UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Escuela de Ingeniería Informática

Trabajo Final de Grado en Ingeniería Informática

SPARQL INTERPRETER

A component for Triskel Architecture

Autor Francisco Umpiérrez Rodríguez
Tutor José Juan Hernández Cabrera

Curso 2013/2014
Convocatoria Extraordinaria
Grado en Ingeniería Informática
Universidad de Las Palmas de Gran Canaria

AGRADECIMIENTOS

Antes de comenzar con el presente documento, quisiera mostrar mi agradecimiento a aquéllas personas que me han apoyado y animado durante la realización del mismo.

A José Juan Hernández no solo por ofrecerme la posibilidad de realizar el proyecto, sino también por su inestimable apoyo como tutor en el proyecto y su orientación durante la realización del mismo.

A Octavio Mayor por ofrecernos esa ayuda y asesoramiento incondicional como co-tutor que ha hecho posible el avance del proyecto, aunque la burocracia no le permitiese figurar como tal en este proyecto.

A Aitor Cardona, Cristian Casal, Xerach Hernández, David Santiago y Román Díaz, cuyo esfuerzo y dedicación han hecho posible el desarrollo del proyecto Triskel.

A todos mis amigos y familiares, los cuales han sido capaces de aguantarme durante la realización de este proyecto ofreciéndome siempre palabras de ánimo.

Finalmente, a todas esas personas que me han ayudado, motivado y animado, durante este recorrido.

A todos ellos…

Gracias.

RESUMEN

SPARQL Interpreter es uno de los cinco componentes de la Arquitectura Triskel, una arquitectura de software para una base de datos NoSQL que intenta aportar una solución al problema de Big Data en la web semántica. Este componente da solución al problema de la comunicación entre el lenguaje y el motor, interpretando las consultas que se realicen contra el almacenamiento en lenguaje SPARQL y generando una estructura de datos que los componentes inferiores puedan leer y ejecutar.

ABSTRACT

SPARQL Interpreter is one of the five components of Triskel Architecture, a software architecture for a NoSQL database that tries to solve the problem of Big Data in Semantic Web. This component gives solution to the problem of communication between the query language and the engine, interpreting the queries launched against the database in SPARQL and generating a data structure that other components can read and execute.

INDICE

1. MOTIVACIÓN

Este Trabajo Fin de Grado presentaba una gran oportunidad para poner en práctica todo lo aprendido durante mi formación como estudiante universitario. No solo planteaba el reto de diseñar una base de datos compleja, atendiendo a tecnologías modernas, sino que además, nos centraríamos en la parte menos trabajada durante mi formación.

Durante mis años de carrera universitaria, la ingeniería del software ha sido una materia que no ha estado tan presente como debiera. Con la posibilidad de elegirla como optativa en la antigua titulación de Ingeniería Técnica en Informática de Sistemas, era una asignatura cuya importancia no era apreciable por el alumno.

Así pues, cuando se presentó la oportunidad de enfrentar el reto de desarrollar una base de datos, atendiendo a conceptos y metodologías que nunca había estudiado, trabajando en un equipo serio de desarrollo bajo prácticas de programación y metodologías ágiles, no necesité más motivos para decidir que quería participar en el proyecto.

2. ESTADO ACTUAL

En el año 2000 nace la idea de desarrollar la Web Semántica, una red de documentos inteligentes que permitan búsquedas inteligentes, es decir, aumentar la inteligencia de los contenidos de las páginas web para dotarlas de un contenido semántico. Con lo cual, no solo se pudiesen obtener datos, sino también, interpretar el sentido de los mismos. Para apoyar los cimientos de esta idea, nacen diversas tecnologías tales como XML, RDF, OWL, etc.

Utilizando anotaciones RDF se pueden representar algunas facetas sobre conceptos de un dominio del conocimiento y se puede, mediante relaciones, crear una jerarquía de conocimiento. Una colección de sentencias RDF representa intrínsecamente un multigrafo dirigido y etiquetado. Como tal, un modelo de datos basado en RDF se adapta de forma más natural a cierto tipo de representación del conocimiento que el modelo relacional.

A posteriori SPARQL aparece como un lenguaje de consulta para navegar por el multigrafo que generaban las sentencias RDF y recuperar información del mismo, consolidándose como estándar y tecnología clave en el desarrollo de la Web Semántica.

Todo esto acaba encauzando la web semántica hasta un nuevo nivel, las bases de datos semánticas, que se basan en los motivos y tecnologías citadas anteriormente para construir una base de conocimiento con una semántica. Nacen todo tipo de bases de datos para almacenar las sentencias RDF (3store, 4store, RDFStore, etc.) y frameworks para manipular la información a este nuevo nivel semántico (Jena, Sesame, etc.).

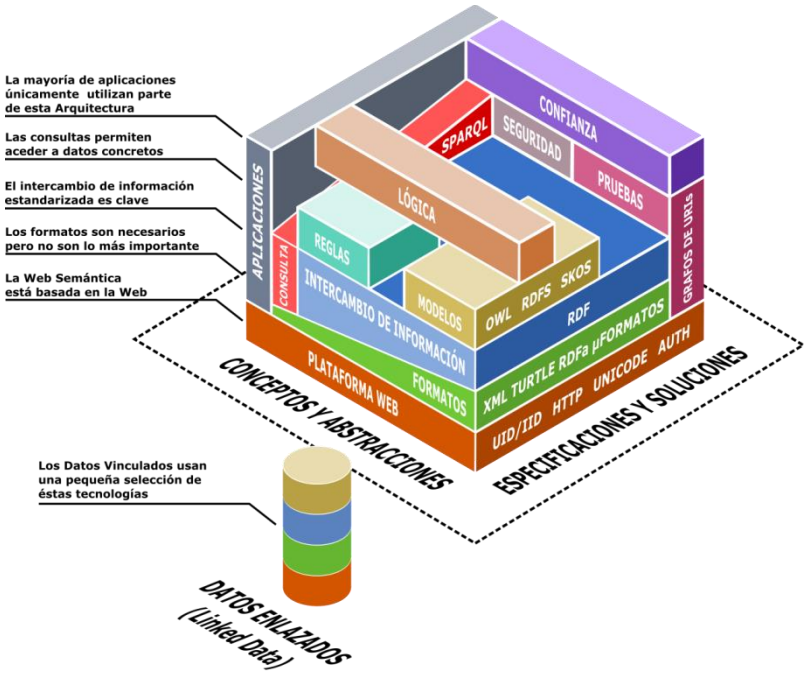

Figura 1. Componentes de la web semántica.

3. COMPETENCIAS

A continuación se relacionan las diferentes competencias cubiertas en la realización de este trabajo, indicando, para cada caso, en qué parte del documento se satisfacen.

IS01:

Capacidad para desarrollar, mantener y evaluar servicios y sistemas software que satisfagan todos los requisitos del usuario y se comporten de forma fiable y eficiente, sean asequibles de desarrollar y mantener y cumplan normas de calidad, aplicando las teorías, principios, métodos y prácticas de la ingeniería del software.

En el capítulo de Metodologías, se explican las diferentes metodologías y prácticas relacionadas con la ingeniería del software que se han utilizado para la realización del proyecto con el fin de alcanzar los objetivos propuestos. Especialmente la práctica de desarrollo basada en tests, o TDD (test-driven development) muestra como el seguimiento de esa práctica nos proporciona las capacidades que lista la competencia.

IS03:

Capacidad de dar solución a problemas de integración en función de las estrategias, estándares y tecnologías disponibles.

Esta competencia se desarrolla en el capítulo Cronología de Desarrollo, en la primera iteración. Donde integramos una librería software, ANTLR, que nos permite dar solución al problema de los analizadores sintáctico y léxicos.

IS04:

Capacidad de identificar y analizar problemas y diseñar, desarrollar, implementar, verificar y documentar soluciones software sobre la base de un conocimiento adecuado de las teorías, modelos y técnicas actuales.

El cumplimiento de esta competencia se puede observar en el capítulo de Cronología de Desarrollo, en el cual se presentan los problemas iniciales con los que nos encontramos

y las soluciones software que aportamos para resolver dichos problemas en base al conocimiento de las teorías, modelos y técnicas actuales en el desarrollo de software.

IS05:

Capacidad de identificar, evaluar y gestionar los riesgos potenciales asociados que pudieran presentarse.

Homólogamente a la competencia anterior, ésta se resuelve también en el capítulo Cronología de Desarrollo, en el cual identificamos y evaluamos los riesgos o problemas potenciales que se nos presentan o pueden presentarse.

SI03:

Capacidad para participar activamente en la especificación, diseño, implementación y mantenimiento de los sistemas de información y comunicación.

Esta competencia es desarrollada a lo largo del proyecto dada la metodología de desarrollo utilizada. En el capítulo Metodologías, explicamos en qué se basa la metodología SCRUM, una metodología que opta por una participación activa de todos los miembros del grupo de desarrollo en el desarrollo del proyecto.

4. OBJETIVOS

A continuación se plantean los objetivos que persigue la Arquitectura Triskel, así como los objetivos del componente a desarrollar en el presente Trabajo Fin de Grado.

4.1 OBJETIVOS DE TRISKEL

En las bases de datos tradicionales la información está vinculada a las relaciones de la misma, de modo que si se quiere realizar algún cambio de las relaciones de una base de datos, se debe realizar un costoso proceso de adaptación de la información a las nuevas relaciones que se definan. El proyecto Triskel nace como idea para solventar este problema y crear una base de datos NoSQL en la que la información está totalmente desvinculada de las relaciones entre la misma.

Este proyecto, tiene como objetivo crear una base de datos RDF que fuese capaz de almacenar grandes cantidades de datos, superando las dificultades habituales en la creación de bases de datos tales como el almacenamiento y la búsqueda.

La arquitectura Triskel, se dividió en cinco componentes básicos:

- SPARQL Interpreter
- QueryPlayer
- TermStore
- TripleStore
- Builder

Cada uno de estos componentes afronta un problema de la base de datos y aporta una propuesta para resolverlo. En este documento, hablaremos del componente SPARQL Interpreter y de cómo este afronta el problema de la comunicación entre el lenguaje de consulta y la base de datos.

4.2 OBJETIVOS DEL COMPONENTE

La finalidad del trabajo es el desarrollo de un intérprete de SPARQL que permita a un usuario realizar consultas contra un almacenamiento basado en tripletas. Dicho intérprete contemplará no solo el análisis de la consulta, sino la implementación de una estructura de operaciones que permita la ejecución de la query, así como la construcción de un objeto resultado manejable.

Para ello, nos hemos apoyado en las librerías de generación de analizadores sintácticos (ANTLR). Esto nos permite obtener analizadores sintácticos y léxicos, basados en una gramática bien construida, que nos servirán para poder implementar un intérprete semántico que sea capaz de resolver las consultas.

Dicho intérprete deberá generar una estructura que sea capaz de contener las operaciones que debe desarrollar una consulta. Ésta estructura será usada por componentes inferiores en la arquitectura más cercanos al almacenamiento.

Además se implementará un objeto resultante, que no solo sea manejable sino que además almacene los resultados de la consulta, devolviendo éste al terminar la ejecución de la query.

5. FUNDAMENTOS TEÓRICOS

Para poder profundizar más en el documento es necesario definir una serie de conceptos claves que permitan la correcta comprensión de las diferentes decisiones y direcciones por las ido transcurriendo el desarrollo del proyecto.

- **SPARQL**: SPARQL es un acrónimo recursivo del inglés SPARQL Protocol and RDF Query Language. Se trata de un lenguaje estandarizado para la consulta de grafos RDF y normalizado por el RDF Data Access Working Group (DAWG) del World Wide Web Consortium (W3C). Al igual que sucede con SQL, es necesario distinguir entre el lenguaje de consulta y el motor para el almacenamiento y recuperación de los datos. Por este motivo, existen múltiples implementaciones de SPARQL, generalmente ligados a entornos de desarrollo y plataformas tecnológicas.

  En un principio SPARQL únicamente incorpora funciones para la recuperación de sentencias RDF. Sin embargo, algunas propuestas también incluyen operaciones para el mantenimiento (creación, modificación y borrado) de datos.

- **RDF**: Es un modelo de datos similar a los enfoques de modelado conceptual clásicos como entidad-relación o diagramas de clases, ya que se basa en la idea de hacer declaraciones sobre los recursos a modo de expresiones sujeto-predicado-objeto. RDF se compone de 3 conceptos principales:
  - *Recurso*: Todas las cosas que son descritas por expresiones RDF se denominan recursos. Un recurso puede ser una página web entera, puede ser una parte de una página web, o también ser toda una colección de páginas. Los recursos son siempre nombrados por URI.
  - *Propiedades*: Una propiedad es un aspecto específico, característica, atributo o relación que se utiliza para describir un recurso. Cada propiedad tiene un significado específico, define sus valores permitidos, los tipos de recursos que puede describir, y su relación con otras propiedades.

- *Sentencias*: Son recursos específicos junto con una propiedad más el valor de la misma. Las sentencias son las construcciones básicas que establecen el modelo RDF. El significado de los datos se expresa mediante un conjunto de sentencias que son representadas por tripletas (sujeto, predicado, objeto).

- **Uri**: Un identificador uniforme de recursos (URI) es una cadena de caracteres que se utilizan para identificar el nombre de un recurso web. Esta identificación permite la interacción con las representaciones del recurso web en una red, por lo general de la World Wide Web, a través de protocolos específicos. Esquemas que especifican una sintaxis concreta y protocolos asociados definen cada URI.

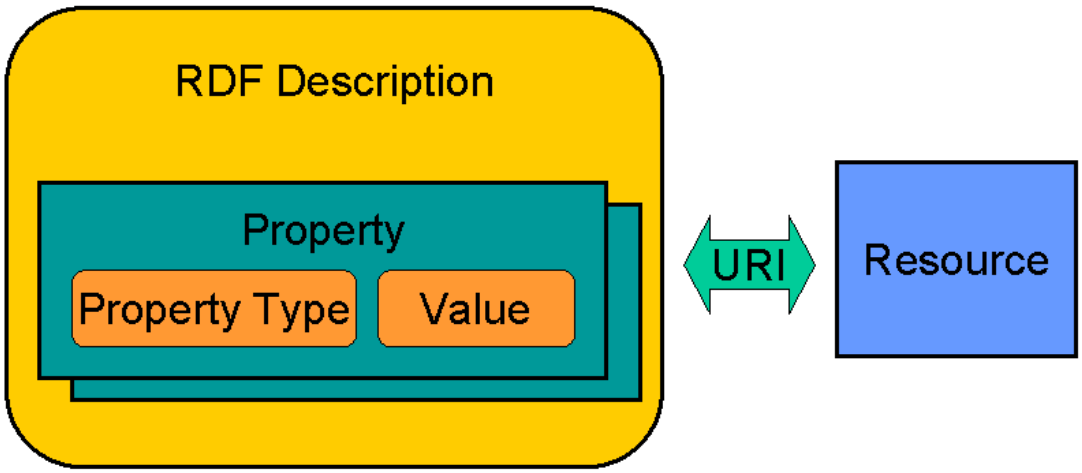

Figura 2. Cualquier recurso puede describirse mediante rdf siempre que éste posea una URI que lo identifique.

- **Tripleta**: Las tripletas son las sentencias que establecen el modelo RDF, se componen de tres elementos identificados como sujeto-predicado-objeto. El sujeto indica el recurso y el predicado denota rasgos o aspectos del recurso y expresa una relación entre el sujeto y el objeto. Por ejemplo, una forma de representar la idea de "El cielo tiene el color azul" en RDF es como un objeto que denota "el cielo", un predicado que denota "tiene el color" y un objeto que denota "azul". Por lo tanto RDF cambia objeto por sujeto que se utilizaría en la notación clásica de un modelo entidad-atributo-valor en diseño orientado a objetos, objeto (el cielo), atributo (color) y el valor (azul).

- **Patrón**: Un patrón es la forma de representar una tripleta como una lista separada por espacios de: sujeto, predicado y objeto. Un patrón tiene diferentes formas para abreviar las sentencias RDF juntando una o más sentencias en una única forma abreviada. Los patrones son elementos básicos en el lenguaje de consulta SPARQL ya que no son solo elementos de relación, sino también criterios de búsqueda.

6. METODOLOGÍA

Uno de los objetivos importantes de este proyecto es la metodología de desarrollo utilizada. En el planteamiento del proyecto, se consideró la posibilidad de utilizar metodologías de desarrollo ágil, más específicamente SCRUM, para llevar el proyecto a cabo, y así tener una experiencia de desarrollo real utilizando esa metodología.

El objeto de utilizar es metodología es aprovechar la capacidad de trabajo en grupo de todos los integrantes del proyecto, gestionándose como un equipo de desarrolladores mediante el modelo SCRUM.

Complementario a la metodología de desarrollo, se han decidido utilizar diversas prácticas de programación modernas tales como TDD (Test-Driven Development) o Programación en parejas (Pair programming). El fin de utilizar estas prácticas complementarias era generar un código de mayor calidad.

6.1 SCRUM

Scrum es un proceso en el que se aplican de manera regular un conjunto de buenas prácticas para trabajar colaborativamente, en equipo, y obtener el mejor resultado posible de un proyecto. Estas prácticas se apoyan unas a otras y su selección tiene origen en un estudio de la manera de trabajar de equipos altamente productivos.

En Scrum se realizan entregas parciales y regulares del producto final, priorizadas por el beneficio que aportan al receptor del proyecto. Por ello, Scrum está especialmente indicado para proyectos en entornos complejos, donde se necesita obtener resultados pronto, donde los requisitos son cambiantes o poco definidos, donde la innovación, la competitividad, la flexibilidad y la productividad son fundamentales.

6.1.1 El proceso

Un proyecto se ejecuta en bloques temporales cortos y fijos (iteraciones de un mes natural y hasta de dos semanas, si así se necesita). Cada iteración tiene que proporcionar un resultado completo, un incremento de producto final que sea susceptible de ser entregado.

El proceso parte de la lista de objetivos/requisitos priorizada del producto, que actúa como plan del proyecto. En esta lista se prioriza los objetivos balanceando el valor que aportan respecto a su coste y quedan repartidos en iteraciones y entregas.

Las actividades que se llevan a cabo en Scrum son las siguientes:

6.1.2 Planificación de la iteración

El primer día de la iteración se realiza la reunión de planificación de la iteración. Tiene dos partes:

Selección de requisitos (4 horas máximo). se presenta al equipo la lista de requisitos priorizada del producto o proyecto. El equipo pregunta las dudas que surgen y selecciona los requisitos más prioritarios que se compromete a completar en la iteración, de manera que puedan ser entregados.

Planificación de la iteración (4 horas máximo). El equipo elabora la lista de tareas de la iteración necesarias para desarrollar los requisitos a que se ha comprometido. La estimación de esfuerzo se hace de manera conjunta y los miembros del equipo se autoasignan las tareas.

6.1.3 Ejecución de la iteración

Cada día el equipo realiza una reunión de sincronización (15 minutos máximo). Cada miembro del equipo inspecciona el trabajo que el resto está realizando (dependencias entre tareas, progreso hacia el objetivo de la iteración, obstáculos que pueden impedir este objetivo) para poder hacer las adaptaciones necesarias que permitan cumplir con el compromiso adquirido. En la reunión cada miembro del equipo responde a tres preguntas:

- ¿Qué he hecho desde la última reunión de sincronización?
- ¿Qué voy a hacer a partir de este momento?
- ¿Qué impedimentos tengo o voy a tener?

Durante la iteración el Scrum Master es un rol que se encarga de que el equipo pueda cumplir con su compromiso y de que no se merme su productividad eliminando los obstáculos que el equipo no puede resolver por sí mismo.

6.1.4 Inspección y adaptación

El último día de la iteración se realiza la reunión de revisión de la iteración. Tiene dos partes:

Demostración (4 horas máximo). El equipo presenta los requisitos completados en la iteración, en forma de incremento de producto preparado para ser entregado con el mínimo esfuerzo. En función de los resultados mostrados y de los cambios que haya habido en el contexto del proyecto, se realizan las adaptaciones necesarias de manera objetiva, ya desde la primera iteración, replanificando el proyecto.

Retrospectiva (4 horas máximo). El equipo analiza cómo ha sido su manera de trabajar y cuáles son los problemas que podrían impedirle progresar adecuadamente, mejorando de manera continua su productividad. El Scrum Master se encargará de ir eliminando los obstáculos identificados.

6.2 TDD

Desarrollo guiado por pruebas de software, o Test-driven development (TDD) es una práctica de programación que involucra otras dos prácticas: Escribir las pruebas primero (Test First Development) y Refactorización (Refactoring). Para escribir las pruebas generalmente se utilizan las pruebas unitarias (unit test en inglés). En primer lugar, se escribe una prueba y se verifica que las pruebas fallan. A continuación, se implementa el código que hace que la prueba pase satisfactoriamente y seguidamente se refactoriza el código escrito. El propósito del desarrollo guiado por pruebas es lograr un código limpio que funcione. La idea es que los requisitos sean traducidos a pruebas, de este modo, cuando las pruebas pasen se garantizará que el software cumple con los requisitos que se han establecido. De forma resumida, esta práctica se estructura en dos fases:

- En primer lugar, diseñamos y escribimos los casos de prueba y las pruebas unitarias en base a requisitos del software y a la arquitectura del proyecto.
- En segundo lugar, codificamos la aplicación de tal manera que el código implementado cumpla los test diseñados previamente.

Una vez los test pasan se revisa el código escrito y se realiza un proceso de Refactorización, es decir, una modificación del código fuente sin cambiar su comportamiento, lo que se conoce informalmente por limpiar el código. La refactorización se realiza a menudo como parte del proceso de desarrollo del software: los desarrolladores alternan la inserción de nuevas funcionalidades y casos de prueba con la refactorización del código para mejorar su consistencia interna y su claridad. Los test aseguran que la refactorización no cambia el comportamiento del código.

El objetivo es mejorar la facilidad de comprensión del código o cambiar su estructura y diseño y eliminar código muerto, para facilitar el mantenimiento en el futuro. Añadir nuevo comportamiento a un programa puede ser difícil con la estructura dada del programa, así que un desarrollador puede refactorizarlo primero para facilitar esta tarea y luego añadir el nuevo comportamiento.

El propósito del desarrollo guiado por pruebas es lograr un código que funcione y sea seguro ante los cambios. La clave es partir de los requisitos. Estos deben ser traducidos a test unitarios de tal forma que cuando todas las pruebas superen los requisitos y todas sus derivadas, se podrá asegurar que la aplicación va a funcionar correctamente.

En la práctica, el diseño de test unitarios es un proceso que va a convivir durante todo el ciclo de vida de un producto, desde su diseño inicial hasta la fase de explotación, con el mantenimiento de la misma.

6.3 PAIR PROGRAMMING

La Programación en Pareja (o Pair Programming en inglés) es una práctica de programación que requiere que dos programadores participen en un esfuerzo combinado de desarrollo en un sitio de trabajo. Cada miembro realiza una acción que el otro no está haciendo actualmente: Mientras que uno codifica las pruebas de unidades el otro piensa en la clase que satisfará la prueba, por ejemplo.

La persona que está haciendo la codificación se le da el nombre de controlador mientras que a la persona que está dirigiendo se le llama el navegador. Se sugiere a menudo para que a los dos socios cambien de papeles por lo menos cada media hora o después de que se haga una prueba de unidad.

Se definen las siguientes ventajas para la aplicación de esta práctica de programación:

- Más Disciplina. Emparejando correctamente es más probable que hagan "lo que se debe hacer" en lugar de tomar largos descansos.
- Mejor código. Emparejando similares es menos probable producir malos diseños ya que su inmersión tiende a diseñar con mayor calidad.
- Flujo de trabajo constante. El emparejamiento produce un flujo de trabajo distinto al trabajar solo. En pareja el flujo de trabajo se recupera más rápidamente: un programador pregunta al otro "¿por dónde quedamos?". Las parejas son más resistentes a las interrupciones ya que un desarrollador se ocupa de la interrupción mientras el otro continúa trabajando.
- Múltiples desarrolladores contribuyen al diseño. Si las parejas rotan con frecuencia en el proyecto significa que más personas están involucradas con una característica en particular. Esto ayuda a crear mejores soluciones, especialmente cuando una pareja no puede resolver un problema difícil.
- Moral mejorada. La programación en parejas es más agradable para algunos programadores, que programar solos.
- Propiedad colectiva del código. Cuando el proyecto se hace en parejas, y las parejas se rotan con frecuencia, todos tienen un conocimiento del código base.
- Enseñanza. Todos, hasta los novatos, poseen conocimientos que los otros no. La programación en pareja es una forma amena de compartir conocimientos.
- Cohesión de equipo. La gente se familiariza más rápidamente cuando programa en pareja. La programación en pareja puede animar el sentimiento de equipo.
- Pocas interrupciones. La gente es más renuente a interrumpir a una pareja que a una persona que trabaja sola.
- Menos estaciones de trabajo. Ya que dos personas van a trabajar en una estación de trabajo, se requieren menos estaciones de trabajo, y las estaciones extras pueden ser ocupadas para otros propósitos.

7. PLAN DE TRABAJO

El plan de trabajo a seguir en el desarrollo del proyecto se ha estructurado en cuatro grandes etapas:

- Estado del arte y contextualización: desarrollo de un conocimiento sobre los términos asociados al tema de trabajo del proyecto, así como una evaluación de lo que existe actualmente y cómo influye eso en el proyecto que se pretende realizar, así como la influencia del mismo en el estado actual de la tecnología.
- Estudio de herramientas: estudio de las diferentes herramientas utilizadas a lo largo de desarrollo.
- Documentación: confección del documento de trabajo fin de grado a partir de toda la documentación, bibliografía y conocimiento reunidos a lo largo de las diferentes etapas del proceso, así como la preparación y dedicación a la defensa del mismo.
- Implementación: desarrollo del componente.

Una estimación aproximada del tiempo invertido en el desarrollo del proyecto, utilizando de referencia las tareas descritas para cada etapa, sería:

Etapa	Estimación (en horas)
Estado del arte y contextualización	25
Estudio de herramientas	15
Documentación	60
Implementación	200
Dedicación total al proyecto	300

En cuanto a la etapa de Implementación se ha divido en cuatro iteraciones globales, enfocando cada una de las iteraciones a un problema concreto de implementación. Dichas iteraciones son las que se enumeran a continuación:

- Implementación de una estructura arbórea de reconocimiento del lenguaje de consulta.
- Implementación de una estructura de almacenamiento de resultados.
- Implementación de una estructura de ejecución de operaciones.
- Implementación de un controlador que unifique y use las estructuras diseñadas.

Durante cada una de las iteraciones, se llevará a cabo la metodología SCRUM explicada en el apartado de Metodología adaptándola a nuestro entorno de desarrollo. Una de las ventajas de SCRUM es que permite la adaptación de la metodología a diferentes entornos de trabajo al ser un compendio de buenas prácticas. En SCRUM se recomienda coger aquéllas prácticas que sirvan al equipo de desarrollo y rechazar aquéllas que lo ralenticen.

7.1 APLICACIÓN DEL SCRUM

Definimos una semana como iteración o sprint, por lo que, al comienzo de cada semana nos reuníamos con el SCRUM Master, en nuestro caso el tutor, y se planteaban los requisitos y objetivos que se querían cumplir en dicho tiempo. Una vez teníamos los objetivos del sprint, el equipo de desarrollo se reunía y se estimaban las horas de duración de cada requisito, así como la asignación de los responsables que lo llevarían a cabo. Durante el sprint, el equipo a menudo se reunía y ponía en común los problemas que encontraban o los avances realizados hasta el momento.

En la siguiente reunión con el SCRUM Master, se comentaban los problemas encontrados durante la iteración, los resultados obtenidos y todo tema que pudiese resultar de interés. Asimismo, una vez tratados los asuntos de la iteración, se planteaban los siguientes requisitos para el nuevo sprint.

En esta aplicación del SCRUM se descartó por completo el uso de ninguna métrica ágil, teniendo en cuenta que realizar dichas métricas resultaba costoso y no suponía un valor para el equipo. Por lo que no se aplicaron métricas de tareas, tales como la tabla de burnout, u otras tablas de seguimiento que recomienda la metodología de desarrollo de SCRUM.

Por otro lado, las prácticas de programación mencionadas en el apartado de Metodología se llevaron a cabo de forma complementaria a la metodología SCRUM.

8. REQUISITOS

Durante el desarrollo del Trabajo Fin de Grado se ha hecho uso de una serie de recursos, hardware, software y otros que han hecho posible la realización del mismo. En este apartado enumeramos dichos recursos.

8.1 REQUISITOS SOFTWARE

Todos los recursos software utilizados son compatibles con distribuciones Windows. En concreto, se ha utilizado durante la realización del Trabajo Fin de Grado la siguiente distribución Windows:

- **Sistema Operativo**: Windows 7 Enterprise Service Pack 1

8.1.1 Librerías, componentes y frameworks

Aparte del sistema operativo y lo implementado en el proyecto, nos hemos apoyado en las siguientes librerías, componentes y frameworks:

- **ANTLR**: ANTLR (ANother Tool for Language Recognition; en español "otra herramienta para reconocimiento de lenguajes") es una herramienta creada principalmente por Terence Parr, que opera sobre lenguajes, proporcionando un marco para construir reconocedores (parsers), intérpretes, compiladores y traductores de lenguajes a partir de las descripciones gramaticales de los mismos (conteniendo acciones semánticas a realizarse en varios lenguajes de programación).
- **JUnit**: JUnit es un conjunto de clases (framework) que permite realizar la ejecución de clases Java de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera. Es decir, en función de algún valor de entrada se evalúa el valor de retorno esperado; si la clase cumple con la especificación, entonces JUnit devolverá que el método de la clase pasó exitosamente la prueba; en caso de que el valor esperado sea diferente al que regresó el método durante la ejecución, JUnit devolverá un fallo en el método correspondiente.

JUnit es también un medio de controlar las pruebas de regresión, necesarias cuando una parte del código ha sido modificado y se desea ver que el nuevo código cumple con los requerimientos anteriores y que no se ha alterado su funcionalidad después de la nueva modificación.

- **TermStore**: componente de la Arquitectura Triskel que contiene los diccionarios de términos y que permite al SPARQL Interpreter traducir los términos a IDs comprensibles para el almacenamiento.
- **Query Player**: componente de la Arquitectura Triskel que se encarga de ejecutar los patrones y las funciones reconocidos por el intérprete y almacenados en la estructura de ejecución de operaciones.

8.1.2 Entornos de desarrollo software y lenguajes de programación

Asimismo, para la implementación y gestión del código fuente del Trabajo Fin de Grado se han utilizado los siguientes recursos y lenguajes:

- **NetBeans**: NetBeans es un entorno de desarrollo integrado libre, hecho principalmente para el lenguaje de programación Java. NetBeans IDE es un producto libre y gratuito sin restricciones de uso. En concreto se ha utilizado la versión 7.4 de Netbeans.
- **Git**: Git es un software de control de versiones pensado para la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando estas tienen un gran número de archivos de código fuente. Netbeans proporciona soporte para el uso de la herramienta Git lo cual nos permite realizar el control de versiones directamente desde el IDE.
- **Bitbucket**: Bitbucket es un servicio de alojamiento basado en web, para los proyectos que utilizan el sistema de control de revisiones Mercurial y Git. Este servicio permite alojar el control de versiones en un repositorio privado. Es un servicio cuya cuenta gratuita ofrece los requisitos mínimos que se necesitaban para el proyecto.
- **Java**: lenguaje de programación utilizado durante la realización del Trabajo Fin de Grado.

8.2 OTROS REQUISITOS

Por último, se listan los requisitos, no software, que han estado presente durante el proyecto.

- **Ordenador**: ordenador con un procesador Intel Core i3 a 3.30GHz con una memoria ram de 4GB.
- **Zona de desarrollo**: zona habilitada para el desarrollo del proyecto, lo que permitía que todo el grupo trabajara junto, que se pudieran realizar las reuniones y mantener la tabla de tareas de SCRUM.
- **Útiles de SCRUM**: diversas utilidades para llevar a cabo la metodología de SCRUM entre las que se incluyen pósits para anotar los requisitos del sprint, una tabla de SCRUM (o SCRUM board), para colocar los pósits en sus respectivos estados de desarrollo, etc.

9. CRONOLOGÍA DE DESARROLLO

Como ya se ha comentado en apartados anteriores la implementación del Trabajo Fin de Grado se dividió en cuatro iteraciones globales, donde cada una de ellas representa un problema concreto del componente a desarrollar.

En este apartado, detallaremos el análisis del problema de cada una de las iteraciones globales y la propuesta implementada para su resolución.

9.1 IMPLEMENTACIÓN DE UNA ESTRUCTURA DE RECONOCIMIENTO DEL LENGUAJE DE CONSULTA

9.1.1 Análisis

Para construir un intérprete de un lenguaje de consulta debemos tener en cuenta tres aspectos básicos: análisis léxico, análisis sintáctico y análisis semántico. En esta primera etapa, nos centraremos en el análisis léxico y el análisis sintáctico.

Definimos el léxico como el conjunto de palabras que forman el lenguaje, por lo que, un analizador léxico es la primera fase del análisis que nos permite, a partir del texto de entrada, reconocer dichas palabras del lenguaje y generar una serie de tokens o símbolos que nos sirvan para los subsiguientes análisis.

Por otro lado, la sintaxis se encarga de definir las reglas y formas en que se combinan las palabras. Así pues se nutre directamente del resultado del análisis léxico para comprobar dichas reglas y formas.

Una vez el análisis sintáctico haya finalizado, no solo habremos comprobado si el texto de entrada cumple con el lenguaje SPARQL, sino que además, poseeremos una estructura con los tokens encontrados en dicha entrada.

Esta primera etapa engloba las decisiones y desarrollo relacionado con los analizadores explicados anteriormente.

9.1.2 Diseño

Uno de los principales problemas encontrados en esta primera etapa era el coste de desarrollo que supondría la implementación de los analizadores léxico y sintáctico. Dichos componentes, aunque necesarios, no eran una parte fundamental del proyecto pues nuestra intención es centrarnos en la resolución de consultas.

Por lo que, durante los sprints iniciales de esta iteración se planteó como objetivo documentarse sobre herramientas alternativas que nos permitiesen desarrollar los analizadores de forma rápida.

Los generadores de parsers (analizadores sintácticos) automáticos son poderosas herramientas que permiten generar, a partir de una gramática los analizadores léxicos y sintácticos. Con lo cual, destacaban como herramientas perfectas para solucionar nuestro primer problema de implementación.

Existen multitud de generadores de parsers que solucionaban aparentemente nuestra inicial necesidad de generación automática de los analizadores anteriormente mencionados. Por lo que, teniendo un extenso catálogo de opciones, comenzamos a evaluar los criterios que nos facilitarían elegir entre uno frente a los demás.

Por un lado, se requería definir criterios en base a las limitaciones que se poseían debido al lenguaje y al tipo de proyecto que se estaba realizando:

- Uno de los principales criterios de selección era el requisito de la limitación del lenguaje. Dado que nuestro lenguaje de implementación es Java, se necesitaba un generador de parsers que respetase el código en el que se desarrolla.
- Otra limitación es el tipo de licencia de la herramienta. Puesto que se está desarrollando un proyecto con vistas a ser software libre no se podía seleccionar aquéllas herramientas con licencias restrictivas.

Una vez seleccionado un subconjunto de todo el catálogo de herramientas que se poseía nos centramos en filtrar por aspectos más específicos y detallados sobre el análisis. Para poder seleccionar dichos aspectos se necesitaba realizar un estudio en profundidad del lenguaje de consulta SPARQL.

Para ello, nos apoyamos en la Recomendación del World Wide Web Consortium publicada el 15 de enero de 2008, teniendo en cuenta las actualizaciones realizadas a posteriori. En dicho estudio del lenguaje, obtuvimos las siguientes conclusiones sobre SPARQL:

SPARQL posee una gramática libre de contexto, es decir, se trata de una gramática formal cuyas reglas de producción son de la forma:

$$V \rightarrow w$$

Donde "V" es un símbolo no terminal, y "w" es un símbolo terminal o no terminal. Esta definición implica que, independientemente del contexto, "V" siempre puede ser sustituida por "w". Con lo cual, definimos que la mejor forma de representar dicha gramática es la utilización de la notación EBNF (Extended Backus–Naur Form). Dicha notación es una extensión de BNF (Backus–Naur Form) y nos permitiría realizar una descripción formal de un lenguaje formal.

Vistas las características lingüísticas del lenguaje de consulta SPARQL, fue fácil deducir que tipo de algoritmo de análisis le vendría mejor. El analizador sintáctico LL es un analizador sintáctico descendente, en el cual la entrada se realiza de izquierda a derecha y posee construcciones de derivación por la izquierda. Dichas construcciones se basan en el principio de sustituir el no terminal que esté situado más a la izquierda.

Con lo cual, tras este estudio se añadieron dos criterios más de selección:

- Que la herramienta aceptara gramáticas en notación EBNF.
- Que el algoritmo de análisis fuese LL.

Se consiguió reducir considerablemente la lista de herramientas que cumplían los criterios establecidos, obteniendo los siguientes tres frameworks para la generación de analizadores: ANTLR, HiLexed y SLK.

Dado que se necesitaba elegir uno de los tres candidatos y ya no se tenían criterios en cuanto a las limitaciones del desarrollo o las del lenguaje, se optó por aplicar el siguiente criterio:

Para facilitar el uso de la herramienta, se consideró como criterio la abundancia de documentación sobre la misma, que contuviese la información necesaria para que permita trabajar cómodamente.

ANTLR al ser una herramienta famosa entre los generadores de analizadores, posee una gran cantidad de información y documentación de ayuda para su uso. Además, tiene una licencia BSD la cual nos permite utilizar el framework sin condiciones altamente restrictivas. Es capaz de generar código Java, y entiende las gramáticas definidas en notación EBNF. Su principal algoritmo de análisis es un LL. Por lo que se decidió utilizar ANTLR para generar los analizadores léxico y sintáctico de nuestro intérprete de SPARQL.

Tras comentar la elección de la herramienta en una reunión de SCRUM se planteó como siguiente requisito aprender ANTLR y generar el código de los analizadores, tanto léxico como sintáctico. Para ello, se tuvo que trabajar en la gramática del lenguaje, generando un documento que cumpliese las especificaciones de ANTLR y que contuviese el lenguaje de SPARQL en notación EBNF.

Una vez obtenido el documento con la gramática de SPARQL, se implementó un pequeño programa en Java que utilizaba el framework ANTLR para, a partir de dicho documento, generar las clases que constituyen los dos analizadores.

ANTLR generó una clase para cada analizador, por un lado tenemos la clase SparqlLexer, que contiene el analizador léxico y por el otro, SparqlParser que contiene el analizador sintáctico. Generado el código, exportamos ambas clases al proyecto de nuestro componente.

Aunque se había generado el código de los analizadores, aún distaba mucho de dar por terminada esta primera iteración, donde el objetivo principal era obtener una estructura con los tokens o símbolos que contuviese la consulta, ordenados tal y como aparecen, de manera que se pudiese preguntar no solo el tipo de símbolo, sino también, su valor.

La clase SparqlParser no solo valida la sintaxis de la consulta, sino que además genera una estructura arbórea como la que se requería. Cada nodo del árbol contiene un objeto de la clase Token la cual nos ofrece una interfaz con las siguientes funciones:

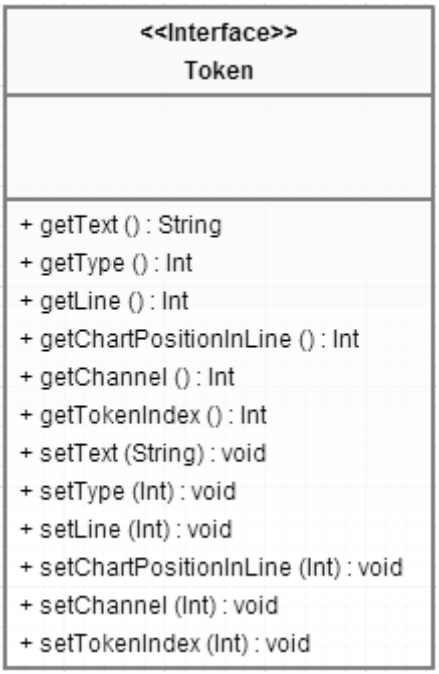

Figura 3. Especificación de la interfaz Token con las diferentes funciones que nos ofrece la librería ANTLR.

Del conjunto de funciones que nos permite utilizar la interfaz, las funciones que más se adaptan a nuestras necesidades son: getText() y getType(). Dichas funciones nos permitirán obtener tanto el valor del token, como el tipo del mismo respectivamente.

Se decidió pues, crear una clase para la estructura que se requería, que contuviese todos aquéllos elementos necesarios para posibilitar el recorrido de la misma entendiendo cada uno de los elementos que se analizaban.

La clase SparqlSemanticTree se define tanto como el contenedor, como el generador de la estructura de reconocimiento de la consulta. El objetivo de esta clase es aportar un objeto el cual contenga toda la información necesaria respecto a los tokens y a la consulta de entrada, de forma que, fuese fácil expandir la clase con funciones útiles de cara a posteriores análisis. Con lo cual, definimos la clase SparqlSemanticTree de la siguiente forma:

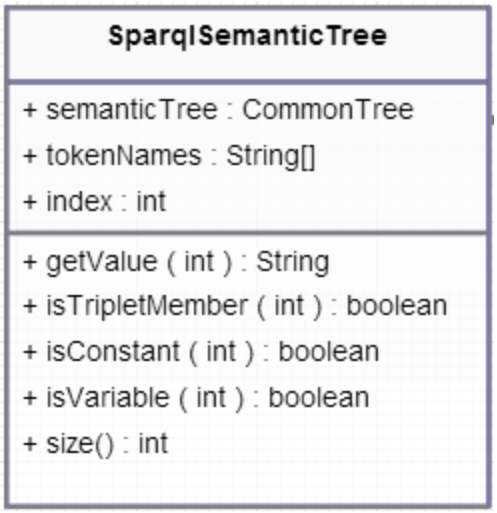

Figura 4. Especificación de la estructura de reconocimiento del lenguaje.

Al constructor de la clase se le ha de pasar como parámetro un objeto de tipo String, que contiene la consulta (query), ya que es en el constructor donde se aplican los dos analizadores, el léxico y el sintáctico, tal y como muestra la imagen.

```
public SparqlSemanticTree(String query) {
    try {
        SparqlLexer sparqlLexer = new SparqlLexer(new ANTLRStringStream(query));
        CommonTokenStream tokenStream = new CommonTokenStream(sparqlLexer);
        SparqlParser sparqlParser = new SparqlParser(tokenStream);
        SparqlParser.query_return queryResult = sparqlParser.query();
        this.semanticTree = (CommonTree) queryResult.tree;
        this.index = semanticTree.getChildCount();
        this.tokenNames = sparqlParser.getTokenNames();
    } catch (Exception exception) {
        System.out.println(exception);
    }
}
```

Figura 5. Constructor de la estructura de reconocimiento, donde se invocan los analizadores.

Tras hacer la llamada de los dos analizadores, se recogen los tres atributos de la clase del árbol semántico: la estructura arbórea (semanticTree), el tamaño de la estructura (index) y la lista de tipos de tokens (tokenNames).

A continuación se listan los métodos de la clase con una breve descripción de los mismos:

- **getValue**: esta función devuelve el valor del token que se encuentra en la posición que se le pasa a la función por parámetro.

- **isTripletMember**: evalúa el token que se encuentra en la posición pasada por parámetro para saber si éste pertenece a una tripleta. Los miembros de una tripleta pueden ser valores constantes o variables.
- **isConstant**: evalúa el token que se encuentra en la posición pasada por parámetro para saber si éste es un valor constante.
- **isVariable**: evalúa el token que se encuentra en la posición pasada por parámetro para saber si éste es una variable. Las variables en SPARQL se representan con un interrogante precediendo al nombre que se le da a la misma.
- **Size**: devuelve el tamaño total del árbol.

Una vez implementada la clase, se cumplía el requisito planteado para esta primera iteración, y ya poseíamos una estructura fiable que se puede recorrer fácilmente en posteriores análisis de la consulta.

9.2 IMPLEMENTACIÓN DE UNA ESTRUCTURA DE ALMACENAMIENTO DE RESULTADOS

9.2.1 Análisis

Las bases de datos triplestore están enfocadas a manejar grandes cantidades de datos, lo cual se conoce como Big Data. Éste problema de volumen de datos requiere un tratamiento de los conjuntos de datos que se obtienen, no solo al finalizar la consulta, sino también, durante la ejecución de la consulta y las operaciones que se lancen.

Asimismo, éste tratamiento de los conjuntos de datos debe ser consensuado entre los dos componentes que se encargan de la resolución de las consultas: el SPARQL Interpreter y el QueryPlayer. El primero se encarga de obtener el conjunto de datos resultante, mientras que el segundo, debe gestionar los diferentes conjuntos de datos obtenidos durante la ejecución de la consulta.

Esta iteración global hace hincapié en las diferentes decisiones de diseño y las diversas propuestas de implementación para llevar a cabo el tratamiento de los conjuntos de datos. El objetivo principal, es crear una estructura que nos permita almacenar, relacionar y manejar los conjuntos de datos que se generen durante la consulta, o como resultado final de la misma.

9.2.2 Diseño

Antes de empezar a desarrollar como se ha afrontado el objetivo de esta iteración necesitamos hacer un breve inciso en donde expliquemos cómo se encontraba la arquitectura Triskel en ese momento.

La separación entre los distintos componentes de la arquitectura había hecho que diversas clases básicas que se utilizaban en más de un componente se tuviesen que externalizar y acoplar al proyecto como librerías. Dichas clases son representaciones de los objetos más básicos que tratamos en la arquitectura triskel, los patrones y las tripletas.

En RDF una tripleta se compone de varios elementos distintos que podemos dividir en dos categorías: Términos y Variables (What). Los términos son elementos finales, es

decir, elementos que ya son un valor de por sí, y que por lo tanto, se conocen. La siguiente imagen muestra una tripleta compuesta por términos:

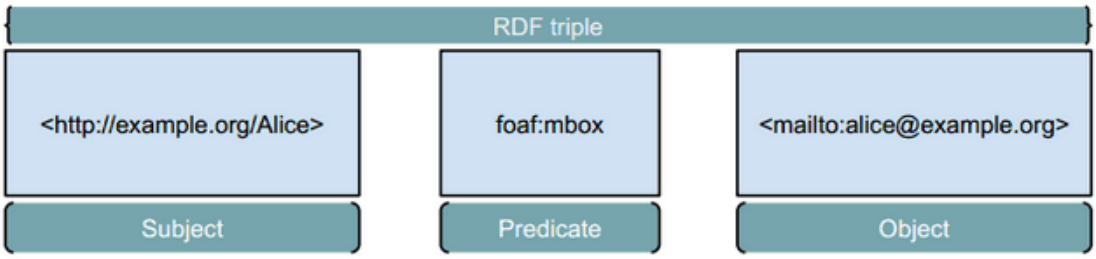

Figura 6. Ejemplo de una tripleta en RDF.

Por otro lado, una tripleta puede contener variables, denominadas What, cuyo valor se determina una vez se ha buscado la existencia de la tripleta con respecto a todos los valores concretos que están contenidos en el almacenamiento. La siguiente imagen ilustra, dado un conjunto de valores almacenados, un ejemplo del resultado de una tripleta con variable:

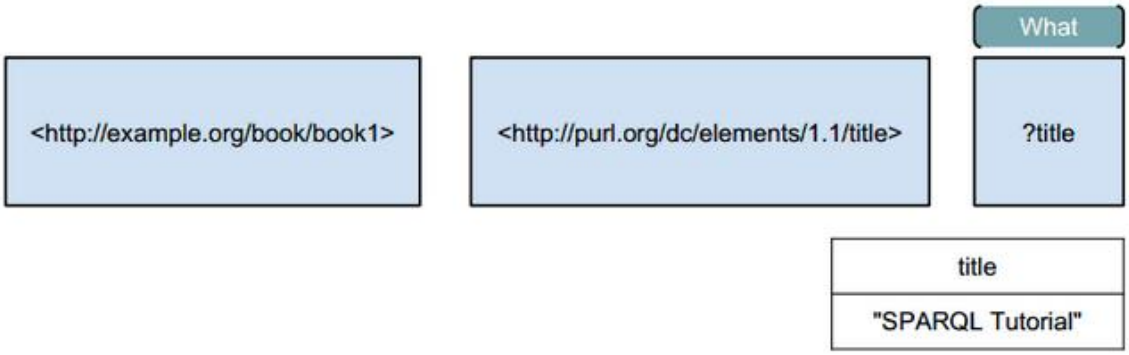

Figura 7. Ejemplo de una tripleta con un What (?title) y la tabla resultado de la consulta.

La existencia de una o más variables en una tripleta la convierten en un patrón, con lo cual, se trata como un elemento de búsqueda y no como un elemento de información. Cada variable devuelve un conjunto de datos, es decir, todos aquéllos valores cuya combinación con los elementos definidos existe en la base de datos.

Por lo tanto, según el número de variables, definimos tres dimensiones para los patrones:

- Patrones Unidimensionales.

- Patrones Bidimensionales.
- Patrones Tridimensionales.

Cuanto mayor número de variables, mayor es la dimensión y por consiguiente, mayor es el coste asociado a la resolución del patrón, llegando a solicitar, en el caso de un patrón tridimensional, todo el contenido de la base de datos.

Dada la estructura del proyecto triskel definimos dos tipos de almacenamiento, por un lado, tenemos el almacenamiento de relaciones (triplestore), que nos indica la relación entre un sujeto, predicado y objeto. Y por otro lado, tenemos el almacén de valores o términos (termstore). La separación de estos almacenamientos se basa en la idea de utilizar el termstore como un diccionario de traducción, el cual, dado un determinado valor, lo traduzca a un identificador numérico, o id. Esto permite al triplestore abstraerse de los valores y trabajar únicamente con dichos valores numéricos, pudiendo crear así una estructura mucho más óptima para realizar búsquedas de tripletas.

Por lo que, tanto los patrones como las tripletas deben ser capaces de almacenar identificadores de forma que podamos diferenciar cuando se trata de un identificador a nivel de almacenamiento, y cuando un valor numérico de la consulta.

Esto nos crea la necesidad de un tercer objeto que nos sitúe en un nivel mucho más superior de conocimiento y que sirva al intérprete para almacenar las tripletas de la consulta, donde solo poseemos términos y variables.

Resumiendo, poseemos estas tres clases para identificar las tripletas:

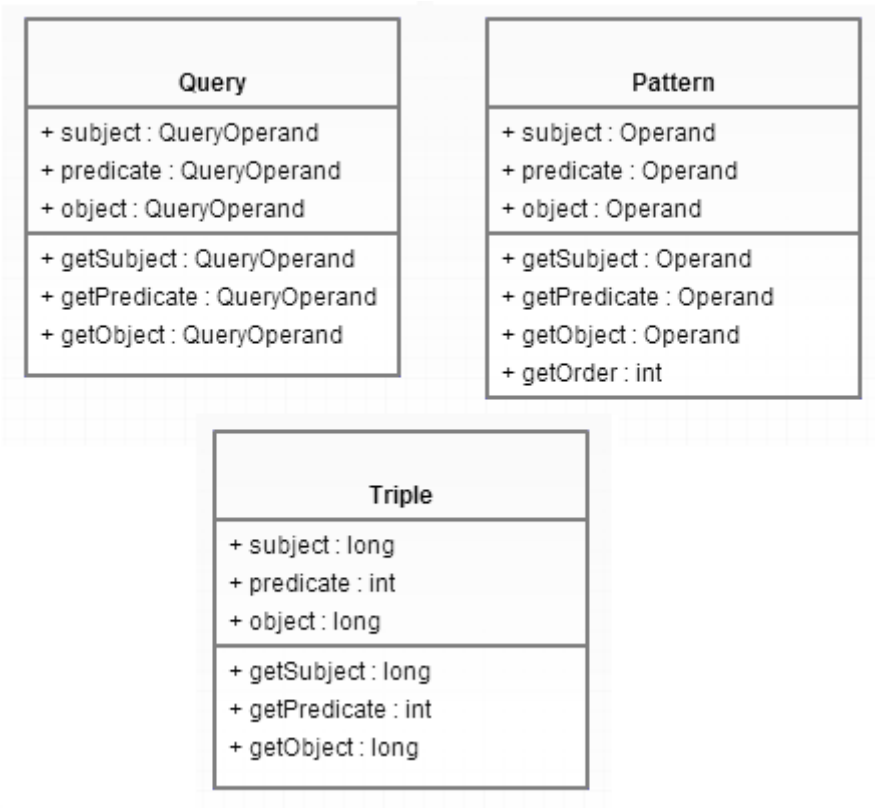

Figura 8. Especificación de las tres clases existentes para representar tripletas en los diferentes niveles de la arquitectura Triskel.

Ya que nuestro componente es parte de los niveles más externos de la arquitectura triskel, la clase triple, que contiene la relación de tres identificadores, no nos será de ninguna utilidad. Así que, centraremos nuestro uso en las clases Query y Pattern. Definimos entonces la creación de objetos de tipo Query mientras se interpreta la consulta, para luego realizar una traducción, mediante el termstore, de Query a tipo Pattern, de forma que éste último sea el objeto que se pase al componente de resolución de consultas.

Por lo tanto, una primera aproximación de tratamiento de los conjuntos de datos, es utilizar la clase Pattern como almacenamiento de resultados, es decir, sustituir sus "What" por los respectivos conjuntos de datos que conforman la resolución del patrón.

Esta propuesta nos permitía conocer los conjuntos de datos y tenerlos asociados con sus variables correspondientes, por lo que, en un principio, era una propuesta viable. Durante el desarrollo del prototipo siguiendo esta aproximación, evaluamos el lenguaje

de consulta SPARQL con esta propuesta, con vistas de encontrar algún caso en el que no nos sirviese.

Las consideraciones que nos hicieron desechar esta primera aproximación fueron las siguientes:

- La relación entre las variables de una consulta, resultaba un proceso costoso debido a la multitud de evaluaciones que se requerían para detectarlas. Cada patrón podía tener una o más variables, por lo que había que evaluar los patrones de forma completa para saber la posición del What o utilizar estructuras adicionales como índices.

- Operaciones como el Optional, donde se devuelven resultados con valores vacíos incluidos, eran inviables en dicho modelo, ya que, como ya se ha mencionado anteriormente, la resolución de un patrón solo devuelve aquéllos valores que existen en la base de datos, y se relacionan con los elementos definidos del mismo, por lo que, el conjunto resultante de un What, nunca contendría un valor vacío.

- El resultado final de la consulta, no tiene porqué cumplir el formato de patrón, pudiéndose solicitar más de tres variables de la consulta como resultado final.

Estos tres motivos, son los que principalmente hicieron que se desechara esta aproximación, y se buscase una forma alternativa de tratar los resultados.

Tras diversas reuniones y consideraciones, se diseñó una aproximación mucho más apropiada que aparentaba cumplir con aquéllos criterios que habían desechado la anterior. En concreto se planteaba realizar el tratamiento con objetos de tipo tabla.

Desde un punto de vista abstracto, todo patrón generaba una tabla de resultados de una, dos o tres columna, que se desligaban completamente del mismo. Por lo tanto, obteníamos una separación entre los resultados, y los tipos básicos del proyecto, lo cual nos generaba una arquitectura mucho más limpia. Esta solución también nos permitía considerar los valores vacíos en el conjunto de resultados, solucionando así el problema de la operación Optional, mediante un pequeño tratamiento de los resultados previo a la generación de la tabla.

Además puesto que podemos definir la tabla como un objeto con n columnas, donde n es el número de variables que se vayan a considerar en la misma, nos servía como objeto resultante para almacenar la solución final de la consulta.

Por lo que, inicialmente, se definió la clase tabla con funcionalidades básicas, que intuitivamente se presentaban como necesarias:

- En la creación de una tabla, se define únicamente las columnas que la componen, dejando los registros como una lista dinámica que crece conforme se añaden más filas a la tabla.
- **Size**: devuelve el número de filas que posee la tabla, independientemente del número de columnas.
- **Get**: dependiendo del nivel de la arquitectura en el que nos encontremos, manejaremos dos formas de obtener un valor. Por un lado, podemos conocer el nombre de la columna, lo cual viene a ser el nombre de la variable o What cuyo conjunto de resultados se almacena en la tabla. Por otro lado, podemos conocer la posición donde se encuentra el valor, pero no el nombre de la columna.
- **GetRow**: nos devuelve una fila completa.
- **AddRow**: es una función para la construcción de la tabla, la cual añade filas a la misma.
- **GetColumns**: devuelve la cabecera de la tabla, es decir, una lista con los nombres de las columnas que conforman la tabla.

La siguiente imagen, representa la clase tabla destinada para el manejo de los resultados.

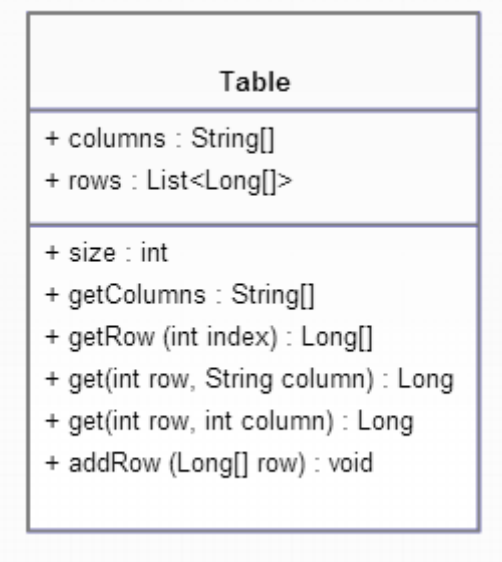

Figura 9. Especificación de la clase Table, con los atributos y las funciones que posee.

SPARQL es un lenguaje de consulta que puede definir algunas operaciones de forma implícita. Dichas operaciones son el Join y el Select.

- La operación Join permite calcular el producto cruzado de todos los registros entre dos tablas; así cada registro en la tabla A es combinado con cada registro de la tabla B; pero sólo permanecen aquellos registros en la tabla combinada que satisfacen las condiciones que se especifiquen. Existen diferentes tipos de Joins utilizados, en el caso de SPARQL, el más utilizado es el Join de equivalencia.

  Dicho Join define como condición una equivalencia de los datos de dos columnas entre dos tablas. Con lo que, la tabla resultante posee los registros, cuyo valor existe en ambas columnas a las que se aplica la condición de equivalencia, incluyendo el resto de columnas de ambas tablas.

  La siguiente imagen, ilustra la ejecución de un Join:

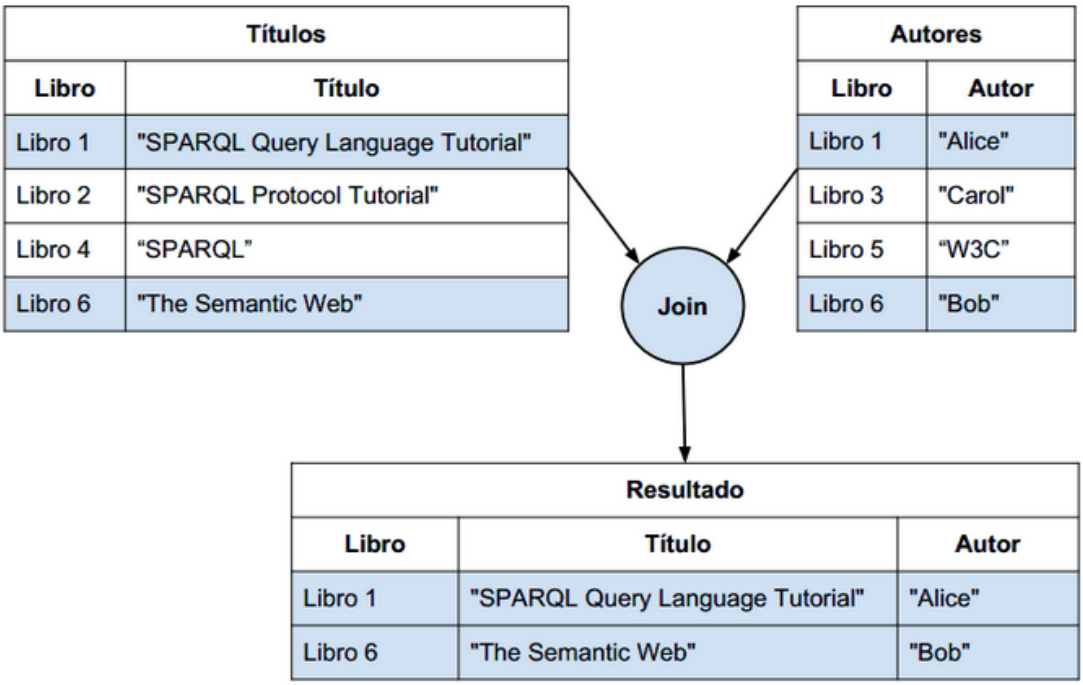

Figura 10. Ejemplo de ejecución de un Join. Las filas que comparten el mismo valor en la columna Libro son las que son insertadas en la tabla resultado.

- El Select, por otro lado, se comporta como una proyección permitiendo extraer columnas de una determinada tabla. Esta operación es muy útil de cara a la obtención del resultado final de la consulta, donde, al igual que en SQL, SPARQL define una sentencia Select, que elige, de entre un conjunto de variables definidas durante la consulta, las variables que son requeridas.

Debido al aspecto implícito de estas operaciones y su relación intrínseca con las tablas se consideró realizar la implementación de las mismas de forma interna a la clase tabla. Añadiendo ambas funciones al conjunto de métodos disponibles para la clase.

Por último, como previsión y como ayuda para resolver la operación de Join, se implementó también la equivalencia entre tablas, es decir, la posibilidad de evaluar si dos tablas son iguales. Lo cual nos permitiría también comparar si dos columnas, obtenidas mediante un select, son iguales.

Con lo cual, creamos una clase para el tratamiento de los resultados robusta y completa, cuya definición es la que se muestra a continuación:

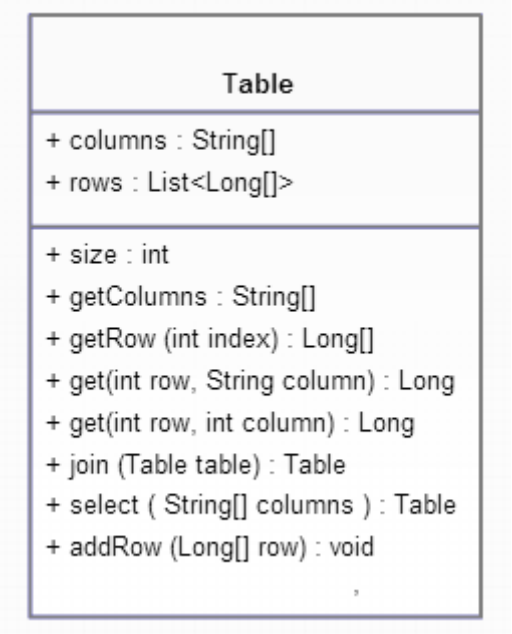

Figura 11. Especificación del tipo Table completa, con la incorporación de las funciones intrínsecas Join y Select.

Como se puede observar en todas las definiciones de la clase tabla enseñadas hasta ahora, únicamente maneja resultados de tipo numéricos. Esto es debido a la explicación inicial sobre los patrones y el modelo de almacenamiento basado en traducciones e identificadores numéricos. La tabla diseñada en esta iteración es una estructura pensada únicamente para el tratamiento interno de los resultados.

De cara a la devolución de los resultados finales, es necesario realizar una traducción inversa, de identificador a término, con lo cual, se devuelvan valores entendibles por el realizador de la consulta, y se abstraiga completamente de la gestión interna que realiza el almacenamiento.

Este problema se tratará en la última iteración global, donde el controlador será el encargado de traducir la tabla resultante de la consulta con identificadores, en una tabla con términos, utilizando el termstore para esta tarea.

En definitiva, la tabla de términos es una clase similar a la tabla de identificadores, diferenciándose de esta en las funciones que posee, siendo la tabla de términos mucho más simple al no necesitar operaciones para su tratamiento.

9.3 IMPLEMENTACIÓN DE UNA ESTRUCTURA DE EJECUCIÓN DE OPERACIONES.

9.3.1 Análisis

Uno de los grandes retos en el desarrollo del intérprete de SPARQL es la interpretación semántica de la consulta, operación que entendemos como la lectura de la consulta de forma que se sepan las diferentes operaciones que se deben realizar.

Ya que la ejecución de los patrones y las operaciones de las consultas es responsabilidad del componente QueryPlayer, era necesario establecer una forma de comunicación en la que se le pudiese pasar el resultado de esa interpretación semántica de una forma fácil de entender, y fácil de manipular.

En el caso de las operaciones matemáticas se pueden construir árboles para representar el orden en el que se ejecutan, en esta iteración se planteará la necesidad de definir un modelo de construcción de la estructura de forma que ésta represente un orden de ejecución para las consultas en SPARQL.

Nos centraremos pues, en la implementación de dicha estructura, las decisiones de diseño tomadas para la misma, y la implementación de las diferentes operaciones de SPARQL contempladas para este Trabajo Fin de Grado.

9.3.2 Diseño

Desde el comienzo de la iteración se planteó como objetivo crear una estructura arbórea que contuviese de forma ordenada la ejecución de las diferentes operaciones que componían la consulta.

Dicha idea de partida nacía de la similitud de esta tarea con los intérpretes de expresiones matemáticas que son capaces de construir un árbol de operaciones a partir de la misma. De la forma que, dada una expresión como la siguiente:

$$((1+2)*(3-4))$$

Figura 12. Expresión matemática de ejemplo para generar un árbol de operaciones

Se pueda generar un árbol de operaciones cuya ejecución respete la evaluación normal de la expresión de entrada:

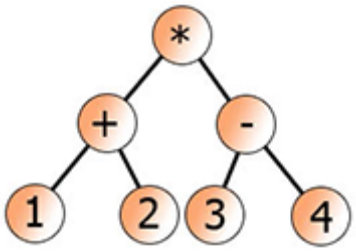

Figura 13. Árbol AST o árbol de operaciones de la expresión ejemplo de la Figura 12

Existen múltiples algoritmos para realizar el análisis de las expresiones algebraicas, como por ejemplo el algoritmo de "shunting yard" de Dijkstra, con el objetivo o la capacidad de generar un árbol de operaciones, comúnmente conocido como Árbol de Sintaxis Abstracta (AST). Estos algoritmos se apoyan en las diferentes características que presentan las matemáticas y su estricta definición de evaluación: prioridad entre operadores, existencia de operadores binarios o unarios, etc.

Por lo que, como primer objetivo en el planteamiento de una solución, se requería investigar el lenguaje SPARQL de forma que se comprobase la viabilidad de aplicar un modelo similar a los ya mencionados algoritmos de análisis de expresiones algebraicas. Para ello, acudimos directamente a la documentación de la versión 1.1 del lenguaje de consulta SPARQL.

Después de dedicar tiempo a la investigación del lenguaje de consulta SPARQL, se desestimó la idea de buscar una similitud, y se optó por investigar otras vías ya que a priori, no se podía utilizar los algoritmos mencionados anteriormente con un lenguaje de consulta. Dada la naturaleza de las matemáticas, las operaciones se adaptan bien a una estructura tal como un árbol binario, donde se puede tener, uno o dos operandos. En el caso de SPARQL, aunque podemos afirmar que existe un árbol binario para definir las consultas, la complejidad en su construcción nos llevó a utilizar un nivel de abstracción mayor.

Se probó a evaluar el lenguaje SPARQL con el fin de que se generase un árbol de operaciones n-ario. El motivo de esta idea, era la existencia de un elemento como los patrones en las consultas. Los patrones no solo son el elemento básico que utilizaban las operaciones, sino que además, eran, en cierto modo, una operación de búsqueda por sí

mismos. Esto encaminaba a la conclusión de que entre patrones no existía prioridad y todos estaban en el mismo nivel, así, una consulta como la siguiente

```
SELECT ?x ?name
{
  ?x  foaf:mbox <mailto:alice@example> .
  ?x  foaf:knows ?a1 .
  ?a1 foaf:knows ?a2 .
  ?a2 foaf:name ?name .
}
```

Figura 14. Definición de una consulta con múltiple patrones.

Se podría traducir en un árbol n-ario como el que sigue.

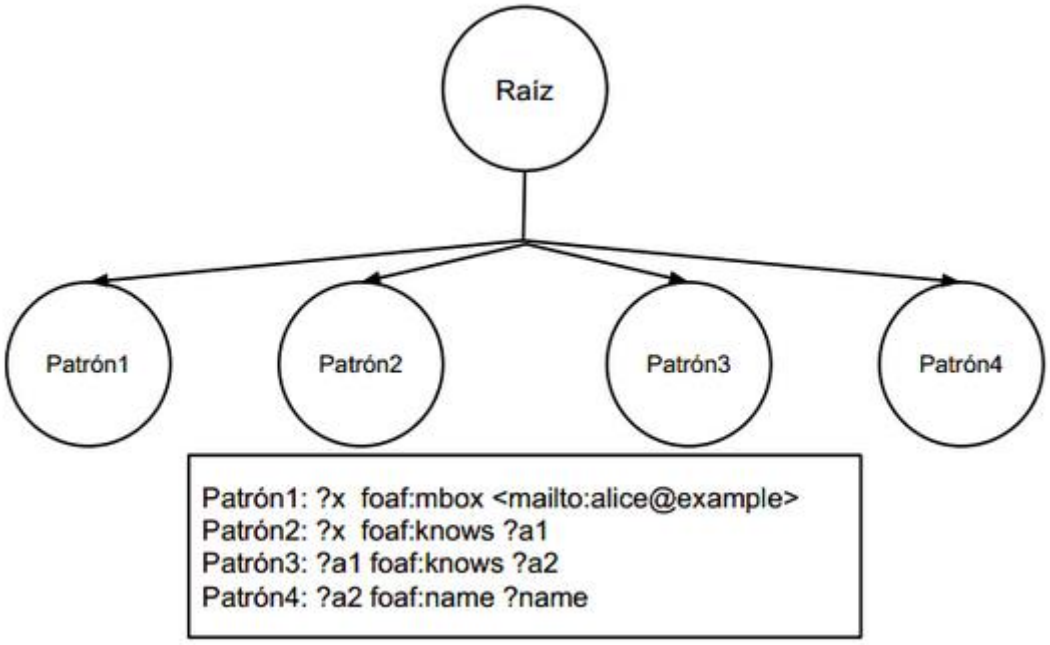

Figura 15. Definición del árbol de operaciones para la consulta mostrada en la Figura 14.

Llegados a este punto, una pregunta obvia es el nodo raíz, ¿cómo podemos definir un nodo raíz en una consulta en la que existan solo patrones? La respuesta a esta pregunta se obtuvo analizando el tratamiento de los resultados de las consultas en la iteración anterior. Indirectamente podemos comprender cada conjunto de patrones, como los n-hijos de una operación Join. Con esto conseguimos unificar todo el conjunto de variables de la consulta en una única tabla que contenga todos los resultados obtenidos hasta el momento y sobre la que se puedan aplicar todas las funciones que ayuden a

filtrar los resultados. Esta tabla final, es la tabla sobre la que realizaremos el select para obtener aquéllas variables que la consulta quiere como devolución.

Identificamos pues, en la imagen anterior, el nodo raíz como una operación Join que unifica todos sus n-hijos, y que, al ejecutar la raíz, obtendríamos el resultado de la consulta.

Utilizando como pruebas los diferentes tipos de consulta de ejemplo expuestos en el estándar 1.1 de SPARQL definido por el World Wide Web Consortium comprobamos de forma teórica que un árbol n-ario cubría las necesidades del lenguaje. Basándonos en dichas conclusiones, creamos nuestro árbol de operaciones como una clase denominada OperationTree.

A continuación, había que definir los tipos de nodos de nuestro árbol de operaciones. Por un lado, existen los nodos que contienen las operaciones, también denominados nodos funciones. Mientras que por otro lado poseemos los nodos patrones, que se comportarían como los elementos básicos del lenguaje, o, haciendo un símil con las expresiones algebraicas, las constantes.

Así pues, el siguiente diagrama representa nuestra arquitectura de nodos diseñada para el árbol de operaciones.

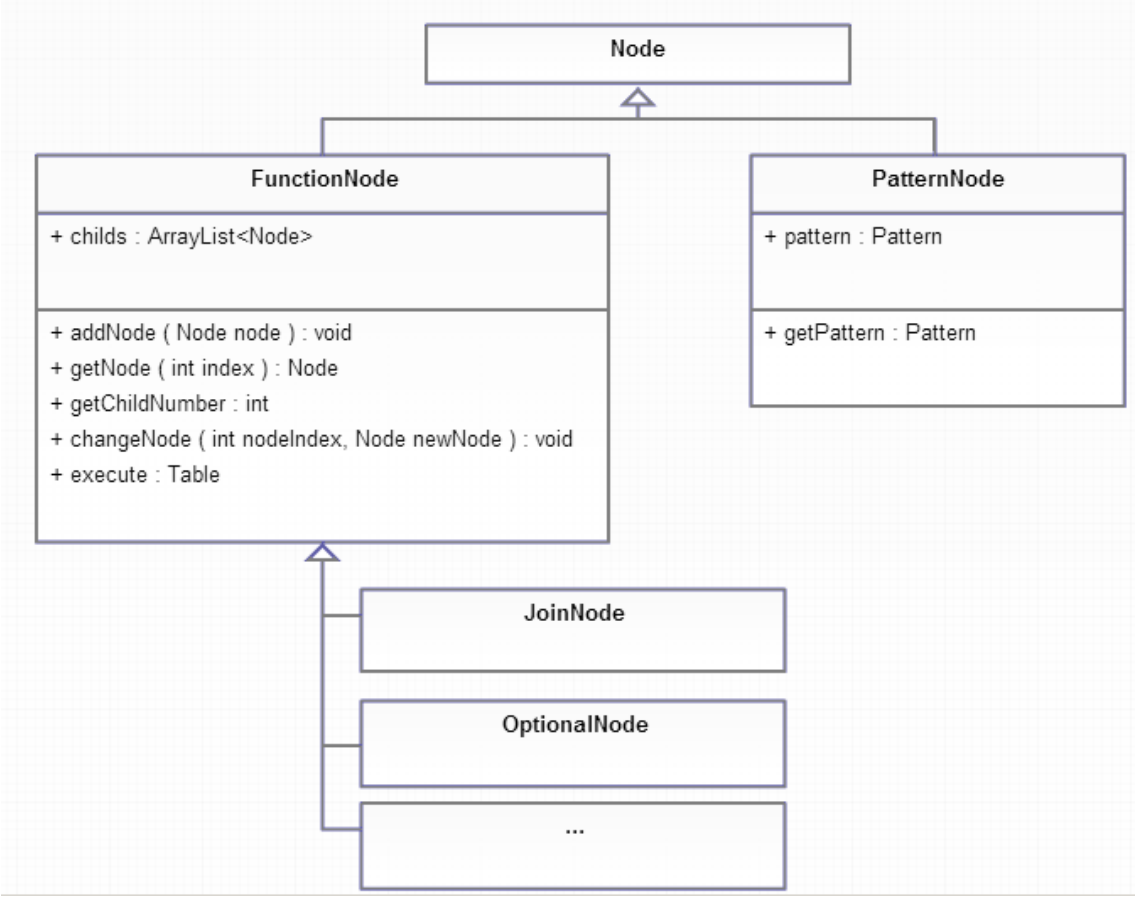

Figura 16. Arquitectura de los nodos en la clase OperationTree.

Como se puede ver en la imagen anterior, todas las clases heredan de un tipo abstracto Node, por un lado, tenemos los FunctionNode, es decir, los nodos que contendrán las operaciones o funciones del lenguaje, por el otro, tenemos el PatternNode, el nodo base que contendrá los patrones de la consulta.

Los PatternNode se comportan como nodos hoja en el árbol de operaciones, de ellos no cuelgan hijos ya que su única función, es almacenar un patrón y relacionarlo con la función padre si la hubiese.

Los FunctionNode se pueden comportar tanto como nodos hojas o como nodos rama, dependiendo del tipo de operación que representen, unaria o binaria respectivamente. Cada una de las clases que heredan de FunctionNode, debe definir una función execute() que es la implementación de la operación.

La siguiente imagen ilustra una consulta en SPARQL que posee tanto patrones como operaciones:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox
WHERE  { ?x foaf:name  ?name .
         OPTIONAL { ?x  foaf:mbox  ?mbox }
       }
```

Figura 17. Ejemplo de una consulta con patrones y operaciones.

Según el planteamiento de nuestra estructura, el árbol generado sería el siguiente:

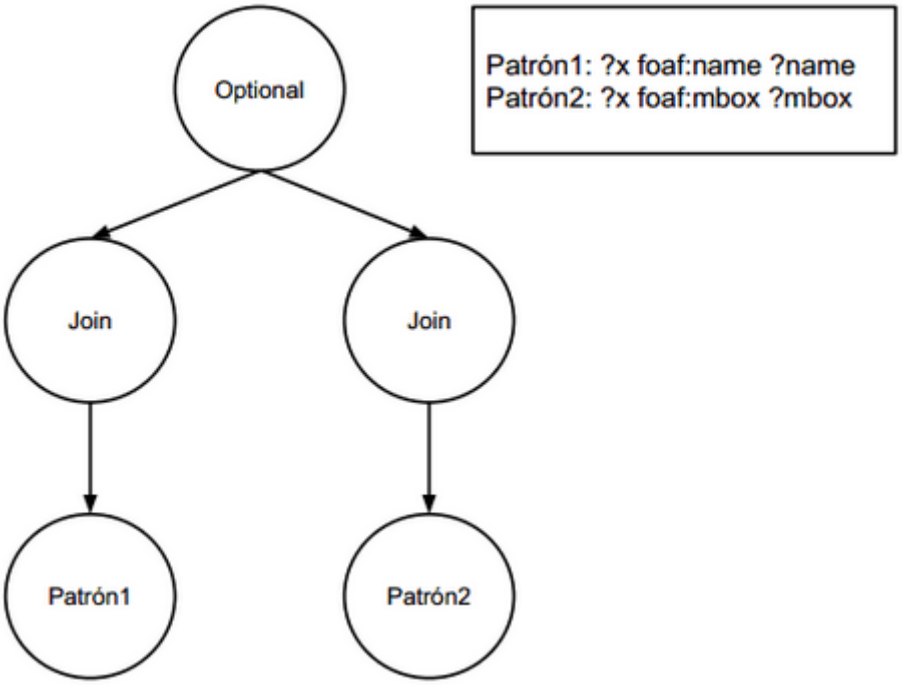

Figura 18. Árbol de operaciones resultante de la consulta de la figura 17.

Como ya se ha mencionado antes, todos los patrones que se encuentren juntos, es decir, que estén contenidos dentro de los mismos corchetes, se considerarán que pertenecen a un mismo conjunto, y por consiguiente serán hermanos dentro de nuestro árbol. En el caso de las funciones, aplicamos un método diferente, pues no las entendemos como pertenecientes al conjunto, sino comprendemos este último como el conjunto objetivo sobre el que se aplicará la función.

Para ilustrar el comportamiento de nuestra estructura cuando encontramos una operación, a continuación se expone un ejemplo mucho más detallado:

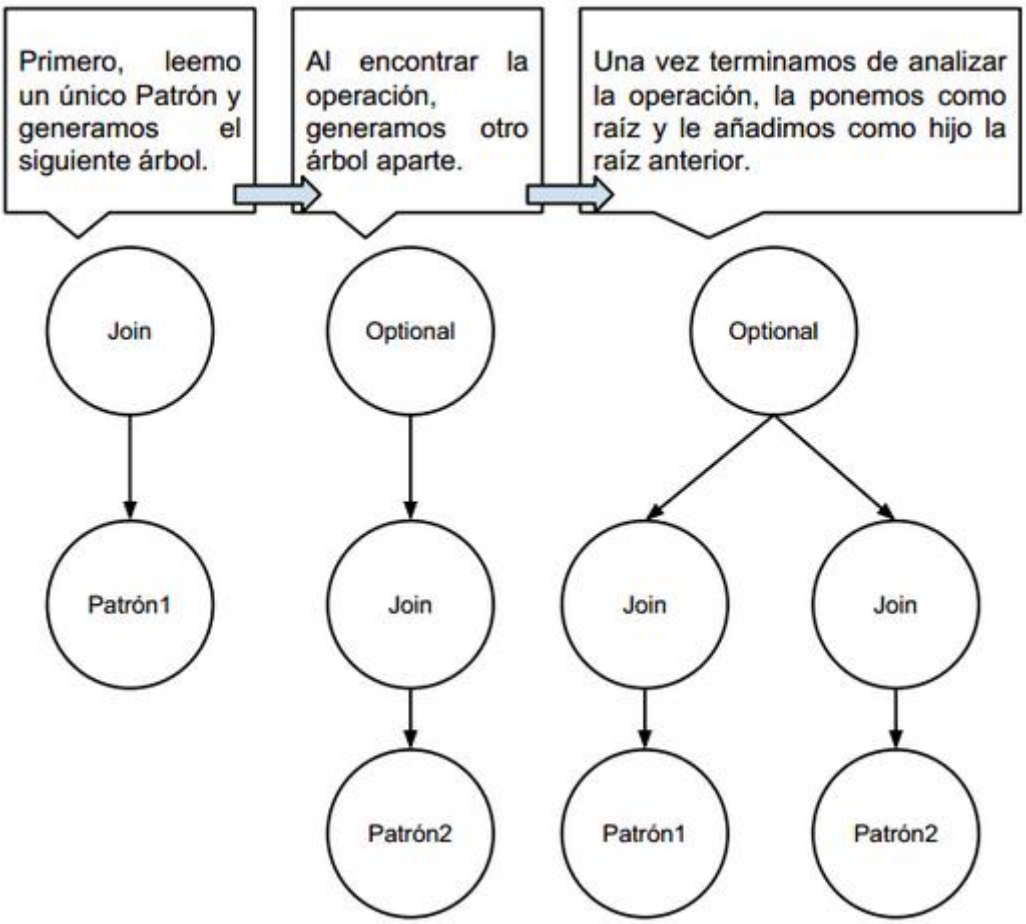

Figura 19. Modificación del árbol cuando se encuentra una operación.

Visto el ejemplo, no nos queda nada más que explicar sobre cómo se construye la estructura de operaciones. Veamos pues, las ventajas que posee, y qué es lo que ha hecho que ésta aproximación sea considerada la más adecuada.

Como ya se mencionó en el análisis de los problemas que se encontrarían en esta iteración, esta estructura es un elemento de comunicación entre dos componente, y mientras que el Intérprete se encarga de construirla, el componente QueryPlayer se encargaría de ejecutarla. Por lo tanto, debíamos hacer una estructura sencilla tanto en su generación como en su manipulación.

Las ventajas de la estructura anteriormente expuesta se explican en los siguientes puntos:

- Debido a que se trata de una estructura arbórea, una de sus principales ventajas es el recorrido. En este caso, pese a que dejamos el orden del recorrido en manos

del componente QueryPlayer, definimos un recorrido en profundidad, el cual nos permite acceder primero a los patrones, para, posteriormente, ir resolviendo de forma recursiva todas las operaciones y funciones que existan en el árbol.

- El árbol no se crea atendiendo a criterios de ordenanza o balanceo, por lo que no incluimos ninguna operación adicional para dichos criterios. Nuestro árbol se resolverá cómodamente independientemente de su forma.
- La estructura de Nodos permite abstraerse completamente de los tipos de los nodos en la ejecución del árbol, teniendo que diferenciar únicamente si el nodo es Patrón o Función (PatternNode o FunctionNode respectivamente). Por lo que su recorrido a nivel de código resulta mucho más sencillo.

Dada esta estructura se comenzó a plantear el tratamiento de los resultados durante la consulta. Es decir, como integrar la tabla resultante de la ejecución de un patrón en la ejecución de las funciones existentes en el árbol.

El compromiso al que se llegó para realizar este tratamiento fue el diseño de la capacidad de sustituir nodos en el árbol. Este compromiso permite al QueryPlayer sustituir los nodos por un nuevo tipo de nodo resultado, el TableNode. Esta nueva incorporación de diseño al árbol de operaciones nos aportaba los siguientes beneficios:

- Aprovechábamos la propia estructura del árbol como estructura para recorrer y almacenar los resultados. Por lo que no se generaban nuevas estructuras que incrementasen el coste en memoria del árbol de operaciones.
- El método execute() de las funciones no necesitaba ser alimentado por ningún parámetro ya que se ejecutaba directamente sobre sus hijos, con lo cual, se minimizaba la probabilidad de error en el paso de parámetros.
- No se mantenía ningún tipo de información inútil en el árbol de operaciones. A medida que se resuelven los patrones, éstos desaparecen siendo sustituidos por sus respectivas tablas de resultados. Por otro lado las funciones, a medida que se resuelven, son sustituidas por sus respectivas tablas resultado, y, al ser eliminado el nodo función, éste muere con todos sus hijos.
- Por último y no menos importante, esta decisión de diseño permitía una depuración mucho más precisa y cómoda de la ejecución del árbol de operaciones, lo cual facilitaba la tarea de detección de errores.

Expuesta la estructura de operaciones, nos centraremos en explicar las diferentes funciones desarrolladas en este Trabajo Fin de Grado, y los detalles de su implementación.

Empecemos pues con la operación Join. Esta operación prácticamente se había resuelto en la iteración anterior, pues recordamos que el Join había sido implementado como una función interna en la clase Table. Así, el método execute() de un nodo Join, no contiene la implementación de la función propiamente dicha, sino que contenía la ejecución ordenada del método join implementado en las tablas entre cada uno de sus hijos.

Principalmente, el nodo Join obtiene de entre sus n-hijos, aquél que posee la tabla más grande, entendiendo el tamaño de la tabla como la que posee más columnas. Para, posteriormente, realizar la operación join entre sus hijos por pares, solucionando así la contradicción de haber definido una operación binaria con n-hijos.

A partir de la tabla más grande, realizamos los diferentes join entre los hijos del nodo, utilizando el resultado como operando en la siguiente ejecución de la función hasta que no queden más hijos.

Otras de las funciones implementadas fueron las funciones de filtrado “Filter Exists” y “Filter Not Exists”. Durante del estudio de SPARQL se observó que existían múltiples funciones de filtrado, y dentro de éstas se encontraban la mayor parte de funciones unarias del lenguaje. Existen multitud de funciones de filtrado en pos de poder preguntar por gran variedad de tipos de datos.

Dichas funciones utilizan directamente el diccionario o termstore para su resolución, por lo que, dependiendo de la forma en la que se resuelvan, pueden ser muy costosas. Por ello, nos centramos en las operaciones de filtrado que se realizan a base a conjuntos de tablas, es decir, los filtrados binarios.

Ambos filtrados definen un conjunto de resultados a partir de una serie de patrones, y en su ejecución deben ser capaces de generar una tabla resultado comparando los dos conjuntos de datos y filtrando según dicte la operación.

La función Minus se implementó por su característica definición en el lenguaje de consulta de SPARQL. Esta función se comporta exactamente igual que el “Filter Not

Exists" salvo por un pequeño matiz que nos llevó a entender mejor el lenguaje de SPARQL y a plantearnos nuevos casos de estudio.

En SPARQL existe lo que se conoce como enlace de variables. Toda variable definida con el mismo nombre dentro de una consulta de SPARQL está ligada con todas sus apariciones. Ilustremos esto con una imagen:

Data:

```
@prefix foaf:       <http://xmlns.com/foaf/0.1/> .

_:a  foaf:name       "Alice" .
_:a  foaf:homepage   <http://work.example.org/alice/> .

_:b  foaf:name       "Bob" .
_:b  foaf:mbox       <mailto:bob@work.example> .
```

Query:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox ?hpage
WHERE  { ?x foaf:name  ?name .
         OPTIONAL { ?x foaf:mbox ?mbox } .
         OPTIONAL { ?x foaf:homepage ?hpage }
       }
```

Query result:

name	mbox	hpage
"Alice"		<http://work.example.org/alice/>
"Bob"	<mailto:bob@work.example>	

Figura 20. Ejemplo de una query con múltiple apariciones de una misma variable.

Como se puede observar, pese a las diferentes apariciones de la variable ?x ésta siempre se referirá al mismo conjunto de datos (todos los sujetos definidos en el apartado Data: "_:a" y "_:b"), aunque éste pueda variar. Este hecho es precisamente lo que nos hace tan útil utilizar la función join en las consultas de SPARQL, dándonos la posibilidad de reutilizar valores una vez calculado un primer conjunto de resultados de una variable.

La operación Minus, sin embargo, es capaz de romper dicho enlace entre variables, produciendo en determinados casos muy concretos, salidas distintas al "Filter Not Exists". En esencia, su funcionamiento es exactamente igual al de la operación de

filtrado, pero al romper el enlace entre variables, los conjuntos a los que se refieren las variables no tienen por qué ser iguales, y por ende, pueden producir resultados diferentes.

Esta operación se implementó por su interesante casuística y por definir una operación que ofrecía un interesante caso de estudio para los diferentes métodos de ejecución del árbol de operaciones que se plantearon inicialmente y cuya implementación fue llevada a cabo por el componente QueryPlayer.

Por último, consideramos la implementación de la función Optional. Esta operación, como ya se mencionó en apartados anteriores, permite la existencia de valores nulos en el conjunto de resultados, por ende, es una operación cuyo tratamiento puede provocar errores en la consulta y suponía un buen caso de desarrollo para detectar y corregir errores.

La función Optional define, igual que el resto de funciones implementadas, un conjunto de patrones en los cuales, las nuevas variables definidas, admiten valores nulos o no existentes. El reto de esta implementación, era desarrollar la operación de forma que no solo fuese capaz de construir la tabla resultante añadiendo aquéllas columnas cuyos valores son opcionales, sino que además, lo hiciese de forma autónoma, dado que el almacenamiento nunca devuelve valores nulos.

Por lo tanto, la función Optional analiza la cabecera de las tablas de sus hijos, detectando las columnas que están definidas como opcionales, y por lo tanto, generando una nueva tabla donde las columnas opcionales posean valor en los registros que ambas tablas tienen en común, mientras que en aquéllos que no, las columnas opcionales se les asigne el valor null.

Resumiendo, en esta iteración se ha cumplido con el objetivo planteado, generando una estructura de operaciones sólida y eficiente. Cuya construcción se basa en el reconocimiento del ámbito de ejecución de las funciones y en el tratamiento de los patrones como los elementos más básicos del lenguaje.

Asimismo, las diferentes decisiones de diseño permiten a la estructura que sea adaptable, y que, a medida que ésta se resuelve, sea capaz de auto eliminarse, mejorando así su rendimiento en memoria.

Además, el desarrollo de las operaciones como nodos del árbol de forma totalmente abstracta a la ejecución del mismo, lo convierte en una estructura con una modularidad extremadamente alta, permitiendo incorporar nuevos elementos, o eliminar funciones obsoletas, sin necesidad de que otros componentes se vean alterados.

9.4 IMPLEMENTACIÓN DE UN CONTROLADOR QUE UNIFIQUE Y USE LAS ESTRUCTURAS DISEÑADAS.

9.4.1 Análisis

En las iteraciones anteriores hemos implementado todas las herramientas necesarias para la construcción de nuestro intérprete. Hemos tratado el análisis, los resultados, y la estructura que permite el entendimiento y ejecución de la consulta.

En esta última iteración del proyecto, unificamos todas esas herramientas en un único controlador que se encargue de coordinar y gestionar dichos recursos para permitirnos obtener el objetivo principal de este Trabajo Fin de Grado, el cual, podemos resumir como: dada una query en lenguaje SPARQL, construir una estructura de operaciones que sea ejecutable.

Definimos pues las tareas que debe contemplar el controlador que desarrollaremos en esta última iteración.

- La primera tarea que se debe realizar es lanzar los analizadores del lenguaje, de forma que se pueda validar que la consulta introducida está bien construida, y al mismo tiempo, obtener nuestra estructura de reconocimiento.
- A partir de la estructura de reconocimiento del lenguaje, obtener el árbol de operaciones identificando cada uno de los elementos que la componen a medida que la recorremos.
- Lanzar a ejecutar el QueryPlayer pasándole el árbol de operaciones.
- Recoger el resultado final de la consulta, invocando al TermStore para traducirlo de identificadores a términos.

9.4.2 Diseño

La principal dificultad en esta iteración es construir el árbol de operaciones a partir de la estructura de reconocimiento. Esta tarea es muy similar a la de implementar un analizador sintáctico, pues al fin y al cabo, debemos recorrer los tokens devueltos por el analizador léxico de forma que a medida que leemos cada símbolo, podamos interpretarlo e introducirlo, si corresponde, en el árbol de operaciones.

Es por ello que en una primera aproximación se consideró la idea de modificar el código del analizador sintáctico e ir construyendo el árbol de operaciones de forma interna. Esta idea se desechó al investigar el código generado para el analizador sintáctico, ya que carecía de la limpieza necesaria para que fuese cómoda la modificación del mismo, exigiendo un gran coste en tiempo para poder realizar cambios. Con lo cual, se optó por una nueva aproximación.

Tras lanzar el analizador sintáctico y léxico, se comprobaba que la consulta estaba bien construida, por ende, cumplía la gramática que se había definido del lenguaje SPARQL. Teniendo esa certeza se consideró implementar una clase intérprete basada en la gramática de forma que, utilizando la estructura de reconocimiento, aplicase las mismas reglas definidas en la gramática generando así un código con un nivel de abstracción mayor al eliminar comprobaciones innecesarias.

Dada esta premisa, nuestro intérprete se convertía en un analizador semántico, recorriendo la estructura de reconocimiento de la consulta de forma que se analizaran los tokens que ésta contiene y se añadiesen en el árbol si corresponde. Utilizando la gramática como apoyo, se comenzó la implementación de la clase Interpreter.

La primera tarea del intérprete es almacenar la lista de variables que la consulta solicita, es decir, las variables que van entre la sentencia SELECT y la sentencia WHERE. Dichas variables deben ser almacenadas de forma que se mantengan accesibles para un posterior uso en la selección de columnas al final de la consulta.

Una vez obtenemos la lista de variables resultado de la consulta, podemos introducirnos en la cláusula WHERE, donde se encuentra el verdadero reto de implementación del intérprete. En esta cláusula nos encontraremos los dos problemas principales del intérprete, el reconocimiento de patrones y la construcción de funciones.

Como ya se ha mencionado antes, la estructura de reconocimiento proporcionada por el analizador léxico y sintáctico contiene los tokens de la consulta. Pero dichos tokens carecen de relación entre sí, es decir, un patrón como el que sigue:

```
?x foaf:name ?name
```

Figura 21. Ejemplo de un patrón con dos variables.

Aparecería dentro de la estructura de reconocimiento como tres elementos separados, identificados según al tipo de token definido en el léxico.

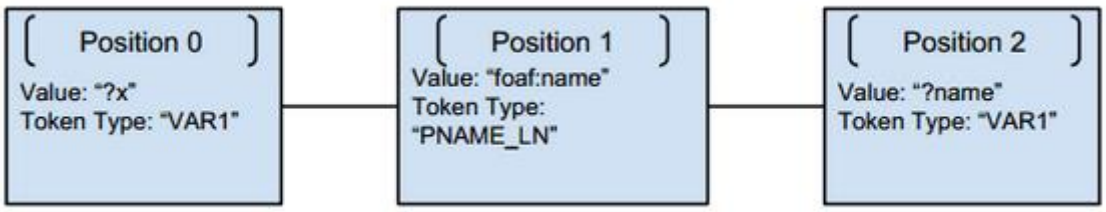

Figura 22. Representación de la aparición del patrón de la figura 19 en la estructura de reconocimiento.

Es tarea del intérprete reconocer qué elementos son parte de un patrón y leerlos de forma que se pueda llegar a construir el patrón de forma correcta para su posterior inserción en el árbol de operaciones. Para solucionar esta tarea hemos optado por diseñar tres clases que permitan modular las diferentes tareas del intérprete y nos posibiliten implementar un código limpio y sencillo de entender. Dichas clases de apoyo son: PatternHandler, PatternAnalizer, PatternBuilder.

El PatternHandler es el manejador principal de los patrones, el cual invoca a las otras dos clases para poder reconocer y construir el patrón. Al PatternHandler se le pasan, el índice de comienzo de un elemento perteneciente a un patrón, y el nodo función que contendría a dicho patrón.

Utilizando el índice de comienzo del patrón, el PatternHandler invoca al PatternAnalizer, cuya responsabilidad es tratar el patrón y sus diferentes formas para generar la lista de predicados y objetos que permitan construir el o los patrones. Para poder entender lo que realiza el PatternAnalizer debemos realizar un inciso y ver las diferentes formas y contracciones que posee SPARQL para escribir patrones.

La forma tradicional o completa de escribir un patrón es la que ya se ha mostrado en diversas ocasiones en este documento. Dicha forma es escribir el patrón como ilustra la Figura 21.

Por otro lado SPARQL permite omitir la especificación del sujeto si éste se repite en varios patrones, de forma que se especifique únicamente el predicado y objeto si éstos son los que varían.

```
?x  foaf:name  ?name ;
    foaf:mbox  ?mbox .
```

Figura 23. Representación de un patrón con omisión de sujeto.

Además se puede construir un único patrón al que se le especifica una lista de objetos, evitando así, generar un patrón por objeto.

```
?x foaf:nick  "Alice" , "Alice_" .
```

Figura 24. Representación de un patrón con lista de objetos.

Por último, estos dos modos de construcción de patrones se pueden mezclar entre sí, de forma que un patrón del estilo:

```
?x  foaf:name ?name ; foaf:nick  "Alice" , "Alice_" .
```

Figura 25. Representación de la escritura de varios patrones mediante los modos expuestos en las figuras 23 y 24.

Sería lo mismo que escribir lo siguiente:

```
?x  foaf:name  ?name .
?x  foaf:nick  "Alice" .
?x  foaf:nick  "Alice_" .
```

Figura 26. Representación del patrón de la figura 25, escrito de forma extendida.

Por lo tanto, es tarea del PatternAnalizer comprobar todos estos casos de forma que se construyan los patrones utilizando el modo más simple para que éstos sean entendibles por los componentes inferiores.

Una vez el PatternAnalizer termine de analizar los elementos que componen el patrón devolverá una lista con dichos elementos, de forma que, el PatternHandler pueda generar tripletas para pasárselas al PatternBuilder, siendo éste último el encargado de construir el patrón.

El propósito del PatternBuilder es construir, con la tripleta de elementos dada por el PatternHandler, un objeto del tipo Query, que tal y como se comentó en la segunda iteración de desarrollo del proyecto reconoce tanto las Variables como los Términos. Mediante este objeto de tipo Query, el PatternBuilder realiza una traducción de tipo a tipo con la clase Pattern cambiando las Variables por What y los Términos por Identificadores (Constant) a través del TermStore.

Una vez se ha construido el patrón, el PatternHandler realiza por último la tarea de construir un nodo patrón (PatternNode) que contenga el patrón devuelto por el PatternBuilder, y añadirlo como hijo del nodo función que se le pasó inicialmente por parámetro.

Resumiendo, el PatternAnalizer es el encargado de analizar los elementos subsiguientes al elemento identificado como sujeto del patrón, considerando los casos en el que el patrón esté reducido o se haya omitido algún elemento. El PatternBuilder, por otro lado, se encarga de, obtenidos los valores del patrón, traducir y construir el patrón que es manejable y entendible por las clases inferiores de la arquitectura. Por último, el PatternHandler actúa de intermediario entre ambas clases realizando el paso de parámetros entre las mismas y además encargándose de construir el nodo del árbol que contiene el patrón.

Análogamente al modelo definido anteriormente para el reconocimiento de patrones, se pretende utilizar un diseño similar para el reconocimiento de las funciones. Se crearán las clases FunctionHandler, FunctionAnalizer y FunctionBuilder, de forma que su comportamiento sea parecido al planteado en la construcción de patrones.

El FunctionHandler funcionará de la misma forma que su homólogo para el caso de los patrones, siendo fundamental como nexo entre el FunctionAnalizer y el FunctionBuilder, además de ser el encargado de generar el nodo función resultante que irá en el árbol de operaciones. Al contrario que en el caso del análisis de patrones, al FunctionHandler solo se le pasa el índice de la estructura de reconocimiento donde se encuentra el principio de la función, puesto que éste no es hijo de ningún nodo.

La primera tarea del FunctionHandler es la de invocar al FunctionAnalizer. Esta clase analizará el token, y los siguientes desde el índice dado de partida, de forma que sea capaz de discriminar qué función es la que se pretende aplicar. Puesto que en SPARQL las funciones pueden estar formadas por más de una palabra (como en el caso de la función implementada “FILTER NOT EXISTS”), es necesario realizar este reconocimiento de los siguientes elementos.

Una vez el FunctionAnalazer detecta qué función es la que hay en la consulta, éste crea un nodo de la misma, y se lo devuelve al FunctionHandler.

Aquí el FunctionHandler discrimina debe estar preparado para discriminar dos formas de actuar. Si la función reconocida es una función binaria, es decir, tiene patrones asociados, el FunctionHandler debe crear un nodo función hijo Join intermedio que será el que contendrá todos esos patrones asociados. En la siguiente imagen se ilustra una consulta con una operación binaria.

```
PREFIX  rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX  foaf:   <http://xmlns.com/foaf/0.1/>

SELECT ?person
WHERE
{
    ?person rdf:type  foaf:Person .
    FILTER NOT EXISTS { ?person foaf:name ?name .
                        ?person foaf:mbox ?mbox }

}
```

Figura 27. Consulta en SPARQL con un Filter como ejemplo de una función binaria.

Al leer la función, el árbol que debe generar el FunctionHandler es el siguiente.

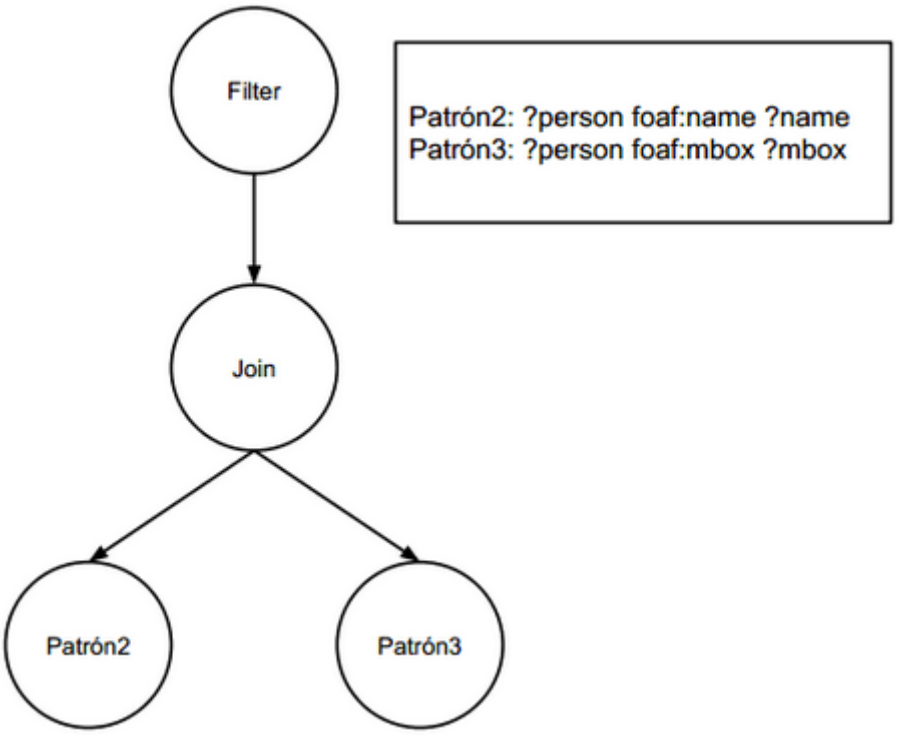

Figura 28. Árbol generado por el FunctionHandler a partir de la consulta de la figura 27.

Como se puede ver en la imagen anterior, los patrones cuelgan exactamente del nodo Join, y éste a su vez, es hijo directo de la función. Este nodo Join es el que se le pasa como parámetro al FunctionBuilder, el cual se encarga de recorrer el ámbito interno de la función, llamando al PatternHandler cuando se encuentre un elemento de un patrón.

Cuando el FunctionBuilder termina, el nodo Join debe poseer tantos hijos como patrones que existen en el ámbito de la función definido en la consulta, es decir, en este punto, ya habríamos generado el árbol de la imagen anterior.

Dicho árbol es devuelto al intérprete, el cual sustituye el nuevo nodo de función por el nodo raíz del árbol, y le adjunta éste como hijo. Hay que tener en cuenta dos aspectos que permiten que este método funcione:

- Toda operación que se aplica es independiente a su localización en la consulta, ya que, las operaciones principalmente filtran los conjuntos de datos o los expanden, sin que afecte en esto el orden de operaciones. Es por ello que todas las operaciones se ejecutan sobre el ámbito más externo.
- El intérprete nunca pierde el nodo Join sobre el que debe añadir como hijos los patrones del ámbito más externo, independientemente si éste es sustituido como nodo raíz o no. Lo cual asegura que, si existe una función entre patrones, los patrones que se lean a continuación de la función, sigan siendo hermanos a los que precedían la función.

Para comprender mejor la estructura de árbol que se genera, utilizaremos el ejemplo anterior de forma que se pueda visualizar el árbol final generado. Así, a partir de la consulta de la Figura 27, será generado un árbol como el que sigue:

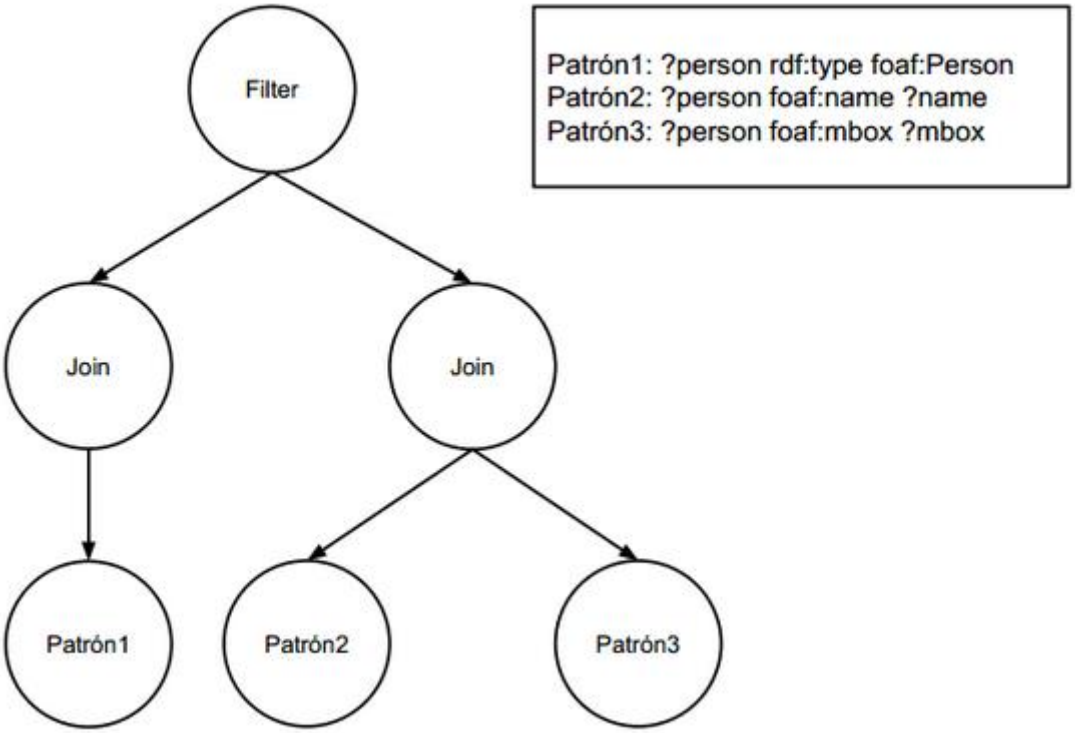

Figura 29. Árbol de operaciones generado por la consulta de la Figura 27.

En el caso de que hubiesen más funciones en la consulta, éstas se seguirían apilando como si fuesen diferentes capas de filtrado. Así en una consulta como la que sigue:

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox ?hpage
WHERE  { ?x foaf:name  ?name .
         OPTIONAL { ?x foaf:mbox ?mbox } .
         OPTIONAL { ?x foaf:homepage ?hpage }
       }
```

Figura 30. Consulta con más de una operación.

Se generaría éste árbol:

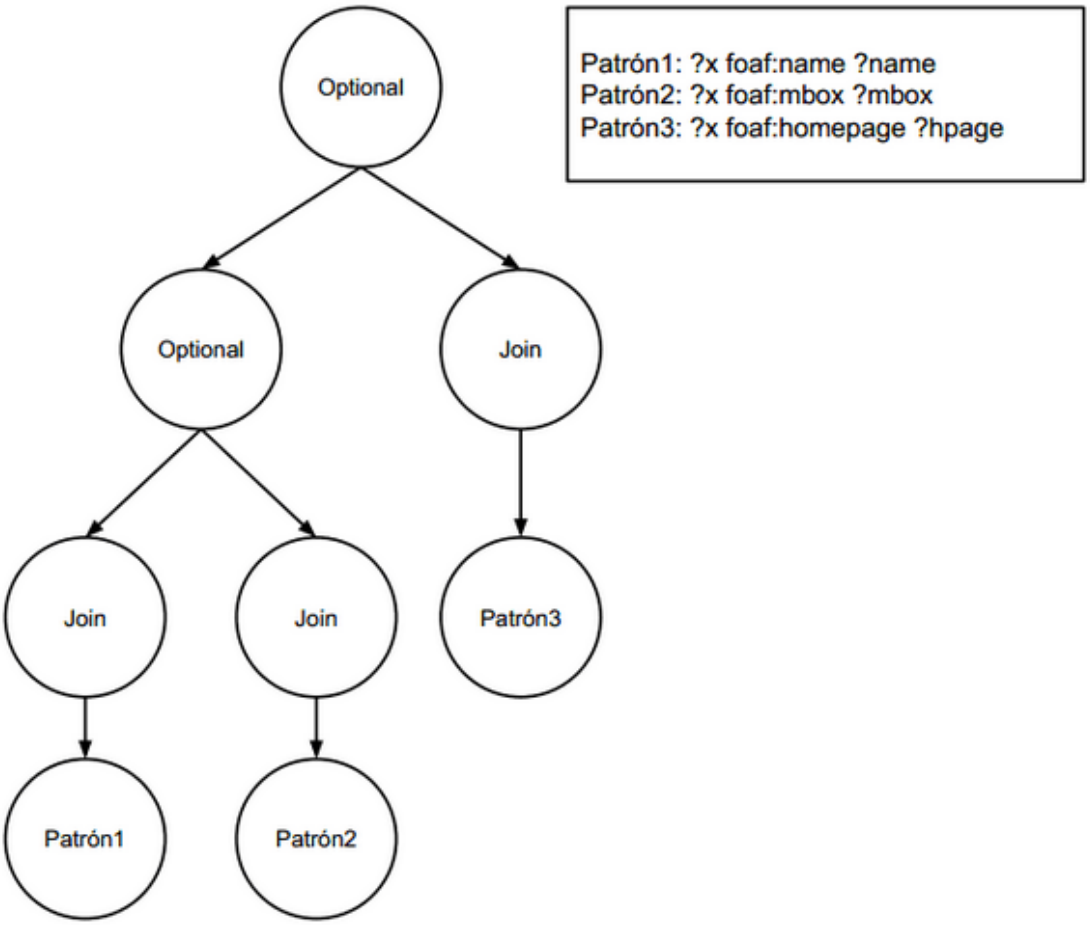

Figura 31. Árbol de operaciones de la consulta de la figura 30.

Por último se decidió realizar un empaquetamiento de forma que las clases se asemejaran a otras interfaces de programación de aplicaciones (APIs), para conseguir una similitud en el manejo y los métodos. En concreto, se pretendió asemejar con la API JDBC, una API diseñada para manejar bases de datos desde Java.

Para ello se creó la clase Statement, similar a la definición existente en JDBC. Dicha clase está pensada para utilizarse de forma muy parecida a la de su homóloga en la API JDBC, donde una clase que controle los almacenamientos fuese capaz de proveer un objeto del tipo Statement, sin necesidad de que se manejen los diferentes almacenamientos de forma externa a la API.

Además, el tipo Statement lanza a ejecutar consultas utilizando la función executeQuery() a la que se le pasa como parámetro una cadena con la consulta.

10. CONCLUSIONES

10.1 CONCLUSIONES

El objetivo de este proyecto era diseñar un intérprete del lenguaje de consulta SPARQL cuya finalidad no quedase únicamente en el análisis léxico y sintáctico de dicho lenguaje, sino que además, fuese capaz de generar un árbol de operaciones. Como objetivo adicional encontrado durante el proyecto se definió la necesidad de realizar un tratamiento de los resultados, tanto los de salida, como los que se obtenían entre medias de la consulta.

Dados estos objetivos, concluimos que los resultados obtenidos son satisfactorios. Se ha conseguido un intérprete del lenguaje de consulta SPARQL capaz de interpretar consultas simples que contengan algunas de las operaciones definidas por el lenguaje.

Durante el desarrollo del proyecto, se han extraído las siguientes conclusiones:

- El lenguaje de consulta SPARQL requiere de mucho estudio y dedicación para poder comprenderlo en su totalidad, entendendiendo las múltiples operaciones y los tratamientos que realiza con las definiciones RDF. Aunque durante la realización de este proyecto se ha dedicado una gran cantidad de tiempo en la lectura y comprensión del estándar de SPARQL propuesto por la World Wide Web Consortium, aún quedan muchos matices y especificaciones del lenguaje pendientes.
- La memoria de los equipos puede repercutir de forma considerable con estructuras poco optimizadas, llegando a ralentizar los procesos de un modo inaceptable.
- A la hora de diseñar software, es esencial seguir una buena praxis de programación, de forma que el código que se implemente sea capaz de aceptar modificaciones sin necesidad de rehacerlo completamente. En proyectos conjuntos, o de investigación, en los cuales puede haber una gran cantidad de cambios un buen software modular, ahorra mucho tiempo.

Por otra parte, merece mención especial las conclusiones obtenidas de las metodologías de desarrollo utilizadas.

La metodología SCRUM, teniendo en cuenta las modificaciones que se realizaron para adaptarla al ámbito del proyecto, es una metodología que fuerza al equipo a trabajar a base de requisitos continuos y con fechas de entrega cortas. Aunque existe presión por esta parte, en un proyecto como el que se llevó a cabo, también es requisito indispensable el compromiso de todos los miembros para con el proyecto, pues, de otra forma no se conseguiría nada.

Pero sin duda, lo más destacable durante la realización del proyecto han sido las prácticas de programación. Tanto TDD como pair programming, han hecho de este proyecto una experiencia de desarrollo muy enriquecedora, no solo por la nueva forma de diseñar el código que nos enseñan, sino también por la forma de afrontar el trabajo.

Asimismo, se consideran alcanzados, de forma satisfactoria, los objetivos planteados para este proyecto.

10.2 TRABAJO FUTURO

Siempre es posible mejorar o ampliar lo ya realizado, es por eso, que en esta sección proponemos diferentes mejoras y líneas de desarrollo con las que se puedan continuar el proyecto:

- Una posible mejora, de cara a eliminar la dependencia generada al incluir la librería ANTLR, es diseñar el analizador sintáctico y léxico de SPARQL. Así evitamos el uso de las clases y código que nos provee ANTLR, cuyo mantenimiento puede resultar muy costoso.
- Mejorar el intérprete de forma que sea capaz de interpretar todas las operaciones de SPARQL que se han quedado fuera de este desarrollo, como por ejemplo, el cierre transitivo.
- Realizar un estudio de rendimiento para el árbol de operaciones, y comprobar si en consultas muy grandes su descenso recursivo y su sustitución de nodos son operaciones óptimas.
- Ampliar el intérprete para que posea la capacidad de trabajar con prefijos en las consultas.
- Por último, adaptar el intérprete para que pueda ser utilizado de forma distribuida y gestione múltiples consultas en paralelo.

11. BIBLIOGRAFÍA

[1] Tecnología RDF, información relativa.

http://es.wikipedia.org/wiki/Resource_Description_Framework

[2] Web semántica, información relativa.

http://es.wikipedia.org/wiki/Web_sem%C3%A1ntica

[3] María J. Lamarca, Hacia la Web Semántica.

http://www.hipertexto.info/documentos/web_semantica.htm

[4] RDF, información relativa.

http://semantizandolaweb.wordpress.com/2011/11/14/introduccion-a-rdf/

[5] W3C: SPARQL Query Language for RDF, Recomendación del 15 de enero 2008 (inglés).

http://www.w3.org/TR/rdf-sparql-query/#sparqlTriplePatterns

[6] W3C: Resource Description Framework (RDF) Model and Syntax Specification. Recomendación del 22 de febrero de 1999 (inglés).

http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/#intro

[7] Uniform resource identifier (URI), información relativa (inglés)

http://en.wikipedia.org/wiki/Uniform_resource_identifier

[8] SCRUM, información relativa.

http://www.proyectosagiles.org/que-es-scrum

[9] TDD, información relativa.

http://www.logiciel.es/tdd

[10] JUnit, información relativa.

http://es.wikipedia.org/wiki/JUnit

[11] ANTLR, información relativa

http://es.wikipedia.org/wiki/ANTLR

[12] ANTLR página oficial (inglés).

http://www.antlr.org/

[13] Comparativa de generadores de parsers actuales (inglés).

http://en.wikipedia.org/wiki/Comparison_of_parser_generators

[14] W3C: SPARQL 1.1 Query Language. Recomendación del 21 de Marzo de 2013 (inglés).

http://www.w3.org/TR/sparql11-query/

[15] Modelo de datos semánticos, información relativa (inglés).

http://en.wikipedia.org/wiki/Semantic_data_model

[16] Triplestore, información relativa (inglés)

http://en.wikipedia.org/wiki/Triplestore

[17] Licencia BSD, Berkeley Software Distribution. Definición de licencia (inglés).

http://www.linfo.org/bsdlicense.html

[18] Creately, página oficial.

https://creately.com