
**Universidad Jaime I**

Dep. de Lenguajes y Sistemas Informáticos

# Consultas analíticas y visualización para datos abiertos enlazados

**Trabajo Fin de Máster**

**SIU043**

— Autor —

Iván Posilio Gellida

—Tutores—

Rafael Berlanga Llavori y María Victoria Nebot Romero

23 de diciembre de 2014

# Índice general

# Índice de figuras

# Capítulo 1

# Introducción

## 1.1. Motivación

Es posible publicar datos poniendo un documento de texto, un archivo HTML o PDF en un sitio web. Y es una buena forma de publicar si el objetivo es proporcionar una narrativa para explicar los datos y que las personas simplemente los lean. Sin embargo, los datos detrás de dicho documento están bloqueados y son inaccesibles. Las personas no pueden tomar esos datos, analizarlos y representarlos. Y por lo tanto, no pueden combinarlos con otros datos para proporcionar nuevos servicios o descubrir nuevos conocimientos. Y ahí es donde entra en juego las tecnologías de la web semántica y los datos enlazados.

La electricidad es un tema importante en nuestra sociedad y por lo tanto es importante analizarlo. Enipedia[1] es un intento de reunir datos e información sobre todas las centrales eléctricas del mundo. El propósito es que esta información esté disponible online para su consulta, visualización, análisis, actualización y expansión. Internamente contiene una base de conocimiento creada a partir de datos estructurados y enlazados de diferentes fuentes. Y ahí es donde nos encontramos con uno de los problemas de los datos enlazados como tecnología semántica. Donde realizar consultas sobre este tipo de datos no es trivial para todo tipo de usuarios. Debido a la dificultad de acceso a esta nueva riqueza de datos semánticos. Provocada por la necesidad de conocer el lenguaje estándar de consulta SPARQL y las tecnologías semánticas asociadas. Además, la mayoría de las personas están acostumbradas a consumir y consultar datos sobre tablas en 2 dimensiones. Por lo tanto, cuantas más personas sean capaces de consultar y analizar datos enlazados, será mejor para explotar el verdadero valor de estos. Con lo que no solo deben de ser accesibles por expertos en web semántica.

[1] `http://enipedia.tudelft.nl/wiki/Main_Page`

## 1.2. Objetivos

- Crear una aplicación web mediante la cual se puedan realizar consultas, de forma interactiva, sobre datos enlazados sin necesidad de conocer SPARQL. Con el objetivo de ayudar a mostrar resultados a los usuarios con relativamente poco conocimiento en las tecnologías de la web semántica.
- Crear un catálogo interno en la aplicación para poder asistir a las consultas visuales. Mediante el cual se presentarán los elementos de Enipedia con los que el usuario podrá realizar las consultas.
- Crear un lenguaje visual para poder realizar consultas sin necesidad de conocer el lenguaje SPARQL. Basado en tablas en dos dimensiones para presentar los elementos de las consultas. De forma que un usuario medio pueda interactuar con un punto de acceso.
- Poder exportar los resultados de las consultas visuales a diferentes formatos que sean útiles para un posterior análisis. El análisis de datos posterior no forma parte del presente proyecto.

# Capítulo 2

# Fundamentos

La realización de búsquedas es uno de los usos más comunes en la web. Y para ello, las herramientas más valiosas son los motores de búsqueda basados en palabras clave. Tales como Bing o Google. Sin embargo, su uso presenta varios problemas asociados:

- Una gran cantidad de páginas quedan fuera del alcance de los buscadores.
- Alta recuperación pero baja precisión. Se recuperan las principales páginas relevantes. Pero también se recuperan muchos documentos medianamente relevantes o irrelevantes.
- Baja recuperación. Hay ocasiones en las que nuestras peticiones no encuentran respuestas.
- No existe referente semántico en las páginas devueltas. Lo que dificulta deducir el significado de una palabra a través del contexto. Por ejemplo, búsquese una palabra de significado ambiguo como banco.

Pero incluso si la búsqueda tiene éxito, es la persona la que debe explorar los documentos seleccionados. Y extraer la información que está buscando de cada uno de ellos. Con lo cual, no hay mucho apoyo para la recuperación de la información [25]. Y esto se convierte en una actividad que requiere mucho tiempo.

Además, los resultados de las búsquedas en Internet no son fácilmente accesibles por otras herramientas de software. Ya que los motores de búsqueda a menudo se aíslan de las aplicaciones.

Hoy en día el significado del contenido web no es accesible para los ordenadores. Debido al hecho de no poder tomar ventaja de las representaciones de los datos. Y eso supone el mayor obstáculo para proporcionar a los usuarios un mejor soporte. La máquina sencillamente no sabe lo que esta información significa. Y por tanto su capacidad de actuación autónoma es muy limitada. Esta misma limitación expresiva hace que la noción de semántica que manejan los buscadores web, se limite a palabras clave con pesos. Pero al tratarse de palabras planas e inconexas, no permiten reconocer ni solicitar significados más elaborados. Por lo tanto, necesitamos:

- Un lenguaje común para la comunicación de la pregunta.
- Que la información no sea ambigua.
- Razonamiento automático.

- Razonamiento con conocimiento común.

Y aquí es donde aparece el concepto web semántica [7]. Permitiendo evolucionar gradualmente la web existente. Proponiendo superar las limitaciones de la web actual mediante la introducción de descripciones explícitas del significado, la estructura interna y la estructura global de los contenidos y servicios disponibles.

## 2.1. Cambio a la web semántica

En 1989, el físico Tim Berners-Lee desarrolló tres tecnologías que hicieron posible que los usuarios se encontraran y compartieran información. Estas fueron el lenguaje HTML, el protocolo HTTP (Hypertext Transfer Protocol) y el sistema de localización de recursos URL (Uniform Resource Locator). Y todo ello para conseguir un espacio común en el cual nos comunicamos al publicar información.

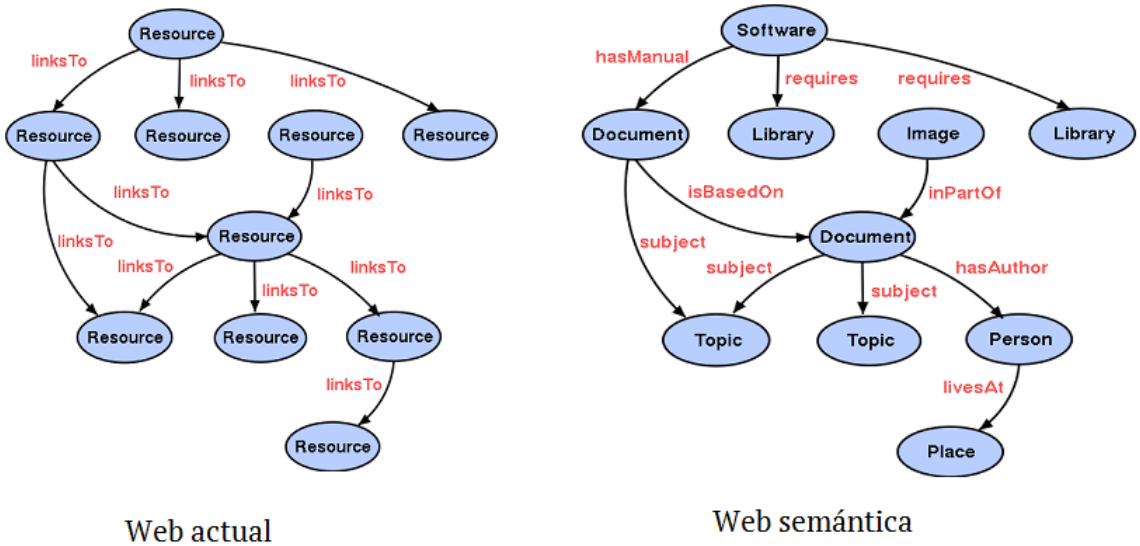



**Figura 2.1**: grafo de la web actual y la web semántica

Con el paso del tiempo, ha habido un crecimiento caótico de recursos y con ausencia de una organización clara. Provocado que la web actual se asemeja a un grafo formado por nodos (recursos) del mismo tipo y arcos (hiperenlaces) igualmente indiferenciados [10]. Lo que hace imposible el acceso a "la información sobre la información" de los recursos. Que ayudaría clasificar, ordenar, relacionar, etc. Sin embargo, tal y como muestra la figura 2.1, la web semántica [7] ha crecido formando una red de nodos tipificados e interconectados mediante clases y relaciones.

La web semántica precisa de definiciones para las relaciones entre conceptos. Junto a reglas lógicas para razonar con ellos. La adopción de definiciones de reglas comunes es clave. Así, todos los que participen de la web semántica puedan trabajar de forma autónoma. Ya sea publicando o consumiendo recursos. Con la garantía de que las piezas encajen. Así, por ejemplo, varias bibliotecas podrían colaborar para dar lugar a una gran *meta-librería* que integre los contenidos de todas ellas. Y un programa que explore esa red de recursos, podría reconocer las distintas unidades de información, obtener datos específicos o razonar sobre relaciones complejas.

## 2.2. Datos enlazados

El termino datos enlazados (Linked Data) [6] se refiere al conjunto de prácticas adecuadas para publicar y enlazar información estructurada en la web. Estos principios fueron introducidos por Tim Berners-Lee [6] [21]:

1. Usar URIs (Uniform Resource Identifier) para nombrar cosas.
2. Utilizar URIs bajo el esquema especificado por el protocolo HTTP, así las personas pueden buscar esos nombres.
3. Cuando alguien busca un URI, proveer información útil, usando estándares como RDF (Resource Description Framework) o SPARQL (SPARQL Protocol and RDF Query Language).
4. Incluir enlaces a otras URIs, así ellos pueden descubrir nuevas cosas.

### 2.2.1. RDF

El paradigma de la web semántica propone un formato común estandarizado. Que posee una sintaxis controlada por el World Wide Web Consortium (W3C) [13]. Además de una semántica expresada en modelos formales. Donde los lenguajes utilizados deben describir el significado del contenido publicado. Y ser capaces de organizarse en capas, posibilitando diferentes grados de comprensión por parte del software [20] [9].

Uno de los pilares básicos de la web semántica es el framework RDF [1] [3], que proporciona un marco de trabajo común para que la información pueda ser intercambiada entre diferentes aplicaciones, sin perder significado.

#### 2.2.1.1. Tripletas RDF

RDF se basa en la idea de describir la realidad a través de afirmaciones [1] [3], que a partir de ahora se llamarán tripletas. Las cuales están compuestas por 3 elementos:

- El sujeto: es aquello que se está describiendo.
- El predicado: es la propiedad o relación que se desea expresar de ese sujeto.
- El objeto: es el valor de la propiedad, que puede ser o bien un literal (un valor constante concreto), o bien otro recurso con el que se establece una relación.

En la figura 2.2 se puede ver una serie de tripletas RDF. Para ello se está utilizando la notación N-Triples [16]. Lo importante es observar que el recurso descrito a través de las tripletas es *http://www.example.com/index.html*. Donde cada predicado declara una de las propiedades: el creador, la fecha de creación y el idioma. Y mientras en la primera tripleta el objeto es otro recurso, en las otras dos sentencias se especifican literales entre comillas.

```
<http://www.testing.com/index.html> <http://purl.org/dc/elements/1.1/creator>
        <http://www.testing.com/staffid/35625> .
<http://www.testing.com/index.html> <http://www.testing.com/terms/creation_date>
        "August 26 , 2001" .
<http://www.testing.com/index.html> <http://purl.org/dc/elements/1.1/language>
        "en" .
```

**Figura 2.2**: Ejemplo de tripleta RDF

En la sección 2.1 se comentó que la web semántica pretende crear un conjunto de nodos interconectados. Con lo que las tripletas RDF pueden visualizarse en forma de grafo [3]. Por ejemplo, un sujeto puede tener varios predicados que lo asocian con otros recursos. Algunos de estos recursos, a su vez, estarán asociados a otros, y así sucesivamente. Tal y como está representado en la figura 2.3.

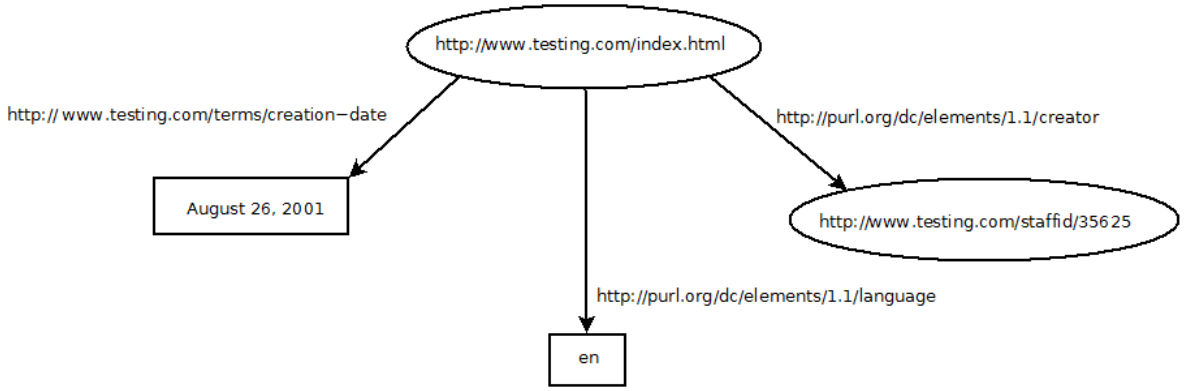



**Figura 2.3**: Grafo representando la tripleta de la figura 2.2

#### 2.2.1.2. RDF y URIs

El nombre de los recursos [21] o cosas de interés es un tema de suma importancia a la hora de publicar datos en Internet. Y para ello se utiliza identificadores únicos: Identificadores Uniformes de Recursos o URIs [5] [33]. Donde cada recurso tendrá un URI diferente. Compuesto por la parte del dominio, que puede ser común para varios URIs con la misma procedencia, y una parte específica de ese recurso concreto.

Los URIs deben ser opacos. El hecho que un URI se parezca a una dirección web es totalmente incidental. Puede que haya o no un sitio web en esa dirección. Y en realidad no importa. Por lo tanto, los URIs son simples nombres para entidades, nada más.

En cada tripleta RDF se identifica unívocamente a que recurso nos estamos refiriendo. Si se pretende que otras personas o máquinas entiendan las relaciones entre los recursos, es necesario utilizar vocabularios estandarizados como: FOAF (Friend of a Friend), DublinCore... A estos vocabularios los llamamos espacios de nombres. Que también estarán identificado por un URI. Y cuyo propósito es el de poner elementos, pertenecientes a un contexto, a disposición de quien utilice el citado vocabulario. Además, para evitar la escritura del URI que identifica el espacio de nombres, cada vez que se utiliza un elemento del mismo, se suelen especificar prefijos.

#### 2.2.1.3. Literales

Se ha comentado que el valor de la propiedad es el elemento objeto de la tripleta RDF. Dicho valor puede ser un recurso, especificado por un URI, o un literal [3]. Este último, formado por una cadena simple de caracteres u otros tipos de datos primitivos.

Hay que tener en cuenta dos cosas de los literales. Primero, el contenido de un literal no es interpretado por RDF en sí mismo. Y segundo, el modelo RDF no permite que los literales sean sujeto de una tripleta.

### 2.2.2. RDF/XML

RDF provee de una sintaxis sobre XML para escribir la representación de grafos o tripletas, conocida como RDF/XML [3] [15]. Siendo uno de los formatos RDF más populares y recomendados por el W3C. A continuación se va a describir la sintaxis básica a partir de la sentencia "*La página http://www.ejemplo.com/index.html fué creada el 30 de febrero*". Donde los elementos de la tripleta a expresar serían los siguientes:

- Sujeto: *http://www.ejemplo.com/index.html*
- Predicado: fecha de creación
- Objeto: 10 de febrero (literal)

Si se supone "ej" como abreviatura de *http://www.ejemplo.com*, se puede utilizar ejterms como prefijo del espacio de nombres "http://www.ejemplo.com/terminos/". Con lo que la secuencia sujeto-predicado-objeto de la tripleta sería: (ej:index.html ejterms:fecha-creacion "10 de febrero"). Y su representación en RDF/XML se muestra en la figura 2.4.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#
    xmlns:ejterms="http://www.ejemplo.com/terminos/">
    <rdf:Description rdf:about="http://www.ejemplo.com/index.html">
        <ejterms:fecha-creacion>10 de febrero</ejterms:fecha-creacion>
    </rdf:Description>
</rdf:RDF>
```

**Figura 2.4**: Ejemplo 1 RDF/XML

A partir del ejemplo de la figura 2.4, se pueden introducir los conceptos más importantes del formato RDF/XML:

- Un documento RDF/XML consiste en una especificación del elemento *rdf:RDF*. Que puede ser parte de un documento XML más grande, o ser el elemento raíz del documento.
- En la línea 2 y 3 se establecen los espacios de nombres.
- Cada etiqueta *rdf:Description* indica la descripción de un recurso. Y se corresponde con el sujeto de una tripleta (actuando de contenedor). Si tiene un atributo *rdf:about* entonces ese es el URI del recurso. Si no lo tiene se trata de un nodo en blanco.
- El contenido de un elemento *rdf:Description* está formado por una serie de sentencias que especifican propiedades. Cada sentencia engloba un par predicado/objeto. Donde el predicado se corresponde con el nombre de la etiqueta XML. Y el objeto

con su contenido. Por lo tanto, en línea 5 se especifica la propiedad *fecha-creacion*, seguida de su valor (30 de febrero). Que indica el objeto de la tripleta.

A continuación se va a mostrar un ejemplo un poco más complejo. Partiendo de las dos sentencias siguientes:

- El blog de John (sujeto) es creado por (predicado) John S. (Objeto)
- John S. (sujeto) es amigo de (predicado) José R. (objeto)

O expresadas mediante URIs:

- http://www.ejemplo.com/blog/john (sujeto), http://purl.org/dc/elements/1.1/creator (predicado), http://www.ejemplo.com/personas/john (objeto)
- http://www.ejemplo.com/personas/john (sujeto), http://xmlns.com/foaf/0.1/knows (predicado), http://www.ejemplo.com/personas/jose (objeto)

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:dc="http://purl.org/dc/elements/1.1/"
   xmlns:foaf="http://xmlns.com/foaf/0.1/">
   <rdf:Description rdf:about="http://www.ejemplo.com/blog/john">
      <dc:creator>
         <rdf:Description rdf:about="http://www.ejemplo.com/personas/john"/>
      </dc:creator>
   </rdf:Description>
   <rdf:Description rdf:about="http://www.ejemplo.com/personas/john">
      <foaf:knows>
         <rdf:Description rdf:about="http://www.ejemplo.com/personas/jose"/>
      </foaf:knows>
   </rdf:Description>
</rdf:RDF>
```

**Figura 2.5**: Ejemplo 2 RDF/XML

En el ejemplo de la figura 2.4, el valor del predicado *ejterms:fecha-creacion* era el literal *10 de febrero*. Pero ahora, como se aprecia en la figura 2.5, los valores de los predicados *dc:creator* y *foaf:knows* son otros recursos, identificados por su URIs. Y por lo tanto, se tienen que expresar con la etiqueta *rdf:Description* y con el atributo *rdf:about* que contendrá la URI identificativa del recurso.

### 2.2.3. Dataset

Un dataset es una colección de datos enlazados, publicados o almacenados en una única fuente. Esta definición general incluye tanto a datasets cuyos datos están representados en formato RDF como a los que no. En el caso del dataset de Enipedia, que utilizamos en el proyecto, el conjunto de datos de alamcenados están en formato RDF.

Es importante mencionar a datahub[1], ya que es la principal plataforma para mantener y publicar datasets de datos abiertos enlazados. Actualmente (al día de escribir esta memoria), hay 8,732 datasets registrados en datahub. Y evidentemente, uno de estos datasets es el de Enipedia (ver figura 2.6).

[1] `http://datahub.io/`

| Field | Value |
|---|---|
| Source | http://enipedia.tudelft.nl |
| Author | Enipedia Team @ Energy and Industry Section, TBM, Delft University of Technology |
| Maintainer | Chris Davis and Alfredas Chmieliauskas |
| links:dbpedia | 1365 |
| links:open-energy-info-wiki | 189 |
| links:world-factbook-fu-berlin | 204 |
| namespace | http://enipedia.tudelft.nl/wiki/ |
| triples | 4180734 |

**Figura 2.6**: Datos del dataset de Enipedia en datahub

### 2.2.4. Extracción de datos enlazados: SPARQL

Una vez integrados todos los datos enlazados, es necesario poder crear consultas que permitan obtener conocimiento. Y para ello, se utiliza una sintaxis derivada de SQL llamada SPARQL. Que está especialmente diseñada para recuperar datos partiendo de documentos RDF o RDF Schema [14] [19]. Además, permite hacer consultas sobre datos que pueden estar distribuidos [32] en diferentes fuentes. Y obtener los resultados en formato RDF, para poder reutilizarlos en las aplicaciones.

#### 2.2.4.1. Puntos de acceso

Un punto de acceso SPARQL permite a los usuarios (humanos o no) consultar una base de conocimientos, contenida en un dataset de datos RDF, mediante el lenguaje SPARQL. Y los resultados se devuelven en uno o más formatos procesables por una máquina. Por lo tanto, un punto de acceso SPARQL se concibe como una interfaz hombre-máquina amigable con una base de conocimientos. El software tras el punto de acceso debe implementar la funcionalidad de formulación de consultas y presentación legible de los resultados.

#### 2.2.4.2. Sintaxis SPARQL

El lenguaje de consulta SPARQL está basado en la comparación de patrones de grafos. Los patrones de grafos contienen patrones de triples. Que son como las tripletas RDF, pero con la opción de poner una variable en las posiciones del sujeto, predicado u objeto. En la figura 2.7 Se puede apreciar como la sintaxis básica de SPARQL es similar a la de SQL. Si se quiere una especificación completa consultar [14].

La función de la palabra clave PREFIX es equivalente a la declaración del espacio de nombres en RDF, es decir, asocia una URI a una etiqueta. Además, SPARQL tiene cuatro formas de consulta. Estas formas de consulta usan los datos de las tripletas tras el punto de acceso. Comparando su concordancia con los patrones triples para formar los conjuntos de resultados. Las formas de consulta son:

- SELECT: Devuelve todas o un subconjunto de las variables vinculadas con un patrón de búsqueda.
- CONSTRUCT: Genera nuevas tripletas RDF a partir de los datos recuperados de las sentencias que cumplen el patrón especificado.
- ASK: Devuelve un valor booleano. Indicando si se encuentra o no una concordancia para un patrón de consulta.

- DESCRIBE: Devuelve un grafo RDF que describe los recursos encontrados.

| Sintaxis básica de una consulta SPARQL | |
|---|---|
| **Prologo (opcional)** | BASE <URI><br>PREFIX prefix: <URI> (pueden haber más de uno) |
| **Formas de consulta (obligatorio, elegir 1)** | SELECT (DISTINCT) secuencia de ?variable<br>SELECT (DISTINCT) *<br>DESCRIBE secuencia de ?variable o <URI><br>DESCRIBE *<br>CONSTRUCT { patrón de grafo}<br>ASK |
| **Especificar conjunto de datos RDF (opcional)** | Añadir grafo por defecto:<br>FROM <URI> (pueden haber más de uno)<br>Añadir grafo con nombre:<br>FROM NAMED <URI> (pueden haber más de uno) |
| **Graph Pattern (optional, obligatorio para ASK)** | WHERE { patrón de grafo z} |
| **Ordenar resultados de la consulta (opcional)** | ORDER BY ... |
| **Seleccionar resultados de la consulta (opcional)** | LIMIT n, OFFSET m |

**Figura 2.7**: Sintaxis básica de SPARQL

Si se realiza una consulta SPARQL sobre un punto de acceso, seguramente se podrá omitir la sección FROM. Debido a que el grafo principal viene dado implícitamente en la mayoría de los puntos de acceso SPARQL. Por lo que no es necesario especificarlo.

#### 2.2.4.3. Consultas SPARQL

A continuación se van a mostrar unos ejemplos de consultas básicas realizadas sobre el punto de acceso http://enipedia.tudelft.nl/wiki/Special:SparqlExtension.

```
BASE <http://enipedia.tudelft.nl/wiki/>
PREFIX article: <http://enipedia.tudelft.nl/wiki/>
PREFIX a: <http://enipedia.tudelft.nl/wiki/>
PREFIX property: <http://enipedia.tudelft.nl/wiki/Property:>
PREFIX prop: <http://enipedia.tudelft.nl/wiki/Property:>
PREFIX category: <http://enipedia.tudelft.nl/wiki/Category:>
PREFIX cat: <http://enipedia.tudelft.nl/wiki/Category:>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX fn: <http://www.w3.org/2005/xpath-functions#>
PREFIX afn: <http://jena.hpl.hp.com/ARQ/function#>
SELECT ?Name ?Point ?Generation_capacity WHERE {
        ?powerPlant prop:Country a:Spain .
        ?powerPlant rdfs:label ?Name .
        ?powerPlant prop:Point ?Point .
        ?powerPlant prop:Generation_capacity_electrical_MW ?Generation_capacity .
}
```

**Figura 2.8**: Consulta SPARQL 1

Como se puede ver en la figura 2.8, las variables empiezan con "?" y luego de la sentencia *WHERE* viene una lista de patrones de grafos. Formados por sujeto, predicado y objeto, que finalizan con un punto. Y que los posibles resultados deben cumplir. Así, en la consulta de la figura 2.8 se buscan resultados cuyo sujeto desconocido tenga *Country* entre sus propiedades. Y cuyo valor sea *España*. Además, se quiere recuperar el valor de sus puntos de localización, nombres y capacidad de generación. Por lo tanto el sujeto *?powerPlant* deberá tener dichos predicados. Finalmente, solo mostramos un subconjunto

de las variables vinculadas con la concordancia de los patrones de búsqueda.

Como los patrones de la consulta de la figura 2.8 comparten sujeto, es posible omitirlo y separar cada patrón con el símbolo punto y coma. Tal y como se muestra en la figura 2.9

```
SELECT ?Name ?Point ?Generation_capacity WHERE {
?powerPlant prop:Country a:Spain ;
            rdfs:label ?Name ;
            prop:Point ?Point ;
            prop:Generation_capacity_electrical_MW ?Generation_capacity .
}
```

**Figura 2.9**: Consulta SPARQL 1 con sintaxis alternativa

Independientemente de la sintaxis utilizada, con las consultas anteriores se obtienen los resultados que se muestra en la figura 2.10:

| Name | Point | Generation_capacity |
|---|---|---|
| "As Pontes Powerplant" | "43.444858828052,-7.8626203528256" | 1400.0 |
| "Compostilla Powerplant" | "42.62746,-6.56333" | 1341.0 |
| "Litoral De Almeria Powerplant" | "36.97827,-1.906643" | 1168 |
| "Teruel Powerplant" | "40.997778,-0.382778" | 1101 |
| "Abono Powerplant" | "43.5528,-5.72276" | 921.7 |
| "La Robla Fenosa Powerplant" | "42.8,-5.616667" | 620.0 |
| "Narcea Powerplant" | "43.337651,-6.414516" | 586.0 |
| "Algeciras Powerplant" | "36.210667,-5.38415" | 800.0 |

**Figura 2.10**: Parte de los resultados consulta SPARQL 1

También es posible añadir restricciones en la sentencia *WHERE* de la consulta. En la consulta de la figura 2.8, como para el predicado *Country* solo se quería que el valor fuera *Spain*, se indicó en la misma tripleta (?powerPlant prop:Country a:Spain). Sin embargo, si también se quisieran resultados de *Portugal*, se debería de usar la restricción *FILTER*.

En la figura 2.11 se muestra una consulta para ver información de las centrales eléctricas de los países España y Portugal, cuyo tipo de combustible sea el carbón y el agua.

```
SELECT ?Name, ?Country, ?Fuel_type WHERE{
    ?powerPlant prop:Country ?Country .
    FILTER (?Country IN (a:Portugal,a:Spain)) .
    ?powerPlant prop:Fuel_type ?Fuel_type .
    FILTER (?Fuel_type IN (a:Coal,a:Hydro)) .
    ?powerPlant rdfs:label ?Name .
}
```

**Figura 2.11**: Consulta SPARQL 2

### 2.2.5. Datos abiertos enlazados

De acuerdo con los estándares de la web semántica:

- Los datos pueden ser enlazados, pero no estar abiertos.
- Los datos pueden ser abiertos, pero no estar enlazados.
- Los datos pueden ser enlazados y abiertos.

Esta tercera opción persigue que determinados datos estén disponibles de forma libre a todo el mundo [38] [8]. Sin restricciones de copyright, patentes u otros mecanismos de control. La idea básica es publicar datos enlazados en un formato lo más abierto y auto-descriptivo posible. Así, se permite el acceso a sujetos distintos de sus creadores. Y se fomenta la reutilización de los datos para nuevos análisis y desarrollo de aplicaciones.

El único requisito de los datos abiertos enlazados es que deben compartirse de la misma manera en que aparecen en el manual de open data[2]. Además, según la Open Knowledge Definition[3], la forma de distribución de los datos abiertos enlazados tiene que satisfacer principalmente las condiciones de libertad de acceso, de redistribución y reutilización. Garantizando la ausencia de restricciones tecnológicas y la posibilidad de utilización de los datos por parte de cualquier persona.

El valor de la disponibilidad pública de estos datos abiertos enlazados está en las posibilidades de su utilización. Lo que tiene un verdadero valor es lo que se puede desarrollar a partir de los estos datos enlazados. Gracias al hecho de estar disponibles.

Y como ejemplos de fuentes de datos abiertos enlazados donde se pueden encontrar puntos de acceso, están la biblioteca nacional española[4], DBpedia[5] [2] y por su puesto Enipedia[6].

## 2.3. Data warehouse

El aumento exponencial de datos utilizados por las empresas hizo necesaria la llegada de los data warehouse. Utilizados para extraer y combinar datos útil de diferentes fuentes heterogéneas. Y así proporcionar información consistente y precisa, necesaria para los sistemas de soporte de decisiones [35].

### 2.3.1. Definición de data warehouse

Un data warehouse [11] [27] es la parte de un entorno global, vista como un repositorio central, que se encarga de integrar y depurar datos de una o más fuentes de datos distintas. Con el objetivo de procesar dichos datos y convertirlos en información lista para el análisis (ver figura 2.12).

[2] `http://opendatahandbook.org/es/`
[3] `http://opendefinition.org/od/index.html`
[4] `http://datos.bne.es/sparql`
[5] `http://dbpedia.org/sparql`
[6] `http://enipedia.tudelft.nl/wiki/Special:SparqlExtension`

Según William H. Inmon [26] un data warehouse tiene las siguientes características:

- Orientado a un tema: sólo los datos necesarios para el proceso de generación del conocimiento del negocio se integran desde el entorno operacional. Los datos se organizan por temas para facilitar su acceso y entendimiento por parte de los usuarios finales.
- Integración: los data warehouse extraen datos de múltiples fuentes, incluso de sistemas externos. Y por lo tanto, tiene que proporcionar una vista unificada de todos esos datos. Con lo que las inconsistencias entre los datos deben de eliminarse.
- No volátil: la información de un data warehouse existe para ser leída. Si se lleva a cabo una actualización, al data warehouse se incorporarán los últimos valores que tomen las distintas variables contenidas en él. Sin ningún tipo de acción sobre la información que ya existía.
- Variable en el tiempo: un data warehouse contiene la historia de lo ocurrido con la información contenida. Permitiendo de esta forma comparaciones.

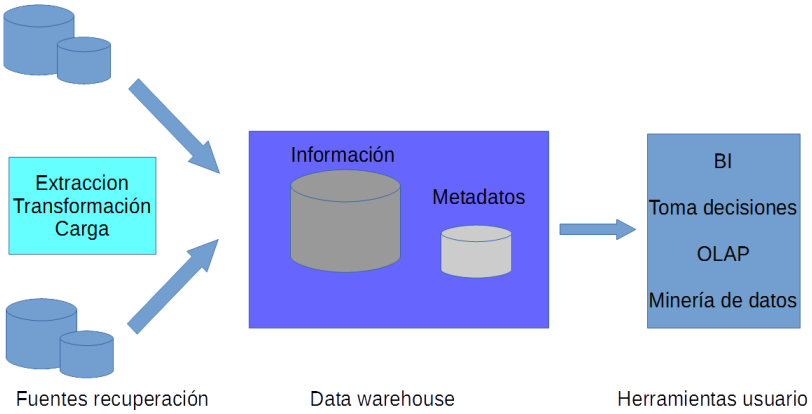



**Figura 2.12**: Visión general de un data warehouse

### 2.3.2. Data warehouse y sistemas operacionales

En un data warehouse se almacena toda la información de interés que luego se quiere analizar. Mientras que en una base de datos operacional se almacenan todas las transacciones, tanto datos útiles como no útiles. En la figura 2.3.2 se indican algunas de las diferencias más importantes entre ambos sistemas:

| Característica | Sistema operacional | Data warehouse |
|---|---|---|
| Objetivo | Almacenar datos de las operaciones del día a día. | Crear información útil para el análisi y toma de decisiones |
| Proceso | De transacciones. Repetitivo y conocido. | De consultas masivas. Puntual y no conocido. |
| Actividad | Consulta y escritura de los datos almacenados | Predomina la consulta |
| Rendimiento | Importancia del tiempo de respuesta de la transacción instantánea | Importancia de la respuesta masiva |
| Explotación | Explotación de la información relacionada con la operativa de cada aplicación | Explotación de toda la información interna y externa relacionada con el negocio/tema |
| Volatilidad | Datos almacenados actualizadables | Carga de nuevos datos, pero no actualización |
| Usuarios destino | Usuarios de perfiles medios o bajos | Usuarios de perfiles altos |
| Organización | Estructura normalmente relacional | Visión multidimensional |
| Granularidad | Datos generales desagregados, al detalle | Datos en distintos niveles de detalle y agregación |
| Número usuarios | Destinado a miles de usuarios. | Destinado a cientos de usuarios |
| Consultas | Consultas OLTP | Consultas OLAP |

**Figura 2.13**: características data warehouse y sistema operacional

### 2.3.3. Modelo multidimensional

La tecnología tras los data warehouse impone un procesamiento y pensamiento distinto, debido a su orientación analítica. La cual se sustenta en un modelado de bases de datos propio, conocido como modelo multidimensional [17] [28] [18]. Que consiste en una técnica de diseño lógico, que tiene como objetivo presentar y almacenar los datos dentro de un marco de trabajo estándar e intuitivo. Permitiendo el acceso a grandes cantidades de información con un alto rendimiento.

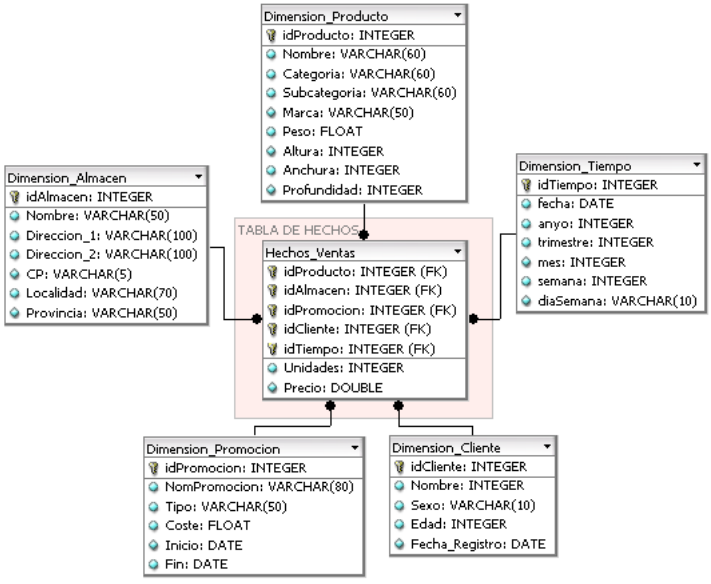



**Figura 2.14**: Ejemplo de esquema dimensional (estrella) de un DW

En una base de datos multidimensional, los datos se organizan alrededor de distintos modelos. Cada modelo se llama esquema en estrella. Y en la figura 2.14 se ve un ejemplo de los principales componentes. Se puede apreciar como un esquema en estrella está compuesto por una tabla llamada tabla de hechos [28]. Y por un conjunto de tablas más pequeñas llamadas tablas de dimensiones. Los elementos [28] del modelo multidimensional se pueden definir de la siguiente forma:

- Hechos: Es una colección de piezas de datos de contexto. Cada hecho representa una parte del negocio, una transacción o un evento. Por lo tanto, un hecho es un concepto de interés primario para el proceso de toma de decisiones. En la tabla de hechos es donde se almacenan las mediciones numéricas del negocio. Como podrían ser las unidades vendidas o el precio.
- Dimensiones: Es una colección de miembros, unidades o individuos del mismo tipo. Se pueden ver como perspectivas sobre la información que puede ser analizada.
- Jerarquías de dimensiones: Las dimensiones se pueden relacionar en jerarquías o niveles. Una jerarquía es un conjunto de miembros de una dimensión. Los cuales se definen por su posición relativa con respecto a los otros miembros de la misma dimensión. Formando en su totalidad una estructura de árbol. Por lo tanto, si se parte de la raíz del árbol, los miembros son progresivamente más detallados hasta llegar a las hojas, donde se obtiene el mayor nivel de detalle. Por ejemplo, la dimensión *localización* puede tener los niveles *ciudad*, *estado*, etc.
- Medidas: Son atributos numéricos que describen un hecho. Representan el comportamiento del negocio relativo a una dimensión. Por ejemplo, un analista quiere ver las ventas (medida) por lugar, tiempo, producto y localización (dimensiones). Los niveles de las dimensiones denotan la granularidad de observación de la medida respecto a una dimensión.

Desde el punto de vista relacional, se puede decir que el modelo multidimensional consiste en una tabla de hechos normalizada con tablas de dimensionales desnormalizadas.

El concepto de dimensión condujo a la metáfora de los cubos [17] como estructura de datos para analizar los modelos multidimensionales. O hiper-cubos si existen más de tres dimensiones. Por lo que se puede derivar los esquemas en estrella a cubos. Donde estos cubos no son más que el conjunto formado por todas dimensiones que pertenecen a un hecho concreto. Y donde las celdas están compuestas por las medidas que describen los hechos (ver figura 2.15).

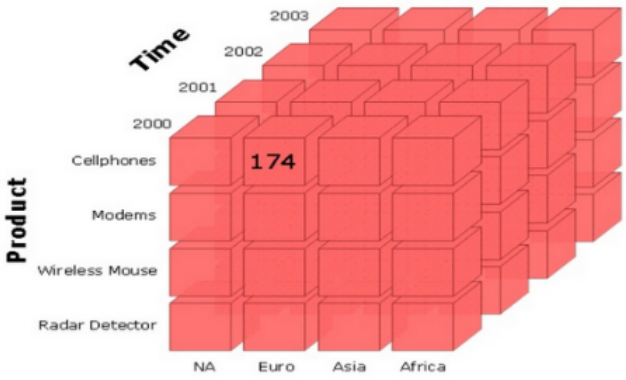



**Figura 2.15**: Ejemplo cubo multidimensional

Como se puede ver en la figura 2.15, la medida *cantidad* puede ser analizada teniendo en cuenta la dimensión *Producto*, *Tiempo* y *Localización*. Aunque también puede ser analizada, en su conjunto, por todas las dimensiones a la vez o cualquier combinación de ellas. Dando al analista una amplia gama de posibilidades. De las cuales puede tomar ventaja para la toma de decisiones.

Disponer los datos en cubos evita una limitación de las bases de datos relacionales, que no son muy adecuadas para el análisis instantáneo de grandes cantidades de datos. Por lo que para acceder a los datos multidimensionales sólo es necesario indexarlos a partir de los valores de las dimensiones.

Sin embargo, el hecho de almacenar físicamente los datos en modelos multidimensionales tiene sus inconvenientes. Ya que una vez poblada la base de datos, ésta no puede recibir cambios en su estructura. Para ello sería necesario rediseñar los modelos.

### 2.3.4. OLAP

Según la definición que le dio Codd [12] en 1993, OLAP (On-Line Analytical Processing) es un tipo de procesamiento de datos que se caracteriza, entre otras cosas, por permitir el análisis multidimensional. OLAP podría ser la principal forma de explotar la información en un data warehouse. Ya que ofrece a los usuarios finales, la oportunidad de analizar y explorar datos sobre una base de datos multidimensional [17] [28] [18] de forma interactiva. Y todo ello, con una gran rapidez de respuesta. Hecho que convierte a OLAP en una de las herramientas más utilizadas en el campo de las soluciones Business Intelligence [4]. Ya que el aspecto multidimensional permite analizar los datos de negocio en su relación con otros datos empresariales. O dicho de otra forma, desde diferentes perspectivas. Hecho crucial para la toma de decisiones.

Esta tecnología es independiente de la implementación y permite el empleo de cualquier base de datos:

- La arquitectura MOLAP (Multidimensional Online Analytical Processing) utiliza bases de datos multidimensionales para proporcionar el análisis.
- La arquitectura ROLAP (Relational Online Analytical Processing) utiliza bases de datos relacionales.

### 2.3.5. Operaciones OLAP

La información de un cubo multidimensional es difícil de manejar debido a su cantidad. Y por ello existen diferentes operaciones para reducir el volumen de información. Hay cuatro tipos básicos de operaciones [27] [37] [36] en OLAP para el análisis de datos multidimensionales. Las operaciones “drill down” y “roll up”, que definen el nivel de granularidad con la que se quiere analizar los datos. Y las operaciones “slice” y “dice”, que permiten navegar entre las dimensiones y aplicar filtros o eliminarlas.

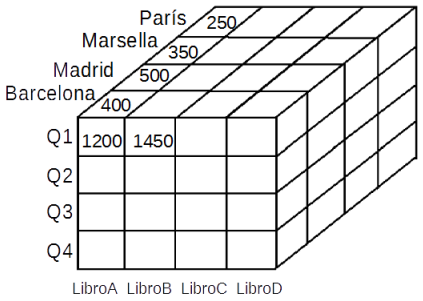
### 2.3.5.1. Roll-up

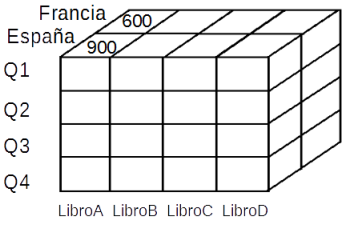
Esta operación proporciona agregación en un cubo multidimensional. Para ello, sube un nivel en la jerarquía de una dimensión. Ver figura 2.16.

| | | Unidades | | | |
|---|---|---|---|---|---|
| | **Mes** | **Norte** | **Sur** | **Este** | **Oeste** |
| **Tiempo** | Enero 2012 | 10000 | 6000 | 9000 | 12000 |
| | Febrero 2012 | 9000 | 8000 | 12000 | 13000 |
| | Marzo 2012 | 7000 | 9000 | 10000 | 15000 |
| | Abril 2012 | 6000 | 9000 | 8000 | 10000 |
| | Mayo 2012 | 8000 | 10000 | 9000 | 8000 |
| | Junio 2012 | 9000 | 12000 | 13000 | 9000 |

Roll-up en la dimensión Tiempo.
Subimos en la jerarquía de mes a cuarto fiscal.

| | | Unidades | | | |
|---|---|---|---|---|---|
| | **Cuarto** | **Norte** | **Sur** | **Este** | **Oeste** |
| **Tiempo** | Q1 2012 | 26000 | 23000 | 31000 | 40000 |
| | Q2 2012 | 23000 | 31000 | 30000 | 27000 |



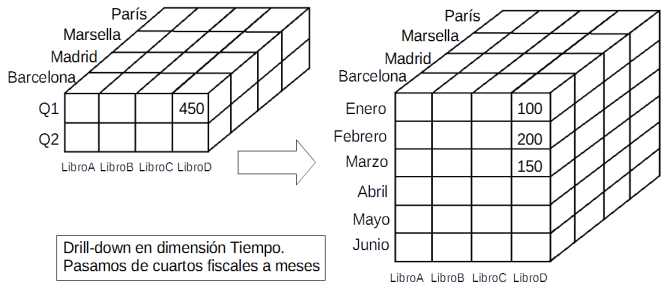

Roll-up en dimensión Localizacion.
Pasamos de ciudades a países

**Figura 2.16**: Ejemplos operaciones *roll-up*

Si se compara esta operación con una consulta de una base de datos relacional, se puede decir que se trata de hacer una agrupación y una operación de agregado.

### 2.3.5.2. Drill-down

La operación drill-down es la complementaria de la operación roll-up. Por lo tanto, como se observa en la figura 2.17, se reduce la agregación de los datos.




Drill-down en dimensión Tiempo.
Pasamos de cuartos fiscales a meses

**Figura 2.17**: Ejemplo operación *drill-down*

### 2.3.5.3. Slice

Consiste en seleccionar un elemento de una dimensión y realizar la proyección sobre el resto de dimensiones. Por lo que se reduce la dimensionalidad del cubo original, tal y como se muestra en la figura 2.18. Equivale a la clausula WHERE de SQL.

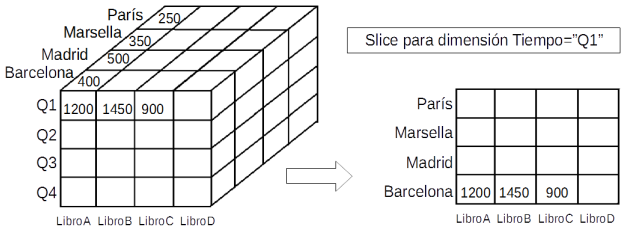



**Figura 2.18**: Ejemplo operación *slice*

#### 2.3.5.4. Dice

Consiste en definir un sub-cubo seleccionando dos o más dimensiones del cubo original. La figura 2.19 muestra un ejemplo.

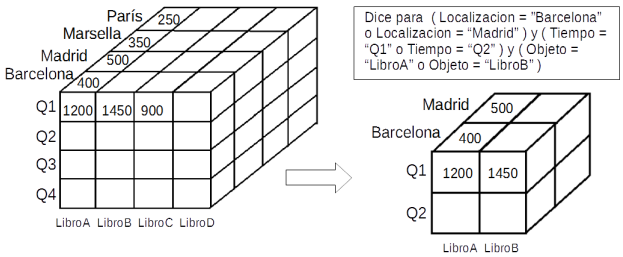



**Figura 2.19**: Ejemplo operación *dice* donde intervienen 3 dimensiones

# Capítulo 3

# Estudio de antecedentes

Para que un usuario pueda realizar consultas sobre los datos abiertos enlazados de una fuente de conocimiento, es necesario que conozca de antemano las relaciones que existen entre esos datos.

La búsqueda de las relaciones que hay entre los datos contenidos en un dataset (ver apartado 2.2.3), es algo esencial para este proyecto. Ya que si no se dispone de ellas, no se posible crear un mecanismo visual para facilitar la creación de consultas. Por lo tanto, es importante estudiar los antecedentes existentes.

Así que en este capítulo, se van a mostrar las herramientas Relfinder y gFacet. En las cuales se combinan visualización e interactividad para mostrar "información" tras los datos enlazados. Además, también se hablará de la herramienta Sgvizler. Que tiene la finalidad de representar gráficamente los resultados obtenidos por una consulta a los datos enlazados.

## 3.1. Relfinder

Relfinder[1] [22] es una herramienta de visualización de grafos que permite representar automáticamente todas las relaciones existentes entre varios términos examinados. Está basado en el framework open source Adobe Flex, por lo que funciona en cualquier navegador que tenga instalado Adobe Flash Player. Además, funciona con cualquier dataset de RDFs estandarizado para proporcionar un punto de acceso mediante consultas SPARQL.

En Relfinder, los términos de búsqueda que son introducidos por el usuario en los campos de entrada, son asociados con objetos únicos de la base de conocimiento utilizada (ver figura 3.1). Estos objetos constituyen los nodos principales del grafo que se pretende extraer. El algoritmo encontrará y unirá todos los nodos y relaciones que se encuentre por el camino. Obteniendo de esta forma, todos los caminos de acceso a los términos introducidos. Además, en el lateral derecho se obtendrá más información sobre cada objeto. Finalmente, pueden aplicarse filtros para aumentar o reducir el número de relaciones que se muestran en el grafo y centrarse en cierta aspectos de interés.

[1] `http://www.visualdataweb.org/relfinder/relfinder.php`

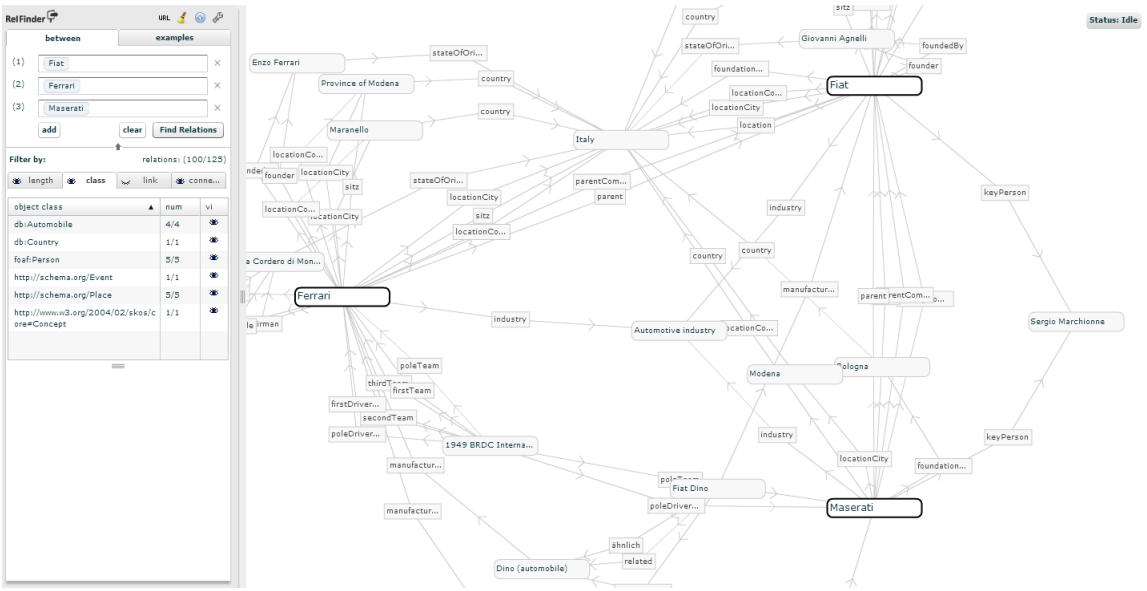



**Figura 3.1**: Ejemplo Relfinder

## 3.2. gFacet

gFacet[2] [23] es una aplicación basada en categorías que permite al usuario explorar, de forma manual, las relaciones de los datos RDF de las distintas fuentes de conocimiento. Combinando una visualización en modo de grafo con las técnicas de filtrado de categorías. Donde cada nodo del grafo representará una categoría.


**Figura 3.2**: Ejemplo gFacet

Inicialmente, se elegirá un termino en su buscador, con lo se obtendrá un lista de posibles clases relacionadas con dicho termino. La clase que se elija aparecerá representada como un nodo del grafo. Junto con su listado de instancias. También aparecerá un menú

[2] `http://www.visualdataweb.org/gfacet/gfacet.php`

desplegable con las posibles relaciones con ese nodo. Y cuando un usuario seleccione una relación, aparecerá un nuevo nodo. Conectado al nodo anterior por la etiqueta de la relación escogida.

Como se puede observar en la figura 3.2, por cada nodo se podrán elegir múltiples relaciones. Con lo que se irán creando múltiples nodos relacionados. Formando de esta forma un grafo.

Además, las instancias de cada nodo pueden actuar como un filtro para las instancias del nodo conectado.

## 3.3. Sgvizler

Sgvizler[3] es una librería JavaScript que renderiza los resultados de las consultas *SELECT* de SPARQL a gráficas o elementos HTML [34]. Gracias a esta librería es posible visualizar los datos de una consulta SPARQL en multitud de gráficas o incluso mapas.

Básicamente, como datos de entrada requiere la dirección del punto de acceso contra el que se lanzará la consulta, la correspondiente consulta SPARQL y el tipo de visualización junto a sus opciones. Además, el punto de acceso debe de ser capaz de devolver los resultados de la petición en formato XML (eXtensible Markup Language) o JSON (JavaScript Object Notation).

A continuación se muestra un ejemplo de como se visualizaría la siguiente consulta en Sgvizler (se omiten los prefijos de la consulta):

```
SELECT ?Powerplant, ?Primary_fuel_type, ?Country
WHERE{
        ?Powerplant prop:Primary_fuel_type ?Primary_fuel_type
        .FILTER ( ?Primary_fuel_type IN (a:Coal))
        .?Powerplant prop:Country ?Country
        .FILTER ( ?Country IN (a:Spain))
}
```

**Figura 3.3**: Ejemplo de consulta SPARQL para visualizar con Sgvizler

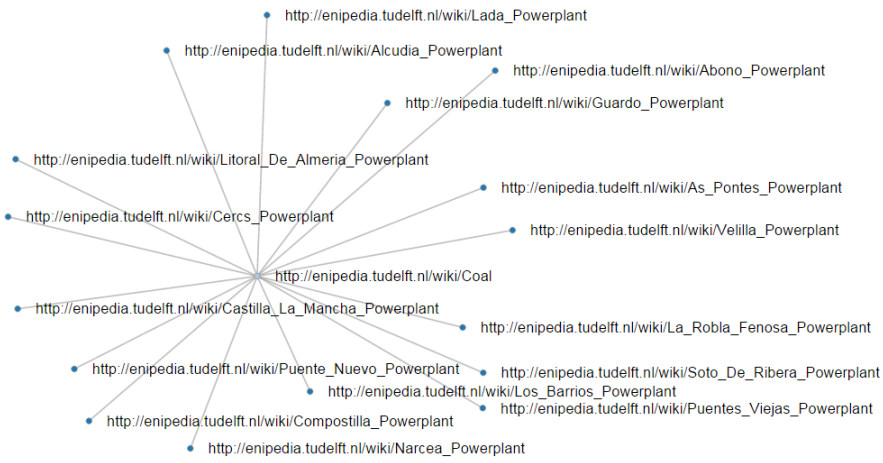



**Figura 3.4**: Ejemplo gráfico D3ForceGraph en Sgvizler

[3] `http://dev.data2000.no/sgvizler/`

# Capítulo 4

# Propuesta

SPARQL, desde su lanzamiento oficial en 2008 [14], se ha convertido en el principal lenguaje de consulta para la web semántica. Es el estándar para la consulta de almacenes de tripletas semánticas. Y es una tecnología clave para la "apertura" de los datos enlazados. Ya que los almacenes de datos abiertos enlazados, facilitan puntos de acceso públicos que requieren de este lenguaje para ser consultados (ver sección 2.2.5).

Sin embargo, escribir consultas SPARQL no es una tarea trivial [31]. La sintaxis es propensa a cometer errores a la hora de formular consultas. Y requiere cierto conocimiento de como funciona RDF. Lo que implica que los usuarios medios puedan tener dificultades para utilizarlo. Impidiéndoles sacar provecho de las fuentes de conocimiento que ofrecen los datos abiertos enlazados.

La aplicación web desarrollada, Linked Data Visual Query Builder, es una herramienta que facilita el acceso y la consulta de los datos albergados en Enipedia[1] sobre su punto de acceso[2]. Presentando un método visual para hacer consultas. Con lo que el usuario simplemente interactuará con la interfaz de la aplicación. Y será esta la que internamente se comunique con el punto de acceso SPARQL. Creando y enviando la consulta.

## 4.1. Exploración y búsqueda de relaciones

Para poder hacer consultas sobre una fuente de datos enlazados, es necesario tener un conocimiento de como están relacionados los datos. Ya que sin dicho conocimiento, será imposible introducir patrones de grafos necesarios para efectuar las consultas SPARQL.

El objetivo del proyecto, es crear una interfaz visual que evite el tener que introducir manualmente consultas SPARQL (ver sección 2.2.4). Y para ello, es necesario tener los patrones que relacionan los diferentes conceptos del dataset. Ya que si se presentan al usuario los diferentes conceptos, junto a sus áreas de interés analizables (relaciones), este podrá elegir que es lo que quiere consultar. Y la aplicación se encargará internamente de realizar dicha consulta.

En la sección 3 se mostraron dos ejemplos de herramientas para descubrir relaciones entre conceptos. Dichas herramientas dependen de la interacción humana para ir descubriendo relaciones. Sin embargo, para este proyecto es necesario un método automático para

[1] `http://enipedia.tudelft.nl/wiki/Main_Page`
[2] `http://enipedia.tudelft.nl/wiki/Special:SparqlExtension`

realizar una exploración previa del dataset y obtener los patrones.

Esta exploración previa ha sido proporcionada por Victoria Nebot, basándose en [30] [29]. Para dicha exploración, se ha partido de las tripletas RDF sujeto-predicado-objeto (s,p,o) del dataset de Enipedia. Y mediante un sistema de razonamiento, capaz de inferir las clases *s_concept* y *o_concept* a partir de los sujetos *s* y objetos *o* de las tripletas iniciales, se han generado las tripletas "tipificadas" (s_concept, p, o_concept). Con el propósito de averiguar patrones o relaciones que hay entre los datos de las tripletas almacenadas en el dataset.

## 4.2. Resultados de la exploración previa

Como resultado de la exploración previa del dataset, se obtienen tres ficheros que contendrán la información necesaria para construir la base de la aplicación.

- Fichero *enipedia_catalogue_main_filtered*. Donde cada linea presenta el siguiente formato: (*s_concept o_concept num_triples num_distinct_predicates*). Siendo *num_triples* el número de tripletas que tienen el mismo s_concept y o_concept. Y como su nombre indica, *num_distinct_predictives* es el número de predicados (p) distintos que hay para el mismo *s_concept* y *o_concept*.
- Fichero *enipedia_catalogue_sec_filtered*. En él se cuentan el número de tripletas tipificadas que cumplen el mismo patrón *s_concept*, *p* y *o_concept*. El fichero tiene el formato: *(s_concept o_concept p num_triples*).
- Fichero *types_statistics*. Simplemente indica el número de instancias que tiene cada concepto.

Es importante mencionar, que a la hora de extraer las tripletas tipificada, se han eliminado los patrones (*s_concept, p, o_concept*) con una cobertura menor del 10 % con respecto a su *s_concept*. Esto quiere decir que se ha eliminado cualquier patrón cuyo número de tripletas es menor del 10 % de instancias que tiene su *s_concept*. Ya que se ha considerado poco relevante para el análisis.

Con estas estadísticas pre-calculadas ya se puede crear la base de datos central de la aplicación. Que a partir de ahora se llamará catálogo intermedio.

## 4.3. Creación del catálogo intermedio

Antes de la creación del catálogo intermedio hace falta pensar en lo que va a ofrecer la interfaz al usuario. Y como se van a presentar las relaciones obtenidas en la exploración previa. Con lo que la mejor alternativa es utilizar la organización temática de los data warehouse, que se explicó en la sección 2.3.1. Donde por cada concepto se ofrecen un conjunto de áreas a analizar. Que pueden ser vistas como diferentes perspectivas.

Como se explicó en la sección 2.3.3, un data warehouse tiene un almacenamiento interno basado en modelos multidimensionales. Donde cada modelo representa un concepto o entidad central de interés para el análisis. Así pues, se han descompuesto los patrones obtenidos en la exploración previa en diferentes elementos. Que pasan a llamarse entidades centrales, medidas y dimensiones. Donde las medidas son cada una de las propiedades numéricas que describen algo de la entidad central. Y las dimensiones son las relaciones

de la entidad central con otras entidades centrales. O expresado de otra forma, la relación del *s_concept* con los *o_concept*. Finalmente, todos estos elementos extraídos serán almacenados en la base de datos central. O también llamada catálogo intermedio.

Los dos ficheros más importantes son *types_statistics* y *enipedia_catalogue_sec_filtered*.

- Del fichero *types_statistics* se ha extraído el nombre de las entidades centrales (*s_concept*). Además del número de instancias de cada una de ellas. Como se observa en la figura 4.1, el nombre de cada entidad central está tras la cadena "Category:".

```
http://enipedia.tudelft.nl/wiki/Category:Refinery 6
http://enipedia.tudelft.nl/wiki/Category:Country_EU_ETS_Profile 27
http://enipedia.tudelft.nl/wiki/Category:Powerplant 74625
http://enipedia.tudelft.nl/wiki/Category:NaturalGasModel 4
```

**Figura 4.1**: Ejemplo de entidades centrales del fichero types_statistics

- A continuación es necesario procesar el fichero *enipedia_catalogue_sec_filtered*. Donde los elementos aparecen con el formato *(s_concept o_concept p num_triples)*. Para extraer las medidas de cada entidad central, se buscan los patrones de tripletas "tipificadas" cuyos objetos (o_concept) son un tipo de dato numérico determinado. Y no otros conceptos. Como muestra el ejemplo de la figura 4.2, el nombre de la medida se correspondería con el texto tras la cadena "Property:". Y el tipo de la medida se corresponde con el texto tras la cadena "XMLSchema#".

```
http://enipedia.tudelft.nl/wiki/Category:Powerplant http://www.w3.org/2001/XMLSchema#decimal
        http://enipedia.tudelft.nl/wiki/Property:Latitude 74617
http://enipedia.tudelft.nl/wiki/Category:Powerplant http://www.w3.org/2001/XMLSchema#decimal
        http://enipedia.tudelft.nl/wiki/Property:Longitude 74619
```

**Figura 4.2**: Patrones de medidas en el fichero enipedia_catalogue_sec_filtered

- Finalmente, en el fichero *enipedia_catalogue_sec_filtered* también se encuentran las dimensiones. Que se corresponden con los patrones de tripletas cuyos objetos (o_concept) son iguales a otros conceptos *s_concept*. Y esos concepto son los nombres de las dimensiones.

```
http://enipedia.tudelft.nl/wiki/Category:Powerplant http://enipedia.tudelft.nl/wiki/Category:EUMember
        http://enipedia.tudelft.nl/wiki/Property:Country 26127
http://enipedia.tudelft.nl/wiki/Category:Powerplant http://enipedia.tudelft.nl/wiki/Category:Europe
        http://enipedia.tudelft.nl/wiki/Property:Country 31434
http://enipedia.tudelft.nl/wiki/Category:Powerplant http://enipedia.tudelft.nl/wiki/Category:Fuel
        http://enipedia.tudelft.nl/wiki/Property:Primary_fuel_type 17095
http://enipedia.tudelft.nl/wiki/Category:Powerplant http://enipedia.tudelft.nl/wiki/Category:Fuel
        http://enipedia.tudelft.nl/wiki/Property:Fuel_type 18045
```

**Figura 4.3**: Patrones de dimensiones en el fichero enipedia_catalogue_sec_filtered

Como se observa en el ejemplo de la figura 4.3, los dos últimos patrones tienen el mismo objeto pero distinto predicado. Esto tiene la siguiente interpretación: en las tripletas tipificadas que describen el concepto Powerplant, se están indicando dos propiedades distintas (Primary_fuel_type y Fuel_type). Cuyos valores pertenecen al conjunto de valores del tipo Fuel. Con lo que el objeto de los dos predicados es del mismo tipo. Y eso tiene que quedar reflejado en el catálogo intermedio. Para que posteriormente, en la interfaz, el usuario pueda escoger si quiere consultar respecto a una propiedad o respecto a otra.

## 4.4. Interfaz de consultas visuales

A continuación se va a explicar como interactúan cada uno de los elementos de los modelos multidimensionales con la interfaz de la aplicación.

### 4.4.1. Entidad central

Como se comentó en el anterior apartado, la aplicación se centra en los modelos multidimensionales creados a partir de los diferentes conceptos extraídos de Enipedia. Ofreciendo al usuario una visión de las medidas y dimensiones disponibles para analizar a cada entidad central.

Por lo tanto, el usuario iniciará la creación de consultas visuales eligiendo una entidad central. Y a continuación se mostrará una interfaz basada en tablas de dos dimensiones. En las cuales se mostrarán las medidas y dimensiones que puede elegir el usuario para formular consultas sobre la entidad central.

Seguidamente se va a mostrar un ejemplo del esquema en estrella de la entidad central *Powerplant*. En la figura 4.4 solo aparecen algunas de las rutas de la estrella. Por lo que no es la estrella completa. Ya que se omiten algunas medidas y dimensiones.

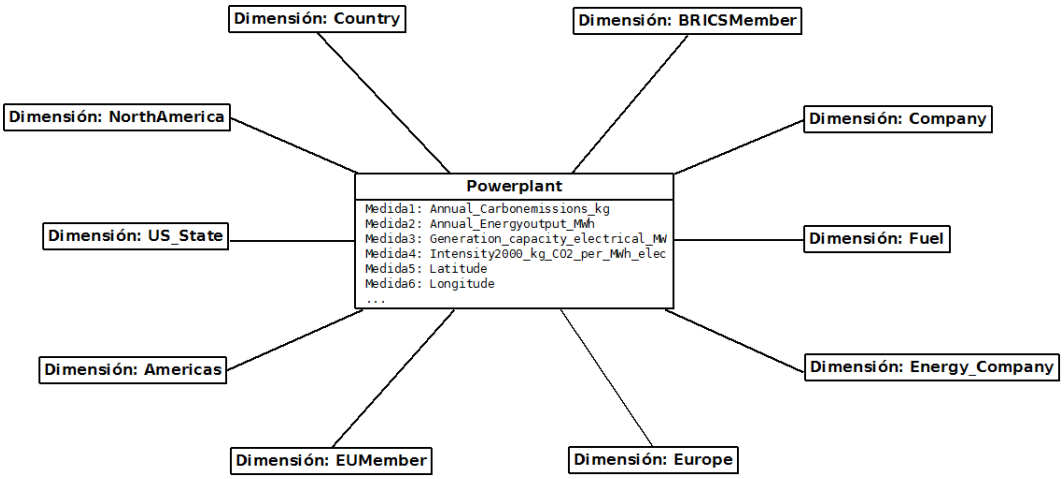



**Figura 4.4**: Esquema en estrella de Powerplant

### 4.4.2. Tabla de medidas

La medidas que puede elegir el usuario están disponibles en la primera tabla de la interfaz destinada a generar consultas (ver figura 4.5). Estas medidas son los datos que se quieren analizar. Y las dimensiones definen la organización de dichas medidas. La combinación de valores para cada dimensión define el espacio lógico donde los valores de las medidas pueden aparecer. Por lo tanto, es necesario elegir al menos una medida para obtener datos numéricos en la consulta. Y si el usuario no elije ninguna, para evitar que solo obtenga como resultado el nombre de los valores de las dimensiones, en la consulta SPARQL generada se introduce automáticamente *count(*)* como medida. Lo que hará que al menos se devuelva un dato numérico.

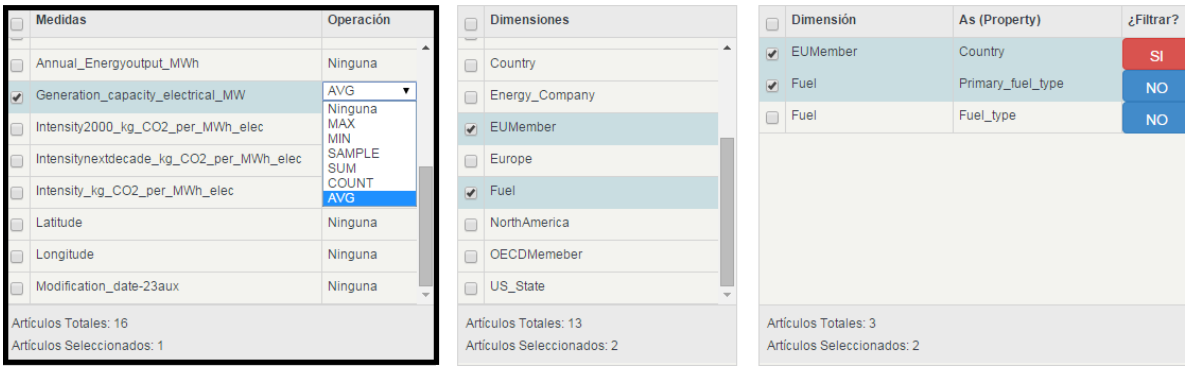



**Figura 4.5**: Interfaz de la tabla de medidas de la aplicación

Un problema encontrado tras la exploración del dataset de Enipedia, fue localizado en las estadísticas pre-calculadas del fichero enipedia_catalogue_sec_filtered (ver sección 4.3). Ya que había patrones correspondientes a medidas, cuyo objeto es un tipo numérico, refiriéndose al mismo predicado pero con diferente tipo de dato. Como se puede ver en la figura 4.6:

```
http://enipedia.tudelft.nl/wiki/Category:Powerplant http://www.w3.org/2001/XMLSchema#decimal
        http://enipedia.tudelft.nl/wiki/Property:Annual_Carbonemissions2000_kg 59180
http://enipedia.tudelft.nl/wiki/Category:Powerplant http://www.w3.org/2001/XMLSchema#double
        http://enipedia.tudelft.nl/wiki/Property:Annual_Carbonemissions2000_kg 15091
```

**Figura 4.6**: Patrones representando la misma medidas pero diferente tipo

Esto implica, que a la hora de crear el catálogo intermedio (ver sección 4.3) se están creado diferentes entradas para la misma medida. Pero con distintos tipos numéricos de datos. Esto no supone un problema grave a la hora de generar las consultas internas de SPARQL. Ya que al punto de acceso solo se le manda el nombre de la medida y no el tipo de dato numérico. Sin embargo, se optó por volver a generar los ficheros con las estadísticas pre-calculadas. Juntando en un mismo tipo de datos las medidas con el mismo nombre. Y así evitar incongruencias en el catálogo intermedio.

Finalmente, aparte de poder elegir entre la lista de medidas que cada entidad central ofrece, la tabla también permite escoger y aplicar funciones de agregado a dichos atributos. Las funciones de agregado de SPARQL se comportan igual que en SQL. Dependiendo de los grupos de dimensiones formados por la sentencia *GROUP BY*. Y obteniendo un resultado por grupo según la función de agregado utilizada sobre las medidas del grupo. La interfaz permite, como muestra la figura 4.5, elegir entre las siguientes funciones de agregado: MAX, MIN, SAMPLE, SUM, COUNT, AVG. Sin embargo, de momento no permite el uso de filtros *HAVING*. Eso es parte de una posible mejora futura.

Finalmente, hay que indicar que las medidas que se refieren al mantenimiento del "wiki" de Enipedia no son mostradas. Ya que no aportan datos que puedan interesar a usuario a la hora de analizar la entidad central.

### 4.4.3. Tabla dimensiones y propiedades

Las dimensiones son cada una de las áreas de interés a analizar. Como ya se comentó en la sección 2.3.3, son fundamentales en un modelo multidimensional. Ya que su función es describir las medidas que se elijan.

El usuario debe de elegir una dimensión de la tabla de dimensiones si desea analizar una medida desde una determinada perspectiva. Sin embargo, esto implicará interactuar con una otra tabla más. Por lo que primero, como se ve en la figura 4.7, es necesario elegir una dimensión. Y a continuación, en la tercera tabla deberá elegir una propiedad que relaciona la entidad central con la dimensión. De entre las diferentes propiedades asociadas. Ya que como se comentó en la sección 4.3, hay patrones de dimensiones con el mismo objeto pero con diferente predicado. Aunque lo normal es que solo aparezca una propiedad relacionada con cada dimensión.

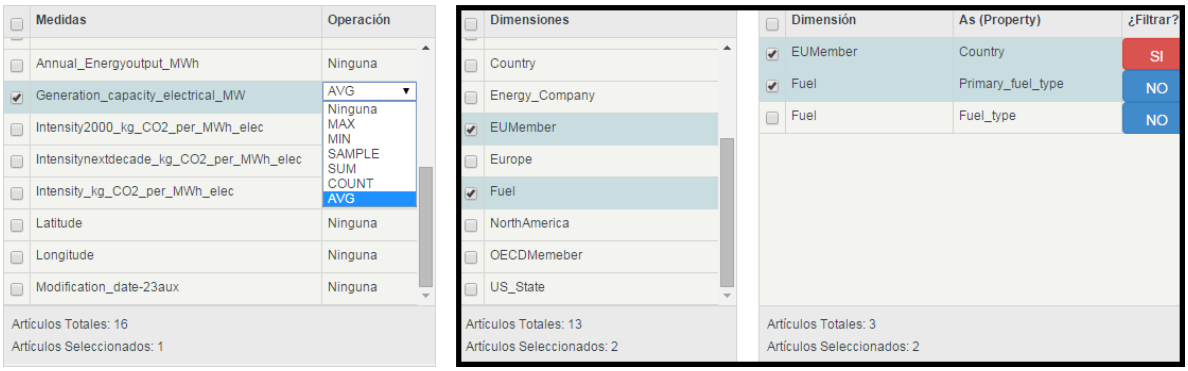



**Figura 4.7**: Interfaz de las tablas de medidas y propiedades de la aplicación

Así, en el ejemplo de la figura 4.7, el usuario elegiría *Primary_fuel_type* como propiedad relacionada con la dimensión *Fuel*. Lo que indicará que la consulta final debe realizarse desde a perspectiva de los resultados que toma la propiedad Primary_fuel_type.

Hay que recordar que para formular la consulta final es necesario que se elija al menos una propiedad. Y por lo tanto una dimensión. Ya que sin dimensiones no hay nada que analizar.

### 4.4.4. Filtros en la tabla de propiedades

El objetivo de la interfaz de consultas visuales es el de obtener un resultado multidimensional. Mediante la elección de medidas y propiedades que pueden tomar las dimensiones. Con lo que se podría decir que el resultado de la consulta final es un cubo multidimensional (ver sección 2.3.3).

El resultado de elegir todas las medidas y todas propiedades sería un hiper-cubo de tamaño enorme. Sin embargo, lo normal es que el usuario solo elija unas pocas medidas y propiedades a analizar.

Además, el usuario también puede aplicar filtros sobre los valores de las propiedades elegidas. De forma que la consulta final no se realizaría sobre todos los valores que toma la propiedad. Si no que solo sobre los valores elegidos. Para ello, se debe pulsar el botón que está al lado de cada propiedad seleccionada. De forma que el botón muestre el valor 'SI'. Lo que activará el botón inferior 'Mostrar valores a filtrar'. Encargado de mostrar los diferentes valores de las dimensiones con filtros elegidas.

El efecto que se consigue con filtrar las dimensiones es similar al que se consigue con las operaciones *slice* y *dice* de OLAP.

- Operación Slice. Ocurre si el usuario elije varias propiedades para la consulta y solo en una de ellas elije filtrar por un valor. Con lo que se forma una especie de rebanada

(slice) del cubo que se formaría si no eligiéramos el filtro.

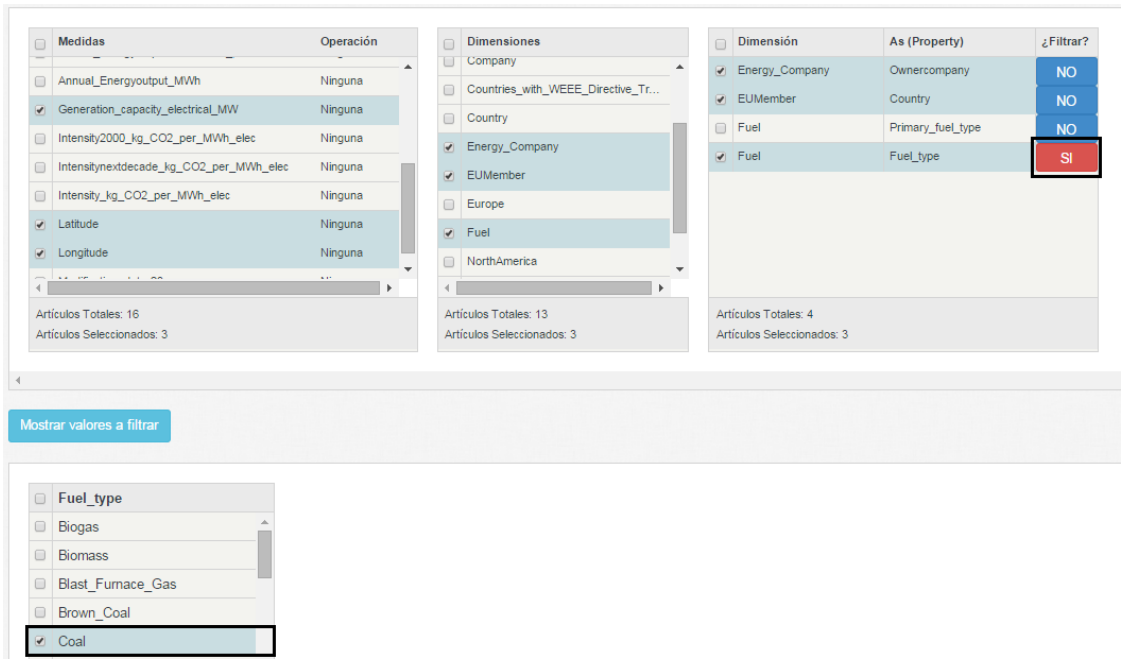



**Figura 4.8**: Filtro slice sobre entidad central Powerplant

- Operación Dice. Al seleccionar varios valores de varias propiedades se forma sub-cubo o dado (dice). Por lo tanto, si se seleccionan varias propiedades, y en cada una de ellas se aplica un filtro, el resultado será un sub-cubo del cubo que se obtendría si no se aplicasen filtros.

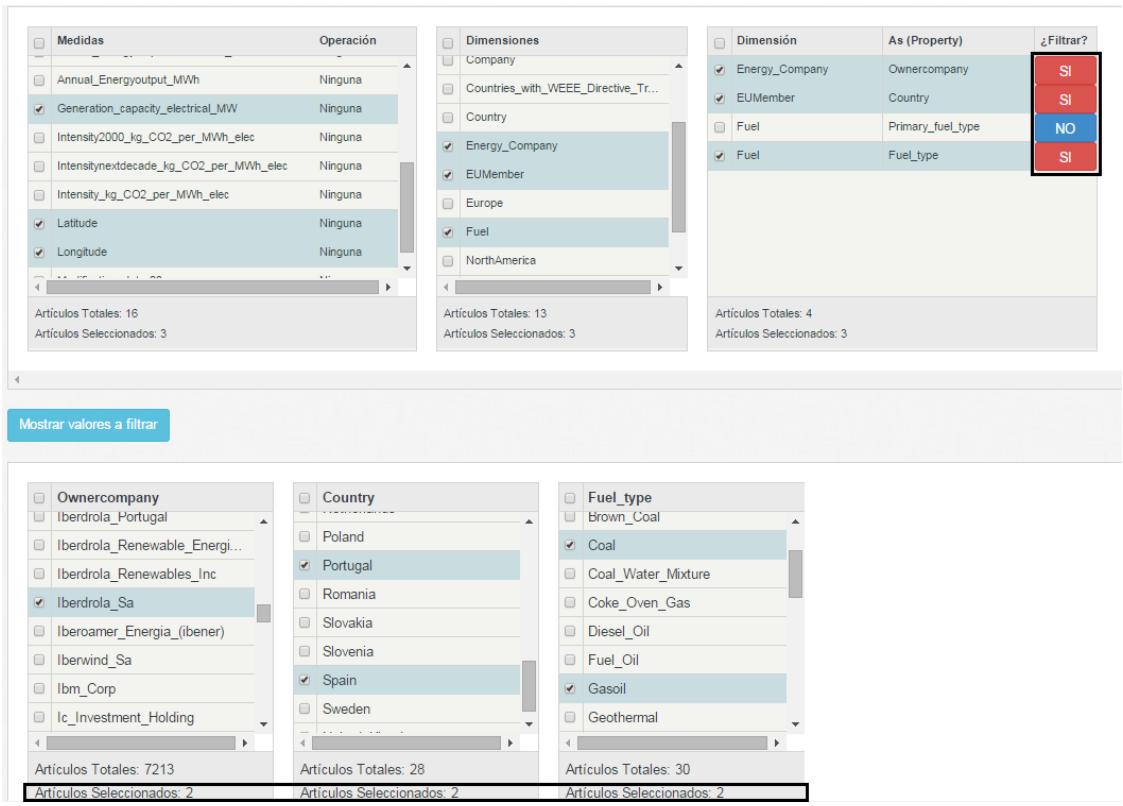



**Figura 4.9**: Filtro dice sobre entidad central Powerplant

Una vez se ha elegido aplicar un filtro a los valores de una propiedad, se tienen que mostrar los valores a filtrar de dicha propiedad. Y para ello, la aplicación internamente necesita efectuar una consulta SPARQL sobre el punto de acceso de Enipedia (ver sección 2.2.4). Para que muestre dichos valores. Una muestra de dichas consultas se puede ver el ejemplo de la figura 4.10. Donde se omiten la definición de los prefijos de los espacios de nombres.

```
SELECT DISTINCT ?y
WHERE{
        ?x prop:Country ?y .
        ?y a cat:EUMember
}
ORDER BY ?y
```

**Figura 4.10**: Ejemplo de consulta SPARQL de los valores de una propiedad

En la primera sentencia de la anterior figura, dentro de la cláusula *WHERE*, se indica que algo (?x) tiene una propiedad llamada *Country* cuyo valor se desconoce (?y). Y en la siguiente sentencia se especifica que ese valor ?y tiene que ser de tipo *EUMember*. También se podría haber utilizado rdf:type para consultar el tipo de recurso. Pero en este ejemplo se a utilizado el alias “a”.

La respuesta de esta consulta, mostrada por la figura 4.11, devuelve el listado de URIs que representan cada uno de los recursos que concuerdan con el patrón especificado en la cláusula *WHERE*. Evidentemente, al usuario solo se le mostrará la parte final del URI. Que se correspondería con el nombre del valor de la propiedad. Ya que es lo que le interesa.


```
{ "head": { "link": [], "vars": ["y"] },
  "results": { "distinct": false, "ordered": true, "bindings": [
    { "y": { "type": "uri", "value": "http://enipedia.tudelft.nl/wiki/Austria" }},
    { "y": { "type": "uri", "value": "http://enipedia.tudelft.nl/wiki/Belgium" }},
    { "y": { "type": "uri", "value": "http://enipedia.tudelft.nl/wiki/Bulgaria" }},
    { "y": { "type": "uri", "value": "http://enipedia.tudelft.nl/wiki/Croatia" }},
    { "y": { "type": "uri", "value": "http://enipedia.tudelft.nl/wiki/Cyprus" }},
    { "y": { "type": "uri", "value": "http://enipedia.tudelft.nl/wiki/Czech_Republic" }},
    { "y": { "type": "uri", "value": "http://enipedia.tudelft.nl/wiki/Denmark" }},
    { "y": { "type": "uri", "value": "http://enipedia.tudelft.nl/wiki/Estonia" }},
    { "y": { "type": "uri", "value": "http://enipedia.tudelft.nl/wiki/Finland" }},
    { "y": { "type": "uri", "value": "http://enipedia.tudelft.nl/wiki/France" }},
    { "y": { "type": "uri", "value": "http://enipedia.tudelft.nl/wiki/Germany" }},
    { "y": { "type": "uri", "value": "http://enipedia.tudelft.nl/wiki/Greece" }},
    { "y": { "type": "uri", "value": "http://enipedia.tudelft.nl/wiki/Hungary" }},
    { "y": { "type": "uri", "value": "http://enipedia.tudelft.nl/wiki/Ireland" }},
    { "y": { "type": "uri", "value": "http://enipedia.tudelft.nl/wiki/Italy" }},
    { "y": { "type": "uri", "value": "http://enipedia.tudelft.nl/wiki/Latvia" }},
    { "y": { "type": "uri", "value": "http://enipedia.tudelft.nl/wiki/Lithuania" }},
    { "y": { "type": "uri", "value": "http://enipedia.tudelft.nl/wiki/Luxembourg" }},
```


**Figura 4.11**: Respuesta del punto de acceso

## 4.5. Consulta final

Cuando el usuario presione el botón para ver los resultados de la consulta, internamente se procederá a crear el código SPARQL de la misma. Como se puede ver en el algoritmo de la figura 4.12, el proceso básicamente consiste en recorrer las estructuras de datos que contienen la información de las diferentes tablas que se muestran en la interfaz. Comprobando si se cumplen unas condiciones u otras para añadir una determinada sentencia de SPARQL. A destacar la comprobación, mediante la variable *hayOpMedida*, para averiguar si el usuario ha seleccionado alguna operación de agregado sobre alguna medida. Ya que de ella depende que se añadan o no las sentencias SPARQL que especifican grupos. Con lo que si se ha seleccionado alguna operación de agregado, serán necesarios crear grupos. Finalmente, y como se comentó en la sección 4.4.2, si el usuario no ha seleccionado ninguna medida hay que insertar la operación de agregado *count(*)*. Esto es comprobado con la variable *hayMedidasSelec*.

El algoritmo (pseudocódigo) se apoya en unas funciones auxiliares necesarias para crear las sentencias SPARQL. De las cuales, *setWhereSPARQL* y *setFilterSPARQL* precisan de una breve descripción:

- *setWhereSPARQL*. Esta función es la encargada de crear las sentencia de tipo WHERE en la consulta SPARQL. Necesita tres cadenas como parámetros de entrada. Donde cada parámetro se corresponde con cada uno de los componentes de una tripleta RDF: sujeto, predicado, objeto. Además, si alguno de estos tres parámetros es introducido como variable necesita anteponer el carácter "?".
- *setFilterSPARQL*. Esta función es la encargada de crear las sentencia de tipo FILTER en la consulta SPARQL. Necesita tres parámetros de entrada. El primero especifica la variable con los valores a filtrar. El segundo parámetro indica el tipo de filtro. Y como tercer parámetro, se ha de introducir un array donde cada componente se corresponde con cada valor a filtrar.

```
Requiere:
    nombreEntidad: nombre de la entidad central que se está utilizando para realizar la consulta,
    dataGridMedidas: array que contiene la información de cada una de las filas de la tabla de medidas,
    dataGridTiposDimensiones: array que contiene la información de cada una de las filas de la tabla de
        propiedades asociadas a las dimensiones escogidas,
    filtrosData: Matriz que contiene un array por cada filtro

INITIALISE hayMedidasSelec TO false
INITIALISE hayOpMedida TO false

FOR elem IN dataGridMedidas DO
    IF elem.seleccionado THEN
        SET hayMedidasSelec TO true
        setWhereSPARQL("?" + nombreEntidad, "prop:" + elem.nombre, "?" + elem.nombre)
        IF elem.operacion == 0 THEN
            setVariableSPARQL("?" + elem.nombre)
        ELSE
            nombreOp = getOpNombre(elem.operacion)
            nombreVariable = nombreOp + "_" + elem.nombre
            setVariableSPARQL(nombreOp + "(?" + elem.nombre + ") as ?" + nombreVariable
            SET hayOpMedida TO true

IF !hayMedidasSelec THEN
    setVariableSPARQL("COUNT( *) as ?COUNT_" + nombreEntidad)
ELSE IF !hayOpMedida THEN
    setVariableSPARQL("?" + nombreEntidad)

FOR elem IN dataGridTiposDimensiones DO
    IF elem.seleccionado AND elem.aplicaFiltro THEN
        IF hayOpMedida THEN
            setGroupBySPARQL("?" + elem.tipoPropiedad)
        setOrderBySPARQL("?" + elem.tipoPropiedad)
    ELSE IF elem.seleccionado AND !elem.aplicaFiltro THEN
        setVariableSPARQL("?" + elem.dimension)
        IF hayOpMedida THEN
            setGroupBySPARQL("?" + elem.dimension)
        setWhereSPARQL("?" + nombreEntidad, "prop:" + elem.tipoPropiedad, "?" + elem.dimension)
        setWhereSPARQL("?" + elem.dimension, "a", "cat:" + elem.dimension)
```

```
                setOrderBySPARQL("?" + elem.dimension)

IF existenFiltros() THEN
        FOR elem IN filtrosData DO
                nombre = elem.tipoPropiedad
                setVariableSPARQL("?" + nombre)
                setWhereSPARQL("?" + nombreEntidad, "prop:" + nombre, "?" + nombre)
                arrayValoresFiltro = obtieneValoresFiltro(elem)
                IF arrayValoresFiltro.length > 0 THEN
                        setFilterSPARQL("?" + nombre, "IN", arrayValoresFiltro)
```

**Figura 4.12**: Algoritmo para crear la consulta final

A continuación, se van a mostrar tres ejemplos de consultas que la aplicación crea internamente. Y que además envía al punto de acceso para obtener los resultados. Todo ello tras la selección de las medidas, dimensiones, propiedades y filtros por parte del usuario.

### 4.5.1. Consulta sin medidas

Consulta final creada visualmente con la interfaz de la aplicación, donde se han seleccionado las propiedades Country y Fuel_type (asociada a la dimensión Fuel). Pero no se ha escogido medida. Por lo que la aplicación añade *count(*)*. Para que por lo menos se muestren el número de elementos de cada grupo.

```
SELECT COUNT(*) as ?COUNT_Powerplant, ?Country, ?Fuel
WHERE
{
        ?Powerplant prop:Country ?Country .
        ?Country a cat:Country .
        ?Powerplant prop:Fuel_type ?Fuel .
        ?Fuel a cat:Fuel
}
ORDER BY ?Country ?Fuel
```

**Figura 4.13**: Ejemplo consulta SPARQL con count(*) como medida

### 4.5.2. Consulta con medidas y sin operaciones de agregados

Consulta final donde se pretende mostrar la capacidad de generación eléctrica que tienen las centrales eléctricas de España y Alemania. Cuyo tipo de combustible es el carbón o el agua. Además, junto a la capacidad eléctrica se recupera la central eléctrica, el país y el tipo de combustible.

```
SELECT ?Powerplant, ?Generation_capacity_electrical_MW, ?Country, ?Fuel_type
WHERE{
        ?Powerplant prop:Generation_capacity_electrical_MW ?Generation_capacity_electrical_MW .
        ?Powerplant prop:Country ?Country .
        FILTER (?Country IN (a:Germany,a:Spain)) .
        ?Powerplant prop:Fuel_type ?Fuel_type .
        FILTER (?Fuel_type IN (a:Coal,a:Hydro))
}
GROUP BY ?Country ?Fuel_type
ORDER BY ?Powerplant ?Country ?Fuel_type
```

**Figura 4.14**: Ejemplo de consulta SPARQL sin funciones de agregados en las medidas

### 4.5.3. Consulta con medidas y funciones de agregados

Consulta final donde se quiere ver la media de capacidad de generación eléctrica que tienen las centrales eléctricas de España y Alemania. Cuyo tipo de combustible es el carbón o el agua. Junto a la media se muestra el país y el tipo de combustible.

```
SELECT AVG(?Generation_capacity_electrical_MW) AS ?AVG_Generation_capacity_electrical_MW,
?Country, ?Fuel_type
WHERE{
        ?Powerplant prop:Generation_capacity_electrical_MW ?Generation_capacity_electrical_MW .
        ?Powerplant prop:Country ?Country .
         FILTER (?Country IN (a:Spain,a:Germany)) .
        ?Powerplant prop:Fuel_type ?Fuel_type .
        FILTER (?Fuel_type IN (a:Coal,a:Hydro))
}
GROUP BY ?Country ?Fuel_type
ORDER BY ?Country ?Fuel_type
```

**Figura 4.15**: Ejemplo de consulta SPARQL con funciones de agregados en las medidas

El resultado sería el mostrado en la figura 4.16

| AVG_Generation_capacity_electrical_MW | Country | Fuel_type |
|---|---|---|
| 436.333333333333333 | http://enipedia.tudelft.nl/wiki/Germany | http://enipedia.tudelft.nl/wiki/Coal |
| 116.123170731707317 | http://enipedia.tudelft.nl/wiki/Germany | http://enipedia.tudelft.nl/wiki/Hydro |
| 752.381818181818182 | http://enipedia.tudelft.nl/wiki/Spain | http://enipedia.tudelft.nl/wiki/Coal |
| 87.129195402298851 | http://enipedia.tudelft.nl/wiki/Spain | http://enipedia.tudelft.nl/wiki/Hydro |

**Figura 4.16**: Resultado consulta SPARQL con funciones de agregados en las medidas

# Capítulo 5

# Arquitectura

La arquitectura del sistema construido se compone de 3 capas principales. La parte cliente del la aplicación, la parte servidor y el catálogo intermedio. A continuación se van a introducir sus principales funciones:

- Cliente. El cliente es la parte donde va a residir la mayor parte de la lógica de la aplicación. Ya que es en aquí donde el usuario va a interactuar con los mecanismos para realizar las consultas. Se han utilizado tecnologías web que permiten aplicar interactividad y comunicación asíncrona con el backend y el catálogo intermedio de la aplicación. Por lo tanto, se utiliza JavaScript para manipular el DOM. AJAX para la comunicación asíncrona. Y HTML5/CSS3 como contenedor para presentar la información.
- Catálogo intermedio. Se ha creado un catálogo intermedio con los resultados de la exploración de la sección 4.1. Ofreciendo al usuario las entidades a las que pertenecen los datos tras el punto de acceso y sus áreas de interés analizables. De forma que el citado usuario pueda formular consultas sin la necesidad de conocer el lenguaje SPARQL. Y sin tener que conocer como son las tripletas de los datos en formato RDF de Enipedia.
- Servidor. La tecnología tras el servidor consiste en un servicio web REST (Representational state transfer) creado con PHP. Este servicio se encargará, principalmente, de recibir y contestar a las peticiones de la parte cliente de la aplicación. Responderá a consultas sobre el catálogo intermedio. Devolviendo los datos (entidades, medidas, dimensiones) con los que el usuario podrá realizar sus consultas a Enipedia.

Se podría añadir el punto de acceso de Enipedia como otro componente de la arquitectura de la aplicación. Ya que el resultado final de la consulta que se quiere obtener, será proporcionado por el citado punto de acceso.

Se podría hacer un símil con la arquitectura de tres capas presente en tantos sistemas informáticos. Donde la capa de presentación se correspondería con el cliente de la aplicación. La capa de negocio se correspondería con la lógica presente en el servidor. Y finalmente, la capa de datos se corresponde con nuestro catálogo intermedio.

La figura 5.1 muestra el esquema general de la arquitectura del sistema desarrollado. Donde se identifican los elementos anteriormente comentados.

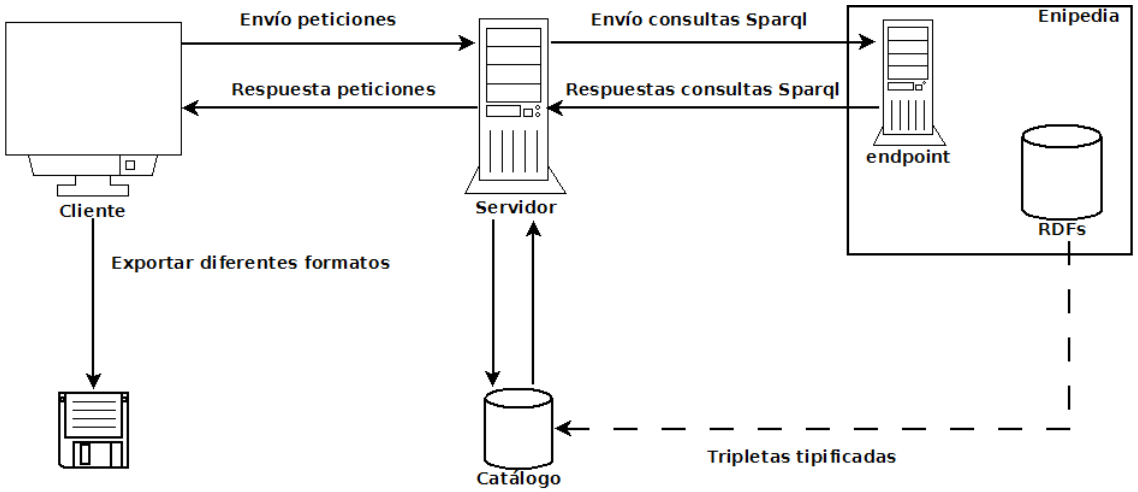



**Figura 5.1**: Esquema general de la aplicación

## 5.1. Tecnología Cliente

El cliente de la aplicación se corresponde con la interfaz que va a ver el usuario. Presentando el sistema, comunicándole la información y capturando la interacción de dicho usuario. Esta capa de la arquitectura se comunica únicamente con la capa del servidor. Ya que este último será el que se encargue de hacer las peticiones de datos almacenados en el catálogo o en el punto de acceso de Enipedia.

El cliente de la aplicación se divide a su vez en dos sub-capas: por un lado se tiene el contenedor HTML que se encarga de la presentación de la interfaz en el navegador del usuario. Por otro lado está la lógica del cliente. Que es la que se encarga de procesar las interacciones del usuario con la interfaz, y presentar los datos e información que mostrará la presentación HTML.

### 5.1.1. Presentación del cliente

Para el desarrollo de la interfaz, se ha utilizado HTML5 como contenedor para los datos e información. HTML5 es la nueva especificación de HTML (HyperText Markup Language) y XHTML (eXtensible HyperText Markup Language). Uno de los principales objetivos de HTML5 es llevar a los usuarios una experiencia de navegación superior. Al mismo tiempo que reduce la necesidad de utilizar componentes de terceras partes. Ya que HTML5 se ocupa de la animación, aplicaciones, música, películas, y todo lo que hay entre medio. Es más, HTML5 es multiplataforma. Eso significa que va a funcionar si estás navegando en una tablet, ordenador portátil o smartphone. Además el 28 de Octubre de 2014 pasó a considerarse como especificación recomendada por la W3C [1].

Lo correcto es separar el contenido de la presentación. Y HTML5 simplemente especifica como estructurar información. Por lo tanto, se debe aplicar un estilo a la presentación de dicha información. Para ello se ha utilizado el lenguaje de estilos CSS3. Y para un desarrollo más ágil en el estilo de la presentación, se ha utilizado la librería Bootstrap [2]. Que incorporá un gran conjunto de plantillas basadas en CSS3 y HTML5.

[1] `http://www.w3.org/TR/2014/REC-html5-20141028/`
[2] `http://getbootstrap.com/`

Hay navegadores web que se adaptan a las especificaciones según se van publicando. Permitiendo el uso correcto y casi completo de HTML5 y CSS3. Aunque se pueden encontrar pequeñas diferencias entre un navegador y otro. Sin embargo, los navegadores desactualizados o los que no implementan especificaciones hasta que estas no se conviertan en estándares, no soportan mucha de las características de CSS3 o HTML5. Y con este último grupo hay que tener cuidado. Ya que, por ejemplo, en los navegadores Internet Explorer inferiores a la versión 9 la aplicación no funciona correctamente.

### 5.1.2. Lógica del cliente: AJAX

En las aplicaciones web tradicionales los usuarios envían peticiones al servidor web. Y este contesta enviando una nueva página web. Esta metodología tiene los siguientes inconvenientes:

- Se desperdicia mucho ancho de banda enviando una página completa.
- Obligación de mostrar toda la página de la respuesta.
- El usuario tiene que esperar hasta recibir la respuesta para poder realizar otra acción en la aplicación web.

Para solucionar esta forma de trabajar aparece la tecnología AJAX (Asynchronous JavaScript And XML). Con la que desde el cliente se envían peticiones, vía HTTP, para obtener únicamente la información necesaria. Y nuevamente es el cliente el que procesa la respuesta del servidor Web. Además esta petición se realiza como proceso de fondo. Por lo que el usuario no tiene que esperar que el proceso concluya en su totalidad para continuar interactuando con la aplicación.

Linked Data Visual Query Builder permite una gran interacción por parte del usuario. Y esto provoca numerosos cambios dinámicos en la interfaz. Esto quiere decir que la mayor parte de la lógica de la aplicación recaerá en el cliente. Donde la parte más importante y compleja está presente a la hora de crear la interfaz para las consultas visuales. Debido a que continuamente se están lanzando peticiones y recibiendo respuestas para el llenado dinámico de las tablas de la interfaz.

La lógica del cliente está desarrollada mediante el lenguaje de programación JavaScript. Y a través de él, se envían las peticiones AJAX a la capa servidor de la aplicación. Como no hay exigencias de seguridad, ya que no se envían datos que supongan un peligro, se ha optado por hacer uso de peticiones de tipo *GET* hacia el servidor. Además, al no estar usando https en el envío de las peticiones, no hay diferencia efectiva en la seguridad entre el uso de *GET* o *POST*.

Como se ha comentado, el peso del cliente recae en la interfaz de consultas visuales. Así que a continuación se van a analizar las peticiones que se realizan en dicha interfaz.

1. Inicialmente, tal como se indica en la figura 5.2, se envían dos peticiones AJAX asíncronas al servidor. La primera de ellas hará una petición para obtener el listado de las medidas de la entidad elegida. Y la segunda petición pretende obtener las dimensiones de la entidad a analizar. Con esto se llenan las dos primeras tablas de la interfaz de forma asíncrona. Esto quiere decir que se puede empezar a seleccionar elementos de la tabla poblada, con los datos de la petición, sin tener que esperar a que la otra tabla se haya completado. Ya que cada petición, y por lo tanto el llenado de cada tabla, es independiente.

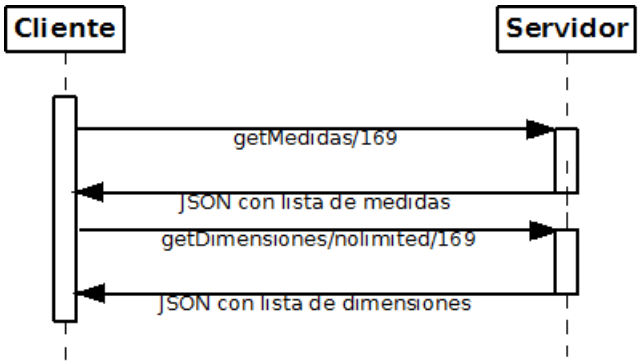



**Figura 5.2**: Esquema de las peticiones AJAX solicitando dimensiones y medidas

2. En el momento que la tabla de dimensiones ya esté poblada, el usuario ya puede seleccionar las dimensiones que le interese analizar. Al seleccionar una de ellas, en la tercera tabla se mostrarán las diferentes propiedades asociadas con la dimensión (ver sección 4.4.3). Aunque es muy posible que solo aparezca una propiedad por dimensión. Por lo tanto, por cada selección de una dimensión se enviará una petición AJAX asíncrona para recibir las propiedades e ir poblando la tabla de propiedades. En la figura 5.3 se puede ver un ejemplo donde se realizan 3 peticiones AJAX. Provocadas por la selección de tres dimensiones por parte del usuario.

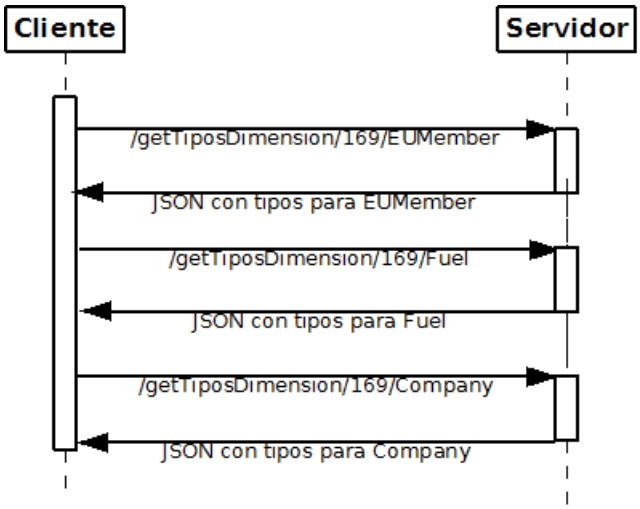



**Figura 5.3**: Esquema de las peticiones AJAX solicitando propiedades

3. Una vez el usuario ha elegido las propiedades asociadas a dimensiones que quiere analizar, puede decidir filtrarlas. De tal forma que solo analizará dichas dimensiones en base a los miembros elegidos en los filtros. Para ello, por cada filtro elegido el cliente lanzará una petición AJAX asíncrona hacia el punto de acceso de Enipedia. Esperando una respuesta sobre los valores que toma la propiedad.

4. Finalmente, la última petición AJAX que se realiza es hacia el punto de acceso. Enviando la consulta final SPARQL creada a partir de toda la interacción del usuario.

Si se pasa de una interfaz a otra de la aplicación, y se vuelve a la pantalla de realizar la consulta sin haber cambiado de entidad central, las dos primeras peticiones AJAX no se efectuarán de nuevo. Ya que sus contestaciones se habrán almacenado.

Las peticiones 3 y 4 de la lista anterior son diferentes a la predecesoras. Ya que las peticiones 1 y 2 se realizan sobre el servidor de la aplicación. Sin embargo, las peticiones

AJAX 3 y 4 se deben de hacer sobre el punto de acceso de Enipedia. Lo que supone hacer peticiones a un servidor externo al dominio de la aplicación. Donde los navegadores tienen restricciones debido a sus políticas del "mismo origen" (cross domain en inglés). Por lo tanto, se han estudiado 2 posibles soluciones:

- Utilizar el mecanismo JSONP[3]. Con este mecanismo, en lugar de viajar el dato a secas como pasa en JSON, lo que viaja es una función generalmente llamada "callback". Esa función es código JavaScript que envuelve el dato JSON que se ha solicitado. De ahí que a veces se conozca a JSONP como JSON con Padding. Con lo que los navegadores no bloquearán la respuesta de la petición. Ya que sí que aceptan la carga de código JavaScript que contenga la etiqueta SCRIPT y el atributo src de otro dominio. Por lo tanto, esta parece una solución fácil al problema. Sin embargo, se tiene que contar con que en el servidor del punto de acceso puede estar definida la cabecera "X-Content-Type-Options: nosniff[4]". Y como se observa en la figura 5.4, el servidor del punto de acceso de Enipedia tiene activa esa cabecera.

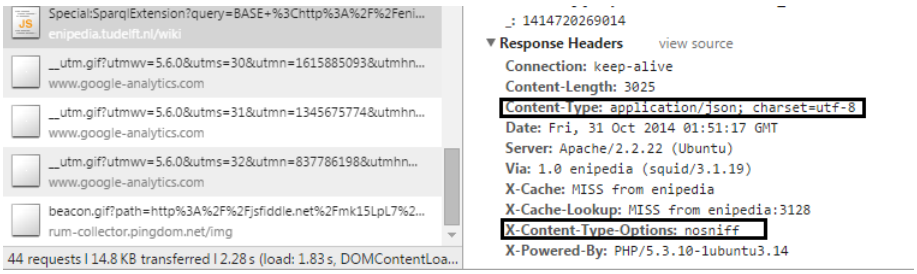



**Figura 5.4**: Cabeceras del punto de acceso de Enipedia

  Al establecer la cabecera anterior al valor "nosniff", el servidor enviará dicha cabecera junto con la respuesta. Esto impide que los navegadores que soportan la opción, puedan utilizar el contenido recibido para determinar el tipo de ese contenido. Con lo que deben basarse estrictamente en la cabecera "content-type" recibida. Y esa cabecera, sin embargo, establece el tipo como "application/json" en lugar de "application/javascript" que busca el navegador. Se debe recordar que una petición JSONP devuelve código JavaScript. Con lo que los navegadores Chrome e IE tratan a la respuesta como JSON y se niegan a ejecutarlo.

- Utilizar un proxy. Como se ha visto, utilizar peticiones AJAX de tipo JSONP no es la opción correcta en el desarrollo de la aplicación. Por lo tanto, queda la opción de utilizar un proxy en la parte servidor de la aplicación. En vez de que el cliente envíe las peticiones AJAX directamente al punto de acceso, el cliente las enviará al proxy y este, a su vez, hará las peticiones al punto de acceso. De tal forma que el proxy, tras recibir la respuesta del punto de acceso, será el que conteste al cliente. Y como ambos pertenecen al mismo dominio (cliente y servidor) se resuelve el problema del "mismo origen" de los navegadores. El único inconveniente es el hecho de estar estableciendo más peticiones. Ya que ahora se deben enviar una petición al proxy y este la enviará al punto de acceso. Pero al fin y al cabo, las peticiones seguirán siendo asíncronas. Ya que las peticiones AJAX que se envían al proxy así lo son. Finalmente, se puede ver un esquema del funcionamiento en la figura 5.5.

[3] `http://json-p.org//`
[4] `https://www.owasp.org/index.php/List_of_useful_HTTP_headers`

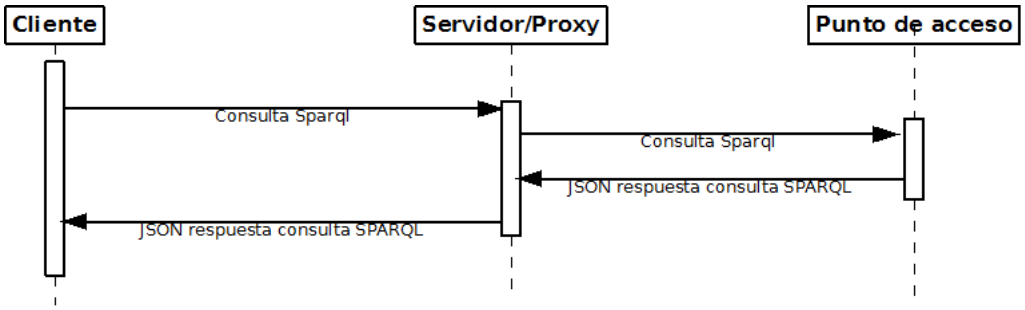



**Figura 5.5**: Esquema peticiones AJAX al punto de acceso mediante un proxy

Si se pudiera manipular el servidor sobre el que está el punto de acceso de Enipedia, y se quisiera que clientes externos al dominio pudiesen hacer peticiones sin tener que utilizar JSONP, estaría la opción de utilizar el mecanismo llamado CORS (Cross-origin resource sharing). CORS trabaja añadiendo nuevas cabeceras HTTP, las cuales permiten especificar en el servidor que tipo de conexión hacer y desde donde son autorizadas.

```
<?php
header("Access-Control-Allow-Origin: *");
header("Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept");
?>
```

**Figura 5.6**: Ejemplo para activar CORS en un servidor PHP

Con el código de la figura 5.6, se estaría dando acceso a solicitudes desde cualquier cliente. Para permitir acceso solo a clientes que sean de confianza habrá que modificar el script cambiando el * por la URL del dominio de confianza. Sin embargo, CORS debe de estar soportado por el cliente. Y la versiones de Internet Explorer inferiores a 10 vuelven a dar problemas [5].

## 5.2. Catalogo intermedio

Como ya se explicó en 4.3, las tripletas tipificadas obtenidas de la exploración son la base para la construcción del catálogo intermedio. Dicho catálogo está formado por una base de datos MySQL relacional compuesta por 3 tablas:

- Tabla *entidad_central*. En ella se almacena el identificador, el nombre y el número de instancias de cada uno de los conceptos extraídos de la exploración del dataset de Enipedia. Cada entidad central, junto a sus medidas y dimensiones, forma un modelo multidimensional (ver sección 4.3).
- Tabla *medida*. En ella se almacenan cada una de las medidas que hay en los modelos multidimensionales. Además del identificador, nombre y tipo numérico de la medida, existe una clave ajena ("id_entidad_central") que apunta al identificador de la entidad central a la que pertenece la medida. Con su regla de integridad correspondiente. Y finalmente, están los campos "seleccionado" y "operacion" necesarios para la creación de la consulta final en el cliente de la aplicación.
- Tabla *dimension*. Al igual que ocurre con las medidas, se deben almacenar las dimensiones pertenecientes a las entidades centrales. Por lo que se debe asociar cada

[5] `http://caniuse.com/#search=CORS`

dimensión con la entidad central a la que pertenece por medio de una clave ajena ("id_entidad_central"). Como se explicó en 4.4.3, una dimensión puede ir asociada a varias propiedades. Y para almacenar una propiedad asociada a la dimensión se utilizará el campo "tipo". Finalmente, están los campos "seleccionado" y "aplicaFiltro" necesarios para la creación de la consulta final en el cliente de la aplicación.

En la figura 5.7 se puede ver el código SQL necesario para crear la base de datos. Y en la figura 5.8 se muestra la estructura de la misma en un gestor de base de datos.

```
DROP DATABASE IF EXISTS enipedia_entities;
CREATE DATABASE IF NOT EXISTS enipedia_entities;
USE enipedia_entities;

CREATE TABLE entidad_central (
        `id` int PRIMARY KEY NOT NULL AUTO_INCREMENT,
        `nombre` varchar(100) NOT NULL,
        `num_instancias` int NOT NULL) ENGINE=InnoDB
DEFAULT CHARACTER SET utf8 DEFAULT COLLATE utf8_general_ci ;

CREATE TABLE medida (
        `id` int PRIMARY KEY NOT NULL AUTO_INCREMENT,
        `id_entidad_central` int NOT NULL,
        `nombre` varchar(100) NOT NULL,
        `tipo` varchar(30) NOT NULL,
        `seleccionado` varchar(5) NOT NULL DEFAULT 'false',
        `operacion` tinyint(1) NOT NULL DEFAULT '0',
        CONSTRAINT `medida_ecentral_fk` FOREIGN KEY (`id_entidad_central`)
        REFERENCES `entidad_central` (`id`) ON DELETE CASCADE
) ENGINE=InnoDB
DEFAULT CHARACTER SET utf8 DEFAULT COLLATE utf8_general_ci;

CREATE TABLE dimension (
        `id` int PRIMARY KEY NOT NULL AUTO_INCREMENT,
        `id_entidad_central` int NOT NULL,
        `nombre` varchar(100) NOT NULL,
        `tipo` varchar(30) NOT NULL,
        `num_instancias` int NOT NULL,
        `seleccionado` varchar(5) NOT NULL DEFAULT 'false',
        `aplicaFiltro` varchar(5) NOT NULL DEFAULT 'false',
        CONSTRAINT `dimension_ecentral_fk` FOREIGN KEY (`id_entidad_central`)
        REFERENCES `entidad_central` (`id`) ON DELETE CASCADE
) ENGINE=InnoDB
DEFAULT CHARACTER SET utf8 DEFAULT COLLATE utf8_general_ci;
```

**Figura 5.7**: Código SQL creación almacén

**dimension**

| Field | Type | Null | Default |
|---|---|---|---|
| id | int(11) | No | |
| id_entidad_central | int(11) | No | |
| nombre | varchar(100) | No | |
| tipo | varchar(30) | No | |
| num_instancias | int(11) | No | |
| seleccionado | varchar(5) | No | false |
| aplicaFiltro | varchar(5) | No | false |

**entidad_central**

| Field | Type | Null | Default |
|---|---|---|---|
| id | int(11) | No | |
| nombre | varchar(100) | No | |
| num_instancias | int(11) | No | |

**medida**

| Field | Type | Null | Default |
|---|---|---|---|
| id | int(11) | No | |
| id_entidad_central | int(11) | No | |
| nombre | varchar(100) | No | |
| tipo | varchar(30) | No | |
| seleccionado | varchar(5) | No | false |
| operacion | tinyint(1) | No | 0 |

**Figura 5.8**: Estructura de la base de datos del catálogo intermedio

Una vez creada la base de datos, esta debe ser poblada mediante un proceso de lectura de los ficheros especificados en la sección 4.3. Procesando iterativamente cada una de las lineas de los ficheros, para extraer y almacenar en la base de datos cada uno de las entidades centrales, medidas y dimensiones.

## 5.3. Tecnología Servidor

La capa del servidor está comprendida por el conjunto de scripts PHP encargados de realizar tres tareas. La primera función que tiene el servidor es mantener un servicio REST encargado de recibir y contestar las peticiones sobre datos del catálogo. Ya que el cliente de la aplicación estará enviando peticiones AJAX al servicio REST cuando solicite las dimensiones, medidas y propiedades. Este servicio no precisa de autenticación, por lo que una simple petición GET con el recurso requerido es suficiente. Además, la configuración del servidor ha sido modificada, mediante el fichero .htaccess, para que los recursos puedan solicitarse mediante un formato de URLs "amigables". También llamadas URLs semánticas.

Destacan 4 tipos de peticiones que proporciona el servicio REST:

- getEntidadesCentrales/[(parámetro = límite)]. Con esta petición el servidor devolverá una lista de entidades centrales ordenadas por orden de importancia. Dependiendo del número de instancias. Con el identidicador, nombre, numero de instancias y el nombre de sus 10 dimensiones más importantes. El número de entidades centrales dependerá del parámetro límite. Aunque si no se especifica, por defecto se devolverá 10 entidades centrales.
- getMedidas/(parámetro = identificador). Con esta petición se obtiene una lista con las medidas de la entidad central identificada por el parámetro identificador.

- getDimensiones/(parámetro1 = cantidad)/(parámetro2 = identificador). Mediante esta petición el cliente obtendrá una lista con las dimensiones de la entidad central identificada por el parámetro identificador. Además, el parámetro cantidad podrá tomar dos valores: "limited" o "nolimited". Dependiendo de si se quiere solo las 10 dimensiones más importantes o todas.
- getTiposDimension/(parámetro1 = identificador)/(parámetro2 = nombre). Con esta petición el servidor devuelve una lista de las propiedades asociadas con la dimensión especificada por el parámetro nombre. Donde el parámetro identificador indicará la entidad central a la que pertenece la dimensión.

El servidor, para gestionar a las peticiones anteriores hará consultas al catálogo intermedio. Y devolverá los resultados en formato JSON al cliente.

La segunda de las funciones que se ha implantado en la capa del servidor es la de proxy. Como ya se explicó en la sección 5.1.2, el cliente no puede lanzar peticiones a un servidor que está fuera del dominio. A no ser que se utilice JSONP en la petición AJAX. Sin embargo, como el servidor destino de la petición tiene activada la cabecera "X-Content-Type-Options: nosniff", ciertos navegadores bloquearán la respuesta de la petición. Aunque se haya utilizado JSONP. Por lo tanto, la opción que se ha tomado es la de crear un proxy en la capa del servidor. De este modo, las peticiones AJAX con consultas a Enipedia pasen por el proxy, en lugar de ir directamente al punto de acceso (ver figura 5.5). El proxy no es más que un script PHP que recoge la petición GET, descompone los parámetros y crea la URL que será enviada al punto de acceso de Enipedia por medio de la herramienta cURL. Posteriormente, cuando reciba la respuesta la devolverá al cliente.

Finalmente, el servidor se encarga de la creación de ficheros con los resultado finales que el usuario quiera generar y exportar. Por razones de seguridad, no es aconsejable que un cliente HTML5, y por lo tanto JavaScript, acceda a los ficheros locales del usuario. Y aunque técnicamente es posible [6], se ha preferido que esta tarea la realice el servidor. Por lo que se ha creado un script PHP que se encarga de procesar la petición AJAX con la respuesta final del punto de acceso. Y dependiendo del tipo de fichero que quiera el usuario, el script PHP generará un fichero u otro.

[6] `http://www.w3.org/TR/FileAPI`

# Capítulo 6

# Elección del framework JavaScript

Como se comentó en el capítulo 5, se ha elegido JavaScript como lenguaje para desarrollar el cliente de la aplicación. Para que sea el encargado de interactuar con el Document Object Model (DOM) de HTML y de dotar de dinamismo a la interfaz. Ahora bien, hay que tener en cuenta que según crece la complejidad del dinamismo de la interfaz, crecen el número de lineas de programación. Y escribir código mantenible y reutilizable es crucial. Por lo que el uso de un framework o una librería Javascript es más que recomendable.

Escribir código fuente de calidad es crucial en el desarrollo de aplicaciones web. Los patrones de diseño son importantes para escribir ese código estructurado. Un patrón es una solución reutilizable que se puede aplicar a los problemas que ocurren comúnmente en el diseño de software. Y uno de los patrones más extendido en el mundo de la programación es el MVC (Model-View-Controller ). Patrón que ayuda a escribir código más organizado y mantenible. Y para ello, tradicionalmente, separa la estructura de una aplicación en tres categorías o componentes independientes:

- Modelo. El modelo de la aplicación representa conocimientos y datos que varían a lo largo del tiempo. El modelo está aislado y no sabe nada de la vista ni de los controladores. Cuando un modelo cambia, normalmente notificará a sus observadores que un cambio ha ocurrido.
- Vista. Las vistas, desde la perspectiva de una aplicación web, se pueden considerar la interfaz de usuario con la que los usuarios interactuarán. Las vistas están aisladas y no saben nada sobre el modelo ni el controlador.
- Controlador. El controlador es el pegamento entre la vista y el modelo. Es el que toma las decisiones. Se encarga de actualizar la vista cuando el modelo cambia. Además de modificar el modelo cuando los usuarios manipulan la vista.

Una de las grandes consecuencias del uso de este patrón, y por lo tanto de la separación de código, es que varios desarrolladores pueden trabajar simultáneamente. Siendo cada uno de ellos responsable de uno de los diferentes elementos de la estructura de la aplicación.

La elección de las herramientas adecuadas es algo fundamental y que marca el desarrollo de todo proyecto. En los siguientes apartados se van a presentar 3 herramientas JavaScript con los que se hicieron pruebas para elegir la idónea para el presente proyecto. La elección se ha basado en términos de estructuración del código de la aplicación y actualización de HTML (interacción con el DOM).

# 6.1. jQuery

## 6.1.1. Estructura de la aplicación

La librería jQuery[1] proporciona una capa de abstracción de propósito general para hacer dinámica la parte cliente de una aplicación web. Ofreciendo un conjunto enorme de métodos que simplifican la realización de ciertas tareas, respecto a realizarlas sin ningún tipo de herramienta. Está enfocado en ser muy compacto y hacer muy fácil la manipulación del DOM. Sin embargo, no proporciona una infraestructura importante para la construcción de aplicaciones a gran escala. jQuery no marca ningún conjunto de normas o hábitos en la programación. O dicho de otra forma, no propone ninguna serie de patrones para implementar las aplicaciones web.

La librería jQuery no se centra en la forma en la que se escribe código. Lo que ocasiona que se acabe teniendo un montón de código "spaghetti" que después se minimiza en un archivo para servir en los sitios o aplicaciones web. Esto a la larga se convierte en una maraña de selectores y secuencias de llamadas. Que el programador tiene que mantener sincronizados con el código HTML para que sigan funcionando como se espera. Lo que provoca grandes problemas en el mantenimiento y modificaciones futuras de las aplicaciones. Por lo tanto, la mayor consecuencia que ocasiona el hecho de no marcar una dirección a la hora de organizar código, es que al final se acaba teniendo una cantidad considerable de funciones que se encargan tanto de la lógica, como de modificar el modelo o las vistas.

Manualmente se puede crear un conjunto de clases para crear el comportamiento de una arquitectura de aplicación MVC. Pero eso requiere tiempo de desarrollo. Tiempo que no se perdería si se elige un framework que ya obligue a utilizar este patrón. Y aquí es donde aparece la diferencia entre librería y framework. Ya que este último concepto si que implica una arquitectura de software determinada. Por lo tanto, jQuery no es la mejor opción si la estructuración es algo primordial para el desarrollo. Como es el caso de la presente aplicación.

## 6.1.2. Manipulación de la interfaz HTML

Hay dos objetivos principales al desarrollar una aplicación interactiva: visualización de contenido y la creación/manipulación de contenido. HTML no fue diseñado para crear vistas dinámicas. Fue diseñado para mostrar contenido estático y no ofrece un medio eficaz para su manipulación después de la representación inicial. jQuery no ayuda eficazmente con esta deficiencia. Como se ha comentado en el apartado anterior, jQuery no utiliza el patrón MVC. Por lo que no se crea ningún objeto que representa el modelo de la aplicación. Esto quiere decir que manipula el DOM directamente. O dicho de otro modo, el DOM es el modelo de la aplicación. Lo que presenta los siguientes inconvenientes:

- No ofrece facilidades para que el cambio del modelo de datos pueda ser sincronizado con el servidor de la aplicación.
- Cada cambio en el modelo requiere acceder al DOM y modificarlo manualmente. No ofrece ningún tipo de manipulación automática. Como se puede ver en la figura 6.1, en todo momento es necesario acceder y manipular el DOM.

[1] `http://jquery.com/`

```
var generateElement = function(params) {
  var parent = $("#entidad-central"), wrapper;

  if (!parent) {
    return;
  }

  wrapper = $("<div />", {
    "class" : "medida-ecentral",
    "id" : params.id,
  }).appendTo(parent);

  $("<div />", {
    "class" : "medida-nombre",
    "text": params.nombre
  }).appendTo(wrapper);

  $("<div />", {
    "class" : "medida-ninstancias"
    "text": params.ninstancias
  }).appendTo(wrapper);

};

generateElement({
  id: "123",
  nombre: "Point",
  ninstancias: 5225
});
```

**Figura 6.1**: Ejemplo jQuery

- El hecho de que no se disponga de ningún objeto que represente a un modelo, provoca que si se quiere acceder al contenido de un elemento de la interfaz, se tenga que estar incluyendo un identificador a cada elemento del DOM. Ya que jQuery solo puede acceder a los elementos del DOM por medio de identificadores, clases o nombre de tags HTML.

Como se ha visto, con jQuery el código a desarrollar está muy estrechamente ligado a la estructura del DOM y eso es un problema. Ya que cualquier cambio dinámico que se produzca en la interfaz requerirá el acceso al elemento DOM correspondiente. Además de su procesado y modificación. Y en una aplicación como la desarrollada, donde el cambio dinámico en las tablas de la interfaz es continuo, es muy poco productivo tener que programar manualmente cada cambio en la estructura del DOM. Por lo tanto, nuevamente este gran inconveniente hace que jQuery no sea la herramienta adecuada para este tipo de desarrollo.

## 6.2. Backbone

### 6.2.1. Patrón MVC de Backbone

Backbone[2][3] es un framework que permite estructurar el código de una aplicación web siguiendo un patrón MVC propio. Para ello hace uso de 4 clases o componentes de primer orden:

- Modelo. Los modelos pueden significar diferentes cosas en diferentes implementaciones de MVC. En Backbone, un modelo representa una entidad singular. El modelo sólo ofrece una manera de leer y escribir propiedades o atributos arbitrarios en un conjunto de datos. En la figura 6.2 se puede ver un ejemplo de modelo.

```
var EntidadSecModel = Backbone.Model.extend({
    initialize: function() {
        this.atributos = null;
        this.id = this.get('id');
        this.nombre = this.get('nombre');
        this.numIstancias = this.get('num_instancias');
    }
});
```

**Figura 6.2**: Ejemplo de modelo de datos en Backbone

- Colección. Una colección en Backbone, como la creada en la figura 6.3, es sencillamente una colección de modelos. Las colecciones desencadenan eventos cuando se añaden o se quitan modelos de ellas. Y del mismo modo que ocurre con los modelos, en una colección está presente la posibilidad de escuchar y producir eventos. Además de incluir una interfaz para la sincronización con el servidor.

```
var EntidadesSecCollection = Backbone.Collection.extend({
        model: EntidadSecModel
});
```

**Figura 6.3**: Ejemplo de colecciones en Backbone

- Vista. Una vista en Backbone puede que se aleje del concepto de vista más purista. Ya que se parece más a un controlador (ver figura 6.4). En Backbone las vistas se usan para reflejar como es el aspecto HTML de los modelo de datos. Por lo que también se encargan de capturar y reaccionar a los eventos entrantes. Sin embargo, Backbone por si solo no tiene ningún motor de plantillas con lo que usa y requiere la librería Underscore.js que si incluye un motor de plantillas.

[2] http://backbonejs.org/
[3] http://backbonetutorials.com/

```
EntidadSecView = Backbone.View.extend({
    type: "EntidadSecView", //for debugging
    template: _.template($("#entidadSecElemTemplate").html()),
    tagName: "tr",
    events: {
        "click td": "viewClicked",
        "mouseenter": "anadirTitleNecesario"
    },
    render: function() {
        var outputHtml = this.template(this.model.toJSON());
        this.$el.html(outputHtml);
        return this;
    },
    viewClicked: function(event) {
        //console.log("viewClicked: " + this.model.get("nombre"));
    },
    anadirTitleNecesario: function() {
        var $td = $(this.el).find("td.nombre");
        if ($td[0].offsetWidth < $td[0].scrollWidth && !$td.attr('title'))
            $td.attr('title', this.model.nombre);
    }
});
```

**Figura 6.4**: Ejemplo de vista en Backbone

Cada instancia de una vista contendrá una propiedad llamada "el". Que será el objeto DOM que tendrá todo el contenido de la vista. Este elemento se crea automáticamente como un elemento "div" vacío de HTML. Salvo que se indique lo contrario con las propiedades *tagName* y *className*. Todas las vistas deben sobreescribir el método "*render*" puesto que es un método vacío. Obviamente, el código del método render debe renderizar la plantilla de la vista con los datos del modelo. Y actualizar la propiedad "el" con el nuevo código HTML.

- Controlador. Los controladores de Backbone no tiene la misma función que los típicos controladores MVC. Los controladores o routers se usan para enrutar URLs de las aplicaciones. Y deben de contener al menos una ruta y una función con la que mapear su comportamiento. Un ejemplo de controlador se muestra en la figura 6.5.

```
var controladorApp = Backbone.Controller.extend({
  routes: {
    "!/": "root",
    "!/prueba": "prueba",
  },
  root: function() {
  },

  prueba: function() {
  },
});
```

**Figura 6.5**: Ejemplo de controlador en Backbone

### 6.2.2. Problemas de Backbone

Backbone tiene una curva de aprendizaje muy lineal y es algo más fácil para empezar que otros frameworks para JavaScript. Ya que básicamente se sustenta en los cuatro conceptos comentados. Sin embargo, tiene 3 puntos negros que hacen que no sea una opción óptima para el desarrollo de la presente aplicación.

- Backbone no obliga a seguir una estructura. Más bien proporciona algunas herramientas básicas que se pueden utilizar para crear la estructura. Dejando a los desarrolladores la decisión de cómo estructurar su aplicación.
- Una aplicación con Backbone debe de dividirse en varios modelos, colecciones y vistas. Si esa aplicación tiene una interfaz con múltiples elementos, que varían dinámicamente dependiendo de la interacción del usuario, el número de modelos o vistas puede ser elevado. Esto quiere decir que habrá que dedicar mucho tiempo escribiendo código para actualizar cada vista cuando su modelo asociado cambie. Y actualizar el modelo cada vez que la vista cambie.
- Las vistas en Backbone manipulan el DOM directamente. Haciéndolas más difíciles de testear, más frágiles y menos reutilizable. La práctica común es buscar elementos DOM usando selectores CSS. Por lo que el cambio de un nombre de clase CSS, o añadir un nuevo elemento con el mismo nombre de clase, puede dar problemas en el funcionamiento de la aplicación. Al igual que con jQuery, las vistas de Backbone trabajan muy estrechamente con el DOM. Lo cual no es la mejor opción.

## 6.3. AngularJS

### 6.3.1. Patrón MVC con AngularJS

AngularJS[4] promueve y usa patrones de diseño de software. En concreto, implementa el citado patrón MVC. Básicamente, y como ya se ha comentado, este patrón marca la separación del código dependiendo de su responsabilidad. Eso permite repartir la lógica de la aplicación por capas. Lo que resulta muy adecuado para aplicaciones de negocio y para las aplicaciones SPA (Single Page Aplication). A continuación se van a describir brevemente los componentes de este patrón en AngularJS:

- Modelo. El modelo en AngularJS puede ser un tipo primitivo. Como una cadena, número, booleano o un tipo complejo como un objeto. El modelo es sólo un simple objeto JavaScript. Y se define mediante la variable "scope", que marca un alcance o ámbito. Esta variable es la que contiene la información del controlador. En él se podrá asignar propiedades nuevas con los datos necesarios. O incluso con funciones. Ya que cada controlador tendrá un ámbito único. En la figura 6.6 se puede ver un ejemplo en el que se define un modelo de datos.

```
<script>
    $scope.dimension = {'nombre':'EUMember','propiedad':'Country','num-instancias':52200};
</script>
```

**Figura 6.6**: Ejemplo de modelo de datos en AngularJS

[4] `https://angularjs.org/`

- Vista. Una vista es lo que los usuarios ven. Y en AngularJs una vista es una plantilla HTML creada con su propio motor de plantillas. Para visualizar los datos del controlador se puede poner una expresión de AngularJS en la vista. Esta expresión enlaza datos desde el modelo interno de un controlador. Y como AngularJS trabaja con el llamado "enlace de datos de doble vía", la vista se actualizará automáticamente si hay un modelo que cambia dentro del controlador asociado. No es necesario escribir código adicional para lograrlo. Lo cual es una gran ventaja respecto jQuery y Backbone. Ya que AngularJS hace una cantidad considerable de trabajo automáticamente. La figura 6.7 propone un ejemplo de vista. Mostrando los elementos que interactúan con ella.

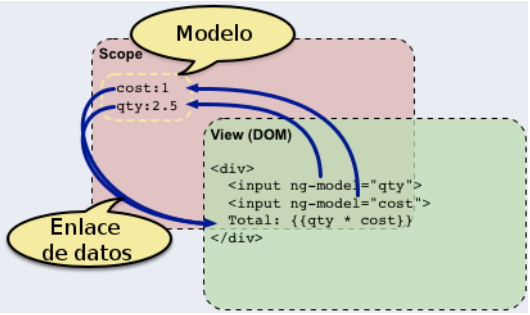



**Figura 6.7**: Esquema vista - enlace de datos AngularJS

- Controlador. El controlador es el lugar donde se pone la lógica de la aplicación. En AngularJS, un controlador está simplemente formada por una clase de JavaScript. Que se encargan principalmente de:

  - Inicializar el estado del ámbito para que la aplicación tenga los datos necesarios para comenzar a funcionar. Y poder presentar información correcta al usuario en la vista.

  - Además, es el lugar adecuado para escribir código que añada funcionalidades o comportamientos al citado ámbito (métodos, funciones,...).

  En un controlador también se pueden llamar a otros componentes o funciones específicas que proporciona el framework. Pero lo más importante es remarcar que en los controladores es donde viven los modelos. Como se observa en la figura 6.8, cada controlador contiene su modelo. Determinado por el alcance o ámbito. Y solo será utilizado por el citado controlador y por la vista asociada gracias al "enlace de datos de doble vía". O dicho de otra forma, cada controlador define su propio ámbito donde se asocian modelo de datos y vista.

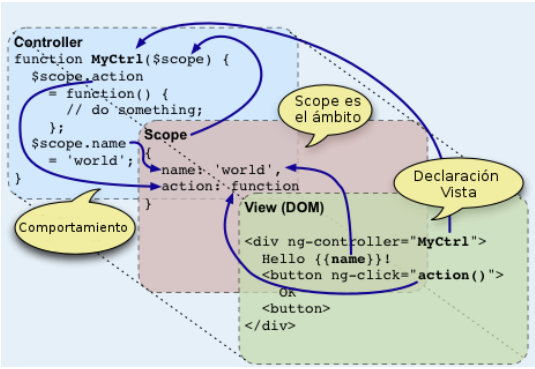



**Figura 6.8**: Esquema controlador AngularJS

Esta estructuración de código que obliga el framework, unido a la facilidad de comunicación entre los tres componentes, hace que AngularJS sea una opción muy válida para el desarrollo de aplicaciones. Sobre todo si hay muchos cambios dinámicos en el modelo de datos y en las vistas. Como en la presente aplicación.

### 6.3.2. Enlace y ámbito de los datos

En AngularJS se pueden tener muchos ámbitos en una misma aplicación. Y por lo tanto se pueden tener muchos modelos de datos. El ámbito "rootScope" es el ámbito de mayor nivel en el contexto de una aplicación. Esto significa que desde cualquier lugar de cualquier vista se pueden referenciar variables que se encuentren en el objeto rootScope. Sin embargo, dejar todas las variables en este único ámbito hace que actúen como variables globales. Ocasionando que en una aplicación grande y compleja surjan problemas para gestionar los datos. Y el código empezaría a ser confuso. Para evitar esto, AngularJS tiene la capacidad de organizar ámbitos mediante relaciones padres/hijos. Pudiendo de esta forma crear un ámbito hijo (scope) para cada controlador de la aplicación. Y de esta forma cada controlador tendrá su modelo de datos propio.

El ámbito es un gran contenedor de datos. Que transporta y hace visible la información desde el controlador a la vista y desde la vista al controlador. En términos de código, el ámbito no es más que un objeto al que se puede asignar propiedades o métodos nuevos. Esos datos y esos métodos están visibles tanto en los controladores como en el HTML de las vistas. Sin que se tenga que realizar ningún código adicional. Ya que AngularJS ya se encarga de ello automáticamente. Además, cuando surgen cambios en los modelos de datos, estos se propagan entre los controladores y las vistas automáticamente. Gracias a dos tipos de enlace de datos:

- Enlace de datos de una vía. La información solamente viaja desde el ámbito hacia la parte visual. O dicho de otro modo, desde el modelo hacia la representación de la información en el HTML. Lo que significa que si se actualiza un dato que hay almacenado en el modelo, también se actualizará automáticamente en la presentación. Y esto se consigue utilizando en las plantillas la sintaxis "Mustache[5]" de las dos llaves. Como se puede ver en el ejemplo de la figura 6.9.

```
<h2>La entidad central <strong>{{nombreEntidad}}</strong>.</h2>
```

**Figura 6.9**: Ejemplo 1 de enlace de datos de una vía en AngularJS

- Enlace de datos de doble vía. En este segundo caso la información viaja desde el ámbito hacia la parte visual y también desde la parte visual hacia el ámbito. Cuando el modelo cambie, el dato que está escrito en un campo de texto se actualizaría automáticamente con el nuevo valor. Además, cuando el usuario cambie el valor del campo de texto, el dato en el ámbito también se actualizará automáticamente.
  Este enlace se implementa por medio de la directiva ngModel usada en las plantillas. La figura 6.10 muestra un ejemplo de como usar esta directiva.

[5] `http://mustache.github.io/`

```
<input data-ng-model="nEntidades" id="nEntidadesInput"
        type="number" min="1" max="20" step="1" required/>
```

**Figura 6.10**: Ejemplo de enlace de datos de doble vía en AngularJS

Por lo tanto, el enlace de datos es un proceso que tiende una conexión entre la vista y la lógica de negocio de la aplicación. A continuación se muestra un ejemplo un poco más complejo en las figuras 6.11 y 6.12.

```
<table data-ng-repeat="entidadCentral in listaEntidadesCentrales"
        id="entidad{{entidadCentral.id}}">
        <tr>
                <th colspan="2" title={{entidadCentral.nombre}}>{{entidadCentral.nombre}}</th>
        </tr>
        <tr data-ng-repeat="entidadSec in entidadCentral.dimensiones">
                <td class="nombre" title={{entidadSec.nombre}}>
                  {{entidadSec.nombre}}
                </td>
                <td class="nInstancias">{{entidadSec.num_instancias}}</td>
        </tr>
</table>
```

**Figura 6.11**: Plantilla del ejemplo 2 de enlace de datos en AngularJS

```
app.controller("Ctrl1", function($scope){
        $scope.listaEntidadesCentrales = null;
        $scope.getEntidades = function(val) {
                val = typeof val !== 'undefined' ? val : 10;
                $http.get("getEntidadesCentrales/" + val).then(function(res) {
                        var entidadesCentrales = [];
                        angular.forEach(res.data, function(item) {
                                entidadesCentrales.push(item);
                        });
                        $scope.listaEntidadesCentrales = entidadesCentrales;
                });
        };
        $scope.getEntidades(nEntidades);
});
```

**Figura 6.12**: Controlador del ejemplo 2 de enlace de datos en AngularJS

Los datos obtenidos del objeto JSON que devuelve la petición GET, estarán conectados con la vista de la figura 6.11. Y el número de elementos que se obtenga en dicho JSON no importará. Ya que será AngularJS en el que se encarga de actualizar la vista. Gracias al acceso a "*listaEntidadesCentrales*" y al uso de directivas en la plantilla HTML. A parte del propio tiempo de desarrollo que ahorra, es una limpieza de código destacable. Ya que no es necesario estar implementando eventos innecesarios, ni enviar datos de un sitio a otro, ni manipular el DOM manualmente. Tal como viene un objeto JSON de una llamada a un servicio web, se vincula al ámbito y automáticamente estará disponible en la vista. Esto es una ventaja notable en comparación con otros frameworks como BackboneJS.

### 6.3.3. Servicios y directivas

Se ha explicado que cada controlador tiene su ámbito propio y por lo tanto su modelo de datos. ¿Pero que pasa si se quiere transmitir datos de un controlador a otro? La primera opción que existe para pasar datos entre dos controladores es mediante parámetros en la URL. Esta es una buena alternativa si se pretenden pasar datos simples entre controladores. Pero cuanto es necesario pasar objetos más complejos empieza a ser poco práctico. Por lo tanto es necesario otra solución. Y para ello AngularJS propone los servicios. Los servicios son componentes que siguen el patrón singleton (clase con única instancia). Por lo que si dos controladores tienen una dependencia de un mismo servicio, en ambos se inyectará la misma instancia. Y podrá ser usada como "puente" para pasar la información. La figura 6.13 muestra un ejemplo de servicio.

```
var app = angular.module("MyApp", []);

app.factory("dataService", function() {
        return {
                data: {
                        nEntidades: 10,
                        idEntidadSeleccionada: -1
                }
        };
});

// controlador asignado a la ruta /view1
app.controller("Ctrl1", function($scope, $location, dataService) {
        $scope.goTo2 = function() {
                dataService.data.nEntidades = 12;
                dataService.data.idEntidadSeleccionada = 1238;
                $location.url("/view2");
        };
});

// controlador asignado a la ruta /view2
app.controller("Ctrl2", function($scope, dataService) {
        $scope.message = dataService.data.nEntidades + " a mostrar ";
});
```

**Figura 6.13**: Ejemplo de uso de un servicio en AngularJS

Por lo tanto, como durante todo el ciclo de vida se usa la misma instancia de cada servicio, se pueden usar como almacenes de datos globales disponibles para todos los controladores.

Por otro lado, están las directivas como componente importante de AngularJS. Es un tipo de componente que se usa en las plantillas HTML. Que cuando AngularJS lo encuentra altera el propio HTML. Ya que AngularJS preprocesa el HTML que se carga en el navegador. Por lo tanto cuando encuentra directivas definidas, generará el contenido HTML que marque su definición.

```
<table data-ng-repeat="entidadCentral in listaEntidadesCentrales"
</table>
```

**Figura 6.14**: Ejemplo de directiva definida por AngularJS

AngularJS tiene definidas multitud de directivas para que se pueda aplicar algo de lógica a las plantillas. En la figura 6.14 se puede ver la directiva *ng-repeat* definida por el framework. Y cuando se expusieron los componentes del patrón MVC de AngularJS, se mostró la directiva *ng-model* (ver sección 6.3.1). Sin embargo, es posible definir directivas

propias. Como se puede ver en el ejemplo de la figura 6.15.

```
app.directive('holaMundo', function() {
        return {
                restrict : 'E',
                template : "Hola Mundo, esto es una prueba!!!"
    }
});
```

**Figura 6.15**: Definición de una directiva propia

En toda directiva creada deben de aparecer dos elementos muy importantes:

- restrict. Declara cómo se va a poder usar esta directiva en el HTML. El carácter "E" indica que la directiva se va a usar como si fuera un elemento o *tag* nuevo de HTML. Pero se podría haber usado el valor "A" indicando que la directiva se utiliza como atributo. O "C" que indica que la directiva va a usarse como clase de un elemento HTML.
- template. Es la cadena que va a usar AngularJS para generar el contenido nuevo.

Ahora, si se quisiera aplicar la directiva definida en la figura 6.15 a una plantilla, se utilizaría la etiqueta HTML <hola-mundo>. Y antes de mostrar el resultado en el navegador, AngularJS sustituirá este elemento del DOM por "Hola Mundo, esto es una prueba!!!". Hay que tener en cuenta que AngularJS automáticamente traduce un nombre de directiva que esté en *camelCase* a la notación de palabras en minúscula separadas por guion medio.

## 6.4. Elección

Después de hacer pruebas con la librería jQuery y los frameworks Backbone y AngularJS, se llegó a la conclusión de que para realizar un desarrollo ágil y dentro del tiempo estimado era necesario:

- Desarrollar la aplicación utilizando un patrón MVC o MV*. Este último termino se debe a que tanto Backbone como AngularJS hacen reinterpretaciones del patrón MVC.
- Un sistema de plantillas potente integrado con el framework o librería a utilizar. jQuery y Backbone no tienen un motor de plantillas y se deja a elección del desarrollador el poder "acoplar" o no alguno.
- Características que simplificaran la continua modificación del DOM. Debido a los constantes cambios en los modelos de datos provocados por la interacción del usuario.

AngularJS es el único que proporciona todas estas características. También es muy importante la comunidad de desarrollo que hay alrededor del framework. En la cual se puede encontrar ayuda y nuevas extensiones de calidad. Además, AngularJS con el paso del tiempo se está convirtiendo en uno de los frameworks más importantes y usados para JavaScript. Tal como indica la comparación de la figura 6.16 en cuando a número de búsquedas (Google Trends[6]). Lo que indica que seguirá evolucionando y mejorando.

[6] `http://www.google.es/trends`

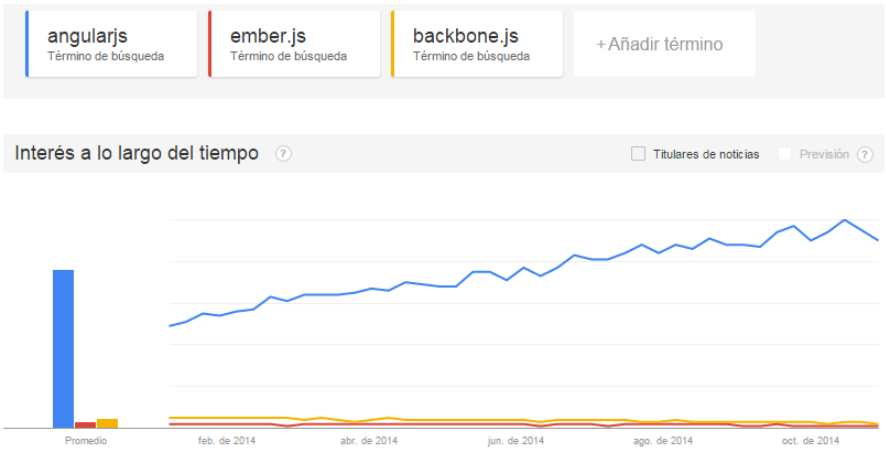



**Figura 6.16**: Búsquedas populares sobre distintos frameworks de JS en 2014

# Capítulo 7

# Resultados

A continuación se van a mostrar y describir cada una de las interfaces de las que está compuesta la aplicación web Linked Data Visual Query Builder.

## 7.1. Filtro opcional

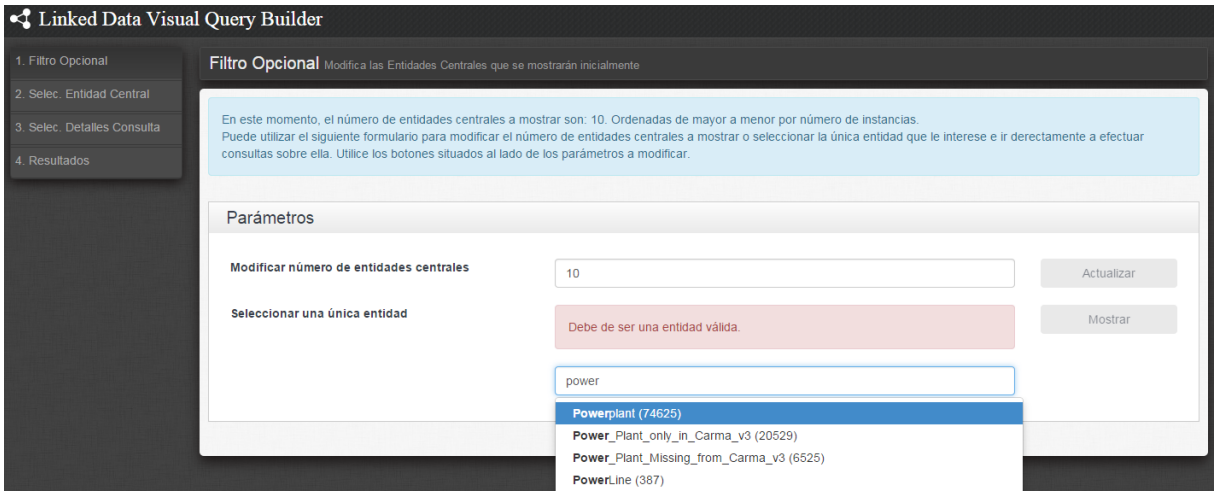



**Figura 7.1**: Interfaz filtro opcional de la aplicación

La interfaz inicial de la aplicación se muestra en la figura 7.1. Y es la de puerta de entrada a la creación de consultas visuales sobre una entidad central. La aplicación se ha construido con la idea que el usuario tiene que coger una entidad central presente en Enipedia. Para posteriormente hacer consultas sobre esta como eje central (ver sección 4.4.1). El usuario puede tener claro que entidad central quiere escoger para hacer consultas sobre ella. O puede que no sepa que entidad central va a escoger y quiera ver una visión de las más importantes. Con lo que es totalmente opcional modificar los dos parámetros que encontrará en esta primera interfaz. Ya que podría ir a la interfaz de selección de entidad central mediante el menú lateral.

Por lo tanto, el usuario tiene 2 opciones para empezar a usar la aplicación desde esta interfaz de entrada:

- Modificar el número de entidades centrales por defecto y presionar el botón *Actualizar*. Con lo que pasará a una segunda interfaz donde se mostrará el número de

entidades centrales elegidas y mostradas por orden de importancia. Esa importancia depende del número de instancias asociadas a cada entidad central. Además, cada entidad central aparecerá con sus 10 dimensiones más importantes.

- Introducir el nombre de una entidad central en el cuadro texto, que irá mostrando las entidades centrales disponibles que contengan los caracteres introducidos. Si tras elegir una entidad central válida el usuario presiona el botón *Mostrar*, irá directamente a la tercera interfaz donde podrá elegir los detalles necesarios para efectuar consultas sobre dicha entidad.

Como se ha comentado, el usuario puede que no modifique ningún parámetro y vaya directamente a la segunda interfaz mediante el menú lateral. Donde se muestran por defecto las 10 entidades centrales más relevantes.

## 7.2. Selección de entidades centrales

El usuario puede ir directamente a la interfaz de la figura 7.2 sin interactuar con la interfaz inicial de la aplicación. O puede que no pase por esta segunda interfaz si en la primera ha elegido la entidad central exacta con la quiere trabajar. Mediante el cuadro de texto y el botón *Mostrar*.

Independientemente de como el usuario haya accedido a esta interfaz de selección, el número de entidades centrales mostradas será 10. A no ser que haya modificado su número en la primera interfaz. Donde cada una de las entidades centrales aparecen junto con la lista de las 10 dimensiones más importantes. Ordenadas por número de instancias de estas. Aunque es posible que haya entidades centrales sin dimensiones.

Esta interfaz sirve para que el usuario elija la entidad central que quiere analizar. Mostrando información para que ayude a su elección e indicado que entidades son más importantes o contienen más datos para analizar.

Para seleccionar una entidad y pasar a la interfaz de los detalles de la consulta, simplemente habrá que pulsar el botón izquierdo de ratón sobre la entidad central elegida.

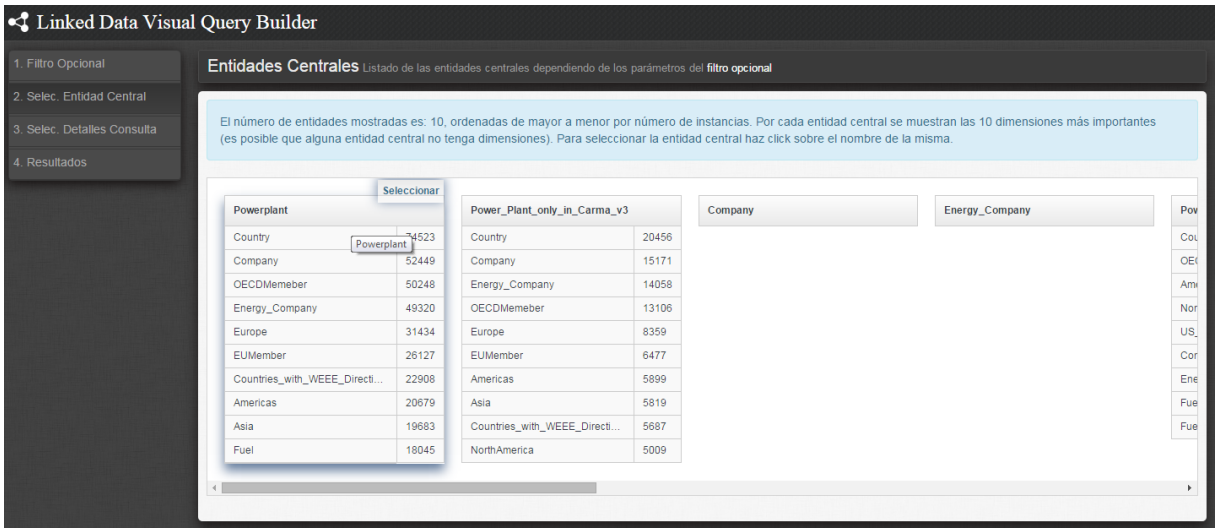



**Figura 7.2**: Interfaz seleccionar entidades centrales de la aplicación

## 7.3. Detalles consulta

Esta es la interfaz más importante de toda la aplicación. Ya que será en esta donde el usuario configurará la consulta que quiere realizar sobre la entidad central elegida anteriormente. Y para que el usuario tenga una visión estructurada del conjunto de datos relacionados con esta entidad central, se presenta la interfaz mostrando sus dimensiones y medidas en distintas tablas. Por lo que el usuario tendrá que elegir que medidas y dimensiones quiere analizar. Obteniendo de esta forma los resultados que se ajusten a los patrones de consulta creados.

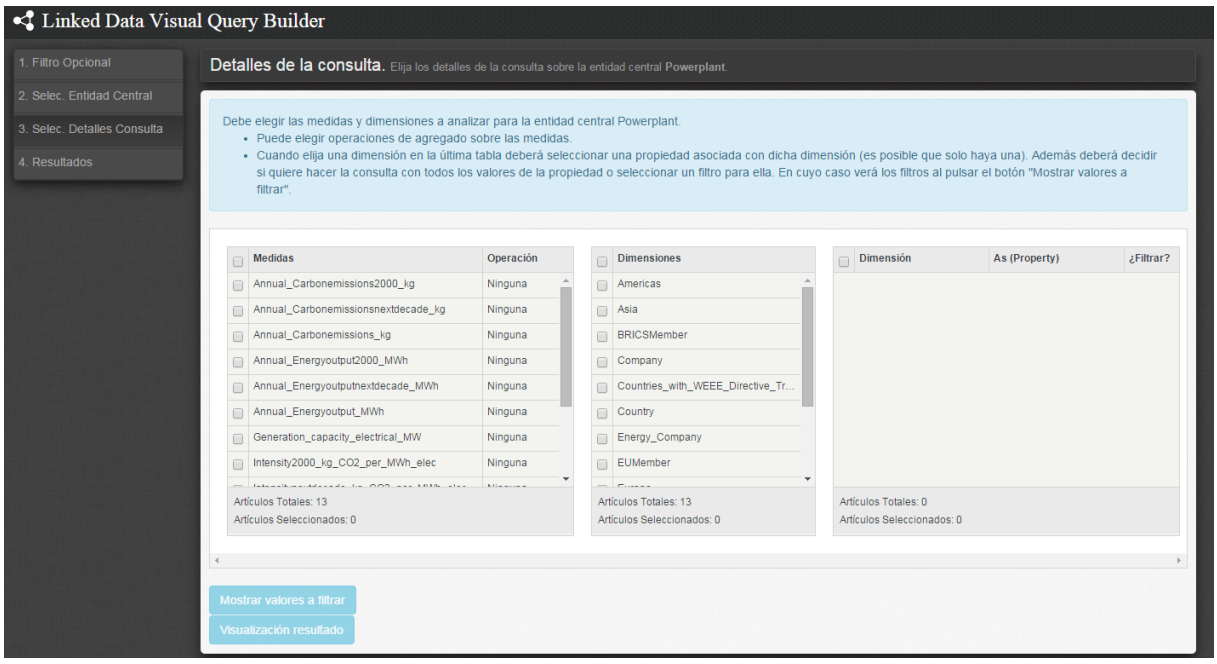



**Figura 7.3**: Interfaz detalles de consulta de la aplicación

Como se muestra en la figura 7.3, al entrar por primera vez en esta interfaz tras elegir la entidad central a analizar, el usuario verá pobladas las tablas de medidas y dimensiones. Según vaya eligiendo elementos de la tabla dimensiones, en la tercera tabla irán apareciendo elementos. Por cada dimensión seleccionada de la segunda tabla, en la tercera tabla aparecerán las propiedades asociadas a las dimensiones seleccionadas (ver sección 4.4.3).

El usuario puede elegir las propiedades que le interese de la tercera tabla y realizar directamente la consulta. Con lo que se estaría cogiendo todos los valores que toman las propiedades seleccionadas para el cálculo del resultado final. Pero también es posible que el usuario quiera restringir la consulta a ciertos valores de las propiedades seleccionadas. Y para eso el usuario tiene que elegir filtros en ellas.

Por lo tanto, además de la propiedad, el usuario tiene que elegir si va a restringir la consulta a ciertos valores o no. Pulsando el botón que hay al lado de cada propiedad y dejando el valor a “SI” o “NO”.

Tras elegir las medidas, dimensiones y propiedades e indicar que quiere filtro o no para dichas propiedades, el usuario tiene dos opciones marcadas por los dos botones que hay debajo de las tablas:

- Mostrar valores a filtrar (ver figura 7.4). Este botón solo se activará si el usuario ha elegido filtrar (valor “SI”) alguna propiedad. Con lo que se accederá a la lista de

valores que toma cada una de esas propiedades. Y el usuario podrá elegir los valores que solo se tendrán en cuenta en la consulta final.

- Visualizar Resultado. Con este botón se enviará la consulta final al punto de acceso de Enipedia y se redirigirá al usuario a la última interfaz. Donde tendrá la posibilidad de ver el resultado en diferentes formatos o incluso exportarlo. Evidentemente, si el usuario no ha elegido ningún valor a filtrar, la consulta final se realizará teniendo en cuenta todos los valores que toma la propiedad.

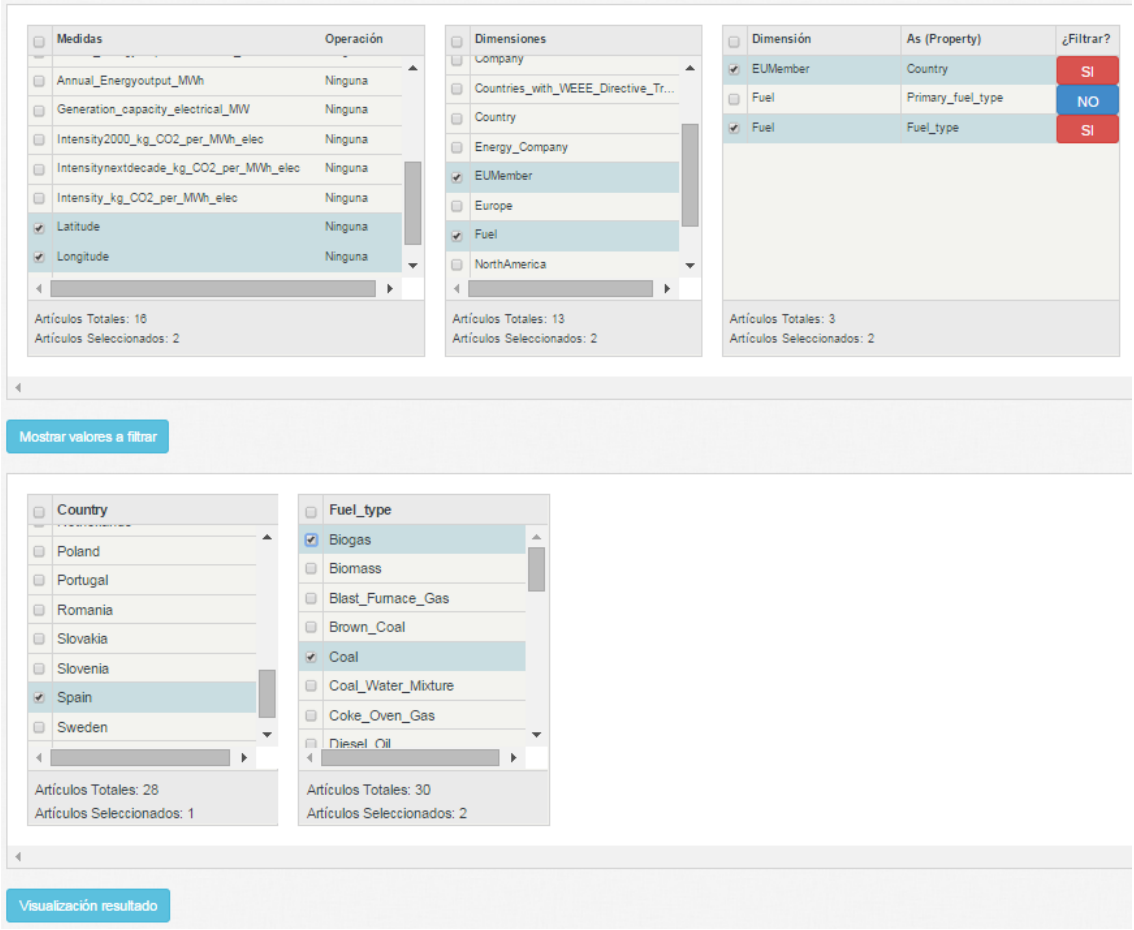



**Figura 7.4**: Ejemplo consulta con filtros en la aplicación

Para que el resultado de la consulta sea útil, es necesario seleccionar alguna medida. Pero además, sobre estas medidas es posible aplicar funciones de agregado. Que servirán para aplicar cálculos sobre un conjunto de resultados. Y para ello será necesario elegir una función de agregado del menú desplegable situado en la segunda columna de la tabla de medidas. La figura 7.5 muestra un ejemplo.


**Figura 7.5**: Ejemplo selección funciones de agregado en las medidas

## 7.4. Ver resultados

Esta última interfaz se puede ver en la figura 7.6. Y permitirá al usuario ver el resultado de la consulta final en diferentes formatos. Simplemente tiene que seleccionar el tipo de formato de visualización del primer menú desplegable. Eligiendo una de las 4 opciones posibles y seguidamente pulsar el botón "*Mostrar resultado*". Los formatos disponibles para visualización son 4: HTML, XML, JSON y CSV.

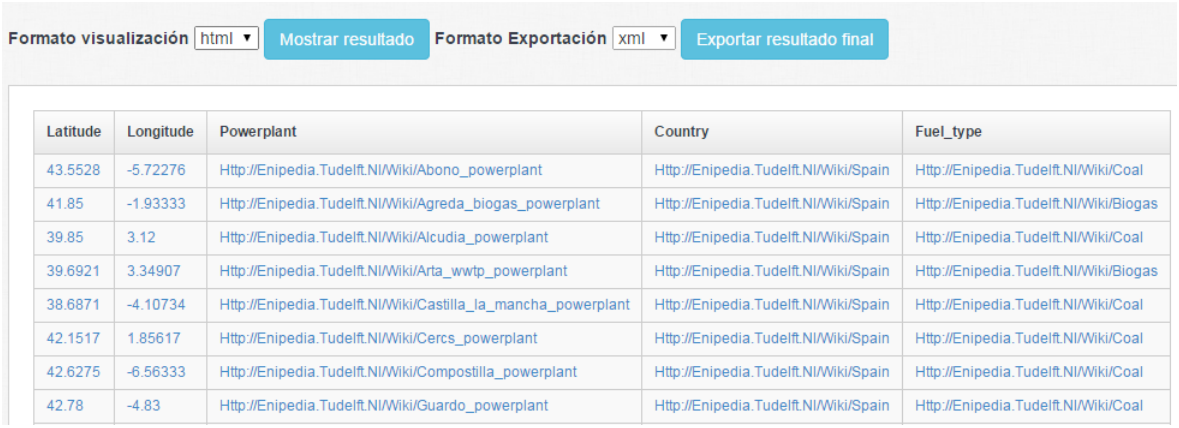



**Figura 7.6**: Ejemplo resultado en formato HTML

Además, si el usuario quiere generar un fichero con los resultados obtenidos, puede elegir entre los formatos XML, JSON y CSV. Simplemente deberá elegirlo del segundo menú desplegable y pulsar el botón "Exportar resultado final".

# Capítulo 8

# Conclusiones y trabajo futuro

## 8.1. Conclusiones

El uso de datos enlazados y la web semántica son tecnologías en constante movimiento y que despiertan interés de grandes instituciones y empresas. El uso de los datos abiertos enlazados (ver sección 2.2.5) proporciona datos accesibles públicamente y en un formato reutilizable. Pero además, proporciona un punto de acceso único a información en el que los datos están interconectados. Y donde es posible que estos datos puedan ser utilizados por sistemas de software de forma automatizada. Finalmente, la apertura y disponibilidad de estos datos a todo el mundo que quiera utilizarlos, puede crear nuevas oportunidades de negocio. Gracias al hecho de permitir a personas ajenas de los datos enlazados creados, el poder desarrollar nuevos servicios de valor añadido. Utilizando los datos abiertos enlazados de forma integrada.

El problema que se plantea a la hora de realizar consultas sobre los datos abiertos enlazados es el necesario conocimiento del lenguaje de consulta SPARQL (ver sección 2.2.4). Lo que complica el acceso a la información, para su posterior análisis, a personas que desconocen la utilización de este lenguaje.

Con la realización de la aplicación Linked Data Visual Query Builder, se ha comprobado que es posible realizar una interfaz que facilite la consulta sobre una gran base de datos abiertos enlazados. Sin tener que formular consultas directamente a un punto de acceso. Con el conocimiento de las distintas tecnologías semánticas que eso supone para poder acceder a la información. Para ello, la aplicación ha partido de una exploración (ver sección 4.3) de los datos tras el punto de acceso. Para transformar las relaciones conceptuales de esos datos enlazados a modelos de estrellas multidimensionales (ver sección 2.3.3). Consiguiendo desarrollar una interfaz sencilla que permite crear consultas básicas de una forma visual. A través de la interacción sobre tablas en dos dimensiones. Pobladas con los elementos de las estrellas multidimensionales. Y obteniendo un conjunto final de datos resultantes de la consulta. Con los cuales es posible hacer un análisis utilizando otro software independiente de la presente aplicación. Que puede ayudar a sacar conclusiones o incluso en la toma de decisiones. Todo a partir del acceso público que proporcionan los datos abiertos enlazados.

Este proyecto ha trabajado con los conceptos de modelos multidimensionales y los datos enlazados abiertos. Y su desarrollo se ha dividido en tres grandes bloques: inicialmente, y partiendo de [30], el primer problema se centraba en como modelar las estrellas y presentarlas al usuario de una forma simple y entendible. Para que pueda crear las consultas

a partir de ellas. Evidentemente, el segundo problema ha sido crear la interfaz. Y finalmente había que comunicarse con el punto de acceso para enviar las consultas, recibir los resultado y presentárselos al usuario.

## 8.2. Trabajo futuro

La aplicación desarrollada presenta una interfaz básica para formular consultas sobre el punto de acceso. Pero actualmente, las operaciones que se pueden realizar sobre las medidas se reducen a aplicar funciones de agregados sobre estas. Por lo que una mejora sería la posibilidad de poder aplicar filtros sobre dichas funciones de agregado (sentencia *HAVING* de SPARQL). Como por ejemplo, poder especificar que el máximo de una medida sea mayor, menor o igual a un valor numérico, o que esté entre varios valores...

Otro punto importante a tener en cuenta para un futura implementación, es el aspecto relacionado con la granularidad en las dimensiones. Sería de utilidad de cara al usuario, poder presentar las dimensiones de las estrellas multidimensionales de una forma jerarquizada. Lo cual implicaría realizar una modificación en el punto de partida. Ya que los patrones iniciales con los que se modeló el catálogo intermedio no contemplan relaciones jerárquicas (ver sección 4.3). Además, se debería de adaptar la interfaz actual para poder mostrar estas jerarquías.

Para ofrecer al usuario herramientas de análisis, se podría implementar un sistema de visualización de gráficas con los datos obtenidos por las consultas. Para ello, la herramienta idónea de visualización de gráficas en un cliente web es Sgvizler[1]. Mediante esta librería se puede visualizar los resultados de una consulta SPARQL a un punto de acceso (ver sección 3.3). Evidentemente no se trata de una implementación trivial. Ya que exige implementar un sistema donde el usuario pueda elegir la gráfica que quiere visualizar y adaptar la consulta final a los requisitos de dicha gráfica. Debido a que cada gráfica requiere unos parámetros necesarios.

Y finalmente, sería muy interesante que el usuario pudiera publicar y hacer accesibles esos datos estadísticos obtenidos. Partiendo de ficheros CSV obtenidos al exportar los resultados multidimensionales de las consultas. Y para ello sería necesario el uso de RDF Data Cube[2]. Además, existen herramientas de visualización de datos multidimensionales en formato de RDF Data Cube [24].

[1] `http://dev.data2000.no/sgvizler/`

[2] `http://www.w3.org/TR/2014/REC-vocab-data-cube-20140116/`

# Bibliografía


[1] Grigoris Antoniou and Frank van Harmelen. *A Semantic Web Primer, 2Nd Edition (Cooperative Information Systems)*. The MIT Press, 2 edition, 2008.

[2] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: A nucleus for a web of open data. In *The Semantic Web*, volume 4825 of *Lecture Notes in Computer Science*, pages 722–735. Springer Berlin Heidelberg, 2007.

[3] Dave Beckett. Rdf/xml syntax specification. `http://www.w3.org/TR/REC-rdf-syntax/`.

[4] Rafael Berlanga, Oscar Romero, Alkis Simitsis, Victoria Nebot, Torben Bach Pedersen, Alberto Abelló, and María José Aramburu. Semantic web technologies for business intelligence. *Business Intelligence Applications and the Web: Models, Systems, and Technologies*, 2012.

[5] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifiers (uri): Generic syntax. 1998.

[6] Tim Berners-Lee. Linked data - design issues, 2006. `http://www.w3.org/DesignIssues/LinkedData.html`.

[7] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001.

[8] Christian Bizer, Tom Heath, Kingsley Idehen, and Tim Berners-Lee. Linked data on the web (ldow2008). In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 1265–1266, New York, NY, USA, 2008. ACM.

[9] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In Ian Horrocks and James Hendler, editors, *The Semantic Web - ISWC 2002*, volume 2342 of *Lecture Notes in Computer Science*, pages 54–68. Springer Berlin Heidelberg, 2002.

[10] Pablo Castells. La web semántica. *Sistemas interactivos y colaborativos en la web*, pages 195–212, 2003.

[11] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and olap technology. *SIGMOD Rec.*, 26(1):65–74, March 1997.

[12] E. F. Codd, S. B. Codd, and C. T. Salley. Providing OLAP (On-Line Analytical Processing) to User-Analysis: An IT Mandate, 1993.

[13] Dan Connolly and Robert Cailliau. A little history of the world wide web. *World Wide Web Consortium*, 2000.

[14] Andy Seaborne Eric Prud'hommeaux. Sparql query language for rdf. `http://www.w3.org/TR/rdf-sparql-query/`.

[15] Guus Schreiber Fabien Gandon. Rdf 1.1 xml syntax. `http://www.w3.org/TR/rdf-syntax-grammar/`.

[16] J. Grant and D. Becket. Rdf test cases - n-triples. Technical report, W3C Recommendation, 2004.

[17] Jim Gray, Adam Bosworth, Andrew Layman, Don Reichart, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. pages 152–159, 1996.

[18] Marc Gyssens and Laks V. S. Lakshmanan. A foundation for multi-dimensional databases. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 106–115, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[19] Olaf Hartig, Christian Bizer, and Johann-Christoph Freytag. Executing sparql queries over the web of linked data. In *The Semantic Web - ISWC 2009*, volume 5823 of *Lecture Notes in Computer Science*, pages 293–309. Springer Berlin Heidelberg, 2009.

[20] Stefan Haustein and Jörg Pleumann. Is participation in the semantic web too difficult? In *In Proc. of 1st Int. Conf. on the Semantic Web (ISWC 2002*, pages 448–454. Springer, 2002.

[21] Tom Heath and Christian Bizer. *Linked Data: Evolving the Web into a Global Data Space*. Morgan & Claypool, 1st edition, 2011.

[22] Philipp Heim, Sebastian Hellmann, Jens Lehmann, Steffen Lohmann, and Timo Stegemann. Relfinder: Revealing relationships in rdf knowledge bases. In *Proceedings of the 4th International Conference on Semantic and Digital Media Technologies (SAMT 2009)*, pages 182–187, Berlin/Heidelberg, 2009. Springer.

[23] Philipp Heim, Jürgen Ziegler, and Steffen Lohmann. gFacet: A browser for the web of data. In *Proceedings of the International Workshop on Interacting with Multimedia Content in the Social Semantic Web (IMC-SSW 2008)*, volume 417 of *CEUR-WS*, pages 49–58, 2008.

[24] Jiří Helmich, Jakub Klímek, and Martin Nečaský. Visualizing rdf data cubes using the linked data visualization model. In *The Semantic Web: ESWC 2014 Satellite Events*, Lecture Notes in Computer Science, pages 368–373. Springer International Publishing, 2014.

[25] Ivan Herman. State of the semantic web. In *Semantic Days*, 2007.

[26] W. H. Inmon. *Building the Data Warehouse,3rd Edition*. John Wiley & Sons, Inc., New York, NY, USA, 3rd edition, 2002.

[27] Matthias Jarke, Y. Vassiliou, P. Vassiliadis, and M. Lenzerini. *Fundamentals of Data Warehouses*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1999.

[28] Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 2002.

[29] Victoria Nebot and Rafael Berlanga. Building data warehouses with semantic web data. *Decis. Support Syst.*, 52(4):853–868, March 2012.

[30] Victoria Nebot and Rafael Berlanga. Towards analytical md stars from linked data. 2014.

[31] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, September 2009.

[32] Bastian Quilitz and Ulf Leser. Querying distributed rdf data sources with sparql. In Sean Bechhofer, Manfred Hauswirth, Jörg Hoffmann, and Manolis Koubarakis, editors, *The Semantic Web: Research and Applications*, volume 5021 of *Lecture Notes in Computer Science*, pages 524–538. Springer Berlin Heidelberg, 2008.

[33] Leo Sauermann, Richard Cyganiak, and Max Völkel. Cool uris for the semantic web. Technical Memo TM-07-01, DFKI GmbH, Deutsches Forschungszentrum f ür Künstliche Intelligenz GmbH, February 2007. Written by 29.11.2006.

[34] Martin G. Skjæveland. Sgvizler: A javascript wrapper for easy visualization of sparql result sets, 2012. Accepted for publication in the workshop proceedings for ESWC 2012.

[35] R.M Sprague and E.D Carlson. *Building Effective Decision Support Systems*. Prentice Hall, Englewood Cliffs, 1982.

[36] Wil M. P. van der Aalst. Process cubes: Slicing, dicing, rolling up and drilling down event data for process mining. In Minseok Song, Moe Thandar Wynn, and Jianxun Liu, editors, *AP-BPM*, volume 159 of *Lecture Notes in Business Information Processing*, pages 1–22. Springer, 2013.

[37] Panos Vassiliadis and Timos Sellis. A survey of logical models for olap databases. *SIGMOD Rec.*, 28(4):64–69, December 1999.

[38] Liyang Yu. Linked open data. In *A Developerś Guide to the Semantic Web*, pages 409–466. Springer Berlin Heidelberg, 2011.