Query Player

A component for Triskel Architecture.

Alumno: Aitor Cardona López

Tutor : José Juan Hernández Cabrera

2013/2014

Grado en Ingeniería Informática

Universidad de las Palmas de Gran Canaria

Trabajo final de grado de la Escuela de Ingeniería Informática de la Universidad de Las Palmas de Gran Canaria presentado por:

Aitor Cardona López

Título del Proyecto: Query Player: A component for Triskel architecture.

Tutor: D. José Juan Hernández Cabrera.

Agradecimientos

Me gustaría dejar constancia de mi agradecimiento hacia las personas que me han permitido llevar este proyecto a buen puerto.

A José Juan Hernández no sólo por brindarme la oportunidad de participar en un proyecto de estas características sino por haberme guiado durante el camino como tutor.

A Octavio Mayor por su asesoramiento durante el desarrollo del proyecto, que no ha podido aparecer como co-tutor oficial debido a temas administrativos.

A Francisco Umpierre, David Santiago, Cristian Casal, Román Díaz y Xerach Hernández que gracias a ellos y su esfuerzo y dedicación Triskel ha salido adelante.

Finalmente a todas esas personas que han sabido hacerme tener días mejores durante esta travesía, animándome y motivándome para conseguir llegar hasta aquí.

A todos, mi más sinceros agradecimientos...

Aitor.

Índice

1. Resumen

Uno de los cinco componentes de la arquitectura triskel, una base de datos NoSQL que trata de dar solución al problema de Big data de la web semántica, el gran número de identificadores de recursos que se necesitarían debido al creciente número de sitios web, concretamente el motor de gestión de ejecución de patrones basados en tripletas y en la tecnología RDF. Se encarga de recoger la petición de consulta por parte del intérprete, analizar los patrones que intervienen en la consulta en busca de dependencias explotables entre ellos, y así poder realizar la consulta con mayor rapidez, además de ir resolviendo los diferentes patrones contra el almacenamiento, un TripleStore, y devolver el resultado de la petición en una tabla.

2. Abstract

One of the five components of the triskel architecture, a NoSQL data base that tries to solve the problem of big data in semantic web due to the large number of resources identifiers that would be needed because of the increasing number of web sites, to be precise the management engine running patterns based on triplets and RDF technology standards. This component has the responsability of collecting the query request by the interpreter, analyze the patterns involved in the query for exploitable dependencies between them in order to make the query more quickly and solving the different patterns against the storage, a triplestore, returning the result of the request in a table.

3. Motivación

Una base de datos RDF, Almacenamiento de datos RDF y TripleStore son tres conceptos con el mismo significado, un sistema de bases de datos especial desarrollado para el almacenamiento y recuperación de sentencias RDF, cada sentencia es un pequeño registro denominado tripleta, siendo una tripleta una entidad de datos compuesta por sujeto-predicado objeto, como “Juan compra coche” o “Juan abona pagaré”, un TripleStore está optimizado para el almacenamiento y recuperación de tripletas así como para realizar consultas al almacenamiento, la consultas son realizadas en SPARQL, lenguaje de consulta para conjuntos de datos RDF.

Pero desarrollar un TripleStore es una tarea más que compleja para sólo una persona, en cambio, para un equipo de trabajo se plantea una tarea mucho más asequible y si a todo eso añadimos una gestión basada en metodologías ágiles obtenemos un buen terrero para explorar el mundo de desarrollar un TripleStore.

El conjunto de todas estas variables ha sido lo que me ha movido para llevar a cabo este proyecto.

4. Estado del arte

Permitámonos el lujo de dar una mirada al pasado y recordar el inicio de internet, al temprano desarrollo de las redes de comunicación. La idea consistía en conseguir una red de ordenadores destinada a la comunicación general entre usuarios de varios ordenadores.

Las versiones más antiguas de estas ideas aparecieron a finales de los años 50, aunque las implementaciones prácticas no llegaron hasta finales de los 60 y a lo largo de los 70, finalmente en la década de los 80 las tecnologías que reconocemos hoy en día como la base del moderno internet comenzaron a expandirse por el mundo.

La infraestructura de internet creció a lo largo y ancho del globo, dando nacimiento a la red mundial de ordenadores que hoy conocemos, saltando de país en país creó un acceso mundial a la información y comunicación sin precedentes.

En los años 60 comenzó la web 1.0, una web en su formato más básico destinada a navegadores sólo de texto, como por ejemplo ELISA, con el paso del tiempo llego HTML (Hyper Text Markup Language) lo que permitió el desarrollo de páginas web mucho más agradables a la vista, cabe destacar que la web 1.0 es sólo lectura, privando al usuario de la capacidad de interactuar con el contenido de la página en cuestión.

El término Web 2.0 fue acuñado por Tim O'Reilly en 2004 para referirse a la segunda generación en la historia del desarrollo de la tecnología web basada en la comunicación entre usuarios, nuevos servicios en internet como podían ser blogs, wikis o redes sociales fomentan la colaboración entre usuarios en el intercambio de información.

Sin embargo, la web como la conocemos, es un gigantesco repositorio de hiperdocumentos cuyo diseño únicamente permite su comprensión por humanos. Estos materiales están confeccionados usando lenguajes de etiquetado que expresan la forma en que los navegadores deben presentar su contenido (colores, maquetación, fuentes, etc.) y no su significado o semántica. Dado el gigantesco y creciente número de estos recursos, los actuales motores de búsqueda encuentran limitaciones a la hora de ofrecer tasas de precisión mínimamente adecuadas en sus resultados, evidenciando que las técnicas léxico-estadísticas no pueden solucionar por sí solas la problemática de la recuperación de información.

Las Web semántica ayuda a resolver estas dos grandes incógnitas partiendo de la base de hacer la actual versión de la web no sólo comprensible por lo seres humanos, como había sido hasta ahora, sino también permitir que estén disponibles de una manera formal para los sistemas inteligentes.

Los elementos que nos permiten obtener una definición adecuada de los datos dando vida a la web semántica son RDF (Resourse Description Framework), SPARQL (Protocol and RDF Query Language) y OWL (Web Ontology Language), de los cuales hablaremos a continuación más en detalle.

Si quisiéramos hacer una equivalencia con un sistema de bases de datos relacionales esta podría quedar de la siguiente forma:

- RDF equivale a los registros de una base de datos, descrito mediantes proposiciones simples, sujeto predicado objeto, también conocido como tripleta.
- SPARQL equivaldría al lenguaje SQL.
- OWL equivaldría al esquema de la base de datos.

Sin embargo, uno de los problemas con los que nos encontramos, dado el gran número de páginas web existentes es la enorme cantidad de recursos descritos, lo que nos lleva a un conjunto de datos que superan la capacidad del software habitual para ser capturados, gestionas y procesados en un tiempo razonable, a este tipo de datos se le conoce como “Big data”.

La problemática de este conjunto de datos ha sido encauzada hacia bases de datos especializadas en el almacenamiento y consulta de tripletas, este tipo de bases de datos reciben el nombre de TripleStore y es donde nuestro proyecto tiene lugar.

Actualmente existen diversas implementaciones de TripleStore, como podrían ser Apache Jena, Mulgara o 4Store.

5. Objetivos

5.1 Generales

Desarrollar un sistema de almacenamiento de tripletas ó TripleStore, que además permita la consulta de las tripletas afirmadas, bautizado como “Triskel Data Base”. Para llevar a cabo esta tarea el proyecto contará de cinto componentes:

- Builder.
- TripleStore.
- TermStore.
- Interpreter.
- Query Player.

Siguiendo un esquema de gestión de proyecto basado en tareas, se pretende fomentar el trabajo en equipo, la participación activa en la especificación, diseño e implementación de las propuestas que vayan saliendo a la luz durante las reuniones, además de la capacidad de análisis de problemas complejos y llevar a cabo un plan de trabajo previamente diseñado aplicando metodologías de desarrollo ágil.

5.2 Específicos

Desarrollar Triskel Query Player, un motor de gestión de ejecución de patrones basados en tripletas y en la tecnología RDF, así como la creación de los resultados devueltos y la consulta con el almacenamiento a más bajo nivel, además que sea independiente de la semántica de los datos y que permita resolver las diferentes consultas.

Se perseguirá fomentar la integración en un equipo de trabajo, la capacidad de toma de decisiones ante un problema determinado, el posterior análisis de resultados una vez llevada a la práctica la solución del problema y dar solución a problemas de integración en función de los compromisos de diseño.

6. Metodología

El desarrollo de software es una tarea laboriosa, prueba de ello es la gran cantidad de propuestas metodológicas existentes. Por un lado tenemos las metodologías tradicionales centradas básicamente en el control del proceso, estableciendo rigurosamente las actividades implicadas, artefactos a producir y herramientas a usar. Las propuestas tradicionales han demostrado su eficacia a lo largo del tiempo, pero no por ello han estado exentas de inconvenientes.

Con el paso del tiempo surgió otra aproximación, centrándose más en las personas. Esta es la filosofía de las metodologías ágiles, las cuales dan una mayor relevancia al individuo, la colaboración con el cliente y al desarrollo incrementar con iteraciones cortas. Este enfoque pone a relucir todo su potencial en ambientes de requisitos muy volátiles.

Las metodologías ágiles están revolucionando la forma de desarrollar software, al mismo tiempo que el debate entre sus seguidores y detractores se incrementa.

6.1 Manifiesto ágil

El 17 de febrero de 2001 diecisiete críticos de los modelos de mejora del desarrollo de software basados en procesos, convocados por Kent Beck, quien había publicado un par de años antes Extreme Programming Explained, libro en el que exponía una nueva metodología denominada Extreme Programming, se reunieron en Snowbird ,Utah, para tratar sobre técnicas y procesos para desarrollar software. En la reunión se acuñó el término "Métodos Ágiles" para definir a los métodos que estaban surgiendo como alternativa a las metodologías formales (CMMI, SPICE) a las que consideraban excesivamente "pesadas" y rígidas por su carácter normativo y fuerte dependencia de planificaciones detalladas previas al desarrollo.

Estamos descubriendo formas mejores de desarrollar software tanto por nuestra propia experiencia como ayudando a terceros. A través de este trabajo hemos aprendido a valorar:

- Valoramos más a los individuos y su interacción que a los procesos y las herramientas.

  Este es el valor más importante del manifiesto. Por supuesto que los procesos ayudan al trabajo. Son una guía de operación. Las herramientas mejoran la eficiencia, pero hay tareas que requieren talento y necesitan personas que lo aporten y trabajen con una actitud adecuada.

- Valoramos más el software que funciona que la documentación exhaustiva.

  Poder anticipar cómo será el funcionamiento del producto final, observando prototipos previos, o partes ya elaboradas ofrece un "feedback" estimulante y enriquecedor, que genera ideas imposibles de concebir en un primer momento, y difícilmente se podrían incluir al redactar un documento de requisitos detallado en el comienzo del proyecto.

- Valoramos más la colaboración con el cliente que la negociación contractual.

  Las prácticas ágiles están indicadas para productos cuyo detalle resulta difícil prever al principio del proyecto; y si se detallara al comenzar, el resultado final tendría menos valor que si se mejoran y precisan con retroinformación continua.

- Valoramos más la respuesta al cambio que el seguimiento de un plan.

  Para desarrollar productos de requisitos inestables, que tienen como factor inherente el cambio y la evolución rápida y continua, resulta mucho más valiosa la capacidad de respuesta que la de seguimiento y aseguramiento de planes. Los principales valores de la gestión ágil son la anticipación y la adaptación, diferentes a los de la gestión de proyectos ortodoxa: planificación y control que evite desviaciones del plan.

Además nos basamos en doce principios.

1. Nuestra principal prioridad es satisfacer al cliente a través de la entrega temprana y continua de software de valor.

2. Son bienvenidos los requisitos cambiantes, incluso si llegan tarde al desarrollo. Los procesos ágiles se doblegan al cambio como ventaja competitiva para el cliente.

3. Entregar con frecuencia software que funcione, en periodos de un par de semanas hasta un par de meses, con preferencia en los periodos breves.

4. Las personas del negocio y los desarrolladores deben trabajar juntos de forma cotidiana a través del proyecto.

5. Construcción de proyectos en torno a individuos motivados, dándoles la oportunidad y el respaldo que necesitan y procurándoles confianza para que realicen la tarea.

6. La forma más eficiente y efectiva de comunicar información de ida y vuelta dentro de un equipo de desarrollo es mediante la conversación cara a cara.

7. El software que funciona es la principal medida del progreso.

8. Los procesos ágiles promueven el desarrollo sostenido. Los patrocinadores, desarrolladores y usuarios deben mantener un ritmo constante de forma indefinida.

9. La atención continua a la excelencia técnica enaltece la agilidad.

10. La simplicidad como arte de maximizar la cantidad de trabajo que se hace, es esencial.

11. Las mejores arquitecturas, requisitos y diseños emergen de equipos que se auto gestionados.

12. En intervalos regulares, el equipo reflexiona sobre la forma de ser más efectivo y ajusta su conducta en consecuencia.

6.2 Desarrollo basado en pruebas

Desarrollo basado en pruebas (TDD) es un enfoque evolutivo para el desarrollo, que combina el desarrollo de la prueba primero en el que escribe una prueba antes de escribir el suficiente código de producción para cumplir con esa prueba y una posterior refactorización.

¿Cuál es el objetivo principal de TDD? Un punto de vista es que el objetivo de TDD es la especificación y no validación .En otras palabras, es una forma de pensar a través de sus necesidades o de diseño antes de escribir su código funcional, otro punto de vista es que TDD es una técnica de programación. Como Ron Jeffries le gusta decir, el objetivo de TDD es escribir código limpio que funciona.

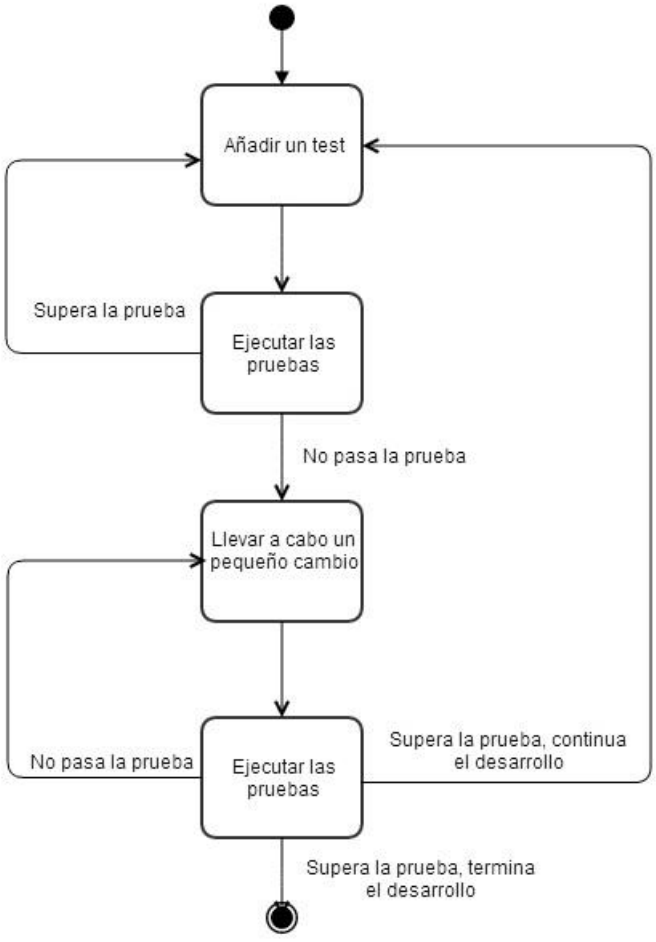


Ilustración 1

6.3 Pair-Programming

La Programación en Pareja (o Pair-Programming en inglés) 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, lo cual se ajusta perfectamente a la intención de usar desarrollo basado en la prueba.

Entre sus ventajas podemos destacar una aumento en la disciplina del trabajo acompañado de un código de mejor calidad en primera instancia, ya luego con tareas de refactor se intentaría dejar el código lo más limpio posible, se puede mantener un mejor flujo de trabajo durante más tiempo, obviamente con sus respectivos "breaks".

Por otro lado debemos tener en cuenta que desarrollando en parejas la codificación, para algunos desarrolladores, se hace más agradable y si a eso unimos la posibilidad de ir aprendiendo los unos de los otros se obtiene un ambiente de trabajo con una moral mejorada y mayor cohesión.

6.4 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.

En Scrum se realizan entregas parciales y regulares del producto final, priorizadas por el beneficio que aportan al receptor del proyecto. Por ello, esta metodología está especialmente recomendada 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.

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.

Teniendo en cuenta lo anteriormente expuesto, Scrum nos aportaba el nicho adecuado para llevar a cabo la gestión de proyecto de una forma organizada, cierto es que no hemos aplicado esta metodología al completo, no obstante, hemos seguido el estilo del trabajo basado en tareas, con sus responsables, estimación en tiempo, requisito y validación, adema de reuniones diarias entre los integrantes del equipo de desarrollo y reuniones periódicas con nuestros tutores.

7. Recursos

Para poder dar vida al componente de Triskel, Query Player, y a la documentación pertinente se ha hecho necesario el uso de los siguientes recursos:

Hardware

- Procesador Intel(R) Core(TM) i3-2120 CPU@3,30Ghz
- Memoria Ram: 4,00 GB DDR2

Sistema Operativo

- Windows 7 Enterprise

Software de desarrollo

- IDE NetBeans 7.2
- Java 7
- JUnit

Software de apoyo

- Google Chrome
- Notepad++
- BitBucket
- Git
- Gliffy: herramienta de diagramas online.

Software de documentación

- Microsoft office 2010
- Microsoft Excel 2010
- StarUML

Logísticos

- Zona de desarrollo
- Zona de reuniones

8. Conceptos

8.1 RDF

El Marco de Descripción de Recursos (del inglés Resource Description Framework, RDF) es una familia de especificaciones de la World Wide Web Consortium (W3C) originalmente diseñado como un modelo de datos para metadatos. Ha llegado a ser usado como un método general para la descripción conceptual o modelado de la información que se implementa en los recursos web, utilizando una variedad de notaciones de sintaxis y formatos de serialización de datos.

El modelo de datos RDF es 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 (en particular, recursos web) en forma de expresiones sujeto-predicado-objeto. Estas expresiones son conocidas como triples en terminología RDF. El sujeto indica el recurso y el predicado denota rasgos o aspectos del recurso y expresa una relación entre el sujeto y el objeto.

RDF está destinado a ser publicado en internet, siendo los nombres para sujetos, predicados y objetos deben ser Identificadores Uniformes de Recursos (URIs). (Técnicamente, se pueden llamar Identificadores de Recursos Internacionalizados pero la distinción no es importante).

8.2 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 normalizado por el RDF Data Access Working Group (DAWG) del World Wide Web Consortium (W3C). Es una tecnología clave en el desarrollo de la Web Semántica que se constituyó como Recomendación oficial del W3C el 15 de Enero de 2008.

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 ligadas a entornos de desarrollo y plataformas tecnológicas.

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

8.3 Big Data

"Big data" es un término aplicado a conjuntos de datos que superan la capacidad del software habitual para ser capturados, gestionados y procesados en un tiempo razonable. Los tamaños del " Big data" se encuentran constantemente en aumento. En 2012 se dimensionaba su tamaño en una docena de terabytes hasta varios petabytes de datos en mismo único data set.

8.4 Tripleta

RDF es un lenguaje genérico para describir recursos, es decir, para identificar unívocamente entidades (como podrían ser compañías, películas o personas) y relaciones o hechos (facts) sobre las mismas.

Cuando la W3C se propuso escribir las especificaciones de RDF, su objetivo era crear una fórmula genérica para representar el conocimiento humano. Se trataba de crear una sintaxis tan amplia que permitiese expresar cualquier hecho (fact), y a la vez tan estructurada que cualquier software pudiese interpretarlo de manera automática.

Para conseguirlo, la fórmula más básica que lograron sintetizar fueron los tripletes o tripletas. Los tripletes son "piezas" de conocimiento que tienen la siguiente estructura:

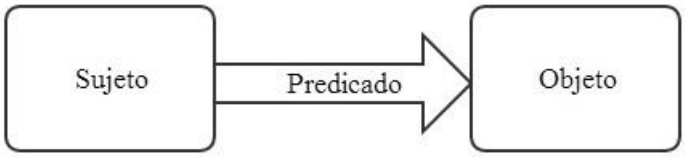


Ilustración 2

- **Sujeto:** indica la entidad sobre la que se describe el hecho (ejemplo: Juan).
- **Predicado:** indica el tipo de hecho a describir (ejemplo: tiene un padre).
- **Objeto:** indica un valor adicional que ayuda a completar el hecho (ejemplo: Martín).

Como se puede apreciar una tripleta es una versión simple de una oración simplificada, así es como los seres humanos representamos nuestro conocimiento. RDF no es más que un estándar para dar un paso más hacia una mejor comprensión de nuestro conocimiento por parte de las máquinas.

Los sujetos y predicados deben ser recursos a los que se pueda hacer referencia, es decir, deben ser identificados mediante una URI (Identificador Único de Recurso). Sin embargo, los objetos pueden ser referenciables o pueden ser valores literales (como strings, fechas, números… etc.).

8.5 Patrón

Se denomina patrón a una tripleta donde uno o varios de sus elementos presentan una incógnita denominada “What”, así es como se denominan en SPARQL a las variables, además los “What” constan de un nombre para poder ser identificados. El patrón es el mínimo componente necesario para realizar una consulta contra el almacenamiento. Además los patrones pueden presentar dependencias entre ellos, lo que nos abre una línea de exploración hacia nuevas formas de gestión de la ejecución.

8.6 TripleStore

Un TripleStore es una base de datos especialmente diseñada para el almacenamiento y recuperación de tripletas, siendo una triple una entidad de datos compuesta de sujeto-predicado-objeto. Al igual que una base de datos relacional, uno almacena la información en una tienda de triple y recuperarlo a través de un lenguaje de consulta. A diferencia de una base de datos relacional, un TripleStore está optimizado para el almacenamiento y recuperación de triples. Algunas TripleStore pueden almacenar miles de millones de triples.

9. Plan de trabajo

La forma de gestionar el plan de trabajo concerniente al componente Query Player ha sido dividir el proceso en cuatro etapas, la cuales veremos a continuación:

Primera etapa

Estado del arte y contextualización: consiste en una familiarización con los distintos conceptos que intervienen en el ámbito del proyecto, además de una valoración de las tecnologías ya existentes y de cómo podrían afectar al proyecto en sí, con una duración estimada de unas veinticinco horas.

Segunda etapa

Estudio de herramientas: toma de contacto con las herramientas que van a ser empleadas durante el desarrollo del componente, además esta etapa está estimada en unas quince horas.

Tercera Etapa

Implementación: Comprende las cuatro iteraciones en las que se ha dividido el desarrollo del componente, en cada una de estas fases se ha planteado un problema específico, estimando una duración aproximada de doscientas horas.

Iteraciones

- Implementación de la búsqueda de patrones.
- Implementación de las operaciones entre patrones.
- Adaptación de la búsqueda de patrones a nuevas estructuras e implementación de la búsqueda bidimensional.
- Implementación y comparativa de diferentes ejecuciones de consultas.

Cuarta etapa

Documentación: Diseño y redacción del documento concerniente al componente, que comprende desde el estado del arte a las conclusiones obtenidas después del desarrollo del componente pasando por la cronología del desarrollo de cada una de las fases en las que fue dividido, la rodaja de tiempo estimada para este apartado sería de unas sesenta horas.

Cabe destacar que la tercera etapa se gestionará siguiendo la Metodología Ágil SCRUM, a la que se hace referencia en el apartado metodología, no obstante, no ha sido una aplicación pura de dicha metodología, dado que se ha adecuado SCRUM a los requisitos del entorno de trabajo.

9.1 SCRUM en la práctica

Dadas las características del proyecto nos era posible usar un Product Backlog, documento de alto nivel para todo el proyecto el cual contiene descripciones genéricas de todos los requisitos, funcionalidades deseables, etc., como consecuencia tampoco no era posible tener el Burn down chart, gráfica que muestra los requisitos del backlog concluidos, por ello nos centramos en aplicar SCRUM de la siguiente manera.

Llegamos al compromiso que un sprint tendría una duración de una semana, con su correspondiente reunión de sprint con el SCRUM master, rol tomado por el tutor del proyecto, al inicio de la semana. Durante las reuniones se pondría de manifiesto los objetivos superados e inconvenientes sufridos durante el pasado sprint, para a continuación establecer los nuevos hitos para el siguiente sprint, además de la reunión con el SCRUM master el equipo de desarrollo se reunía para desglosar los nuevos requisitos en tareas, estimarlas y asignar a sus responsables. Una vez hecho esto el equipo de desarrollo se reunía una vez al día para comentar los avances e inconvenientes que se iban encontrando durante el desarrollo de las tareas asignadas.

10. Cronología del desarrollo

A lo largo de esta sección se lleva a cabo un recorrido por las diferentes iteraciones del componente QueryPlayer, cada iteración consta de un apartado, análisis del problema, donde se expone el problema a abordar durante la iteración y de otro apartado donde se encuentra recogido el diseño y la implementación que se llevó a cabo para cada uno de los diferentes problemas que nos fuimos encontrando a lo largo del desarrollo del componente.

10.1 Implementación de la búsqueda de patrones

Análisis del problema

Tripleta, compuesta de sujeto, predicado y objeto, son guardadas en el almacenamiento, el primer paso dentro de esta iteración fue comprobar si una tripleta estaba contenida en el almacenamiento, es decir, el patrón más sencillo de todos [x,y,z] como por ejemplo [“Juan”, “es”, “Padre”] del que podríamos saber si se encuentra o no contenido en los dominios del almacenamiento.

Sin embargo [x,y,z] no es la única nomenclatura para un query que se aceptaría, pues pasamos a tener consultas más complejas, en las cuales cualquiera de sus componentes, sujeto predicado u objeto podían venir definidos por una tupla de sujetos, predicados u objetos, como por ejemplo podría ser [{“Juan”, “José”}, “es”, ”Varón”], donde en el sujeto tenemos la tupla {Juan, José} ,que englobaría una consulta del tipo [“Juan”, “es”, “Varón”] y [“José”, “es”, “Varón”] y para cada una de ellas se debería obtener su correspondiente respuesta.

Sin embargo, consultar si una tripleta o no está contenida en el almacenamiento no es suficiente, otro hito al que llegar sería la de obtener información del almacenamiento, ser capaces de preguntar por el conjunto sujetos relacionado con un predicado y objeto determinado, los predicados relacionados con cierto sujeto y objeto y finalmente los objetos relacionados con un sujeto y un predicado.

Recapitulando, en esta iteración nos enfrentamos a tres problemas

- Consulta simple.
- Consulta compuesta.
- Consulta para obtener información del almacenamiento.

Diseño e implementación

El primer paso se encaminó a dar formato a la consulta, ¿De qué consta el patrón? Sus elementos vienen a ser sujeto, predicado y objeto, en una primera aproximación decidimos recoger la consulta por partes, primero el sujeto, luego el predicado y después el objeto, de esta manera ya teníamos todos los elementos del patrón diferenciados.

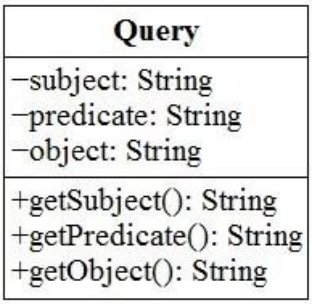


Ilustración 3

Sin embargo, el almacenamiento no entiende ristras pues almacena identificadores para cada uno de los valores que intervienen en una tripleta, por ello se hacía necesario pasar por un proceso de traducción de la consulta, pasando de ristra a identificador, esto podíamos hacerlo gracias al diccionario, otro de los componentes de Triskel Data Base, encargado, entre otras cosas, de la traducción de identificadores.

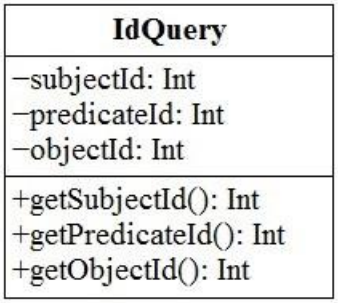


Ilustración 4

La clase IdQuery era prácticamente el modelado de una tripleta, por ello en un posterior refactor dejó de existir como intermediario y se pasó a trabajar directamente con tripletas, las cuales denominaríamos “Facts”, al ser hechos que podían estar contenidos en el almacenamiento.

Una vez obtenida la consulta en forma de identificadores ya estábamos listos para realizar la consulta, no obstante, no podemos acceder a la estructura interna del almacenamiento para buscar la tripleta, para ello solicitamos un método al almacenamiento que nos permitiera comprobar si una tripleta se encontraba en sus dominios, favoreciendo así la modularidad del software así como el principio de responsabilidad única, de esta manera el método conocido como "matchPattern" el cual solicita un identificador de sujeto, otro de predicado y uno de objeto nos indicaba si existía o no la tripleta en cuestión.

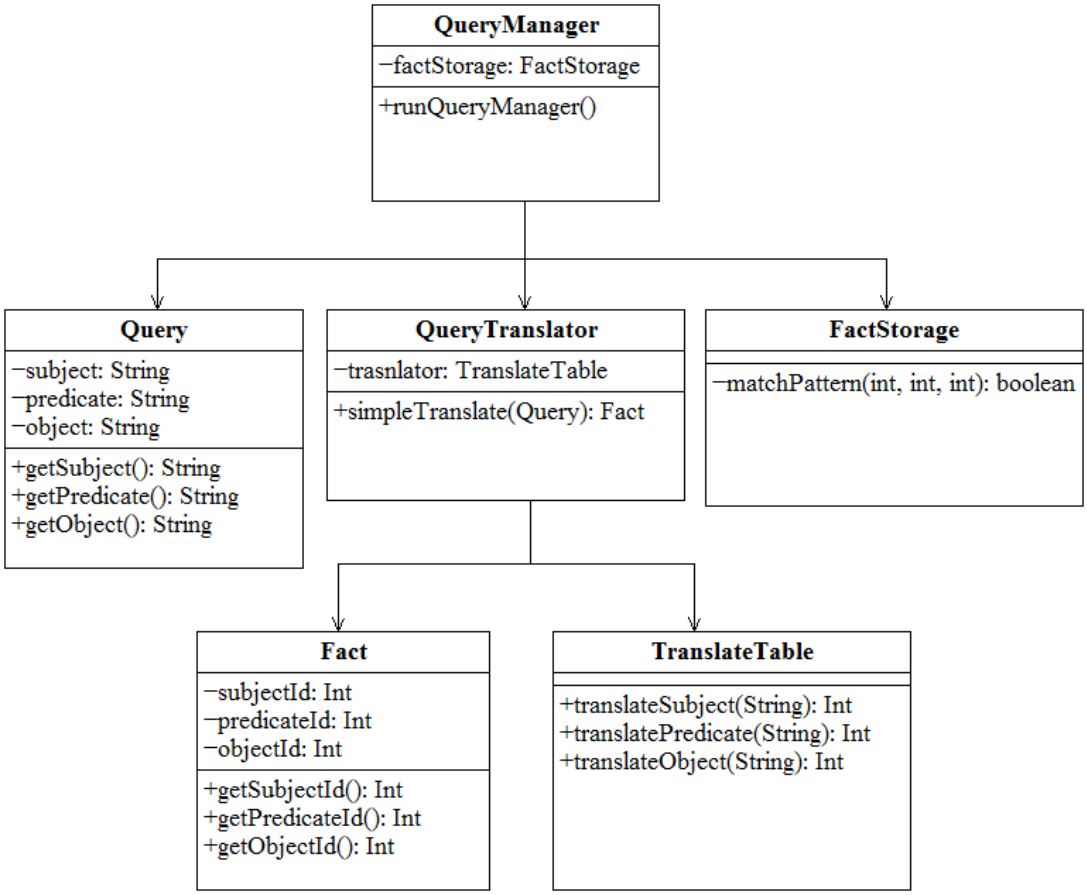


Ilustración 5

La clase QueryManager se encargaba se solicitar los parámetros del Query, una vez obtenido eran traducidos a un hecho (Fact) y comprobado en el almacenamiento mediante matchPattern.

Llegado a este punto somos capaces de ejecutar consultas sencillas, luego ya estábamos en un buen momento para comenzar con consultas más complejas, estas son, las consultas compuestas, pero ¿Qué entendemos por una consulta compuesta? Pues toda aquella consulta en la que uno o varios de sus componentes ya sea sujeto, predicado y objeto viene definido por una tupla de valores, por ejemplo podría ser [{"María", "José"}, "es", "Varón"], donde en el sujeto tenemos la tupla {María, José} ,que englobaría una consulta del tipo ["María", "es", "Varón"] y ["José", "es", "Varón"].

La forma de afrontar este nuevo hito del desarrollo consistió en descomponer la consulta compleja en tantas consultas simples fuera necesario, así pues, en el momento de recoger la consulta pasaríamos de recoger una única ristra por sujeto, predicado y objeto, a recoger una tupla con al menos un elemento para luego descomponerla en todas las tuplas básicas que fuera necesario, de esta manera obtuvimos una lista de consultas simples que evaluar en el almacenamiento.

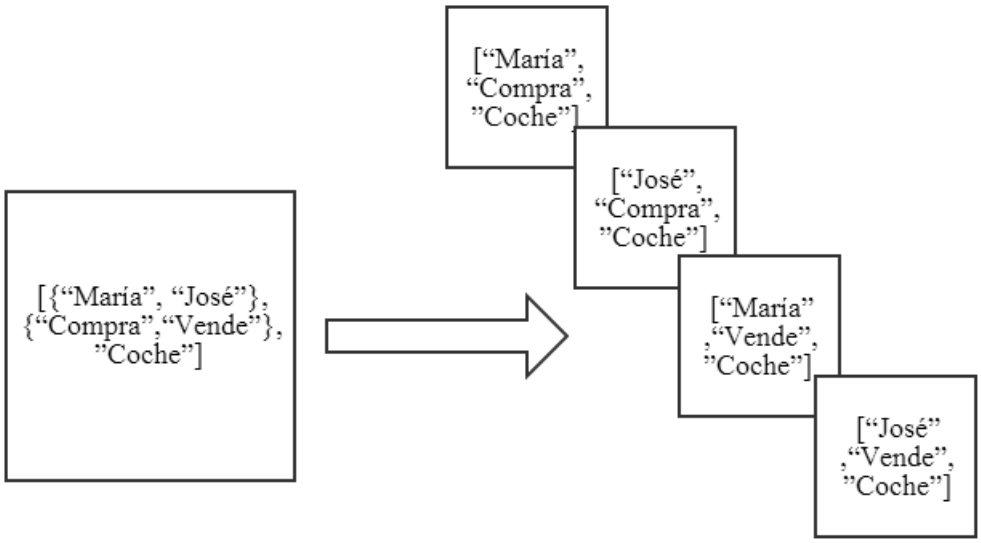


Ilustración 6

Para llevar a cabo esta actualización, la clase Query pasaría a tener una lista de queries básicas, de tal manera que podríamos ir solicitando a la clase Query la siguiente Query básica a resolver.

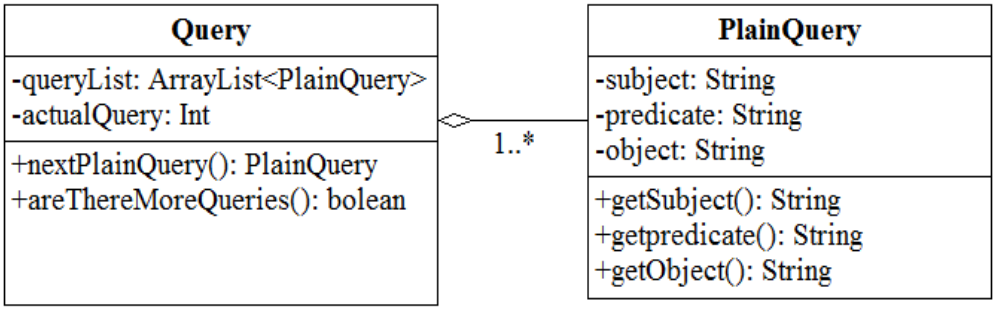


Ilustración 7

Además la clase QueryManager también se vería afectada, ahora también debería recoger las tuplas de sujetos, predicados y objetos que se le facilitasen.

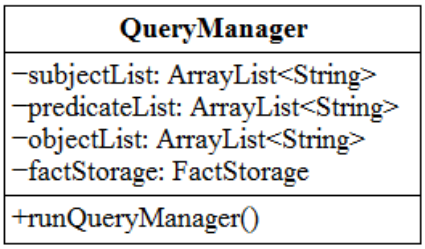


Ilustración 8

Con las consultas compuestas resueltas, comenzamos con la tarea de conseguir extraer información del almacenamiento, estas consultas recibirían el nombre de unidimensionales, las consultas unidimensionales se caracterizan porque uno de los elementos de patrón es una incógnita, en SPARQL se denominan a las incógnitas “What”, así que adoptamos ese nombre para nuestras incógnitas, lo que dio lugar a un nuevo tipo de patrones, en una primera aproximación decidimos denotar a las incógnitas o “Whats” por un interrogante, ya que hacían alusión a los elementos solicitados por la consulta, los nuevos patrones tenían la siguiente forma, [“?”, “x”, “y”] aunque también podían ser patrones compuestos [“?”, {“x”, “z”}, “y”], los cuales se afrontarían de forman análoga a las consultas compuestas sin “Whats”, esto es, para la consulta [“?”, {“x”, “z”}, “y”] tendríamos que dar respuesta a [“?”, {“z”}, “y”] y [“?”, {“x”}, “y”].

Durante la primera aproximación tan sólo contábamos con la función de comprobar si una tripleta se encontraba en el almacenamiento o no, por lo que tuvimos que arreglarnos con lo que teníamos, la forma de proceder consistió en que solicitamos al diccionario todo el conjunto de identificadores válidos para el elemento por el que se preguntaba, lo que vendría a ser todos los sujetos, predicados u objetos , según fuera la consulta, que estuviesen registrados en el diccionario, una vez obtenidos estos datos comenzábamos un proceso de construcción de tripletas, para cada uno de los identificadores facilitados por el diccionario generábamos una consulta simple que después sería evaluada contra el almacenamiento por medio de la función de “patternMaching”, el conjunto de tripletas que habían sido validadas como verdaderas era el conjunto resultado de la consulta, pasa por un proceso de traducción de identificador a ristra y era mostrado al usuario.

En el diagrama que se muestra a continuación se muestra la actualización que sufrió el software de cara a resolver las búsquedas unidimensionales, en caso de que se tratase de una búsqueda unidimensional el proceso de traducción que se desencadenaría sería el complexTranslate, dado que como se ha expuesto anteriormente se solicitarían todas los identificadores de sujeto, objeto o predicado, según se necesitase, al diccionario.

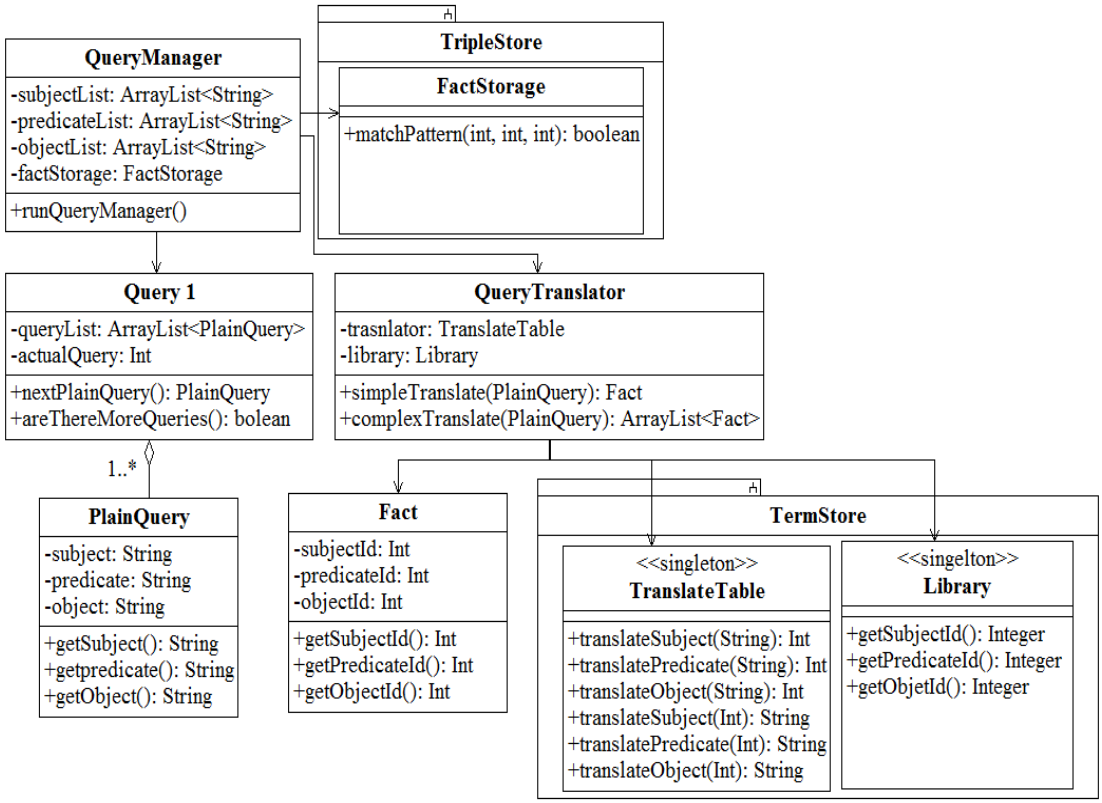


Ilustración 9

En el diagrama de la ilustración 9 se ha abusado del concepto de subsistema para representar fielmente la delimitación entre los diferentes componentes de la arquitectura Triskel.

Sin embargo, conforme el almacenamiento iba evolucionando quedó rápidamente obsoleta, pues en una segunda aproximación solicitamos al almacenamiento que nos proporcionase la forma de realizar búsquedas parciales, es decir, búsquedas unidimensionales, esto dió lugar a que se delegara la responsabilidad de devolver el conjunto de valores por los que se estaba preguntando al almacenamiento, de esta manera por medio de una búsqueda parcial el almacenamiento nos devolvería el conjunto de sujetos, objetos o predicados que coincidieran con la relación establecida en la consulta.

En un posterior refactor las operaciones anteriormente solicitadas al almacenamiento para preguntar si una tripleta se encontraba en el almacenamiento, y para realizar consultas unidimensionales pasaron a formar parte de una interfaz de comunicación entre los componentes de consultas y el almacenamiento, concluyendo así la primera iteración.

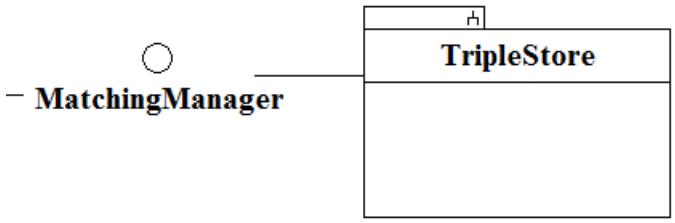


Ilustración 10

10.2 Implementación de las operaciones entre patrones

Análisis del problema

Llegados a este punto nos comenzamos a plantear las operaciones entre patrones, debíamos ser capaces de realizar una unión y una intersección entre los conjuntos de datos devueltos por dos patrones, además de las proyecciones de sujeto, predicado y objeto respectivamente.

Una tupla se define como una función finita que asocia unívocamente los nombres de los atributos de una relación con los valores de una instanciación de la misma. Lo que vendría a ser una columna de una tabla relacional.

Proyección

Permite extraer columnas de una relación, dando como resultado un subconjunto vertical de atributos de la relación, en nuestro caso correspondería al conjunto de datos perteneciente al sujeto, al predicado o al objeto.

Unión

Nos permite obtener el conjunto de tuplas que se encuentran en el primer patrón o en el segundo patrón, o en ambos.

Intersección

Al igual que en teoría de conjuntos, la intersección corresponde al conjunto de todas la tuplas que están en el primer patrón y que también están en el segundo patrón.

Recapitulando, los objetivos para esta iteración son:

- Implementar la Unión de patrones.
- Implementar Intersección de patrones.
- Implementar Proyección de patrones.

Diseño e implementación.

Con el comienzo de la segunda iteración nos centramos en el desarrollo de las operaciones entre patrones, siendo la proyección la primera en llevarse a cabo, concretamente la proyección para sujetos, para poder diseñar dicha operación debíamos tener en cuenta que se trata de una operación unaria aplicada sobre el conjunto resultante de resolver un patrón, luego para una lista de hechos obtenida del almacenamiento se pasaba el filtro para sujetos, obteniendo así el conjunto de sujetos de la lista de tripletas resultado proveniente de la resolución exitosa de un patrón contra el almacenamiento, de manera análoga se llevaron a cabo las proyecciones para predicados y objetos.

Las operaciones de proyección pasaban los test y habían quedado validadas, ahora era el turno de la unión, basándonos en teoría de conjuntos, implementamos la unión de forma que el conjunto solución de la unión fuera el resultante de la fusión de las listas de hechos válidos provenientes de los dos patrones pasados por parámetros, sin embargo, alguno o los dos patrones pasado por parámetro podían ser patrones básicos del estilo [“Juan”, “tiene”, coche”], ante este tipo de circunstancias decidimos que en caso de verificarse el patrón sería añadido al conjunto solución de realizar la unión.

Para la intersección nos basamos nuevamente en la teoría de conjuntos, el conjunto resultado de una intersección serían las tuplas comunes a los patrones implicados en la intersección, una vez ambos patrones hayan sido validados contra el almacenamiento, de forma análoga a la unión, si alguno de los patrones pasado por parámetro era un patrón básico se procedería de la misma forma que en la unión sólo si cumplía la condiciones para formar parte de la intersección, eso es que sea común a la tuplas resultado de los patrones implicados.

En el diagrama que se muestra a continuación podemos ver el diseño de la clase “Operation”, donde estarían contenidas las diferentes operaciones que se suministrarían y la relación con las demás clases que necesita para desempeñar correctamente su labor, a estas altura ya se usa la interface de comunicación con el almacenamiento MatchinManager que proporciona todos los tipos de búsqueda desarrollados en al apartado anterior.

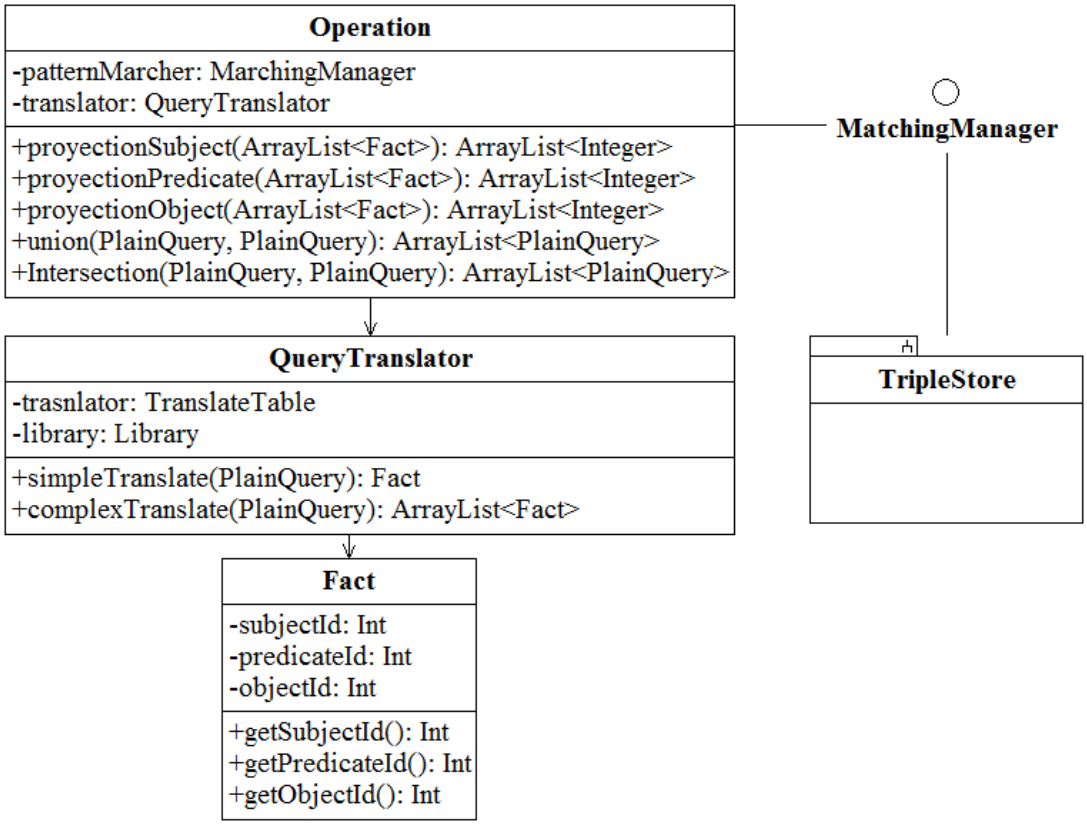


Ilustración 11

Hasta el momento el desarrollo había transcurrido sin muchos inconvenientes, sin embargo, al terminar de desarrollar y validar las operaciones comenzaros a darnos cuenta que el sistema de petición de consulta que teníamos implementado no se ajustaba a los nuevos requisitos del software, hasta ahora la lista de patrones básicos ya fueran con "What" o sin "What" había cumplido su función a la perfección, en cambio ahora nos encontramos con la dificultad de establecer la forma de solicitar una operación entre patrones, una de las ideas que se planteó fue recoger la cadena de algebra y mediante un parser crear nuestra propia estructura de operaciones, sin embargo este planteamiento dejaba aun mayor constancia de un problema subyacente, ¿es realmente responsabilidad de este componente llevar a cabo ese análisis de la consulta? Después de más de un debate, llegamos a la conclusión de que no era responsabilidad del gestor de consultas llevar a cabo dicho análisis, por ello se delegaron estas responsabilidades al componente *Interpreter* de la arquitectura Triskel, que nos suministraría un árbol de operaciones para afrontar la problemática que había surgido durante la iteración, abordaremos el tema del árbol cuarta iteración.

Además decidimos utilizar una estructura de trasiego de información más adecuada a nuestros fines, así pues, dejamos atrás la estructura de listas, facilitada por Java, la cual había quedado obsoleta para nuestros fines y se hizo necesario definir una nueva estructura de trasiego de información entre componentes, esta estructura recibiría el nombre de ConstantSet y consistiría en una estructura capaz de almacenar el tipo de datos que manejase el almacenamiento, de cara a devolver en ella los conjuntos resultado que se obtuviesen de una validación correcta al realizar una petición al almacenamiento.

10.3 Adaptación dela búsqueda de patrones a nuevas estructuras e implementación de la búsqueda bidimensional

Análisis del problema

Teniendo en cuenta la nueva propuesta al final de la segunda iteración de cara a las listas, se hacía necesario actualizar las partes del software que fueran necesarias para que el trasiego de información entre los distintos componentes se realizase mediante el tipo de datos ConstantSet.

Por otro lado nos enfrentábamos al problema de gestionar las peticiones de consulta con la nueva estructura de los patrones, además de establecer el límite entre gestión de la consulta y almacenamiento, tratando de dotar al componente de una buena modularidad

Sin embargo, una vez actualizado el software para resolver desde las consultas más básicas (“x”, “y”, “z”) a las consultas unidimensionales (“?”, “y”, “z”) había que afrontar un problema hasta ahora pospuesto, los patrones bidimensionales, que nos permitirían obtener la combinación sujeto-objeto para un predicado dado, sujeto-predicado para un objeto dado y predicado-objeto para un sujeto dado.

Recapitulando, los objetivos para esta iteración son:

- Migrar de listas a ConstantSet.
- Desarrollo del sistema de Pattern Matching para la nueva estructura del pattern.
- Implementar búsqueda bidimensional.

Diseño e implementación

Teniendo presente los inconvenientes acontecidos al final de la segunda iteración, comenzamos actualizando las distintas estructuras que usaban listas para que pasaran a usar ConstantSet, ahora el almacenamiento nos devolvía ConstantSet y debíamos gestionarlos de manera adecuada para devolver los conjuntos resultado correctamente.

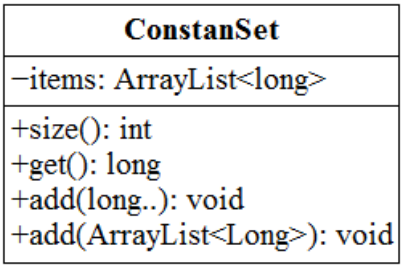


Ilustración 12

Sin embargo, para poder realizar la gestión de las consultas, el patrón tal y como lo habíamos conocido hasta ahora tuvo que sufrir un actualización para poder satisfacer las nuevas exigencias del software, con el árbol de operaciones en camino, el antiguo patrón quedaba obsoleto, por lo que pasó a ser una estructura dotada de operandos, donde cada operando podía ser o bien una constante o bien un "What", entendiendo como constante al identificador de un sujeto, predicado u objeto, según fuera el caso, y como "What" a la incógnita por la que se pregunta, además los "What" pasarían a tener un nombre asociado, de esta manera podíamos diferenciarlos unos de otros, la responsabilidad que tenía en un principio el QueryManager de traducir los elementos implicados en un patrón había sido delegada al *Interpreter* de Triskel, de esta manera el gestor de consultas no tendría que verse inmiscuido en el tipo de datos que usase el almacenamiento para organizar su información, trabajaría directamente a nivel de conjuntos de datos (ConstantSets), y no conocería al diccionario o mejor dicho al componente *TermStore*, de esta manera fomentamos la modularidad del software, definíamos mejor los componentes y sus respectivas responsabilidades.

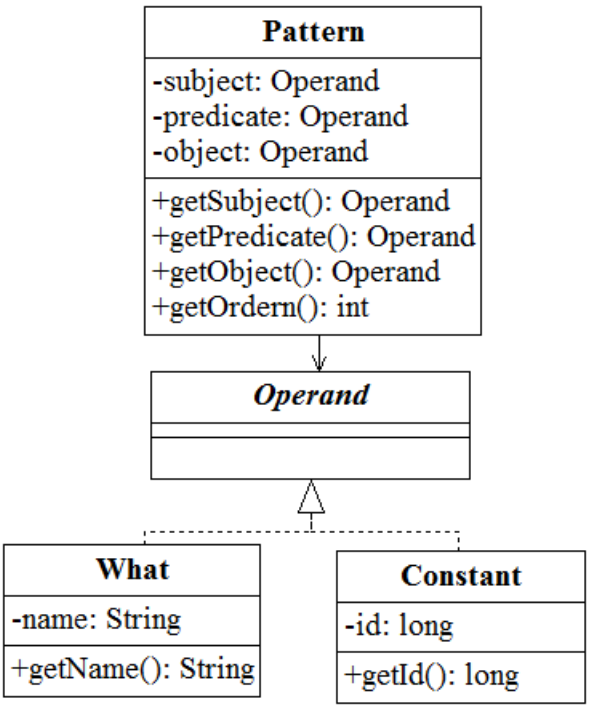


Ilustración 13

En la imagen anterior se muestra la clase Pattern y como modelamos su estructura, siendo Operand la clase abstracta de la que heredan la clase What y Constant, cabe decir, que la clase Pattern es común a varios componentes de la arquitectura Triskel (*Interpreter, QueryPlayer y TripleStore*), de esta manera somos capaces de declarar patrones de cualquier tipo, ya sea con o sin incógnitas, además de obtener su orden, número de incógnitas del patrón.

Con los elementos circundantes a las consultas resueltos, nos dimos cuenta de que debía crearse una canal de comunicación más adecuado con el almacenamiento que la de acceder directamente al a sus métodos de búsqueda, por ello nació el concepto de PatternQuery, el PatternQuery vendría a ser como denominaríamos formalmente a la consulta de un patrón en concreto, y poniendo en ejecución dicha estructura se desencadenaría la evaluación del patrón, sin embargo, para evitar que el PatternQuery conociese detalles de implementación del almacenamiento se delegó la responsabilidad de realizar las búsquedas internas al almacenamiento en sí, de esta manera toda implementación de almacenamiento que se llevase a cabo debería implementar una serie de métodos abstractos perteneciente a esta clase, por el momento sólo se trataría de las búsqueda unidimensionales, de esta manera el PatternQuery se encargaba de descomponer el patrón, identificar cual es el motivo de la consulta, ya fuese solicitar sujetos, predicado u objetos y de registrar en un hashMap el nombre del "What" implicado en la consulta y del ConstantSet asociado a la respuesta dada por el almacenamiento al patrón facilitado en la consulta, con esto seguíamos conservando una buena modularidad del software y ocultábamos detalles de implementación a otros componentes.

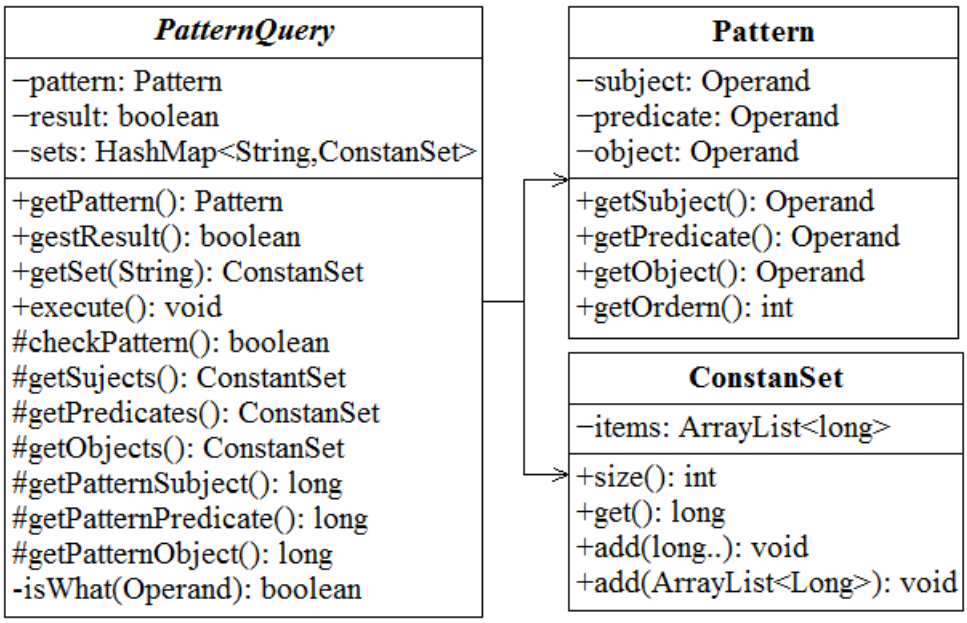


Ilustración 14

La clase PatternQuery nos suministra los elementos necesarios para que al realizar un execute de la clase se desencadene el proceso adecuado para solicitar la información que se está pidiendo, ya sea comprobar un patrón concreto (checkPattern) o solicitar un conjunto de valores pertenecientes al conjunto de los sujetos, predicados u objetos, una vez finalizada la consulta se podrá obtener del hashMap el ConstantSet a la variable asociada a la consulta.

Cuando llegó el momento de enfrentarnos a la búsqueda bidimensional coincidimos en que sería un patrón con dos "What" entre sus elementos, no obstante, cualquier combinación con una incógnita en el predicado no es muy común en SPARQL, pues estaríamos preguntando por todas las combinaciones predicado-objeto para una consulta del estilo ["x", "What Predicate", "What Object"] o por todas la combinaciones sujeto-predicado para un determinado objeto como por ejemplo ["What Subject", "What Predicate", "z"] por ello en una primera aproximación decidimos desarrollar la búsqueda bidimensional para sujeto y objeto como por ejemplo ["What Subject", y ," What Object"], lo que nos llevaría a al conjuntos de soluciones de todos los sujetos y objetos relacionados mediante el predicado dado.

Sin embargo, conforme íbamos debatiendo sobre que búsqueda bidimensional se iba a implementar, nos surgió un problema a la hora de devolver el conjunto resultado, el ConstantSet como hasta ahora lo conocemos no soportaba el almacenamiento de una tupla de datos, para el caso que implementamos sería la tupla {sujeto, objeto}, por ello tuvimos que ir pensando en otra forma de almacenar el resultado de estas consultas, con lo que ello conllevaba, una actualización del componente de trasiego de información entre los componentes, que hasta ahora había sido el ConstanSet.

El nuevo elemento pasaría a ser una tabla, donde cada una de sus columnas se correspondería con el nombre de cada una de las incógnitas implicadas en la consulta y cada una de sus filas sería un identificador de sujeto, predicado u objeto según fuera la naturaleza de la consulta, por ejemplo para el patrón [“What”, “tiene”, “coche”], obtendríamos una tabla de una columna identificada con el nombre que se le asignase a la incógnita y tantas filas como sujetos registrados con coche.

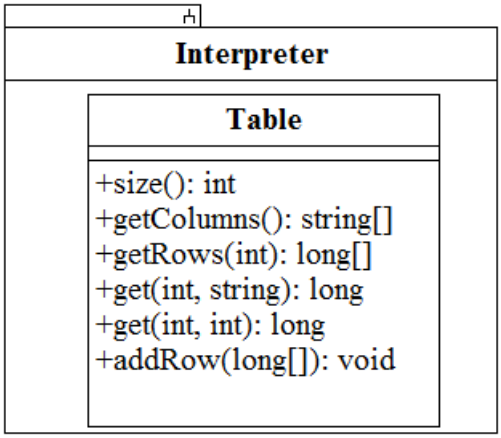


Ilustración 15

Una vez más nos hemos permitido el lujo de utilizar el subsistema UML para dejar constancia que la tabla tomo forma y vida en el componente Interpreter de la arquitectura Triskel, la clase Table nos suministra una serie de operaciones para poder rellenar la tabla con los identificadores válidos del almacenamiento y la capacidad de acceder a estos elementos para aprovechar la dependencia entre patrones de cara a agilizar la respuesta de las consultas, sin embargo, esto lo veremos durante la cuarta iteración.

Mientras la tabla iba siendo desarrollada procedimos a actualizar las zonas del software pertinentes para la migración de ConstantSet a Table, además de actualizar también el PatternQuery para identificar patrones bidimensionales, como consecuencia tuvimos que solicitar que los almacenamientos que se desarrollasen, aparte de implementar las búsqueda unidimensionales, deberían también implementar la búsqueda bidimensional, así de manera análoga a los patrones unidimensionales ocultábamos detalles de implementación para patrones bidimensionales, una vez identificado que se trata de un patrón bidimensional se crearía la tabla pertinente y se solicitaría la búsqueda al almacenamiento, facilitándole la tabla para que añada los resultados para su posterior devolución a quien nos hubiese invocado.

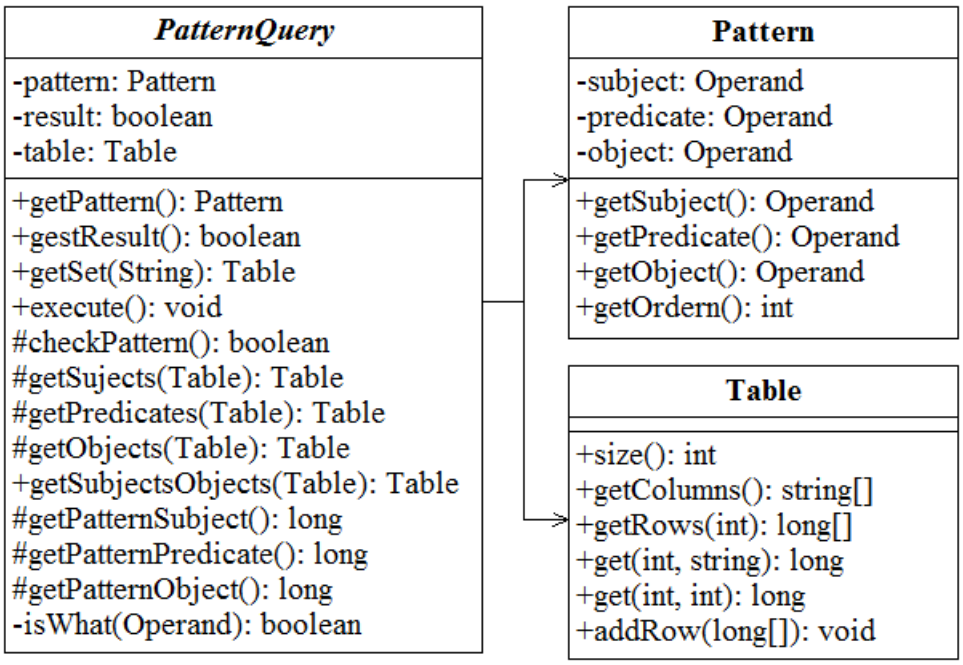


Ilustración 16

10.4 Implementación y comparativa entre diferentes ejecuciones de consultas

Análisis del problema

El problema que nos habíamos encontrado en la segunda iteración de cara a modelar la peticiones que contuviesen operaciones en una lista había quedado resuelto mediante un árbol de operaciones, ahora era el momento de tratar dicho árbol, para ello llevamos a cabo diferentes implementaciones del sistema de ejecución de patrones, conocido como QueryPlayer, dichas implementaciones fueron:

Secuencial

Se ejecutarán secuencialmente todos los patrones y operaciones contenidas en el árbol de operaciones facilitado.

Con prioridad

Se irán ejecutando los patrones desde los más básicos, como por ejemplo una búsqueda unidimensional, a los más complejos, como podrían ser búsquedas bidimensionales y finalmente las operaciones.

Con análisis de dependencias en el mismo nivel

Se tratará de explotar la existencia de dependencia entre patrones que se encuentren en el mismo nivel del árbol de operaciones, tratando de agilizar las consultas.

Con análisis de dependencias en diferentes niveles

Se tratará de explotar la dependencia entre patrones no sólo del mismo nivel, sino en niveles inferiores del árbol de operaciones para mejorar el rendimiento de cara a las consultas.

Además para cada uno de los sistemas de ejecución de patrones se valorará el rendimiento individual de cada uno de ellos ante diferentes árboles de operaciones, comparando las métricas obtenidas para cada una de las diferentes versiones del componente.

Recapitulando, objetivos para esta iteración corresponden con:

- Implementar un player secuencial.
- Implementar un player con prioridad.
- Implementar un player con detección de dependencias en el mismo nivel.
- Implementar un player con detencción de dependencias entre niveles.
- Realizar pruebas de rendimiento.

Diseño e implementación

Durante esta iteración planteamos un desarrollo incremental, como a priori no podíamos estimar que forma de resolver los patrones sería la más acertada, se decidió implementar las que fueron surgiendo, para después enfrentarlas en una comparativa de rendimiento, de esta manera se desarrollaron varios QueryPlayer.

Para poder realizar su cometido el QueryPlayer necesita que le faciliten tanto un árbol de operaciones como un almacenamiento contra el que resolver los patrones, de hecho será la interfaz de comunicación con el almacenamiento, los árboles de operación únicamente tendrán tres tipos de nodos que debemos tener en cuenta:

FunctionNode: Son aquellos nodos que contienen las operaciones, para el QueryPlayer es totalmente indiferente que operación contiene, con la salvedad que llegado el momento se procederá a ejecutar el FunctionNode y la operación que encierra quedará resuelta.

PatternNode: Son aquellos nodos que contienen los patrones, una vez encontrado un nodo de estas características se procederá a su evaluación contra el almacenamiento facilitado.

TableNode: Son aquellos nodos que contienen la solución de los patrones, para cada PatternNode habrá un TableNode con el resultado obtenido al lanzar el patrón contra el almacenamiento.

Los QueryPlayer irán recorriendo el árbol de operaciones en busca de los PatternNode, una vez los vayan encontrando resolverán cada patrón contra el almacenamiento, la tabla resultante será encapsulada en una TableNode y el PatternNode que dio lugar a ese TableNode será cambiado por dicho TableNode, finalmente irán ejecutando las operaciones que se vayan encontrando y se devolverá una tabla final con la solución a la consulta.

Este sería el aspecto de un árbol de operaciones que contiene un Join entre dos patrones:

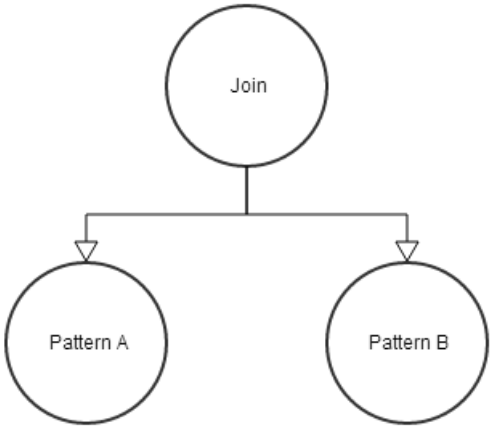


Ilustración 17

Sin embargo como hemos explicado antes los árboles son vistos de manera diferente, haciendo totalmente transparente el tipo de la operación al QueryPlayer.

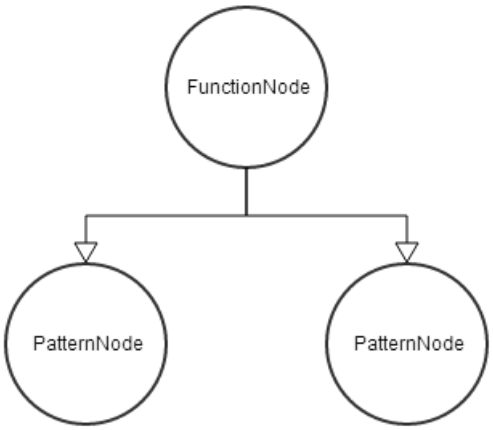


Ilustración 18

Después de que el árbol sea recorrido por el QueryPlayer resolviendo los patternNode y siendo cambiados por sus respectivos TableNode justo antes de mandar a ejecutar la operación tendrá este aspecto:

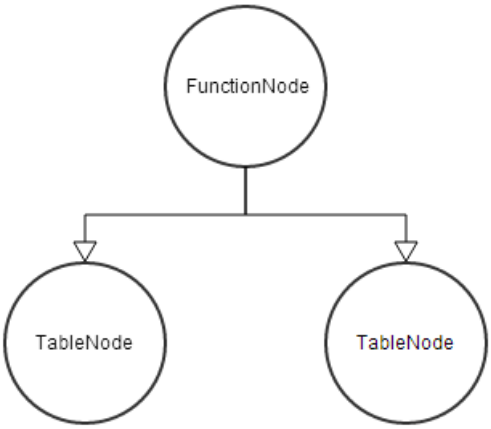


Ilustración 19

Esta es la forma en la que quedaría el árbol de operaciones antes de que el functionNode que contiene la operación sea ejecutado, una vez se haya resuelto el functionNode será devuelta la tabla con el resultado final, el Join entre las dos tablas.

Implementar un player secuencial

En una primera aproximación decidimos hacer un QueryPlayer que fuera capaz de recorrer la estructura del árbol de operaciones ejecutando los nodos como se ha comentado anteriormente, por ello el desarrollo se centró en realizar un recorrido en orden (in order) resolviendo cada uno de los nodos implicados en la consulta.

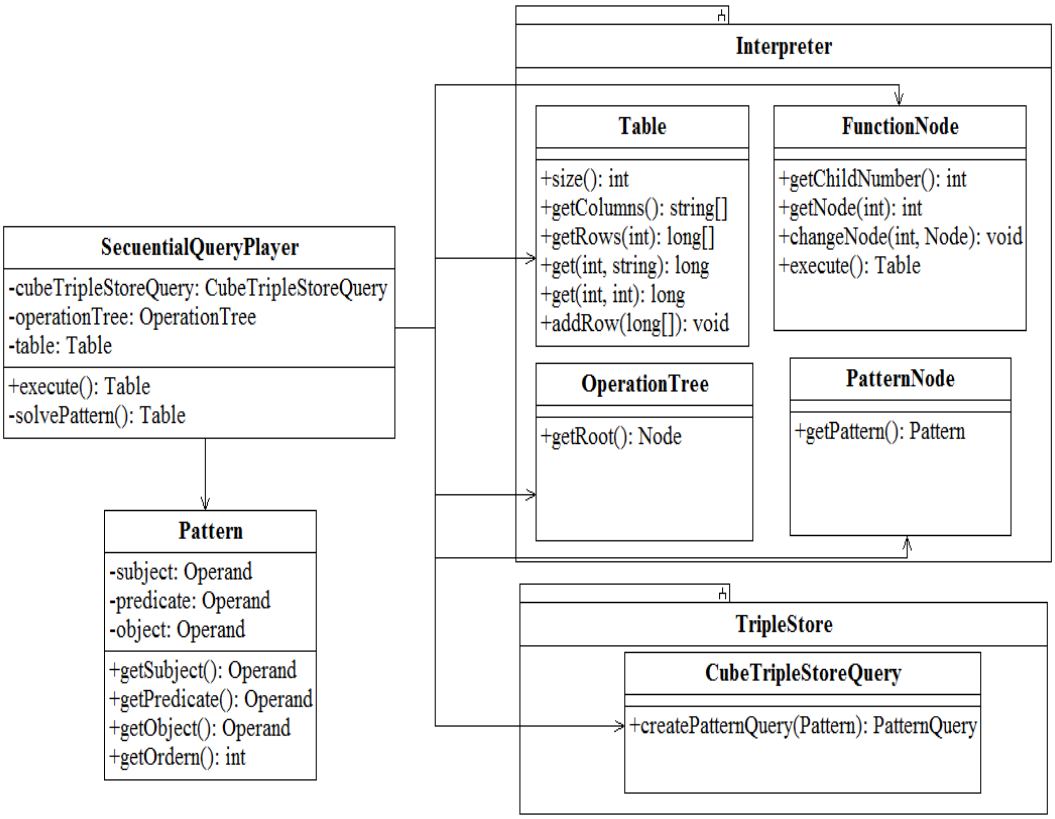


Ilustración 20

En la ilustración superior se muestran las dependencia del QueryPlayer con otros componentes de Triskel con los que se relaciona directamente, para su correcto funcionamiento necesita una almacenamiento al que solicitar las Query, CubeTripleStoreQuery cumple esa función proporcionándonos un PatternQuery que ejecutar para resolver un patrón, además hemos de conocer el árbol de operaciones para la consulta que se va a realizar, dicho árbol de operaciones nos proporciona los Pattern a resolver y finalmente necesitaremos la tabla donde almacenar el resultado de la consulta.

Tomaríamos los test desarrollados durante esta aproximación para validar los otros QueryPlayer que se desarrollasen, además se añadirían los específicos para cada uno de los siguiente QueryPlayer.

Implementar un player con prioridad

Ya habíamos rebasado la barrera de recorrer el árbol de operaciones, resolver los patrones y devolver una tabla con los resultados, ahora nos planteábamos si resolver los patrones de menor grado, y entendemos grado como el número de incógnitas que presenta un patrón, aportaría alguna mejora, por ello desarrollamos esta versión del QueryPlayer, donde los hijos de los FunctionNode son ordenados de menor a mayor grado, siendo los patrones de menor grado los unidimensionales, después los bidimensionales y finalmente otros FunctionNode.

Con el criterio de ejecución establecido nos preguntábamos ahora si andar reorganizando los hijos de los FunctionNode tendría alguna incidencia directa sobre el resultado final, tras debatirlo, teniendo en cuenta como se creaban los árboles de operaciones pudimos concluir que no afectaban al resultado final.

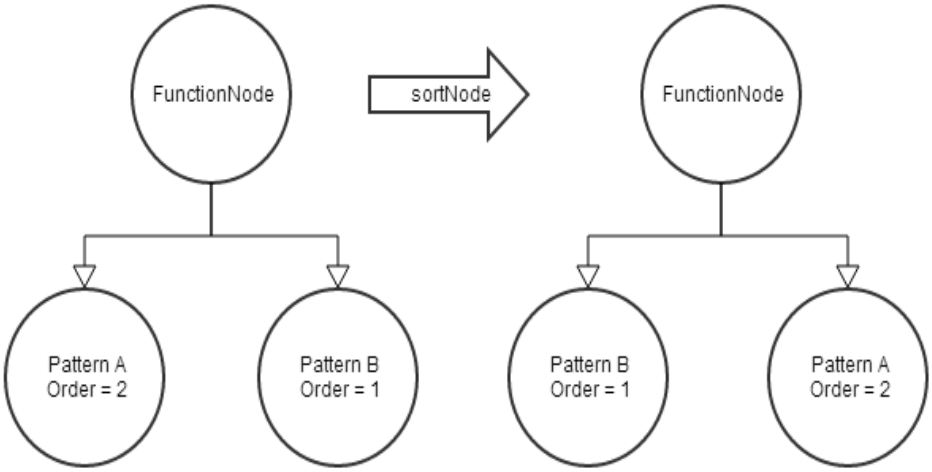


Ilustración 21

Como se puede apreciar en la ilustración al hacer uso de la función de ordenar los nodos (sortNode) los hijos del nodo en tratamiento quedarían ordenados según su orden, para una vez resueltos los patrones por medio de la ejecución del functionNode se obtuviese la tabla final, cierto es que para pasar los test se necesitaba un almacenamiento, por ello, se creó un almacenamiento pequeño para comprobar que los test se ejecutaban correctamente, además también se crearon una serie de clases que nos permitían obtener árboles de operación directamente, esto nos ayudó a tener test más limpios y legibles.

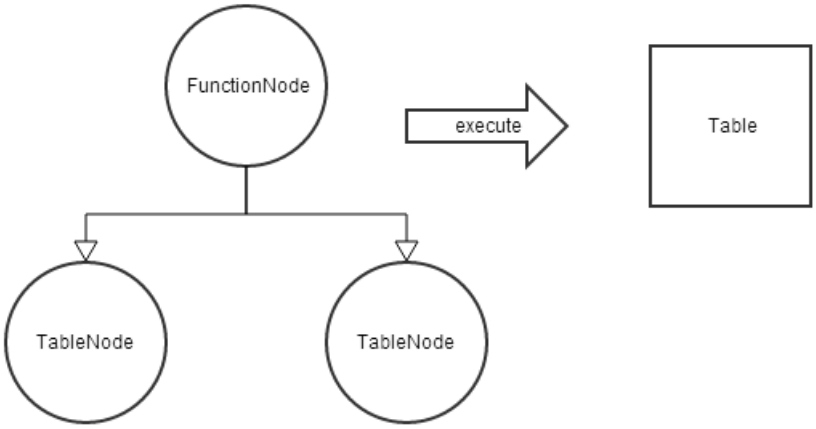


Ilustración 22

En un refactor posterior nos dimos cuenta que el QueryPlayer con prioridad estaba teniendo también la responsabilidad de ordenar los nodos, así que decidimos sacar esa característica como una clase aparte que se encargara de realizar dicha ordenación.

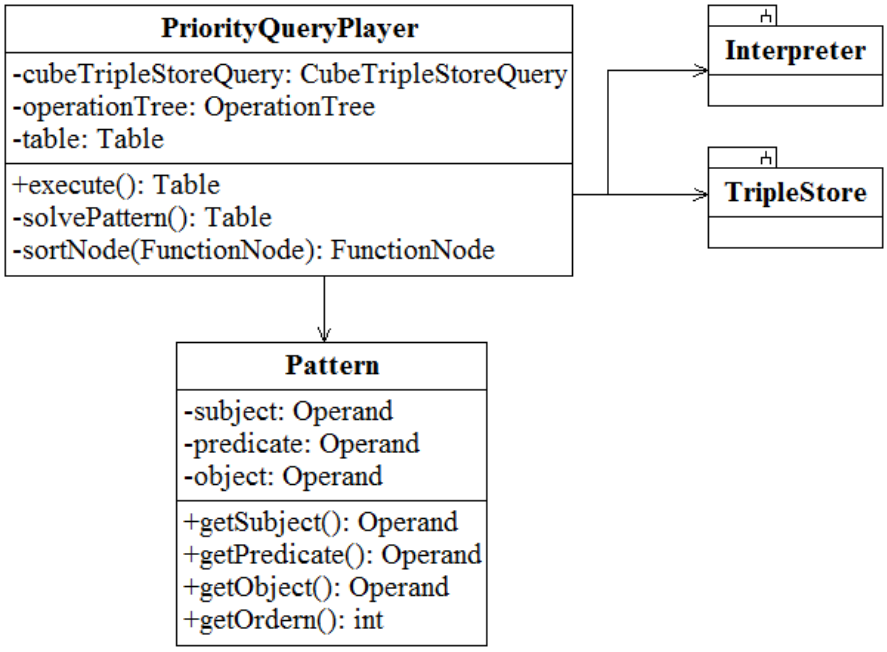


Ilustración 23

El QueryPlayer con prioridad seguía dependiendo de las mismas clases que su antecesor dependía, además incorporaba los mecanismos necesarios para ordenar cada uno de los hijos de los functionNode que pudiese encontrar durante el recorrido del árbol de operaciones.

Implementar un player con detección de dependencias en el mismo nivel.

Llegados a este punto la ejecución de patrones por prioridad había llegado a su fin, sin embargo, ahora nos planteábamos una nueva actualización del software, el análisis de dependencia entre patrones, para ello entendemos que existe una dependencia entre patrones cuando el nombre de una incógnita en una patrón unidimensional es el mismo que una de las incógnitas de una patrón bidimensional por ejemplo [“What:**N**ame”, “Has”, “Passport”] y [“What:**N**ame”, “Is-citizen”, “What:Country”] tendrían una dependencia en el sujeto.

Sin embargo, los sujetos podían ocupar el lugar de los objetos y viceversa, no planteamos que pasaría con este tipo de dependencias, no obstante, decidimos no resolverlas, pues para poderles hacer frente se hacía necesario un proceso de traducción intermedio, de identificador de sujeto a identificador de objeto y debido a los compromisos que habíamos llegado anteriormente quedaba fuera de las responsabilidades del QueryPlayer realizar procesos de traducción.

Decidimos que la forma de afrontar las dependencias, sería reducir la consulta bidimensional a la unión de tantas unidimensionales como tamaño tuviera el conjunto de soluciones de la consulta unidimensional con la que tiene la dependencia.

La forma de proceder el QueryPlayer sería la siguiente, inicialmente se organizarían los hijos ,del FunctionNode en tratamiento, de menor a mayor grado, una vez hecho esto se desencadenaría el proceso de análisis de dependencias, para ello se irían resolviendo los patrones de grado uno (unidimensionales) y para cada uno de ellos se comprobaría si presentas dependencia con alguno de los patrones de grado dos de ese nivel, en caso de no haber ningún patrón de grado dos en el nivel en tratamiento se revolvería como si se tratase de una ejecución por prioridad.

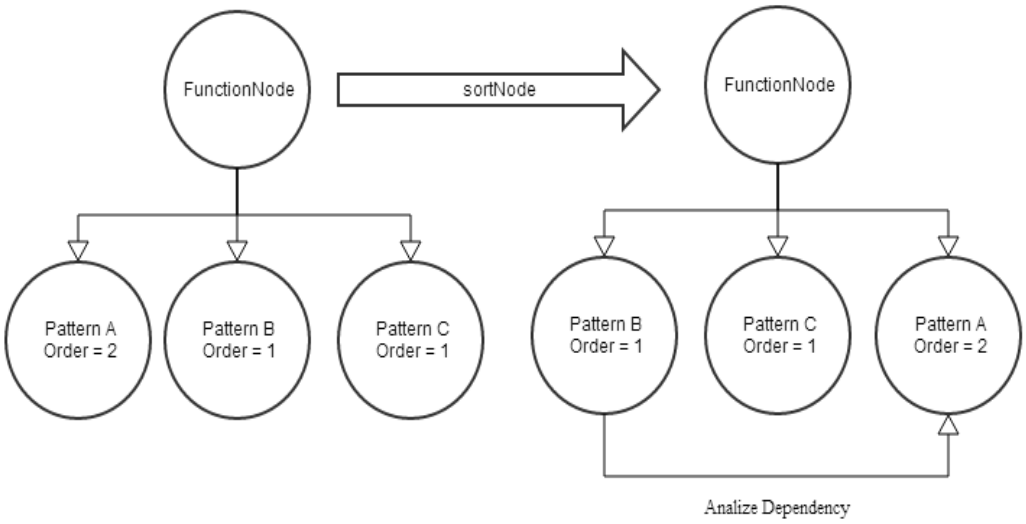


Ilustración 24

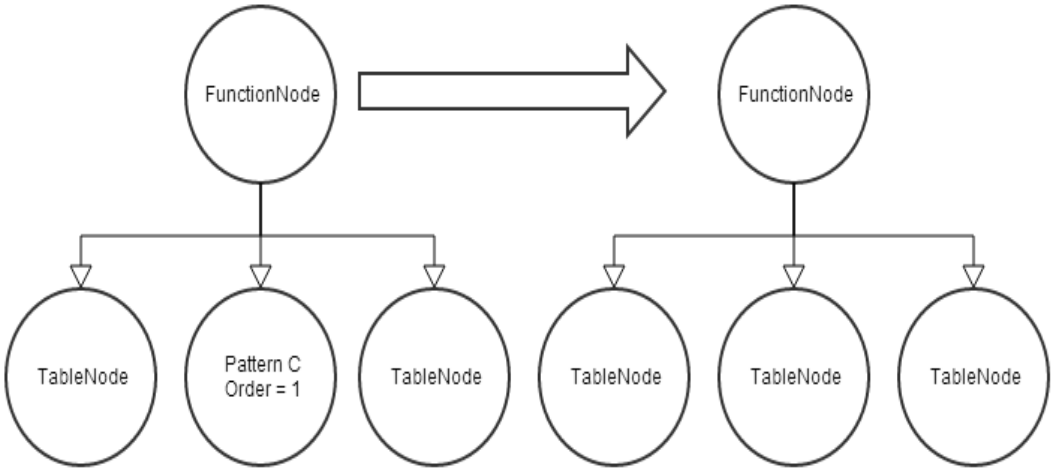


Ilustración 25

Sólo nos quedaba contemplar el caso de una ejecución donde hubiesen patrones de orden uno y orden dos y no presentasen una dependencia, al entrar en el analizador los patrones de grado uno quedarían resueltos, sin embargo los de grado dos no, por ello una vez salimos del analizador de dependencia se repasa el nivel antes de pasar al siguiente para resolver los patrones que quedasen pendientes u otros FunctionNodes que interviniesen en la consulta, los TableNode que pudiese existir eran pasado por alto.

Sin embargo nos asaltó la duda de si al hacer esto estábamos sesgando el conjunto resultado, tras debatirlo nuevamente y como pasó en el QueryPlayer con prioridad la forma de crearse los árboles de operaciones nos blindaba contra el problema que se nos estaba planteando.

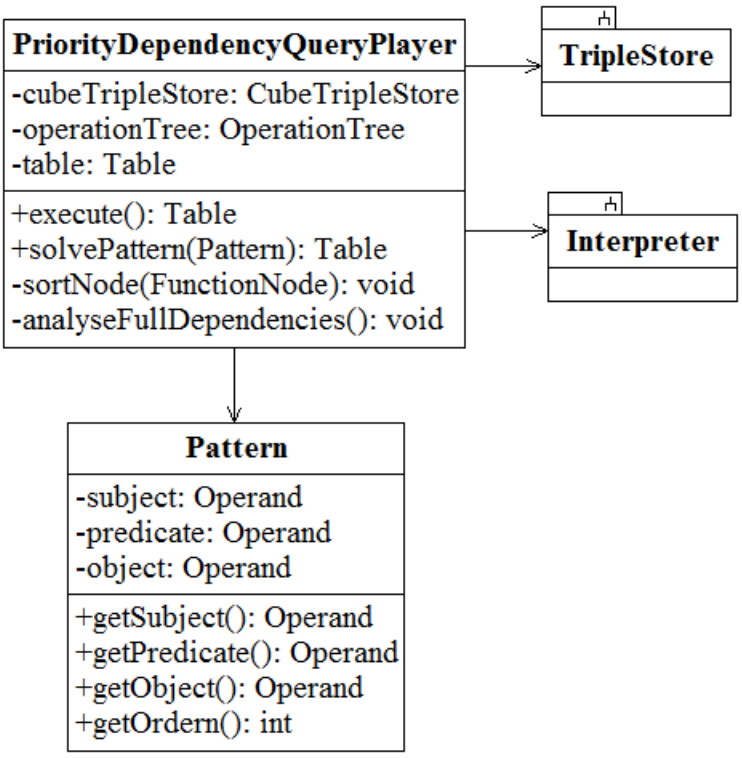


Ilustración 26

No obstante, el QueryPlayer que analizaba las dependencias en un solo nivel había alcanzado un tamaño considerable, por lo que se procedió a una labor de refactor para definir mejor su estructura.

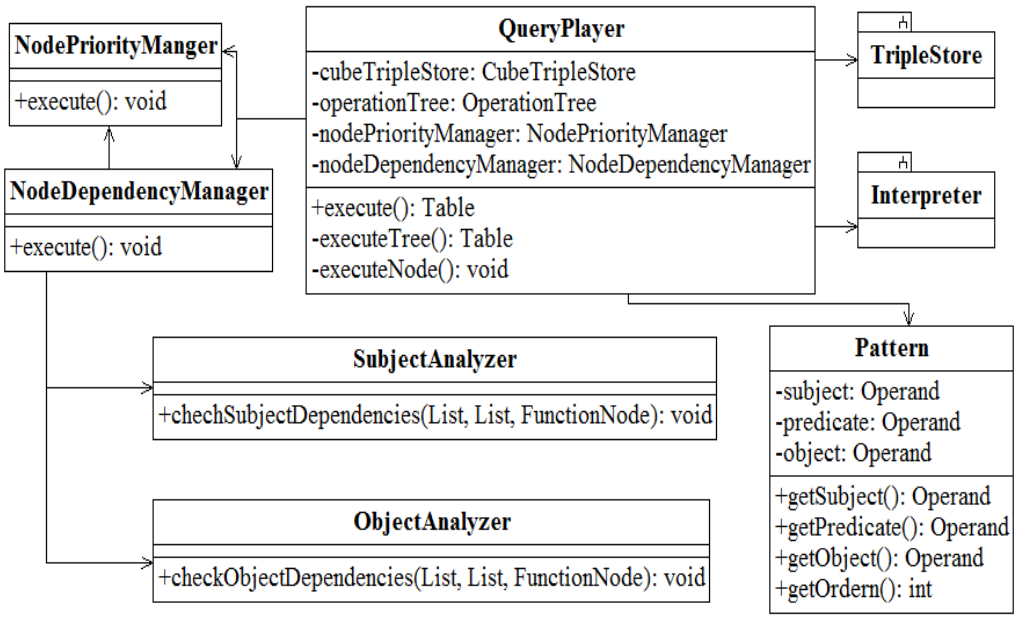


Ilustración 27

Nacieron las clases de gestión de prioridad y dependencia, así como los analizadores de dependencias tanto para sujeto como para predicado, cabe destacar que teniendo en cuenta la decisión tomada anteriormente, las dependencia sujeto-objeto y viceversa no serían tratadas debido a que implicarían una solicitud de traducción al *TermStore*.

Implementar un player con detección de dependencias entre niveles.

Habíamos conseguido recorrer el árbol de operaciones con éxito, ordenar sus elementos por prioridad y analizar la existencia de dependencias en un nivel, no obstante empezamos a plantearnos el caso de la dependencia entre patrones entre distintos niveles.

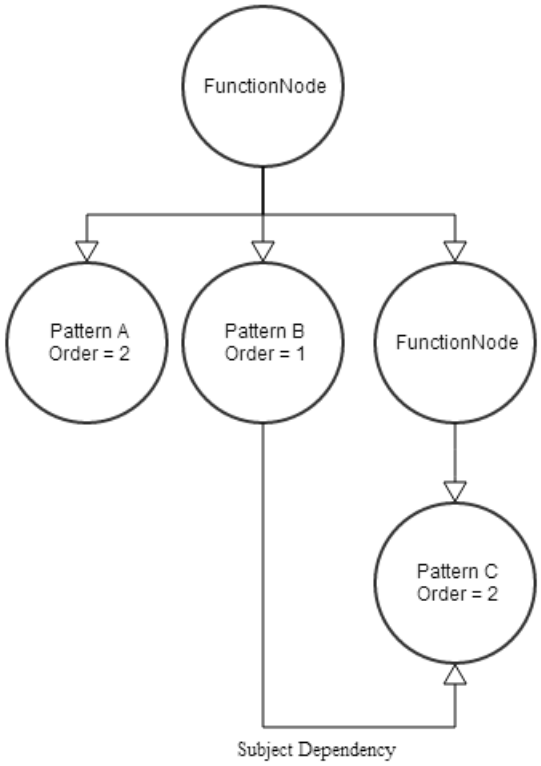
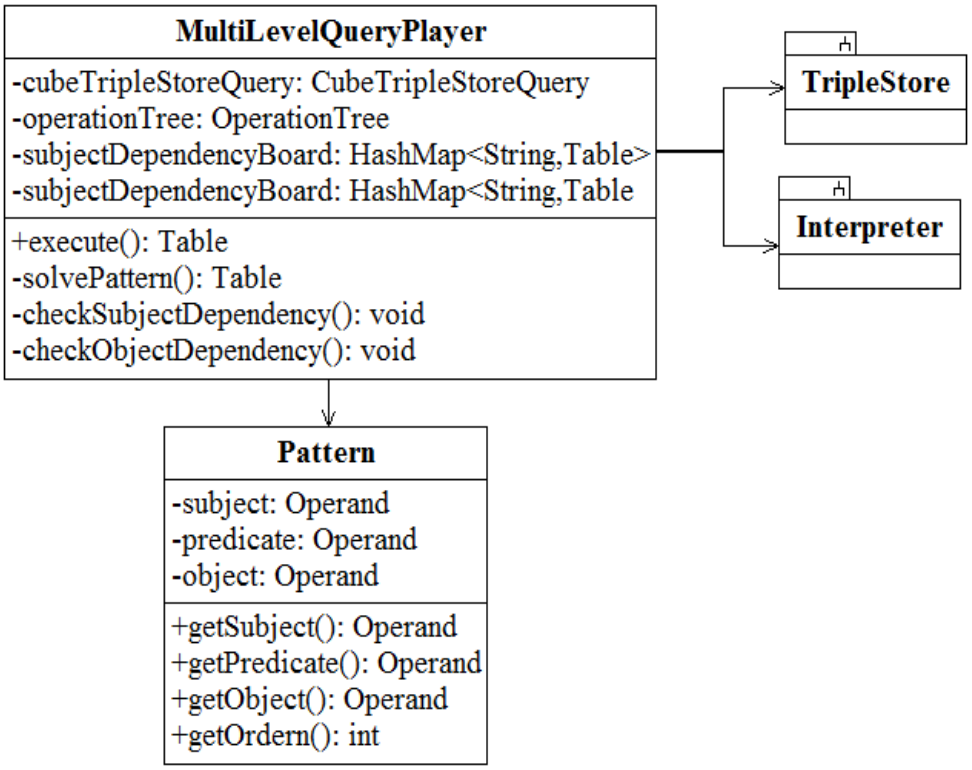


Ilustración 28

El planteamiento que teníamos anteriormente no era válido pues tendríamos que pasar de nivel a nivel información entra, sin embargo, se nos ocurrió que los patrones bidimensionales podían preguntar si había sido resuelto algún patrón unidimensional para el que alguna de sus incógnitas coincidiera y si era así obtener el conjunto resultado asociado al patrón unidimensional resuelto para simplificarse a la unión de una serie de patrones unidimensionales, tal y como explicamos en el apartado anterior.

La manera de llevar a cabo esta implementación fue por medio de hashMaps, uno para las incógnitas de tipo sujeto y otras para las de tipo objeto, pues como expusimos en el apartado anterior aunque un sujeto puede ser objeto y viceversa los identificadores pueden no ser los mismos.

Entonces, la manera de proceder del QueryPlayer sería la siguiente, vamos recorriendo el árbol de operaciones ordenando cada nivel, de patrón de menor grado a patrón de mayor grado, una vez hecho esto se procede a ir ejecutando los patrones, en caso de ser de grado uno es ejecutado normalmente contra el almacenamiento y antes de cambiarlo por su respectivo TableNode, es registrado en el hashMap de sujeto o de objetos según proceda, en caso de tratarse de un patrón de grado dos (Bidimensional) identificaría cuales son los nombres de sus incógnitas e iría primero al hashMap de sujetos a ver si puede simplificarse, si es así queda simplificado como se expuso en el apartado anterior y se pasa al siguiente patrón, en caso de no ser así se comprueban las dependencias de objeto y se procede de forma análoga a la dependencia con el sujeto, sin embargo, puede ocurrir no se encuentre ninguna dependencia, esto nos llevaría a resolver el patrón sin ser simplificado y a colocar su correspondiente TableNode donde proceda.



Realizar pruebas de rendimiento

Habíamos conseguido desarrollar cuatro QueryPlayers, por lo que había llegado la hora de las pruebas de rendimiento, para ello decidimos hacer dos tipos de pruebas, las pruebas en el mejor de los casos, una consulta simple preguntando por una o dos variables si implicar operaciones, y por otro lado el peor de los casos, una consulta donde se tuviese que ejecutar una operación pesada, en este caso la operación que escogimos fue el Join.

Sin embargo los tiempos de ejecución de la misma prueba de rendimiento podían dar valores diferentes, esto puede ser debido a:

- La máquina virtual de java puede hacer diferentes optimizaciones entre las distintas invocaciones.
- Estamos midiendo tiempo transcurrido, así que puede existir la posibilidad que otros procesos aparte de Java se estén ejecutando en el equipo.
- El procesador y la memoria RAM es posible que estén en "caliente" en las siguientes invocaciones.

Teniendo esto en cuenta decidimos que teníamos que hacer múltiples invocaciones para hacernos una idea los más precisa posible del tiemplo de ejecución de un método, pero a qué número asciende ese "múltiples", investigando un poco parece que debe ser del orden de miles, por ello preparamos los test para que se realizaran diez mil invocaciones del método del que queríamos hacernos una idea del tiempo que transcurría en ejecutarse, además cada prueba de rendimiento generaría un fichero de log donde quedaría reflejado el tiempo de ejecución del método para búsquedas que entrañasen distintos conjuntos de resultados, decidimos que se ejecutarían los métodos para búsquedas de mil, diez mil, cien mil y un millón de elementos, de esta manera compraríamos el rendimiento de cada uno de los QueryPlayer ante esta situaciones, y podríamos en base a esta métrica tomar una mejor decisión sobre cual satisface más nuestras necesidades, a continuación se exponen cada uno de las pruebas de rendimiento realizadas y los resultados que obtuvimos de ellas.

Consulta Unidimensional

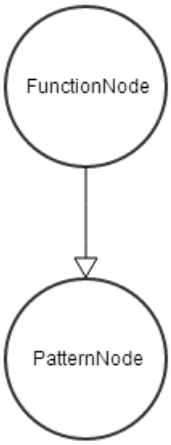


Ilustración 29

Antes de pasar a ver los resultados obtenidos, cabe destacar un rendimiento similar para las búsqueda unidimensionales con una incógnita en el objeto, por lo que no se han incluido en este apartado, además debido a que el número de predicados siempre será mucho menor que el número de sujetos y objetos tampoco se han incluido las pruebas de rendimiento de predicado en este apartado.

Incógnita en el sujeto.

Supongamos un árbol de operaciones cuyo functionNode es un Join, sin embargo, tiene un único hijo y además se trata de un patrón unidimensional, al resolver los árboles para los diferentes QueryPlayer implementados se obtuvieron los siguientes resultados para diferentes conjuntos de datos devueltos.

Tabla devuelta de mil elementos

Diez mil ejecuciones				
Milisegundos	Sequential	Priority	Dependency	MultiLevel
25	9999	10000	10000	10000
50	1	0	0	0
100	0	0	0	0
250	0	0	0	0
500	0	0	0	0
750	0	0	0	0
1000	0	0	0	0
1500	0	0	0	0
2000	0	0	0	0

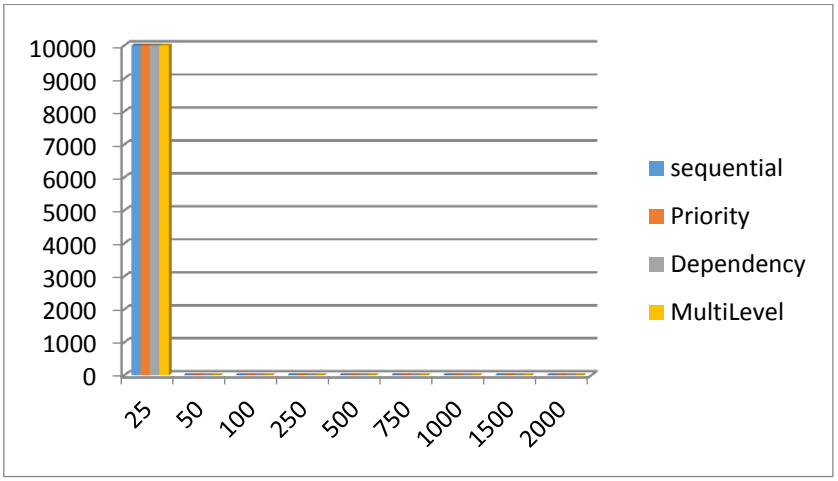


Ilustración 30

En la tabla superior podemos ver como se distribuyen las ejecuciones a lo largo del tiempo, de todas las ejecuciones, a excepción de una, concretamente la primera de todas, las consultas para el árbol anteriormente mencionado se concentran en torno a los 25 milisegundos para un conjunto resultado de mil elementos.

Tabla devuelta de diez mil elementos

Diez mil ejecuciones				
Milisegundos	Sequential	Priority	Dependency	MultiLevel
25	10000	10000	10000	10000
50	0	0	0	0
100	0	0	0	0
250	0	0	0	0
500	0	0	0	0
750	0	0	0	0
1000	0	0	0	0
1500	0	0	0	0
2000	0	0	0	0

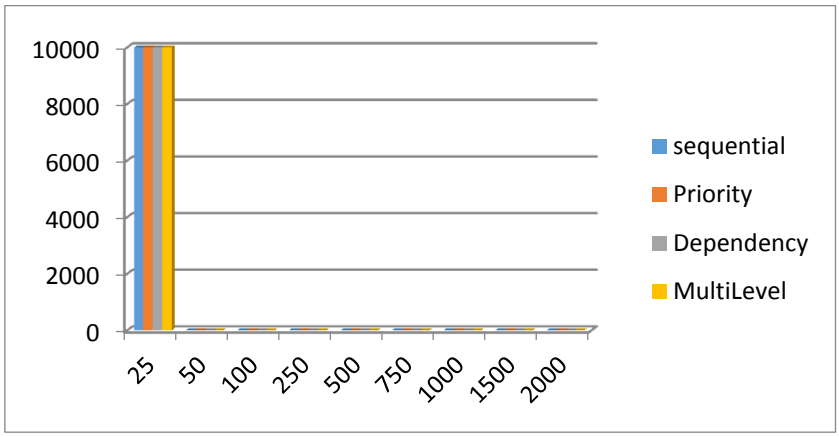


Ilustración 31

De forma análoga a la consulta anterior, una consulta para obtener un conjunto de datos de diez mil elementos, todas las consultas tardaron del orden de 25 milisegundos.

Tabla devuelta de cien mil elementos

Diez mil ejecuciones				
Milisegundos	Sequential	Priority	Dependency	MultiLevel
25	10000	10000	9988	9895
50	0	0	11	104
100	0	0	0	0
250	0	0	1	1
500	0	0	0	0
750	0	0	0	0
1000	0	0	0	0
1500	0	0	0	0
2000	0	0	0	0

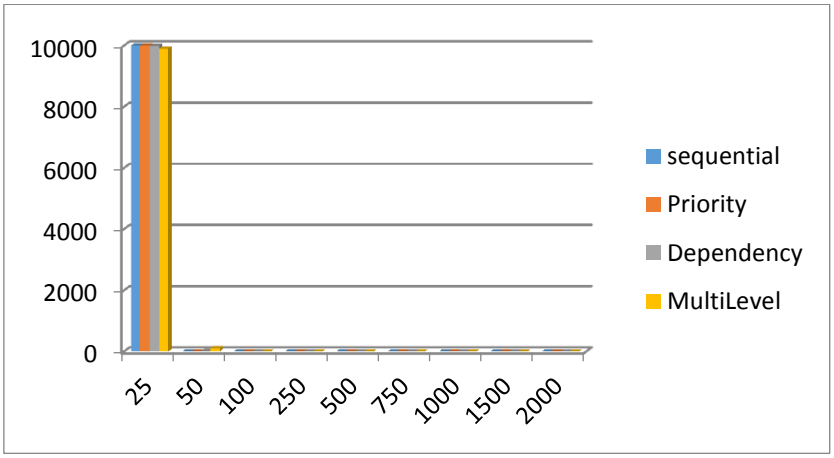


Ilustración 32

Para cien mil elementos comenzamos a ver los primeros cambios en cuestión de rendimiento, no obstante todos los QueryPlayer implementados seguían tardando del orden de 25 milisegundos en realizar una consulta con un conjunto devuelto de cien mil elementos.

Tabla devuelta de un millón de elementos

Diez mil ejecuciones				
Milisegundos	Sequential	Priority	Dependency	MultiLevel
25	360	2	132	411
50	4329	41	1433	3979
100	5173	2258	79	276
250	121	7488	6614	4918
500	17	0	0	0
750	0	107	0	0
1000	0	102	8	0
1500	0	1	1734	95
2000	0	0	0	321

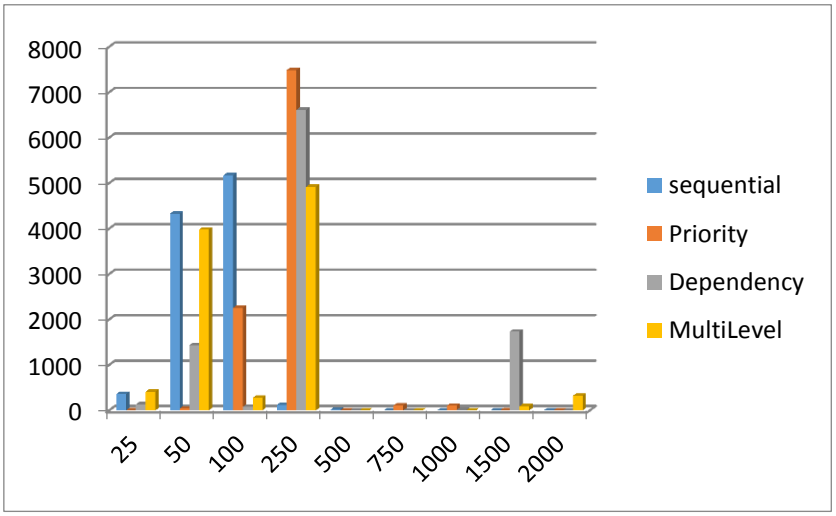


Ilustración 33

Para devolver un conjunto de datos de un millón de elementos nos encontramos las consultas para todos los QueryPlayer se encontraban en su mayoría entre los 50 y 250 milisegundos, posiblemente debido a crear una tabla en memoria de un millón de elementos y según fuera la disponibilidad de la CPU en esos momentos.

Consulta bidimenisonal

Supongamos un árbol de operaciones cuyo functionNode es un Join, sin embargo, tiene un único hijo y además se trata de un patrón bidimensional, con incógnita en el sujeto y objeto, al resolver los árboles para los diferentes QueryPlayer implementados se obtuvieron resultados muy parecidos a los obtenidos en la búsqueda unidimensional.

Tabla devuelta de un millón de elementos

Diez mil ejecuciones				
Milisegundos	Sequential	Priority	Dependency	MultiLevel
25	360	2	132	411
50	4329	41	1433	3979
100	5173	2258	79	276
250	121	7488	6614	4918
500	17	0	0	0
750	0	107	0	0
1000	0	102	8	0
1500	0	1	1734	95
2000	0	0	0	321

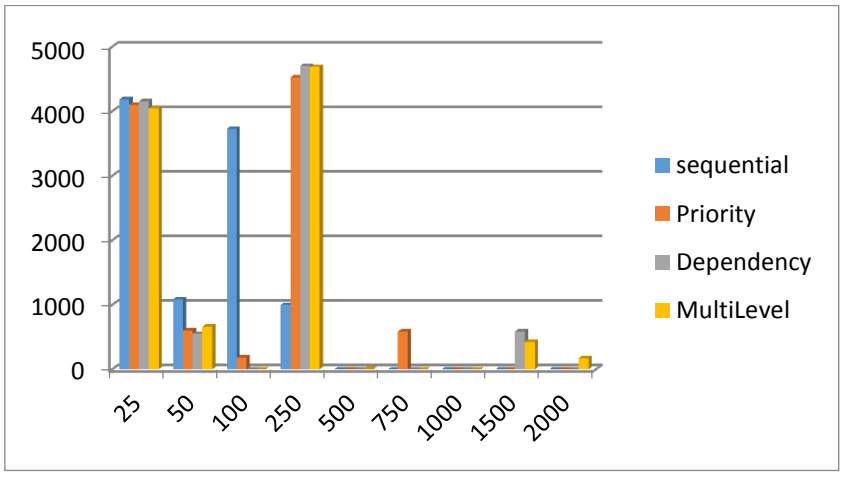


Ilustración 34

Al igual que en las consultas unidimensionales, no encontramos que para la devolución de un conjunto de datos de un millón de elementos, los tiempo volvían a estar entre 50 y 250 milisegundos.

Consulta con operación Join

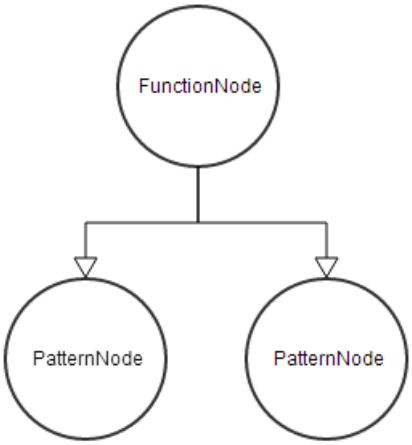


Ilustración 35

Supongamos que nos facilitan un árbol de operaciones cuyo FunctionNode es un Join, y que tiene dos hijos, los cuales son patrones unidimensionales, al ejecutarlos con los diferentes QueryPlayers se obtuvieron los siguientes resultados:

Para la consulta de dos tablas de mil elementos el tiempo de ejecución se mantuvo sobre los 25 milisegundos, sin embargo cuando comenzamos a aumentar el número de elementos en las tablas obtenidas de los patrones los tiempos de ejecución pasaron de 25 milisegundos a 131 para la consulta de tablas de diez mil elementos a 24765 para consultas de cien mil elementos, casi 25 segundos, y del orden de $5x10^6$ milisegundos para el millón de elementos, cierto es que estamos en el peor de los casos, pero el coste de la operación es muy elevado, debido a esto no se realizaron las diez miel ejecuciones de las consultas que superaban los 25 segundos y pasamos a plantear el experimento que viene a continuación.

Consulta con operación Join y dependencia.

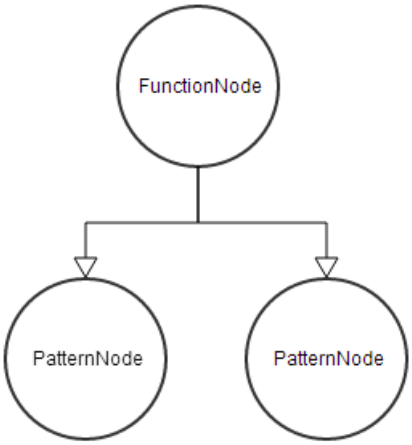


Ilustración 36

Supongamos que nos facilitan un árbol de operaciones cuyo FunctionNode es un Join, y que tiene dos hijos, uno de ellos es un patrón unidimensional y otro es un patrón bidimensional, además presentan una dependencia en el sujeto, finalmente son ejecutados en los diferentes QueryPlayers.

De forma análoga al caso anterior el QueryPlayer secuencial y el que prioriza los nodos mantenían su comportamiento, sin embargo, los QueryPlayer que explotaban la dependencias para acelerar las consultas daban una considerable mejora de tiempo, mientras que el QueryPlayer secuencial podía tardar del orden de $5x10^6$ milisegundos para realizar la operación de dos tablas de un millón de elementos, tanto el QueryPlayer con búsqueda de dependencias en el mismo nivel y el QueryPlayer con búsqueda de dependencia en varios niveles reducían ese tiempo a de $3x10^6$ milisegundos, esto es debido a que al simplificar la consulta bidimensional el conjunto resultado es mucho menor por lo que la operación se realiza con una tabla de menos elementos, aunque es una reducción de tiempo considerable, aún es un tiempo de ejecución muy elevado.

Consulta con operación Join y dependencia en el segundo nivel

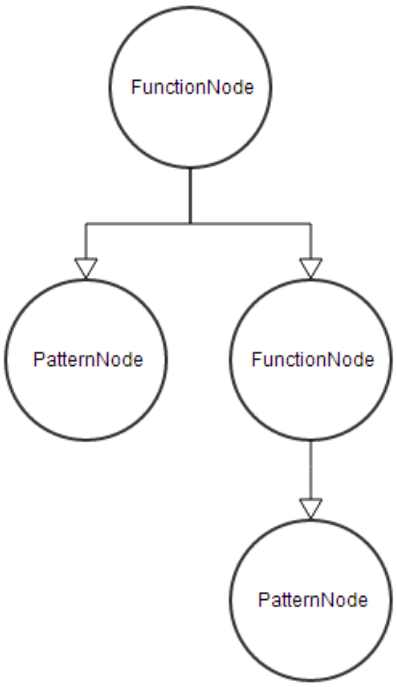


Ilustración 37

Supongamos que nos facilitan un árbol de operaciones cuyo FunctionNode es un Join, y que tiene dos hijos, uno de ellos es un patrón unidimensional y otro es otro FunctionNode, también con una operación de Join pero con un único hijo, un patrón bidimensional que además presentan una dependencia en el sujeto, finalmente son ejecutados en los diferentes QueryPlayers.

De forma análoga al caso anterior el QueryPlayer secuencial y el que prioriza los nodos mantenían su comportamiento, además se unía a ellos, como era de esperar el QueryPlayer que se aprovechaba de las dependencias entre patrones al mismo nivel para acelerar las consultas, sin embargo, el QueryPlayer que detectaba la dependencia a distintos niveles conseguía un rendimiento, aunque aún muy alto, mejor que los anteriormente mencionados, mientras que los tres primero invertían del orden de $5x10^6$ milisegundos en realizar la operación el QueryPlayer con búsqueda de dependencia en varios niveles reducían ese tiempo a de $3x10^6$ milisegundos, aún es un tiempo de ejecución muy elevado.

11. Conclusiones

El cometido de este proyecto era diseñar un motor de gestión de ejecución de patrones basados en tripletas y en la tecnología RDF, así como la creación de los resultados devueltos y la consulta con el almacenamiento a más bajo nivel, además que sea independiente de la semántica de los datos y que permita resolver las diferentes consultas.

Teniendo en cuenta los objetivos nombrados, concluimos que se han alcanzado los hitos concernientes al QueryPlayer. Se ha desarrollado un motor de gestión de ejecución de patrones basado en tripletas, capaz de recibir una petición de consulta por parte del componente Interpreter de Triskel, con capacidad para identificar y aprovechar las dependencias entre patrones y comunicarse con el almacenamiento de Triskel, el TripleStore.

Además a la hora de desarrollar software ha quedado más que demostrado que llevar unas buenas maneras de programación se hace esencial y más aún cuando trabajas en equipo, desarrollar código modular, limpio, legible y flexible obtiene un papel más que destacable en el mundo del desarrollo.

Tampoco podemos olvidar las disciplinas de desarrollo empleadas en este proyecto, desarrollar siguiendo un desarrollo basado en pruebas ha convertido la tarea del desarrollo en toda una experiencia, si a eso unimos el desarrollo en parejas, las soluciones obtenidas gozan de mayor riqueza al ser el código desarrollado afrontado desde distintos puntos de vista.

En cuanto a la gestión del proyecto SCRUM, aun con los ajustes necesarios acorde al nicho donde iba a ser aplicado, no ha permitido gestionar al equipo y las tareas de forma eficiente, además de ser un recordatorio de las diferentes tareas que se han llevado a cabo durante todo el proyecto.

Por tanto, consideramos cumplidos los hitos concernientes al proyecto QueryPlayer.

12. Trabajo futuro

En el terreno de seguir construyendo un mejor QueryPlayer los pasos deberían encaminarse hacia los siguientes hitos.

- Profundizar en la resolución de patrones contemplando detalles de algunas operaciones, como podría ser el cierre transitivo.

- Desarrollar un QueryPlayer basado en hilos, lanzándose a ejecutar los diferentes patrones que intervienen en la consultas y recuperándolos para volver a lanzar las operaciones.

- Ampliar el QueryPlayer para que pueda aceptar varias peticiones de consulta en paralelo.

13. Bibliografía

- Información concerniente a Web semántica

  http://www.w3c.es/Divulgacion/GuiasBreves/WebSemantica

- Información concerniente a SPARQL

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

- Información concerniente a RDF

  http://www.w3.org/RDF/

- Información concerniente a SCRUM

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

- Información concerniente a TDD

  http://www.agiledata.org/essays/tdd.html

- Manifiesto Agil

  http://agilemanifesto.org/iso/es/

- Pair-Programming

  http://www.extremeprogramming.org/rules/pair.html