ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
Grado en Ingeniería de Computadores

Microservicio REST basado en Apache Spark para cruce-SQL de Fuentes de Datos Cassandra

REST Microservice based on Apache Spark for SQL-Join of Cassandra's Data Sources

Realizado por
Juan Antonio Aguilar Jiménez
Tutorizado por
María del Mar Roldán García - Antonio J. Nebro Urbaneja
Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, Junio 2018

Fecha defensa:
El Secretario del Tribunal

Resumen: En este trabajo Fin de Grado (TFG) se ha desarrollado una herramienta genérica, siguiendo una arquitectura de Microservicio REST, para implementar operaciones de cruce (join) en fuentes de datos Cassandra de gran volumen con Apache Spark. Además, la herramienta se ha aplicado a un caso de uso de la Web Semántica, con el que se ha conseguido evaluar consultas SPARQL en un repositorio de datos Apache Cassandra que almacena una ontología OWL materializada. Apache Cassandra es una base de datos NoSQL (Not only SQL) distribuida orientada a columna, cuyo lenguaje de consultas, por razones de rendimiento y de la propia arquitectura de la base de datos, no permite hacer operaciones de tipo join entre tablas. La herramienta genérica desarrollada en este TFG cubre esta carencia de forma escalable gracias al uso de Apache Spark. Además, se ha conseguido desacoplar la lógica necesaria para realizar dichos cruces para el Caso de uso Específico. Esto permite aplicar dicha herramienta genérica a otros casos de uso futuros. Como producto final, se ha desarrollado un interfaz Web que permite ejecutar consultas SPARQL sobre una ontología con información sobre diferentes disciplinas artísticas. Las consultas son modificables por el usuario, pudiendo éste generar cualquier consulta nueva sobre el conocimiento almacenado.

Palabras claves: Cassandra, Spark, Web Semántica, SPARQL, Microservicio, REST, API, Big Data, Join, Python, Node, Angular, Apollo, Daphne.

Abstract: In this End-of-Degree project (TFG) a generic tool has been developed, following a REST Microservice architecture, to implement join operations in large volume Cassandra data sources with Apache Spark. Besides, the tool has been applied to a Semantic Web use case, with which it has been possible to evaluate SPARQL queries in an Apache Cassandra data repository that stores a materialized OWL ontology. Apache Cassandra is a distributed column-oriented NoSQL (Not only SQL) database, whose query language, for performance reasons and the database architecture itself, does not allow join-type operations between tables. The generic tool developed in this TFG covers this lack in a scalable way thanks to the use of Apache Spark. Therefore, it has been possible to decouple the logic necessary to make such crossings for the specific Use Case. This allows applying this generic tool to other future use cases. As a result, a Web interface has been developed that allows executing SPARQL queries about an ontology with information concerning different artistic disciplines. The queries can be modified by the user, who can generate any new query regards the stored knowledge.

Keywords: Cassandra, Spark, Semantic Web, SPARQL, Microservice, REST, API, Big Data, Join, Python, Node, Angular, Apollo, Daphne.

Dedico este trabajo a mis amigas Sara y Mónica por escucharme y servirme de apoyo. Y por supuesto a mis padres, Francisco y Antonia, porque no sería posible sin lo que ellos han luchado para que mis hermanos y yo pudiéramos progresar.

Agradecimientos

Quiero agradecer al grupo de investigación Khaos del Departamento de Lenguajes y Ciencias de la Computación de la Universidad de Málaga, y en especial a mis tutores, María del Mar Roldán y Antonio Nebro, que forman parte del mismo, por la oportunidad que me han dado.

Índice general

Índice de figuras

Código Fuente

1

Indroducción

> It is possible to invent a single machine which can be used to compute any computable sequence.
>
> Alan Turing

Apache Cassandra [21] [1] *[fig. 1]* es una base de datos distribuida de tipo *Not Only SQL* 2.1 (NoSQL), cuyo lenguaje de consultas, denominado *Cassandra Query Language [26]* (CQL), no permite hacer operaciones tipo *join* entre tablas (*Inner Join*, *Left Join*, *Right Join*, *Full Join*, *XOR Join*, *Cross Join*) (2.2.3). Esta limitación de Cassandra se debe a razones de rendimiento y de la propia arquitectura de la base de datos.

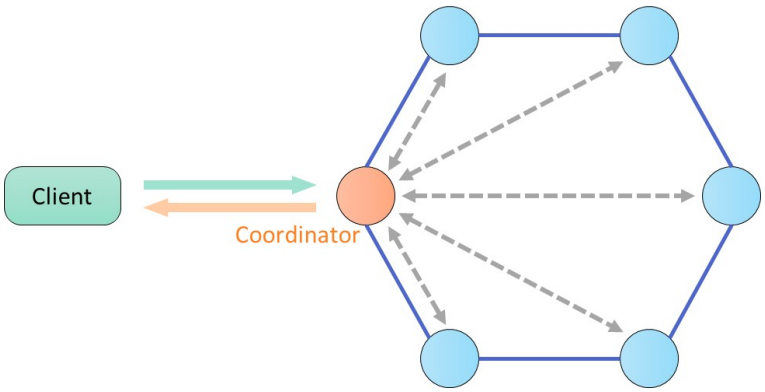

Figura 1.1: Estructura de nodos en Apache Cassandra [21]

Este Trabajo Fin de Grado (TFG), dentro de la línea de "TFG Análisis del Big Data", se aplicará para el caso de uso, en un proyecto del grupo de investigación Khaos[2], entre cuyas líneas de investigación están tanto la de este TFG como la de "Tecnologías Semánticas". En el marco de las investigaciones que llevan a cabo, tienen la necesidad de razonar y almacenar grandes volúmenes de datos de ontologías estructuradas del campo de la Web semántica. Partimos de un repositorio resultado de otro proyecto [LM14] en el cual se construye un razonador *Web Ontology Language [40]* (OWL) sobre una base de datos Apache Cassandra (2.3). Tanto el razonamiento semántico, como las consultas sobre el conocimiento inferido deben ser realizados sin generar congestión en el sistema, independientemente del tamaño de la ontología razonada. Por tanto, ha de realizarse con técnicas escalables. Apache Cassandra cumple estos requerimientos, salvo porque no provee mecanismos para hacer operaciones tipo *join*. Éstas operaciones son necesarias, por ejemplo en el caso de uso propuesto(3.2), para

[1] ver *Acerca de cómo leer esta memoria* (1.2).

[2] ***Khaos*** es el nombre de un grupo de investigación del Departamento de Lenguajes y Ciencias de la Computación de la Universidad de Málaga.

poder extraer conocimiento de una ontología materializada en una base de datos orientada a columna mediante consultas *SPARQL Protocol and RDF Query Language* (SPARQL) (3.1.7).

1.1. Objetivos

El objetivo principal de este proyecto es poder realizar operaciones tipo *join* definidas en el álgebra relacional, sin penalizar el rendimiento al analizar grandes volúmenes de datos almacenados en una base de datos NoSQL, escalable y distribuida, como es Apache Cassandra (2.3). Por este motivo se utiliza Apache Spark (2.4) cuyas características intrínsecas, permiten paralelizar procesos de analítica de datos usando ordenadores en *cluster* (incluso en la nube), escalando horizontalmente para no crear cuellos de botella que impidan la utilidad práctica del análisis. En definitiva, herramientas dentro del ecosistema del ***big data*** y a la altura de los requerimientos técnicos de este tipo de retos computacionales.

1.2. Acerca de cómo leer esta memoria

Los acrónimos están subrayados, de forma que en la versión digital se puede seguir el hiperenlace para verlo, y en la copia en papel indica que hay que ir al apéndice de acrónimos para localizarlo. Al igual que se pueden seguir las referencias cruzadas, que se denotan con el número de la sección, subsección, etc. entre (paréntesis).

Las palabras en **negrita** denotan un término, cuya definición está en el apartado de términos al final de la memoria.

Los términos anglosajones están escritos en *cursiva*, menos los nombres propios (p.ej.: aplicaciones, módulos, bibliotecas, etc.).

Las referencias tanto de la *World Wide Web* (WWW) como bibliográficas están puestas entre [corchetes].

Las figuras irán con letra cursiva y entre corchetes, se denotará con el prefijo fig., *(*[fig. <figura>]).

1.3. Descripción de Contenidos

Introducción (1) Se refiere a éste mismo capítulo, donde se introduce al problema que se pretende solucionar en este TFG: Motivación, Objetivos y una sección con un pequeño resumen del contenido de cada capítulo.

- **Situación Actual (2)**. En este capítulo se revisa la situación previa al TFG. Nos adentraremos en el *state-of-the-art* de las tecnologías de bases de datos NoSQL y su vinculación con el ***big data***. Analizaremos el álgebra relacional, haciendo hincapié en la funcionalidad no implementada en la base de datos Cassandra (2.3). Dicha funcionalidad se suplirá usando técnicas escalables. Haremos un recorrido sobre Apache Cassandra (2.3), también sobre las características de Spark (2.4), ***framework*** que ha

sido escogido para realizar este TFG y que nos va a permitir el cruce de datos entre las distintas "tablas" de Cassandra (2.3), apoyándonos para ello en el Conector Spark-Cassandra (2.5).

- **Tecnologías de la Web Semántica (3)**. Aquí nos adentramos en el Caso de Uso y en qué tecnologías se utilizan actualmente en la Web Semántica (3.1), cómo se representa el conocimiento de forma eficaz (3.1.1), qué es una Ontología de la Web Semántica, qué lenguajes se utilizan para representar el conocimiento y qué es un razonador semántico. También haremos una introducción sobre el lenguaje de consulta SPARQL (3.1.5), y de cómo una Ontología Materializada puede incrementar el rendimiento ante ontologías muy grandes.

- **Desarrollo del proyecto (4)**. Este capítulo detalla las fases ejecutadas en la realización de este TFG. Se ha seguido una metodología ágil, con ciclos iterativos, desde el diseño a las pruebas. Partiendo de la idea inicial ya mencionada, implementaremos una herramienta que pueda realizar *joins*, a la manera de las bases de datos *Structured Query Language* (SQL), pero sin penalizar el rendimiento. Con este comienzo, iterar modificando el prototipo hasta convertirlo en una pieza software para aplicarla al Caso de Uso de la Web Semántica (6), con el objetivo final de que pueda ser utilizada como **microservicio** común a aquellas aplicaciones y experimentos del grupo de investigación que requieran cruzar datos entre fuentes de datos Cassandra.

- **Microservicio *Apollo* (5)**. En este capítulo se describen las especificaciones de la pieza software que es el núcleo fundamental de este TFG, que es capaz de recibir una serie de consultas en un formato que hemos denominado `Apollo Query`, basado en *Javascript Object Notation [39]* (JSON), y de traducir dichas consultas a instrucciones sobre Spark de forma adecuada (2.4) para realizar las consultas. En resumen, extraer los datos desde Cassandra y hacer los cruces de datos necesarios para obtener la información de acuerdo a la consulta.

- **Aplicación al Caso de Uso (6)**. Descripción de cómo se utilizado el **microservicio** propuesto para acoplarlo a un ejemplo real cuyas necesidades, en cuanto a cruce de datos, quedan cubiertas por la herramienta desarrollada. Para acoplar la herramienta génerica *Apollo*, se describe el desarrollo de otro **microservicio** que traducirá las consultas SPARQL en el formato especificado para ser interpretadas por *Apollo* (5). Además, se ha desarrollado un cliente a modo de ejemplo de consumo.

- **Conclusiones y trabajos futuros (7)**. Conclusiones acerca de la consecución de los objetivos planteados en el TFG, así como una reflexión sobre mejoras y trabajos futuros.

- **Tecnologías utilizadas (A)**. Apéndice con una breve descripción de las tecnologías utilizadas.

2

Situación Actual

> We may say most aptly that the Analytical Engine weaves algebraical patterns just as the Jacquard loom weaves flowers and leaves.
>
> Ada Byron

En plena explosión de las tecnologías de la información aplicadas a la Web, y de nuevos paradigmas de almacenamiento, las bases de datos NoSQL están siendo cada vez más usadas, siendo objeto de I+D por parte de entidades de primer nivel.

Una característica que ha disparado esta tendencia es su carácter ***schemaless***, lo que implica que son más flexibles a la hora de almacenar información no estructurada o semi-estructurada. Otra característica importante de muchas de ellas es que son distribuidas y escalan bien horizontalmente. Debido a la ingente cantidad de datos que se genera hoy en día, gran parte desde la Web y especialmente desde la llamada Web 2.0, con gran cantidad de datos: blogs, redes sociales, ***linked data***, Internet de las Cosas, Ciudades Inteligentes, comercio electrónico, analítica Web, aplicaciones corporativas, logs de plataformas en la nube, etc. Para ello, se necesitan estructuras capaces de recoger esta información, o un resumen en su caso, con la posibilidad de escalar en función del tamaño y de la rapidez con que se generan dichos datos.

2.1. Bases de datos NoSQL

Las bases de datos NoSQL también se les denomina tipo: *Basically Available, Soft state and Eventual Consistency* (BASE) como contrapartida al acrónimo para denominar a las bases de datos relacionales: *Atomicity, Consistency, Isolation and Durability* (ACID), que denota la flexibilidad de la filosofía NoSQL frente a la Relacional.

Existen multitud de bases de datos NoSQL, y pueden ser clasificadas por varios criterios, uno de ellos sería el formato de la información que almacenan, siendo las más demandadas las que alcanzan un mayor grado de especialización en este sentido. Por ejemplo, existen bases de datos NoSQL orientadas al formato clave-valor (p.ej.: Redis), son en definitiva grandes tablas *hash* que permiten recuperar información de forma rápida a pesar de su enorme tamaño. En cambio otras, están orientadas a documento (p.ej: Mongo DB, Elasticsearch), es decir, no se guarda un valor simple, sino que el valor guardado es todo un documento, datos semi-esctructurados en formato JSON (p.ej. MongoDB), o cualquier tipo de documento ya sea en texto plano, *eXtensible Markup Language* (XML) o *Hyper Text Markup Language* (HTML) (p.ej. Elasticsearch). La información almacenada es en muchos casos de tipo *Write Once Read Many* (WORM), indexada para búsquedas. También existen orientadas a grafos

(p.ej: Neo4J, ArangoDB), en las cuales se da más importancia a la relación entre los datos y donde se pueden aplicar técnicas de análisis de grafos. En el caso de la base de datos Apache Cassandra, se habla de estructuras de datos orientadas a columna (2.3), en el que dicha columna está formada por un conjunto indeterminado de pares clave/valor.

En las bases de datos NoSQL hay también varios tipos en cuanto a su clasificación según el **Teorema CAP de Brewer**, cuestión a tener en cuenta a la hora de elegir una en función del caso de uso concreto *[fig. 2.1]*.

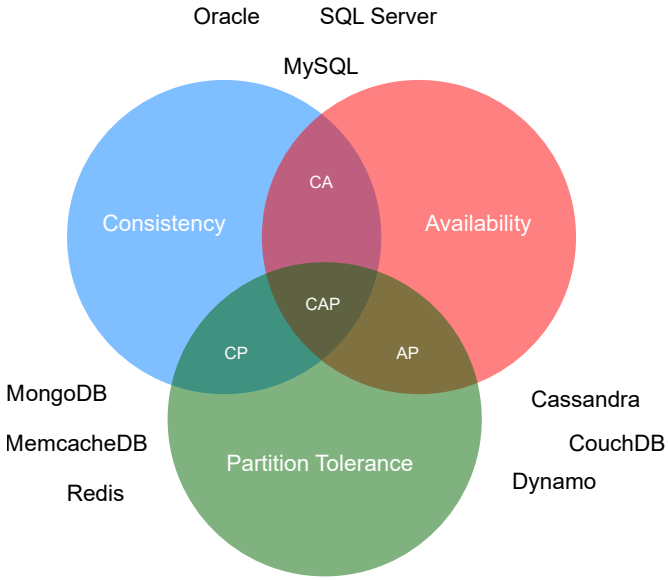

Figura 2.1: Clasificación de Bases de Datos NoSQL según Teorema CAP de Brewer.

Otra característica que suele diferenciar a las bases de datos NoSQL y las bases de datos relacionales, es que no implementan operaciones tipo *join*[1], característica fundamental como motivación para este TFG.

2.2. Álgebra Relacional

El álgebra es la parte de las matemáticas que se encarga de generalizar las operaciones aritméticas entre relaciones, que se definen como conjuntos de tuplas.

En el caso de las bases de datos SQL[2], el álgebra relacional se aplica definiendo las distintas operaciones "aritméticas" entre tablas compuestas por filas.

El álgebra relacional consta de operaciones unarias o binarias, que tienen como entrada relaciones, obteniendo como resultado otra relación. Se contemplan varias operaciones: unas básicas y otras derivadas a partir de las anteriores. [44] [Cod72]

[1]con tipo *join*, además de todos los tipos de *SQL join* podemos también incluir las operaciones del álgebra relacional: Unión, Intersección y Resta

[2]las bases de datos relacionales también se definen como SQL porque soportan este método de consulta.

2.2.1. Operaciones básicas

1. Proyección (Π), es una operación unaria donde se crea una nueva tabla a partir de solo algunas columnas de la tabla original $\boxed{Z = \prod_{R_1,R_2,\ldots,R_n}(R)}$ donde Z tendría los atributos $R_1..R_n$ de cada tupla de la relación R.

2. Selección (σ) , es una operación unaria, donde se crea una nueva relación a partir de solo algunas tuplas de la relación original. Es decir, los datos de la relación original son filtrados por un criterio P. $\boxed{\sigma_P(R)}$

3. Renombrar (ρ), es una operación unaria donde se renombra un atributo con nombre b, de una relación R, con el nombre a. Por lo demás, la relación resultado es igual a la relación R. $\boxed{Z = R_{a/b}}$

4. Producto Cartesiano ($\times$) de dos relaciones *[fig. 2.2]*. Cada tupla de la primera relación R se combina con cada tupla de la segunda relación S, formando nuevas tuplas por la combinación de ambas. El número de tuplas de la relación resultado Z, será el producto del número de tuplas de R por el número de tuplas de S. $\boxed{Z = R \times S}$

5. Unión ($\cup$) *[fig. 2.3]*, operación binaria. En la unión de R y S se añaden a la relación resultado Z las tuplas de las dos relaciones originales, cuyos atributos deben ser concordantes en número. $\boxed{Z = R \cup S}$

6. Resta ($-$) *[fig. 2.5]*, operación binaria que selecciona las tuplas de la primera relación que no estén presentes en la segunda. $\boxed{Z = R - S}$

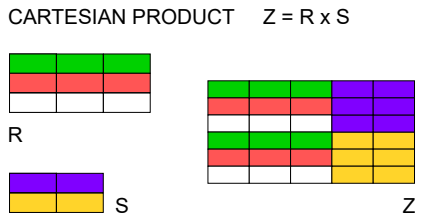

Figura 2.2: Producto Cartesiano.

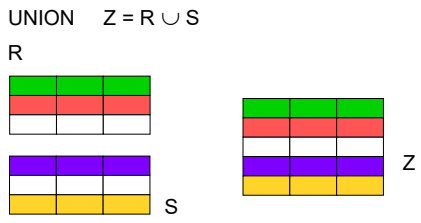

Figura 2.3: Union.

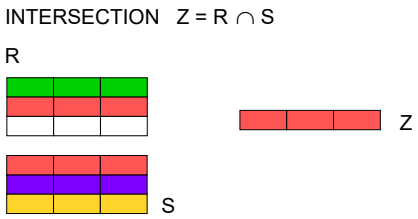

Figura 2.4: Intersección.

2.2.2. Operaciones derivadas

1. Intersección ($\cap$) *[fig. 2.4]*, operación binaria que dadas dos relaciones R y S, la relación resultante Z contendrá las tuplas de R que también estén en S. Puede expresarse como operación derivada. $\boxed{Z = R \cap S = R - (R - S)}$ *[fig. 2.4]*

2. Unión Natural ($\bowtie$), Producto Natural o *Join* es una operación binaria que dadas dos relaciones R y S, combina tuplas seleccionadas de ambas relaciones por un criterio θ, de forma que la relación resultado proyecta todas las columnas de las dos relaciones. Se trata de una operación derivada de otras básicas. $\boxed{Z = \sigma_\theta(R \times S)}$

   En las bases de datos SQL se contemplan diferentes tipos de "uniones naturales" o *joins* (2.2.3) *[fig. 2.6]*.

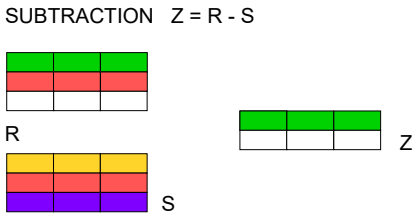

Figura 2.5: Resta.

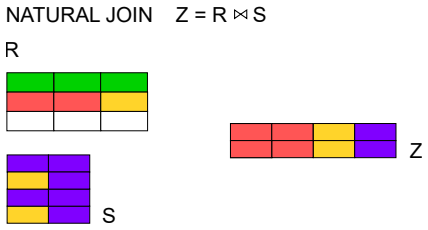

Figura 2.6: Unión Natural.

2.2.3. Tipos de *SQL Join*

En la figura *[fig. 2.7]* se pueden ver representadas mediante diagramas de Venn, las diferentes posibilidades que las bases de datos relacionales implementan de la Unión Natural entre dos tablas. En cada una de ellas puede verse el esquema simple de como sería una consulta SQL. Algunas de ellas son primitivas y las otras pueden expresarse como una modificación de las operaciones primitivas (p.ej.: operación de selección σ_P).

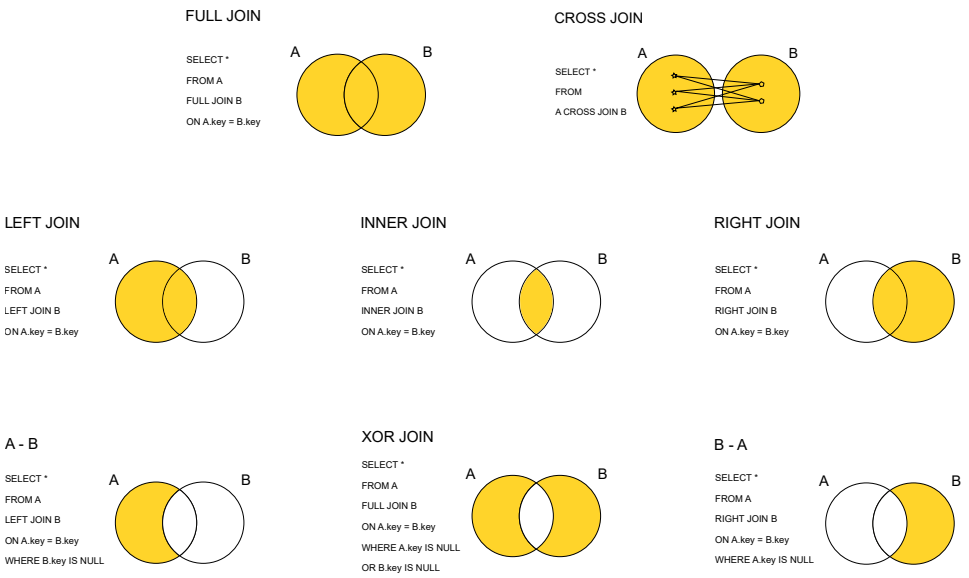

Figura 2.7: Tipos de *SQL join*.

- ***Cross Join***. Se trata del producto cartesiano de dos tablas $A \times B$ (2.2.1), donde cada fila de la tabla A se relaciona con cada una de las filas de la tabla B. De forma que $A_{nm} \times B_{pq} = Z$, donde Z tendría $n * p$ filas y $m + q$ columnas. No establece ningún criterio de comparación, el criterio sería "*todas con todas*".

- ***Inner Join***. *Inner Join* o Unión Natural de dos tablas $A \bowtie B$ se corresponde con la Unión Natural del álgebra relacional (2.2.2), derivada de otras básicas (2.2.1). $\boxed{\sigma_\theta(A \times B)}$

- ***Left Join***. Otro tipo de *join* es la operación *Left Join* entre dos tablas, también denominada *Left Outer Join*, $A \ltimes B$. Es una extensión de la operación *Inner Join*, donde también se añadirían al resultado las filas de A que no cumplan el criterio θ, siendo las columnas de B de valor *nulo* en estas últimas.

- ***Right Join***. La operación A *Left Join* B ó $A \rtimes B$ es equivalente a $B \ltimes A$. Aunque, por claridad de las consultas, es conveniente usarla como *Right Join*.

- ***Full Join***. *A Full Join B* ó $A \rtimes\ltimes B$, la consulta equivalente en SQL puede verse en *[fig. 2.7]*, también podría expresarse como $A \rtimes B \cup A \bowtie B$ sin que se repitan las filas de la interseción[3].

- $\boldsymbol{A - B}$. Se podría definir como la operación de seleccionar las filas de A que no "casan" con B dado un criterio θ. Hay que distinguirla de la resta del álgebra relacional (2.2.1), que es una operación puramente de conjuntos y sin establecer un criterio θ. También podría expresarse como la resta algebraica. $\boxed{(A \rtimes B) - (A \bowtie B)}$

- $\boldsymbol{B - A}$. De forma equivalente a $A - B$, podría expresarse como $(A \rtimes\ltimes B) - (A \bowtie B)$.

- ***Exclusive OR Join***. La operación *Exclusive OR* o Disyunción Exclusiva ($\triangle$), se puede descomponer en $(A - B) \cup (B - A)$. El resultado tendrá las filas de A y de B que no pertenezcan a la intersección de A y B.

2.3. Apache Cassandra

Apache Cassandra es una base de datos NoSQL, de tipo clave-valor, orientada a columna: los datos siempre se acceden por la clave, y el valor está compuesto por un número de columnas indeterminado *[fig. 2.8]*. En una misma "tabla"[4] podemos tener filas con diferente número de columnas, en un gran número como límite. Esta característica, unida a su arquitectura en ***cluster***, hace a la base de datos Cassandra idónea para su uso dentro del ecosistema ***big data***[5].

El lenguaje de consulta es el CQL, sustituyendo al SQL de las bases de datos relacionales. Es bastante parecido, permite: proyección, selección, renombrar columnas y agregaciones. Pero, al contrario que SQL, no permite consultar tablas relacionadas unas con otras debido a la propia arquitectura de la base de datos Cassandra y a la propia filosofía de las bases de datos NoSQL[6].

La base de datos Cassandra presenta una arquitectura distribuida (1), que escala horizontalmente gracias a la estructura en ***cluster***. Cada nodo contiene una parte de los datos denominada partición, y puede replicar los datos de otros nodos para implementar la tolerancia a fallos. Existe un nodo primario, que es con el que comunican los clientes. Cuando un cliente inserta un dato, el nodo primario aplica una función ***hash*** que decide a que partición pertenece, comunicando con el nodo que gestiona dicha partición o, si éste está "caído", con los que mantienen una réplica de dicha partición. El número de nodos que mantienen una

[3]en el álgebra relacional, basada en la teoría de conjuntos, no habría repeticiones, pero en las bases de datos relacionales, dependiendo de la implementación SQL, podrían darse duplicados, por lo que habría que tenerlo en cuenta. Dichas implementaciones suelen incluir el modificador *DISTINCT* para las proyecciones, que indica que la tabla resultado no contendrá duplicados. También en algunas, operaciones como la unión tiene dos posibilidades: *Union*, sin duplicados, y *Union All*, con duplicados.

[4]denominada *colum family* antes de la versión 3, a partir de aquí, recibe el nombre de *table*. El nombre *super-column family* sería el equivalente a una vista en bases de datos SQL. Desde la versión 3 se renombra a *view*.

[5]hasta 2^{31} celdas por partición en la versión 3.3 [14]

[6]modelos de datos denormalizados, con datos repetidos en detrimento de la consistencia, datos almacenados en el formato que se quiere sean consultados, etc.

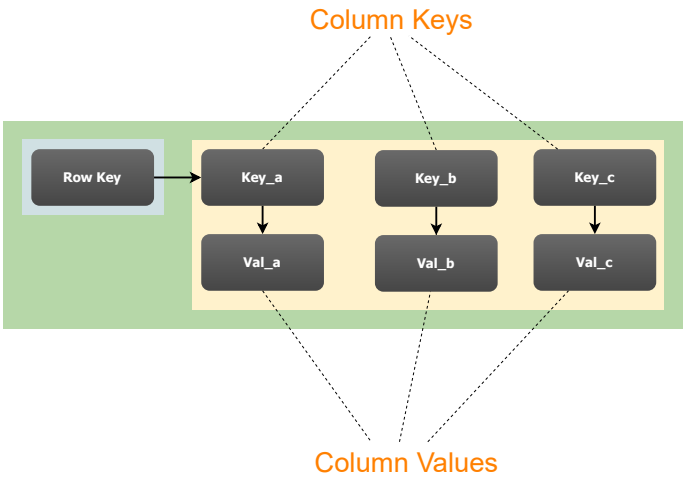

Figura 2.8: Esquema de las estructuras clave-valor en Cassandra.

partición, se modela al definir el *keyspace*, que es un nivel organizativo superior al de tabla y sería equivalente al concepto de *schema* en las bases de datos relacionales. A la hora de definir un *keyspace*, se decide qué factor de replicación (parámetro `replication_factor`) va a tener. Aunque los datos se replican en el momento de la inserción, existe un protocolo denominado *gossip* (o rumor, en castellano) por el cual, cada cierto tiempo, los nodos que contienen réplicas comparan los códigos Cyclic Redundancy Check (CRC) de sus datos con los de los nodos replicados, intercambiando los que no están actualizados para mantener la consistencia[7]. En la definición de una tabla, se especifica la *primary key*. Tiene dos partes: a la primera, formada por uno o varios campos, se le denomina *partition key*, que es la que determinará en que nodo se graban los datos (partición). La segunda parte (opcional) se denomina *clustering key*, y definirá el orden de la tupla dentro de la partición, en caso de ser especificada *[fig. 2.9]*.

2.4. Apache Spark

Es un ***framework*** software de código abierto, originalmente desarrollado por la Universidad de Berkeley en California y actualmente mantenido por la Fundación Apache. Está especializado en el procesamiento paralelo de grandes volúmenes de datos. Presenta una arquitectura en ***cluster***. Está optimizado para velocidad, siendo mucho más rápido que MapReduce, aunque éste último está diseñado para mayor volumen de datos. Soporta *Application Programming Interfaces* (APIs) para varios lenguajes: Scala, Python, Java y R. Además tiene bibliotecas de alto nivel; como por ejemplo Spark SQL, que se utiliza para conectar con diversos motores de bases de datos, entre las cuales está Apache Cassandra[8].

[7] pese a que Cassandra estaría encuadrado en el vértice A-P según el **Teorema CAP de Brewer**, gracias a este protocolo se atenúa en gran medida la falta de consistencia, por lo que podría decirse que es Eventualmente Consistente, tal y como se refiere en el término BASE, aunque algunos nodos no hayan estado disponibles durante el proceso de inserción. Para la lectura, igualmente, si un nodo no está *online*, según el factor de replicación pre-establecido, se podrá servir el dato de una de las réplicas.

[8] gracias a *Spark-Cassandra Connector* (A)

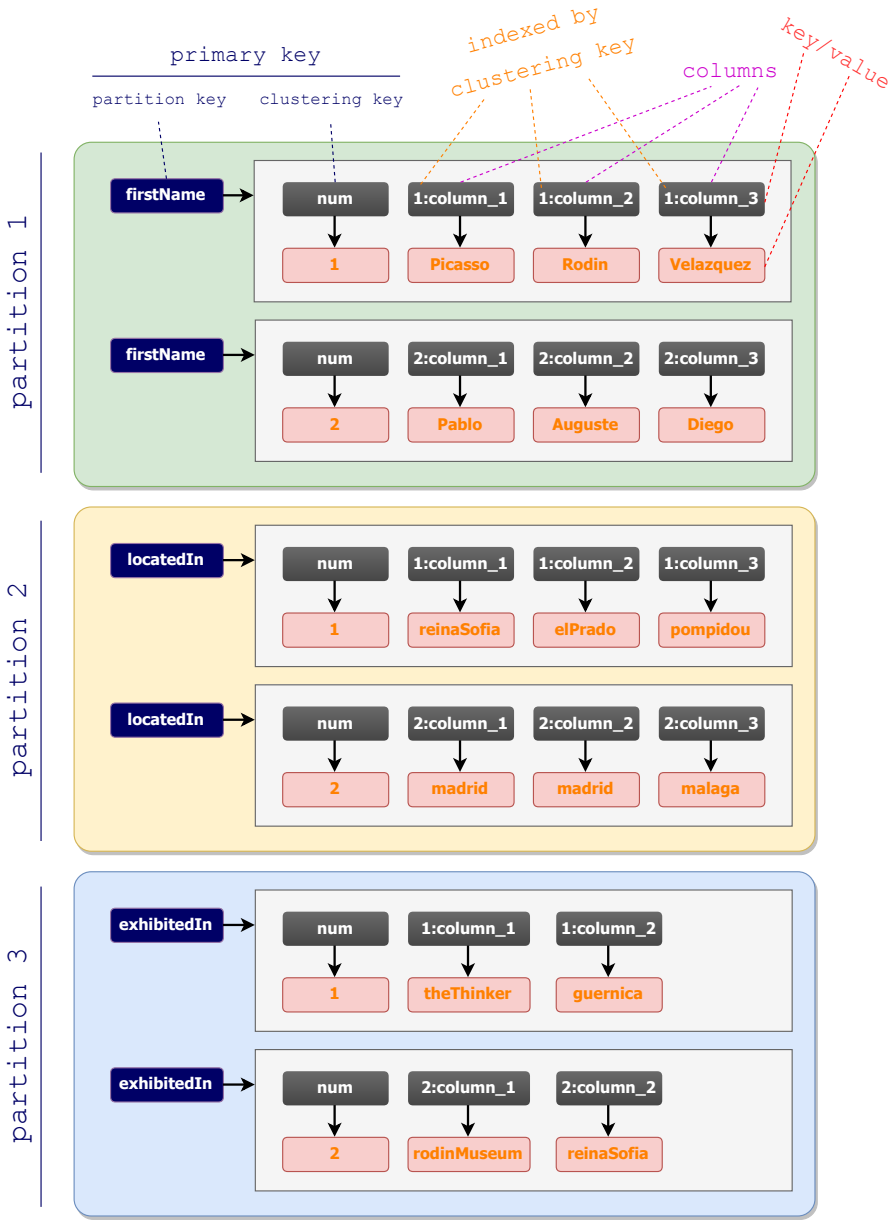

Figura 2.9: Esquema de la estructura interna de los datos en Apache Cassandra.

Al igual que Cassandra, Spark puede escalar horizontalmente y está considerado como un ***framework*** para ***big data***.

La computación en Apache Spark se basa en una estructura de datos denominada *Resilient Distributed Dataset* (RDD)[9], la cual puede ser almacenada y manipulada por uno, varios o todos los nodos del ***cluster***. Un RDD es inmutable, de solo lectura. Es decir, una vez que se ha creado, no se puede modificar. Las diferentes operaciones que trabajan con RDD generan un nuevo RDD transformado.

Un RDD soporta dos tipos de operaciones [24]:

1. Transformaciones, son aquellas que devuelven un nuevo RDD. Entre ellas tenemos:

[9]la R significa *Resilient*: Resiliente, tolerante a fallos.

map, flatMap, filter, join, union, intersection, reduceByKey, groupByKey, etc.

2. Acciones, son aquellas que devuelven un resultado: reduce, count, collect, take, etc.

Una estructura de más alto nivel que los RDDs son los Datasets, y Spark provee de mecanismos para poder pasar de Dataset a RDD y viceversa, aunque gran parte de las operaciones que pueden realizarse con RDDs también pueden aplicarse a Datasets. Dataset es un tipo abstracto, en implementaciones como PySpark se trabaja con estructuras que heredan de estos Datasets denominadas DataFrame. El concepto de Dataset es más cercano al concepto de tabla, donde los datos se disponen en filas y columnas. Dichas columnas, pueden accederse por nombre, lo que simplifica enormemente su manipulación y la legibilidad del código, pues se acerca más a la lógica de negocio de las aplicaciones, y la correspondencia con las estructuras que se cargan en los Datasets es mucho más directa.

2.5. Conector Spark-Cassandra

El Conector Spark-Cassandra es una pieza software de código abierto, desarrollada por Datastax, empresa que mantiene Apache Cassandra. Su código fuente está escrito en Scala, pudiendo ser compilado a *bytecodes* de Java empaquetados en formato *Java ARchive* (JAR). Este JAR se integra con la instalación de Spark permitiendole acceder a Cassandra y cargar los datos en un RDD (o un DataFrame de PySpark) [16].

3

Caso de Uso: Tecnologías de la Web Semántica

Knowledge comes, but wisdom lingers.

Alfred Tennyson

La Web Semántica [BLHL01] es una evolución de la WWW donde la información está distribuida en documentos que se enlazan unos con otros mediante hipervínculos. En la WWW la finalidad es que los datos sean generados, mantenidos, accedidos e interpretados por humanos. Es decir, una comunicación bidireccional humano-máquina, por transitividad, humano a humano. Mientras que en el caso de la Web Semántica, la finalidad es permitir también la comunicación máquina-máquina, siendo éstas las que sean capaces de "comprender" la información y razonarla, dándole valor añadido, para ser también consumida por los usuarios humanos.

3.1. Antecedentes del Caso de Uso

El lenguaje para construir la WWW es el HTML, que permite enlazar otros documentos e incrustar recursos multimedia, tales como imágenes, audio y vídeos. Así fue en un primer momento, pero ha ido evolucionando en el tiempo incorporando otros formatos estructurados, como el XML, además de código interpretable por los navegadores para generar cierta interactividad con la Web, como p.ej.: Javascript. Pero ha sido con la introducción de los lenguajes semánticos cuando se produce la verdadera evolución, a lo que también se ha denominado la Web de Datos, de la mano del *World Wide Web Consortium* (W3C) y del mismo creador de la WWW: Tim Berners-Lee.

Lenguajes semánticos, como p.ej.: *Resource Description Framework* (RDF) y RDF Schema han permitido dotar de significado a los propios datos así como de nuevas herramientas para la representación del conocimiento, siendo así interpretables por las máquinas.

3.1.1. Representación del Conocimiento

El conocimiento es una abstracción de nivel superior al de la información, y ésta a su vez de los datos mismos. El conocimiento nos permite la toma de decisiones cuyo resultado aportará un nuevo conocimiento.

La representación del conocimiento se ha convertido en una disciplina fundamental de la rama de la Inteligencia Artificial. El término representación, indica que se pretende emular el conocimiento mismo, entendiendo dicho conocimiento como la comprensión del mundo que nos rodea. Una representación del conocimiento que nos va a permitir razonar sobre el mundo

sin necesidad de actuar sobre él. En definitiva, se trata de investigar el mundo extrayendo nuevo conocimiento inferido de lo ya conocido [DSS93]. Tarea para la cual necesitamos expresar tanto el conocimiento de partida, como el nuevo conocimiento razonado de forma automática, en un proceso de retroalimentación positiva[1].

Existen diversas tecnologías de representación del conocimiento: Etiquetado, Marcos, Reglas, Redes Semánticas. Dentro de las Redes Semánticas tenemos las redes IS-A, donde existe un grafo de taxonomías cuyos enlaces etiquetados representan herencia semántica entre los nodos.

Existen también muchas notaciones y lenguajes formales dentro del ecosistema de la web Semántica:

- Lenguajes semánticos: RDF, RDF Schema, OWL, etc.
- Lenguajes de consulta: RQL, RDQL, SPARQL

3.1.2. Ontologías de la Web Semántica

La palabra ontología ha sido utilizada desde la antigüedad para designar una rama de la filosofía. En el contexto de la Web semántica, es un término sinónimo de conceptualización: una descripción formal de un dominio de un conocimiento compartido y de las relaciones entre los elementos de dicho dominio. Es pues una conceptualización de un conocimiento compartido.

3.1.3. OWL

Es un lenguaje para expresar ontologías Web basado en RDF. Los lenguajes como RDF y RDF Schema, que es una extensión de éste, están ambos derivados de XML. Estos lenguajes, son meramente formatos para escribir conocimiento, digamos que puramente sintaxis, pero OWL añade información semántica. p.e.: la frase "las casas comen peces" sería una frase sintácticamente correcta, pero desde el punto de vista semántico no tendría sentido, OWL definirá qué relaciones son posibles entre los elementos del lenguaje; es decir, una ontología.

Las ontologías pueden definirse a varios niveles de abstracción, lo que nos va a permitir profundizar más o menos en el dominio del conocimiento que se quiera describir.

3.1.4. Logica de Descripciones y Bases de Conocimiento

La Lógica de Descripciones (DL) es una manera formal de representar y estructurar el conocimiento y la base actual para el diseño de ontologías. Está basada en la lógica de primer orden y permite razonar sobre el dominio que representa.

[1]buen ejemplo de ello sería esta memoria, la cual parte de unos conocimientos previos, adquiridos con anterioridad o consultados, además de unos nuevos conocimientos y experiencias aprendidas durante el desarrollo del proyecto. Dichos conocimientos, han sido codificados tanto en lenguaje natural como en programas, siendo ahora transmitidos para poder ser indexados en la *hypermedia*, consultados y/o utilizados por otros humanos. Los cuales a su vez, podrán aportar nuevo conocimiento cerrando el ciclo.

```
...
<!-- http://khaos.uma.es/Ontologies/artists.owl#city -->
<owl:Class rdf:about="#city"/>

<!-- http://khaos.uma.es/Ontologies/artists.owl#museum -->
<owl:Class rdf:about="#museum"/>

<!-- http://khaos.uma.es/Ontologies/artists.owl#exhibits -->
<owl:ObjectProperty rdf:about="#exhibits"/>

<!-- http://khaos.uma.es/Ontologies/artists.owl#locatedIn -->
<owl:ObjectProperty rdf:about="#locatedIn">
    <rdfs:range rdf:resource="#city"/>
    <rdfs:domain rdf:resource="#museum"/>
</owl:ObjectProperty>

<!-- http://khaos.uma.es/Ontologies/artists.owl#sculptor -->
<owl:Class rdf:about="#sculptor">
    <owl:equivalentClass>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#creates"/>
            <owl:someValuesFrom rdf:resource="#sculpture"/>
        </owl:Restriction>
    </owl:equivalentClass>
    <rdfs:subClassOf rdf:resource="#artist"/>
</owl:Class>

<!-- http://khaos.uma.es/Ontologies/artists.owl#sculpture -->
<owl:Class rdf:about="#sculpture">
    <rdfs:subClassOf rdf:resource="#artwork"/>
</owl:Class>

<!-- http://khaos.uma.es/Ontologies/artists.owl#reinaSofia -->
<owl:Thing rdf:about="#reinaSofia">
    <locatedIn rdf:resource="#madrid"/>
</owl:Thing>

<!-- http://khaos.uma.es/Ontologies/artists.owl#guernica -->
<owl:Thing rdf:about="#guernica">
    <exhibitedIn rdf:resource="#reinaSofia"/>
</owl:Thing>
...
```

Código 3.1: Extracto en lenguaje OWL de la Ontología Artist

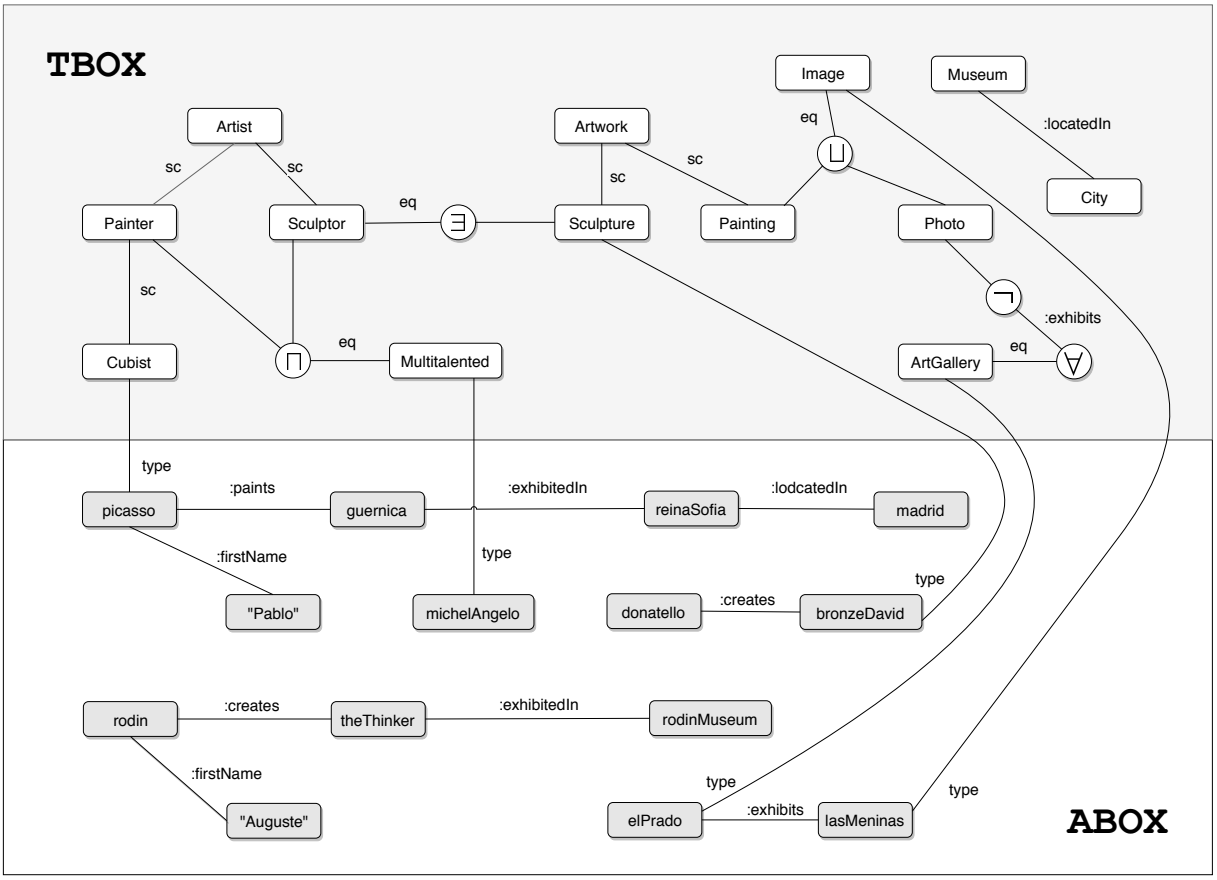

Figura 3.1: Bases de Conocimiento: TBOX y ABOX.

Una Base de Conocimiento es una implementación de una DL, que está formada por dos estructuras denominadas *Terminological Box* (TBox) y *Assertional Box* (ABox). La TBox contiene todas las relaciones entre las clases que permitirá inferir conocimiento usando un razonador semántico; mientras que la ABox tiene la información de los individuos: a qué clase pertenecen y cómo se relacionan unos con otros de forma que, aplicando reglas de la TBox, podemos agregar nuevas relaciones entre los individuos de la ABox. En *[fig. 3.1]* podemos ver un ejemplo de la Base de Conocimiento sobre la que hemos implementado el caso de uso (3.2).

3.1.5. SPARQL

Es un lenguaje estandarizado para consulta de datos en documentos RDF aunque, como pasa con SQL, existen diversas implementaciones en función del motor de almacenamiento y recuperación de documentos. Las consultas tienen una cabecera donde se indican los valores que serán retornados por la consulta, mientras que el c uerpo presenta una cláusula *WHERE* que se estructura en un conjunto ordenado de "tripletas", y cuya forma básica sería: `sujeto-predicado-objeto`. Estas tripletas, presentan una forma similar a la notación RDF, con la diferencia de que podemos introducir variables, que se denotan por un identificador precedido de un símbolo de interrogación (`?`). Cuando se ejecuta una consulta SPARQL, se comparan las tripletas con los nodos RDF de forma que se busca la coincidencia de los

```
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbp: <http://dbpedia.org/ontology/>
SELECT ?musico ?nombreMusico ?fechaNacimiento ?fechaFallecimiento
WHERE {
  ?musico dcterms:subject <http://dbpedia.org/resource/Category:Spanish_musicians>;
  rdfs:label ?nombreMusico ;
  dbp:birthDate ?fechaNacimiento ;
  dbp:deathDate ?fechaFallecimiento .
FILTER (LANG(?nombreMusico) = "es")
}
```

Código 3.2: Ejemplo de consulta SPARQL

literales, y las variables se instancian con cualquier valor con el que "casen", según defina la clausula *WHERE*.

En [*Cód.* 3.2] podemos ver un ejemplo de consulta SPARQL sobre la ***DBpedia***, donde se consultan los músicos españoles (nombre, fecha de nacimiento y fecha de fallecimiento) de los registros almacenados en la base de datos. En [18] podemos encontrar un SPARQL ***endpoint*** donde se pueden ejecutar este tipo de consultas.

3.1.6. Razonadores Semánticos

Un razonador es un algoritmo que a partir de las reglas semánticas de una ontología, es capaz de inferir conocimiento nuevo: nuevos individuos de clases o subclases (herencia), nuevos roles, o incluso nuevas reglas, de forma que aplicando un razonador de forma iterativa obtenemos un conocimiento incremental. Los Razonadores OWL son aquellos que pueden entender el formato OWL.

En el caso de las bases de conocimiento el razonamiento se realiza sobre la TBox y el conocimiento inferido se almacena en la ABox (3.1.4).

3.1.7. Materalización de ontologías sobre bases de datos NoSQL orientadas a columna

En los últimos tiempos son muchas las implementaciones de almacenamiento RDF en bases de datos NoSQL [KKTC12]. Pero no así con soporte de razonamiento OWL. En [RARGAM17] van más allá, implementando un modelo de almacenamiento OWL/RDF orientado a columna y un prototipo de herramienta gráfica para almacenar dicha ontología ya razonada[2], hasta no obtener conocimiento nuevo (materialización). Al evitar el razonamiento en memoria, se puede atacar el problema de la consulta de datos de ontologías muy grandes con una arquitectura altamente escalable. Además, permite la adición de conocimiento de manera incremental, teniendo así una fuente de conocimiento dinámico actualizada.

[2]utilizando para ello un razonador basado en MapReduce [23].

3.2. Caso de Uso: Consultas sobre una Ontología Materializada

3.2.1. Contexto

Tenemos una Ontología Materializada sobre una base de datos NoSQL; es decir, en el formato ya comentado (3.1.7); que representa una base de conocimiento sobre artistas, almacenada en un *keyspace* de Cassandra, con los elementos TBox y ABox dispersos en varias tablas (3.1.4). Nos interesa concretamente la ABox una vez razonada (3.1.7), implementada de la siguiente forma:

- La tabla `cf110`: contiene los individuos de cada clase. La estructura de `cf110` se compone de un valor clave como *partition key*, que representa la clase, y de los valores columna, que son las instancias de dicha clase. `THING` es la clase padre a la que pertenecen todos los individuos del dominio y `NOTHING` su contrapartida, la clase vacía.

ABox → cf110									
key	**column1**	**column2**	**column3**	**column4**	**column5**	**column6**	**column7**	**column8**	**column9**
NOTHING	null	null	null	null	null	null	null	null	null
#sculpture	#bronzeDavid	null	null	null	null	null	null	null	null
#artGallery	#elPrado	null	null	null	null	null	null	null	null
#painter	#picasso	#michelAngelo	null	null	null	null	null	null	null
#photo	null	null	null	null	null	null	null	null	null
#artwork	#bronzeDavid	#lasMeninas	null	null	null	null	null	null	null
#sculptor	#rodin	#michelAngelo	#donatello	null	null	null	null	null	null
#artist	#rodin	#picasso	#michelAngelo	#donatello	null	null	null	null	null
THING	#michelAngelo	#lasMeninas	#theThinker	#madrid	#guernica	#museorodin	#picasso	#reinaSofia	#donatello
#museum	#reinaSofia	null	null	null	null	null	null	null	null
#cubist	null	null	null	null	null	null	null	null	null
#image	null	null	null	null	null	null	null	null	null
#paintwork	#lasMeninas	null	null	null	null	null	null	null	null
#multitalented	null	null	null	null	null	null	null	null	null
#city	#madrid	null	null	null	null	null	null	null	null

- La tabla `cf111`: contiene las relaciones entre las instancias. Donde `key` es la *partition key* y `num` es la *clustering key*, ambas forman la `primary key`. El valor de la clave representa al predicado de la tripleta semántica. Así como, los valores con el mismo número de columna y el mismo valor en la clave `key`, están relacionados semánticamente. De esta forma, los valores de filas con valor `1` en el campo `num` corresponden con el sujeto, y los valores cuya fila presenta valor `2` en el campo `num` corresponden con el objeto.

ABox → cf111			
key	**num**	**column1**	**column2**
#exhibits	1	#elPrado	null
#exhibits	2	#lasMeninas	null
#firstname	1	#rodin	#picasso
#firstname	2	#auguste	#pablo
#locatedIn	1	#reinaSofia	null
#locatedIn	2	#madrid	null
#paints	1	#picasso	null
#paints	2	#guernica	null
#creates	1	#rodin	#donatello
#creates	2	#theThinker	#bronzeDavid
#exhibitedIn	1	#guernica	#theThinker
#exhibitedIn	2	#reinaSofia	#museorodin

3.2.2. Planteamiento

Ambas tablas pueden tener tantas columnas como sea necesario. Por ejemplo podríamos querer saber qué museos hay en Madrid, para ello, necesitamos:

1. Consultar las filas de la tabla `cf110` con clave `#museum`, es decir, las intancias de la clase museo.

2. Consultar las filas de la tabla `cf111` con clave `#locatedIn` y valor en alguna de las columnas de datos `#madrid`[3]

3. Por último hacer un *inner join* con los resultados de los puntos anteriores, usando como criterio de cruce, la columna con la instancia museo.

Como ya se ha comentado anteriormente, Apache Cassandra no permite resolver el cruce de datos planteado, por razones de arquitectura de la propia base de datos, pero para este modelo de datos es necesario, es por eso que acudimos a una herramienta como Apache Spark, por su caracter transversal y escalable.

Para ello, se utilizará una herramienta lo más genérica posible que pueda realizar cruces de datos y que se pueda aplicar al mayor número de casos de uso diferentes, pero a la vez, tan especializada que no realice ninguna otro tipo de operaciones, es por ello que se elige una arquitectura de **microservicio**, y con una interfaz tipo *Representational State Transfer* ***REST*** (REST) que garantiza la mayor accesibilidad por parte de las posibles plataformas software que se deseen utilizar como cliente, ***middleware***, etc.

[3]expresado en una consulta tipo SQL, supondría igualar el valor a todas las columnas, con el consiguiente reto que supone que el número de columnas sea indefinido. Más adelante veremos como se ha resuelto (5.1.1).

4

Desarrollo del proyecto

> Everything should be made as simple as possible, but not simpler.
>
> Albert Einstein

Desde un primer momento se pensó usar metodologías ágiles para el desarrollo de este proyecto, las cuales funcionan bien con equipos pequeños/medianos, normalmente. Se barajaron algunas de ellas, como p.ej.: *Lean*, *Scrum*, *Extreme Programming*, Desarrollo en Espiral o TDD. Como ninguna de ellas encajaría exactamente con la dinámica de este trabajo al ser un TFG individual, se ha optado por una metodología híbrida de éstas, cogiendo lo más práctico de cada una.

4.1. Metodología

Como ya se ha comentado, la metodología se ha basado en las metodologías ágiles más utilizadas, que tienen como común denominador el desarrollo del proyecto en ciclos cortos e iterativos, estableciendo, en nuestro caso, una ventana de tiempo de una semana, ya que se definieron reuniones semanales con los tutores de este TFG[1].

En cada iteración se han llevado a cabo una serie de tareas:

- Análisis de nuevas funcionalidades.
- Diseño de una batería de test, o modificación de la anterior, para comprobar que efectivamente se incorporan dichas funcionalidades.
- Implementación del código de mejora.
- Evaluación de test y comprobación de la cercanía al objetivo.
- En caso de no haber alcanzado el objetivo, repetir todo el proceso completo.

En la evaluación de los resultados, se comprueba si se ha conseguido el resultado objetivo: una herramienta que se adecúe al caso de uso (3.2) que cubra las necesidades de cruce de datos de los usuarios finales de la herramienta desarrollada para que puedan integrarla en sus propias aplicaciones. Para supervisar el proceso completo, se cuenta con los tutores del TFG, que a su vez son miembros del grupo de investigación ***Khaos***; que en terminología *Scrum* serían los *Product Owners*. Además, como desarrollador principal y único, el autor del TFG.

[1] p.ej.: en el caso de Scrum dicha ventana de tiempo se denomina ***sprints*** y es de dos semanas de duración.

4.2. Análisis

En esta primera etapa, se parte de los antecedentes del caso de uso (3.2), en la que tenemos la ABox de una ontología materializada en una partición de Cassandra, compuesta por las tablas *cf110* y *cf111*, de forma que necesitamos hacer consultas de tres tipos, que expresadas de forma algebraica serían:

1. $\Pi_{A_1,..,A_n}(\sigma_P(cf110 \bowtie cf111))$
2. $\Pi_{A_1,..,A_n}(\sigma_P(cf111 \bowtie cf111))$
3. $\Pi_{A_1,..,A_n}(\sigma_P(cf111 \bowtie cf110))$

También en algunos casos habrá que realizar operaciones para renombrar campos. Tanto las operaciones de selección σ, de proyección Π, como de renombrado de campos ρ (2.2) es posible realizarlas con el lenguaje de consulta CQL, pero no así la operación *natural join*, al no ser una característica de Apache Cassandra (2.3).

4.2.1. Análisis de requisitos

En la reunión preliminar para la realización del TFG, el encargo que se acordó fue poder realizar operaciones tipo *join* sobre la ABox de una ontología materializada, cuya estructura se explica en el artículo [RARGAM17] realizado por varios miembros del grupo de investigación ***Khaos***, grupo donde se trabaja en varios proyectos, con diversas tecnologías y utilizando varios lenguajes, estando entre ellos: *Java*, *Python* y *Scala*.

La idea de desarrollar una aplicación en Apache Spark (2.4) surgió de la experiencia en otros proyectos de investigación, de los que destaca como una plataforma de análisis de propósito general altamente escalable y flexible, además de otras muchas características:

1. Código Abierto mantenido por la Apache Foundation.
2. Varios lenguajes soportados.
3. Proyecto maduro.
4. Gran comunidad de usuarios.
5. Buena documentación en línea.
6. Utilidades en Internet de licencia libre y con código abierto.
7. SparkSQL: un módulo de Apache Spark para trabajar con datos estructurados, y con una sintáxis que recuerda a SQL.

Se decidió entonces realizar una batería de test para verificar si era factible aplicar Spark para resolver el problema. Se acuerda realizar reuniones semanales e ir viendo el progreso[2].

[2]para hacer iteraciones rápidas se decide que sean semanales, teniendo en cuenta que no hay un equipo como tal que coordinar.

4.2.2. Batería de Test Iniciales

Esta batería de test se propuso en varias reuniones con los *Product Owners*, con el objetivo de analizar la viabilidad de la solución a desarrollar. A medida que se iban consiguiendo los propuestos en la iteración anterior, se proponían nuevos a modo de ***sprint*** semanal:

1. Primera Iteración

   - Ejemplo 1: Cargar datos en tablas de Cassandra.
     - Crear una tabla con la estructura de datos a cargar.
     - Guardar datos desde un fichero CSV en la tabla creada.

     Este ejemplo no tiene que usar Spark necesariamente, los datos pueden cargarse mediante cualquier medio, dado que el producto final no tiene en principio por qué cargar datos.
   - Ejemplo 2: Capturar datos en un RDD en Spark y hacer alguna transformación.
     - Localizar la tabla cargada en el Ejemplo 1.
     - Cargar el contenido de la tabla en un RDD.
     - Hacer un ejemplo de agrupación de datos sobre el RDD del punto anterior (p.ej.: un conteo de algún dato determinado).
   - Ejemplo 3: Hacer una operación *Join* sobre un RDD y guardar los datos de nuevo en Cassandra.
     - Hacer una Union de dos RDD.
     - Hacer una cálculo sobre el nuevo RDD.
     - Cruzarlo (*join*) con otro RDD obteniendo un RDD resultado.
     - Crear una tabla destino con la misma estructura que el RDD resultado.
     - Guardar los datos en la tabla de Cassandra creada para tal fin.

2. Segunda Iteración

   - Dataset Join 01[3]. Ejemplo en Scala de la unión de dos *datasets* desde Cassandra por una columna común.
     - Recuperar desde Cassandra dos *datasets* de sendas tablas con una columna común.
     - Unión Natural de dos *datasets* igualando los datos de las columnas en común (*inner join*).
   - Dataset Join 02. Ejemplo en Scala de como unir dos conjuntos de datos y grabar el resultado en Cassandra, en una tabla no pre-existente.
     - Recuperar desde Cassandra dos datasets de sendas tablas con una columna común.

[3]En este caso, usamos la estructura de datos de alto nivel de Spark, solo disponible a partir de la versión 2.0, y cuya estructura es tabular, asemejándose así a las operaciones de las bases de datos relacionales

- Unir los dos *datasets* por una columna en común.
- Crear una tabla con una estructura concordante con el *dataset* resultado.
- Salvar los datos cruzados en la nueva tabla.

- Dataset Join 03. Realizar un ejemplo similar al anterior, pero esta vez usando la biblioteca para Python: pyspark, wrapper para la interfaz PySpark (módulo de Spark). De forma que las operaciones se realicen en un *script* independiente, sin depender de la herramienta *Spark Shell*.
  - Recuperar desde Cassandra dos tablas en sendos *datasets*.
  - Unión Natural filtrando por una columna común.
  - Guardar el resultado de nuevo en Cassandra.
  - Transformar el resultado en un formato de intarcambio de datos, que sea serializable, como p.ej. JSON.

- **Entorno de desarrollo**

  Se ha creado un repositorio en Github con los pasos concretos a modo de guía, que pueda ser usado por cualquier desarrollador que quiera replicar los ejemplos [10].

  El primer paso para poder desarrollar los test, es la puesta a punto de un entorno de desarrollo adecuado con el que poder trabajar. Se necesita montar sobre el sistema operativo, en este caso Ubuntu (A), el motor de bases de datos de Cassandra, Apache Spark y el Conector Spark-Cassandra (2.5). Además, hay que instalar las dependencias [11]. Para los ejemplos se usará tanto Scala como Python, así como bibliotecas y utilidades.

- **Desarrollo de los ejemplos**. Para llevar a cabo esta primera iteración de pruebas, y que pudieran verse resultados tangibles, se ha cargado una serie de datos de prueba en varias tablas, siendo el modelo muy similar a cualquier modelo de datos de una base de datos relacional. Concretamente en el Ejemplo 1 de la batería de test, es en el que hemos realizado esta tarea, con dos *scripts* en Python (sin utilizar Spark).

  Se ha utilizado un herramienta llamada Mockaroo [34] para generar datos ficticios y poder trabajar con ellos en los ejemplos[4]. Se ha llevado a cabo mediante dos *scripts* en código Python:

  - Ejemplo 1
    - `mock_data`: Datos de ejemplo de personas, con datos ficticios[5].
    - `mock_cars`: Datos ficticios de coches cuyo dueño esta representado por un registro del conjunto `mock_data`[6].

[4]es una herramienta con un modelo de negocio ***freemium***, donde en la parte gratuita hay una limitación de 2K registros. En este TFG se han usado algunos conjuntos de datos de 4K registros, generando dos conjuntos de datos con dicha herramienta y uniéndolos.

[5]http://bit.ly/py-upload-data

[6]http://bit.ly/py-upload-cars

Hay varias formas de ejecutar código en Spark, una de ellas usando Spark Shell, que es una herramienta de línea de comandos, donde se van ingresando comandos en lenguaje Scala, siendo éstos interpretados uno a uno. En principio usamos esta vía para familiarizarnos con los comandos, la configuración del entorno, la carga de un RDD e ir probando transformaciones y acciones. Durante el desarrollo de la batería de test, se ha creado un repositorio de Github con una descripción detallada de dichos ejemplos, de cuyas entradas se pone la referencia:

- Ejemplo 2 [7].
- Ejemplo 3 [8].

Otra vía para la ejecución de *scripts* en Spark, es usando la herramienta spark-submit, compilando la aplicación en Scala mediante, por ejemplo, sbt ([11]). Por esta vía, además de los ejemplos 2 y 3, podemos ejecutar los siguientes ejemplos de la batería de test iniciales:

- Dataset Join 01 [4].
- Dataset Join 02 [5].

Por último, hemos explorado el uso de la biblioteca pyspark para Python, ya que podemos integrar el código de análisis de datos en cualquier programa Python. Esta opción es la que nos acerca mejor a los objetivos del TFG. Esta vía se ha desarrollado en el último test de la batería.

- Dataset Join 03 [6].

- **Conclusiones de los test**. Una vez concluidos los test, se llega a la conclusión de que se han alcanzado los objetivos iniciales para poder elaborar una herramienta basada en las tecnologías exploradas. En el test final se han ejecutado operaciones parecidas a lo que se quiere conseguir, usando un *script* en Python, que puede ser integrado en una aplicación que siga la arquitectura propuesta por este trabajo. Es decir, se concluye que se puede desarrollar un **microservicio** tipo REST que interactúe con Spark y Cassandra, usando la interfaz de alto nivel de Spark, llamada Dataset, para llegar a cumplir los objetivos del TFG. De esta forma podemos desacoplar la plataforma, y el lenguaje con el que se desarrolla, de la plataforma en la que se desarrolle la aplicación final (aplicación, ***middleware***, etc.).

4.3. Diseño

La etapa de diseño se ha completado con la suma de todas las etapas de diseño dentro del proceso iterativo: análisis, diseño e implementación. Se ha dividido en dos subsistemas, cada uno de ellos con un propósito diferenciado:

1. *Apollo*: Subsistema de consultas a Cassandra usando Spark. En la *[fig. 4.1]* podemos ver el diagrama UML de actividad.

2. *Daphne*: Subsistema de aplicación del Caso de Uso (3.2). En la *[fig. 4.2]* podemos ver el diagrama UML de actividad, y en la *[fig. 4.3]* puede verse el diagrama de actividad completo del sistema `Daphne-Apollo`.

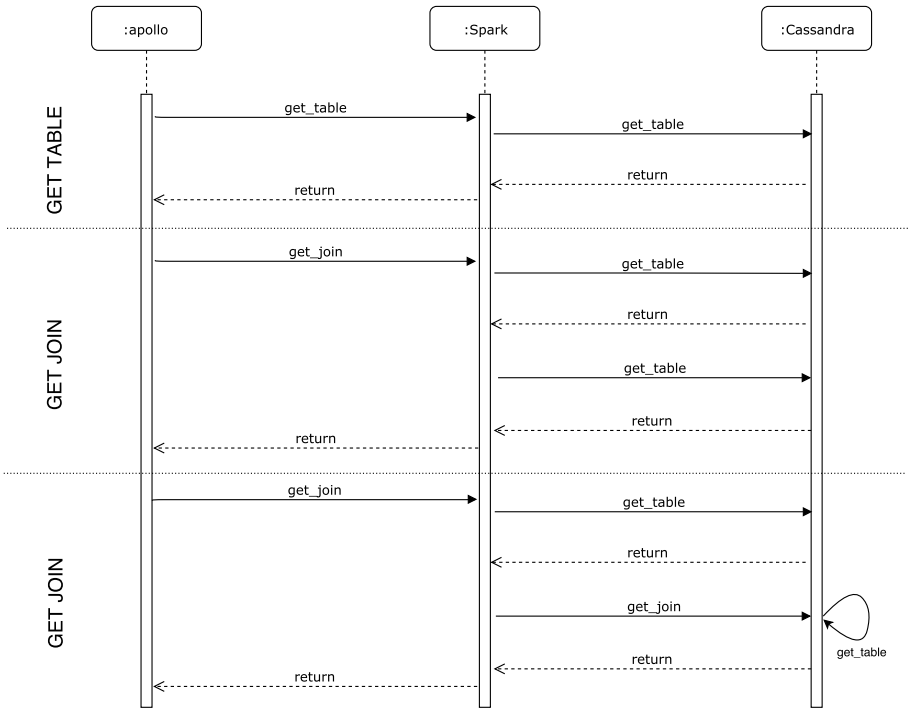

Figura 4.1: Diagrama de Actividad de *Apollo*.

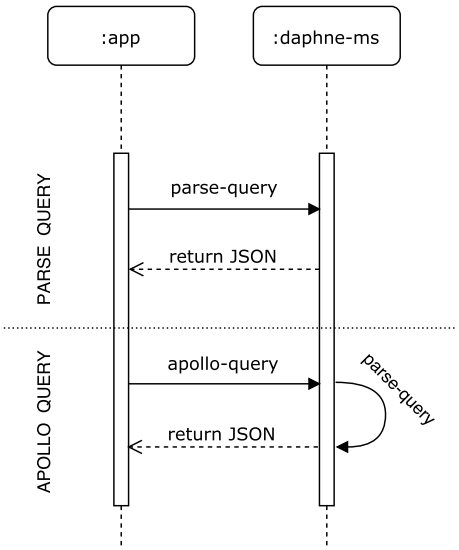

Figura 4.2: Diagrama de Actividad de *Daphne*.

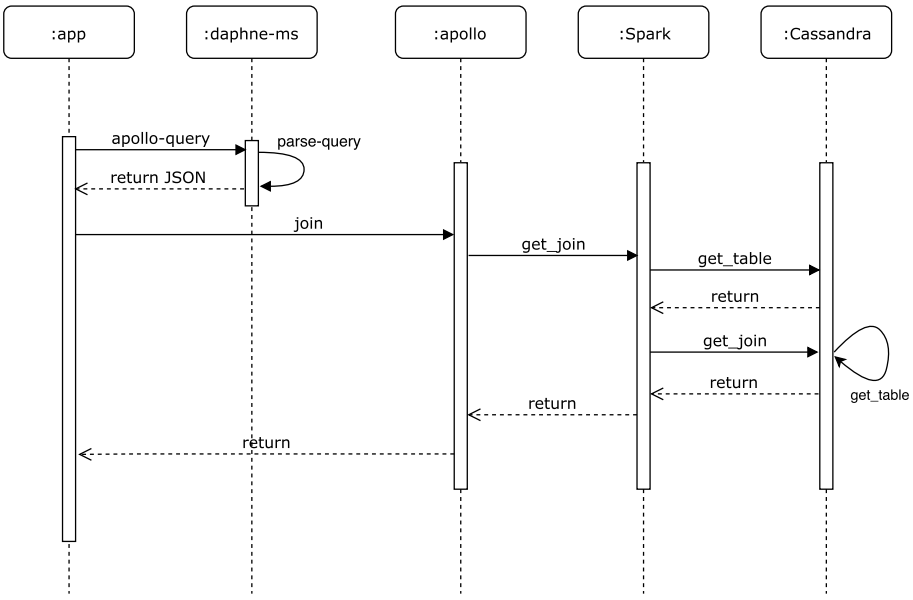

Figura 4.3: Diagrama de Actividad de *Daphne-Apollo*.

4.3.1. *Apollo*

La filosofía de diseño de *Apollo* partió de las siguientes características deseables:

- Accesible mediante *Hyper Text Transfer Protocol* [13] (HTTP). Toda la funcionalidad accesible mediante verbos HTTP: GET, POST.
- Datos de respuesta en formato JSON. Pensamos que el estándar JSON cubre todas las necesidades de la herramienta.
- Genérica, en cuando a desacoplada del caso de uso. Queremos una herramienta que se adecúe a múltiples casos de uso, por tanto, se establece como requerimiento de diseño, construir una herramienta lo más genérica posible.
- Flexible, en cuanto a la plataforma de programación o lenguaje que haga uso de la herramienta.
- Especializada en cruce de datos, similar a SQL: selección, proyección, renombrado, join, union, intersección. Es decir, no se desea una herramienta que realice operaciones diferentes al cometido para el que se diseña.

Características por las cuales se elige una arquitectura de **microservicio** REST, ya que sus características intrínsecas son compatibles al cien por cien de las características mencionadas.

4.3.2. *Daphne*

Es la herramienta que adapta *Apollo*[7] al caso de uso. Como se ha dicho en el punto anterior, *Apollo* es genérica, y por tanto necesita cierta adaptación al caso de uso. Esta herramienta que hemos llamado *Daphne*[8] tiene dos funcionalidades básicas:

1. Parsear una consulta en formato SPARQL a un formato JSON con las diferentes tripletas de la consulta. Dicho formato lo denominaremos `SPARQL-JSON`.

2. Transformar una consulta en formato `SPARQL-JSON` a un formato `Apollo Query`, también basado en JSON.

4.3.3. Flujo de Trabajo

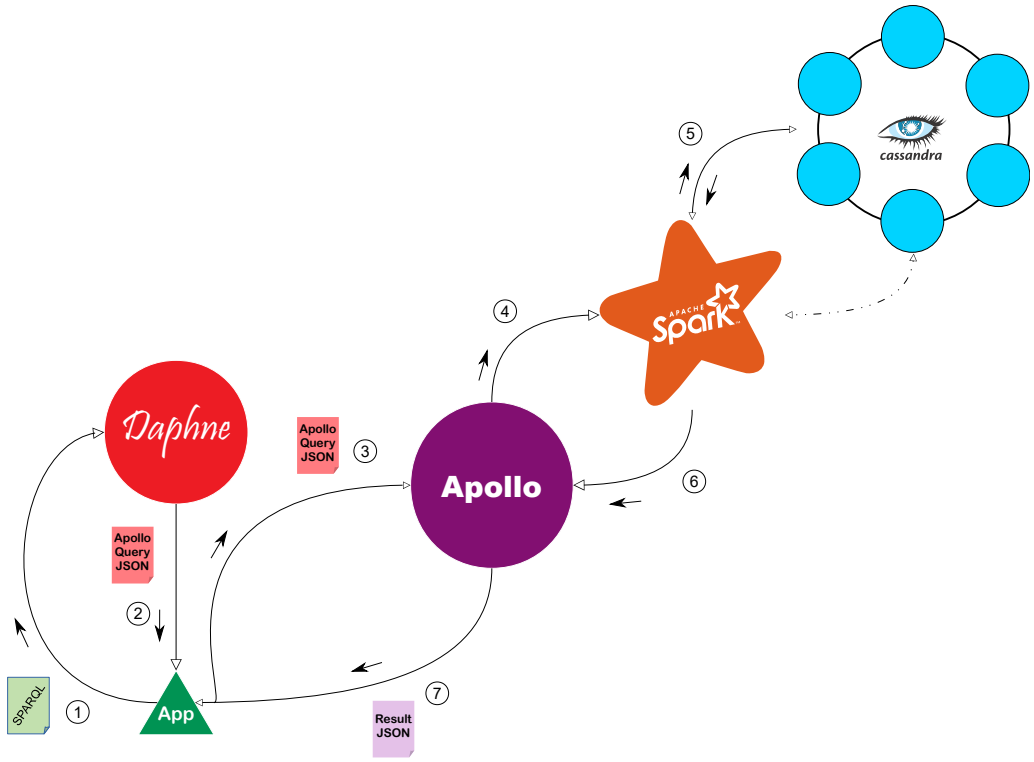

Figura 4.4: Esquema Sistema Daphne-Apollo.

Los puntos etiquetados en la *[fig. 4.4]* corresponden con los pasos ejecutados en el proceso completo desde que la aplicación cliente solicita una consulta en formato *SPARQL* hasta que obtiene los resultados en formato JSON, de forma transparente al usuario:

[7]se ha denominado así por el dios mitológico, cuyo destino se cruzó con el de la princesa Cassandra, a la cual dio el don de la clarividencia, pero siendo rechazada por ésta, la maldijo con que sus premoniciones no fueran creídas por nadie.

[8]según la mitología, fue una ninfa, cuyo fatídico destino también se cruzó con el de *Apollo*, y para evitar al dios, se transformó en un laurel.

1. La aplicación cliente, solicita una consulta SPARQL en texto plano.
2. El microservicio *Daphne*, devuelve la traducción de la consulta en formato `Apollo Query`[9], basado en formato JSON.
3. La aplicación cliente, hace la petición de consulta al **microservicio** *Apollo*.
4. *Apollo* solicita a Apache Spark los conjuntos de datos y los cruces necesarios para ejecutar la consulta.
5. Spark se comunica con Cassandra para obtener los datos.
6. Apollo recibe el dataset resultado de la consulta y convierte los registros a formato JSON.
7. El JSON con los resultados es devuelto a la aplicación cliente.

4.4. Implementación

La implementación de los dos subsistemas propuestos se ha llevado a cabo siguiendo la misma metodología usada en análisis y diseño, o mejor dicho, intercalada con el análisis y diseño de forma iterativa hasta llegar a los prototipos propuestos. La implementación de *Daphne* se inició una vez que existía un prototipo viable de *Apollo*, debido a su dependencia funcional.

Para la implementación de *Apollo* se ha usado el ***framework*** Flask/Python para la interfaz. La lógica de negocio se ha podido desarrollar gracias a la biblioteca pyspark para Python, permitiéndonos gestionar la API de Spark adecuadamente, según los requerimientos de la aplicación.

Adicionalmente se ha implementado un cliente Web, como ejemplo de implementación basada en `Apollo-Daphne`. Para ello, se ha utilizado el ***framework*** Angular 4 (A), basado en HTML/CSS/javascript (6.2.2).

A lo largo de todo el desarrollo se ha hecho un seguimiento de las diferentes versiones usando herramientas de control de versiones de código, también para la documentación, teniendo en cada momento control sobre el estado de los diferentes subsistemas *[fig. 4.5]*[10]. Los repositorios Git están accesibles tanto *online* como en el CD-ROM que acompaña a la memoria [12].

Los detalles de ambos prototipos se han organizado en los capítulos 5 y 6.

[9] Sería el formato de documento basado en JSON que entiende el **microservicio** *Apollo*.

[10] la imagen corresponde con una interfaz gráfica como cliente de Git denominada GitKraken (A)

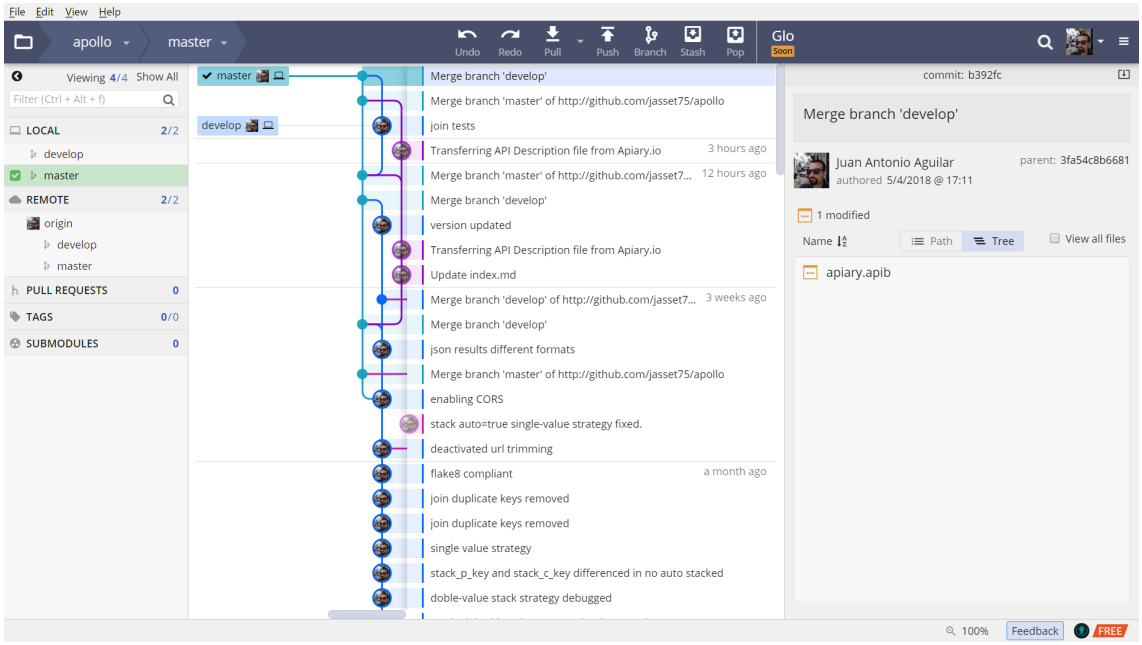

Figura 4.5: Control de versiones Git de *Apollo*.

4.5. Pruebas

4.5.1. Pruebas de *Apollo*

Para el desarrollo de las pruebas, se ha utilizado la herramienta Insomnia (A), que nos ha permitido configurar los ***endpoints***, definiendo los *Uniform Resource Identificator*s (URIs) correspondientes y los documentos JSON con la petición de datos a la interfaz de *Apollo*. Esta aplicación permite ejecutar verbos de peticiones HTTP con un interfaz gráfico bastante usable, y que nos ha permitido alcanzar buena productividad en un corto tiempo (A).

En [*Código* 4.1] podemos ver un ejemplo de cómo se invocaría a la interfaz de *Apollo* desde un *script* Python. En la ruta `$CD-ROM$/apollo/test/apollo.json`[11] se encuentra un fichero de configuración para un entorno de la herramienta Insomnia, donde se puede acceder a más pruebas, que han sido desarrolladas en el transcurso de la implementación de la aplicación.

4.5.2. Pruebas de *Daphne*

Para las pruebas del **microservicio** `daphne-ms` (4.3.2) también se ha utilizado la herramienta Insomnia; además, se han utilizado scripts en Python para comprobar tanto la utilidad práctica de la herramienta como para hacer pruebas programables y poder así depurar los resultados.

En `$CD-ROM$/daphne/test/daphne.json` tenemos las llamadas a la API y en la ruta `$CD-ROM$/archer/`[12] se pueden ver los test y ejemplos relacionados con el caso de uso (3.2),

[11] también disponible *online* [12]
[12] también disponible *online* [12]

```
import requests
import json

url = "http://localhost:5000/get-table"

payload = {
    "keyspace": "examples",
    "tablename": "mock_data",
    "filter": 'email like "%am%"',
    "sortby": [
        {"gender": "asc"},
        {"first_name": "asc"}
    ]
}

headers = {'content-type': 'application/json'}

response = requests.request("POST", url, data=json.dumps(payload),
                            headers=headers)
print(response.text)
```

Código 4.1: Script Python de test para *Apollo*

scripts de Python, a modo de cliente de los dos **microservicios**. Se amplía la información más adelante (6.2.2), pero usando Angular como cliente.

5

Microservicio *Apollo*

> Houston, Tranquility Base here. The Eagle has landed.
>
> Neil Armstrong

Este capítulo está dedicado exclusivamente a uno de los entregables de este TFG, y aunque de él se ha comentado en capítulos anteriores, aquí se recogen de forma agrupada los aspectos más técnicos.

Como ya se ha comentado, *Apollo* integra una API tipo REST para desacoplar la implementación del cliente, que puede estar implementada en cualquier plataforma, y cualquier lenguaje, siempre que pueda intercomunicarse utilizando el protocolo HTTP, usando verbos GET y POST, y utilizando JSON como formato de intercambio de datos, tanto en la llamada como en la interpretación de la respuesta.

```
// Apollo endpoint: /about
{
  "api_name": "Apollo",
  "author_email": "juanantonioaguilar@gmail.com",
  "author_name": "Juan A. Aguilar-Jimenez",
  "documentation": "http://jasset75.github.io/apollo",
  "license": "Apache 2.0",
  "project_repository": "https://github.com/jasset75/apollo.github",
  "version": 1.0
}
```

Código 5.1: Respuesta JSON: Acerca de *Apollo* API

5.1. Interfaz

La interfaz está descrita tanto en la documentación [2], donde podemos ver una guía de uso para el desarrollador, como a través del lenguaje de descripción API Blueprint en un fichero en el CD-ROM que acompaña a esta memoria, disponible también *online* [12].

Se añaden a partir de aquí algunos puntos que cabe destacar.

5.1.1. *Get Table*

`/get-table endpoint`

Es una función primitiva de la herramienta, ya que las operaciones de *join* y de *union* necesitan de las acciones a bajo nivel de esta operación. Permite realizar tres de las operaciones básicas del álgebra relacional (2.2): Proyección, Selección y Renombrado.

```
// Apollo endpoint: /get-table
{
  "keyspace": "examples",
  "tablename":"mock_data",
  "select":[{"id": "key"},"first_name","last_name","email","age",{"language":"mother_tongue
    "}],
  "calculated": {
    "age": "round(months_between(current_date(),birth_date)/12,0)"
  },
  "filter": "email like \"%elo%\"",
  "sortby": [
    {"gender": "asc"},
    {"first_name": "asc"}
  ],
  "orient_results": "records"
}
```

Código 5.2: Ejemplo de llamada: *Get Table, Apollo* API

En [*Código* 5.2] tenemos un ejemplo de cómo consultar registros con una proyección de determinadas columnas y renombrándolas en el resultado. También se utiliza un campo calculado en el que se usa una expresión de Spark SQL para calcular la edad en función de la fecha de nacimiento. Hay también una declaración de ordenación por los campos `gender` y `first_name`. Además, se realiza una selección de filas (filtro) en función de una subcadena de la columna `email`[1], concretamente se seleccionan todos los registros donde el *email* contenga la cadena `elo`. En [*Código* 5.3] podemos ver el resultado.

- ***Stacked*** . Otra opción clave en la operación *get table*, puede verse en [*Código* 5.4] y que permite pivotar de una estructura orientada a columna, a una orientada a filas. Dicha opción, es especialmente importante para el caso de uso (3.2), ya que en él se maneja una estructura de datos donde una clave está vinculada con un número indeterminado de columnas; es decir, un predicado semántico puede tener indefinidos sujetos e indefinidos predicados. En dicha estructura, sería muy complejo hacer un *join* al desconocerse el número exacto de columnas. Por tanto, se transforma a una estructura con un número variable de filas y fijo de columnas, en la cual no se necesita conocer el número exacto de filas en el momento de hacer los cruces de datos, dada la naturaleza intrínseca de la operación *join* (2.2.3). Es por ello que se hace la citada transformación, de *column-oriented* a *row-oriented*.

  Para poder hacer esa transformación, se añade el campo `rowid`, identificando así los valores de la 'misma fila'. Adicionalmente, podemos necesitar añadir un campo a la *clustering key* para identificar valores relacionados, p.e. `num`, que sería también, por tanto, parte de la *primary key*.

[1] usando los datos de ejemplo definidos en la documentación [10]

```
// Response 200 OK.
...
"data": [
  {
    "age": 31.0,
    "email": "rrupelon@google.pl",
    "first_name": "Ronna",
    "key": 887,
    "last_name": "Rupel",
    "mother_tongue": "Korean"
  },
  {
    "age": 48.0,
    "email": "crobelow3d@who.int",
    "first_name": "Conway",
    "key": 121,
    "last_name": "Robelow",
    "mother_tongue": "Tsonga"
  },
  {
    "age": 57.0,
    "email": "ncolgravelo@npr.org",
    "first_name": "Norby",
    "key": 780,
    "last_name": "Colgrave",
    "mother_tongue": "Tajik"
  }
],
...
```

Código 5.3: Ejemplo de resultados: *Get Table, Apollo* API

```
    // Apollo endpoint: /about
    // -- Stacked option
    ...
    "stacked": {
        "auto": false,
        "strategy": "double-value",
        "stack_p_key": ["key"],
        "stack_c_key": ["num"],
        "stack_pair": "rowid",
        "stack_column": "value",
        "filter_field": "num",
        "filter_left_value": 1,
        "filter_right_value": 2
    }
    ...
```

Código 5.4: *Get Table, Apollo* API, opción *stacked*.

Ejemplo de datos:

key	**num**	**column1**	**column2**	**column3**
firstName	1	Picasso	Rodin	Velazquez
firstName	2	Pablo	Auguste	Diego

Fields explained:

- `auto`, si es `false`, `stack_p_key` es obligatorio, además `stack_c_key` se necesitaría solamente para `"strategy": "single-value"`.
- `strategy`, dos posibilidades:
  - `single-value`: se hace la transformación de las columnas de datos a filas, y para identificar valores relacionados en una misma fila de la tabla original, se añade un campo `rowid` con un valor único, p.e. ***UUID***.

key	**rowid**	**num**	**value**
firstName	544ca336-2d9c-36bb-8433-17371498d2fe	1	Picasso
firstName	544ca336-2d9c-36bb-8433-17371498d2fe	2	Pablo
firstName	f1c25492-21b1-314a-8218-75da2d4e8fbd	1	Rodin
firstName	f1c25492-21b1-314a-8218-75da2d4e8fbd	2	Auguste
firstName	f1c25492-21b1-314a-8218-1608134e5815	1	Velazquez
firstName	f1c25492-21b1-314a-8218-1608134e5815	2	Diego

  - `double-value`: para asociar los dos valores de datos que forman un par. A cada valor de la misma columna se le asocia un código ***UUID*** para identificarlo unívocamente, y se cogen dos conjuntos de datos: el de la izquierda, filtrando la *clustering key* por el valor `filter_left_value`; el de la derecha, filtrando por el valor `filter_right_value`; se hace un *join* de los dos conjuntos de datos, tomando como criterio de comparación la *partition key* más el campo `rowid`.

key	**rowid**	**value1**	**value2**
firstName	544ca336-2d9c-36bb-8433-17371498d2fe	Picasso	Pablo
firstName	f1c25492-21b1-314a-8218-75da2d4e8fbd	Rodin	Auguste
firstName	f1c25492-21b1-314a-8218-1608134e5815	Velazquez	Diego

- `stack_p_key` es la lista de campos que forman parte de la *partition key*.
- `stack_c_key`, lista de campos de la *clustering key*, que junto a la *partition key* forman la *primary key* de la tabla.
- `stack_pair` es el nombre que se le da a un campo autogenerado, por defecto `rowid`, que identifica valores ubicados en la misma columna con la misma clave de la tabla original. Será también parte de la *clustering key* de la tabla.
- `filter_left_value` y `filter_right_value`, parámetros de configuración necesarios en un escenario de estrategia `double-value` donde se utilizan para hacer la transformación de columnas de datos a filas.

```
// Apollo endpoint: /create-table
{
    "keyspace": "examples_bis",
    "tablename": "new_table_2",
    "columns": [
        {
            "db_field": "key_1",
            "db_type": "Integer",
            "partition_key": "true"
        },
        {
            "db_field": "key_2",
            "db_type": "Integer",
            "partition_key": "true"
        },
        {
            "db_field": "first_name",
            "db_type": "Text",
            "primary_key": "true"
        },
        {
            "db_field": "last_name",
            "db_type": "Text",
            "primary_key": "true"
        },
        {
            "db_field": "age",
            "db_type": "Integer"
        }
    ]
}
```

Código 5.5: Ejemplo de llamada: *Create Table, Apollo* API

5.1.2. *Create Table*

`/create-table endpoint`

Aunque no es indispensable para este TFG, en el que las tablas de datos a analizar son pre-existentes. Sin embargo, interesa poder crear nuevas tablas donde guardar resultados obtenidos de otras operaciones y así poder usarlos como base en posteriores análisis.

En [*Código* 5.5] podemos ver un ejemplo de creación de una tabla, donde tenemos una *partition key* compuesta por dos campos, que junto a otros dos campos forman la *primary key*[2].

5.1.3. *Join*

`/join endpoint`

Esta operación, quizás la que da sentido a este proyecto ya que suple la carencia de Cassandra para cruce de datos, implementa la Unión Natural del álgebra relacional (2.2).

[2]en Apache Cassandra (2.3) el primer campo corresponde a la *partition key*, es decir, en qué nodo se almacenará el dato, pudiendo ser una *partition key* compuesta por varios campos. El resto de campos definidos como *primary key* componen la *clustering key* que define cómo se ordenan los registros dentro de la partición

```
// Apollo endpoint: /join
{
    //join interno
  "join_a": {

    "table_a": {
      "keyspace": "examples",
      "tablename": "mock_data",
      "join_key": ["id_people"],
      "select": [{"id": "id_people"},{"first_name": "name"},"last_name","email","gender","
    drinker"]
    },
    "table_b": {
      "keyspace": "examples",
      "tablename": "mock_cars",
      "join_key": ["id_owner"],
      "select": ["car_id"]
    },
    "join_key": ["id_people","car_id"],
    "join-type": "inner" //default
  },
  "table_b": {
    "keyspace": "examples",
    "tablename": "cars_owned_by_drinkers",
    "join_key": ["id",{"car_id": "car_id_2"}],
    "select": ["car_make","car_model"]
  },
  "select": ["id_people","name","car_id","car_make"],
  "join-type": "inner" //default
}
```

Código 5.6: *Join*, *Apollo* API

Es un operador binario donde los operandos pueden tener una estructura recursiva, de forma que desencadenará tantas operaciones como niveles de recursión se definan.

Los operandos pueden ser de dos tipos, una tabla o la descripción de una operación. Esta descripción admite una Unión Natural *(join)* o la Unión de conjuntos del álgebra relacional (2.2). De forma que existen nueve posibilidades:

Join
table_a ⋈ table_b
table_a ⋈ join_b
table_a ⋈ union_b
join_a ⋈ table_b
join_a ⋈ join_b
join_a ⋈ union_b
union_a ⋈ table_b
union_a ⋈ join_b
union_a ⋈ union_b

En el ejemplo, en [*Código* 5.6] podemos ver un join 'multiple', donde la descripción más externa tiene como parámetros `join_a` y `table_b`, lo que significa que se hará el *join* interno,

en el que se describen dos nuevas tablas, y el resultado se cruzará con `table_b` para generar el resultado final.

5.1.4. *Union*

`/union endpoint`

Esta operación implementa la unión descrita por el álgebra relacional (2.2), donde dados dos conjuntos de datos, el conjunto de datos resultado contiene las filas de ambos *[fig. 2.3]*.

La operación se describe con un objeto JSON, que consta de dos operadores. Cada uno de ellos puede ser una operador recursivo que requiera de la ejecución de otras operaciones básicas.

Unión		
table_a	∪	table_b
table_a	∪	join_b
table_a	∪	union_b
join_a	∪	table_b
join_a	∪	join_b
join_a	∪	union_b
union_a	∪	table_b
union_a	∪	join_b
union_a	∪	union_b

Además también admite parámetros para modificar el resultado o para indicar cuál es la `join_key`. Más información puede verse en la documentación en linea ([2]).

5.2. Detalles de Implementación

5.2.1. interfaz *bow*

Este módulo del proyecto es el que implementa la interfaz REST. Llamado así por el dios *Apollo* de la mitología griega, conocido también como "el arquero" o "el cazador".

Está implementado sobre Flask, un ***framework*** de Python que nos permite definir los ***endpoints***, bien sea GET o POST, en los que se recoge un documento JSON con los parámetros de la llamada. Una vez parseados dichos parámetros, se invoca a una serie de funciones internas y de bibliotecas propias que trabajan tanto con los datos como con las interfaces sobre Spark o contra Cassandra en algunos casos.

5.2.2. biblioteca *admix*

Se ha denominado así por ser una mezcla de funcionalidades, entre las cuales está la de obtener información sobre las propias estructuras de Cassandra (tablas, keyspaces, etc.), como para invocar las funciones del driver de Cassandra (2.3), o para crear nuevas tablas. La documentación sobre esta biblioteca puede consultarse *online* [3].

5.2.3. biblioteca *quiver*

Este es el *core* del proyecto. Permite interactuar con Apache Spark, haciendo posible la computación de los datos desde Cassandra. El nombre de *quiver*, del término que significa en castellano "carcaj", además de otras acepciones, que es el instrumento donde se guardan las flechas; haciendo una referencia, una vez más al dios Apolo, "el arquero". En (5.7) puede verse un trozo de código que realiza una función especialmente interesante: la parte en la que se transforma un DataFrame de Spark en un RDD para poder transformar, a su vez, las columnas de datos en filas. Usado en la implementación de la opción *stacked* de `get_table` y que es clave para realizar la unión natural entre conjuntos de datos con un número de columnas indeterminado (5.1.1). Posteriormente se vuelve a convertir en DataFrame para poder hacer *join* a alto nivel, con una sintaxis mucho más clara.

La documentación sobre esta biblioteca puede consultarse *online* [3].

```
def _map_stack(h_row, stack_p_key, primary_key):
    """
        Converts one row's column into new rows, keeping original keys in all rows,
        and adds new unique identifier in order to identify related columns
    """
    columns = {}
    # uuid seed
    stack_p_key_value = {}
    # common elements
    for idx, key in enumerate(primary_key):
        if key in stack_p_key:
            stack_p_key_value[key] = h_row[idx]
        columns[key] = _trim_str(h_row[idx])
    # stack elements
    return [
        Row(
            **columns,
            # quiver_pair is a hash value for elements (columns) of the same key and column
     position
            # from the original dataset, so it must be part of the key to join associated
    elements
            # to the previous dataset's key plus column index
            quiver_pair_=str(uuid3(NAMESPACE_URL, '{}_{}'.format(str(stack_p_key_value),
    indx))),
            # different column values
            quiver_column_= val
        ) for indx, val in enumerate(h_row[len(primary_key):])  # iterates over data
    columns
    ]
...
# stack main part
rdd = dataset.rdd.flatMap(
    lambda row: _map_stack(row, stack_p_key, primary_key)
)

# renames internal names to definitive names
df_stacked = _rename_column(
    _rename_column(
        spark.createDataFrame(rdd), _pair, stack_pair
    ), _column, stack_column
)
...
```

Código 5.7: Implementación de conversión de filas a columnas: *stack*

6

Aplicación al Caso de Uso

> With four parameters I can fit an elephant, and with five I can make him wiggle his trunk.
>
> John von Neumann

Para verificar la aplicación de la herramienta genérica de cruce de datos desde Apache Cassandra con Apache Spark, *Apollo*, se ha propuesto un Caso de Uso sobre una aplicación de la Web Semántica. Se ha diseñado una batería de test para comprobar la bondad de la solución implementada.

Cada uno de los test de la batería corresponden con una consulta SPARQL, que podría ser factible sobre un origen RDF, pero en este caso el repositorio donde reside la información que queremos consultar está en una base de datos Cassandra con un formato determinado (3.2).

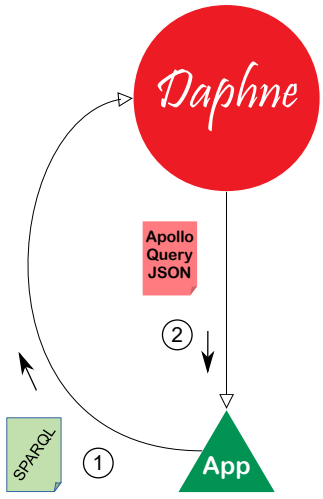

Figura 6.1: Esquema Daphne.

En *[fig. 6.1]* tenemos el esquema que sigue una aplicación para poder obtener una documento JSON con una consulta genérica en el formato de *Apollo* desde una consulta en formato SPARQL[1], y que en (4.3.2) denominamos formato `SPARQL-JSON`.

Una vez que tenemos la consulta en el formato que Apollo es capaz de entender, lanzamos la consulta contra el **microservicio** *Apollo* para obtener el documento JSON con el resultado final, como puede verse en el esquema *[fig. 6.2]*.

[1] en 4.3.3 se describe cual es el flujo de trabajo.

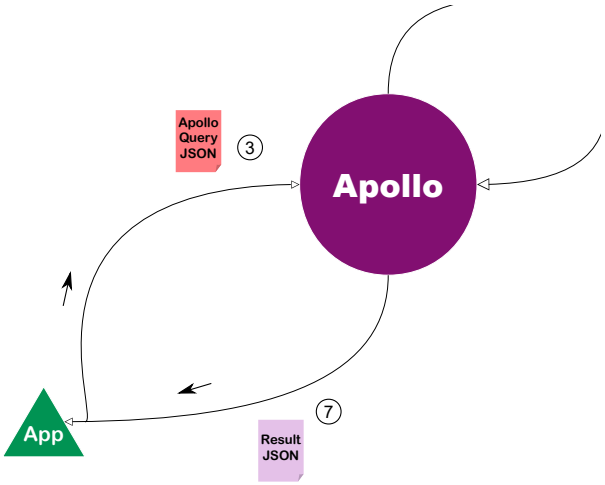

Figura 6.2: Esquema Apollo.

```
PREFIX art:<http://khaos.uma.es/Ontologies/artists.owl#>
PREFIX rdf:<http://khaos.uma.es/Ontologies/artist.owl/>
SELECT ?m WHERE {
    ?m rdf:type art:museum .
    ?m art:locatedIn art:madrid .
}
```

Código 6.1: Consulta SPARQL - Caso de Uso - Test 01

6.1. Batería de test sobre el caso de uso

6.1.1. Test 01

En [*Código* 6.1] tenemos una consulta que puede describirse en lenguaje natural de la siguiente forma:

`"Quiero saber los museos ?m localizados en Madrid"`

En *[fig. 6.2]* tenemos el documento JSON como entrada para `daphne-ms/apollo-query` y en [*Código* 6.3] tenemos la consulta obtenida (6.2.1).

6.1.2. Test 02

En [*Código* 6.3] tenemos una consulta que puede describirse en lenguaje natural de la siguiente forma:

`"Quiero saber los individuos ?p que pintaron un elemento ?st exhibido en cualquier ciudad ?c"`

En [*Código* 6.4] tenemos la consulta que obtenemos del **microservicio** *Daphne* (6.2.1).

```
{
  "query": "PREFIX art:<http://khaos.uma.es/Ontologies/artists.owl#> PREFIX rdf:<http://
    khaos.uma.es/Ontologies/artist.owl/> SELECT ?m WHERE { ?m rdf:type art:museum . ?m art:
    locatedIn art:madrid . }"
}
```

Código 6.2: Consulta SPARQL en formato `SPARQL-JSON` - Caso de Uso - Test 01

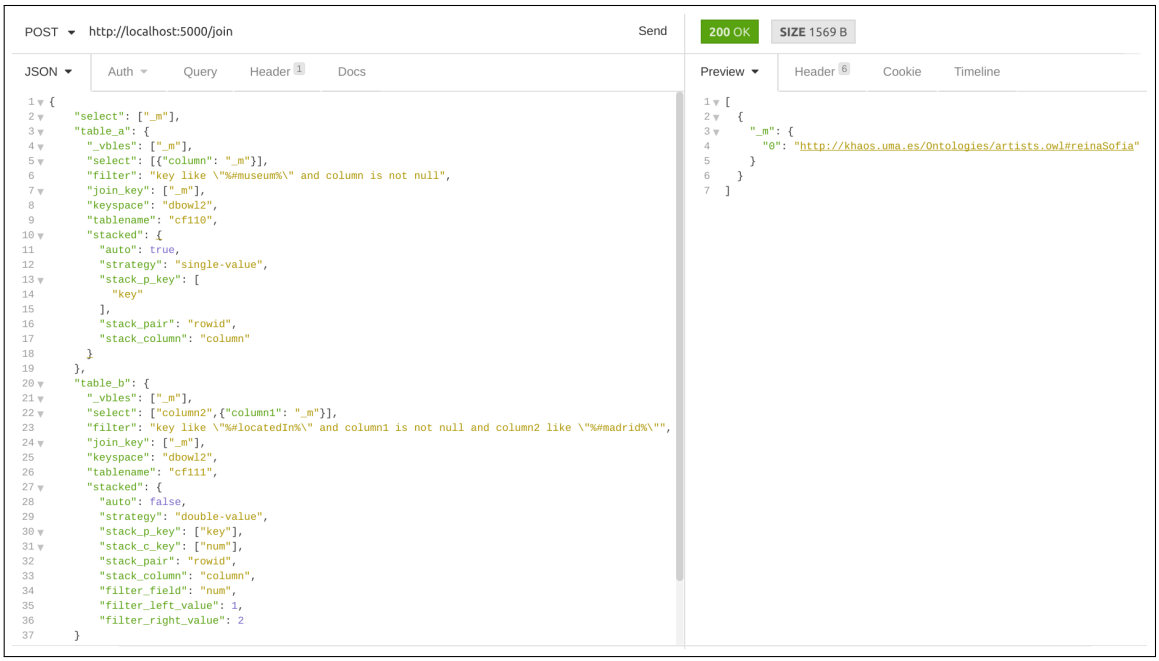

Figura 6.3: Daphne Caso de Uso - Test 01 - `Apollo Query`.

6.1.3. Test 03

En [*Código* 6.4] tenemos una consulta que puede describirse en lenguaje natural de la siguiente forma:

`"Quiero saber los museos ?m localizados en cualquier ciudad ?c"`

En [*Código* 6.5] tenemos la consulta que obtenemos del **microservicio** *Daphne* (6.2.1).

6.2. Detalles de implementación

6.2.1. Microservicio daphne-ms

Este **microservicio** específico para el caso de uso (3.2), está implementado usando Node Js, plataforma Javascript de programación de servidor. Para implementar los ***endpoints*** usamos el ***framework*** *express* (A).

Figura 6.4: Daphne Caso de Uso - Test 02 - `Apollo Query`.

```
PREFIX art:<http://khaos.uma.es/Ontologies/artists.owl#>
SELECT ?p ?c WHERE {
    ?p art:paints ?st .
    ?st art:exhibitedIn ?m .
    ?m art:locatedIn ?c .
}
```

Código 6.3: Consulta SPARQL - Caso de Uso - Test 02

6.2.1.1. Interfaz

Consta de dos ***endpoints***:

- `/parse-query`, esta funcionalidad traduce una consulta SPARQL representada en una cadena de caracteres, a un objeto JSON. Esto nos permitirá traducir al formato de *Apollo* con mayor facilidad. En [*Código* 6.5] tenemos el resultado de la llamada a `daphne-ms`.
- `/apollo-query`, este punto de entrada a la API engloba la funcionalidad de `/parse-query`, por tanto, parte de la misma entrada: una cadena de caracteres con una consulta SPARQL, obteniendo la consulta en formato `Apollo Query`, con la descripción de la consulta final contra Apache Cassandra y que el **microservicio** *Apollo* es capaz de entender y ejecutar.

```
PREFIX art:<http://khaos.uma.es/Ontologies/artists.owl#>
PREFIX rdf:<http://khaos.uma.es/Ontologies/artist.owl/>
SELECT ?m WHERE {
    ?m rdf:type art:museum .
    ?m art:locatedIn ?c .
    ?c rdf:type art:city .
}
```

Código 6.4: Consulta SPARQL - Caso de Uso - Test 03

```
[
  {
    "subject": "?m",
    "predicate": "http://khaos.uma.es/Ontologies/artist.owl/type",
    "object": "http://khaos.uma.es/Ontologies/artists.owl#museum"
  },
  {
    "subject": "?m",
    "predicate": "http://khaos.uma.es/Ontologies/artists.owl#locatedIn",
    "object": "http://khaos.uma.es/Ontologies/artists.owl#madrid"
  }
]
```

Código 6.5: Consulta SPARQL parseada - Caso de Uso - Test 01

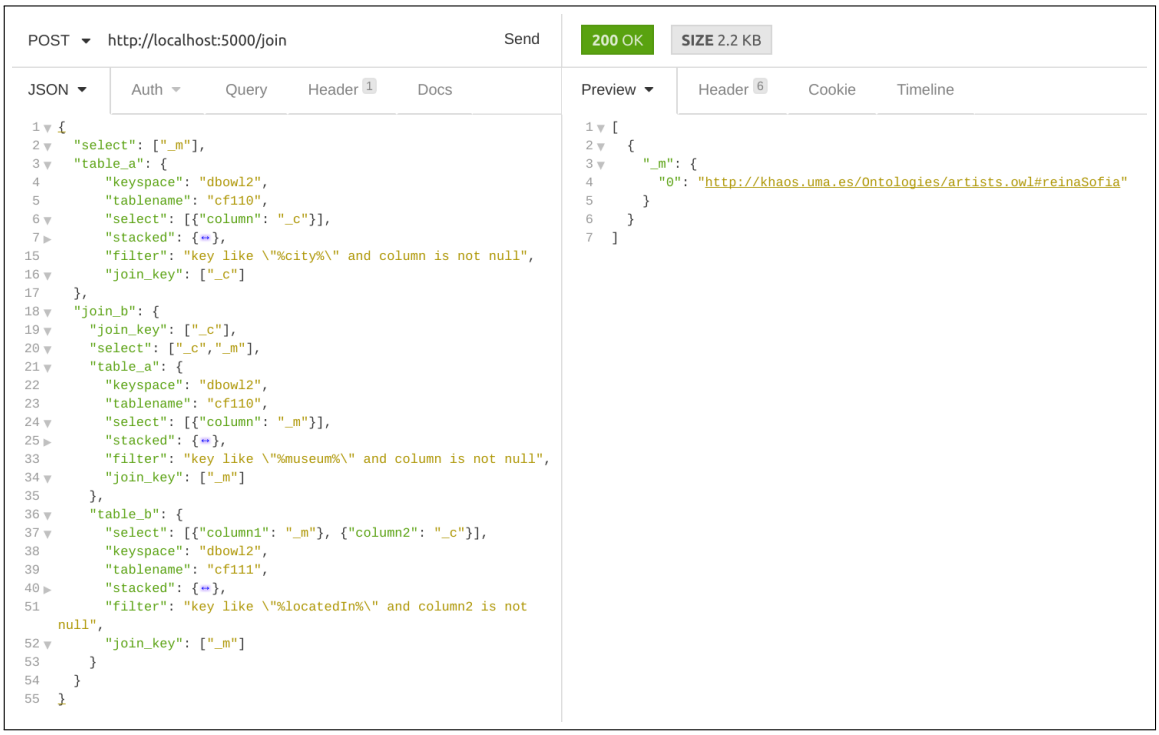

Figura 6.5: Daphne Caso de Uso - Test 03 - `Apollo Query`.

```
{
  "_apollo_query": {
    "select": [
      "_m"
    ],
    "table_a": {
      "_vbles": [
        "_m"
      ],
      "select": [
        {
          "column": "_m"
        }
      ],
      "filter": "key = \"http://khaos.uma.es/Ontologies/artists.owl#museum\" and column is not null",
      "join_key": [
        "_m"
      ],
      "orient_results": "records",
      "keyspace": "dbowl2",
      "tablename": "cf110",
      "stacked": {
        "auto": true,
        "strategy": "single-value",
        "stack_p_key": [
          "key"
        ],
        "stack_pair": "rowid",
        "stack_column": "column"
      }
    },
    "table_b": {
      "_vbles": [
        "_m"
      ],
```

```
      "select": [
        "column2",
        {
          "column1": "_m"
        }
      ],
      "filter": "key = \"http://khaos.uma.es/Ontologies/artists.owl#locatedIn\" and
    column1 is not null and column2 = \"http://khaos.uma.es/Ontologies/artists.owl#
    madrid\"",
      "join_key": [
        "_m"
      ],
      "orient_results": "records",
      "keyspace": "dbowl2",
      "tablename": "cf111",
      "stacked": {
        "auto": false,
        "strategy": "double-value",
        "stack_p_key": [
          "key"
        ],
        "stack_c_key": [
          "num"
        ],
        "stack_pair": "rowid",
        "stack_column": "column",
        "filter_field": "num",
        "filter_left_value": 1,
        "filter_right_value": 2
      }
    },
    "join_a": {},
    "join_b": {},
    "orient_results": "records",
    "_triples": [
      {
        "subject": "?m",
        "predicate": "http://khaos.uma.es/Ontologies/artist.owl/type",
        "object": "http://khaos.uma.es/Ontologies/artists.owl#museum"
      },
      {
        "subject": "?m",
        "predicate": "http://khaos.uma.es/Ontologies/artists.owl#locatedIn",
        "object": "http://khaos.uma.es/Ontologies/artists.owl#madrid"
      }
    ],
    "join_key": []
  },
  "_queryType": "select",
  "_apollo_url": "http://localhost:5000/join"
}
```


Código 6.6: Consulta `Apollo Query` - Caso de Uso - Test 01

El subsistema es uno de los productos entregables de este TFG, podemos encontrarlo en `$CD-ROM$/daphne/daphne-ms` , disponible también *online* [12]

6.2.2. Cliente daphne-web

En *[fig. 6.1]* tenemos como entidad genérica "App", que representa cualquier cliente o ***middleware*** que utilice al sistema *Apollo* para cruzar datos desde Cassandra. También se ha desarrollado una implementación de un cliente Web que permite ejecutar los ejemplos

Figura 6.6: Daphne Caso de Uso - Test 01 - Cliente Web Resultados.

de la batería de test (6.1) desde una interfaz gráfica. Hemos denominado `daphne-web` a la aplicación para distinguirla del **microservicio** `daphne-ms` (6.2.1).

Está implementada usando *Angular*, ***framework*** Javascript de gran productividad. La implementación realizada explota varios puntos fuertes de este ***framework***, que permite interactuar con APIs tipo REST (A). En particular desde `daphne-web` se invoca tanto a `daphne-ms` como a *Apollo* mediante sendos Servicios de Angular, destinados a interactuar con servicios remotos. En dicha implementación, se ha utilizado el patrón de diseño *Observable*, donde los servicios son observados por los componentes que consumen los datos de las respuestas de los Servicios.

En las figuras: *[fig. 6.6]*, *[fig. 6.7]* y *[fig. 6.8]* podemos ver el pantallazo de la ejecución cada unos de los test de la batería de test propuesta, lanzados desde `daphne-web`.

El subsistema es uno de los productos entregables de este TFG, podemos encontrarlo en `$CD-ROM$/daphne/daphne-web`, disponible también *online* [12]

Figura 6.7: Daphne Caso de Uso - Test 02 - Cliente Web Resultados.

Figura 6.8: Daphne Caso de Uso - Test 03 - Cliente Web Resultados.

7

Conclusiones y trabajos futuros

> If you put your mind to it, you could accomplish anything.
>
> Dr. Emmet Brown

El objetivo principal de este TFG de poder realizar operaciones tipo *join* para cruce de fuentes de datos desde Apache Cassandra (2.3) utilizando técnicas escalables, se ha cumplido. Para ello se ha utilizado una plataforma software horizontalmente escalable como es Apache Spark (2.4). Más allá de este objetivo, se ha conseguido crear una herramienta genérica, sin conocimiento previo del modelo de datos al que debe atacar, de forma que no se contamina la lógica implementada con la aplicación a un caso de uso concreto.

También se ha conseguido aplicar a un caso de uso concreto (3.2), de un problema real en el que se necesita un tipo de herramienta como ésta y que fue la base para la idea del TFG. Se ha realizado una adaptación de la herramienta genérica, *Apollo* (5), al caso de uso, que permite ejecutar consultas SPARQL. Adaptación hecha utilizando otra herramienta específica: `daphne-ms` (6.2.1), que nos permite transformar las consultas SPARQL a consultas en formato `Apollo Query` (5).

Además, se ha implementado una aplicación Web, accesible desde un navegador, que provee de una interfaz gráfica y que comunica con ambos **microservicios**, tanto el específico como el genérico, para obtener los resultados del cruce de datos desde la base de datos Apache Cassandra.

7.1. Entregables

Los productos entregables de este TFG, podemos encontrarlos en el CD-ROM que acompaña esta memoria, y también *online* [12]. Hallaremos los repositorios Git, por lo que se puede acceder a todo el código fuente, así como al histórico de versiones durante el proceso de desarrollo.

- `$CD-ROM$/apollo`
- `$CD-ROM$/daphne/daphne-ms`
- `$CD-ROM$/daphne/daphne-web`
- `$CD-ROM$/dbowl`[1]

[1] en esta carpeta podemos encontrar la base de datos del caso de uso de la Web semántica y un *script* para crear tanto la estructura de datos como los datos para poder reproducir los ejemplos.

7.2. Trabajos Futuros

En el desarrollo de este TFG se han sentado las bases, se ha desarrollado una herramienta escalable, pero más allá de las pruebas funcionales y de la prueba en un entorno de desarrollo con un único nodo para cada plataforma: Apache Spark y Cassandra, no se han realizado pruebas con grandes volúmenes de datos ni se ha hecho una prueba de estrés sobre un ***cluster***. Sería también interesante, realizar pruebas de rendimiento en un entorno de producción con varios nodos por ***cluster***, dando lugar a posibles mejoras.

En cuanto a la ontología materializada (3.1.7) sobre la que se ha implementado el caso de uso (3.2), es una ontología relativamente pequeña utilizada más a modo didáctico (3.1). Por ejemplo, hay una ontología desarrollada para facilitar la evaluación a modo de *benchmark* [1], y sería interesante poder hacer pruebas con la versión materializada de dicha ontología en una base de datos Cassandra. Dicha materialización excede del ámbito de este TFG y requiere de otras personas para poder realizarlo. Por ello, y por cuestiones de tiempo debido a que las fases deben estar acotadas a un calendario y a un número de créditos concreto, no se ha realizado a lo largo del desarrollo de este proyecto.

Apendice A

Apéndice – Tecnologías utilizadas

> If I have seen further, it is by standing upon the shoulders of giants
>
> Isaac Newton

A lo largo de este trabajo, se han utilizado múltiples herramientas, plataformas, lenguajes y ***frameworks*** de desarrollo en todas sus fases: tanto para el estudio previo, como para el proceso de desarrollo propiamente dicho, incluso como subsistemas integrados en la propia solución. Casi todas ellas de código abierto [41] o licencia gratuita. De las restantes, se ha usado la parte no comercial dentro de un licenciamiento ***shareware*** o dentro de un modelo de negocio ***freemium***.

- Editores de código
  - Sublime Text, es un editor de texto diseñado especialmente para código, capaz de resaltar la sintaxis de multitud de lenguajes tanto de programación como de marcado o codificados en diferentes formatos. Es extensible a través de *plugins* escritos en Python. Licencia ***shareware*** para demo por tiempo ilimitado. Se ha utilizado para editar el código Python y Node Js de este trabajo [32].
  - Visual Studio Code, editor de Microsoft, bajo licencia MIT. Se ha utilizado para el código Angular.
- Editores de texto
  - ShareLatex, editor *online* de Latex, con licencia ***freemium*** que permite una cuenta personal de uso gratuito. Se ha utilizado para elaborar la memoria.
- Editores de gráficos vectoriales
  - Inkscape, editor de gráficos en vectorial (SVG, PDF), que exporta tambíen a imágenes en formato PNG. Licencia GNU GPL. Se ha utilizado para algunas figuras de la memoria de este TFG.
  - Libreoffice Draw, editor de la suite ofimática de licencia gratuita. Se ha utilizado junto con inkscape, para componer algunos de los gráficos de la memoria y de la presentación.
  - draw.io, editor vectorial *online* con multitud de plantillas para tipos de gráficos estándar. Se ha usado para los diagramas UML tanto de la memoria como de la presentación.
- Lenguajes/Entornos de programación

- Scala, es un lenguaje funcional que ejecuta sobre la Java Virtual Machine (JVM), se ha usado para las primeras pruebas de análisis de este TFG.
- Python, además de un lenguaje interpretado de programación de propósito general, es también como se conoce al intérprete del lenguaje. Se ha usado para desarrollar tanto la herramienta principal del proyecto *Apollo* (5), como para hacer test de diseño y pruebas.
- Node Js, un entorno de ejecución de JavaScript orientado a eventos asíncronos. Se utiliza para scripts de servidor por su escalabilidad. Se ha usado para la herramienta `daphne-ms` (6.2.1).
- Javacript, usado directamente con Node Js, e indirectamente con Angular, ya que Typescript "transpila" a Javascript.

- Lenguajes de marcado
  - HTML, es el lenguaje base para la Web. Se ha utilizado embebido en el desarrollo de `daphne-web` (6.2.2), para construir las vistas de los componentes de Angular y la página de inicio.
  - Markdown, es un lenguaje ligero de marcado, que permite formatear textos usando una notación muy simple. Se ha utilizado para la documentación *online*, ya que plataformas como Github, permiten este tipo de marcado; para después, usando *Github Pages*, transformar de forma automática a HTML [2] [10]. También es parte del lenguaje de descripción de APIs `API Blueprint` para, precisamente, generar la parte de documentación.

- Lenguajes de Estilo
  - *Cascading Style Sheet* (CSS), es un lenguaje de estilos para presentación de documentos HTML. Utilizado en `daphne-web`

- Lenguajes de intercambio de datos
  - *YAML Ain't Markup Language [20]* (YAML), es un lenguaje de marcado, que se suele utilizar tanto para serialización, como para ficheros de configuración, descripción de APIs. En este trabajo se ha usado para configuración en *Apollo* (5) y *Daphne* (6).
  - JSON, es una notación para descripción de objetos Javascript, aunque hoy en día se usa en multitud de entornos ya que se ha instaurado como estándar *de facto* para el intercambio y serialización de información, sustituyendo en muchos casos a notaciones más formales, como es el XML. En este trabajo se a utilizado profusamente, tanto como formato de llamada/respuesta de *Apollo* (5) y *Daphne* (6), como internamente en *Daphne*, como para ficheros de configuración.

- Lenguajes de descripción de APIs

    - Blueprint API, es un lenguaje de descripción de APIs, que permite tanto documentar como diseñar las pruebas para que puedan ser ejecutadas por software de validación de APIs contra la propia API, como p.ej.: Dredd [33]. Se ha usado en este TFG tanto como documentación como para descripción de la API para pruebas automáticas.

- Lenguajes de consulta

    - CQL, lenguaje de consultas de Apache Cassandra. Se ha utilizado tanto en la fase de análisis y diseño como para realizar pruebas posteriores.
    - SPARQL, lenguaje de consulta para documentos RDF. Se ha utilizado en el cliente `daphne-web` (6.2.2), de forma que se puede consultar la base de datos Cassandra, ya que `daphne-ms` transforma de este lenguaje al formato de consulta de *Apollo* (5).

- Bibliotecas y drivers

    - cassandra-driver, biblioteca que permite consultar y gestionar la base de datos Cassandra desde código Python de forma programática.
    - Spark-Cassandra Connector [16], es una biblioteca de código abierto, bajo licencia Apache, que permite interactuar con Cassandra, lanzado consultas contra la base de datos y cargando en las estructuras de datos de Spark, como son los RDDs y Datasets (DataFrames en PySpark)
    - Flask-Cors, es una biblioteca de funciones que permite implementar el standard Cross-Origin Resource Sharing (CORS); los navegadores, por motivos de seguridad, impiden que las aplicaciones accedan a recursos localizados en otro dominio diferente si no se especifican una serie de encabezados HTTP para poder hacer uso de dichos recursos "cruzados". En el caso de este trabajo se ha utilizado porque tanto *Apollo* (5) como *Daphne* (6) pueden estar disponibles en diferente dominio, o IP, o en el caso del entorno de desarrollo en diferente puerto, por lo que requiere de esta implementación, que en el caso de Flask se puede hacer con la ayuda de esta biblioteca.
    - pandas, biblioteca Python para el análisis y manipulación de datos en forma de tabla. Es una biblioteca basada en estructuras de datos Numpy, trabaja en memoria y es muy flexible: puede indexar, filtrar, cruzar datos entre tablas, agrupar datos, exportar a varios formatos, y un largo etcétera de características [36]. En este trabajo se ha usado en *Apollo* (5) para transformar a formato JSON los datos desde Cassandra o desde Spark, manteniendo la estructura tabular.
    - pyspark, biblioteca de funciones para Python, que actúa como capa de acceso a Apache Spark. Se ha usado en *Apollo* para interactuar con Spark.
    - PyYAML, biblioteca de funciones para Python que permite manipular datos en notación YAML.

- sparqljs, biblioteca javascript cuya función principal es la de parsear consultas en lenguaje SPARQL y transformar las tripletas a formato JSON para que puedan ser procesadas en un algoritmo de forma programática.

- Gestores de paquetes/entornos
  - virtualenv, gestor de entornos virtuales para Python. Permite aislar entornos con diferentes bibliotecas, diferentes versiones de Python, sin que se interfieran unas a otras.
  - pip, gestor de paquetes para Python. Permite instalar bibliotecas desde repositorios de código *online*.
  - npm, gestor de paquetes para Node Js. Permite instalar bibliotecas desde repositorios de código *online*.

- ***frameworks***
  - Java OpenJDK, ***framework*** de desarrollo Java que permite ejecutar código compilado para la JVM.
  - Flask, ***framework*** para Python para creación rápida de aplicaciones Web. En este trabajo se ha usado para programar los ***endpoints*** de la API de *Apollo* (5).
  - express, ***framework*** para Node para creación rápida de aplicaciones Web. En este trabajo se ha usado para programar los ***endpoints*** de la API de `daphne-ms` (6.2.1).
  - Angular 4, ***framework*** Javascript del lado cliente, que ofrece multitud de reglas para la creación de clientes Web, y evitar los problemas de código ofuscado fortuitamente en el que se convierten muchos proyectos Javascript. El lenguaje usado es Typescript, que es un superconjunto de Javascript. A la hora de pasarlo a producción es transpilado a Javascript, debido a que los navegadores no "entienden" directamente Typescript. En este proyecto se ha usado para desarrollar el cliente `daphne-web` (6.2.2).

- Herramientas desarrollo Software
  - Insomnia, es una herramienta ***freemium*** para usarse como cliente API, permite validar los ***endpoints*** de la API y almacenar las peticiones; también permite extraer *snippets* de código de diferentes lenguajes con las peticiones definidas, lo que permite hacer pruebas rápidas con dichos lenguajes como clientes de la API testada.
  - apiari.io, es una herramienta *online* que permite describir una API usando varios lenguajes como API Blueprint o Swagger. Se integra con Github si queremos controlar las distintas versiones en dicha herramienta. También facilita el testeo de la API integrándose con Dredd, si la API está online. En el caso de este trabajo se ha usado por su integración con Github, pero el testeo se ha usado directamente con Dredd, ya que el entorno de desarrollo no está disponible *online*.

- Dredd. Es una utilidad de línea de comandos, usado para validación de APIs capaz de interpretar test desde una descripción hecha con varios lenguajes de descripción, como p.ej.: Blueprint API o Swagger, y ejecutar dichos test contra la propia API de forma automática. Se ha utilizado para el testeo de *Apollo* (5).
- Angular CLI, es una utilidad de línea de comandos que facilita el uso de Angular: creación de componentes, servicios, *pipes*; ejecuta el cliente en un servidor Web integrado para pruebas, etc. En este trabajo se ha usado para el desarrollo de `daphne-web` (6.2.2).

- Versión de código/documentación

  - Git, es un software de gestión de versiones multiplataforma. Git es distribuido y puede volcar información a un repositorio remoto, de forma que se pueda colaborar en un proyecto por un equipo de trabajo, donde cada uno de los miembros tendría un control de versiones local, que iría volcando al repositorio remoto, que a su vez puede volcar a otro remoto, y así sucesivamente hasta un repositorio origen. En este TFG se ha utilizado para controlar las versiones de todos los fuentes de las diferentes aplicaciones, teniendo como remoto tanto a Github como a Bitbucket.
  - Bitbucket, es una herramienta Web de control de versiones y documentación basada en *Git* y *Mercurial*. Ofrece servicios en un modelo de negocio [42]. Se ha utilizado, la parte *free*, tanto para mantener las diferentes versiones de *Daphne* (6).
  - Github, es una herramienta Web de control de versiones y documentación *online* basado en *Git*. Ofrece servicios en un modelo de negocio ***freemium***. Se ha utilizado, la parte *free*, para mantener las diferentes versiones de Apollo (5), como para la documentación *online* de este TFG, y pruebas iniciales.
  - GitKraken, cliente de Git con interfaz gráfica. Utilizado para el clonado de los repositorios en el CD-ROM que acompaña la memoria. Para el trabajo diario se ha utilizado el cliente git de línea de comandos, desde el terminal de Ubuntu.

- Sistemas Operativos

  - Ubuntu, es una distribución Linux, basada en Debian, en su mayoría bajo licencia GNU Not Unix General Public License (GNU GPL). El entorno de desarrollo se ha montado sobre una máquina virtual VMWare.

- Herramientas de Virtualización

  - VmWare Player, software de virtualización, gratuito para uso no comercial. Se ha utilizado para la máquina virtual con Sistema Operativo Ubuntu donde montar el entorno de desarrollo.

- Gestores Bases de de Datos y Analítica Big Data.

  - Apache Cassandra (2.3). Se ha utilizado como base para este trabajo.

    - Apache Spark (2.4). Se ha utilizado como base para este trabajo.

- Navegadores

    - Mozilla Firefox, es el navegador donde se han hecho las pruebas del cliente.
    - Google Chrome, se ha usado para depurar `daphne-ms` (6.2.1), ya que permite integrar un *plugin* de Node para la ejecución paso a paso, inspección de variables, etc.

- Almacenamiento en la nube

    - Dropbox, el contenido del CD-ROM también está disponible *online* gracias a esta herramienta ***freemium***, en la que pueden compartirse ficheros con una URL [12]. Dicha URL se ha acortado con la herramienta *online* Bit.ly, herramienta que también permite personalizar la URL con un nombre "amigable" en lugar de un identificador alfanumérico sin ninguna relación con el contenido. Esta herramienta es muy utilizada dentro del ámbito del *marketing online*.

Glosario

REST o **RESTful**, se refiere a un principio de arquitectura de software basado en hipermedia. Actualmente el término se ha generalizado a aplicaciones donde las acciones se desencadenan a través del protocolo HTTP. Entre las características de este principio tenemos: que las llamadas se realizan con todo el estado representado en la propia llamada y que cualquier punto de entrada definido en la interfaz es accesible a través de un URI. La tendencia actual de las aplicaciones que cumplen este principio, es que el formato de transferencia de los datos sea JSON. 21, g, k

SOA en castellano Arquitectura Orientada a Servicio, es un paradigma de desarrollo software que es una evolución del paradigma Cliente-Servidor, donde la parte del servidor se diseña mediante módulos independientes cuya funcionalidad sea reutilizada lo máximo posible, evitando duplicidad en la lógica de negocio. g, h

Teorema CAP de Brewer habla sobre tres características fundamentales de cualquier base de datos, y de cómo son mutuamente excluyentes dos a dos; es decir, ninguna puede cumplir las tres características, que son: Disponibilidad, Consistencia y Tolerancia a Fallos; cualquier base de datos que usemos, sea o no relacional, puede cumplir como máximo dos de ellas. El nombre del teorema, CAP, se debe a las siglas de dichas características en inglés: Consistency, Availability and Partition tolerance[38]. 6, 11

microservicio englobado dentro de una Arquitectura Service Oriented Architecture (SOA), un microservicio es una entidad software que realiza una función muy específica (p.ej: de un área de negocio de una compañía), de tamaño ligero y con la idea de realizar una o varias tareas necesitadas por una cantidad no determinada de aplicaciones. Al igual que un ***framework*** cumple el principio *Don't Reinvent Yourself* (DRY). Se habla también de Arquitectura de Microservicios para darle un nivel equivalente al de **SOA**, aunque haciendo hincapé en el caracter ligero de los microservicios. En la actualidad se asocian al concepto de **REST**, porque es común publicar una API REST para interactuar con las aplicaciones que demandan sus servicios. 3, 21, 27, 29, 31–33, 45–48, 52, 55

DBpedia es un proyecto abierto, donde la comunidad en torno al mismo ha extraído y generado información semántica de la wikipedia inglesa durante años, y el proyecto se ha extendido a 15 idiomas más hasta el momento [17]. 19

Khaos es un grupo de Investigación de la Universidad de Málaga, integrado principalmente por miembros del Departamento de Lenguajes y Ciencias de la Computación de dicha universidad. Sus líneas de investigación se centran en tecnologías de la Web Semántica. Principalmente: aplicaciones, integración de datos y bases de datos. En la actualidad están enfocados en ***middleware*** semántico. Estando entre sus principales objetivos la aplicación práctica de estas tecnologías [19]. 1, 23, 24

UUID Universally Unique Identifier, es un valor de 128 bits generado de forma pseudo-aletoria, que es usado para identificar de forma única elementos en un contexto. Dado el número de valores posibles: $3,4 * 10^{38}$, la probabilidad de colisión de dos valores es realmente baja y puede decirse que es virtualmente único. 38

big data es un término bastante amplio y de hecho, da para una gran cantidad de datos acerca del mismo, valga la redundancia; pero podríamos resumirlo diciendo que se refiere a la integración de diversas fuentes de datos con un Volumen, Variedad (heterogeneidad) y Velocidad de generación tal, que representa un problema computacional almacenar, clasificar e interpretar dichos datos. La complejidad del problema hace que no pueda ser abarcado con bases de datos tradicionales. Estas características intrínsecas de la composición del Big Data han acuñado el término como "las tres V's del Big Data", a las que algunas definiciones añaden 2 más, para describir la utilidad práctica del Big Data, que serían: Veracidad y Valor; es decir, que la información puede ser verificada como cierta y que añade valor a las empresas, gobiernos o a la sociedad en general que hacen uso de estas tecnologías de la información [CNBG+16]. 2, 10, 12

cluster en castellano se podría traducir como racimo o conglomerado, es un grupo de ordenadores que se conectan en red para hacer trabajos compartidos o dar un servicio distribuido. Normalmente los ordenadores que forman un *cluster* son ordenadores comunes, aunque pueden ser servidores, pero no tiene por qué ser un modelo homogéneo, por lo que en general tiene un menor coste que un gran ordenador que obtenga la misma capacidad de cálculo; además, permite reutilizar equipos que son obsoletos de forma individual, pero gracias a esta técnica pueden alargar su vida útil. 10–12, 56

endpoint o punto de entrada a un servicio, es la dirección que se debe conocer de un servicio, proceso o cola de servicio, para poder tener acceso al mismo. En el caso de una API Web se trata de un URI que como recurso tendría el acceso a una función de esa API. Es un término acuñado para describir arquitecturas **SOA**. d, 19, 32, 41, 47, 48

framework o también Marco de Trabajo en castellano, se refiere a un conjunto de reglas estandarizadas para resolver problemas con un común denominador de forma similar. En el desarrollo software esta definición se puede extender diciendo que es el conjunto de reglas, estructuras, estilos, etc., basados en un conjunto de módulos y bibliotecas software usados en base a dichas reglas para la resolución de un tipo de problema computacional. El problema a resolver, puede ir desde una página Web al desarrollo de un compilador, o desde la programación de un dispositivo móvil hasta la programación de un robot, o cualquier otro propósito que pueda ser unificado. Otra característica común a los diferentes *frameworks* es que el flujo de control suele ser responsabilidad del propio marco de trabajo, que define ciertos puntos en el flujo de ejecución donde añadir la lógica particular de cada problema. d, 2, 11, 12, a, 31, g, 41, 47, 52

freemium es un modelo de negocio de servicios, normalmente *online*, en el que se ofrecen ciertas características del mismo, de forma gratuita, dejando las características *premium* para los usuarios que pagan por estas características extra [42]. d, a, 26, e, f

hash es un valor de longitud fija que se obtiene aplicando una función sobre un conjunto de datos de entrada, de forma que se puede establecer una relación unívoca entre un dato y valor hash, también denominado resumen, ya que un valor de la imagen M de una función hash tiene un una longitud fija independientemente del tamaño del valor del dominio U. Dos valores diferentes de U pueden tener una misma correspondencia en M, en este caso se habla de colisión. Mientras que U puede contener un número infinito de elementos, la cardinalidad de M es finita. La función que obtiene el valor *hash*, se denomina función *hash*. 10

linked data en castellano datos enlazados, trata de redes de datos a nivel mundial, que interconectan fuentes de información, son accesibles mediante URIs, compuestas de datos estructurados o semi-estructurados en formatos tales como RDF, JSON, *Comma-Separated Values* (CSV), XML, YAML, etc. A menudo los datos enlazados son también ***open data***, pero también pueden ser datos corporativos o de grupos de usuarios que no permiten el acceso al público en general. 5

middleware capa software que se sitúa entre la capa de aplicación y las capas inferiores. La función de estas capas, es la de abstraer o simplificar el diálogo con dichas capas inferiores. Tales como drivers, sistemas operativos, APIs, etc. Dando pues, una solución de más alto nivel y más cercana a la capa de aplicación. 21, 27, g, 51

open data en castellano datos abiertos, son datos publicados tanto por administraciones públicas, dentro de una filosofía denominada Open Government [9], como por entidades privadas, incluso particulares, que ceden los derechos sobre los datos que publican para que sean usados por terceros en beneficio de la sociedad en general. i

schemaless sin esquema, se refiere a estructuras de datos cuyo esquema está implícito en los datos, al contrario de lo que ocurre en las bases de datos relacionales donde las tablas tienen un esquema explícito. Estas estructuras son válidas tanto para datos estructurados como para semi-estructurados, donde desde la misma fuente de datos pueden variar los datos con el tiempo, bien añadiendo un campo o cambiando el nombre de un atributo; y la estructura de datos "sin esquema" se adaptaría al cambio sin que fueran rechazados [27]. 5

shareware es un tipo de licencia de distribución de software, por la cual se da uso para evaluar por tiempo limitado o indefinido un producto software, bien completa, o bien con ciertas restricciones o solo para un subconjunto de características. a

sprint en terminología Scrum, un sprint es un bloque temporal en el que se divide el desarrollo software (normalmente con una duración de dos semanas a un mes) tras el cual en una reunión de sincronización del equipo se debe constatar un incremento del producto, que sea potencialmente entregable; es decir, funcional y en la medida de lo posible, con la lógica de negocio propuesta en la lista de tareas de la anterior reunión de sincronización [37]. 23, 25

Acrónimos

ABox *Assertional Box*. 18–20, 24

ACID *Atomicity, Consistency, Isolation and Durability*. 5

API *Application Programming Interface*. c, d, 11, b, e, 31, 32, h, 35, 48, 52

BASE *Basically Available, Soft state and Eventual Consistency*. 5, 11

CORS Cross-Origin Resource Sharing. c

CQL *Cassandra Query Language [26]*. c, 1, 10, 24, n

CRC Cyclic Redundancy Check. 11

CSS *Cascading Style Sheet*. b

CSV *Comma-Separated Values*. i

DL Lógica de Descripciones. 16, 18

DRY *Don't Reinvent Yourself*. g

GNU GPL GNU Not Unix General Public License. e

HTML *Hyper Text Markup Language*. 5, 15, b

HTTP *Hyper Text Transfer Protocol* [13]. c, 29, g, 35

JAR *Java ARchive*. 13

JSON *Javascript Object Notation [39]*. i, c, d, 3, 5, 26, b, 29–31, g, 35, 41, 45, 46, 48

JVM Java Virtual Machine. d, b

NoSQL *Not Only SQL* 2.1. 1, 2, 5, 6, 10, 19, 20

OWL *Web Ontology Language [40]*. 1, 16, 19

RDD *Resilient Distributed Dataset*. c, 12, 13, 25, 27, 42

RDF *Resource Description Framework*. i, c, 15, 16, 18, 19, 45

REST *Representational State Transfer* ***REST***. 21, 27, 29, 35, 41, 52

SPARQL *SPARQL Protocol and RDF Query Language*. c, d, 2, 3, 16, 18, 19, 30, 31, 45, 48, 55

SQL *Structured Query Language*. 3, 6, 8–10, 18, 21, 24, 29

TBox *Terminological Box*. 18–20

TFG Trabajo Fin de Grado. c, 1–3, 6, 23, 24, a, 27, e, 35, 39, 51, 53, 55, 56

URI *Uniform Resource Identificator*. 32, h

W3C *World Wide Web Consortium*. 15

WORM *Write Once Read Many*. 5

WWW *World Wide Web*. 2, 15

XML *eXtensible Markup Language*. i, 5, 15, 16, b

YAML *YAML Ain't Markup Language [20]*. i, c, b

Referencias

[1] Swat projects - the lehigh university benchmark (lubm). [http://swat.cse.lehigh.edu/projects/lubm/ — accedido: 2018-04-19].

[2] Juan A. Aguilar-Jiménez. Apollo documentation. [https://jasset75.github.io/apollo/ — accedido: 2018-04-04].

[3] Juan A. Aguilar-Jiménez. Apollo repository. [https://github.com/jasset75/apollo — accedido: 2018-04-04].

[4] Juan A. Aguilar-Jiménez. Dataset join 01 example. [https://jasset75.github.io/Spark-Cassandra-Notes/Examples/dataset-join-01.html — accedido: 2018-03-22].

[5] Juan A. Aguilar-Jiménez. Dataset join 02 example. [https://jasset75.github.io/Spark-Cassandra-Notes/Examples/dataset-join-02.html — accedido: 2018-03-22].

[6] Juan A. Aguilar-Jiménez. Dataset join 03 example. [https://jasset75.github.io/Spark-Cassandra-Notes/Examples/dataset-join-03.html — accedido: 2018-03-24].

[7] Juan A. Aguilar-Jiménez. Mock data. [https://jasset75.github.io/Spark-Cassandra-Notes/Examples/mock-example.html — accedido: 2018-03-22].

[8] Juan A. Aguilar-Jiménez. Mock data save. [https://jasset75.github.io/Spark-Cassandra-Notes/Examples/mock-example-save.html — accedido: 2018-03-22].

[9] Juan A. Aguilar-Jiménez. Open government como factor de la innovación. [https://www.linkedin.com/pulse/open-goverment-como-factor-de-la-innovaci%C3%B3n-juan-antonio-aguilar/ — accedido: 2018-03-17].

[10] Juan A. Aguilar-Jiménez. Setting up the environment. [https://jasset75.github.io/Spark-Cassandra-Notes/ — accedido: 2018-03-26].

[11] Juan A. Aguilar-Jiménez. Setting up the environment. [https://jasset75.github.io/Spark-Cassandra-Notes/Environment.html — accedido: 2018-03-21].

[12] Juan A. Aguilar-Jiménez. TFG CD-ROM. [http://bit.ly/tfg-cdrom — accedido: 2018-06-10]].

[13] Dan Connolly. Hypertext transfer protocol – http/1.1. [https://www.w3.org/Protocols/rfc2616/rfc2616.html — accedido: 2018-03-16].

[14] Datastax. Cql limits. [https://docs.datastax.com/en/cql/3.3/cql/cql_reference/refLimits.html — accedido: 2018-03-18].

[15] Datastax. Datastax academy. [https://academy.datastax.com/ — accedido: 2018-03-13].

[16] Datastax. Spark-cassandra connector. [https://github.com/datastax/spark-cassandra-connector — accedido: 2018-03-18].

[17] dbpedia.org. DBpedia. [http://wiki.dbpedia.org/ — accedido: 2018-04-03]].

[18] dbpedia.org. Virtuoso SPARQL Query Editor. [http://wiki.dbpedia.org/ — accedido: 2018-04-03]].

[19] Grupo de Investigación Khaos. Universidad de Málaga. Grupo de Investigación Khaos. [http://khaos.uma.es/es — accedido: 2018-03-10].

[20] Clark C. Evans. YAML Ain't Markup Language. [http://yaml.org/ — accedido: 2018-03-12].

[21] Apache Software Foundation. Apache Cassandra. [http://cassandra.apache.org/ — accedido: 2018-03-10].

[22] Apache Software Foundation. Apache cassandra documentation. [http://cassandra.apache.org/doc/latest/ — accedido: 2018-03-13].

[23] Apache Software Foundation. Apache hadoop: Mapreduce. [http://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html#Purpose — accedido: 2018-03-13].

[24] Apache Software Foundation. Spark programming guide. [https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html#rdd-operations — accedido: 2018-03-18].

[25] Apache Software Foundation. Spark sql, dataframes and datasets guide. [https://spark.apache.org/docs/latest/sql-programming-guide.html — accedido: 2018-03-13].

[26] Apache Software Foundation. The Cassandra Query Language CQL. [http://cassandra.apache.org/doc/latest/cql/ — accedido: 2018-03-10].

[27] Martin Fowler. Schemaless data structures. [https://martinfowler.com/articles/schemaless — accedido: 2018-03-17].

[28] Google. Angular docs. [https://angular.io/docs — accedido: 2018-03-13].

[29] Stack Exchange Inc. Ask ubuntu. [https://askubuntu.com/ — accedido: 2018-03-13].

[30] Stack Exchange Inc. Ask ubuntu. [http://flask.pocoo.org/ — accedido: 2018-03-13].

[31] María Jesús Lamarca Lapuente. Hipertexto – ontologías. [http://www.hipertexto.info/documentos/ontologias.htm — accedido: 2018-04-13].

[32] Sublime HQ Pty Ltd. Sublime text editor. [https://www.sublimetext.com/ — accedido: 2018-03-18].

[33] Adrián Matellanes. Como construir un API de la que tus padres se sientan orgullosos. [https://speakerdeck.com/amatellanes/como-construir-un-api-del-que-tus-padres-se-sientan-orgullosos — accedido: 2018-03-13].

[34] Mockaroo. Random data generator. [https://mockaroo.com/ — accedido: 2018-03-22].

[35] Mozilla. Control de acceso http (cors). [https://developer.mozilla.org/es/docs/Web/HTTP/Access_control_CORS — accedido: 2018-03-24].

[36] pandas. Python data analysis library. [https://pandas.pydata.org/ — accedido: 2018-03-18].

[37] proyectosagiles.org. Ejecución de la iteración (sprint). [https://proyectosagiles.org/ejecucion-iteracion-sprint/ — accedido: 2018-03-21].

[38] rubenfa. Nosql: clasificación de las bases de datos según el teorema cap. [https://www.genbetadev.com/bases-de-datos/nosql-clasificacion-de-las-bases-de-datos-segun-el-teorema-cap — accedido: 2018-03-17].

[39] Ed T. Bray. RFC 8259: Javascript Object Notation. [https://www.rfc-editor.org/info/rfc8259 — accedido: 2018-03-12].

[40] W3C. Sitio Web del W3C — *Web Ontology Language*. [https://www.w3.org/OWL/ — accedido: 2018-03-10].

[41] Wikipedia. Código Abierto — Wikipedia, The Free Encyclopedia. [https://es.wikipedia.org/wiki/C%C3%B3digo_abierto — accedido: 2018-03-12].

[42] Wikipedia. Freemium — Wikipedia, The Free Encyclopedia. [https://es.wikipedia.org/wiki/Freemium — accedido: 2018-03-12].

[43] Wikipedia. NoSQL — Wikipedia, The Free Encyclopedia. [https://es.wikipedia.org/wiki/NoSQL — accedido: 2018-03-10].

[44] Wikipedia. Álgebra Relacional — Wikipedia, The Free Encyclopedia. [https://es.wikipedia.org/wiki/%C3%81lgebra_relacional — accedido: 2018-03-20].

Bibliografía

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001.

[CNBG+16] José A. Cordero, Antonio J. Nebro, Cristóbal Barba-González, Juan J. Durillo, José García-Nieto, Ismael Navas-Delgado, and José F. Aldana-Montes. Dynamic Multi-Objective Optimization with jMetal and Spark: A Case Study. In Panos M. Pardalos, Piero Conca, Giovanni Giuffrida, and Giuseppe Nicosia, editors, *Machine Learning, Optimization, and Big Data*, pages 106–117. Springer International Publishing, 2016.

[Cod72] E. F. Codd. Relational completeness of data base sublanguages. In *Database Systems*, pages 65–98. Prentice-Hall, 1972.

[DSS93] Randall Davis, Howard E. Shrobe, and Peter Szolovits. What is a knowledge representation? *AI Magazine*, 14(1):17–33, 1993.

[KKTC12] Vaibhav Khadilkar, Murat Kantarcioglu, Bhavani Thuraisingham, and Paolo Castagna. Jena-hbase: A distributed, scalable and efficient rdf triple store. Technical report, 2012.

[LM14] Jesús Luque Muñoz. *Un razonador OWL sobre una base de datos NoSQL.*, Diciembre 2014.

[RARGAM17] Liudmila Reyes-Álvarez, María del Mar Roldán-García, and José F. Aldana-Montes. A tool for materializing owl ontologies in a column-oriented database. 2017.