y del nodo *EMPLEADOS*, que están dentro de la colección *Prueba*. Si se ejecuta desde XQuery Sandbox, se obtiene otro resultado, pues el contexto no es el mismo.
- /departamentos, devuelve todos los datos de los departamentos, es decir, incluye todas las etiquetas que cuelgan del nodo raíz *departamentos*.
- /departamentos/DEP_ROW, devuelve todas las etiquetas dentro de cada DEP_ROW.
- /departamentos/DEP_ROW/node(), devuelve todas las etiquetas dentro de cada DEP_ROW, no incluye DEP_ROW.
- /departamentos/DEP_ROW/DNOMBRE, devuelve los nombres de departamentos de cada DEP_ROW, entre etiquetas.
- /departamentos/DEP_ROW/DNOMBRE/text(), devuelve los nombres de departamentos de cada DEP_ROW, ya sin las etiquetas.
- //LOC/text(), devuelve todas las localidades, de toda la colección (//), solo hay 4.
- //DEPT_NO, devuelve todos los números de departamentos, entre etiquetas. Observa que en este caso devuelve 23 filas en lugar de 4, es porque recoge todos los elementos DEPT_NO de la colección, y recuerda que en la colección también hemos incluido el documento *empleados.xml*, que tiene 14 empleados con su DEPT_NO, y *departamentosnuevo.xml* con 5 DEPT_NO.

- ***El operador*** * se usa para nombrar a cualquier etiqueta, se usa como comodín. Por ejemplo:
 - El descriptor /*/DEPT_NO selecciona las etiquetas DEPT_NO que se encuentran a 1 nivel de profundidad desde la raíz, en este caso ninguna.
 - /*/*/DEPT_NO selecciona las etiquetas DEPT_NO que se encuentran a dos niveles de profundidad desde la raíz, en este caso 23.
 - /departamentos/* selecciona las etiquetas que van dentro de la etiqueta departamentos y sus subetiquetas.
 - /* ¿Qué salida produce este descriptor?, depende del contexto en el que se ejecute. Se mostrarán todas las etiquetas de todos los documentos, es decir, se mostrarán todos los documentos de la colección.
- ***Condiciones de selección.*** Se utilizarán los corchetes para seleccionar elementos concretos, en las condiciones se pueden usar los comparadores: < >, <= , >= , = , != , or, and y not (or, and y not deben escribirse en minúscula). Se utilizará el separador | para unir varias rutas. Ejemplos:
 - /EMPLEADOS/EMP_ROW[DEPT_NO=10], selecciona todos los elementos o nodos (etiquetas) dentro de EMP_ROW de los empleados del DEPT_NO 10.
 - /EMPLEADOS/EMP_ROW/APELLIDO/EMPLEADOS/EMP_ROW/DEPT_NO selecciona los nodos APELLIDO y DEPT_NO de los empleados.
 - /EMPLEADOS/EMP_ROW[DEPT_NO=10]/APELLIDO/text(), selecciona los apellidos de los empleados del DEPT_NO=10.
 - /EMPLEADOS/EMP_ROW[not(DEPT_NO = 10)], selecciona todos los empleados (etiquetas) que NO son del DEPT_NO igual a 10.
 - /EMPLEADOS/EMP_ROW[not(OFICIO = 'ANALISTA')]/APELLIDO/text(), selecciona los APELLIDOS de los empleados que NO son ANALISTAS.
 - /EMPLEADOS/EMP_ROW[DEPT_NO=10]/APELLIDO | /EMPLEADOS/EMP_ROW[DEPT_NO=10]/OFICIO, selecciona el APELLIDO y el OFICIO de los empleados del DEPT_NO=10.
 - //*[DEPT_NO=10]/DNOMBRE/text(), /departamentos/DEP_ROW[DEPT_NO=10]/DNOMBRE/text(), estas dos consultas devuelven el nombre del departamento 10.
 - //*[OFICIO="EMPLEADO"]//EMP_ROW, devuelve los empleados con OFICIO "EMPLEADO", por cada empleado devuelve todos sus elementos. Busca en cualquier parte de la colección //.
 - Esto devuelve lo mismo:
`/EMPLEADOS/EMP_ROW[OFICIO="EMPLEADO"]//EMP_ROW.`
 - /EMPLEADOS/EMP_ROW[SALARIO>1300 and DEPT_NO=10], devuelve los datos de los empleados con SALARIO mayor de 1300 y del departamento 10.
 - /EMPLEADOS/EMP_ROW[SALARIO>1300 and DEPT_NO=20]/APELLIDO | /EMPLEADOS/EMP_ROW[SALARIO>1300 and DEPT_NO=20]/OFICIO, devuelve el APELLIDO y el OFICIO de los empleados con SALARIO mayor de 1300 y del departamento 20. Se utiliza el separador | para unir las dos rutas.

- **Utilización de funciones y expresiones matemáticas.**

- Un número dentro de los corchetes representa la posición del elemento en el conjunto seleccionado. Ejemplos:

/EMPLEADOS/EMP_ROW[1], devuelve todos los datos del primer empleado.

/EMPLEADOS/EMP_ROW[5]/APELLIDO/text(), devuelve el APELLIDO del quinto empleado.

- La función **last()** selecciona el último elemento del conjunto seleccionado. Ejemplos:

/EMPLEADOS/EMP_ROW[last()], selecciona todos los datos del último empleado.

/EMPLEADOS/EMP_ROW[last()-1]/APELLIDO/text(), devuelve el APELLIDO del penúltimo empleado.

- La función **position()** devuelve un número igual a la posición del elemento actual.

/EMPLEADOS/EMP_ROW[position()=3], obtiene los elementos del empleado que ocupa la posición 3.

/EMPLEADOS/EMP_ROW[position()=3]/APELLIDO, selecciona el apellido del los elementos cuya posición es menor de 3, es decir, devuelve los apellidos del primer y segundo empleado.

- La función **count()** cuenta el número de elementos seleccionados. Ejemplos:

/EMPLEADOS/count(EMP_ROW), devuelve el número de empleados.

/EMPLEADOS/count(EMP_ROW[DEPT_NO=10]), cuenta el nº de empleados del departamento 10.

/EMPLEADOS/count(EMP_ROW[OFICIO="EMPLEADO" and SALARIO>1300]), cuenta el nº de empleados con oficio EMPLEADO y SALARIO mayor de 1300.

//*[count(*)=3], devuelve elementos que tienen 3 hijos.

//*[count(DEP_ROW)=4], devuelve los elementos que contienen 4 hijos DEP_ROW, devolverá la etiqueta departamentos y todas las subetiquetas.

- La función **sum()** devuelve la suma del elemento seleccionado. Ejemplos:

sum(/EMPLEADOS/EMP_ROW/SALARIO), devuelve la suma del SALARIO.

Si la etiqueta a sumar la considera *string* **hay que convertirla a número utilizando la función number.**

sum(/EMPLEADOS/EMP_ROW[DEPT_NO=20]/SALARIO), devuelve la suma de SALARIO de los empleados del DEPT_NO 20.

- Función **max()** devuelve el máximo, **min()** devuelve el mínimo y **avg()** devuelve la media del elemento seleccionado. Ejemplos:

max(/EMPLEADOS/EMP_ROW/SALARIO), devuelve el salario máximo.

min(/EMPLEADOS/EMP_ROW/SALARIO), devuelve el salario mínimo.

min(/EMPLEADOS/EMP_ROW[OFICIO="ANALISTA"]/SALARIO), devuelve el salario mínimo de los empleados con OFICIO ANALISTA..

avg(/EMPLEADOS/EMP_ROW/SALARIO), devuelve la media del salario.

`avg(/EMPLEADOS/EMP_ROW[DEPT_NO=20]/SALARIO)`, devuelve la media del salario de los empleados del departamento 20.

- La función **name()** devuelve el nombre del elemento seleccionado. Ejemplos:
`/*[name()='APELLIDO']`, devuelve todos los apellidos, entre sus etiquetas.
`count(/*[name()='APELLIDO'])`, cuenta las etiquetas con nombre APELLIDO.
- La función **concat(cad1, cad2, ...)** concatena las cadenas. Ejemplos:
`/EMPLEADOS/EMP_ROW[DEPT_NO=10]/concat(APELLIDO, " - ",OFICIO)`
 Devuelve el apellido y el oficio concatenados de los empleados del departamento 10
`/EMPLEADOS/EMP_ROW/concat(APELLIDO, " - ", OFICIO, " - ", SALARIO)`
 Devuelve la concatenación de apellido, oficio y salario de los empleados.
- La función **starts-with(cad1, cad2)** es verdadera cuando la cadena cad1 tiene como prefijo a la cadena cad2. Ejemplos:
`/EMPLEADOS/EMP_ROW[starts-with(APELLIDO,'A')]`, obtiene los elementos de los empleados cuyo APELLIDO empieza por ‘A’.
`/EMPLEADOS/EMP_ROW[starts-with(OFICIO,'A')]/concat(APELLIDO," - ",OFICIO)`
 obtiene APELLIDO y NOMBRE concatenados de los empleados cuyo OFICIO empieza por ‘A’.
- La función **contains(cad1, cad2)** es verdadera cuando la cadena cad1 contiene a la cadena cad2.
`/EMPLEADOS/EMP_ROW[contains(OFICIO,'OR')]/OFICIO`, devuelve los oficios que contienen la sílaba ‘OR’ .
`/EMPLEADOS/EMP_ROW [contains(APELLIDO,'A')]/APELLIDO`, devuelve los apellidos que contienen una ‘A’.
- La función **string-length(argumento)** devuelve el número de caracteres de su argumento.
`/EMPLEADOS/EMP_ROW(concat(APELLIDO,' = ', string-length(APELLIDO)))`, devuelve concatenados el apellido con su número de caracteres.
- /EMPLEADOS/EMP_ROW[string-length(APELLIDO)<4], devuelve los datos de los empleados cuyo APELLIDO tiene menos de 4 caracteres.
- Operador matemático **div()** realiza divisiones en punto flotante.
`/EMPLEADOS/EMP_ROW/concat(APELLIDO,' , ',SALARIO,' - ',SALARIO div 12)`, devuelve los datos concatenados de APELLIDO, SALARIO y el salario dividido por 12.
`sum(/EMPLEADOS/EMP_ROW/SALARIO) div count(/EMPLEADOS/EMP_ROW)`, devuelve la suma de salarios dividido por el contador de empleados.
- Operador matemático **mod()** calcula el resto de la división
`/EMPLEADOS/EMP_ROW/concat(APELLIDO,' , ',SALARIO,' - ',SALARIO mod 12)`, devuelve los datos concatenados de APELLIDO, SALARIO y el resto de dividir el SALARIO por 12.

/EMPLEADOS/EMP_ROW[(SALARIO mod 12)=4], devuelve los datos de los empleados cuyo resto de dividir el SALARIO entre 12 sea igual a 4.

- **Otras funciones.**

- ***data(expresión XPath)***, devuelve el texto de los nodos de la expresión sin las etiquetas.
- ***number(argumento)***, para convertir a número el argumento, que puede ser cadena, booleano o un nodo.
- ***abs(num)***, devuelve el valor absoluto del número.
- ***ceiling(num)***, devuelve el entero más pequeño mayor o igual que la expresión numérica especificada.
- ***floor(num)***, devuelve el entero más grande que sea menor o igual que la expresión numérica especificada.
- ***round(num)***, redondea el valor de la expresión numérica.
- ***string(argumento)***, convierte el argumento en cadena.
- ***compare(exp1,exp2)***, compara las dos expresiones, devuelve 0 si son iguales, 1 si $exp1 > exp2$, y -1 si $exp1 < exp2$.
- ***substring(cadena,comienzo,num)***, extrae de la *cadena*, desde la posición indicada en *comienzo* el número de caracteres indicado en *num*.
- ***substring(cadena,comienzo)***, extrae de la *cadena*, los caracteres desde la posición indicada por *comienzo*, hasta el final.
- ***lower-case(cadena)***, convierte a minúscula la *cadena*.
- ***upper-case(cadena)***, convierte a mayúscula la *cadena*.
- ***translate(cadena1,caract1,caract2)***, reemplaza dentro de *cadena1*, los caracteres que se expresan en *caract1*, por los correspondientes que aparecen en *caract2*, uno por uno.
- ***ends-with(cadena1,cadena2)***, devuelve true si la *cadena1* termina en *cadena2*.
- ***year-from-date(fecha)***, devuelve el año de la fecha, el formato de fecha es AÑO-MES-DIA.
- ***month-from-date(fecha)***, devuelve el mes de la fecha.
- ***day-from-date(fecha)***, devuelve el día de la fecha.

Puedes encontrar más funciones en la URL: http://www.w3schools.com/xsl/xsl_functions.asp

ACTIVIDAD 5.1

Dado el documento *productos.xml* que está dentro de la colección *ColeccionPruebas*, con información de datos de productos, y cuya estructura es la siguiente:

```
<produc>
    <cod_prod>xxxxxx</cod_prod>
    <denominacion>xxxxxxxxxxxx</denominacion>
    <precio>xxxx</precio>
    <stock_actual>xxx</stock_actual>
    <stock_minimo>xxxx</stock_minimo>
    <cod_zona>xxxx</cod_zona>
</produc>
```

Realiza las siguientes consultas XPath:

- Obtén los nodos denominación y precio de todos los productos.
- Obtén los nodos de los productos que sean placas base.

- Obtén los nodos de los productos con precio mayor de 60 € y de la zona 20.
- Obtén el número de productos que sean memorias y de la zona 10.
- Obtén la media de precio de los micros.
- Obtén los datos de los productos cuyo stock mínimo sea mayor que su stock actual.
- Obtén el nombre de producto y el precio de aquellos cuyo stock mínimo sea mayor que su stock actual y sean de la zona 40.
- Obtén el producto más caro.
- Obtén el producto más barato de la zona 20.
- Obtén el producto más caro de la zona 10.

5.5.2.2. Nodos atributos XPath

Un nodo puede tener tantos atributos como se desee, y para cada uno se le creará un nodo atributo. Los nodos atributo NO se consideran como hijos, sino más bien como etiquetas añadidas al nodo elemento. Cada nodo atributo consta de un nombre, un valor (que es siempre una cadena) y un posible "espacio de nombres".

Partimos del documento ***universidad.xml***, que se encuentra dentro de la *ColecciónPruebas* subida en los anteriores apartados. El documento está formado por los nodos atributo: teléfono y tipo, que pertenecen al elemento departamento y salario que pertenece al elemento empleado. El documento es el siguiente:

<pre> <universidad> <departamento telefono="112233" tipo="A"> <codigo>IFC1</codigo> <nombre>Informática</nombre> <empleado salario="2000"> <puesto>Asociado</puesto> <nombre>Juan Parra</nombre> </empleado> <empleado salario="2300"> <puesto>Profesor</puesto> <nombre>Alicia Martín</nombre> </empleado> </departamento> <departamento telefono="990033" tipo="A"> <codigo>MAT1</codigo> <nombre>Matemáticas</nombre> <empleado salario="1900"> <puesto>Técnico</puesto> <nombre>Ana García</nombre> </empleado> <empleado salario="2100"> <puesto>Profesor</puesto> <nombre>Mª Jesús Ramos</nombre> </empleado> </departamento> </pre>	<pre> <empleado salario="2300"> <puesto>Profesor</puesto> <nombre>Pedro Paniagua</nombre> </empleado> <empleado salario="2500"> <puesto>Tutor</puesto> <nombre>Antonia González</nombre> </empleado> </departamento> <departamento telefono="880833" tipo="B"> <codigo>MAT2</codigo> <nombre>Análisis</nombre> <empleado salario="1900"> <puesto>Asociado</puesto> <nombre>Laura Ruiz</nombre> </empleado> <empleado salario="2200"> <puesto>Asociado</puesto> <nombre>Mario García</nombre> </empleado> </departamento> </universidad > </pre>
---	---

El árbol del documento se muestra en la Figura 5.14., los atributos se representan con elipses.



Figura 5.14. Árbol del documento *universidad.xml*

Para referirnos a los atributos de los elementos se usa `@` antes del nombre, por ejemplo, `@telefono`, `@tipo`, `@salario`. En un descriptor de ruta los atributos se nombran como si fueran etiquetas hijo pero anteponiendo `@`.

Ejemplos de consultas utilizando nodos atributo:

- `/universidad/departamento[@tipo]`, se obtienen los datos de los departamentos que tengan el atributo tipo. Si ponemos `data(/universidad/departamento[@tipo])`, nos devuelve los datos sin las etiquetas.
- `/universidad/departamento/empleado[@salario]`, se obtienen los datos de los empleados que tengan el atributo salario.
- `/universidad/departamento[@telefono="990033"]`, se obtienen los datos del departamento cuyo teléfono es 990033. Si ponemos `data(/universidad/departamento[@telefono="990033"])`, devuelve lo mismo, pero sin las etiquetas de los elementos.
- `/universidad/departamento[@telefono="990033"]/nombre/text()`, se obtienen el nombre de departamento cuyo teléfono es 990033.
- `//departamento[@tipo='B']`, se obtienen los datos de los departamentos cuyo tipo es B.
- `/universidad/departamento[@tipo="A"]/empleado`, se obtienen los datos de los empleados de los departamentos del tipo A.
- `/universidad/departamento/empleado[@salario>"2100"]`, se obtienen los datos de los empleados cuyo salario es mayor de 2100.
- `/universidad/departamento/empleado[@salario>"2100"]/nombre/text()`, se obtienen los nombres de los empleados cuyo salario es mayor de 2100.
- `/universidad/departamento/empleado[@salario>"2100"] /concat(nombre, ', @salario)`, se obtienen los datos concatenados del nombre de empleado y su salario, de los empleados cuyo salario es mayor de 2100.
- `/universidad/departamento[@tipo="A"]/count(empleado)`, devuelve el número de empleados que hay en los departamentos de tipo=A.

- `/universidad/departamento[@tipo="A"]/concat(nombre, ',count(empleado))`, devuelve por cada departamento del tipo=A, la concatenación de su nombre y el número de empleados.
 - `/universidad/departamento(concat(nombre, ',count(empleado)))`, devuelve el número de empleados por cada departamento
 - `sum(//empleado/@salario)`, devuelve la suma total del salario de todos los empleados, hace lo mismo que esto: `sum(/universidad/departamento/empleado/@salario)`
 - `/universidad/departamento(concat(nombre, ' Total=', sum(empleado/@salario)))`, obtiene por cada departamento la concatenación de su nombre y el total salario.
 - `min(//empleado/@salario)`, devuelve el salario mínimo de todos los empleados.
 - `/universidad/departamento(concat(nombre, ' Minimo=', min(empleado/@salario)))`
`/universidad/departamento(concat(nombre, ' Máximo=', max(empleado/@salario)))`
- La primera obtiene por cada departamento la concatenación de su nombre y el mínimo salario, y la segunda el máximo salario.
- `/universidad/departamento(concat(nombre, ' Media=', avg(empleado/@salario)))`, obtiene por cada departamento la concatenación de su nombre y la media de salario.
 - `/universidad/departamento[count(empleado)>3]`, obtiene los datos de departamentos con más de 3 empleados. `/universidad/departamento[count(empleado)>3]/nombre/text()`, en este caso devuelve el nombre de los departamentos con más de 3 empleados.
 - `/universidad/departamento[@tipo="A" and count(empleado)>2]/nombre/text()`, devuelve el nombre de los departamentos de tipo A y con más de 2 empleados.

ACTIVIDAD 5.2

Dado el documento *sucursales.xml* que se encuentra dentro de la colección **ColeccionPruebas**. Este documento contiene los datos de las sucursales de un banco. Por cada sucursal tenemos el teléfono, el código, el director de la sucursal, la población y las cuentas de la sucursal. Y por cada cuenta tenemos el tipo de cuenta AHORRO o PENSIONES, el nombre de la cuenta, el número, el saldo haber y el saldo debe. Estos datos son:

```
<sucursales>
  <sucursal telefono="xxxxxxxxx" codigo="xxxx">
    <director>xxxxxxxxxxxxxxxx</director>
    <poblacion>xxxxxxxxxx</poblacion>
    <cuenta tipo="xxxxxxxxx">
      <nombre>xxxx</nombre>
      <numero>xxxx</numero>
      <saldohaber>xxxxxx</saldohaber>
      <saldodebe>xxxx</saldodebe>
    </cuenta>
    . . .
  </sucursal>
. . .
</sucursales>
```

Realiza las siguientes consultas XPath:

- Obtener los datos de las cuentas bancarias cuyo tipo sea AHORRO.

- Obtener por cada sucursal la concatenación de su código, y el número de cuentas del tipo AHORRO que tiene.
 - Obtener las cuentas de tipo PENSIONES de la sucursal con código SUC3.
 - Obtener por cada sucursal la concatenación de los datos, código sucursal, director, y total saldo haber.
 - Obtener todos los elementos de las sucursales con más de 3 cuentas.
 - Obtener todos los elementos de las sucursales con más de 3 cuentas del tipo AHORRO.
 - Obtener los nodos del director y la población de las sucursales con más de 3 cuentas.
 - Obtener el número de sucursales cuya población sea Madrid.
 - Obtener por cada sucursal, su código y la suma de las aportaciones de las cuentas del tipo PENSIONES.
 - Obtener los nodos número de cuenta, nombre de cuenta y el saldo haber de las cuentas con saldo haber mayor de 10000.
 - Obtener por cada sucursal con más de 3 cuentas del tipo AHORRO, su código y la suma del saldo debe de esas cuentas.
-

5.5.2.3. Axis XPath

Un AXIS o eje, especifica la dirección que se va a evaluar, es decir, si nos vamos a mover hacia arriba en la jerarquía o hacia abajo, si va a incluir el nodo actual o no, es decir, define un conjunto de nodos relativo al nodo actual. Los nombres de los ejes son los siguientes:

Nombre de Axis	Resultado
ancestor	Selecciona los antepasados (padres, abuelos, etc.) del nodo actual
ancestor-or-self	Selecciona los antepasados (padres, abuelos, etc.) del nodo actual y el nodo actual en sí
attribute	Selecciona los atributos del nodo actual
child	Selecciona los hijos del nodo actual
descendant	Selecciona los descendientes (hijos, nietos, etc.) del nodo actual
descendant-or-self	Selecciona los descendientes (hijos, nietos, etc.) del nodo actual y el nodo actual en sí
following	Selecciona todo el documento después de la etiqueta de cierre del nodo actual
following-sibling	Selecciona todos los hermanos que siguen al nodo actual
parent	Selecciona el padre del nodo actual
self	Selecciona el nodo actual

La sintaxis para utilizar ejes es la siguiente: *Nombre_de_eje::nombre_nodo[expresión]*

En la ventana del **Diálogo de consulta** del *Cliente de Administración de eXist*, se puede ver en la parte de *Resultados*, y dentro de la pestaña *Trace*, la traza de las consultas, y en la traza podemos observar los ejes utilizados. Véase Figura 5.15.

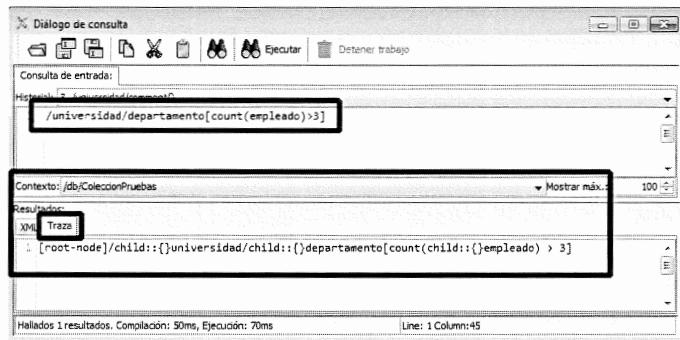


Figura 5.15. Ejes en la traza de ejecución de las consultas XPath.

Ejemplos con Axis XPath :

- /universidad/child::* , es lo mismo que /child::universidad/child::element(). Devuelve todos los hijos de universidad, es decir, los nodos de los departamentos.
 - /universidad/departamento/descendant::*, devuelve los descendientes del nodo departamento, esto hace lo mismo:
- ```
/child::universidad/child::departamento/descendant::element()
```
- /universidad/departamento/descendant::empleado , devuelve los nodos empleado descendientes de los nodos departamento.
  - /universidad/descendant::nombre , devuelve todos los elementos nombre descendientes de universidad, tanto nombres de departamentos como de empleados. Si ponemos esto nos devuelve el texto del nombre: data(/universidad/descendant::nombre).
  - /universidad/departamento/following-sibling::\*, selecciona todos los hermanos de departamento a partir del primero, siguiendo el orden en el documento.

Si ponemos /universidad/departamento[2]/following-sibling::\*, selecciona todos los hermanos de departamento a partir del segundo.

- //empleado/following-sibling::node() , selecciona todos los hermanos de los elementos empleado que encuentre en el contexto.

En este caso //empleado/following-sibling::empleado[@salario>2100], selecciona todos los hermanos de los elementos empleado que tienen el salario >2100.

- //empleado[nombre="Ana García"]/following-sibling::\* , selecciona los nodos hermanos de Ana García.
- //empleado[nombre="Ana García"]/following-sibling::empleado/nombre/text() , selecciona los nombres de los empleados hermanos de Ana García.
- //empleado[nombre="Ana García"]/following-sibling::empleado[puesto="Profesor"]/nombre/text() , selecciona los nombres de los empleados hermanos de Ana García que son profesores.
- //empleado/parent::departamento/nombre , selecciona el nombre de los padres de los elementos empleado.
- //empleado[nombre="Ana García"]/parent::departamento/nombre , selecciona el nombre del padre de la empleada Ana García.

- `/descendant::departamento[1]`, selecciona los descendientes del departamento que ocupa la posición 3 en el documento.
- `/child::universidad/child::departamento[count(child::empleado)>3]`, es lo mismo que `/universidad/departamento[count(empleado)>3]`, obtiene los departamentos con más de 3 empleados.
- `/child::universidad/child::departamento/child::nombre`, obtiene las etiquetas con los nombres de los departamentos. Es lo mismo que `/universidad/departamento/nombre`, y que `data(/universidad/departamento/nombre)`.
- `/child::universidad/child::departamento/child::nombre/child::text()`, obtiene los nombres de los departamentos.
- `/child::universidad/child::departamento[attribute::tipo = "B"] [count(child::empleado) >= 2]/child::nombre/child::text()`, devuelve el nombre de los departamentos de tipo B y con 2 o más empleados. Es lo mismo que poner `/universidad/departamento[@tipo="B" and count(empleado)>=2]/nombre/text()`.

#### 5.5.2.4. Consultas XQuery

Una consulta en XQuery es una expresión que lee datos de uno o más documentos en XML y devuelve como resultado otra secuencia de datos en XML, en la Figura 5.16. se ve el procesamiento básico de una consulta XQUERY. XQuery contiene a XPath, toda expresión de consulta en XPath es válida y devuelve el mismo resultado en XQuery. Xquery nos va a permitir:

- Seleccionar información basada en un criterio específico.
- Buscar información en un documento o conjunto de documentos.
- Unir datos desde múltiples documentos o colección de documentos.
- Organizar, agrupar y resumir datos.
- Transformar y reestructurar datos XML en otro vocabulario o estructura.
- Desempeñar cálculos aritméticos sobre números y fechas.
- Manipular cadenas de caracteres a formato de texto.

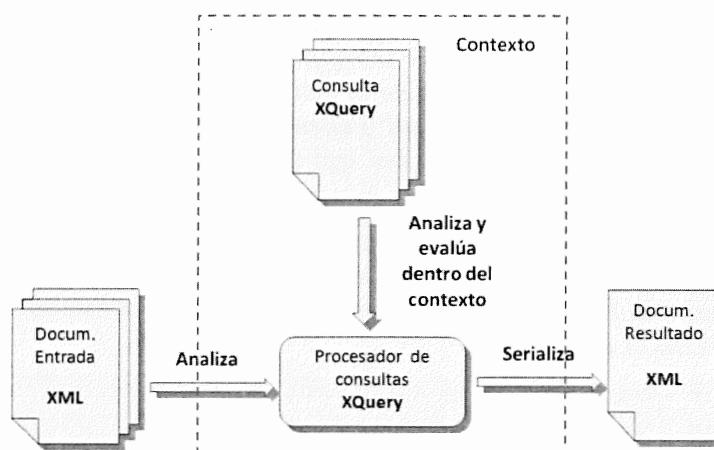


Figura 5.16. Procesamiento de una consulta XQuery.

En las consultas XQuery podemos utilizar las siguientes funciones para referirnos a colecciones y documentos dentro de la BD, son las siguientes:

- **collection("ruta")**, indicamos el camino para referirnos a una colección.
- **doc("/ruta/documento.xml")**, indicamos el camino de un documento de una colección, y el nombre del documento.

Si no indicamos esas funciones la bd busca los elementos en el contexto actual. Así por ejemplo:

1. La consulta **collection(/ColeccionPruebas)**, devuelve el contenido de la colección de ruta absoluta /ColeccionPruebas, es decir, visualiza todos los documentos incluidos en esa colección. Todas las consultas parten de la ruta /db.
2. La consulta **doc("/ColeccionPruebas/productos.xml")** devuelve el documento *productos.xml* completo que se encuentra en *ColeccionPruebas*. Esto hace lo mismo: **doc("/db/ColeccionPruebas/productos.xml")**.

Si en una colección se prevé que pueda haber varias etiquetas raíz con el mismo nombre, es muy importante al hacer consultas sobre un documento indicar el nombre del documento utilizando la función **doc()** para referirnos a él.

Otros ejemplos de consultas XQuery utilizando estas funciones:

- La consulta **collection(/ColeccionPruebas)/departamentos/DEP\_ROW** devuelve los nodos **DEP\_ROW** que cuelgan de la etiqueta raíz *departamentos*, que aparezcan dentro de la colección. Buscará todos los nodos **departamentos/DEP\_ROW** que estén dentro de la colección.
- La consulta **collection(/ColeccionPruebas)/sucursales/sucursal[@codigo='SUC1']** devuelve el nodo *sucursal* cuyo código es SUC1, y que se encuentra dentro de nodos *sucursales*, y dentro de la colección.
- **doc("/ColeccionPruebas/productos.xml")/productos/produci[precio>50]/denominación** esta consulta devuelve los productos cuyo precio sea mayor de 50. Selecciona el documento de la colección con doc, y la consulta solo se realizará para ese documento.
- **doc("/ColeccionPruebas/universidad.xml")/universidad/departamento[@tipo="A"]**, esta consulta devuelve los departamentos de tipo A, que se encuentran en el documento *universidad.xml*.
- **También podemos hacer consultas sobre ficheros XML guardados en disco.** En este caso escribiremos la ruta completa de la ubicación del archivo. Por ejemplo, si pongo: **doc("file:///D:/misXMLs/clientes.xml")/clientes/cliente/nombre**, obtengo los nombres de los clientes del documento *clientes.xml* que se encuentran en la carpeta *misXMLs* de la unidad *D*.

En XQuery las consultas se pueden construir utilizando expresiones **FLWOR** (leído como flower), que corresponde a las siglas de **For, Let, Where, Order y Return**. Permite a diferencia de XPath manipular, transformar y organizar los resultados de las consultas. La sintaxis general de una estructura FLWOR es esta:

```

for <variable> in <expresión XPath>
let <variables vinculadas>
where <condición XPath>
order by <expresión>
return <expresión de salida>
```

- **For:** se usa para seleccionar nodos y almacenarlos en una variable, similar a la cláusula from de SQL. Dentro del for escribimos una expresión XPath que seleccionará los nodos. Si se especifica más de una variable en el for se actúa como producto cartesiano. Las variables comienzan con \$.

Las consultas XQuery deben llevar obligatoriamente una orden **Return**, donde indicaremos lo que queremos que nos devuelva la consulta. Por ejemplo, estas consultas devuelven la primera los elementos EMP\_ROW, y la segunda los apellidos de los empleados. Unas escritas en XQuery y las otras en XPath:

| XQuery                                                   | XPath                       |
|----------------------------------------------------------|-----------------------------|
| for \$emp in /EMPLEADOS/EMP_ROW<br>return \$emp          | /EMPLEADOS/EMP_ROW          |
| for \$emp in /EMPLEADOS/EMP_ROW<br>return \$emp/APELLIDO | /EMPLEADOS/EMP_ROW/APELLIDO |

- **Let:** permite que se asiganen valores resultantes de expresiones XPath a variables para simplificar la representación. Se pueden poner varias líneas let una por cada variable, o separar las variables por comas.

En el siguiente ejemplo se crean 2 variables, el APELLIDO del empleado se guarda en **\$nom**, y el OFICIO en **\$ofi**. La salida sale ordenada por OFICIO, y se crea una etiqueta <APE\_OFI> </APE\_OFI> que incluye el nombre y el oficio concatenado. Se utilizarán las llaves en el return {} para añadir el contenido de las variables. Véase el ejemplo:

```
for $emp in /EMPLEADOS/EMP_ROW
let $nom:=$emp/APELLIDO, $ofi :=$emp/OFICIO
order by $emp/OFICIO
return <APE_OFI> {concat($nom, ' ', $ofi)} </APE_OFI>
```

La salida de esta consulta es:

```
<APE_OFI>GIL ANALISTA</APE_OFI>
<APE_OFI>FERNANDEZ ANALISTA</APE_OFI>
```

En este otro ejemplo se obtienen los nodos:

```
for $emp in /EMPLEADOS/EMP_ROW
let $nom:=$emp/APELLIDO, $ofi :=$emp/OFICIO
order by $emp/OFICIO
return <APE_OFI> {($nom, ' ', $ofi)} </APE_OFI>
```

La salida de esta consulta es:

```
<APE_OFI>
 <APELLIDO>GIL</APELLIDO> <OFICIO>ANALISTA</OFICIO>
</APE_OFI>
<APE_OFI>
 <APELLIDO>FERNANDEZ</APELLIDO> <OFICIO>ANALISTA</OFICIO>
</APE_OFI>
```

La cláusula let se puede utilizar sin for, prueba el siguiente caso y observa la diferencia:

| SIN FOR                                                                                                                                                                             | CON FOR                                                                                                                                                                                                           |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>let \$ofi := /EMPLEADOS/EMP_ROW/OFICIO return &lt;OFICIOS&gt;{\$ofi}&lt;/OFICIOS&gt;</pre>                                                                                     | <pre>for \$ofi in /EMPLEADOS/EMP_ROW/OFICIO return &lt;OFICIOS&gt;{\$ofi}&lt;/OFICIOS&gt;</pre>                                                                                                                   |
| <p>La cláusula let vincula la variable \$ofi con todo el resultado de la expresión. En este caso vincula todos los oficios creando un elemento &lt;OFICIOS&gt; con todos ellos.</p> | <p>La cláusula for vincula la variable \$ofi con cada nodo oficio que encuentre en la colección de datos, creando una elemento por cada oficio. Por eso aparece la etiqueta &lt;OFICIOS&gt; para cada oficio.</p> |

- **Where:** filtra los elementos, eliminando todos los valores que no cumplan las condiciones dadas.
- **Order by:** ordena los datos según el criterio dado.
- **Return:** construye el resultado de la consulta en XML, se pueden añadir etiquetas XML a la salida, si añadimos etiquetas los datos a visualizar los encerramos entre llaves {}. Además en el return se pueden añadir **condicionales usando if-then-else** y así tener más versatilidad en la salida. Si se usa la condicional, hay que tener en cuenta que la cláusula else es obligatoria y debe aparecer siempre en la expresión condicional, se debe a que toda expresión XQuery debe devolver un valor. Si no existe ningún valor a devolver al no cumplirse la cláusula if, devolvemos una secuencia vacía con else (). El siguiente ejemplo devuelve los departamentos de tipo A encerrados en una etiqueta:

```
for $dep in /universidad/departamento
return if ($dep/@tipo='A')
 then <tipoA>{data($dep/nombre)}</tipoA>
 else ()
```

Utilizaremos la función **data()** para extraer el contenido en texto de los elementos. También se utiliza **data()** para extraer el contenido de los atributos p.ej. esta consulta XPath **//empleado/@salario** es errónea, pues salario no es un nodo, pero esta otra consulta **data//empleado/@salario** devuelve los salarios.

Dentro de las asignaciones **let** en las consultas XQuery podremos utilizar expresiones del tipo: **let \$var:=//empleado/@salario** esto no da error, pero si queremos extraer los datos pondremos **let \$var:=data//empleado/@salario** o si hay que devolver el salario pondríamos **return data(\$var)**.

### Ejemplos de consultas XQuery:

|                                                                                                                                                                                                                              |                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>for \$emp in /EMPLEADOS/EMP_ROW order by \$emp/APELLIDO return if (\$emp/OFICIO='DIRECTOR') then &lt;DIRECTOR&gt;{\$emp/APELLIDO/text()}&lt;/DIRECTOR&gt; else &lt;EMPLE&gt;{data (\$emp/APELLIDO)}&lt;/EMPLE&gt;</pre> | <p>Devuelve los nombres de los empleados, los que son directores entre las etiquetas &lt;DIRECTOR&gt;&lt;/DIRECTOR&gt;, y los que no lo son entre las etiquetas &lt;EMPLE&gt;&lt;/EMPLE&gt;.</p> <pre>&lt;EMPLE&gt;ALONSO&lt;/EMPLE&gt; &lt;EMPLE&gt;ARROYO&lt;/EMPLE&gt; &lt;DIRECTOR&gt;CEREZO&lt;/DIRECTOR&gt; &lt;EMPLE&gt;FERNANDEZ&lt;/EMPLE&gt; &lt;EMPLE&gt;GIL&lt;/EMPLE&gt; &lt;DIRECTOR&gt;JIMENEZ&lt;/DIRECTOR&gt; . . . . .</pre> |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

|                                                                                                                                                                                                                                                                  |                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>for \$de in doc('file:///D:/XML/pruebaxquery/NUEVOS_DEP.xml')/NUEVOS_DEP/DEP_ROW return \$de</pre>                                                                                                                                                          | <p>Devuelve los nodos DEP_ROW de un documento ubicado en una carpeta del disco duro.</p>                                                                                                                                                                                                                                                                                                                                                                              |
| <pre>for \$prof in /universidad/departamento[@tipo='A']/empleado let \$profe:= \$prof/nombre, \$puesto:= \$prof/puesto where \$puesto='Profesor' return \$profe</pre>                                                                                            | <p>Obtiene los nombres de empleados de los departamentos de tipo A, cuyo puesto es Profesor. Esto hace lo mismo:</p> <pre>for \$prof in /universidad/departamento[@tipo='A']/empleado where \$prof/puesto='Profesor' return \$prof/nombre</pre> <p>El resultado es:</p> <pre>&lt;nombre&gt;Alicia Martín&lt;/nombre&gt; &lt;nombre&gt;Mª Jesús Ramos&lt;/nombre&gt; &lt;nombre&gt;Pedro Paniagua&lt;/nombre&gt;</pre>                                                 |
| <pre>for \$dep in /universidad/departamento return if (\$dep/@tipo='A') then &lt;tipoA&gt;{data(\$dep/nombre) } &lt;/tipoA&gt; else &lt;tipoB&gt;{data(\$dep/nombre) } &lt;/tipoB&gt;</pre>                                                                      | <p>Devuelve el nombre de departamento encerrado entre las etiquetas &lt;tipoA&gt;&lt;/tipoA&gt;, si es del tipo = A, y &lt;tipoB&gt;&lt;/tipoB&gt;, si no lo es.</p> <pre>&lt;tipoA&gt;Informática&lt;/tipoA&gt; &lt;tipoA&gt;Matemáticas&lt;/tipoA&gt; &lt;tipoB&gt;Análisis&lt;/tipoB&gt;</pre>                                                                                                                                                                     |
| <pre>for \$dep in /universidad/departamento let \$nom:= \$dep/empleado return &lt;depart&gt;{data(\$dep/nombre) } &lt;emple&gt;{count(\$nom) }&lt;/emple&gt; &lt;/depart&gt;</pre>                                                                               | <p>Obtiene los nombres de departamento y los empleados que tiene entre etiquetas:</p> <pre>&lt;depart&gt;Informática&lt;emple&gt;2&lt;/emple&gt; &lt;/depart&gt; &lt;depart&gt;Matemáticas&lt;emple&gt;4&lt;/emple&gt; &lt;/depart&gt; &lt;depart&gt;Análisis&lt;emple&gt;2&lt;/emple&gt; &lt;/depart&gt;</pre>                                                                                                                                                       |
| <pre>for \$dep in /universidad/departamento let \$emp:= \$dep/empleado let \$sal:= \$dep/empleado/@salario return &lt;depart&gt;{data(\$dep/nombre) } &lt;emple&gt;{count(\$emp) }&lt;/emple&gt; &lt;medsal&gt;{avg(\$sal)}&lt;/medsal&gt; &lt;/depart&gt;</pre> | <p>Obtiene los nombres de departamento, los empleados que tiene y la media del salario entre etiquetas:</p> <pre>&lt;depart&gt;Informática&lt;emple&gt;2&lt;/emple&gt; &lt;medsal&gt;2150&lt;/medsal&gt;&lt;/depart&gt; &lt;depart&gt;Matemáticas&lt;emple&gt;4&lt;/emple&gt; &lt;medsal&gt;2200&lt;/medsal&gt;&lt;/depart&gt; &lt;depart&gt;Análisis&lt;emple&gt;2&lt;/emple&gt; &lt;medsal&gt;2050&lt;/medsal&gt;&lt;/depart&gt;</pre>                              |
| <pre>for \$dep in /universidad/departamento let \$emp:= \$dep/empleado let \$sal:= \$dep/empleado/@salario let \$maxi := max(\$dep/empleado/@salario) let \$emplmax:= \$dep/empleado[@salario = \$maxi] return</pre>                                             | <p>Obtiene los nombres de departamento, los empleados que tiene y la media del salario entre etiquetas y el empleado con salario máximo:</p> <pre>&lt;depart&gt;Informática&lt;numemples&gt;2&lt;/numemples&gt; &lt;medsal&gt;2150&lt;/medsal&gt; &lt;salariomax&gt;2300&lt;/salariomax&gt; &lt;emplmax&gt;Alicia Martín - 2300&lt;/emplmax&gt; &lt;/depart&gt; &lt;depart&gt;Matemáticas&lt;numemples&gt;4&lt;/numemples&gt; &lt;medsal&gt;2200&lt;/medsal&gt;</pre> |

|                                                                                                                                                                                                                                                                                                           |                                                                                                                                                                                                                                                                                                                                             |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> &lt;depart&gt; {data(\$dep/nombre) } &lt;numemples&gt;{count(\$emp) } &lt;/numemples&gt; &lt;medsal&gt;{avg(\$sal) }&lt;/medsal&gt; &lt;salariomax&gt;{\$maxi}&lt;/salariomax&gt; &lt;emplemax&gt;{\$emplmax/nombre/text( )} - {data(\$emplmax/@salario) } &lt;/emplemax&gt; &lt;/depart&gt; </pre> | <pre> &lt;salariomax&gt;2500&lt;/salariomax&gt; &lt;emplemax&gt;Antonia González - 2500&lt;/emplemax&gt; &lt;/depart&gt; &lt;depart&gt;Análisis&lt;numemples&gt;2&lt;/numemples&gt; &lt;medsal&gt;2050&lt;/medsal&gt; &lt;salariomax&gt;2200&lt;/salariomax&gt; &lt;emplemax&gt;Mario García - 2200&lt;/emplemax&gt; &lt;/depart&gt; </pre> |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### 5.5.2.5. Operadores y funciones más comunes en XQuery

Las funciones y operadores soportados por XQuery prácticamente son los mismos que los soportados por XPath. Soporta operadores y funciones matemáticas, de cadenas, para el tratamiento de expresiones regulares, comparaciones de fechas y horas, manipulación de nodos XML, manipulación de secuencias, comprobación y conversión de tipos y lógica booleana. Los operadores y funciones más comunes se muestran en la siguiente lista.

- Matemáticos: +, -, \*, div (se utiliza div en lugar de la /), idiv(es la división entera), mod.
- Comparación: =, !=, <, >, <=, >=, not().
- Secuencia: union (), intersect, except.
- Redondeo: floor(), ceiling(), round().
- Funciones de agrupación: count(), min(), max(), avg(), sum().
- Funciones de cadena: concat(), string-length(), starts-with(), ends-with(), substring(), upper-case(), lower-case(), string().
- Uso general: *distinct-values()* extrae los valores de una secuencia de nodos y crea una nueva secuencia con valores únicos, eliminando los nodos duplicados. *empty()* devuelve cierto cuando la expresión entre paréntesis está vacía. Y *exists()* devuelve cierto cuando una secuencia contiene, al menos, un elemento.
- Los comentarios en XQuery van encerrados entre caras sonrientes: (*: Esto es un comentario :*)

**Ejemplos** utilizando el documento EMPLEADOS.xml:

- Los nombres de oficio que empiezan por P.

```

for $ofi in /EMPLEADOS/EMP_ROW/OFICIO
where starts-with(data($ofi), 'P')
return $ofi

```

**SALIDA:**

```
<OFICIO>PRESIDENTE</OFICIO>
```

- Obtiene los nombres de oficio y los empleados de cada oficio. Utiliza la función distinct-values para devolver los distintos oficios.

```

for $ofi in distinct-values(/EMPLEADOS/EMP_ROW/OFICIO)
let $cu:=count(/EMPLEADOS/EMP_ROW[OFICIO = $ofi])
return concat($ofi, ' = ', $cu)

```

**SALIDA:**

```
EMPLEADO = 4
VENDEDOR = 4
```

```
DIRECTOR = 3
ANALISTA = 2
PRESIDENTE = 1
```

- Obtiene el número de empleados que tiene cada departamento y la media de salario redondeada:

```
for $dep in distinct-values(/EMPLEADOS/EMP_ROW/DEPT_NO)
let $cu:=count(/EMPLEADOS/EMP_ROW[DEPT_NO =$dep])
let $sala:= round(avg(/EMPLEADOS/EMP_ROW[DEPT_NO =$dep]/SALARIO))
return concat('Departamento: ', $dep, '. Num empleos = ', $cu, '. Media salario = ', $sala)
```

SALIDA:

```
Departamento: 20. Num empleos = 5. Media salario = 2274
Departamento: 30. Num empleos = 6. Media salario = 1736
Departamento: 10. Num empleos = 3. Media salario = 2892
```

Si se desea devolver el resultado entre etiquetas pondremos en el return (p.ej):

```
return <depart><cod>{$dep}</cod> <empleos>{$cu}</empleos>
<medsal>{$sala}</medsal> </depart>
```

Y saldría así:

```
<depart><cod>20</cod><empleos>5</empleos><medsal>2274</medsal></depart>
<depart><cod>30</cod><empleos>6</empleos><medsal>1736</medsal></depart>
<depart><cod>10</cod><empleos>3</empleos><medsal>2892</medsal></depart>
```

### ACTIVIDAD 5.3

Utilizando el documento ***productos.xml***. Realiza las siguientes consultas XQuery:

- Obtén por cada zona el número de productos que tiene.
- Obtén la denominación de los productos entre las etiquetas `<zona10></zona10>` si son del código de zona 10, `<zona20></zona20>` si son de la zona 20, `<zona30></zona30>` si son de la 30 y `<zona40></zona40>` si son de la 40.
- Obtén por cada zona la denominación del o de los productos más caros.
- Obtén la denominación de los productos contenida entre las etiquetas `<placa></placa>` para los productos en cuya denominación aparece la palabra Placa Base, `<memoria></memoria>`, para los que contienen a la palabra Memoria `<micro></micro>`, para los que contienen la palabra Micro y `<otros></otros>` para el resto de productos.

Utilizando el documento ***sucursales.xml***. Realiza las siguientes consultas XQuery:

- Devuelve el código de sucursal y el número de cuentas que tiene de tipo AHORRO y de tipo PENSIONES.
- Devuelve por cada sucursal el código de sucursal, el director, la población, las suma del total debe y la suma del total haber de sus cuentas.
- Devuelve el nombre de los directores, el código de sucursal y la población de las sucursales con más de 3 cuentas.
- Devuelve por cada sucursal, el código de sucursal y los datos de las cuentas con más saldo debe.
- Devuelve la cuenta del tipo PENSIONES que ha hecho más aportación.

### 5.5.2.6. Consultas complejas con XQuery

Dentro de las consultas XQuery podremos trabajar con varios documentos XML para extraer su información, podremos incluir tantas sentencias for como se deseen, incluso dentro del return. Además, podremos añadir, borrar e incluso modificar elementos, eso sí generando un documento XML nuevo. A continuación se muestran ejemplos de diversa complejidad:

- **Joins de documentos**

- Visualizar por cada empleado del documento *empleados.xml*, su apellido, su número de departamento y el nombre del departamento que se encuentra en el documento *departamentos.xml*

```
for $emp in (/EMPLEADOS/EMP_ROW)
let $emple:= $emp/APELLIDO
let $dep:= $emp/DEPT_NO
let $dnom:= (/departamentos/DEP_ROW [DEPT_NO =$dep] /DNOMBRE)
return <res>{$emple, $dep, $dnom} </res>
```

La salida sería esta:

```
<res>
 <APELLIDO>SANCHEZ</APELLIDO>
 <DEPT_NO>20</DEPT_NO>
 <DNOMBRE>INVESTIGACION</DNOMBRE>
</res>
<res>
 <APELLIDO>ARROYO</APELLIDO>
 <DEPT_NO>30</DEPT_NO>
 <DNOMBRE>VENTAS</DNOMBRE>
</res>
<res>
 <APELLIDO>SALA</APELLIDO>
 <DEPT_NO>30</DEPT_NO>
 <DNOMBRE>VENTAS</DNOMBRE>
</res>
.
```

- Utilizando los documentos *departamentos.xml* y *empleados.xml*, obtener por cada departamento, el nombre de departamento, el número de empleados, y la media de salario.

```
for $dep in /departamentos/DEP_ROW
let $d:=$dep/DEPT_NO
let $tot:=sum(/EMPLEADOS/EMP_ROW [DEPT_NO=$d] /SALARIO)
let $cu:=count(/EMPLEADOS/EMP_ROW [DEPT_NO=$d] /EMP_NO)
return <resul>{$dep/DNOMBRE}<sumsalario>{$tot}</sumsalario>
 <totemple>{$cu}</totemple></resul>
```

La salida sería esta:

```
<resul>
 <DNOMBRE>CONTABILIDAD</DNOMBRE>
 <sumsalario>8675</sumsalario>
 <numemple>3</numemple>
</resul>
<resul>
 <DNOMBRE>INVESTIGACION</DNOMBRE>
 <sumsalario>11370</sumsalario>
```

```

<numemple>5</numemple>
</resul>
<resul>
 <DNOMBRE>VENTAS</DNOMBRE>
 <sumasalario>10415</sumasalario>
 <numemple>6</numemple>
</resul>
<resul>
 <DNOMBRE>PRODUCCION</DNOMBRE>
 <sumasalario>0</sumasalario>
 <numemple>0</numemple>
</resul>

```

- Convertir la salida de la consulta anterior, de manera que el total salario, y el total empleados, sean atributos de cada departamento. Hacemos que la salida que se cree sea una concatenación de los datos a obtener:

```

for $dep in /departamentos/DEP_ROW
let $d:=$dep/DEPT_NO
let $tot:=sum(/EMPLEADOS/EMP_ROW[DEPT_NO=$d]/SALARIO)
let $cu:=count(/EMPLEADOS/EMP_ROW[DEPT_NO=$d]/EMP_NO)
return concat('<departamento sumasalario="',$tot,'" totemple="',$cu,'">',
 data($dep/DNOMBRE), '</departamento>')

```

La salida sería esta:

```

<departamento sumasalario="8675" totemple="3">CONTABILIDAD</departamento>
<departamento sumasalario="11370" totemple="5">INVESTIGACION</departamento>
<departamento sumasalario="10415" totemple="6">VENTAS</departamento>
<departamento sumasalario="0" totemple="0">PRODUCCION</departamento>

```

- Utilizando los documentos departamentos.xml y empleados.xml, obtener por cada departamento, el nombre de empleado que más gana.

```

for $emp in /EMPLEADOS/EMP_ROW
let $d:=$emp/DEPT_NO, $nom:=$emp/APELLIDO, $sal:=$emp/SALARIO
let $ndep:=(/departamentos/DEP_ROW[DEPT_NO=$d]/DNOMBRE)
let $salmax:= max(/EMPLEADOS/EMP_ROW[DEPT_NO=$d]/SALARIO)
return
if ($sal=$salmax)
then
 <depart>{data($ndep)}<salmax>{data($sal)}</salmax><emple>{data($nom)}</emple></depart>
else ()

```

La salida sería esta:

```

<depart>VENTAS <salmax>3005</salmax><emple>NEGRO</emple></depart>
<depart>INVESTIGACION<salmax>3000</salmax><emple>GIL</emple></depart>
<depart>CONTABILIDAD<salmax>4100</salmax><emple>REY</emple></depart>
<depart>INVESTIGACION<salmax>3000</salmax><emple>FERNANDEZ</emple></depart>

```

## ACTIVIDAD 5.4

Sube a la colección *Pruebas* el documento *zonas.xml*, contiene información de las zonas donde se venden los productos del documento *productos.xml*. Utilizando estos dos documentos realiza las siguientes consultas XQuery:

- Obtén los datos denominación, precio y nombre de zona de cada producto, ordenado por nombre de zona.
- Obtén por cada zona, el nombre de zona y el número de productos que tiene.
- Obtén por cada zona, el nombre de la zona, su código y el nombre del producto con menos stock actual.

- **Utilización de varios for.**

La utilización de varios for es muy útil para consultas en documentos XML anidados, y también cuando utilizamos varios documentos unidos por una cláusula where como una combinación de tablas en SQL. Ejemplos:

- Esta consulta visualiza por cada departamento del documento *universidad.xml*, el número de empleados que hay en cada puesto de trabajo. Utilizamos un for para obtener los nodos departamento, y el segundo for para obtener los distintos puestos de cada departamento.

```
for $dep in /universidad/departamento
for $pue in distinct-values($dep/empleado/puesto)
let $cu:=count($dep/empleado[puesto=$pue])
return <depart>{data($dep/nombre)}<puesto>{data($pue)}</puesto>
 <profes>{$cu}</profes></depart>
```

La salida sería esta:

```
<depart>Informática<puesto>Asociado</puesto><profes>1</profes></depart>
<depart>Informática<puesto>Profesor</puesto><profes>1</profes></depart>
<depart>Matemáticas<puesto>Técnico</puesto><profes>1</profes></depart>
<depart>Matemáticas<puesto>Profesor</puesto><profes>2</profes></depart>
<depart>Matemáticas<puesto>Tutor</puesto><profes>1</profes></depart>
<depart>Análisis<puesto>Asociado</puesto><profes>2</profes></depart>
```

- Esta consulta visualiza por cada departamento del documento *universidad.xml*, el salario máximo y el empleado que tiene ese salario. El primer for obtiene los nodos departamento, y el segundo for los empleados de cada departamento. Para sacar el máximo en la salida preguntamos si el salario es el máximo.

```
for $dep in /universidad/departamento
for $emp in $dep/empleado
let $emple:= $emp/nombre
let $sal:= $emp/@salario
return if ($sal = $dep/max(empleado/@salario))
 then
 <depart>{data($dep/nombre)}<salamax>{data($sal)}</salamax>
 <empleado>{data($emple)}</empleado></depart>
 else ()
```

También se pueden poner los dos for de la siguiente manera:

```
for $dep in /universidad/departamento, $emp in $dep/empleado
```

La salida sería esta:

```
<depart>Informática
 <salamax>2300</salamax>
 <empleado>Alicia Martín</empleado>
</depart>
```

```

<depart>Matemáticas
 <salamax>2500</salamax>
 < empleado>Antonia González</ empleado>
</depart>
<depart>Análisis
 <salamax>2200</salamax>
 < empleado>Mario García</ empleado>
</depart>

```

- Esta consulta visualiza por cada puesto del documento *universidad.xml*, el empleado con salario máximo, y ese salario. El primer for obtiene los distintos puestos de trabajo, y el segundo for obtiene los empleados que tienen ese puesto de trabajo. En el if se pregunta si el salario del empleado es el salario máximo de los empleados del oficio del primer for.

```

for $pue in distinct-values (/universidad/departamento/empleado/puesto)
for $emp in /universidad/departamento/empleado[puesto=$pue]
let $sal:= $emp/@salario
let $nom:= $emp/nombre
return
if ($sal = max(/universidad/departamento/empleado[puesto=$pue] /@salario))
 then
<puesto>{data($pue)}<maxsalario>{data($sal)}</maxsalario>
 < emple>{data($nom)} </emple> </puesto>
 else ()

```

La salida sería esta:

```

<puesto>Asociado<maxsalario>2200</maxsalario>
 < emple>Mario García</emple></puesto>
<puesto>Profesor<maxsalario>2300</maxsalario>
 < emple>Alicia Martín</emple></puesto>
<puesto>Profesor<maxsalario>2300</maxsalario>
 < emple>Pedro Paniagua</emple></puesto>
<puesto>Técnico<maxsalario>1900</maxsalario>
 < emple>Ana García</emple></puesto>
<puesto>Tutor<maxsalario>2500</maxsalario>
 < emple>Antonia González</emple></puesto>

```

- También podemos resolver la consulta utilizando un solo for, de la siguiente manera:

```

for $emp in (/universidad/departamento/empleado)
let $pue:=$emp/puesto
let $sal:= $emp/@salario
let $nom:= $emp/nombre
order by $pue
return
if ($sal = max(/universidad/departamento/empleado[puesto=$pue] /@salario))
 then
 <puesto>{data($pue)}<maxsalario>{data($sal)}</maxsalario>
 < emple>{data($nom)}</emple> </puesto>
 else ()

```

- La primera consulta del apartado anterior la podemos escribir con dos for y el where:

```

for $emp in (/EMPLEADOS/EMP_ROW), $dep in /departamentos/DEP_ROW
let $emple:= $emp/APELLIDO
let $d:= $emp/DEPT_NO
where data($d) = data($dep/DEPT_NO)

```

```
return <res>{$emple, $d} {$dep/DNOMBRE} </res>
```

## ACTIVIDAD 5.5

Utiliza el documento *sucursales.xml* para realizar las siguientes consultas XQuery:

- Obtén por cada sucursal el mayor saldo haber y el nombre de la cuenta que tiene ese saldo.
- Obtén por cada sucursal el nombre de la cuenta del tipo AHORRO cuyo saldo debe sea el máximo. Obtén también el máximo.

Utiliza los documentos *productos.xml* y *zonas.xml*

- Visualiza los nombres de productos con su nombre de zona. Utiliza dos for en la consulta.
- Visualiza los nombres de productos con stock\_minimo > 5. su código de zona, su nombre y el director de esa zona. Utiliza dos for en la consulta.

### 5.5.2.7. Sentencias de actualización de eXist

Estas sentencias permiten hacer altas, bajas y modificaciones de nodos y elementos en documentos XML. Se pueden usar las sentencias de actualización en cualquier punto pero si se utiliza en la cláusula RETURN de una sentencia FLWOR, el efecto de la actualización es inmediato.

Todas las sentencias de actualización comienzan con la palabra **UPDATE** y a continuación la instrucción. Son las siguientes:

- **Insert**, se utiliza para insertar nodos. El lugar de inserción se especifica con: **into** (el contenido se añade como último hijo de los nodos especificados); **following** (el contenido se añade inmediatamente después de los nodos especificados), o **preceding** (el contenido se añade antes de los nodos especificados). El formato es:

```
update insert ELEMENTO into EXPRESIÓN XPATH
update insert ELEMENTO following EXPRESIÓN XPATH
update insert ELEMENTO preceding EXPRESIÓN XPATH
```

<pre>update insert &lt;zona&gt;&lt;cod_zona&gt;50&lt;/cod_zona&gt; &lt;nombre&gt;Madrid-OESTE &lt;/nombre&gt; &lt;director&gt;Alicia Ramos Martín&lt;/director&gt; &lt;/zona&gt; into /zonas</pre>	<p>Inserta una zona en <i>zonas.xml</i>, en la última posición</p>
<pre>update insert &lt;cuenta tipo="PENSIONES"&gt;&lt;nombre&gt;Alberto Morales &lt;/nombre&gt;&lt;numero&gt;30302900&lt;/numero&gt; &lt;aportacion&gt;5000&lt;/aportacion&gt;&lt;/cuenta&gt; into /sucursales/sucursal [@codigo="SUC1"]</pre>	<p>Inserta una cuenta en el documento <i>sucursales.xml</i> del tipo PENSIONES a la sucursal SUC1</p>
<pre>for \$de in doc('file:///D:/XML/pruebaxquery/NUEVOS_DEP.xml') /NUEVOS_DEP/DEP_ROW return update insert \$de into /departamentos</pre>	<p>Inserta en el documento departamentos de la BD los nodos DEP_ROW del documento externo <i>NUEVOS_DEP.xml</i> ubicado en la carpeta D:/XML/pruebaxquery/</p>

- **Replace**, sustituye el nodo especificado en NODO con VALOR\_NUEVO (véase formato). NODO debe devolver un único ítem: si es un elemento, VALOR\_NUEVO debe ser también un elemento. Si es un nodo de texto o atributo su valor será actualizado con la concatenación de todos los valores de VALOR\_NUEVO.

`update replace NODO with VALOR_NUEVO`

<pre>update replace /zonas/zona[cod_zona=50]/director with &lt;directora&gt;Pilar Martín Ramos &lt;/directora&gt;</pre>	Cambia la etiqueta director de la zona 50 y su contenido, en el documento zonas.xml
<pre>update replace /departamentos/DEP_ROW[DEPT_NO=10] with &lt;DEPT_ROW&gt;&lt;DEPT_NO&gt;10&lt;/DEPT_NO&gt; &lt;DNOMBRE&gt;NUEVO10&lt;/DNOMBRE&gt; &lt;LOC&gt;TALAVERA&lt;/LOC&gt;&lt;/DEPT_ROW&gt;</pre>	Cambia el nodo completo DEP_ROW del departamento 10, por los nuevos datos y las etiquetas que escribamos

- **Value**, actualiza el valor del nodo especificado en NODO con VALOR\_NUEVO. Si NODO es un nodo de texto o atributo su valor será actualizado con la concatenación de todos los valores de VALOR\_NUEVO.

`update value NODO with 'VALOR_NUEVO'`

<pre>update value /EMPLEADOS/EMP_ROW[EMP_NO=7369]/APELLIDO with 'Alberto Montes Ramos'</pre>	Cambia el apellido del empleado 7369, del documento empleados.xml
<pre>update value /sucursales/sucursal[@codigo='SUC3']/cuenta[1]/@tipo with 'NUEVOTIPO'</pre>	Cambia el atributo tipo de la primera cuenta de la sucursal SUC3, del documento sucursales.xml
<pre>for \$em in /EMPLEADOS/EMP_ROW[DEPT_NO=10] let \$sal := \$em/SALARIO return update value \$em/SALARIO with data(\$sal)+200</pre>	Cambia el salario de los empleados del departamento10, del documento empleados.xml, subirles 200

- **Delete**, elimina los nodos indicados en la expresión: `update delete expr xpath`

<pre>update delete /zonas/zona[cod_zona=50]</pre>	Elimina la zona con código 50, en el documento zonas.xml
---------------------------------------------------	----------------------------------------------------------

- **Rename**. Renombra los nodos devueltos en NODO (debe devolver una relación de nodos o atributos) por el NUEVO\_NOMBRE.

`update rename NODO as NUEVO_NOMBRE`

<pre>update rename /EMPLEADOS/EMP_ROW as 'fila_emple'</pre>	Cambia de nombre el nodo EMP_ROW del documento empleados.xml
-------------------------------------------------------------	--------------------------------------------------------------

---

### ACTIVIDAD 5.6.

A partir del documento *universidad.xml*

- Añade un empleado al departamento que ocupa la posición 2. Los datos son el salario: 2340, el puesto: Técnico, y el nombre: Pedro Fraile.
  - Actualiza el salario de los empleados del departamento con código MAT1. Suma al salario 100.
  - Renombra el nodo DEP\_ROW del documento *departamentos.xml* por *filadepar*.
- 

### 5.5.3. Acceso a eXist desde Java

Ya se ha estudiado en la UNIDAD 1 cómo leer documentos XML, accediendo a su estructura y contenido utilizando los parser o analizadores **DOM** (Modelo de Objetos de Documento) y **SAX** (API Simple para XML). En este apartado vamos a ver distintas APIs que acceden a la BD eXist para procesar documentos XML. Estas son:

- **API XML:DB**, cuyo objetivo es la definición de un método común de acceso a SGBD XML, permitiendo la consulta, creación y modificación de contenido. La última actualización de este trabajo es del año 2001 y la actividad desde el año 2005 es prácticamente nula. Sin embargo, proporciona clases bastante útiles para el manejo de colecciones y documentos.
- **API XQJ**: es una propuesta de estandarización de interfaz Java para el acceso a bases de datos XML nativas basado en el modelo de datos XQuery. El objetivo es conseguir un método sencillo y estable de acceso a bases de datos XML nativos. Es una api similar a JDBC para bases de datos relacionales.

#### 5.5.3.1. La API XML:DB para bases de datos XML

La estructura de XML:DB gira en torno a los siguientes componentes básicos:

- **Los drivers** son implementaciones de la interfaz de base de datos que encapsula la lógica de acceso a la base de datos XML. Los proporciona el proveedor del producto y debe ser registrado con el gestor de la base de datos. Ejemplo:

```
String driver = "org.exist.xmldb.DatabaseImpl"; //Driver para eXist
Class cl = Class.forName(driver); //Cargar del driver
Database database = (Database) cl.newInstance(); //Instancia de la BD
DatabaseManager.registerDatabase(database); //Registro del driver
```

- Una **colección** es un contenedor de recursos y otras subcolecciones. La API define dos recursos diferentes: **XMLResource** y **BinaryResource**. Un XMLResource representa un documento XML o un fragmento del documento, seleccionados por la ejecución de una consulta XPath. Una vez utilizado el recurso se debe cerrar. Para conectarnos a una colección lo escribimos así (en URI indicamos la colección):

```

String URI="xmldb:exist://localhost:8080/exist/xmlrpc/db/Pruebas";
//Colección
String usu="admin"; //Usuario
String usuPwd="admin"; //Clave
Collection col = DatabaseManager.getCollection(URI, usu, usuPwd);

```

- Los servicios se solicitan para tareas como consultar una colección con XPath, o la gestión de una colección. Ejemplo:

```

XPathQueryService servicio =
 (XPathQueryService) col.getService("XPathQueryService", "1.0");
ResourceSet result = servicio.query("for $em in /EMPLEADOS/EMP_ROW
return $em");

```

Internamente, eXist no distingue entre las expresiones XPath y XQuery. *XPathQueryService* y *XQueryService* son lo mismo, aunque la segunda proporciona algunos métodos adicionales.

Para ejecutar la consulta llamaremos al método *nombre\_servicio.query(xpath)*, este método devuelve un *ResourceSet*, que contiene el recurso, en el ejemplo anterior el resultado de la consulta es devuelto en *result*. El siguiente código lo escribiremos para recorrer el recurso *result* devuelto por la consulta anterior:

```

ResourceIterator i; //se utiliza para recorrer un set de recursos
i = result.getIterator();
if (!i.hasMoreResources()){
 System.out.println(" LA CONSULTA NO DEVUELVE NADA.");
}
while (i.hasMoreResources()) {
 Resource r = i.nextResource();
 System.out.println((String) r.getContent());
}

```

Donde *result.getIterator()* nos da un iterador sobre el recurso, cada recurso contiene el valor seleccionado por la expresión XPath. El método *getContent()*, devuelve el contenido del recurso, en nuestro ejemplo devuelve la consulta y el tipo es String.

Para localizar si existe un documento en una colección utilizaremos este código:

```

Collection col= DatabaseManager.getCollection(URI, usu, usuPwd);
XMLResource res = null;
res = (XMLResource)col.getResource("empleados.xml");
if(res == null)
 System.out.println("NO EXISTE EL DOCUMENTO");

```

- **Resource**: representa un recurso, un archivo de datos, hay dos tipos:
  - **XMLResource**: representa un documento XML o parte de un documento obtenido con una consulta. Es el recurso que más utilizaremos en nuestros ejercicios.
  - **BinaryResource**: que representa una secuencia de información binaria, como un mapa de bits.

Para realizar un programa que consulte la BD eXist debemos incluir las siguientes librerías: *exist.jar*, *exist-optional.jar*, *xmldb.jar*, *xml-apis-1.3.04.jar*, *xmlrpc-client-3.1.1.jar*, *xmlrpc-common-3.1.1.jar* situadas en la instalación de eXist y *log4j-1.2.15.jar*.

En el siguiente ejemplo vemos cómo utilizar las clases vistas anteriormente, este ejercicio realiza una conexión con la BD para acceder al documento *empleados.xml* y obtener en XML los empleados del departamento 10.

```
import org.xmldb.api.*;
import org.xmldb.api.base.*;
import org.xmldb.api.modules.*;

public static void verempleados10() throws XMLDBException {
 String driver = "org.exist.xmldb.DatabaseImpl"; //Driver para eXist
 Collection col = null; // Colección
 //Datos para la connexion a la colección URI, usuario y pasword.
 String URI =
 "xmlDb:exist://localhost:8083/exist/xmlrpc/db/ColeccionPruebas";
 String usu="admin"; //Usuario
 String usuPwd="admin"; //Clave
 try {
 Class cl = Class.forName(driver);
 Database database =(Database) cl.newInstance();
 DatabaseManager.registerDatabase(database);
 col = DatabaseManager.getCollection(URI, usu, usuPwd);
 if(col == null)
 System.out.println(" *** LA COLECCION NO EXISTE. ***");
 XPathQueryService servicio = (XPathQueryService)
 col.getService("XPathQueryService", "1.0");
 ResourceSet result = servicio.query (
 "for $em in /EMPLEADOS/EMP_ROW[DEPT_NO=10] return $em");
 System.out.println(" Se han obtenido " + result.getSize()
 + " elementos.");
 // Recorrer los datos del recurso.
 ResourceIterator i;
 i = result.getIterator();
 if (!i.hasMoreResources())
 System.out.println(" LA CONSULTA NO DEVUELVE NADA.");
 while (i.hasMoreResources()) {
 Resource r = i.nextResource();
 System.out.println((String) r.getContent());
 }
 col.close();
 } catch (Exception e) {
 System.out.println("Error al inicializar la BD eXist");
 e.printStackTrace();
 }
} // FIN verempleados10
```

---

### ACTIVIDAD 5.7.

- Realiza los cambios necesarios al ejercicio anterior para leer de teclado un departamento y visualizar sus empleados. Utiliza la entrada estándar.
- Realiza un programa Java que inserte, elimine y modifique departamentos del documento *departamentos.xml*. Utiliza las *Sentencias de actualización de eXist*. Los datos se leerán de la entrada estándar de teclado. Hacer que la función *main()* llame y ejecute los siguientes métodos (no devuelven nada):
  - *Insertadep()*, este método leerá de teclado un departamento, su nombre y su localidad, y deberá añadirlo al documento. Si el código de departamento existe visualiza que no se puede insertar porque ya existe.

- **Borradep()**, este método leerá de teclado un departamento, y deberá borrarlo si existe, si no existe visualiza que no se puede borrar porque ya existe.
  - **Modificadep()**, este método leerá de teclado un departamento, su nombre nuevo y la localidad nueva y deberá actualizar todos los datos si existe, si no existe visualiza que no se puede modificar porque ya existe.
- 

## Operaciones sobre colecciones y documentos

La API **XML:DB** nos va a permitir, además de consultar documentos en la BD, crear y eliminar colecciones, y crear y eliminar documentos.

- **Crear una colección:** para crear una nueva colección, se llama al método **createCollection** del servicio **CollectionManagementService**. El siguiente ejemplo crea la colección NUEVA\_COLECCION dentro de la colección col:

```
CollectionManagementService cserv = (CollectionManagementService)
 col.getService("CollectionManagementService", "1.0");
cserv.createCollection("NUEVA_COLECCION");
```

- **Borrar una colección:** para borrar utilizamos el método **removeCollection**:

```
cserv = (CollectionManagementService)
 col.getService("CollectionManagementService", "1.0");
cserv.removeCollection("NUEVA_COLECCION");
```

- **Crear un nuevo documento:** este ejemplo añade un documento nuevo a la colección col, el documento se llama NUEVOS\_DEP.xml, y se encuentra en nuestro disco, en la carpeta donde está el programa. Utilizamos el paquete **java.io.File**, para declarar el archivo a subir a la BD, y el método **createResource** para crear el recurso.

```
import java.io.File;
. . .
File archivo= new File("NUEVOS_DEP.xml");
if(!archivo.canRead()) System.out.println("ERROR AL LEER EL FICHERO");
else
{ Resource nuevoRecurso = col.createResource
 (archivo.getName(), "XMLResource");
nuevoRecurso.setContent(archivo); //Asigno el archivo
col.storeResource(nuevoRecurso); //Lo almaceno en la colección
}
```

- **Borrar un documento de la colección:** este ejemplo borra el docuemto creado anteriormente, comprueba si existe. Se utiliza el método **removeResource**:

```
try
{ Resource recursoParaBorrar = col.getResource("NUEVOS_DEP.xml");
 col.removeResource(recursoParaBorrar);
} catch(NullPointerException e)
{ System.out.println("No se puede borrar. No se encuentra."); }
```

Este método visualiza el número de colecciones, los nombres de las colecciones de la base de datos, los documentos XML de las colecciones y el contenido de los documentos:

```
public static void verrecursosdelascolecciones() {
String driver = "org.exist.xmldb.DatabaseImpl";
try {
 Class cl = Class.forName(driver);
 Database database = (Database) cl.newInstance();
```

```
DatabaseManager.registerDatabase(database);
String URI = "xmldb:exist://localhost:8083/exist/xmlrpc/db/";
String usu = "admin"; String usuPwd = "admin";
Collection col = DatabaseManager.getCollection(URI, usu, usuPwd);
System.out.println("Número de colecciones: " +
 col.getChildCollectionCount());
//Se carga la lista de colecciones en un array de cadenas
String[] colecciones = col.listChildCollections();
for (int j = 0; j < colecciones.length; j++) {
 System.out.println("-----");
 System.out.println(colecciones[j]);
 //Se extrae una colección
 Collection colecc = col.getChildCollection(colecciones[j]);
 // Se cargan los recursos de la colección en un array
 String[] lista = colecc.listResources(); //Lista de recursos
 for (int i = 0; i < lista.length; i++) {
 //Se extrae el recurso y se visualiza
 Resource res = (Resource) colecc.getResource(lista[i]);
 System.out.println("ID del documento: " + res.getId());
 System.out.println("Contenido del documento:\n" +
 res.getContent());
 }
}
} catch (ClassNotFoundException ex) {
 System.out.println(" ERROR EN EL DRIVER. COMPRUEBA CONECTORES.");
} catch (IllegalAccessException ex) {
 System.out.println("Error Instancia-cl.newInstance()");
} catch (XMLDBException ex) {ex.printStackTrace();
} catch (InstantiationException ex) {ex.printStackTrace(); }
```

---

## ACTIVIDAD 5.8.

- Haz un programa Java que cree la colección GIMNASIO y suba los documentos que se encuentran en la carpeta **ColeccionGimnasio**. Los documentos son los siguientes:
  - *socios\_gim.xml* contiene información de los socios que asisten a hacer deporte en un Gimnasio.
  - *actividades\_gim.xml* contiene información de las actividades que se pueden realizar en el Gimnasio. Hay 3 tipos de actividades:
    - Tipo 1: son actividades de libre horario, el socio no paga cuota adicional por ellas, por ejemplo: aparatos o piscina.
    - Tipo 2: representan actividades que se realizan en grupo, como por ejemplo: aerobic o pilates. El socio paga una cuota adicional de 2€ por cada hora que dedique a la actividad.
    - Tipo 3: representan actividades en las que se alquila un espacio, por ejemplo pádel o tenis. El socio paga una cuota adicional de 4€ por cada hora que dedique a la actividad.
  - *Uso\_gimnasio.xml*, contiene las actividades que realizan los socios en el Gimnasio durante el año, cada fila representa una actividad realizada por el socio con la fecha

(dd/mm/yy), la hora de inicio (por ejemplo 17) y la hora de finalización (por ejemplo 18).

- A partir de esos documentos haz un método java para obtener por cada socio la cuota que tiene que pagar. Obtén el CODSOCIO y la CUOTA\_FINAL.

Esta CUOTA\_FINAL será igual a la suma de la CUOTA\_FIJA y las CUOTAS ADICIONALES que dependerán de las actividades realizadas por el socio.

El programa Java debe crear un documento XML intermedio con nombre **socios\_cuotaadicional.xml** (la raíz del documento llamadla **socios\_cuotaadicional**) que calcule la cuota adicional a obtener por cada actividad realizada por cada usuario, el documento debe contener estas etiquetas:

```
<datos><COD>xxxxxx</COD><NOMBRESOCIO>xxxxxxxxxx</NOMBRESOCIO>
<CODACTIV>xxxxxxxx</CODACTIV><NOMBREACTIVIDAD>xxxxxxxxxx</NOMBREACTIVIDAD>
<horas>xxxx</horas><tipoact>xxxxxx</tipoact><cuota_adicional>xxxxxx</cuota_adicional>
</datos>
```

Añade el documento **socios\_cuotaadicional.xml** a la colección GIMNASIO. Una vez creado y añadido el documento, el programa Java debe recorrer ese documento para obtener por cada socio la cuota final total, que será la suma de las cuotas adicionales de las actividades mas la cuota fija. Obtén las siguientes etiquetas:

```
<datos>
 <COD>xxxxxx</COD>
 <NOMBRESOCIO>xxxxxxxxxxxxx</NOMBRESOCIO><CUOTA_FIJA>xxxxxx</CUOTA_FIJA>
 <suma_cuota_adic>xxxxxx</suma_cuota_adic><cuota_total>xxxxxxxx</cuota_total>
</datos>
```

### Localizar un documento en la bd y bajar el documento a un archivo en disco:

Para localizar si existe un documento en una colección utilizaremos el método **getResource**. En el método de ejemplo *bajardocumento*, se busca el documento **zonas.xml** de la colección **BDProductosXML** (previamente hay que subir esta colección). Si no existe, el método devuelve null, y si existe devuelve el documento. El documento devuelto lo almacenamos en un objeto **DOM** (*Modelo de Objetos de Documento*), luego se visualiza en consola y finalmente se copia a un archivo en disco de nombre **zonas.xml**.

Se va a necesitar los siguientes *import*:

- Del paquete **javax.xml.transform**, donde se encuentran las clases para transformar el árbol DOM al fichero XML (se utilizaron en la unidad 1):

```
import javax.xml.transform.Result;
import javax.xml.transform.Source;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerConfigurationException;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
```

- Del paquete **org.w3c.dom** la clase **Node** (**import org.w3c.dom.Node;**), necesaria para crear el documento DOM:

```
Node document = (Node) res.getContentAsDOM();
```

```
Source source = new DOMSource(document);
```

- Y de la api XMLDB se necesitan:

```
import org.xmldb.api.DatabaseManager;
import org.xmldb.api.base.Collection;
import org.xmldb.api.base.Database;
```

El método es el siguiente:

```
public static void bajardocumento() throws TransformerConfigurationException,
TransformerException {
 //Localizar un documento, extraerlo y guardararlo en disco
 String driver = "org.exist.xmldb.DatabaseImpl"; //Driver para eXist
 try {
 Class cl = Class.forName(driver); //Cargar del driver
 Database database = (Database) cl.newInstance(); //Instancia de la BD
 DatabaseManager.registerDatabase(database); //Registro del driver
 String URI =
 "xmldb:exist://localhost:8083/exist/xmlrpc/db/BDProductosXML";
 String usu = "admin"; //Usuario
 String usuPwd = "admin"; //Clave
 Collection col = DatabaseManager.getCollection(URI, usu, usuPwd);
 XMLResource res = (XMLResource) col.getResource("zonas.xml");
 if (res == null) {
 System.out.println("NO EXISTE EL DOCUMENTO");
 } else {
 System.out.println("ID del documento: " + res.getDocumentId());
 //Volcado del documento a un árbol DOM
 Node document = (Node) res.getContentAsDOM();
 Source source = new DOMSource(document);
 // Volcado del documento de memoria a consola
 Transformer transformer =
 TransformerFactory.newInstance().newTransformer();
 Result console = new StreamResult(System.out);
 transformer.transform(source, console);
 //Volcado del documento a un fichero
 Result fichero = new StreamResult(new java.io.File("./zonas.xml"));
 transformer = TransformerFactory.newInstance().newTransformer();
 transformer.transform(source, fichero);
 }
 } catch (ClassNotFoundException ex) {
 System.out.println(" ERROR EN EL DRIVER. COMPRUEBA CONECTORES.");
 } catch (InstantiationException ex) {
 System.out.println("Error al crear Instancia de la BD " +
 "(Database) cl.newInstance()");
 } catch (IllegalAccessException ex) {
 System.out.println("Error al crear Instancia de la BD " +
 "(Database) cl.newInstance()");
 } catch (XMLDBException ex) {
 ex.printStackTrace();
 }
}
```

### Ejecutar consultas almacenadas en ficheros

El siguiente método ejecuta una consulta que se encuentra almacenada en un fichero. El fichero se llama *miconulta.xq*, y se encuentra en la carpeta del proyecto. El método recibe como parámetro el nombre del fichero. El fichero será de texto y lo leemos hasta el final, lo vamos guardando en una cadena y una vez leído el fichero ejecutaremos la consulta.

```
public static void ejecutarconsultaarchivo(String fichero) {
 String driver = "org.exist.xmldb.DatabaseImpl"; //Driver para eXist
```

```

try {
 Class cl = Class.forName(driver);
 Database database = (Database) cl.newInstance();
 DatabaseManager.registerDatabase(database);
 String URI=
 "xmlDb:exist://localhost:8083/exist/xmlrpc/db/BDProductosXML";
 String usu = "admin"; //Usuario
 String usuPwd = "admin"; //Clave
 Collection col = DatabaseManager.getCollection(URI, usu, usuPwd);
 // Leer el fichero y guardarla en una cadena
 System.out.println("Convirtiendo el fichero a cadena...");
 BufferedReader entrada = new BufferedReader(new FileReader(fichero));
 String linea = null;
 StringBuilder stringBuilder = new StringBuilder();
 String salto = System.getProperty("line.separator"); //El salto linea
 while ((linea = entrada.readLine()) != null) {
 stringBuilder.append(linea);
 stringBuilder.append(salto);
 }
 String consulta = stringBuilder.toString();
 System.out.println("Consulta: " + consulta);
 // Ejecutar consulta
 XPathQueryService servicio = (XPathQueryService)
 col.getService("XPathQueryService", "1.0");
 ResourceSet result = servicio.query(consulta);
 ResourceIterator i; //se utiliza para recorrer un set de recursos
 i = result.getIterator();
 if (!i.hasMoreResources()) {
 System.out.println(" LA CONSULTA NO DEVUELVE NADA.");
 }
 while (i.hasMoreResources()) {
 Resource r = i.nextResource();
 System.out.println("Elemento: " + (String) r.getContent());
 }
} catch (ClassNotFoundException ex) {
 System.out.println("ERROR EN EL DRIVER.");
} catch (InstantiationException ex) {
 System.out.println("ERROR AL CREAR LA INSTANCIA.");
} catch (IllegalAccessException ex) {
 System.out.println("ERROR AL CREAR LA INSTANCIA.");
} catch (XMLDBException ex) {
 System.out.println("ERROR AL OPERAR CON EXIST.");
} catch (FileNotFoundException ex) {
 System.out.println("El fichero no se localiza: " + fichero);
} catch (IOException ex) { ex.printStackTrace(); }
}
}

```

### 5.5.3.2. La API XQJ (XQuery)

La API XQJ es una propuesta de estandarización de interfaz Java para el acceso a bases de datos XML nativas basado en el modelo de datos XQuery. El objetivo es conseguir un método sencillo y estable de acceso a bases de datos XML nativos, este estándar es un estándar independiente del fabricante, soporta al estándar XQuery 1.0 y muy fácil de utilizar. El inconveniente es que solo se pueden hacer consultas, no se puede operar con colecciones y documentos.

Al igual que en JDBC la filosofía gira en torno al origen de datos y la conexión a este, y partiendo de la conexión poder lanzar peticiones al sistema. Para descargar la API accedemos a la URL: <http://xqj.net/exist/>.

Para trabajar con esta API necesitamos los siguientes import:

```
import javax.xml.xquery.XQConnection;
import javax.xml.xquery.XQDataSource;
import javax.xml.xquery.XQException;
import javax.xml.xquery.XQPreparedExpression;
import javax.xml.xquery.XQResultItem;
import javax.xml.xquery.XQResultSequence;
import net.xqj.exist.ExistXQDataSource;
```

### Configurar una conexión

- **XQDataSource**: identifica una fuente física de datos a partir de la cual crear conexiones; cada implementación definirá las propiedades necesarias para efectuar la conexión, siendo básicas las propiedades user y password.

Por ejemplo, este código realiza la conexión con eXist, hay que poner el nombre del servidor (Localhost), el puerto de la BD (8083), el usuario (admin) y su password (admin). Aunque obligatoria es solo la propiedad *serverName* si estamos en local.

```
XQDataSource server = new ExistXQDataSource();
server.setProperty("serverName", "localhost");
server.setProperty("port", "8083");
server.setProperty("user", "admin");
server.setProperty("password", "admin");
```

- **XQConnection**: representa una sesión con la base de datos, manteniendo información de estado, transacciones, expresiones ejecutadas y resultados. Se obtiene a través de un XQDataSource, en nuestro ejemplo es *server*. Ejemplo:

```
XQConnection conn = server.getConnection();
```

También se puede indicar el usuario y password que abre la sesión:

```
XQConnection conn = server.getConnection("admin", "admin");
```

La conexión la cerramos escribiendo *conn.close()*;

### Clases y métodos para procesar los resultados de una consulta:

- **XQExpression**: objeto creado a partir de una conexión para la ejecución de una expresión una vez, retornando un *XQResultSetSequence* con los datos obtenidos, podemos decir que en un paso se evalúa el contexto estático o expresión y el dinámico. La ejecución se produce llamando al método *executeQuery*, y se evalúa teniendo en cuenta el contexto estático en vigor.
- **XQPreparedExpression**: objeto creado a partir de una conexión para la ejecución de una expresión múltiples veces, retornando un *XQResultSetSequence* con los datos obtenidos, en este caso la evaluación del contexto estático solo se hace una vez, mientras que el proceso del contexto dinámico se repite. Igual que en **XQExpression** la ejecución se produce llamando al método *executeQuery*, y se evalúa teniendo en cuenta el contexto estático en vigor.
- **XQDynamicContext**: representa el contexto dinámico de una expresión, como puede ser la zona horaria y las variables que se van a utilizar en la expresión.

- **XQStaticContext:** representa el contexto estático para la ejecución de expresiones en esa conexión. Se puede obtener el contexto estático por defecto a través de la conexión. Si se efectúan cambios en el contexto estático no afecta a las expresiones ejecutándose en ese momento, solo en las creadas con posterioridad a la modificación. También es posible especificar un contexto estático para una expresión en concreto, de modo que ignore el contexto de la conexión.
- **XQItem:** representación de un elemento en *XQuery*. Es inmutable y una vez creado su estado interno no cambia.
- **XQResultItem:** objeto que representa un elemento de un resultado, inmutable, válido hasta que se llama al método close suyo o de la *XQResultSequence* a la que pertenece.

```
XQPreparedExpression consulta;
XQResultSequence resultado;
consulta = conn.prepareExpression("/EMPLEADOS/EMP_ROW[DEPT_NO=10]");
resultado = consulta.executeQuery();
XQResultItem r_item;
while(resultado.next()){
 r_item = (XQResultItem) resultado.getItem();
 System.out.println("Elemento: " + r_item.getItemAsString(null));
}
```

- **XQSecuence:** representación de una secuencia del modelo de datos *XQuery*, contiene un conjunto de 0 o más *XQItem*. Es un objeto recorrible.
- **XQResultSecuence:** resultado de la ejecución de una sentencia; contiene un conjunto de 0 o más *XQResultItem*. Es un objeto recorrible.

Este ejemplo obtiene los empleados del departamento 10, utilizamos *getItemAsString* para que devuelva los elementos como cadenas:

```
XQPreparedExpression consulta;
XQResultSequence resultado;
consulta = conn.prepareExpression("/EMPLEADOS/EMP_ROW[DEPT_NO=10]");
resultado = consulta.executeQuery();
while(resultado.next())
 System.out.println("Elemento: "
 +resultado.getItemAsString(null));
```

Con XQJ no se necesita seleccionar la colección de los documentos XML, la búsqueda la realiza en todas las colecciones. Tampoco admite el uso de sentencias de actualización de eXist, con lo cual las inserciones, borrados y actualizaciones resultan bastante engorrosas.

Partimos de que se ha subido la colección ***BDProductosXML***, y la consulta la haremos en el contexto **/db/*BDProductosXML***. Recuerda que con XQJ NO nos conectamos a la colección, nos conectamos a la base de datos, con lo cual a la hora de hacer las consultas indicaremos la colección o el documento de la colección.

## EJEMPLOS:

- Este método realiza una consulta al documento *productos.xml* de la colección ***BDProductosXML***, y devuelve los nodos.
- Si escribimos esta consulta XQJ busca los productos en esa colección:

```
for $pr in collection('/db/BDProductosXML')/productos/produc
```

```

 return $pr
}

Si escribimos esta otra consulta XQJ busca los productos en toda la base de datos y todas las colecciones. Pueden existir varios documentos con nodos productos/produc.
for $pr in /productos/produc return $pr

public static void verproductos() {
 try {
 XQDataSource server = new ExistXQDataSource();
 server.setProperty("serverName", "localhost");
 server.setProperty("port", "8083");
 server.setProperty("user", "admin");
 server.setProperty("password", "admin");
 XQConnection conn = server.getConnection();
 XQPreparedExpression consulta;
 XQResultSequence resultado;
 System.out.println("-- Consulta documento productos.xml --");
 consulta = conn.prepareExpression("for $pr in
collection('/db/BDProductosXML')/productos/produc return $pr");
 resultado = consulta.executeQuery();
 while (resultado.next()) {
 System.out.println("Elemento " +
 resultado.getItemAsString(null));
 }
 conn.close();
 } catch (XQException ex) {
 System.out.println("Error al operar.");
 }
}

```

- Estas instrucciones devuelven el número de productos con precio mayor de 50.

```

XQPreparedExpression consulta = conn.prepareExpression(
 " count(collection('/db/BDProductosXML')
 /productos/produc[precio>50]) ");
XQResultSequence resultado = consulta.executeQuery();
resultado.next();
System.out.println("Número de productos con precio > de 50: "
 + resultado.getInt());
conn.close();

```

- El siguiente método ejecuta una consulta almacenada en un fichero. El fichero está en la carpeta del proyecto, y se llama *miconsulta.xq*.

```

public static void ejecutarconsultadefichero() {
 try {
 XQDataSource server = new ExistXQDataSource();
 server.setProperty("serverName", "localhost");
 server.setProperty("port", "8083");
 server.setProperty("user", "admin");
 server.setProperty("password", "admin");
 XQConnection conn = server.getConnection();
 InputStream query;
 query = new FileInputStream("miconsulta.xq");
 XQExpression xqe = conn.createExpression();
 XQSequence resultado = xqe.executeQuery(query);
 }
}

```

```

 while (resultado.next()) {
 System.out.println(resultado.getItemAsString(null));
 }
 conn.close();
 } catch (XQException ex) {
 System.out.println("Error en las propiedades del server.");
 } catch (FileNotFoundException ex) {
 System.out.println("Error fichero.");
 }
}

```

### ACTIVIDAD 5.9.

Crea los siguientes métodos utilizando la API XQJ:

- Realiza una consulta al documento *zonas.xml* para que se visualice el nombre de zona y el número de productos de cada zona. Utiliza los documentos de la colección *BDProductosXML*.
- Utilizando el documento *universidad.xml*, visualiza los datos de los empleados de los departamentos de tipo A.

- El siguiente método crea un archivo XML en el disco con nombre *NUEVO\_EMPL10.xml*, a partir de los datos extraídos de una consulta. La consulta devuelve los empleados del departamento 10, más el título del documento *empleados.xml*, de la colección *ColeccionPruebas*. La consulta es la siguiente:

```

let $titulo:=
 doc('/db/ColeccionPruebas/empleados.xml')/EMPLEADOS/TITULO
return <EMPLEADOS>{$titulo}
 {for $em in doc('/db/ColeccionPruebas/empleados.xml')
 /EMPLEADOS/EMP_ROW[DEPT_NO=10]
 return $em}
</EMPLEADOS>

```

El ejercicio nos queda así:

```

public static void creaemple10() {
String nom = "NUEVO_EMPL10.xml";
File fichero = new File(nom);
XQDataSource server = new ExistXQDataSource();
try {
 server.setProperty("serverName", "localhost");
 server.setProperty("port", "8083");
 XQConnection conn = server.getConnection();
 XQPreparedExpression consulta = conn.prepareExpression(
 "let $titulo:= doc('/db/ColeccionPruebas/empleados.xml')"
 + "/EMPLEADOS/TITULO "
 + " return <EMPLEADOS>{$titulo} "
 + " {for $em in doc('/db/ColeccionPruebas/empleados.xml')"
 + "/EMPLEADOS/EMP_ROW[DEPT_NO=10] return $em} "
 + " </EMPLEADOS> ");
 XQResultSequence result = consulta.executeQuery();
 if (fichero.exists()) { // borramos y creamos
 if (fichero.delete())

```

```
 System.out.println("Archivo borrado. Creo de nuevo.");
 else
 System.out.println("Error al borrar el archivo");
 }
try {
 BufferedWriter bw = new BufferedWriter(new FileWriter(nom));
 bw.write("<?xml version='1.0' encoding='ISO-8859-1'?>" + "\n");
 result.next();
 String cad = result.getItemAsString(null);
 System.out.println("Salida: " + cad); // visualizamos
 bw.write(cad + "\n"); // grabamos en el fichero
 bw.close(); // Cerramos el fichero el fichero
 System.out.println("Fichero Creado");
} catch (IOException ioe) {ioe.printStackTrace();
}
conn.close();
} catch (XQException e) {e.printStackTrace();}
}
```

---

#### ACTIVIDAD 5.10.

A partir del los documentos *productos.xml* y *zonas.xml* de la colección *BDProductosXML*. Realiza un método que realice lo siguiente:

Crea un documento externo con nombre *zonas20.xml* que contenga los productos de la zona 20 y las siguientes etiquetas para cada producto: <cod\_prod>, <denominacion>, <precio>, <nombre\_zona>, <director> y <stock>, este stock debe ser el cálculo del stock\_actual - stock\_minimo.

---

#### 5.5.4. Tratamiento de excepciones

En este apartado se prueban las excepciones XMLDBEXCEPTION y XQEXCEPTION, para ver cuando se producen. A lo largo de la unidad se han realizado ejemplos capturando las excepciones o bien a nivel de los métodos utilizando la palabra clave **throws**, para luego ser tratadas en main(), o bien utilizando bloques **try-catch** dentro de los métodos. En este apartado se gestionan las excepciones utilizando bloques **try-catch**.

##### XMLDBException

**XMLDBException** se lanza cuando se produce un error en la API XML:DB. Contiene dos códigos de error, uno es el código de error XML de la API, y el otro es definido por el proveedor específico. El error del proveedor vendrá definido por *ErrorCodes.VENDO\_ERROR*. **XMLDBException** hereda de *java.lang.Exception*.

Cuando se produce un error con **XMLDBException** podemos acceder a cierta información usando el método *getMessage()* que devuelve una cadena que describe el error. En el siguiente ejemplo podemos ver cómo capturar los posibles errores que nos pueden ocurrir:

```
import org.xmldb.api.*;
import org.xmldb.api.base.*;
import org.xmldb.api.modules.*;

public class pruebaexcepciones {
```

```

protected static String driver = "org.exist.xmldb.DatabaseImpl";
public static String URI =
 "xmlDb:exist://localhost:8080/exist/xmlrpc";
private static Database database;
private static String usu="admin";
private static String pwd="admin";
private static Class cl=null;
private static XPathQueryService service;
private static ResourceSet result=null;
private static Collection col = null; // Colección
public static void main(String[] args) {
 try {
 cl = Class.forName(driver);
 } catch (ClassNotFoundException e) {
 System.out.println("No se encuentra la clase del driver: "
 + e.getMessage());
 } try {
 database = (Database) cl.newInstance();
 DatabaseManager.registerDatabase(database);
 } catch (InstantiationException e) {
 System.out.println("Error instanciando el driver. ");
 } catch(NullPointerException e) {
 System.out.println("Error al instanciar la clase del driver: "
 + e.getMessage());
 } catch (IllegalAccessException e) {
 System.out.println("Se ha producido una IllegalAccessException");
 } catch (XMLDBException e) {
 System.out.println("Error XMLDB :" + e.getMessage());
 } try {
 col = DatabaseManager.getCollection(URI+"/db/Pruebas",usu,pwd);
 } catch (XMLDBException e) {
 System.out.println("ERROR XMLDBException en getCollection."+
 e.getMessage());
 } try {
 service =(XPathQueryService)
 col.getService("XPathQueryService", "1.0");
 }catch (NullPointerException n){
 System.out.println("Error en getService, no se puede "+
 "crear el servicio.");
 }catch (XMLDBException e) {
 System.out.println("ERROR XMLDBException, en get service."
 + e.getMessage());
 } //Consulta a la BD
 try { result = service.query("for $b in
 /EMPLEADOS/EMP_ROW[APELLIDO='TOVAR'] return $b");
 }catch (NullPointerException n){
 System.out.println("Error en query, no se ha inicializado"+
 " la BD o el servicio.");
 }catch (XMLDBException e) {
 System.out.println("Error XMLDBException en la query: "+
 e.getMessage());
 } try {
 ResourceIterator i;
 i = result.getIterator();
 if (!i.hasMoreResources()){

```

```

 System.out.println("LA CONSULTA NO DEVUELVE NADA");
 }
 while (i.hasMoreResources()) { //Procesamos el resultado
 Resource r = i.nextResource();
 System.out.println((String) r.getContent());
 }
} catch (NullPointerException n){
 System.out.println("Error getIterator. Problemas con el "+
 " servicio.");
} catch (XMLDBException e) {
 System.out.println("Error XMLDBException, getIterator. : "+
 e.getMessage());
}
try {
 col.close();
} catch (NullPointerException n){
 System.out.println("Error en el cierre de la colección.");
} catch (XMLDBException e) {
 System.out.println("Error XMLDBException, col.close. : "+
 e.getMessage());
}
}//fin main
}// fin de la clase pruebaexcepciones

```

- El error **NullPointerException** es el error que se dispara en todos los casos y siempre que no se haya inicializado la BD, es el caso de escribir mal driver.
- Si escribimos mal la URI se disparará **XMLDBException** a la hora de asignar la colección en *getCollection*, y a partir de ahí se disparan todos los *NullPointerException*, pues el servicio para esa colección no se ha podido crear.
- Si escribimos mal la carpeta donde se encuentra la colección, se disparan los *NullPointerException* siguientes, ya que el servicio no se ha podido crear.
- Si escribimos mal la query se dispara el error **XMLDBException** a la hora de crear el *service.query*, en el mensaje visualiza en qué línea de la consulta está el error. Por ejemplo, si yo escribo esta consulta:

```
res = service.query(
 "forr $b in /EMPLEADOS/EMP_ROW[APELLIDO='TOVAR'] return $b");
```

Visualiza este mensaje:

```
Error XMLDBException en la query: Failed to invoke method queryP in
class org.exist.xmlrpc.RpcConnection:
org.exist.xquery.StaticXQueryException: exerr:ERROR
org.exist.xquery.XPathException: err:XPST0003 unexpected token: $ [at
line 1, column 6]
```

Para saber más sobre esta excepción acceder al sitio:

<http://xmldb-org.sourceforge.net/xapi/api/org/xmldb/api/base/XMLDBException.html>

---

### ACTIVIDAD 5.11.

Carga esta clase en tu ordenador y prueba las distintas situaciones de error.

---

## XQException

Cuando se produce un error con **XQException** disponemos de dos métodos que nos van a permitir saber la causa del error y poderlo arreglar. Estos métodos son *getMessage()* que devuelve una cadena que describe el error y *getCause()* que nos indica la causa del error para proceder a su solución.

Para saber más sobre esta excepción acceder al sitio:

<http://xqj.net/javadoc/javax/xml/xquery/XQException.html>

En el siguiente ejemplo vemos cómo capturar los posibles errores a la hora, sobre todo, de realizar consultas:

```
import javax.xml.xquery.XQConnection;
import javax.xml.xquery.XQDataSource;
import javax.xml.xquery.XQException;
import javax.xml.xquery.XQPreparedExpression;
import javax.xml.xquery.XQResultSequence;
import net.xqj.exist.ExistXQDataSource;

public class pruebaexcepcionesxqj{
 public static void main(String[] args)
 {
 XQConnection conn = null;
 XQDataSource server = new ExistXQDataSource();
 XQPreparedExpression consulta=null;
 XQResultSequence rs=null,resultado=null;
 try {
 server.setProperty("serverName", "localhost");
 server.setProperty("port", "8083");
 }catch (XQException e) {
 System.out.println("Error en Server. Mensaje:"+e.getMessage());
 System.out.println("Error en Server. Causa: "+e.getCause());
 try {
 conn = server.getConnection();
 }catch (XQException e) {
 System.out.println("Error en la conexión : "+ e.getMessage());
 }

 System.out.println("----Ejemplo Consulta Productos -----");
 try {
 consulta =conn.prepareExpression("for $de in " +
 "/productos/produc return $de");
 }catch (XQException e) {
 System.out.println("Error en la expresión. Mensaje : "+
 e.getMessage());
 System.out.println("Error en la expresión. Causa : "+
 e.getCause());
 try {
 resultado = consulta.executeQuery();
 while(resultado.next()){
 System.out.println("Elemento "+
 resultado.getItemAsString(null));
 }
 }catch (XQException e) {
 System.out.println("Error en la ejecución. Mensaje: "+
 e.getMessage());
 System.out.println("Error en la ejecución. Causa: "+
 e.getCause());
 }
 }
}
```

```

 e.getcause());
 try { conn.close();
 } catch (XQException e) {
 System.out.println("Error al cerrar la conexión.");
 }
}
}

```

### ACTIVIDAD 5.12.

Carga esta clase en tu ordenador y provoca situaciones de error, escribiendo mal la consulta, o el nombre del servidor o cerrando la BD.

## 5.6. BASE DE DATOS MONGODB

MongoDB es un sistema de base de datos multiplataforma orientado a documentos, se podrá almacenar cualquier tipo de contenido sin obedecer a un modelo o esquema. Está escrito en C++, con lo que es bastante rápido a la hora de ejecutar sus tareas. Además, está licenciado como GNU AGPL 3.0, de modo que se trata de un software de licencia libre. Funciona en sistemas operativos Windows, Linux, OS X y Solaris.

Una de sus características principales es la velocidad y la sencilla forma que tiene para hacer consultas a los contenidos. MongoDB se utiliza para cualquier aplicación que necesite almacenar datos semiestructurados, caso de aplicaciones CMS, aplicaciones móviles, de juegos, o plataformas e-commerce. MongoDB no soporta JOINS ni transacciones, aunque posee índices secundarios, un propio lenguaje de consulta muy expresivo, operaciones atómicas en un solo documento (pero no soporta transacciones de múltiples documentos), y lecturas consistentes.

La mayor diferencia entre las bases de datos relacionales y MongoDB es la forma en que se crea el modelo de datos, el modelo relacional es un modelo rígido y estructurado mientras que el modelo MongoDB es un modelo dinámico. En la siguiente tabla se muestra la terminología utilizada en el modelo relacional y el modelo de documento de MongoDB:

Modelo Relacional	MongoDB
Base de datos	Base de datos
Tabla	Colección
Fila	Documento
Columna	Campo
Índice	Índice
Join	Documento embebido o referencia

Con el modelo MongoDB se pasa de un modelo de datos rígido basado en estructuras de datos bidimensionales, formado por tablas, filas y columnas a un modelo de datos de documentos rico y dinámico con subdocumentos y matrices embebidas. En MongoDB se pueden crear colecciones sin definir su estructura, también se puede alterar la estructura de los documentos simplemente añadiendo nuevos campos o borrando los ya existentes. Esta característica convierte a Mongo en una BD muy flexible con respecto a las alternativas relacionales.

MongoDB almacena documentos JSON (JavaScript Object Notation) en una representación binaria llamada BSON (Binary JSON). BSON es una serialización codificada en binario de documentos JSON, soporta todas las características de JSON e incluye los tipos de datos int, long, float o arrays. El documento (el registro en el modelo relacional) representa la unidad básica de datos en MongoDB.

## 5.6.1 Estructuras JSON

JSON (JavaScript Object Notation - Notación de Objetos de JavaScript) es un formato ligero de intercambio de datos. Leerlo y escribirlo es simple para humanos, mientras que para las máquinas es simple interpretarlo y generararlo. Está basado en un subconjunto del Lenguaje de Programación JavaScript, *Standard ECMA-262 3rd Edition - Diciembre 1999*. JSON es un formato de texto que es completamente independiente del lenguaje, pero utiliza convenciones que son ampliamente conocidos por los programadores de la familia de lenguajes C, incluyendo C, C++, C#, Java, JavaScript, Perl, Python, y muchos otros. Estas propiedades hacen que JSON sea un lenguaje ideal para el intercambio de datos. (Fuente: <http://www.json.org/json-es.html>)

JSON está constituido por dos estructuras:

- **Una colección de pares de nombre/valor.** En varios lenguajes esto es conocido como un objeto, registro, estructura, diccionario, tabla hash, lista de claves o un array asociativo.
- **Una lista ordenada de valores.** En la mayoría de los lenguajes, esto se implementa como arrays, vectores, listas o secuencias.

Estas son estructuras universales; virtualmente todos los lenguajes de programación las soporan de una forma u otra. Es razonable que un formato de intercambio de datos que es independiente del lenguaje de programación se base en estas estructuras.

En JSON, se presentan de estas formas:

- Como un **objeto**, conjunto desordenado de **pares nombre/valor**. Un objeto comienza con { (llave de apertura) y termine con } (llave de cierre). Cada nombre es seguido por : (dos puntos) y los pares **nombre/valor** están separados por , (coma). En el ejemplo creo un objeto persona con nombre y oficio, y un objeto zona con su código y su nombre:

```
{
 "persona": { "nombre": "Alicia", "oficio": "Profesora" },
 "zona": { "codzona": 10, "nombre": "Madrid" }
```

- Un **array**, es decir, una colección de valores. Un array comienza con [ (corchete izquierdo) y termina con ] (corchete derecho). Los valores se separan por , (coma). En el ejemplo creo el objeto persona, un array de dos elementos, no tienen por qué tener los mismos pares **nombre/valor**, y el objeto zona con dos zonas:

```
{
 "persona": [
 { "nombre": "Alicia", "oficio": "Profesora", "ciudad": "Talavera" },
 { "nombre": "María Jesús", "oficio": "Profesora" }
]
}

{
 "zona": [
 { "codzona": 10, "nombre": "Madrid" },
 { "codzona": 15, "nombre": "Barcelona" }
]
}
```

- ```

        {"codzona": 20, "nombre": "Toledo", "tasa": 15}
    ]
}

• Un valor puede ser una cadena de caracteres con comillas dobles, o un número, o true o false o null, o un objeto o un array. Estas estructuras pueden anidarse. En el ejemplo se muestra un objeto de nombre ventana con distintos tipos de nombre/valor:

{
    "ventana": {
        "titulo": "Gestión Artículos",
        "alto": 300,
        "ancho": 500,
        "menu": null,
        "modal": true,
        "botones": ["ok", "cancel"]
    }
}

• Una cadena de caracteres es una colección de cero o más caracteres Unicode, encerrados entre comillas dobles, usando barras divisorias invertidas como escape. Un carácter está representado por una cadena de caracteres de un único carácter. Una cadena de caracteres es parecida a una cadena de caracteres C o Java.

• Un número es similar a un número C o Java, excepto que no se usan los formatos octales y hexadecimales.

```

Los espacios en blanco pueden insertarse entre cualquier par de símbolos *nombre/valor*.

Ejemplos XML vs JSON:

Para comprobar si un objeto JSON está bien escrito lo validaremos desde algún *validator*, por ejemplo, <http://jsonlint.com/>, o <https://jsonformatter.curiousconcept.com/>.

El documento *departamentos.xml*

```

<departamentos>
    <TITULO>DATOS DE LA TABLA DEPART</TITULO>
    <DEP_ROW>
        <DEPT_NO>10</DEPT_NO>
        <DNOMBRE>CONTABILIDAD</DNOMBRE>
        <LOC>SEVILLA</LOC>
    </DEP_ROW>
    <DEP_ROW>
        <DEPT_NO>20</DEPT_NO>
        <DNOMBRE>INVESTIGACION</DNOMBRE>
        <LOC>MADRID</LOC>
    </DEP_ROW>
    <DEP_ROW>
        <DEPT_NO>30</DEPT_NO>
        <DNOMBRE>VENTAS</DNOMBRE>
        <LOC>BARCELONA</LOC>
    </DEP_ROW>
    <DEP_ROW>
        <DEPT_NO>40</DEPT_NO>
        <DNOMBRE>PRODUCCION</DNOMBRE>
        <LOC>BILBAO</LOC>
    </DEP_ROW>
</departamentos>

```

Podemos representarlo en JSON así:

```
{
  "departamentos": [
    {
      "TITULO": "DATOS DE LA TABLA DEPART",
      "DEP_ROW": [
        {"DEPT_NO": 10, "DNOMBRE": "CONTABILIDAD", "LOC": "SEVILLA"},  

        {"DEPT_NO": 20, "DNOMBRE": "INVESTIGACIÓN", "LOC": "MADRID"},  

        {"DEPT_NO": 30, "DNOMBRE": "VENTAS", "LOC": "BARCELONA"},  

        {"DEPT_NO": 40, "DNOMBRE": "PRODUCCION", "LOC": "BILBAO"}
      ]
    }
}
```

O también, como DEP_ROW se repite, podemos representarlo como un array:

```
{
  "departamentos": [
    {
      "TITULO": "DATOS DE LA TABLA DEPART",
      "DEP_ROW": [
        {"DEPT_NO": 10, "DNOMBRE": "CONTABILIDAD", "LOC": "SEVILLA"},  

        {"DEPT_NO": 20, "DNOMBRE": "INVESTIGACIÓN", "LOC": "MADRID"},  

        {"DEPT_NO": 30, "DNOMBRE": "VENTAS", "LOC": "BARCELONA"},  

        {"DEPT_NO": 40, "DNOMBRE": "PRODUCCION", "LOC": "BILBAO"}
      ]
    }
}
```

En el siguiente documento (*sucursales.xml*) podemos representar *sucursal* y *cuenta* como arrays, ya que tenemos varias sucursales, y las sucursales tienen varias cuentas. Los atributos se representan como los demás elementos (pares nombre/valor).

```
<sucursales>  

  <sucursal telefono="112233" codigo="SUC1">  

    <director>Alicia Gómez</director>  

    <poblacion>Madrid</poblacion>  

    <cuenta tipo="AHORRO">  

      <nombre>Antonio García</nombre>  

      <numero>123456</numero>  

      <saldohaber>21000</saldohaber>  

      <saldodebe>200</saldodebe>  

    </cuenta>  

    <cuenta tipo="AHORRO">  

      <nombre>Pedro Gómez</nombre>  

      <numero>1111456</numero>  

      <saldohaber>12000</saldohaber>  

      <saldodebe>0</saldodebe>  

    </cuenta>  

  </sucursal>  

  <sucursal telefono="2023345" codigo="SUC2">  

    <director>Fernando Rato</director>  

    <poblacion>Talavera</poblacion>  

    <cuenta tipo="AHORRO">  

      <nombre>Marcelo Saez</nombre>  

      <numero>30303036</numero>  

      <saldohaber>15000</saldohaber>  

      <saldodebe>12000</saldodebe>  

    </cuenta>  

    <cuenta tipo="AHORRO">  

      <nombre>María Jesús Ramos</nombre>  

      <numero>4444222</numero>
  
```

```
<saldohaber>5000</saldohaber>
    <saldodebe>0</saldodebe>
  </cuenta>
</sucursal>
</sucursales >
```

Podemos representarlo en JSON así:

```
{
  "sucursales": {
    "sucursal": [
      {
        "telefono": 112233, "codigo": "SUC1",
        "director": "Alicia Gómez", "poblacion": "Madrid",
        "cuenta": [
          {
            "tipo": "AHORRO", "nombre": "Antonio García",
            "numero": 123456, "saldohaber": 21000,
            "saldodebe": 200
          },
          {
            "tipo": "AHORRO", "nombre": "Pedro Gómez",
            "numero": 1111456, "saldohaber": 12000,
            "saldodebe": 0
          }
        ]
      },
      {
        "telefono": 2023345, "codigo": "SUC2",
        "director": "Fernando Rato", "poblacion": "Talavera",
        "cuenta": [
          {
            "tipo": "AHORRO",
            "nombre": "Marcelo Saez", "numero": 30303036,
            "saldohaber": 150000, "saldodebe": 12000
          },
          {
            "tipo": "AHORRO", "nombre": "María Jesús Ramos",
            "numero": 4444222, "saldohaber": 5000,
            "saldodebe": 0
          }
        ]
      }
    ]
  }
}
```

ACTIVIDAD 5.13.

Convierte el documento *universidad.xml* de la colección *ColeccionPruebas* a objeto JSON.

5.6.2. Instalación MongoDB

En este apartado instalaremos la base de datos *NoSQL MongoDB*. Descargamos el archivo desde la página <https://www.mongodb.com/download>, para este libro se ha descargado la versión 3.2.6, (*mongodb-win32-x86_64-2008plus-ssl-3.2.6-signed.msi*) válida para Windows 32 y 64, que es la versión estable a fecha de hoy.

Para la instalación ejecutamos el archivo y seguimos el asistente (Figura 5.17), se acepta la licencia, se selecciona el tipo de instalación completa o custom (personalizada), se elige completa, y se pulsa el botón **Install**.

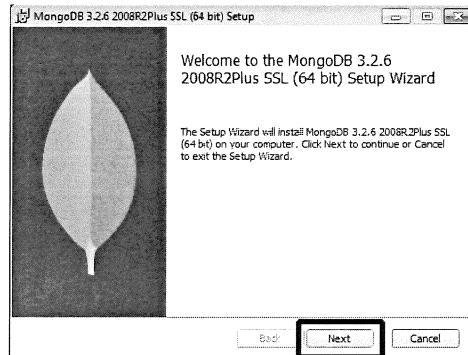


Figura 5.17. Instalación de MongoDB.

La base de datos se instala por defecto en (*Archivos de programa*) **C:\Program Files\MongoDB**. Para arrancar la base de datos buscaremos la carpeta bin de MongoDB (**C:\Program Files\MongoDB\Server\3.2\bin**), dentro de esa carpeta se encuentra el archivo **mongod.exe**, que es el que arranca la BD.

Importante: antes de iniciar la base de datos crearemos la carpeta **data** en la unidad donde se ha instalado la BD, por ejemplo **C:**, y dentro de data se crea **db** (**C:\data\db**), esa es la carpeta que por defecto MongoDB va a utilizar para almacenar la información.

Si se quiere *utilizar otro directorio*, por ejemplo, incluir la carpeta **data** dentro del propio Mongo DB (**C:\Program Files\MongoDB\data**), primero se crearán las carpetas, y luego **al arrancar la BD tenemos que indicar dónde se encuentra la carpeta data**. Por ejemplo, si yo me situó dentro de **bin**, para asignar la carpeta **data** escribiré lo siguiente desde la línea de comandos **mongod.exe --dbpath camino** (con ..\ voy un nivel hacia atrás):

```
C:\Program Files\MongoDB\Server\3.2\bin>mongod.exe --dbpath ..\..\..\data
```

Una vez arrancada la BD, se muestra una pantalla como la de la Figura 5.18, en la que se inicializa la BD, y se queda escuchando por el puerto **27017** las conexiones de los clientes.

Figura 5.18. Server MongoDB iniciado.

Si ahora queremos conectarnos como cliente a la BD, ejecutaremos el programa **mongo.exe** de la carpeta **bin**, también desde la línea de comando este programa inicia un cliente. Y desde el cliente podremos trabajar con la base de datos. Al conectarnos por defecto nos conecta a la BD test, véase la Figura 5.19.

```

Símbolo del sistema - mongo.exe
Microsoft Windows [Versión 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Reservados todos los derechos.

C:\Users\ARM>mongo.exe
'mongo.exe' no se reconoce como un comando interno o externo.
programa o archivo por lotes ejecutable.

C:\Users\ARM>cd C:\Program Files\MongoDB\Server\3.2\bin
C:\Program Files\MongoDB\Server\3.2\bin>mongo.exe
2016-05-28T01:56:05.111+0000 I CONTROL [main] Hotfix KB2731284 or later update is not installed, will zero-out data files
MongoDB shell version: 3.2.6
connecting to: test
>

```

Figura 5.19. Cliente MongoDB.

5.6.3. Operaciones básicas en MongoDB

Todos los comandos para operar con esta base de datos se escriben en minúscula, los más comunes son los siguientes:

- Listar las bases de datos: `show databases`
- Mostrar la base de datos actual: `db`
- Mostrar las colecciones de la base de datos actual: `show collections`
- Usar una base de datos (similar a MySQL): `use nombrebasedatos`, si no existe no importa, la creará en el momento que añadamos un objeto JSON, con las funciones `.save` o `.insert`.
- Si queremos saber el número de documentos dentro de las colecciones, utilizaremos la función `count`, escribiremos: `db.nombre_colección.count()`. También se utilizan las funciones `size()` y `length()`.
- Para añadir comentarios utilizamos los caracteres // de comentario de Java.

CREAR REGISTROS

Para añadir datos a la base de datos utilizaremos los comandos `.save` o `.insert` según este formato:

```

db.nombre_colección.save(dato JSON)
db.nombre_colección.insert(dato JSON)

```

Donde **db** es la base de datos actual, la que estemos usando (la abriremos con `use`) y nombre de colección es la colección donde se van a añadir los registros, si no existe se crea en ese momento.

Ejemplo: creo la base de datos **mibasedatos**, y dentro de ella la colección **amigos** con dos amigos:

```

use mibasedatos;
Amigo1={nombre:'Ana',teléfono:545656885,curso:'1DAM', nota:7};
Amigo2={nombre:'Marleni',teléfono:3446500,curso:'1DAM',nota:8};
db.amigos.save(Amigo1);
db.amigos.save(Amigo2);

```

Añado un amigo más, pero ahora utilizo la orden `insert`:

```

db.amigos.insert({nombre:'Juanito', teléfono:55667788,curso:'2DAM',
nota:6 });

```

IDENTIFICADOR DE OBJETOS, EL ObjectId (campo _id)

Los identificadores de cada documento (registro) son únicos. Se asignan automáticamente al crear el documento, se generan de forma rápida y ordenada. También se pueden crear de forma manual. Es un número hexadecimal que consta de 12 bytes, los primeros 4 son una marca de tiempo, los tres siguientes la identificación de la máquina, 2 bytes de identificador de proceso y un contador de 3 bytes empezando en un número aleatorio. El ***ObjectId*** o ***_id***, es como si fuese la clave del documento, no se repetirá en una colección. Si un documento no tiene ***_id*** MongoDB se lo asignará automáticamente, es lo que ocurre cuando insertamos y no indicamos el identificador.

5.6.4. Consultar registros

Para consultar datos de una colección utilizaremos la orden ***.find()***, escribiremos:

```
db.nombre_colección.find()
```

En el ejemplo si queremos ver la colección *amigos* escribiremos: ***db.amigos.find()***. Se muestran los identificativos (***_id***) de cada objeto JSON, únicos por colección, con el resto de campos. Véase la Figura 5.20.

Si se desea que la salida sea ascendente por uno de los campos, utilizamos el operador ***.sort***, por ejemplo, para obtener los datos de la colección ordenados por nombre escribimos:

```
db.amigos.find().sort({nombre:1});
```

El número que acompaña a la orden indica el tipo de ordenación, **1 ascendente** y **-1 descendente**. Por ejemplo, esto visualiza los datos ordenados descendentemente por nombre:

```
db.amigos.find().sort({nombre:-1});
```

Si se desean hacer búsquedas de documentos que cumplan una o varias condiciones, utilizamos el siguiente formato:

```
db.nombre_colección.find(filtro, campos)
```

- En ***filtro*** indicamos la condición de búsqueda, podemos añadir los pares *nombre:valor* a buscar. Si omitimos este parámetro devuelve todos los documentos, o pasa un documento vacío ({})
- En ***campos*** se especifican los campos a devolver de los documentos que coinciden con el filtro de la consulta. Para devolver todos los campos de los documentos omitimos este parámetro. Si se desean devolver uno o más campos escribiremos ***{nombre_campo1: 1, nombre_campo2: 1, ...}***. Si no se desean que se seleccionen los campos escribimos ***{nombre_campo1: 0, nombre_campo2: 0, ...}***. También podemos poner ***true*** o ***false*** en lugar de 1 o 0.

Por ejemplo, para buscar el amigo con nombre Marleni lo escribimos así:

```
db.amigos.find({nombre : "Marleni"})
```

Si solo deseo saber su teléfono escribo:

```
db.amigos.find({nombre : "Marleni"}, {teléfono:1})
```

Si deseo buscar el nombre y la nota de los alumnos de 1DAM escribiremos:

```
db.amigos.find({curso : "1DAM"}, {nombre:1, nota:1})
```

Si queremos saber el número de registros que devuelve una consulta pondremos `db.nombre_colección.find({filtros}).count()`, por ejemplo, para saber cuántos son del curso 1DAM escribiremos:

```
db.amigos.find({curso : "1DAM"}).count();
> use mibasedatos;
switched to db mibasedatos
> Amigo1={nombre: 'Ana',teléfono:545656885,curso:'1DAM', nota:7};
{ "nombre" : "Ana", "teléfono" : 545656885, "curso" : "1DAM", "nota" : 7 }
> Amigo2={nombre:'Marleni',teléfono:3446500,curso:'1DAM',nota:8};
{
    "nombre" : "Marleni",
    "teléfono" : 3446500,
    "curso" : "1DAM",
    "nota" : 8
}
> db.amigos.save(Amigo1);
WriteResult({ "inserted" : 1 })
> db.amigos.save(Amigo2);
WriteResult({ "inserted" : 1 })
> db.amigos.insert({nombre:'Juanito', teléfono:55667788,curso:'2DAM', nota:6 });
WriteResult({ "inserted" : 1 })
> db.amigos.find()
{ "_id" : ObjectId("5767678e7682441c00e99186"), "nombre" : "Ana", "teléfono" : 545656885, "curso" : "1DAM", "nota" : 7 }
{ "_id" : ObjectId("5767678e7682441c00e99187"), "nombre" : "Marleni", "teléfono" : 3446500, "curso" : "1DAM", "nota" : 8 }
{ "_id" : ObjectId("5767679c7682441c00e99188"), "nombre" : "Juanito", "teléfono" : 55667788, "curso" : "2DAM", "nota" : 6 }
```

Figura 5.20. Creación de *mibasedatos* y la colección *amigos*.

Se pueden hacer consultas más complejas añadiendo selectores de búsquedas.

SELECTORES DE BÚSQUEDAS DE COMPARACIÓN

- ***\$eq***, igual a un valor. Esta orden obtiene los documentos con nota = 6:

```
db.amigos.find({ nota : { $eq : 6 } })
```

- ***\$gt***, mayor que y ***\$gte*** mayor o igual que. Esta orden obtiene los documentos con nota >= 6:

```
db.amigos.find({ nota : { $gte : 6 } })
```

- ***\$lt***, menor que y ***\$lte***, menor o igual que. El ejemplo muestra los amigos de 1DAM con notas entre 7 y 9 incluidas, preguntamos por un intervalo >=7 y <=9:

```
db.amigos.find({curso : "1DAM", nota : { $gte : 7, $lte : 9 } })
```

- ***\$ne***, distinto a un valor. El siguiente ejemplo obtiene los documentos con nota distinta de 7:

```
db.amigos.find({ nota : { $ne : 7 } })
```

- ***\$in***, entre una lista de valores y ***\$nin***, no está entre la lista de valores. En el ejemplo se obtienen los documentos cuya nota sea uno de estos valores: 5,7 y 8:

```
db.amigos.find({ nota : { $in : [5, 7, 8] } })
```

Ahora añado el nombre y el curso:

```
db.amigos.find({ nota : { $in : [5, 7, 8] } }, {nombre:1, curso:1})
```

SELECTORES DE BÚSQUEDAS LÓGICOS

- ***\$or***. La siguiente orden obtiene los documentos de los cursos 1DAM , o los que tienen nota > de 7:

```
db.amigos.find({ $or : [ { nota: { $gt: 7 } }, {curso : "1DAM"} ] })
```

Esta consulta obtiene los amigos con nombre Ana o Marleni:

```
db.amigos.find({ $or: [ {nombre : "Ana"}, {nombre: "Marleni"} ] })
```

- **\$and.** Este operador se maneja implícitamente, no es necesario especificarlo. Las siguientes órdenes hacen lo mismo, obtienen los amigos del curso 2DAM y con nota 6:

```
db.amigos.find({ $and : [{curso : "2DAM"}, {nota : 6}] })
db.amigos.find({curso : "2DAM", nota : 6})
```

Esta otra consulta devuelve el documento con nombre Marleni y con teléfono 3446500:

```
db.amigos.find( {nombre : "Marleni", teléfono : 3446500} )
db.amigos.find( { $and : [{nombre:"Marleni"},{teléfono:3446500}] })
```

- **\$not.** Representa la negación, en el ejemplo obtengo los amigos con nota no mayor de 7.

```
db.amigos.find({ nota : { $not: { $gt: 7 } } })
```

Esta otra consulta visualizará el nombre, el curso y la nota de los que su nota no mayor de 7:

```
db.amigos.find({ nota : { $not: { $gt: 7 } } },{_id:0, nombre:1, curso:1, nota:1} )
```

- **\$exists,** este operador booleano permite filtrar la búsqueda tomando en cuenta la existencia del campo de la expresión. Este ejemplo obtiene los registros que tengan nota.

```
db.amigos.find( { nota : {$exists:true} })
```

5.6.5. Actualizar registros en MongoDB

Para actualizar datos utilizaremos el comando **.update**, según este formato de uso:

```
db.nombre_colección.update(
    filtro_búsqueda,
    cambios_a_realizar,
    {
        upsert: booleano,
        multi: booleano
    });
}
```

Donde:

En **filtro_búsqueda**, se indica la condición para localizar los registros o documentos a modificar.

En **cambios_a_realizar**, se especifican los cambios que se desean hacer. Hay que tener cuidado al utilizar esta orden, pues en **cambios_a_realizar** se indica cómo quedará el documento que se busca si este existe, es decir: el resultado final del documento es lo que se escriba en **cambios_a_realizar**.

Si no se escriben todos los campos que tenía el documento, estos no los incluye en la modificación, entonces, los elimina. Por ejemplo:

```
db.amigos.update({nombre:"Ana"},{nombre: "Ana María" } );
```

Esta orden cambia el documento con nombre *Ana* por nombre *Ana María*. El resto de campos como no aparecen en **cambios_a_realizar** los elimina.

Ahora cambio el teléfono de Marleni por 925666555, y solo mantengo el campo nombre y teléfono, el resto al no aparecer en **cambios_a_realizar** se eliminan:

```
db.amigos.update({nombre: 'Marleni'}, { nombre: 'Marleni' ,teléfono: 925666555 } );
```

Nos encontramos con dos tipos de cambios: cambiar el documento completo por otro que indiquemos, o modificar solo los campos especificados, para ello utilizamos los parámetros ***upsert*** y ***multi***, ambos son opcionales y su valor por defecto es ***false***:

- ***upsert*** – si asignamos ***true*** a este parámetro, se indica que si el filtro de búsqueda no encuentra ningún resultado, entonces, el cambio debe ser insertado como un nuevo registro.
- ***multi*** – en caso de que el filtro de búsqueda devuelva más de un resultado, si especificamos este parámetro a ***true***, el cambio se realizará a todos los resultados, de lo contrario solo se cambiará al primero que encuentre, es decir, al que tenga menor identificativo de objeto, ***_id***.

Ahora cambio el teléfono de Pepita por 12345, y utilizo ***upsert***. Como Pepita no existe lo va a añadir por indicar ***upsert: true***.

```
db.amigos.update({nombre: 'Pepita'}, { nombre: 'Pepita' ,teléfono: 12345 }, { upsert: true } );
```

El comando ***.update*** cuenta con una serie de operadores para realizar actualizaciones más complejas. Algunos de estos operadores son los siguientes:

OPERADORES DE MODIFICACIÓN

- ***\$set***, permite actualizar con nuevas propiedades a un documento (o conjunto de documentos). Por ejemplo, añado la edad 24 a Ana María y 34 a Marleni:


```
db.amigos.update({nombre:"Ana María"}, { $set: {edad:24} } )
db.amigos.update({nombre:"Marleni"}, { $set: {edad:34} } )
```

 Si el documento ya tiene ese campo, no lo añade, cambiaría el valor si es distinto.
- ***\$unset***, permite eliminar propiedades de un documento. Por ejemplo, borro la edad 34 de Marleni:


```
db.amigos.update({nombre:"Marleni"}, { $unset: {edad:34} } )
```
- ***\$inc***, incrementa en una cantidad numérica especificada en el valor del campo a incrementar. Por ejemplo sumo 1 a la edad de Ana María:


```
db.amigos.update({nombre:"Ana María"}, { $inc: {edad:1} } )
```
- ***\$rename***, renombra campos del documento. Por ejemplo, cambiamos los campos nombre y edad de Ana María y los ponemos en inglés:


```
db.amigos.update({nombre:'Ana María'}, { $rename:{edad:'age', nombre:'name'}})
```

Cargamos de nuevo los datos iniciales de amigos y probamos ahora las siguientes consultas. Subimos la nota 1 punto a los de 1DAM, podemos escribir estas órdenes:

```
db.amigos.update({curso:"1DAM"}, { $inc: {nota:1} } )
db.amigos.update({curso:"1DAM"}, { $inc: {nota:1} },{multi: true} )
```

La primera consulta solo sube la nota al primero que se encuentra, es decir, al primer *_id* que cumpla la condición. La segunda sube la nota a todos los del 1DAM por utilizar *{multi: true}*.

En este ejemplo añado población Talavera a todos los del curso 1DAM:

```
db.amigos.update({curso:"1DAM"}, { $set: {población: "Talavera"} }, {multi: true})
```

ACTIVIDAD 5.14.

Crea la colección *empleados* dentro de la base de datos *mibasedatos*, y añade los siguientes registros:

```
Emp_no:1,nombre:"Juan",dep:10, salario:1000, fechaalta:"10/10/1999"
Emp_no:2,nombre:"Alicia",dep:10, salario:1400, fechaalta:"07/08/2000",
oficio: "Profesora"
Emp_no:3,nombre:"María Jesús",dep:20, salario:1500, fechaalta:
"05/01/2005", oficio: "Analista", comisión:100
Emp_no:4,nombre:"Alberto",dep:20, salario:1100, fechaalta:"15/11/2001"
Emp_no:5,nombre:"Fernando",dep:30, salario:1400, fechaalta:
"20/11/1999", comisión:200, oficio: "Analista"
```

Realiza las siguientes consultas:

- Visualiza los empleados del departamento 10.
- Visualiza los empleados del departamento 10 y 20.
- Obtén los empleados con salario >1300 y oficio Profesora.
- Sube el salario a los analistas en 100€, a todos los analistas.
- Decrementa la comisión en 20€ (escribir -20), solo a los que tengan comisión.

OPERACIONES CON ARRAYS

En este apartado veamos cómo realizar consultas en documentos que contienen arrays. Creamos la colección libros con tres libros, y un array con temas del libro:

```
db.libros.insert({codigo:1,nombre:"Acceso a datos", pvp: 35,
editorial:"Garceta", temas:["Base de datos", "Hibernate", "Neodatis"]})

db.libros.insert({codigo:2,nombre:"Entornos de desarrollo", pvp: 27,
editorial:"Garceta", temas:["UML", "Subversión", "ERMaster"]})

db.libros.insert({codigo:3,nombre:"Programación de Servicios", pvp: 25,
editorial:"Garceta", temas:["SOCKET", "Multihilo"]})
```

Para consultar los elementos del array escribimos el array y el elemento a consultar. Ejemplos:

Libros que tengan el tema UML: db.libros.find({temas: "UML"})

Libros que tengan el tema UML o Neodatis:

```
db.libros.find( { $or: [ {temas: "UML"}, {temas: "Neodatis"}] } )
```

Libros de la editorial Garceta, con pvp > 25 y que tengan el tema UML o Neodatis:

```
db.libros.find( { editorial:"Garceta", pvp: { $gt:25} , $or: [
{temas:"UML"} , {temas: "Neodatis"}] } )
```

OPERACIONES DE MODIFICACIÓN PARA ARRAYS

- **\$push**, añade un elemento a un array. Este ejemplo añade el tema *MongoDB* al libro con código 1:

```
db.libros.update( { codigo:1 }, { $push : {temas: "MongoDB" } } )
```
- **\$addToSet**, agrega elementos a un array solo si estos no existen. En el ejemplo se añade el tema *Base de datos* a todos los libros que no lo tengan. Primero preguntamos si el libro tiene el campo *temas*. Para que se añada a todos los libros indicamos **multi:true**:

```
db.libros.update({ temas : { $exists:true} }, { $addToSet: {temas: "Base de datos" } }, {multi:true})
```
- **\$each**, se usa en conjunto con **\$addToSet** o **\$push** para indicar que se añaden varios elementos al array.

```
db.libros.update({codigo:1}, {$push:{temas:{$each: ["JSON", "XML"]}}})
```



```
db.libros.update({codigo:2}, { $addToSet :{temas: { $each: ["Eclipse", "Developer"] }}})
```
- **\$pop**, elimina el primer o último valor de un array. Con valor -1 borra el primero, con otro valor el último. En el ejemplo se borra el primer tema del libro con código 3:

```
db.libros.update({codigo:3}, {$pop: { temas:-1 } })
```
- **\$pull**, elimina los valores de un array que cumplen con el filtro indicado. En el ejemplo se borran de todos los libros los elementos "*Base de datos*" y "*JSON*", si los tienen:

```
db.libros.update( {}, { $pull:{ temas: { $in: ["Base de datos", "JSON"] } } }, { multi: true } )
```

ACTIVIDAD 5.15.

Utilizando la colección libros realiza las siguientes consultas:

- Visualiza los libros de la editorial Garceta, con pvp entre 20 y 25 incluidos y que tengan el tema SOCKET.
 - Agrega el tema SOCKET a los libros que no lo tengan.
 - Baja a 5 el precio de los libros de la editorial Garceta.
-

5.6.6. Borrar registros

Para borrar datos JSON podemos utilizar las órdenes **.remove** y **.drop**. Se puede eliminar los documentos que cumplen una condición, o todos los documentos de la colección o la colección completa.

- Para borrar un documento que cumpla una condición utilizaremos la orden **remove({ nombre: valor })**. Por ejemplo, se borra a Marleni:

```
db.amigos.remove({nombre : "Marleni"});
```

- Si se desea borrar al elemento con nombre Ana y teléfono 545656885 escribimos:

```
db.amigos.remove({nombre : "Ana", telefono : 545656885});
```

- Para borrar todos los elementos de la colección ponemos: `db.amigos.remove({})`;

- Para borrar la colección escribiremos: db.amigos.drop();

5.6.7. Funciones de agregado

Al igual que en otra base de datos MongoDB dispone de funciones matemáticas y de cadenas para utilizarlas en las consultas. Algunas de ellas son las siguientes:

FUNCIONES ARITMÉTICAS:

| Función | Descripción |
|-------------------|--|
| \$abs | Devuelve el valor absoluto de un número |
| \$add | Añade números a una cantidad o a una fecha, en este caso suma milisegundos |
| \$ceil | Devuelve el entero menor, mayor o igual que el número especificado |
| \$divide | Devuelve el resultado de dividir el primer número por el segundo. Tiene 2 argumentos |
| \$floor | Devuelve el entero mayor, menor o igual que el número especificado |
| \$mod | Devuelve el resto de dividir el primer número por el segundo. Tiene 2 argumentos |
| \$multiply | Multiplica varios números, acepta varios argumentos |
| \$pow | Eleva un número a la potencia especificada |
| \$sqrt | Calcula la raíz cuadrada |
| \$subtract | Devuelve el resultado de restar el primer número menos el segundo. Si los dos valores son números, devuelve la diferencia. Si los dos valores son fechas, devuelve la diferencia en milisegundos |
| \$trunc | Trunca un número |

FUNCIONES DE CADENAS:

| Función | Descripción |
|---------------------|---|
| \$concat | Concatena varias cadenas, las que se pongan en la expresión |
| \$substr | Devuelve una subcadena de una cadena, a partir de una posición indicada hasta una longitud especificada. Lleva 3 argumentos, la cadena, la posición de inicio y la longitud |
| \$toLower | Convierte una cadena a minúsculas |
| \$toUpper | Convierte una cadena a mayúsculas |
| \$strcasecmp | Compara cadenas y devuelve 0 si las dos cadenas son equivalentes, 1 si la primera cadena es mayor que la segunda, y -1 si la primera cadena es menor que la segunda |

FUNCIONES DE GRUPO:

| Función | Descripción |
|----------------|---|
| \$sum | Devuelve la suma de valores numéricos. Ignora los valores no numéricos |
| \$avg | Devuelve la media de valores numéricos. Ignora los valores no numéricos |
| \$first | Devuelve el primer valor del grupo |
| \$last | Devuelve el último valor |
| \$max | Devuelve el valor máximo de un grupo o de un array |
| \$min | Devuelve el valor mínimo de un grupo o de un array |

FUNCIONES DE FECHA:

| Función | Descripción |
|------------------------------------|--|
| <code>\$dayOfYear</code> | Devuelve el día del año. Un número entre 1 y 366 |
| <code>\$dayOfMonth</code> | Devuelve el día del mes. Un número entre 1 y 31 |
| <code>\$dayOfWeek</code> | Devuelve el día de la semana. Un número entre 1 (Domingo) y 7 (Sábado) |
| <code>\$year</code> | Devuelve el año, formato yyyy, por ejemplo, 2016 |
| <code>\$month</code> | Devuelve el número de mes entre 1 (Enero) y 12 (Diciembre) |
| <code>\$hour</code> | Devuelve la hora entre 0 y 23 |
| <code>\$minute</code> | Devuelve los minutos entre 0 y 59 |
| <code>\$second</code> | Devuelve los segundos entre 0 y 6 |
| <code>\$dateToString</code> | Devuelve la fecha en formato String |

USO DE ESTAS FUNCIONES:

Estas funciones se utilizan en las *operaciones de agregación*, o *consultas de agregación*, que lo que hacen es procesar los registros y obtener nuevos resultados, calculados o transformados.

La agregación opera con grupos de valores de múltiples documentos y se puede realizar una variedad de operaciones sobre los datos agrupados para devolver un solo resultado. El objetivo es presentar datos calculados, formateados y/o filtrados de manera diferente a como se encuentran en los documentos.

MongoDB ofrece tres formas de realizar la agregación: *la agregación pipeline*, la función de *map-reduce* y la *agregación de propósito único*. En este tema estudiaremos la agregación más común para hacer consultas complejas, la *pipeline*.

Consulta esta URL para saber más sobre la agregación:
<https://docs.mongodb.com/manual/aggregation/>

5.6.8. La agregación pipeline

La *agregación pipeline* o *tuberías de agregación* se basa en someter una colección a un conjunto de *operaciones* o *etapas*, estas etapas irán convirtiendo y transformando el conjunto de documentos pertenecientes a la colección, hasta obtener un conjunto de documentos con el resultado deseado.

Se le llama tubería ya que cada etapa irá modificando, moldeando y calculando la estructura de los documentos para pasarlo a la etapa que le sigue. Las etapas son las siguientes:

| Etapa | Descripción | Multiplicidad |
|-------------------------------|---|---------------|
| <code>\$project</code> | Cambia la forma del documento. La <i>proyección</i> permite modificar la representación de los datos, por lo que en general se emplea para darles una nueva forma con la que resulte más cómodo trabajar. | 1:1 |

| Etapa | Descripción | Multiplicidad |
|-----------------|---|---------------|
| \$match | Filtrar los resultados. La etapa match permite <i>filtrar</i> los documentos para que en el resultado de la etapa solo estén aquellos que cumplen ciertos criterios. Se puede filtrar antes o después de agregar los resultados, en función del orden en que definamos esta etapa | n:1 |
| \$group | Agrupación. Permite <i>agrupar</i> distintos documentos según comparten el valor de uno o varios de sus atributos, y realizar operaciones de agregación sobre los elementos de cada uno de los grupos. Se utilizan las funciones <i>sum</i> , <i>max</i> , <i>min</i> , <i>avg</i> , etc. | n:1 |
| \$sort | Ordenación de documentos | 1:1 |
| \$skip | Salta N elementos | n:1 |
| \$limit | Elige N elementos para el resultado | n:1 |
| \$unwind | Normaliza arrays | 1:n |
| \$out | Envía el resultado a una salida, se almacena en la BD como una nueva colección | 1:1 |

La multiplicidad se refiere a cuántos documentos obtenemos como resultado después de aplicar la etapa, por ejemplo, **1:1** se aplica a 1 documento y se obtiene 1. **n:1** se aplica a n documentos y se obtiene n. **1:n** se aplica a un documento y se obtienen n.

Formato para utilizar las etapas:

```
db.mi_coleccion.aggregate([
  {
    $etapa1: {
      ....
    }
  , {
    $etapa2: {
      ....
    }
  , ....
])
```

EJEMPLOS:

Para los ejemplos creamos la colección **artículos**, que se encuentra en el archivo de colecciones *MongoDB*, de la carpeta de recursos de la unidad. Cada documento artículo está formado por los campos: *código*, *denominación*, *pvp*, *categoría*, *uv*, y *stock*.

- Obtener las denominaciones de los artículos y la categoría convertida a mayúsculas. Se utiliza la etapa **\$project** pues cambiamos el aspecto del documento. **Para referirnos a los campos del documento los ponemos entre comillas y con el prefijo \$**:

```
db.articulos.aggregate(
  [
    {
      $project:
      {
        denominación: { $toUpperCase: "$denominación" },
        ...
      }
    }
])
```

```

        categoría: { $toUpperCase: "$categoría" }
    }
}
])
```

Si esta salida se desea almacenar en la base de datos, añadimos la etapa ***out***. Por ejemplo:

```

db.articulos.aggregate(
[ {
    $project:
    {
        denominación: { $toUpperCase: "$denominación" },
        categoría: { $toUpperCase: "$categoría" }
    }
},
{
    $out: "salidanueva"
}
])
```

Obtener la denominación en mayúsculas, el importe de las ventas, que serán las *uv* * el *pvp*, y el stock actual que será *stock* menos *uv*. Se utiliza la etapa ***\$project***, la función ***\$multiply*** para multiplicar las *uv* por el *pvp* y ***\$subtract*** para restar las *uv* del *stock*.

•

```

db.articulos.aggregate(
[ {
    $project:
    {
        artículo: { $toUpperCase: "$denominación" },
        importe: { $multiply: ["$pvp", " $uv"] },
        stockactual: { $subtract: ["$stock", " $uv"] }
    }
}
])
```

Condiciones de agregación:

Podemos añadir las siguientes condiciones a las consultas de agregación:

| Name | Descripción |
|------------------------|---|
| <i>\$cond</i> | Este operador evalúa una expresión y dependiendo del resultado, devuelve el valor de una de las otras dos expresiones. Recibe tres expresiones en una lista ordenada o tres parámetros con nombre. Formato:
<code>{ \$cond: [<boolean-expression>, <caso-true>, <caso-false>] }</code> |
| <i>\$ifNull</i> | Devuelve o bien el resultado no nulo de la primera expresión o el resultado de la segunda expresión si la primera expresión da como resultado un resultado nulo. Acepta dos expresiones como argumentos. El resultado de la segunda expresión puede ser nulo.
<code>{ \$ifNull: [<expression>, <expresionsiesnull>] }</code> |

Ejemplos:

- A la consulta anterior, vamos a preguntar si el stock actual es negativo, asignaremos a un campo nuevo llamado ***areponer*** true si es menor que 0 y false si no lo es. La condición que se añade es:

```
{ $cond: [ { $lte: [ { $subtract: ["$stock", " $uv"] } , 0 ] },
```

```
        true , false ]
}
```

La consulta completa queda así:

```
db.articulos.aggregate( [ {
    $project: {
        artículo: { $toUpper: "$denominación" },
        importe: { $multiply: ["$pvp", "$uv"] },
        stockactual: { $subtract: ["$stock", "$uv"] },
        areponer: {
            $cond: [{ $lte: [{ $subtract: ["$stock", "$uv"] } , 0]}, {
                true, false]
            }
        }
    } ] )
```

- En la siguiente consulta obtenemos por cada categoría el número de artículos, el total unidades vendidas de artículos, y el total importe, la suma de los pvp*unidades. Es como una select con *group by*. En este caso se utiliza la etapa **\$group**, cuando se utiliza esta etapa se debe añadir el identificador de objeto *_id*, en este caso como agrupamos por categoría lo indicamos en el *_id*. Que a su vez será el identificador del resultado. Para contar artículos se utiliza la función **\$sum**, sumando 1:

```
db.articulos.aggregate( [ {
    $group: {
        _id: "$categoría",
        contador: { $sum: 1 },
        sumaunidades: { $sum: "$uv" },
        totalimporte: { $sum: { $multiply: ["$pvp", "$uv"] } }
    }
} ] )
```

- En la siguiente consulta obtenemos el número de documentos de la categoría **Deportes**, el total de unidades vendidas de sus artículos, el total importe y la media de unidades vendidas. Se utilizan las etapas **\$match** para seleccionar la categoría, y luego **\$group** para obtener resultados agrupados, en este caso en *_id* ponemos cualquier valor:

```
db.articulos.aggregate( [
    { $match: { categoría: "Deportes" } },
    { $group: {
        _id: "deportes",
        contador: { $sum: 1 },
        sumaunidades: { $sum: "$uv" },
        media: { $avg: "$uv" },
        totalimporte: { $sum: { $multiply: ["$pvp", "$uv"] } }
    } }
] )
```

- En la siguiente consulta obtenemos el precio más caro, se agrupan los registros y obtenemos el máximo del pvp:

```
db.articulos.aggregate( [
    { $group: {
        _id: null,
        maximo: { $max: "$pvp" } }
    }
] )
```

- En la siguiente consulta obtenemos el artículo con el precio más caro. Utilizamos dos consultas:

1-Primero obtenemos los datos pvp y denominación, de todos los artículos, ordenados descendente por precio y denominación. Para ello utilizamos la etapa **\$sort**

2-El resultado obtenido se agrupa con **\$group**, para luego obtener el primero con la función **\$first**:

```
db.articulos.aggregate(
  [
    { $sort: { pvp: -1, denominación: -1 } },
    { $group:
      { _id: null,
        mascaro: { $first: "$denominación" },
        precio: { $first: "$pvp" } }
    }
  ]
)
```

- En la siguiente consulta obtenemos la suma de importe de los artículos cuya denominación empieza por M o P. Para realizar esta consulta pasamos por 3 etapas:

1-Obtenemos de todos los artículos el primer carácter de la denominación utilizando la función **\$substr** y el importe de cada artículo:

```
{
  $project: {
    primercarac: { $substr: ["$denominación", 0, 1] } ,
    impor: { $multiply: ["$pvp", "$uv"] } }
}
```

2-En la siguiente etapa se seleccionan, de los datos obtenidos en la primera etapa (**primercarac**, **impor**), los que tienen en **primercarac** P o M:

```
{ $match: { "primercarac": { $in: ["M", "P"] } } }
```

3-Y finalmente se agrupa ese resultado, se añade un **_id: 1**, y se suman los importes.

```
{
  $group: { _id: 1,
            totalimporte: { $sum: "$impor" } }
}
```

La consulta completa quedará así:

```
db.articulos.aggregate([
  { $project: {
    primercarac: { $substr: ["$denominación", 0, 1] } ,
    impor: { $multiply: ["$pvp", "$uv"] } }
},
  { $match: { "primercarac": { $in: ["M", "P"] } } },
  {
    $group: { _id: 1,
              totalimporte: { $sum: "$impor" } }
  }
])
```

- En la siguiente consulta obtenemos por cada categoría el artículo con el precio más caro. Para ello primero ordenamos descendente por pcategoría, pvp y denominación, utilizando la etapa **\$sort**. Y el resultado obtenido se agrupa con **\$group** para luego obtener el primero de cada categoría con la función **\$first**:

```
db.articulos.aggregate(
```

```
[  
  { $sort: { categoría: -1, pvp: -1, denominación: -1 } },  
  { $group:  
    { _id: "$categoría",  
      mascaro: { $first: "$denominación" },  
      precio: { $first: "$pvp" } }  
  }  
]  
)
```

UTILIZACIÓN DE ARRAYS, CAMPOS COMPUESTOS Y AGREGADOS

Carga la colección *Trabajadores* que se encuentra en el archivo de colecciones de los recursos de la unidad. El formato de un trabajador es este:

```
db.trabajadores.insert( {  
  nombre: {nomb:"Alicia",ape1:"Ramos", ape2:"Martín"},  
  dirección: {población: "Madrid", calle : "Avda Toledo 10"},  
  salario: 1200,  
  oficios: ["Profesora", "Analista"],  
  primas: [20,30,40],  
  edad:50  
} )
```

Observa que el trabajador está formado por dos campos compuestos: *nombre* y *dirección*, y dos arrays: *oficios* y *primas*.

- La siguiente consulta de agregado devuelve la población, el nombre descompuesto en nombre, ape1 y ape2, el primer oficio del array oficios, el segundo oficio y el último. Si no los tiene no devuelve nada. Ordenados por población ascendente. Para acceder a los campos compuestos navegamos como si fuesen un objeto, *nombre.nom*, o *dirección.población*. Por ejemplo, esta consulta devuelve los que tienen la población Toledo:

```
db.trabajadores.find({ "dirección.población":"Toledo" })
```

Para acceder a los elementos de un array utilizamos la función **\$arrayElemAt: ["\$Nombre_del_array", posición_del_elemento_a_consultar]**. La posición es 0 para el primer elemento, y -1 para el último.

```
db.trabajadores.aggregate(  
[  
  { $sort: { "dirección.población": 1 } },  
  { $project:  
    {  
      población: "$dirección.población",  
      nombre: "$nombre.nomb" ,  
      ape1: "$nombre.apel" ,  
      ape2: "$nombre.ape2" ,  
      oficio1: { $arrayElemAt: [ "$oficios", 0 ] } ,  
      oficio2 : { $arrayElemAt: [ "$oficios", 1 ] } ,  
      oficioultimo: { $arrayElemAt: [ "$oficios", -1 ] }  
    }  
  }  
])
```

Otras funciones para arrays son:

| Nombre | Descripción |
|------------------------------------|---|
| <code>\$arrayElemAt</code> | Devuelve el elemento especificado en el índice |
| <code>\$concatArrays</code> | Devuelve un array concatenado en una cadena |
| <code>\$filter</code> | Selecciona elementos de un array y devuelve otro array con esos elementos |
| <code>\$isArray</code> | Determina si el operando es una array o no. Devuelve true o false. |
| <code>\$size</code> | Devuelve el número de elementos del array |
| <code>\$slice</code> | Devuelve un sub-set de elementos del array, se especifica el número. |

- La siguiente consulta de agregado devuelve los elementos que tienen los arrays de los trabajadores (oficios y primas), y los arrays concatenados. Se utiliza la función "`$$ifNull`" para comprobar que los arrays existan en los trabajadores, y evitar errores de salida ("*The argument to \$size must be an Array, but was of type: EOO*"). Se pregunta si el array es null, si lo es devuelve el array vacío, el tamaño del array vacío será 0.

```
db.trabajadores.aggregate( [
  { $project:
    {
      nombre: "$nombre.nomb",
      numerooficios: { $size: { $$ifNull: ["$oficios", []] } },
      numeroprimas: { $size: { $$ifNull: ["$primas", []] } },
      oficiosconcatenados: {$concatArrays: ["$oficios", "$primas"] }
    }
  }
])
```

- La siguiente consulta de agregado devuelve el número de trabajadores y la media de edad de los trabajadores que han tenido el oficio de *Analista*.

```
db.trabajadores.aggregate( [
  { $match: { oficios: "Analista" } },
  { $group:
    {
      _id: "analista",
      contador: { $sum: 1 },
      media: { $avg: "$edad" }
    }
  }
])
```

ACTIVIDAD 5.16.

Utilizando la colección trabajadores realiza las siguientes consultas:

- Visualiza la edad media, la media de salario y el número de trabajadores que hayan tenido una prima de 30 o de 80.
- Visualiza por población el número de trabajadores, el salario medio y el máximo salario.
- Visualiza el nombre, ape1 y ape2 del empleado que tiene máximo salario.
- A partir de la consulta anterior, obtén ahora el nombre, ape1, ape2 y salario del empleado que tiene máximo salario por cada población:

Para saber más sobre consultas, consulta las siguientes URLs de MongoDB:

Operadores para agregación: <https://docs.mongodb.com/manual/reference/operator/aggregation/>

Operadores para consultas: <https://docs.mongodb.com/manual/reference/operator/query/>

Operadores de actualización: <https://docs.mongodb.com/manual/reference/operator/update/>

Comandos de MongoDB: <https://docs.mongodb.com/manual/reference/command/>
 Funciones para las colecciones: <https://docs.mongodb.com/manual/reference/method/js-collection/>

5.6.9. Relaciones entre documentos en MongoDB

MongoDB utiliza 2 métodos o patrones que nos van a permitir establecer la estructura de los documentos y sus relaciones. Vamos a ver cómo son las relaciones entre documentos comparándolas con el modelo relacional. Los métodos son utilizar referencias:

- **Referencias Manuales:** en la que se guarda el campo `_id` de un documento como referencia en otro documento. Similar al concepto de clave ajena del modelo relacional. En este método la aplicación debe ejecutar una segunda consulta para devolver los datos relacionados. Este es el método más utilizado.

Ejemplo: se van a crear las colecciones `emple` y `depart`, cada elemento con su `_id` creado manualmente. Y se va a simular una relación 1 a muchos, 1 departamento va a tener a varios empleados.

Colección `emple`, con 4 empleados.

```
db.emple.insert({_id:'emp1', nombre:"Juan", salario:1000,
fechaalta:"10/10/1999"})
db.emple.insert({_id:'emp2', nombre:"Alicia", salario:1400,
fechaalta:"07/08/2000", oficio: "Profesora"})
db.emple.insert({_id:'emp3', nombre:"María Jesús", salario:1500,
fechaalta: "05/01/2005", oficio: "Analista", comisión:100})
db.emple.insert({_id:'emp4', nombre:"Alberto", salario:1100,
fechaalta:"15/11/2001"})
```

Colección `depart` con dos departamentos, asignamos los dos primeros empleados al primer departamento, y los dos siguientes al segundo. Para asignar los empleados ponemos el nombre de la colección (`emple`) y entre corchetes dentro de un array de referencias, los `_id` de los empleados a incluir, por ejemplo `emple:['emp1', 'emp2']`:

```
db.depart.insert({_id:'dep1', nombre:"Informática", loc:'Madrid',
emple:['emp1', 'emp2']})
db.depart.insert({_id:'dep2', nombre:"Gestión", loc:'Talavera',
emple:['emp3', 'emp4']})
db.depart.find()
```

Para visualizar los datos de la combinación de las colecciones necesitaremos hacer dos consultas, una para obtener el departamento a consultar, y la otra para obtener los empleados de ese departamento, que están dentro del array del departamento. Por ejemplo, se desea visualizar los empleados del departamento con identificativo `_id` igual a `dep1`.

1º Cargamos el departamento con `_id:dep1` en una variable, utilizamos el método `.findOne`. El método `.findOne()` siempre incluye el campo `_id` incluso si el campo no se especifica explícitamente en el parámetro de consulta:

```
departrabajo = db.depart.findOne({_id:'dep1'})
```

2º Recuperamos los empleados cuyo `_id` se encuentre enlazado a este departamento (`departrabajo` en el ejemplo):

```
emplestdep = db.emple.find({ _id: { $in : departrabajo.emple } } )
```

Si se añade el método `.toArray` los datos se devuelven en una matriz que contiene todos los documentos de la consulta, es decir, devuelve un array de documentos:

```
emplestdep = db.emple.find({ _id: { $in : departrabajo.emple } } )  
.toArray()
```

La siguiente consulta devuelve los empleados del departamento dep2 que tienen el salario > de 1400:

Se carga el departamento: `departrabajo = db.depart.findOne({ _id: 'dep2' })`

Y luego los empleados:

```
emplestdep = db.emple.find({ _id: { $in : departrabajo.emple } },  
salario: { $gt: 1400 } ) .toArray()
```

ACTIVIDAD 5.17.

Utilizando la transformación del documento `sucursales.xml` a JSON, realizada en los apartados anteriores, se pide crear las colecciones `cuentas` y `sucursales`. Para crearlas primero crea la colección `cuentas` y luego crea la colección `sucursales` asignando a las sucursales las cuentas correspondientes.

Una vez creadas las colecciones realiza las siguientes consultas:

- Visualiza las cuentas de las sucursales de Madrid.
- Visualiza las cuentas con `saldohaber > 10000` cuyo director sea Fernando Rato
- Sube 300 el `saldohaber` de las cuentas de la sucursal con código SUC1 .

-
- **DBRefs** son referencias de un documento a otro utilizando el valor del campo `_id` del primer documento, el nombre de la colección, y, opcionalmente, el nombre de base de datos. Con la inclusión de estos nombres, los DBRefs permiten que documentos que se encuentran en varias colecciones sean vinculados para ser documentos de una sola colección. Los **DBRefs** proporcionan en esencia una semántica común para la representación de los vínculos entre documentos. Los **DBRefs** también requieren consultas adicionales para devolver los documentos de referencia.

5.6.10. Herramienta Robomongo

Robomongo es una herramienta multiplataforma con la que se pueden administrar de forma más visual las bases de datos MongoDB.

Esta herramienta integra la Shell de mongoDB en un entorno gráfico con todas sus funcionalidades, además se podrá trabajar con múltiples conexiones a las bases de datos, podremos navegar por las colecciones y a la hora de hacer las consultas contaremos con el resaltado de sintaxis y autocompletado del código, muy útil para detectar los errores.

Robomongo se descarga de la URL <https://robomongo.org/>. Existe la versión instalable la .exe, o la versión portable .zip. En esta unidad se ha optado por utilizar la versión portable robomongo-0.9.0. Para trabajar con ella descomprimimos el zip en una carpeta y ejecutamos el archivo **Robomongo.exe**. Pide conexión con la base de datos, debemos tener la base de datos arrancada para que la detecte. El puerto de escucha de MongoDB es 27017. Véase Figura 5.21.

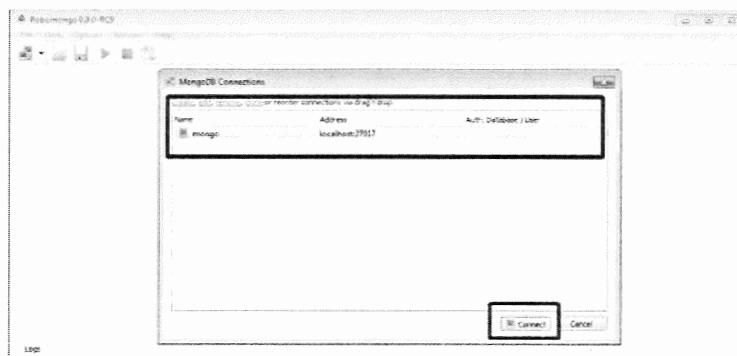


Figura 5.21. Conexión a MongoDB desde Robomongo.

Desde la ventana de Robomongo podemos navegar por las bases de datos, las colecciones, y los objetos de las colecciones. Al hacer clic en una colección se abrirá una pestaña donde se podrán ver los documentos de la colección, también podemos ver los campos del documento si hacemos doble clic en el objeto o si lo desplegamos. En la parte superior es donde escribiremos las consultas, véase la Figura 5.22.

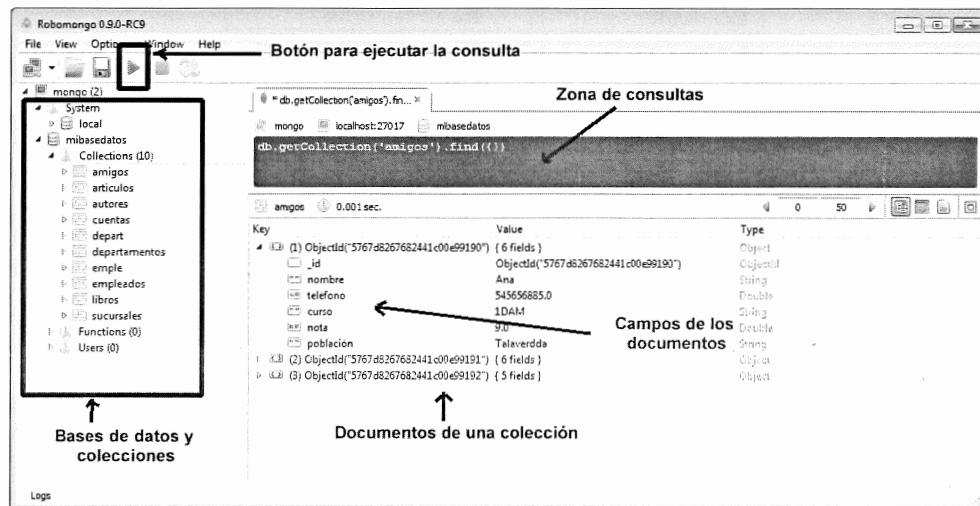


Figura 5.22. Ventana de trabajo de Robomongo

Desde los menús contextuales se podrán realizar todo tipo de operaciones:

- Desde el menú contextual de la conexión a la base de datos podremos crear nuevas bases de datos, abrir la *Shell* de MongoDB, que se abrirá en una pestaña para hacer las consultas, obtener información de la base de datos o desconectarnos.
- Desde el menú contextual asociado a una base de datos, además de abrir la *Shell*, se podrán obtener estadísticas, reparar o borrar la base de datos.

- Desde el menú contextual de una colección podremos ver los documentos, insertar, modificar o borrar documentos. Podremos cambiar el nombre de la colección, hacer una copia o borrar la colección.

5.6.11 MongoDB en Java

Para trabajar en Java con MongoDB necesitamos descargar el driver desde la URL de MongoDB <https://mongodb.github.io/mongo-java-driver/>. La versión que se va a utilizar en este libro es la **mongodb-java-driver-3.2.2.jar**.

Antes de trabajar con MongoDB definamos BSON: **BSON** es un formato de serialización binaria, se utiliza para almacenar documentos y hacer llamadas a procedimientos en MongoDB. La especificación BSON se encuentra en bsonspec.org. BSON soporta los siguientes tipos de datos como valores en los documentos, cada tipo de dato tiene un número y un alias que se pueden utilizar con el operador **\$type** para consultar los documentos por tipo BSON. Algunos de los tipos BSON son los siguientes:

| Tipo | Número | Alias |
|-------------|--------|-------------|
| Double | 1 | “double” |
| String | 2 | “string” |
| Object | 3 | “object” |
| Array | 4 | “array” |
| Binary data | 5 | “binData” |
| ObjectId | 7 | “objectId” |
| Boolean | 8 | “bool” |
| Date | 9 | “date” |
| Null | 10 | “null” |
| Symbol | 14 | “symbol” |
| Timestamp | 17 | “timestamp” |

Al comparar los valores de los diferentes tipos BSON, MongoDB utiliza el siguiente orden de comparación, de menor a mayor: *Null, Numbers (ints, longs, doubles), Symbol, String, Object, Array, BinData, ObjectId, Boolean, Date, Timestamp*

CONEXIÓN A LA BD

Para conectarnos a la base de datos creamos una instancia **MongoClient**, por defecto crea una conexión con la base de datos local, y escucha por el puerto 27017. Todos los métodos relacionados con operaciones **CRUD (Create, Read, Update and Delete)** en Java se acceden a través de la interfaz **MongoCollection**. Las instancias de **MongoCollection** se pueden obtener a partir de una instancia **MongoClient** por medio de una **MongoDatabase**. Así pues para conectarme a la base de datos *mibasedatos* y a la colección *amigos* escribiré lo siguiente:

```
MongoClient cliente = new MongoClient();
MongoDatabase db = cliente.getDatabase("mibasedatos");
MongoCollection <Document> colección = db.getCollection("amigos");
```

También se puede escribir el nombre del servidor y el puerto:

```
MongoClient cliente = new MongoClient("localhost", 27017);
```

MongoCollection es una interfaz genérica: el parámetro de tipo TDocument es la clase que los clientes utilizan para insertar o modificar los documentos de una colección, y es el tipo predeterminado para devolver búsquedas (**find**) y agregados (**aggregate**). El método de un solo argumento **getCollection** devuelve una instancia de **MongoCollection < Document >**, y así es como podemos trabajar con instancias de la clase de documento.

VISUALIZAR LOS DATOS DE UNA COLECCIÓN

Los datos de una colección se pueden cargar en una lista utilizando el método **find().into()** de la siguiente manera:

```
List<Document> consulta = colección.find()
    .into(new ArrayList<Document>());
for (int i = 0; i < consulta.size(); i++) {
    System.out.println(" - " + consulta.get(i).toString());
}
```

También podemos recuperar los valores de los campos del documento, utilizando los métodos **get** del objeto **Document**, reciben como parámetro el nombre de la clave. Si se sabe el tipo de dato de la clave elegiremos el método correspondiente, y si no utilizamos **get()** que devuelve un objeto. Primero cargamos el elemento de la lista en un **Document**. Si la clave no existe en el documento visualizará null:

```
for (int i = 0; i < consulta.size(); i++) {
    Document amig = consulta.get(i);
    System.out.println(" - " + amig.getString("nombre") + " - " +
        amig.getDouble("teléfono") + " - " +
        amig.getString("curso") + " - " + amig.getDouble("nota"));
}
```

INSERTAR DOCUMENTOS

Para insertar documentos, creamos un objeto **Document**, con el método **put** asignamos los pares *clave-valor*, donde el primer parámetro es el nombre del campo o la clave, y el segundo el valor. Y con el método **insertOne** se inserta en la colección. El siguiente código añade un amigo a la colección:

```
Document amigo = new Document();
amigo.put("nombre", "Pepito");
amigo.put("teléfono", 925677);
amigo.put("curso", "2DAM");
amigo.put("fecha", new Date());
colección.insertOne(amigo);
```

También se puede insertar documentos utilizando el método **append** de **Document**. Por ejemplo, se va a insertar el siguiente documento, se crea en **curso** un nuevo documento con dos pares *clave-valor*:

```
{ "Nombre" : "Pedro", "teléfono" : 12345,
  "curso" : { curso1: "1DAM", curso2: "2DAM" } }
```

```
Document amigo2 = new Document("nombre", "Pedro")
    .append("teléfono", 1234)
    .append("curso", new Document("curso1",
        "1DAM").append("curso2", "2DAM"));
colección.insertOne(amigo2);
```

A la hora de visualizar el curso utilizaremos el método **get()** en lugar de **getString()**.

Se puede insertar en la base de datos una lista de documentos en una colección utilizando el método ***insertMany***. Por ejemplo:

```
List<Document> listadocs = new ArrayList<Document>();
for (int i = 0; i < 100; i++) {
    listadocs.add(new Document("Valor de i", i));
}
colección.insertMany(listadocs);
```

Si se desea saber los documentos de la colección se puede utilizar el método ***count***:

```
colección.count();
```

CONSULTAR DOCUMENTOS

Anteriormente se ha visto cómo cargar los documentos en una lista utilizando el método ***find().into()***. El método ***find()*** devuelve un cursor, devuelve una instancia ***FindIterable***. Podemos utilizar el método ***iterator()*** para recorrer el cursor. En el ejemplo recuperaremos todos los documentos de la colección y se visualizan en formato ***Json***:

```
MongoCursor<Document> cursor = colección.find().iterator();
while (cursor.hasNext()) {
    Document doc = cursor.next();
    System.out.println(doc.toJson());
}
cursor.close();
```

Si solo se desea obtener el primer documento utilizamos el método ***first()***:

```
Document doc = colección.find().first();
```

UTILIZAR FILTROS EN LAS CONSULTAS

El método ***find()*** admite la utilización de filtros. Para utilizar los métodos de la clase *Filters* hacemos un ***import static*** de la clase *Filters* de la siguiente manera:

```
import static com.mongodb.client.model.Filters.*;
```

En el ejemplo se busca el documento con la clave *nombre* igual a *Ana*, devuelve solo el primero, por añadir el método ***first***:

```
Document doc = colección.find(eq("nombre", "Ana")).first();
try {
    System.out.println("Localizado: " + doc.toJson());
} catch (NullPointerException e) {
    System.out.println("No se encuentra.");
}
```

Si el filtro devuelve varios documentos los recuperamos con un cursor, o bien con una lista como ya vimos al principio. En el ejemplo recuperamos los datos de los amigos con nota >5 y se visualizan en formato Json:

```
MongoCursor<Document> docs = colección.find(gt("nota", 5)).iterator();
while (docs.hasNext()) {
    Document docu = docs.next();
    System.out.println(docu.toJson());
}
docs.close();
```

Esta condición recupera los amigos con un 5 o un 8 en nota:

```
colección.find( or(eq("nota", 5), eq("nota", 8)) )
```

Esta condición recupera los amigos de 1DAM y nota 8:

```
colección.find( and(eq("curso", "1DAM"), eq("nota", 8)) )
```

Si se desea extraer los objetos *BSON* de un documento, utilizaremos los filtros:

```
System.out.println(" - --- Objetos Bson -----");
MongoCursor<Document> cursor2 = colección.find().iterator();
while (cursor2.hasNext()) {
    Document doc2 = cursor2.next();
    Bson id = eq("_id", doc2.get("_id"));
    Bson nombre = eq("nombre", doc2.get("nombre"));
    Bson curso = eq("curso", doc2.get("curso"));
    System.out.println("Id: " + id + ". Nombre: "
        + nombre.toString() + ". Curso : " + curso.toString());
}
```

ORDENAR RESULTADOS

Para ordenar el resultado de una consulta importamos los métodos de la clase *Sorts*:

```
import static com.mongodb.client.model.Sorts.*;
```

La siguiente condición consulta los amigos del curso 2DAM ordenados descendenteamente por el campo *nombre*:

```
colección.find(eq("curso", "2DAM"))
    .sort(descending("nombre")).iterator();
```

UTILIZAR PROYECCIONES

A veces no se necesitan todos los datos contenidos en un documento, se pueden utilizar proyecciones para cambiar las salidas. Se necesitan importar los métodos de la clase *Projection*, estos métodos devuelven un tipo *BSON*, que podrá ser utilizado en otro método. El *import* debe ser el siguiente

```
import static com.mongodb.client.model.Projections.*;
```

El siguiente ejemplo devuelve el nombre y la nota de los amigos del curso 1DAM, se utiliza el método *include* para añadir solo nombre y nota:

```
MongoCursor<Document> docs3 = colección.find(eq("curso", "1DAM"))
    .sort(ascending("nombre"))
    .projection(include("nombre", "nota")).iterator();
while (docs3.hasNext()) {
    Document docu = docs3.next();
    System.out.println(docu.toJson());
}
docs3.close();
```

Si no se desea incluir en la consulta algunos de los campos utilizamos el método *exclude*, en la consulta no se desea visualizar el *_id*, la nota y el nombre de *Ana*:

```
Document dd = colección.find(eq("nombre", "Ana"))
    .projection(exclude("_id", "nota", "nombre")).first();
```

UTILIZAR AGREGACIONES

Para utilizar los agregados se necesitan importar los métodos de la clase *Aggregates*. Cada método devuelve una instancia del tipo *BSON*, que a su vez se puede pasar al método de agregado de *MongoCollection*. El *import* debe ser el siguiente:

```
import static com.mongodb.client.model.Aggregates.*;
```

Para añadir las etapas de agregado se utiliza un *Arrays.asList* de *java.util*. Este ejemplo visualiza los amigos del curso 1DAM, utiliza la etapa *match*:

```
MongoCursor<Document> docs4 =leccion.aggregate(
    Arrays.asList(match(eq("curso", "1DAM")))).iterator();
while (docs4.hasNext()) {
    Document docu = docs4.next();
    System.out.println(docu.toJson());
}
docs4.close();
```

En el siguiente ejemplo se visualiza el nombre y la nota de los amigos de 1DAM, se utiliza también la la etapa *project*:

```
MongoCursor<Document> docs5 =leccion.aggregate(Arrays.asList(
    match(eq("curso", "1DAM")),
    project(fields(include("nombre", "nota"), excludeId())))
).iterator();
```

Calculo ahora la suma y la nota media, agrupada por curso. Para utilizar las funciones de cálculo importamos los métodos de la clase *Accumulator*:

```
import static com.mongodb.client.model.Accumulators.*;
```

La consulta quedará así:

```
MongoCursor<Document> docs5 =leccion.aggregate(Arrays.asList(
    group("$curso", sum("sumanota", "$nota"), avg("medianota", "$nota"))
)).iterator();
```

Si se desea calcular una media, o una suma de un campo, global para todos los documentos, el primer parámetro del *group* lo dejamos vacío:

```
group("", sum("sumanota", "$nota"), avg("medianota", "$nota"))
```

Si se desea que la salida de la consulta se almacene en una nueva colección en la base de datos añadimos la etapa *out*. En el ejemplo las notas medias y las sumas de las notas se almacenarán en la colección *mediascurso*:

```
MongoCursor<Document> docs7 =leccion
.aggregate(Arrays.asList(group("$curso", sum("sumanota", "$nota"),
    avg("medianota", "$nota")),
    out("mediascurso")))
.iterator();
```

ACTUALIZAR DOCUMENTOS

Para realizar actualizaciones se necesita importar los métodos de la clase *Updates*:

```
import static com.mongodb.client.model.Updates.*;
```

En este ejemplo actualizamos la nota de *Ana* y la asignamos la nota 5. Para actualizar un único documento se utiliza el método ***updateOne***, si hay varios con nombre Ana, actualiza el primero. La actualización devuelve un ***UpdateResult***:

```
colección.updateOne(eq("nombre", "Ana"), set("nota", 5));
```

Si ahora deseamos actualizar varios registros que cumplan una condición utilizamos el método ***updateMany***, este ejemplo sube la nota a todos los amigos de 1DAM, la sube 1 punto. La actualización devuelve un ***UpdateResult*** que tiene métodos para decir cuántos se seleccionan y cuántos se actualizan:

```
UpdateResult updateResult = colección.updateMany(
    eq("curso", "1DAM"), inc("nota", 1));
System.out.println("Se han modificado: " +
    updateResult.getModifiedCount());
System.out.println("Se han seleccionado: " +
    updateResult.getMatchedCount());
```

Si se desean actualizar todos los registros de la colección, utilizamos la función ***exists("_id")*** para que devuelvan todos los documentos que tengan *_id*:

```
updateResult = colección.updateMany(exists("_id"), inc("nota", 2));
```

Para añadir un campo se utiliza también la función ***set***, si el campo no existe lo crea. Esta consulta añade la población a Marleni:

```
colección.updateOne(eq("nombre", "Marleni"), set("población", "Toledo"));
```

Para eliminar un campo utilizamos el método ***unset***, en el ejemplo borro la población a los de 1 DAM, borrará el campo a los que lo tengan:

```
colección.updateMany(eq("curso", "1DAM"), unset("población"));
```

BORRAR UN DOCUMENTO DE LA COLECCIÓN

Como en el caso anterior, para borrar un documento de la colección utilizamos el método ***deleteOne***, y para borrar varios ***deleteMany***. También se devuelve ruEn el ejemplo se borra el documento con nombre María, borrará solo el primero. En el segundo borra todos los documentos de la colección. Devuelven un ***DeleteResult***.

```
// Borro un documento
DeleteResult del = colección.deleteOne(eq("nombre", "Ana"));
System.out.println("Se han borrado: " + del.getDeletedCount());
//Borro todos
del = colección.deleteMany(exists("_id"));
System.out.println("Se han borrado: " + del.getDeletedCount());
```

CREAR Y BORRAR UNA COLECCIÓN

Para crear una colección utilizamos el método ***createCollection***, asociado a la base de datos, y para borrarla utilizamos el método ***drop*** asociado a la colección:

```
//Crear Colección:
MongoClient cliente = new MongoClient();
MongoDatabase db = cliente.getDatabase("mibasedatos");
db.createCollection("nuevacolección");
```

```
//Borro la colección
MongoCollection<Document> cnueva = db.getCollection("nuevacoleccion");
cnueva.drop();
```

LISTAR LAS COLECCIONES DE LA BASE DE DATOS

El método *listCollectionNames* devuelve las colecciones de la base de datos en un *MongoIterable*, para listar la lista podemos elegir una de las siguientes maneras:

```
// Listar las colecciones de la BD
System.out.println("Listado de colecciones: ");
MongoIterable<String> colecciones = db.listCollectionNames();
Iterator col = colecciones.iterator();
while (col.hasNext())
    System.out.println(col.next());

// También se pueden listar así:
for (String name : db.listCollectionNames()) {
    System.out.println(name);
}
```

CREAR, LISTAR Y BORRAR BASES DE DATOS

Para crear una base de datos se llama al método *getDatabase* desde un objeto *MongoClient*, sin embargo, la base de datos no se creará hasta que no se inserte un documento. El siguiente código creará la base de datos *nueva* y la colección *colecnueva* con un documento:

```
MongoClient cliente = new MongoClient();
MongoDatabase db = cliente.getDatabase("nueva");
MongoCollection<Document> clnue = db.getCollection("colecnueva");
Document doc1 = new Document("nombre", "Pedro").append("teléfono",
    1234).append("curso", "2DAM");
clnue.insertOne(doc1);
```

El siguiente código obtiene los nombres de las bases de datos:

```
for (String name: cliente.listDatabaseNames()) {
    System.out.println(name);
}
```

Par borrar una base de datos se utiliza la orden:

```
cliente.getDatabase("base_datos_a_borrar").drop();
```

PASAR DATOS DE MONGODB A UN FICHERO DE TEXTO

En este método extraemos los documentos de la colección *amigos* y los guardamos en un fichero de texto en formato JSON llamado *amigos.json*.

```
public static void crearficherojson() {
    MongoClient cliente = new MongoClient();
    MongoDatabase db = cliente.getDatabase("mibasedatos");
    MongoCollection<Document> colección = db.getCollection("amigos");

    File fiche = new File("./amigos.json");
    FileWriter fic;
    try {
```

```

fis = new FileWriter(fiche);
BufferedWriter fichero = new BufferedWriter(fis);
// Recorro la colección amigos:
System.out.println(" -----");
List<Document> consulta = colección.find()
    .into(new ArrayList<Document>());
for (int i = 0; i < consulta.size(); i++) {
    System.out.println(" Grabo elemento " + i +
        ", " + consulta.get(i).toString());
    fichero.write(consulta.get(i).toJson());
    fichero.newLine();
}
fichero.close();
} catch (IOException e) {e.printStackTrace();}
}

```

En este método hacemos lo contrario, leemos el fichero de texto **amigos.json** que contiene documentos JSON, y los almacenamos en la BD en una colección llamada **amigosfile**. Para convertir una cadena a formato JSON utilizamos el método **parse** de la clase **Document**:

```

private static void leerficheroyguardardatos() {
try {
    FileReader fr = new FileReader("./amigos.json");
    BufferedReader bf = new BufferedReader(fr);

    MongoClient cliente = new MongoClient();
    MongoDatabase db = cliente.getDatabase("mibasedatos");
    MongoCollection<Document> colección =
        db.getCollection("amigosfile");

    String cadenajson;
    while ((cadenajson = bf.readLine()) != null) {
        System.out.println(cadenajson);
        Document docu = new
            Document(org.bson.Document.parse(cadenajson));
        colección.insertOne(docu);
    }
} catch (FileNotFoundException e) {e.printStackTrace();}
} catch (IOException e) { e.printStackTrace();}
}

```

ACTIVIDAD 5.18.

Realiza un método que lea los datos de la tabla Empleados de la BD MySQL *ejemplo* (creada en los primeros temas), y guarde los datos en MongoDB en una colección con nombre **empleadosmysql**. Los nombres de las claves de los documentos deben ser los nombres de las columnas de la tabla, convertidas a minúscula, utiliza la Interfaz **ResultSetMetaData** de jdbc.

Crea también los siguientes métodos:

Para visualizar los datos de la colección creada.

Para crear un fichero de texto con los datos de esta colección en formato JSON.

Método para subir 100 al salario de los empleados del oficio EMPLEADO

COMPRUEBA TU APRENDIZAJE

1. A partir del documento ***departamentos.xml*** de la colección ***ColeccionPruebas***. Realiza un programa Java para gestionar dicho documento. Utiliza las clases **Main.java** y **Pantalla.java** que se incluyen en la carpeta de recursos del tema.

El programa debe mostrar la pantalla inicial (Figura 5.23) donde podremos consultar, insertar, borrar y eliminar nodos <DEPT_ROW> del documento *departamentos.xml*.

En todas las operaciones nos aseguraremos de trabajar con el documento ***departamentos.xml***, utilizaremos la función ***doc(documento)*** en las consultas.

Comprobar que al insertar un departamento no exista ya en el documento, igualmente a la hora de borrar o de modificar hay que comprobar que el departamento exista. Visualizar los mensajes informativos en cada caso.

Crear una clase para gestionar las operaciones sobre el documento, esta clase debe contener los métodos para conectarnos a eXist, insertar, borrar, modificar o consultar en el documento. Utilizar las *Sentencias de actualización de eXist*.

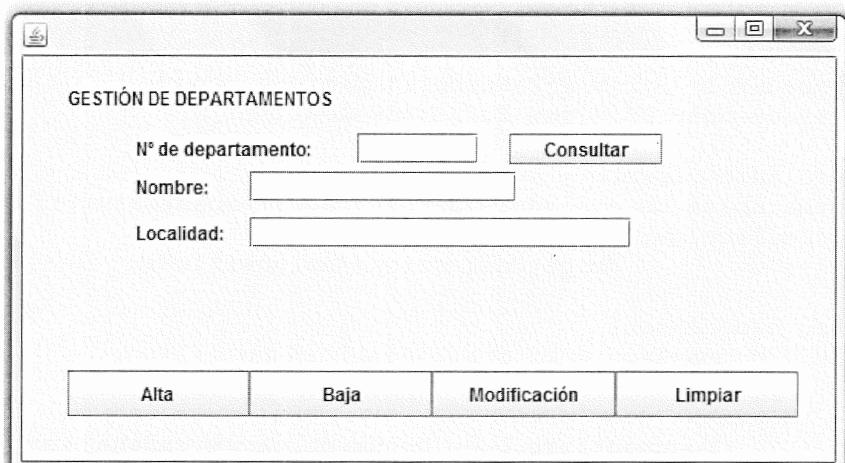


Figura 5.23. Pantalla inicial ejercicio Gestión de Departamentos.

A partir de la colección VENTAS

Haz un programa Java que cree la colección **VENTAS** y suba los documentos que se encuentran en la carpeta ***ColeccionVentas***. Esta colección contiene los siguientes documentos xml:

clientes.xml: contiene datos de clientes que compran los productos. Por cada cliente tenemos

```
<clien numero="nn"> <nombre>xxxxxx</nombre>
    <poblacion>xxxxxxxx</poblacion>
    <tlf>xxxxxx</tlf>
    <direccion>xxxxxxxx</direccion>
</clien>
```

productos.xml: contiene los datos de los productos que son comprados por los clientes. Por cada producto tenemos su categoría y precio, que van como atributos, el código de producto, el nombre y el stock del producto:

```
<product categoria="xxx" pvp="xxxx">
```

```

<codigo>xxxxxx</codigo>
<nombre>xxxxxx</nombre>
<stock>xxxxxx</stock>
</product>
```

facturas.xml: contiene los datos de las facturas de los clientes. Por cada factura tenemos el número de factura como atributo, la fecha de factura, el importe y el número de cliente, los nodos son:

```

<factura numero="xxxx">
    <fecha>xxxxxx</fecha>
    <importe>xxxxxx</importe>
    <numcliente>xxxxxx</numcliente>
</factura>
```

detallefacturas.xml: contiene el detalle de cada factura, es decir, los datos de los productos y las unidades compradas por los clientes. También cada producto de la factura lleva asociado un porcentaje de descuento. Por cada factura se dispone de la siguiente información

```

<factura numero="xxxx">
    <codigo>xxxxxx</codigo>
    <producto descuento="xxx">
        <codigo>xxx</codigo>
        <unidades>xxxx</unidades>
    </producto>
    <producto descuento="xxx">
        <codigo>xxxx</codigo>
        <unidades>xxxx</unidades>
    </producto>
    . . .
</factura>
```

Realiza los siguientes ejercicios:

2. Realiza una consulta XQuery para obtener las facturas que tiene cada cliente, que aparezca solo el nombre del cliente y el número de factura entre las etiquetas `<facturaclientes></facturaclientes>`. Debe obtener algo parecido a esto

```

<facturasclientes> <nombre>Pilar Martín</nombre><nufac>10</nufac></facturasclientes>
<facturasclientes> <nombre>Pilar Martín</nombre><nufac>11</nufac></facturasclientes>
<facturasclientes> <nombre>Antonio Reus</nombre><nufac>12</nufac></facturasclientes>
. . .
```

3. Realiza una consulta XQuery para obtener el detalle de las ventas de los productos de la factura número 10. Obtén por cada producto de esa factura, el código, nombre, la cantidad vendida, el precio, y el importe que será la suma de las unidades por el precio del producto. Utiliza los documentos **productos.xml** y **detallefacturas.xml**. La salida debe ser similar a lo que se muestra:

```

<detalle><codigo>1</codigo><nombre>Silla Plegable</nombre>
    <cant>10</cant><pvp>100</pvp><importe>1000</importe>
</detalle>
<detalle><codigo>2</codigo><nombre>BH Prisma</nombre>
    <cant>3</cant><pvp>900</pvp><importe>2700</importe>
</detalle>
. . .
```

4. Realiza una consulta XQuery para obtener el detalle de las ventas de los productos de cada una de las facturas del documento ***detallefacturas.xml***. Crea una salida que muestre para cada factura, el número de factura y el código como atributos dentro de la etiqueta **<factura>**. Y a continuación los artículos, el código de artículo será un atributo de la etiqueta **<artículo>**. Utiliza los documentos ***productos.xml*** y ***detallefacturas.xml***. La salida debe ser similar a lo que se muestra:

```

<factura numero="10" codigo="FACT10">
  <articulo codigo="1"><nombre>Silla Plegable</nombre>
    <cant>10</cant><pvp>100</pvp><importe>1000</importe>
  </articulo>
  <articulo codigo="2"><nombre>BH Prisma</nombre>
    <cant>3</cant><pvp>900</pvp><importe>2700</importe>
  </articulo>
  . . .
</factura>
<factura numero="11" codigo="FACT11">
  . . .
  
```

5. Haz un programa Java que genere un fichero XML a partir de esta consulta, llamarle al nuevo fichero ***totalfacturas.xml***, el elemento raíz se llamará también ***totalfacturas***. Una vez creado subir el fichero a la colección *Ventas* de la base de datos. Recuerda que el documento XML debe estar bien formado porque si no, no se podrá cargar en la base de datos. Crea un método para cada operación.
6. Utilizando el documento creado (***totalfacturas.xml***), realiza una consulta de actualización para actualizar la etiqueta **<importe>** del documento ***facturas.xml***. La actualización consiste en actualizar el importe de cada factura con la suma de los importes de los artículos que componen la factura, obtén la suma de importes por cada factura a partir del documento creado anteriormente (***totalfacturas.xml***).
7. Utilizando los documentos ***facturas.xml*** (ya actualizado) y ***clientes.xml***, realiza una consulta XQuery para obtener todo lo que tiene que pagar cada cliente. La salida debe ser similar a:

```

<clien>
  <nombre>Alicia Díaz</nombre>
  <total>0</total>
</clien>
<clien>
  <nombre>Pilar Martín</nombre>
  <total>7880</total>
</clien>
  . . .
  
```

8. Utilizando los datos devueltos por la consulta anterior, haz un programa Java para obtener el nombre del cliente que más importe tiene que pagar. Guardar el resultado de la consulta en disco, en la carpeta del proyecto, y luego hacer la consulta sobre el fichero que se encuentra en la carpeta. Recuerda para hacer consultas a ficheros en disco pondremos el camino del documento, en este ejemplo el fichero a consultar se encuentra en D:/Misdatos y se llama prueba.xml:

```

for $pru in doc('file:///D:/Misdatos/prueba.xml')/prueba/datos
return $pru
  
```

ACTIVIDADES DE AMPLIACIÓN

1. Crea la BD MySQL BDARTICULOSCLien, copia y ejecuta el script para crear las tablas y las relaciones que se muestran en la figura. El script se encuentra en la carpeta de recursos de la unidad.

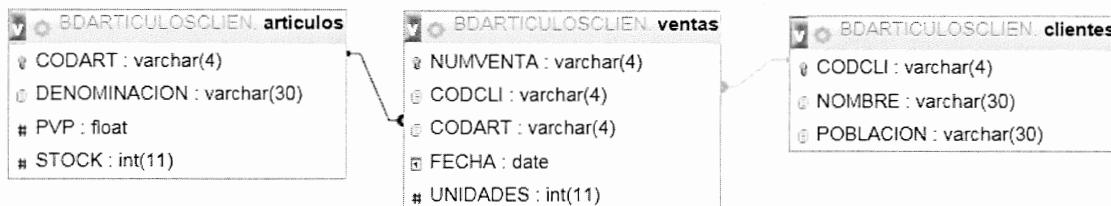


Figura 5.24. Tablas y relaciones del ejercicio.

A partir de estas tablas se pide hacer un programa Java para cargar los datos de estas tablas en la BD **mibasedatos** de MongoDB. Hay que crear una colección para cada tabla: *artículos*, *clientes* y *ventas*. Y, además, en las ventas hay que añadir en lugar del *codcli* y *codart*, la referencia al cliente y al artículo de las correspondientes colecciones para simular las claves ajena. Los nombres de las claves de los documentos deben ser los nombres de las columnas de la tabla, convertidas a minúscula, utiliza la *Interfaz ResultSetMetaData* de jdbc.

Utiliza como identificativo de cada documento (el *_id*) las claves primarias de las tablas MySQL para poder hacer así las referencias.

Una vez creados todos los documentos realiza en Java los siguientes métodos con MongoDB:

- Obtén el detalle de cada venta, los datos a obtener son:

Numventa, codcliente, nombre, codarticulo, denominación, fecha venta, unidades, importe.

Importe es el PVP * UNIDADES

- Visualiza para cada artículo, su código, su denominación, el total de unidades vendidas (suma de unidades), el total de importe (suma de unidades * pvp), y el stock actual (resta de stock menos unidades)
- Visualiza para cada cliente: su código, su nombre, el número de compras y el total de las compras que será la suma de unidades * pvp.