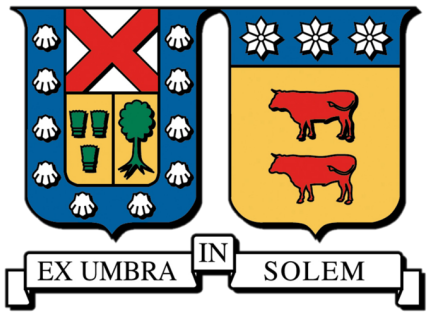
UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
VALPARAÍSO - CHILE



PREDICCIÓN DE RENDIMIENTO EN CONSULTAS SPARQL CON DEEP NEURAL NETWORKS

DANIEL ARTURO CASALS AMAT

MEMORIA PARA OPTAR AL TÍTULO DE
MAGÍSTER EN CIENCIAS DE LA INGENIERÍA INFORMÁTICA

PROFESOR GUÍA	:	CARLOS BUIL A.
PROFESOR CO-GUÍA	:	CARLOS VALLE V.
PROFESOR CORREFERENTE INTERNO	:	..
PROFESOR CORREFERENTE EXTERNO	:	..

DICIEMBRE 2020

RESUMEN

Las tecnologías de las Web Semántica están cambiando las formas en la que se comparte la información, sustituyendo los grandes volúmenes de información en formato HTML por datasets en los que el dato en bruto es tratado como un “ciudadano de primera clase”. Este nuevo enfoque busca persuadir a las organizaciones, empresas e individuos a que publiquen sus datos libremente siguiendo los estándares propuestos por la W3C y enlazando diferentes áreas del conocimiento generando la llamada Web de los Datos Enlazados. El público objetivo para consumir estos datos incluye tanto personas como aplicaciones de software.

En los últimos años,la aplicaciones de software han incrementado las capacidades de extraer información útil de estos grandes volúmenes de datos estructurados utilizando lenguajes como SPARQL que es el estándar para consultar datos RDF y se ha implementado en una amplia variedad de motores. Estos motores brindan el acceso a los datos a través de endpoints públicos en la Web, los cuales reciben miles de consultas diariamente. En muchos casos, estos endpoints enfrentan dificultades al evaluar consultas complejas o cuando reciben demasiadas al mismo tiempo. Esto provoca que los tiempos de respuesta percibidos por los clientes que ejecutan las consultas se vean afectados, sobre todo porque algunas de estas consultas necesitan grandes cantidades de recursos para ser procesadas. Todos estos motores tienen un optimizador de consultas interno que propone un plan de ejecución de consultas supuestamente óptimo, sin embargo, esta es una tarea difícil ya que puede haber miles de posibles planes de consulta a considerar y el optimizador puede no elegir el mejor.

En dependencia de los recursos computacionales disponibles es posible implementar también arquitecturas más complejas como réplicas y balances de carga, o incluso nuevos conceptos “Self-Driving Database Management Systems”.

Sin embargo, todos estos mecanismos dependen de buenos estimadores de la latencia de ejecución de las consultas. Hasta donde sabemos, en general, los estimadores de latencia para consultas SPARQL se basan en heurísticas sobre información estadística de las bases de datos. Otros técnicas como el uso de Support Vector Machine han mejorado las predicciones.

En esta propuesta se utilizan redes neuronales profundas para la creación de un estimador de latencias de consultas SPARQL que supera los resultados obtenidos en técnicas anteriores y que puede servir como base de apoyo para la construcción de técnicas de

optimización más avanzadas.

El estimador fue evaluado en bases de datos sintéticas y reales. Los resultados muestran que el desempeño de redes neuronales profundas supera las propuestas anteriores en el contexto de la predicción de latencia en consultas SPARQL.

Palabras Clave— Predicción latencia; Consultas Sparql; redes neuronales.

ABSTRACT

The technologies of the Semantic Web are changing how information is shared. The publication of large volumes of information in HTML format is being replaced by ways in which raw data is treated as a “first-class citizen”. This new approach seeks to persuade organizations, companies, and individuals to freely publish their data following the standards proposed by the W3C and linking different areas of knowledge. Today data from government, companies, communities, scientific publications, and many others are available, generating the so-called Web of Linked Data. Another motivation for this change is that the target audience for consuming this data no longer only includes people, but also software applications. The growth of these in recent years have increased the capacities to extract useful information from these large volumes of structured data.

The SPARQL query language is the standard for querying RDF data and has been implemented in a wide variety of engines. These engines provide access to data through public endpoints on the Web which receive thousands of queries daily. In many cases, these endpoints face difficulties when evaluating complex queries or when they receive too many at the same time. This causes that the response times perceived by the clients that execute the queries are affected, especially because some of these queries require large amounts of resources to be processed. All of these engines have an internal query optimizer that proposes a supposedly optimal query execution plan, however, this is a difficult task as there may be thousands of possible query plans to consider and the optimizer may not choose the best one.

Depending on the available computational resources, it is also possible to implement more complex architectures such as replicas and load balancers, or even new “Self-Driving Database Management Systems” concepts. However, all of these mechanisms rely on good estimators of query execution latency. As far as we know, in general, the latency estimators for SPARQL queries are based on heuristics on statistical information from the databases. Other techniques such as using the Support Vector Machine have improved predictions. In this proposal, deep neural networks are used to create a SPARQL query latency estimator that exceeds the results obtained in previous techniques and that can serve as a support base for the construction of more advanced optimization techniques.

The estimator was evaluated in synthetic and real databases. The results show that the performance of deep neural networks outperform the previous proposals in the context of

latency prediction in SPARQL queries.

Keywords— Latency prediction; Sparql Queries; neural networks.

Índice general

Índice de figuras

Índice de tablas

Índice de algoritmos

Capítulo 1

Introducción

La información disponible libremente en la Web sigue experimentando un incremento masivo sin precedentes. Entre los factores que influyen en esta tendencia se encuentran: la facilidad de generación de datos mediante dispositivos inteligentes (móviles, sensores, el internet de las cosas); la disminución del costo de procesamiento de estos grandes volúmenes de información; el incremento de las capacidades de almacenamiento, servicios en la nube y la rapidez de acceso a datos, entre otros [1, 2].

Este crecimiento ha exigido cambios en las formas en que se publican estos grandes volúmenes de información. La información publicada en formato HTML está siendo sustituida por formas en las que el dato en bruto debe ser tratado como el 'ciudadano de primera clase'. Con este nuevo enfoque se busca persuadir a las organizaciones, empresas e individuos a que publiquen sus datos libremente, siguiendo estándares y enlazando diferentes áreas del conocimiento para generar la 'Web de los Datos Enlazados'. La mayoría de estos datos se encuentran accesibles desde páginas web, gracias a muchas iniciativas que van desde el Schema.org [3] (por Google, Bing, Yahoo y Yandex), hasta el conjunto de tecnologías de la Web Semántica del World Wide Web Consortium (W3C). Esta nueva estructura posibilita la generación de información que pueda ser consumida tanto por humanos como por algoritmos informáticos. El objetivo de las Tecnologías de la Web Semántica es precisamente construir una infraestructura semántica para los datos en la Web, que sea fácilmente entendible por los computadores, para ello el W3C crea el Resource Description Framework (RDF [4]): el modelo de metadatos que sirve como base de dicha infraestructura y el lenguaje de consulta SPARQL [5]. La adopción del modelo de datos enlazados o 'graph data', necesita una gestión adecuada de los datos y

los sistemas de gestión de bases de datos (DBMS) tradicionales han tenido que adaptarse. Nuevos motores de bases de datos orientados a grafos (GDBMS), capaces de gestionar estos grandes volúmenes de datos enlazados han surgido. Tanto las soluciones comerciales de Oracle, Amazon Neptune o Neo4J, como las soluciones de código abierto: Blazegraph [1] (en su versión open source), Virtuoso Open Source Edition [2] y Apache Jena Fuseki [3], constituyen ejemplos de estos motores. En los últimos años la Web Semántica se ha centrado principalmente en la creación y explotación de grandes grafos de conocimientos (Knowledge Graph, KG). Una amplia lista de bases de datos públicas que incluye su dominio y las relaciones entre ellas puede ser encontrada en el proyecto "The Linked Open Data Cloud' [9]. Los KG privados también se han convertido en piezas fundamentales para las organizaciones: el KG de Google, el Facebook Graph Search y el Microsoft Semantic Graph constituyen algunos de los más conocidos. Entre las principales motivaciones para la creación de estos grandes volúmenes de datos enlazados se encuentra el creciente interés para extraer información implícita mediante los algoritmos de aprendizaje de máquinas. En la compilación de la 18va edición de International Semantic Web Conference [10], se aprecia como las técnicas de aprendizaje profundo se están aplicando hoy en tareas como la predicción de relaciones, el completamiento de grafos, Question Answering, el entrenamiento de embeddings para tareas de similaridad, entre otras. Sin embargo, también ha crecido el interés por mejorar las capacidades de los sistemas de bases de datos en general utilizando técnicas de aprendizaje de máquinas.

Una de las áreas más activas de investigación está centrada en la mejora de los optimizadores de consultas utilizando redes neuronales [11–13]. Los motores de bases de datos cuentan con un optimizador de consultas interno que propone un plan de ejecución de consultas (QEP) con cierto nivel de optimización. La selección del plan de consulta es una tarea difícil ya que puede haber miles de planes de consulta posibles a considerar [14]. El desarrollo de un buen planificador de consultas se considera el 'Santo Grial' en los sistemas de gestión de bases de datos. Los DBMS utilizan la estimación de cardinalidad como entrada para el modelo de costos que elige la alternativa más barata. Sin embargo, los estimadores de cardinalidad generalmente asumen simplificaciones de los datos (por ejemplo: principio de inclusión, suposiciones de uniformidad o independencia) las

[1] Almacén de triples Blazegraph: https://blazegraph.com/.
[2] Almacén de triples Virtuoso Open Source Edition: http://vos.openlinksw.com/owiki/wiki/VOS.
[3] Almacén de triples Apache Jena Fuseki versión 2: https://jena.apache.org/documentation/fuseki2/.

cuales frecuentemente no se cumplen, guiando a planes sub-óptimos y en algunos casos desastrosos [14].

Como se señala en [15], aún hay mucho margen de mejora en la optimización de consultas SPARQL. Resultados recientes muestran cómo modelos de redes neuronales pueden sustituir los estimadores de cardinalidad convencionales [16–19]. En DQ [20] y ReJOIN [21] se usan técnicas de aprendizaje reforzado (Reinforcement Learning) con modelos de costos tradicionales diseñados por humanos para aprender automáticamente planes de consulta optimizados.

Sin embargo, los más recientes enfoques abordan la estimación de latencia en lugar de las cardinalidades en un enfoque más natural y que obtiene mejores resultados [13, 22]. En Neo [13], se utiliza un modelo profundo que aprovecha operaciones de convoluciones [23] sobre árboles para aprender de los QEP parciales del proceso de optimización. A su vez en [22], se propone un estimador de latencia que utiliza unidades neuronales (pequeñas redes neuronales densas) que se interconenctan para aprender de los operadores lógicos de los QEP.

Como se indica en [24] “la latencia es la métrica más importante en un DBMS, ya que captura todos los aspectos del rendimiento”. La estimación de rendimiento de consultas es también de interés para la asignación de cargas de trabajo en sistemas más complejos como los servicios de bases de datos en la nube y el creciente interés en la implementación de conceptos como “Self-Driving Database Management Systems” [24]. En general, la latencia se utiliza como métrica para cumplir objetivos de Quality-of-Service [25, 26].

1.1. Identificación del problema

A diferencia del área de DBMS, la predicción de rendimiento en el área de bases de datos de grafos no ha sido muy abordada. Para la predicción de consultas en bases de datos de grafos, los últimos resultados vienen dados por el uso de técnicas de aprendizaje de máquinas como SVRs o algoritmos de clusterización como k-nearest neighbor(K-nn) [27]. Hay dos desafíos en la predicción de rendimiento en consultas SPARQL utilizando modelos de aprendizaje automático. El primero es encontrar la representación de consulta más adecuada en un formato que pueda ser interpretado por los algoritmos de Machine Learning (ML). El segundo (que depende del primer desafío), es el algoritmo de aprendizaje a utilizar para predecir el rendimiento.

La representación utilizada en [27] para consultas SPARQL tiene un conjunto de deficiencias identificadas. La principal es que rescatan solo información de las estructuras de las consultas como luego se analizará en detalle. No incluyen información como los predicados utilizados, joins entre predicados, el orden de ejecución entre los triples de la consulta y cardinalidades. La ausencia de información semántica de la representación utilizada afecta la calidad de los datos utilizados para el entrenamiento. Esta afectación se traduce en que para consultas con iguales representaciones se obtienen tiempos de ejecución diferentes lo cual dificulta la generalización de los modelos de aprendizaje utilizados.

Las diferencias entre los RDBMS y las bases de datos de grafos introduce dificultades adicionales que imposibilitan la aplicación directa de estas investigaciones. Los RDBMS están orientados al almacenamiento de datos altamente estructurados, cuyas relaciones se establecen entre algunos atributos de tablas y se encuentran bien definidas en un esquema rígido por naturaleza. Los estimadores de latencia en sistemas RDBMS aprovechan esta característica para codificar información de los planes de ejecución como el orden de las operaciones de join. Las bases de datos de grafos en cambio, están diseñadas para flexibilizar las relaciones entre los datos y en algunos casos no es obligatorio respectar el esquema u ontología definida. Dicha flexibilidad obliga a la definición de técnicas de extracción de características específicas para estas.

Hasta donde sabemos no existe ninguna solución que utilice técnicas de Deep Learning para la predicción de latencia de consultas SPARQL sobre bases de datos de grafos. Finalmente, la brecha que existe en la predicción de rendimiento dificulta el avance en cuanto a la optimización de consultas SPARQL y demás aplicaciones antes mencionadas.

1.2. Solución propuesta

Las similitudes del proceso de optimización de una consulta tanto en los sistemas de bases de datos de grafos como de los RDBMS sustentan la adopción y aplicación de enfoques similares para la tarea de predicción del tiempo de ejecución en consultas SPARQL. Teniendo en cuenta la problemática planteada y los avances descritos en ambas áreas se propone lo siguiente.

Caracterización de las consultas

La propuesta incluye el uso características que definen información semántica de la estructura de árbol relativa al plan de ejecución de las consultas. También se incluyen

otras técnicas de extracción de características más generales de las consultas para lo cual se evaluará la factibilidad del uso de características de: patrones de grafos, características algebraicas de las consultas, codificación de correlaciones de joins y estadísticas de predicados.

Modelo

Siendo coherentes con las características anteriores, proponemos una arquitectura de red neuronal profunda para predecir el tiempo de ejecución en consultas SPARQL. Esta recibirá características generales de las consultas SPARQL que serán procesadas con capas densas (fully-connected layers) apiladas. Los vectores de salida de las capas anteriores servirán para enriquecer las características de cada nodo de la estructura de árbol obtenida de los planes de consultas. Seguidamente se utilizarán capas apiladas de convoluciones sobre árboles según se propone en Tree Convolutional Based Neural Network (TBCNN), para aprender patrones de los planes de ejecución en una operación que tiene como salida otra estructura de árbol. Luego se aplicará una operación para convertir esa estructura en un vector fijo que pasará por algunas capas densas añadidas para mapear dicho vector a un valor numérico que corresponde al tiempo de ejecución de la consulta.

1.3. Hipótesis

La extracción de características que codifican la estructura del árbol de los planes de ejecución de consultas SPARQL; conjuntamente con la aplicación de arquitecturas de redes neuronales profundas que aprovechan técnicas de convoluciones sobre árboles, mejora la precisión de las predicciones del tiempo de ejecución.

1.4. Objetivos

El objetivo general de esta investigación es: Proponer técnicas de extracción de características y aplicar arquitecturas de redes neuronales profundas para mejorar la predicción del tiempo de ejecución en consultas SPARQL.

1.4.1. Objetivos específicos

Para cumplir con el objetivo general se plantean los siguientes objetivos específicos:

1. Identificar las bases de datos de grafos reales o sintéticas a utilizar así como generar un conjunto de datos provenientes de consultas SPARQL que permitan el entrenamiento y evaluación del modelo.

2. Definir e implementar una estrategia adecuada para la extracción de características de las consultas SPARQL que puedan ser utilizadas como entrada para el modelo de redes neuronales.

3. Definir e implementar el modelo de redes neuronales que reciba como entrada las características extraídas de consultas SPARQL para predecir el tiempo de ejecución.

4. Evaluar el desempeño del modelo propuesto con respecto a los existentes utilizando métricas acordes a problemas de regresión.

Capítulo 2

Estado del arte

El análisis del problema y la definición de la solución propuesta requiere conocimientos previos de algunas de las tecnologías de la Web Semántica y los modelos aprendizaje automático. De estos últimos principalmente, es necesario identificar modelos de aprendizaje profundo para tareas de predicción de valores continuos. En esta sección se introducen brevemente dichos tópicos.

En la sección 2.1 se describe qué es la Web Semántica con mayor profundidad, se explican algunas de las tecnologías más importantes que la componen (RDF y SPARQL). También se mencionan algunos otros componentes de más alto nivel como pueden ser las bases de datos y los motores que permiten la interacción con estos. La sección 2.2 introduce brevemente los algoritmos de aprendizaje automáticos. Por último, en la sección 2.3 se presentan variados trabajos que abordan la predicción del rendimiento en consultas SPARQL y SQL.

2.1. Web Semántica

La Web Semántica es un conjunto de actividades propuestas por la *World Wide Web Consortium* (desde ahora W3C) con el objetivo de generar tecnologías para la publicación de datos en la Web, de tal manera que sean procesables por las máquinas. Se basa en la idea de añadir metadatos semánticos y ontológicos para describir el contenido y las relaciones entre los datos publicados. Con esto se logra mejorar la interoperatividad de internet, pues los programas podrán acceder a la información en un lenguaje que comparte el mismo vocabulario y modelo de datos, para así procesar su contenido, razonar en base a éste y

combinarlo para resolver problemas cotidianos automáticamente.

Para lograr los objetivos de la Web Semántica, se han generado múltiples estándares de representación, entre ellos XML, XML Schema, RDF, RDF Schema, OWL y SPARQL. Estas tecnologías se utilizan actualmente para generar datos enlazados (*Linked Data*), que buscan enlazar información arbitraria en la web generando así una "*red de las cosas del mundo, descrita por los datos en la Web*" [28].

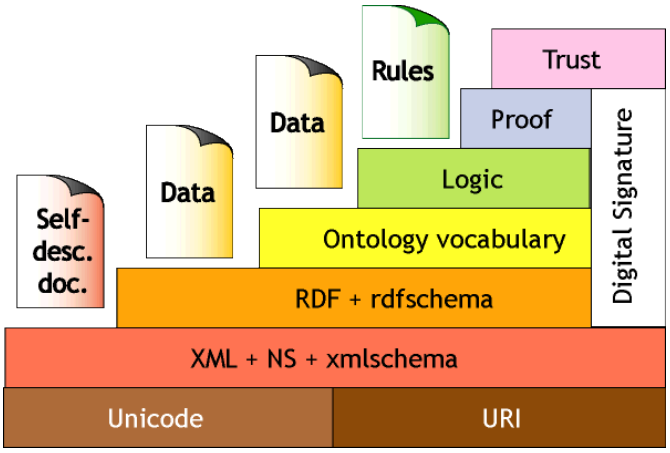


Figura 2.1: Pila de tecnologías de la Web Semántica. En las siguientes secciones se describirán más detalladamente algunos de los conceptos presentados en la Figura 2.1 comenzando de manera ascendente con URI, RDF y el lenguaje de consulta SPARQL.

De *Semantic Web - XML2000* - Tim Berners-Lee [29].

2.1.1. URI

Una URI (del las siglas del ingles *Uniform Resource Identifier*) es una cadena de caracteres utilizada para identificar un recurso inequívocamente. El estándar inicial fue extendido el 2005 a IRI (*Internationalized Resource Identifier*) que puede contener caracteres Unicode/ISO 10646, incluyendo chino, japonés, koreano, entre otros [30]. La sintaxis actual de una URI fue definida por Berners-Lee et al. el 2005 [31].

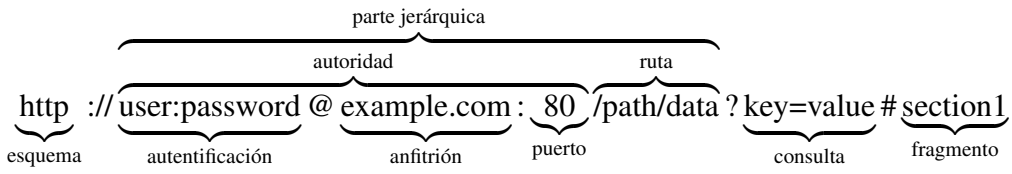


Figura 2.2: Ejemplo de una URI.

La Figura 2.2 es un ejemplo de URL (*Uniform Resource Locator*). La diferencia entre URL y URI es que la primera, además de identificar un recurso Web permite obtener una representación del mismo generalmente en formato HTML vía HTTP. En cambio la URI no necesariamente debe apuntar a una localización que exista realmente. En este documento no se diferenciará entre URI e IRI.

2.1.2. RDF

RDF (del inglés *Resource Description Framework*) es una familia de especificaciones del W3C diseñada como un modelo de datos para metadatos. Fue adoptado como una recomendación del W3C en 1999, mientras que la especificación 1.0 fue publicada el 2004 y la 1.1 el 2014 [32].

El modelo de datos RDF se basa en la declaración de relaciones o características entre y sobre recursos Web (URIs), utilizando expresiones $\langle sujeto, predicado, objeto \rangle$ que son llamados triples RDF. El $sujeto$ indica el recurso mientras que el $predicado$ denota la relación de éste con el $objeto$. Así podemos decir que un triple RDF sigue la clásica notación `entidad` - `atributo` - `valor` de los modelos orientados a objetos, permitiendo además, gracias a su simpleza, modelar todo tipo de conceptos abstractos.

Digamos U un conjunto de URIs, B un conjunto de recursos anónimos y L un conjunto de literales XSD, podemos denotar un triple RDF como:

$$\langle s, p, o \rangle \in (U \cup B) \times U \times (U \cup B \cup L).$$

Además, un conjunto de triples RDF será representado naturalmente por un grafo dirigido[1] (con s y o como nodos y p como el arco que los une). Esta característica faculta la tecnología para ser parte fundamental de la Web semántica, pues permite relacionar información de diferentes fuentes sin mayor problema y representarla en un esquema fácilmente identificable.

El vocabulario incluido en la especificación RDF es muy básico y por ello fue extendido a *RDF Schema* [33], por lo que la gran mayoría de las bases de datos RDF actuales contienen ambos vocabularios.

RDF es un modelo abstracto con varios formatos de serialización, por lo que la codificación de un triple varía dependiendo del tipo de archivo en el que se guarde. RDF/XML

[1] específicamente por un property graph (grafo con propiedades)

fue la primera codificación estándar para serializar RDF (en un archivo XML) y si bien es potente, es difícil de leer por las personas. Turtle [34] y N-Triples [35] son otros formatos comúnmente utilizados, su simpleza los hace ideales para el entendimiento y procesamiento de los triples.

N-Triples es una subsección del lenguaje Turtle y su característica principal es la facilidad que presenta a la hora de analizar su sintaxis. En un archivo N-Triples (`.nt`) solo se pueden escribir tres URIs seguidas de un punto para representar un triple RDF y todo lo que esté después de un numeral (`#`) se considera un comentario.

Podemos encontrar una descripción completa de las características y sintaxis de Turtle en [34], de N-Triples en [35] y de RDF/XML en [36].

2.1.3. SPARQL

SPARQL (del ingles *SPARQL Protocol and RDF Query Language*), es un lenguaje estandarizado para consultar grafos RDF. Este se constituyó como una recomendación oficial por la W3C en el 2008 [32] y su versión actual es la 1.1 [37].

SPARQL provee un set completo de operaciones analíticas para sus consultas definidas directamente en la especificación. Particularmente provee 4 formas de consultas:

- **`SELECT`**: Retorna valores en forma de tabla.
- **`CONSTRUCT`**: Retorna valores en forma de triples RDF.
- **`ASK`**: Retorna un resultado binario a la consulta (`True/False`).
- **`DESCRIBE`**: Retorna un grafo RDF con contenido que el administrador del *endpoint* SPARQL considere información útil.

A excepción de `DESCRIBE`, las demás consultas necesitan un bloque `WHERE` con sintaxis similar a Turtle en el cual se determinan las restricciones de la búsqueda en forma de triples RDF con variables y URIs.

Además de las consultas, SPARQL provee múltiples funciones como son las condicionales (`if`, `exists`, etc), de conversión (`str`, `lang`, etc), de comprobación (`isNumber`, `isBlank`, etc) y modificadores de respuesta como `ORDER BY`, `DINSTINCT`, `REDUCED`, `LIMIT` y `OFFSET`. Una descripción completa del lenguaje puede encontrarse en [38] y en [37].

```
1  PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
2  PREFIX dbo: <http://dbpedia.org/ontology/>
3  PREFIX : <http://dbpedia.org/resource/>
4
5  SELECT ?person ?birth ?death WHERE {
6        ?person dbo:birthDate ?birth .
7        ?person rdf:type  dbo:Athlete.
8        ?person dbo:birthPlace :Cuba .
9
10
11       FILTER (?birth < "1900-01-01"^^xsd:date) .
12       OPTIONAL{
13           ?person dbo:deathDate ?death .
14 }
15 } LIMIT 100
```

Figura 2.3: Ejemplo de consulta tipo `SELECT`.

La mayoría de las formas de consulta SPARQL contienen un conjunto de patrones triples llamados *basic graph pattern* (bgp). Los patrones triples son como triples RDF excepto que cada sujeto, predicado y objeto puede ser una variable (en la Figura 2.3, "?person" "?birth" y "?death" son ejemplos de variables). Un *bgp* coincide con un subgrafo de los datos RDF cuando los términos RDF de ese subgrafo pueden sustituirse por las variables y el resultado es un grafo RDF equivalente al subgrafo. [37]

En la Figura 2.3 se ve un ejemplo de una consulta SPARQL al *endpoint* de DBpedia[2], en ella seleccionan todos los recursos cuyo lugar de nacimiento es Cuba, que además son atletas y que tienen una fecha de nacimiento menor a 1990 (restricción explicitada mediante el operador FILTER). Mediante el operador OPTIONAL se solicita también la fecha de muerte en caso de estar registrada, este es similar a una operación de *LEFT OUTER JOIN* en bases de datos relacionales. La consulta retornará un máximo de 100 resultados y estos serán representados en una tabla con tres columnas: "?person" "?birth" y "?death".

Una diferencia importante entre los lenguajes SQL y SPARQL es que en este último el concepto de 'Tablas' y 'atributos' no existen. Lo que sería una tabla en SQL se construye de manera dinámica en dependencia de los joins entre triples que contienen variables en común. Por ejemplo, en la Figura 2.3 se forma algo similar a una tabla de manera dinámica mediante los joins de los triples de las líneas 6,7,8 y 13.

En la Figura 2.3 se muestra una consulta SPARQL más compleja en la cual se observa el operador *UNION* cuya función es unir los resultados de la sub-consulta superior (definida por los bgp de las líneas 6,7 y 8) e inferior(líneas 12,13 y 14).

En [39] se estudia la complejidad en la evaluación de consultas SPARQL teniendo en

[2] `http://dbpedia.org/sparql`

```
1  PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
2  PREFIX dbo: <http://dbpedia.org/ontology/>
3  PREFIX : <http://dbpedia.org/resource/>
4
5  SELECT ?person ?birth  WHERE {{
6        ?person dbo:birthPlace :Cuba .
7        ?person rdf:type  dbo:Athlete.
8        ?person dbo:birthDate ?birth .
9        FILTER (?birth < "1900-01-01"^^xsd:date) .
10    }
11    UNION {
12       ?person dbo:birthPlace :Cuba .
13       ?person rdf:type  dbo:Artist.
14       ?person dbo:birthDate ?birth .
15       FILTER (?birth < "1900-01-01"^^xsd:date) .
16
17    }
18 }
```

Figura 2.4: Ejemplo de consulta tipo `SELECT`.

cuenta varios aspectos del lenguaje. Se presenta, mediante lógica de conjuntos, operaciones básicas por las cuales se puede resolver de manera eficiente la evaluación de una consulta, gracias a la conmutatividad y asociatividad de los operandos. Además se evidencia la problemática de la obtención del mínimo conjunto de datos y la transformación de operaciones como `OPTIONAL`, `FILTER`, `MINUS`, entre otras, en una combinación de las básicas.

2.1.4. Modelos de datos de grafos

En [40] se define formalmente una base de datos de la siguiente forma:

'Sea M un modelo de base de datos. Un esquema de base de datos en M es un conjunto de restricciones semánticas permitidas a M. Una instancia de base de datos en M es una colección de datos representados según M. Una base de datos en M es un par ordenado $D^M = (S^M, I^M)$, donde S^M es un esquema e I^M es una instancia. Si una base de datos D^M no define su esquema, es decir, $D^M = (\emptyset, I^M)$, diremos que D^M es una base de datos sin esquema. Si D^M no define su instancia, es decir $(S^M, \emptyset)$, entonces D^M se define como una base de datos vacía. Si D^M define tanto el esquema como la instancia, entonces tenemos una base de datos completa'.

Nótese que la definición anterior no establece que la instancia de la base de datos satisfaga las restricciones definidas por el esquema de la base de datos. Lo anterior es precisamente lo que pasa de manera común en muchas bases de datos de grafos, donde se define una ontología (el esquema) y un conjunto de datos que suelen violar dichas restricciones.

Además de las bases de datos RDF o $RDFstore$ que se basan en el modelo RDF definido en 2.1.2, existen otros tipos de bases de datos como las basadas en los modelos $ProperyGraph$ [41] y $RDF*$ [42].

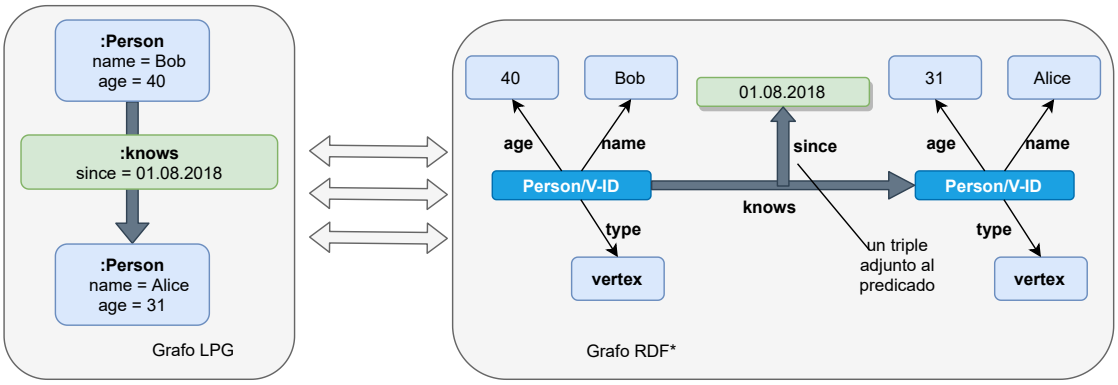


Figura 2.5: Comparación entre los grafos LPG y $RDF*$: una transformación de LPG a $RDF*$, que permite adjuntar triples a predicados de triples. Ambos modelos proveen la capacidad para etiquetar las relaciones.

Un grafo etiquetado de propiedades (LPG) es un gráfico múltiple dirigido etiquetado cuya característica principal es que los vértices y aristas pueden contener un conjunto (posiblemente vacío) de pares nombre-valor denominados propiedades. Desde el punto de vista del modelado de datos, cada nodo representa una entidad, cada arista representa una relación (entre dos entidades) y cada propiedad representa una característica específica (de una entidad o relación). Una definición formal del modelo puede ser encontrada en [40].

Por su parte $RDF*$ se basa en extender el modelo RDF con una noción de triples anidados. Concretamente, esta extensión le permite a los triples representar metadatos sobre otro triple utilizando directamente este otro triple como su sujeto o su objeto [42].

En la Figura 2.8 se evidencia como con $RDF*$, se pueden representar las aristas o relaciones de LPG de forma natural. Específicamente, las aristas se pueden almacenar como triples, y las propiedades de la arista se pueden vincular al triple de la arista a través de otros triples.

2.1.5. Sistemas de bases de datos y planes de ejecución

En [43] se presenta un estudio bastante exhaustivo de sistemas de bases de datos de grafos. Se identifican y analizan categorías fundamentales de estos sistemas, los modelos de grafos asociados, técnicas de organización de datos y diferentes aspectos en la distribución

de datos y la ejecución de consultas.

Entre los retos identificados en este se plantea que el establecimiento de un modelo de grafos único para estos sistemas está lejos de ser completo. Si bien el LGP se usa con mayor frecuencia, su definición es muy amplia y rara vez tiene un respaldo completo. A su vez, RDF también es muy popular en el contexto de almacenamiento y administración de grafos.

Esta investigación se utilizará el framework Apache Jena, el cual provee un conjunto de herramientas de código abierto para la administración e interacción con un almacén de triples. Entre estas herramientas se encuentra *TDB2*, componente que permite consultas y almacenamiento de datos utilizando el modelo RDF. Este también es compatible con la gama completa de API de Jena y se puede utilizar como un almacén RDF de alto rendimiento en una sola máquina. También puede ser utilizado con Apache Jena Fuseki, otro componente que provee un servidor SPARQL y funciona como una capa de alto nivel para interactuar con Jena. La implementación de los componentes del framework Jena está disponible en el lenguaje java y se encuentra bien documentado, lo cual hace del framework una solución atractiva para la integración con componentes de terceros y la investigación.

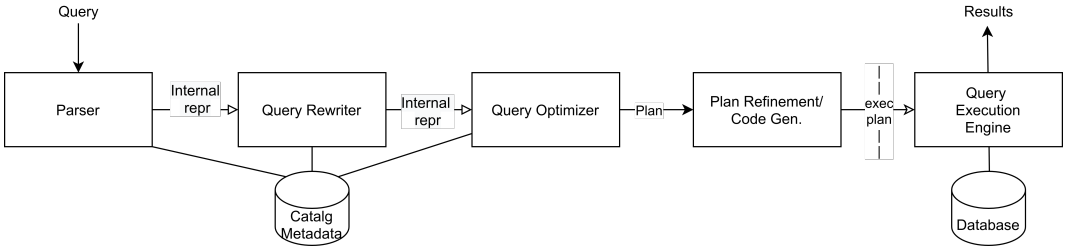


Figura 2.6: Fases genéricas del procesamiento de consultas.

Desde la perspectiva de una consulta, en [44] se presenta una arquitectura general (Figura 2.6) con los siguientes componentes: (1) analizador de consultas, (2) reescritura de consultas, (3) optimizador de consultas, (4) componente de refinamiento de planes y motor de ejecución de consultas. El analizador lee la consulta y la transforma en la representación interna del sistema. A continuación, en (2) se reescribe la consulta creando un plan de consulta lógico a partir de esta representación interna. El optimizador de consultas es el encargado de aplicar diferentes optimizaciones en función del tipo de sistema, el *estado físico*, los índices a utilizar, los nodos a los que enviar la consulta, etc. Como resultado, el optimizador de consultas genera un plan de consultas optimizado que especifica cómo se ejecutará la consulta. Este plan es refinado y transformado en un plan ejecutable

por el componente de refinamiento del plan. Este plan será ejecutado por el motor de ejecución de consultas en cada nodo local. El motor de ejecución de consultas proporciona implementaciones genéricas para cada operador en el plan de consultas. Finalmente, el componente de catálogo (o metadatos), almacena información sobre las bases de datos (esquema, tablas, vistas o información física sobre ellas) que se puede utilizar durante el análisis, la reescritura de consultas y la optimización de consultas [45].

```
1  PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
2  PREFIX dbo: <http://dbpedia.org/ontology/>
3  PREFIX : <http://dbpedia.org/resource/>
4  SELECT  ?person ?birth ?death
5  FROM <http://dbpedia.org>  WHERE {
6    {
7    ?person dbo:birthPlace :Cuba ;
8    tdf:type dbo:Athlete .
9    ?person dbo:birthDate ?birth ;
10      FILTER (isLITERAL (?birth ))
11      FILTER ( ?birth < "1900-01-01"^^xsd:date )
12   }
13   OPTIONAL {  ?person dbo:deathDate ?death . }
14 }
15 LIMIT  100
```

```
1  (stats
2    (meta
3      ...
4      (count 89337663))
5    ...
6    ((VAR rdf:type dbo:Athlete>) 114848)
7    (dbp:birthPlace 41949)
8    ...
9    (dbo:birthDate 232640)
10   ...
11   (dbo:deathDate 69706)
12   ...
13   (dbo:federalState 81)
14   ...
```

Figura 2.7: A la izquierda, la consulta 2.3 optimizada según las estadísticas (derecha) de cardinalidades.

En la Figura 2.7 se observan algunos detalles del proceso de optimización ejecutado para la consulta de la Figura 2.3. En la línea 4 se añade la cláusula *FROM* que limitar el subgrafo específico que va a ser consultado, esto forma parte de optimizaciones internas. El orden de los triples está determinado por las estimaciones de cardinalidades para predicados y tipos de triples. Nótese que en este caso se ejecuta primeramente el triple (*?person dbo:birthPlace :Cuba*) debido que el predicado *dbo:birthPlace* es el de menor cardinalidad definido en el archivo de estadísticas. Nótese además que el triple (*?person dbo:birthDate ?birth*) se ejecuta es el tercero ejecutado debido a que es un triple de tipo var_uri_var por lo que suele ser más costoso durante la ejecución. La línea 8 de la Figura 2.7, incluye la aplicación de un filtro *FILTER (isLITERAL(?birth))* que no pertenece a la consulta original. Este forma parte de optimizaciones internas que pueden o no ser realizadas durante el proceso de optimización.

Durante el proceso de optimización, se escoge entre varios planes lógicos el plan físico a ejecutar. El tiempo de ejecución de una consulta está estrechamente ligado al plan de físico de ejecución generado. Una selección inadecuada del orden en que se ejecutan los resultados intermedios de una consulta puede provocar afectaciones importantes en el

tiempo de ejecución de una consulta, así como de los recursos de hardware. La selección del mejor plan físico para cada consulta es el 'santo grial' de los DBMS.

2.2. Algoritmos de aprendizaje automático

Tom Mitchell en su libro [46], definió un algoritmo de aprendizaje automático como:

"Se dice que un programa de computadora aprende de la experiencia **E** con respecto a alguna clase de tareas **T** y medida de desempeño **P**, si su desempeño en tareas en **T**, medido por **P**, mejora con la experiencia **E**".

Los algoritmos de aprendizaje automático se pueden clasificar en términos generales como no supervisados o supervisados por el tipo de experiencia (**E**) que se les permite tener durante el proceso de aprendizaje. En particular, un algoritmo de aprendizaje supervisado aproxima una función o *hipótesis* $f^* : \mathcal{X} \longrightarrow \mathcal{Y}$ a partir de un conjunto de ejemplos $S = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_M, y_M)\}$. El algoritmo define un mapeo $\mathbf{y} = f(\mathbf{x}; \theta)$ donde se aprenden el valor de los parámetros θ que resultan en la mejor función que aproxime las salidas $\mathbf{y}$ a partir de la entradas $\mathbf{x}$.

El término θ, se refiere a los parámetros propios de la función de aproximación aprendidos durante el proceso de entrenamiento y no deben ser confundidos con el término *hiperparámetro*. La mayoría de los algoritmos de aprendizaje automático tienen configuraciones llamadas hiperparámetros, que deben determinarse fuera del propio algoritmo de aprendizaje y que controlan muchos aspectos del comportamiento de este. Algunos de estos hiperparámetros afectan el tiempo y el costo de memoria de ejecutar el algoritmo. Otros afectan la calidad del modelo obtenido en el proceso de entrenamiento y su capacidad para inferir resultados correctos cuando se evalúa en nuevas entradas. Por ejemplo, algunos hiperparámetros utilizados en las redes neuronales son: la cantidad de capas, unidades por capas, funciones de activación, entre otras.

Cuando $\mathcal{Y} = \mathbb{R}$ estamos en presencia de una tarea de regresión, por ejemplo, aprendemos una hipótesis $f : \mathbb{R}^n \to \mathbb{R}$ [47]. Entre los modelos comúnmente utilizados en esta tarea se encuentran: Regresión lineal, un tipo de Máquinas de vectores de soporte conocidas como SVR, y Redes neuronales ajustadas para problemas de regresión. En nuestro caso, un ejemplo de entrada consiste en un conjunto de características que representan una consulta SPARQL y su tiempo de ejecución es el objetivo.

Para evaluar las habilidades de un algoritmo de aprendizaje automático, debemos diseñar

una medida cuantitativa de su rendimiento. Por lo general, esta medida de rendimiento **P** o *loss function* es específica para la tarea **T** que está llevando a cabo el sistema [47]. Por ejemplo, para la configuración de regresión, el error cuadrático medio (MSE) se usa comúnmente. Considerando todos los pares $(\mathbf{x}, y)$ del conjunto de entrenamiento S, este mide el promedio de los errors cuadrados para cada predicción $f(\mathbf{x}_i) = \hat{y}_i$ con respecto a su target original y_i:

$$MSE_{(test)} = \frac{1}{m}\sum_{i=1}^{m}(y_i^{(test)} - \hat{y}_i^{(test)})^2. \tag{2.1}$$

El súper índice $(test)$ en cada término, enfatiza que la evaluación de las métricas, deben ser reportadas sobre un conjunto de datos totalmente aislado de los datos utilizados en el entrenamiento, este se conoce habitualmente como conjunto de *test*.

Normalmente, al entrenar un modelo de aprendizaje automático se tiene acceso a un conjunto de entrenamiento. Sobre este se calcula alguna medida de error obteniéndose lo que se denomina *error de entrenamiento*. Esta es luego utilizada en el proceso de optimización que consiste en reducir el error de entrenamiento. Lo que separa el aprendizaje automático de la optimización, es que se desea que el algoritmo sea capaz de obtener errores bajos también sobre un conjunto de entradas nunca antes vistas durante el entrenamiento. La capacidad que tienen los algoritmos de aprendizaje automático para ofrecer un buen rendimiento sobre entradas no observadas previamente se denomina *generalización* y es el desafío central en el aprendizaje automático [47].

2.3. Predicción de rendimiento

Como hemos mencionado, la predicción de rendimiento de consultas SPARQL no ha sido muy abordada. Hasta donde sabemos, el trabajo más relevante es el propuesto por Hasan y Gandon en 2014 [27]. En este se aplican los algoritmos de aprendizaje automático Support Vector Machine para regresión (Nu-SVR) y K-Nearest Neighbors(Knn) para predecir el tiempo de ejecución de las consultas. Las consultas SPARQL se codifican como vectores de entrada de los algoritmos mediante la mezcla de los siguientes enfoques:

- **Algebraicas:** Se extrae la frecuencia de los operadores presentes en el cuerpo de la consulta. Cada operador definido en el lenguaje SPARQL representa una característica en el vector asociado a la consulta, siendo la frecuencia el valor asociado a esta. En

la práctica, los operadores que brindan información útil son: BGP, JOIN, LEFTJOIN, UNION, FILTER, PROJECT y DISTINCT. Luego se agrega una característica para la cantidad de triples presente en la consulta y otra para la profundidad del árbol de la consulta. Esta información se extrae utilizando el componente analizador (parser) ARQ SPARQL [48] de Jena.

- **Patrones de grafos:** se representan las consultas como grafos para extraer patrones de estos. En este caso las características representan la similaridad del grafo (la distancia de edición en este caso) de cada consulta con respecto a las K consultas más representativas del conjunto de entrenamiento. Estas consultas corresponden a los centros (medoids) resultantes de aplicar el algoritmo K-medoids sobre el dataset de consultas [27]. La intuición está relacionada a que las consultas SPARQL están en esencia formadas por patrones de triples que buscan coincidencias en los datos almacenados en la bases de datos. Los patrones de grafos se conforman con estos patrones de triples, por lo cual si los patrones de grafos de dos consultas son muy similares pudieran tener asociados tiempos de ejecución similares.

Los mejores resultados se obtienen con el modelo Nu-SVR con un híbrido de ambos métodos de codificación de las consultas. El dataset de entrenamiento se generó a partir de 25 plantillas de patrones comunes que siguen las consultas de DBpedia. En total se entrenó sobre 1260 consultas generadas. El modelo se evaluó utilizando las métricas R2 Score y Root Mean Squared Error.

En [49], los autores utilizan enfoques similares de extracción de características y modelos a lo propuesto en [27]. Además del tiempo de ejecución se predicen otras métricas de rendimiento como el uso de CPU y memoria. Similar a [27], se utilizan los siguientes enfoques para la extracción de características: Algebraicas: ocurrencias de los operadores en la consulta, altura mínima y máxima en la que aparece cada operador en el árbol que devuelve el parser ARQ de Jena. Patrones de grafos: a diferencia de [27], se modela cada tipo de triple como un subgrafo estructuralmente diferente con el objetivo de diferenciar más los ejemplos de entrenamiento. El cálculo de la distancia de edición entre todas las consultas se sustituye mediante la selección aleatoria de los K centroides de las consultas generadas en cada plantilla utilizada en [27]. También se considera la diferenciación de las predicciones con respecto a la primera ejecución de las consultas(cold stage) y el promedio de las siguientes 10 ejecuciones de la misma consulta (warm stage). Los resultados reportados

mejoran ligeramente con respecto al enfoque utilizado en [27]. En este caso el algoritmo k-NN obtiene mejor resultado que el SVR. Sin embargo, no se disponibiliza los datos ni el código utilizado para garantizar las reproducción del experimento.

En [50] se propone un marco de trabajo de aprendizaje general que hace uso de estadísticas y emplea modelos de regresión como Linear Regression, Random forest, Knn y SVR para predecir el rendimiento de consultas de analítica de grafos. Las técnicas de extracción de características utilizadas son las siguientes:

- **Query:** estadísticas de instancias de consultas que codifican restricciones topológicas (el tamaño, profundidad, presencia de ciclos), y restricciones semánticas de los términos de la consulta (labels, funciones de transformación).
- **Sketch:** estadísticas que estiman la especificidad y ambigüedad de una consulta al aplicar técnicas de muestreo(sampling) de los datos a los que accederán las consultas.
- **Algoritmos:** características de los algoritmos de consulta de grafos utilizados (reachability, dual-simulation y top-k)

Las métricas para validar la efectividad de los predictores propuestos son: *R-Squared*, *Mean Absolute Error* (MAE) y *Normalized Root Mean Squared Error* (NRMSE). Los datos y modelos utilizados no se encuentran disponibles. En esta investigación se señala que las características de operadores, patrones de grafos, así como la caracterización a priori de planes de consultas (como las utilizadas específicamente en consultas SPARQL), no son idóneas para las consultas de analítica de grafos utilizadas, por lo que no se realiza una comparación con los métodos anteriormente presentados.

2.3.1. Predicción de cardinalidades en RDBMS

A diferencia de los GDB, la predicción de rendimiento utilizando modelos de redes neuronales, si está siendo ampliamente abordada en los RDBMS. Los trabajos iniciales estuvieron vinculados a la optimización de planes de ejecución mediante la predicción de cardinalidades como los sistemas de optimización tradicionales. Seguidamente incluimos algunos por su importancia sobre todo para el proceso de extracción de características.

Akdere y su grupo proponen en el 2012 DQ [20]. En este se usan técnicas de aprendizaje reforzado con modelos de costos tradicionales diseñados por humanos para aprender

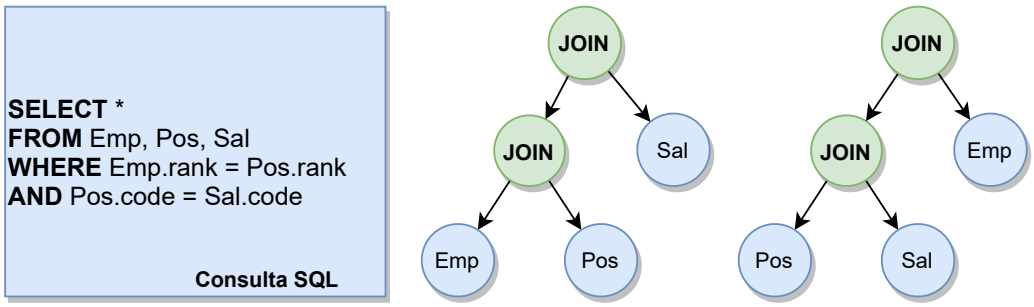


Figura 2.8: Planes lógicos para una consulta SQL. Un optimizador basado en cardinalidades escoge el plan de ejecución entre 1 o más planes lógicos teniendo en cuenta la cardinalidades estimadas.

automáticamente planes de consulta optimizados. En DQ se propone una red neuronal densa para minimizar el costo (cardinalidad de los resultados intermedios) de un subplan resultante de una operación de join. De los subplanes se codifican las columnas visibles (incluidas en el operador SELECT) luego de una operación de join utilizando vectores '1-hot' que luego se concatenan. Durante la búsqueda del plan óptimo para una consulta se itera sobre las predicciones de los posibles joins eligiendo en cada paso el menos costoso hasta que se conforma el plan completo. Un procedimiento similar a DQ lo proponen Marcus y Papaemmanouil el 2018 con ReJOIN [21]. En este se reconstruye el plan de la consulta optimizado mediante técnicas de aprendizaje reforzado. En cada paso o episodio, el optimizador sugiere el próximo join a realizar en el plan de ejecución y se genera el próximo estado generando un árbol binario. El vector de características de cada estado se obtiene aplicando los siguientes enfoques:

- **Tree Structure:** captura datos de la estructura del árbol, codificando cada sub-árbol binario respetando el orden de los joins realizados en cada episodio. Se representa con un vector fila v donde cada columna v_i representa un join del conjunto de posibles joins realizables en la base de datos. El valor de v_i es 0, si la relación no está presente en el estado actual de la sub-consulta o $1/h(i/x)$, siendo $h(i/x)$ la altura de la relación en el árbol del plan de ejecución generado.

- **Join Predicates:** captura información de los joins en cada estado. Se representa como una matriz en la que filas y columnas corresponde a la tablas. Se marca con 1 en la $[fila; columna]$ que contiene un join, 0 en caso contrario.

- **Selection Predicados:** similar al anterior esta vez se registran los atributos presentes

en cada estado con un vector de k dimensiones (número de atributos presentes en todas las relaciones de tablas).

El 2018 Kipf y su grupo proponen un estimador de cardinalidades con aprendizaje profundo denominado Multi-Set Convolutional Network (MSCN) [17]. Este predice correlaciones de joins dentro de los datos, abordando algunos puntos débiles en técnicas como el muestreo de estadísticas básicas por tabla. MSCN se basa en Deep Sets [51] que permite expresar características de consulta mediante conjuntos, que posteriormente pueden utilizarse como entrada para los modelos de deep learning. Las características de una consulta se extraen mediante la definición de 3 sets:

- **Table Set:** se representa con un vector one-hot para cada tabla en la consulta y adicionalmente información de muestreo de cada tabla.
- **Join Set:** se representa al igual que en el 'Table Set' con vectores one-hot para los joins entre tablas que ocurren en la consulta.
- **Predicate Set:** se representan los predicados en triples de la forma (col, op, val), utilizando vectores one-hot para el predicado (col), el operador (op) y para el valor un número normalizado entre $[0 : 1]$.

El modelo utilizado no es aplicable a la predicción de latencia de consultas si se tiene en cuenta que los conjuntos no respetan el orden de los joins que se suceden. Sin embargo, la codificación utilizada aporta algunos elementos reutilizables.

En general estos estimadores de cardinalidad se sustentan en el uso de estadísticas de predicados y operadores. En algunos casos se representan la estructura de las consultas como árboles, sin embargo se utilizan modelos MLP (fully connected neural nets) los cuales no poseen el sesgo inductivo adecuado para estas [52]. Marcus y otros en [13], concluyen que la predicción de cardinalidad no necesariamente guía a la optimización de un buen plan.

2.3.2. Predicción de rendimiento en RDBMS with ML

Unos de los trabajos pioneros en la predicción de rendimiento utilizando técnicas de aprendizaje de máquinas se presenta en [26]. Las consultas se codifican utilizando características a nivel de los operadores conjuntamente con información del plan de ejecución de las consultas:

- **A nivel de plan:** características de estimaciones del optimizador de consultas, como cardinalidades del operador y costos de ejecución del plan, junto con el conteo de ocurrencias de cada tipo de operador en el plan de consulta. Para este tipo de características se utiliza un solo modelo(se experimentó con Nu-SVR y Kernel Canonical Correlation Analysis (KCCA)).

- **A nivel de operadores:** se extraen características que aplican a todos los operadores en base a estadísticas proporcionadas por el motor utilizado como: estimaciones de I/O, número de tuplas, selectividad, y los tiempos de ejecución de cada operador. En este caso se utilizan modelos lineales multivariados para cada operador. Estos se anidan según el orden de ejecución provisto por el motor, utilizando las predicciones de los operadores de menor nivel como entradas para el próximo operador en el árbol hasta retornar la predicción completa del plan.

La información de las características de las consultas se extraen utilizando PostgreSQL debido a que este provee información de las optimizaciones como: estimaciones estadísticas, estructura del plan de consulta, operadores de selectividad. La evaluación se realiza con los datos de TPC-H, de los cuales se generan consultas utilizando 14 plantillas. Utilizan Mean Relative Error(MRE) como métrica de evaluación.

En [22] se propone una red neuronal profunda orientada también a entrenar estructuras de planes de consultas. Se sigue un concepto similar al propuesto en [34] en el que se anidan pequeños módulos o unidades de redes neuronales densas (nodos), para construir una red con estructura de árbol. Cada unidad neuronal procesa una operación en el plan, las unidades hojas corresponden a operaciones de scan sobre tablas. Lo mismo ocurre con el resto de los nodos que predicen la latencia de la operación de join. La red es alimentada con información relativa al plan en cada nodo de consulta que devuelve el optimizador de PostgreSQL. Se utilizan como características: las estimaciones de costos, I/O, resultados intermedios, tipo de join, algoritmos de hash, método de ordenamiento, entre otros datos. Además de la latencia intermedia del operador, cada unidad hoja predice también un vector representativo de la entrada procesada. Luego este vector se utiliza como entrada para el siguiente nodo que recibe como entrada los dos nodos (el izquierdo y derecho) más las características inherentes al tipo de join, hasta llegar a la raíz del pan de consulta donde se produce la latencia del plan. Debido a que las características utilizadas son únicamente las que entrega el optimizador de PostgreSQL esta solución no necesita de un diseño de

características definido por expertos. Sin embargo, su éxito depende de que el optimizador entregue información relevante de cada operador del plan de consulta. Esto afecta su reutilización en el contexto de los motores de bases de datos de grafos donde es difícil obtener información tán descriptiva del optimizador, más allá del árbol optimizado.

Neo [13] es una de las investigaciones más recientes y abarcadoras que utilizan modelos de redes neuronales y aprendizaje reforzado para construir una solución end-to-end de optimización de consultas en sistemas RDBMS. El sistema propuesto aprende a tomar decisiones de optimización teniendo en cuenta el orden de joins, operadores físicos y selección de índices. El rendimiento obtenido es competitivo con los optimizadores de sistemas RDBMS empresariales como Microsoft SQL Server y Oracle. Neo agrupa algunas de las técnicas más útiles antes vistas para codificar en vectores las consultas SQL. Las características a nivel de consultas se dividen en 2 componentes. El primero utiliza la mitad superior una matriz de adyacencia para codificar los cruzamientos de joins, cuyos vectores son concatenados luego. El segundo componente codifica las columnas de predicados utilizando alguno de los 3 métodos:

- Vector 1-hot que marca la presencia o no de un predicado.
- Histogramas: similar al anterior pero se sustituye el 1 en la posición del predicado por la selectividad de este(normalizada entre $[0, 1]$) utilizando histogramas.
- R-vector: rescata información semántica en incrustaciones de los predicados utilizando modelos basados en word2vec [53].

A nivel de planes de ejecución se codifican características preservando la estructura inherente de estos como árboles. Para cada plan o sub-plan se crea árbol de vectores donde cada nodo contiene información de: tipos de joins, tipo de operaciones de búsqueda sobre los datos(table, index). La codificación anterior es procesada por una red neuronal profunda llamada 'Value Network' que utiliza convoluciones basadas en árboles (TBCNN) [23] y puede predecir el rendimiento de ejecución de las consultas o subconsultas. Su rendimiento se compara con los optimizadores de consultas de Microsoft SQLServer u Oracle. Las capas *CNN* proveen el *sesgo inductivo* adecuado para problemas de procesamiento de imágenes(computer vision en inglés). Lo mismo ocurre en las series temporales donde el sesgo inductivo adecuado lo provee el uso de *RNN*. De manera similar, la arquitectura de Neo (en especial el uso de *TBCNN*), está diseñada para crear un sesgo inductivo adecuado

para la optimización de consultas. Esto se debe a que refleja la intuición de los expertos en la identificación de patrones que afectan el tiempo de ejecución de los planes de consulta.

Las *Graph Neural Networks*(GNN) son una familia de arquitecturas de aprendizaje profundo apropiadas para datos estructurados en forma de grafos. Estas han mostrado un rendimiento excelente para diversas aplicaciones en dominios como las interacciones químicas [54] y las interacciones sociales [55], predicción de relaciones en grafos de conocimiento, etc. En $RTOS$ se entrena un modelo profundo de aprendizaje reforzado para la optimización del orden de joins en consultas SQL. Este aprovecha el uso del modelo TreeLSTM que es un tipo de GNN. A diferencia de las estructuras LSTM tradicionales que toman datos en secuencia como entrada, El modelo TreeLSTM lee directamente una estructura de árbol como entrada y genera una representación del árbol. Para codificar los árboles de los planes de ejecución RTOS utiliza la implementación Child-Sum TreeLSTM y *N-ary TreeLSTM* propuestos por Tai y su grupo [56] para codificar los sub nodos no ordenados y los órdenes de ejecución en los joins respectivamente. Los resultados reportados por RTOS mejoran otros enfoques ya mencionados como DQ y REJOIN, sin embargo no se realizan comparaciones con soluciones más novedosas como NEO ni se encuentra disponible la implementación.

Capítulo 3

Desarrollo de la solución

En base a la problemática planteada en la sección 1.1 y los avances en la materia descritos en la sección 2.3, a continuación describimos nuestra solución. Como se mencionó con anterioridad, un modelo de predicción basado en redes profundas enfrenta dos principales retos para la predicción del tiempo de ejecución de las consultas: (i) seleccionar el conjunto adecuado de características para codificar una consulta; (ii) la definición de la arquitectura de red que explote las características definidas en (i) para realizar predicciones.

En la sección 3.1 se detallan las técnicas de extracción de características exploradas, que luego aplicaremos a las fuentes de datos identificadas en la sección 3.2. Finalmente, en la sección 3.3.1 se detalla la arquitectura de red neuronal propuesta que utiliza las características extraídas para el entrenamiento y posterior evaluación de la propuesta.

3.1. Extracción de características

La extracción de características de las consultas SPARQL es uno de los dos retos fundamentales para esta investigación. Para algunas tareas de aprendizaje existe un consenso sobre las formas de representación de los datos utilizados. Por ejemplo, las imágenes se representan generalmente utilizando una matriz en la que cada celda representa la intensidad de un píxel en la escala [0,255]. Para imágenes en colores se utiliza el formato RGB que es representado por 3 matrices (una por cada canal). En otras tareas de aprendizaje relacionadas con el procesamiento de señales de audio es común utilizar codificaciones como: 'formas de onda' (waveform) y espectrogramas [57].

A diferencia de estas tareas, las consultas SPARQL no tienen una codificación aceptada

por la comunidad para la tarea de predicción del tiempo de ejecución. En esta investigación se sigue la metodología para la extracción de características en consultas SQL definida en Neo, la cual incluye la definición de características generales de la consulta y de características a nivel de planes de ejecución. A continuación proponemos la extracción de características que utilizamos.

3.1.1. Características a nivel de consulta

Para las características a nivel de consulta utilizamos dos enfoques, la extracción de características *algebráicas* y la extracción de características de *patrones de grafos*.

Características algebraicas

Las características algebraicas registran las frecuencias de los operadores SPARQL presentes o no en las consultas como características. Más en detalle, seleccionamos como características la ocurrencia de operadores: BGPs, joins, OPTIONAL, UNION y FILTER, entre otros. También incluimos como características de la consulta el número de patrones triples (bgps) en las consultas y la profundidad del árbol. Adicionalmente, en lugar de incluir únicamente las ocurrencias de los joins entre BGP, también incluimos los joins entre patrones triples. Esto permite incluir más información en el modelo, ya que la mayoría de los joins ocurren entre los patrones triples de la consulta. Si el operador LIMIT se encuentra definido, se registra su valor numérico.

- bgp
- join
- leftjoin
- union
- filter
- graph
- extend
- minus
- path*
- pathN*
- path+
- pathN+
- path?
- order
- project
- distinct
- group
- assign
- sequence
- slice
- treesize
- triples

La Figura 3.1(parte izquierda) muestra un ejemplo de la extracción de características algebráicas. El procedimiento seguido es el siguiente:

1. Utilizar Apache Jena ARQ para extraer el álgebra en forma de árbol de la consulta SPARQL.

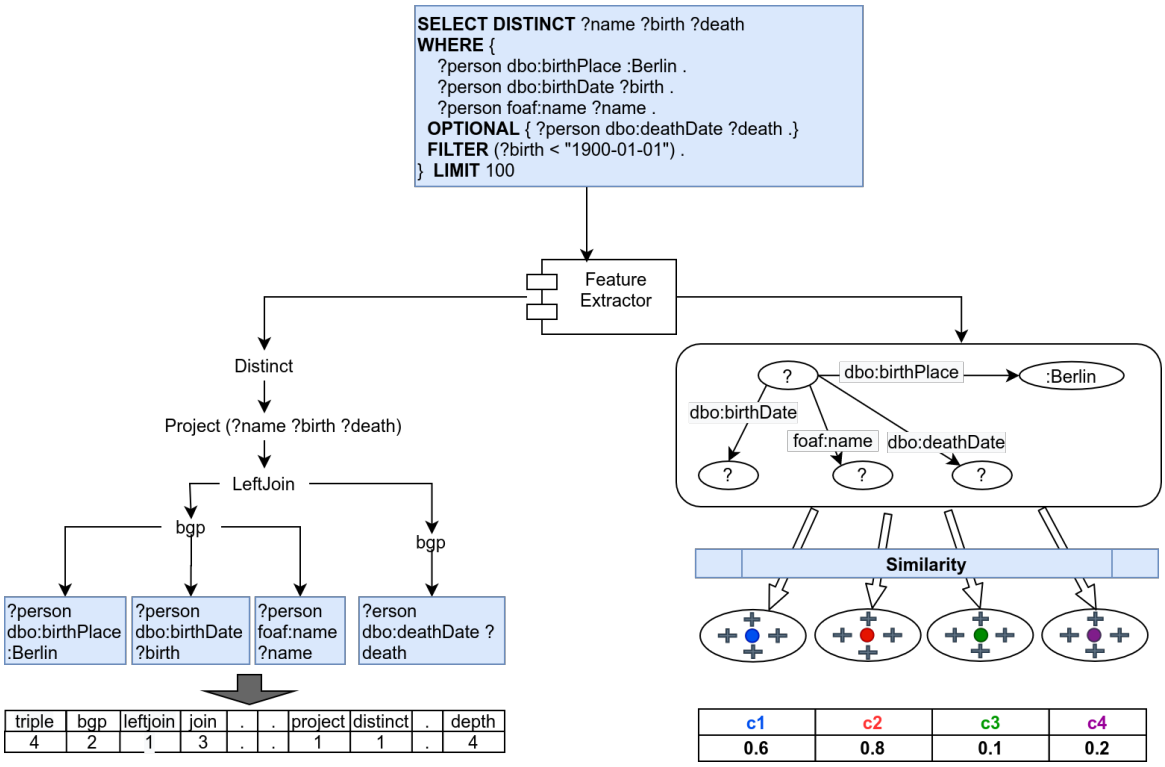


Figura 3.1: Histograma del tiempo de ejecución(en escala logarítmica) de las consultas SPARQL extraídas.

2. Contar frecuencia de los términos usados por consulta.

3. Producir un vector de n características (una por término) por consulta.

4. Agregar la profundidad del árbol y la cantidad de bgps.

Patrones de grafos

El segundo enfoque se beneficia de la capacidad de modelar consultas SPARQL como grafos para extraer patrones de estos. Este usa patrones de similitud entre consultas, representadas como grafos usando un vector de K_{gp} dimensiones, donde el valor de cada característica es la similitud estructural entre ese patrón de consulta y cualquiera de los otros patrones representativos del conjunto de datos.

La Figura 3.1 (parte derecha) muestra un ejemplo de extracción de características de patrones de grafos para una consulta SPARQL. Describimos el proceso de la manera siguiente:

- El primer paso es construir un grafo representativo de la consulta SPARQL. Para

esto los triples se unen teniendo en cuenta los nodos que son variables comunes entre ellos. Finalmente, tanto las variables como los literales son reemplazados por un nodo con un símbolo común (en la Figura 3.1 con el '?').

- El segundo paso es hacer agrupaciones (clusters) a partir de las consultas utilizando el algoritmo k-medoids [58]. K-medioids elige los centroides (centroides Kgp) usando una función de distancia como la *Distancia de Edición*. En la Figura 3.1 cada círculo representa un grupo de consultas en el que cada $centroide$ se identifica con un color.

- Finalmente, para obtener la representación vectorial de K_{gp} (el patrón de consulta), calculamos la similitud estructural entre un grafo de consulta p_j y el centro $k-i$ del grafo de consulta del cluster $C(k)$. El término $d(p_i, C(k))$ es la distancia de edición entre los grafos de consultas p_i y $C(k)$, obteniendo una similaridad normalizada entre 0 y 1 (siendo 0 el más diferente y 1 el más igual).

$$sim(p_i, C(k)) = \frac{1}{1 + d(p_i, C(k))}. \tag{3.1}$$

Los resultados de [27] proponen la combinación de ambos enfoques (*Algebra Features* y *Graph Pattern Features*) para obtener un mejor rendimiento. Usamos esa combinación para extraer las características del conjunto de datos a nivel de consultas. En la siguiente sección definiremos las características a nivel de planes de ejecución.

3.1.2. Histogramas de predicados

Una de las codificaciones a nivel de consulta propuestas en Neo es la codificación de los predicados que incluye una consulta en un vector 'sparse', en el que los predicados constituyen las características del vector de tamaño fijo. El valor asociado a cada característica o predicado, es la selectividad del mismo en el contexto de la consulta (Figura 3.2).

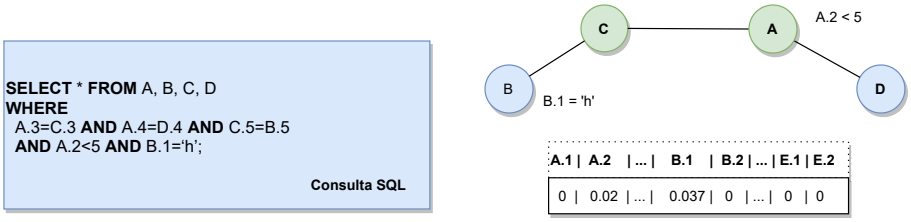


Figura 3.2: Codificación de histogramas de predicados para consulta SQL.

En esta investigación se propone un enfoque similar, en el que utilizaremos como características los predicados de los triples presentes en la consulta. Como valor asociado a estas características proponemos utilizar la selectividad que propone un optimizador tradicional como el de JENA. El término 'predicado' en las consultas SPARQL y SQL son diferentes. En SPARQL los predicados forman parte de los patrones de triples de la consulta, estableciendo la relación entre el sujeto y el objeto de cada triple. Su selectividad está asociada a las coincidencias de aplicar el patrón sobre la base de datos.

El optimizador de Jena propone una estimación de cardinalidad para cada triple. El valor propuesto se calcula en dependencia del tipo de triple y del predicado que incluye este. Por ejemplo, un triple de tipo VAR_URI_VAR, donde el predicado utilizado es $wdt : P31$ ('instance of') tiene una cardinalidad asociada de más de 51 millones de resultados, sin embargo, si está presente en un triple de tipo VAR_URI_URI el optimizador entrega una cardinalidad de '4'. Utilizaremos las cardinalidades del optimizador tradicional normalizadas entre $[0, 1]$ como valor asociado a cada predicado, siendo 1 la máxima cardinalidad entre todos los predicados muestreados.

3.1.3. Características a nivel de plan de ejecución

Independientemente de los operadores que intervienen en una consulta SPARQL, el rendimiento está determinado en gran medida por el orden en que se ejecutan los patrones de grafos que contiene. Como mencionamos en la sección 2.1.5, las consultas son preprocesadas y un componente de optimización suele definir el plan de ejecución de cada consulta. Por ejemplo, en JenaARQ se define en base a la selectividad de los predicados que contiene la consulta. El orden de los JOINs se define seleccionando primero los triples cuyo predicado tiene menor selectividad de acuerdo a un fichero que almacena las estadísticas por predicado. Es por esto que una representación adecuada de la consulta debe codificar su respectivo plan de ejecución como un árbol binario en este caso.

Los optimizadores de consultas como el de JenaARQ no realizan un completo reordenamiento a priori, en su lugar, ejecutan optimizaciones para los resultados parciales. Como consecuencia, si un resultado intermedio no devuelve resultados y pertenece a una operación de INNER JOIN, la ejecución de la consulta termina más allá de los triples no procesados. Esta característica es problemática a la hora de obtener el árbol de ejecución porque obliga a realizar una ejecución completa de la consulta para obtener el árbol real

ejecutado.

El procedimiento para obtener un árbol que se sigue en esta investigación es el siguiente:

- Extraer el árbol del plan lógico de ejecución de la consulta utilizando funcionalidades de JenaARQ.
- Visitar cada nodo del árbol aplicando un *reordenamiento* con la función de optimización de Jena. Dentro de cada BGP se codifican los JOINS entre triples según el paso anterior.
- Unir los BGPs con operaciones de INNER JOIN y LEFT JOINS en caso de estar unidos con el operador OPTIONAL.

El procedimiento anterior tiene la desventaja que genera árboles con el plan de ejecución que en teoría debiera ejecutarse. No obstante, puede contener nodos que en la práctica nunca llegan a ejecutarse, lo que puede agregar ruido a las predicciones. A pesar de ello, es la única vía identificada para evitar la ejecución de la consulta en el proceso de obtención del árbol de ejecución.

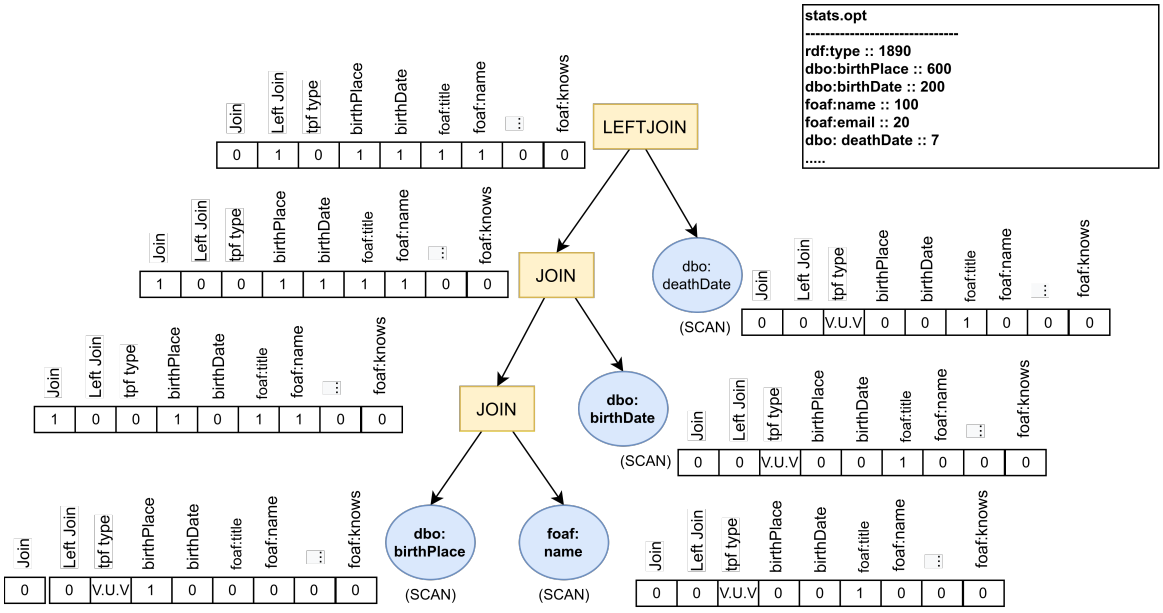


Figura 3.3: Consulta de la Figura 3.1 codificada como un árbol binario.

La Figura 3.3 muestra la codificación de la consulta mostrada en la Figura 3.1. El árbol binario extraído representa el plan de ejecución de la consulta. Los nodos hojas del árbol representan los tripes que ejecutan operaciones de SCAN sobre la base de datos. Por su

parte, los nodos intermedios representa las operaciones de JOIN entre los triples. El plan de ejecución debe analizarse de abajo hacia arriba, siendo los nodos inferiores a la izquierda los primeros en ejecutarse.

Para esta representación del plan de ejecución nosotros utilizamos la siguientes características por tipo de nodo:

Nodos Hojas:

- **triple_type:** Codifica el tipo de triple. Este es un valor categórico que incluye los valores: VAR_URI_VAR, VAR_URI_URI, URI_URI_VAR, LABEL_URI_VAR, VAR_URI_LABEL. Se codifica como un vector one-hot.
- **predicate:** Codifica el predicado del triple. Esta es un valor categórico que todos los predicados muestreados de las consultas. Se codifica como un vector one-hot.

Nodos Intermedios(JOINS):

- **join_type:** Codifica el tipo de join entre los dos nodos hijos del árbol. Esta es un valor categórico que incluye los valores: INNER_JOIN, LEFT_JOIN. Se codifica como un vector one-hot de largo 2.
- **child_predicates:** Representa la suma de los predicados de los nodos hijos. Por lo que la información se replica hacia arriba en la codificación de los JOIN. Por lo cual el nodo ROOT, contiene todos los predicados del árbol.

Los árboles del plan de ejecución de la consulta, pueden ser de profundidad variable, en dependencia de los triples que contenga la consulta. En cambio, los vectores de los nodos son de igual tamaño debido a que contienen todas las características. Esto garantiza una estructura adaptable donde toda la información de los planes puede ser representada y utilizada para entrenar un estimador que aproveche la misma.

Esta información de los nodos puede ser enriquecida agregando información semántica de las uniones y los nodos. Por ejemplo, podría agregarse: la cardinalidad estimada para el nodo según el optimizador, información costo, etc. Lo anterior depende de la granularidad de información que entrega el optimizador para las optimizaciones intermedias. Nótese además que los nodos del árbol incluyen la codificación de predicados específicos utilizados en la consulta. Esto introduce una alta dimensionalidad en los vectores de nodo debido

a que las bases de datos están compuestas por miles de predicados. Por citar un ejemplo, en el caso de las consultas extraídas de Wikidata está presente el uso de más de 2000 predicados distintos en al menos una consulta. Este problema no es común en el contexto de los sistemas relacionales, cuyos joins se ejecutan entre un número bastante reducido de tablas. En la sección 3.3.3 definimos un mecanismo para mitigar este problema.

3.2. Identificación de datasets

El tremendo crecimiento que ha experimentado el aprendizaje profundo se debe en gran parte al uso de computadoras más potentes, conjuntos de datos (en lo adelante *datasets*) más grandes y técnicas para entrenar redes más profundas. Es por esto que la identificación de los datasets a utilizar en nuestra propuesta es una tarea fundamental. En la actualidad, se encuentran disponibles para algunas tareas grandes datasets que han servido para mejorar el desempeño de algunas arquitecturas profundas y que permiten comparar la efectividad de los modelos en un entorno controlado. Por poner un ejemplo, en la tarea de clasificación de imágenes es común encontrar comparaciones en torno al uso de: CIFAR-10 [59](60 mil imágenes), Mnist-10 [60](60 mil) e ImageNet [61](14 millones de imágenes).

Como hemos mencionado, la tarea de predicción del tiempo de ejecución no ha sido abordada exhaustivamente, por lo que no existen datasets disponibles con los cuales realizar comparaciones. La mayoría de los datasets disponibles están orientados a otro tipo de investigaciones fuera del contexto de aprendizaje automático por lo que son pequeños. Quizás el más adecuado puede ser LSQ [62] que contiene miles de logs de consultas útiles de diferentes bases de datos como: DBpedia(3.5.1),LGD, SWDF. Sin embargo, como mencionaremos más adelante, nuestra solución depende de tener un control completo sobre la base de datos y el proceso de ejecución de las consultas.

Por lo anterior, proponemos la conformación de 1 dataset considerando algunos de los procedimientos definidos en LSQ. A continuación se definen los pasos para la creación del dataset utilizando logs de las bases de datos Wikidata:

1. Descargar los archivos de las bases de datos Wikidata [1].

2. Limpiar ficheros con la herramienta Apache Riot y cargar cada uno en una base de datos TDB2 de Apache Jena.

[1] Wikidata latest-trusty: https://dumps.wikimedia.org/wikidatawiki/entities/

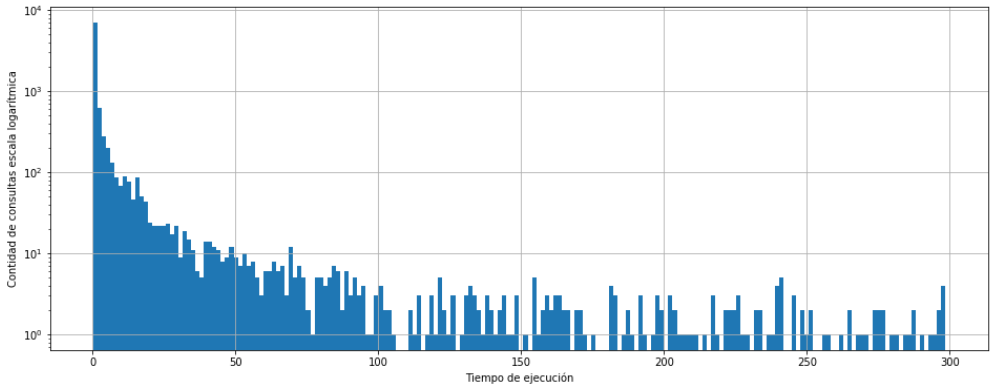


Figura 3.4: Histograma del tiempo de ejecución(en escala logarítmica) de las consultas SPARQL extraídas.

3. Obtener logs de ejecuciones de consultas SPARQL asociados a cada versión de las base de datos.

4. Descartar consultas con operadores como SERVICE que solicitan información externa a las bases de dato local.

5. Implementar y ejecutar las consultas SPARQL para registrar tiempo de ejecución y cardinalidad.

6. Descartar consultas con errores y cero cardinalidad.

La Figura 3.4 muestra el histograma (en escala logarítmica) de consultas SPARQL extraídas para Wikidata. Se observa como la mayoría de las consultas terminan en unos pocos milisegundos/segundos y pueden llegar hasta 300 segundos que es el límite que consideramos. A medida que aumenta el tiempo de ejecución, disminuye la cantidad de consultas disponibles para el entrenamiento. Lo anterior es problemático porque las consultas que más tiempo de ejecución asociado tienen, suelen ser las de mayor interés a predecir con cierta efectividad. Por otro lado, estas diferencias crean un desbalance en el dataset que afecta el proceso de aprendizaje.

Para mitigar esta problemática, extendimos el dataset utilizando consultas SPARQL generadas de manera sintética a partir de 17 plantillas de patrones básicos de consultas utilizando un procedimiento descrito en [63]. Los procedimientos y resultados descritos en las secciones siguientes están acotados a las consultas en el rango $(0, 65]$ segundos. El dataset generado cuenta con las siguientes dimensiones.

- Total de consultas: 29333.
- Total de consultas con tiempo de ejecución $t <= 65$ segundos: 24666.

Finalmente, se decidió separar el 12 % de los datos para el conjunto de *test*. El resto se separó en una proporción de 70 % y 30 % para los conjuntos de entrenamiento y validación respectivamente.

3.3. Arquitectura de red neuronal profunda

A continuación, presentamos la arquitectura similar a la propuesta en NEO, una red neuronal entrenada para predecir la latencia de consultas para un plan de ejecución completo P_f. A diferencia de NEO, nuestro objetivo no es aprender de planes parciales de consultas $Pi \subset Pf$, puesto que esta investigación está acotada a la predicción de latencia. Sin embargo, una propuesta de optimizador que utilice esta arquitectura en el contexto de SPARQL, se debería entrenar sobre los sub-planes P_i para aproximar la mejor estimación posible de latencia tal que $Pi \subset Pf$.

La *experiencia* E de esta Arquitectura, está compuesta por un conjunto de planes de ejecución $P_f \in E$ con una latencia conocida $L(P_f))$. Entrenamos un modelo M para aproximar los $P_f \in E$:

$$M(P_f, \theta^*) \approx \{L(P_f)|P_f \in E\}. \tag{3.2}$$

donde $C(P_f)$ es el *costo* de un plan de ejecución.

El objetivo es minimizar el error de estimar la latencia de la consulta utilizando una función pérdida cuadrática (L2) sobre el conjunto de entrenamiento $A = \{(x_n, y_n)\}_{n=1}^{N}$:

$$\theta^* = \text{argmin}_\theta \text{MSE}(A, \theta). \tag{3.3}$$

siendo *MSE* el error cuadrático medio definido en la ecuación (2.1) y θ^* el conjunto de parámetros del modelo.

3.3.1. Arquitectura

La Figura 3.5, muestra la arquitectura genérica propuesta. Esta fue diseñada para crear un *sesgo inductivo* adecuado para la predicción de la latencia de consultas: el diseño de la

red codifica detalles de la intuición de las causas que producen la ejecución rápida o lenta de una consulta. Los expertos que estudian los planes de ejecución de consultas aprenden a reconocer planes subóptimos o buenos mediante la coincidencia de patrones. La intuición detrás de esta arquitectura es que estos patrones pueden reconocerse analizando subárboles de un plan de ejecución de consultas. En consecuencia, el modelo es esencialmente experto en el reconocimiento de estos patrones que se aprenden automáticamente, a partir de los datos mismos, utilizando una técnica llamada convolución de árboles [23].

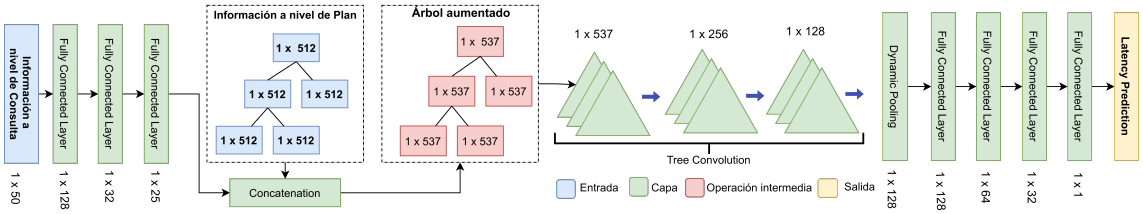


Figura 3.5: Arquitectura genérica de la red neuronal propuesta que incluye las características a nivel de consultas y a nivel de planes de ejecución. Similar a la presentada en [13]. Las unidades por capas son válidas solo para el ejemplo.

Como se muestra en la Figura 3.5, cuando el modelo evalúa un plan de consulta, la codificación a nivel de consulta se introduce en el modelo a través de varias capas densas (fully-connected layers). (i) El vector producido por la última capa densa se concatena con la codificación a nivel de plan, es decir, a cada nodo del árbol se le agrega el mismo vector. Ésta es una técnica conocida como 'replicación espacial' [13], para combinar datos de tamaño fijo (codificación a nivel de consulta 3.1.1) y datos de tamaño dinámico (codificación a nivel de plan 3.1.3). (ii) Una vez que se ha aumentado cada vector de nodo del árbol, este se envía a través de varias capas de convolución de árboles [23], una operación que tiene como entrada y salida un árbol con la misma estructura pero diferente dimensión a nivel de nodos. (iii) Posteriormente, se aplica una operación de agrupación dinámica [23], aplanando la estructura del árbol en un solo vector. Se utilizan varias capas adicionales completamente conectadas para asignar este vector a un valor único que corresponde a la predicción del modelo para el plan introducido.

3.3.2. Convoluciones sobre árboles

Los modelos de redes neuronales como las CNN [64] toman tensores de entrada con una estructura fija, como un vector o una imagen. En la arquitectura propuesta, las

características relativas a cada plan de ejecución están estructuradas como nodos en un árbol (por ejemplo, Figura 3.5). Por lo tanto, usamos la convolución de árbol [23], una adaptación de la convolución tradicional sobre imágenes para datos estructurados en árbol. La convolución del árbol (TreeConvolution) es un ajuste natural para la estructura de los planes. Esta desliza un conjunto de filtros compartidos sobre cada parte del árbol del plan. De manera intuitiva, estos filtros pueden capturar una amplia variedad de relaciones locales entre los nodos padres e hijos. Por ejemplo, los filtros pueden buscar combinaciones del tipo de JOIN además de combinaciones entre los predicados específicos presentes en un JOIN. La salida de estos filtros proporciona señales utilizadas por las capas finales de la red; Los resultados del filtro podrían significar factores relevantes, como las implicaciones del orden en la combinación de predicados, o un filtro podría estimar las implicaciones del uso de un predicado con baja cardinalidad en el lado izquierdo o derecho de un JOIN, el costo asociado a ejecutar primero un tipo de triple determinado. Por ejemplo, ejecutar primero un triple de tipo VAR_URI_URI o VAR_URI_LITERAL disminuye significativamente la cardinalidad de los resultados intermedios y finales y por ende el tiempo de ejecución, mientras que un triple del tipo VAR_URI_VAR suele estar asociado a una alta cardinalidad y por ende a consultas de alta latencia.

En esta investigación se utilizó una implementación [2] de convolución sobre árboles simplificada respecto a la propuesta original. Se utiliza una operación de convolución 1D tradicional aplicada sobre el árbol representado en forma de secuencia como se representa en la Figura 3.6. La secuencia se genera con un recorrido en preorden, en el que en cada paso se agrega no solo la raíz, sino también el hijo izquierdo y luego el derecho del árbol, generando en cada paso la localidad de un árbol (padre / hijo izquierdo / hijo derecho). Debido que a las secuencias son de largo variable, las secuencias de los árboles de menor profundidad se homogeneizan agregando nodos vacíos, teniendo en cuenta el tamaño del árbol de mayor profundidad.

Dado que cada nodo del árbol de consultas tiene exactamente dos nodos hijos, cada filtro consta de tres vectores de ponderación, $\mathbf{e}_p, \mathbf{e}_l, \mathbf{e}_r$. Cada filtro se aplica a cada 'triángulo' local formado por el vector $\mathbf{x}_p$ de un nodo y sus hijos izquierdo y derecho, $\mathbf{x}_l$ y $\mathbf{x}_r$ ($\vec{0}$ si el nodo es una hoja), para producir un nuevo nodo de árbol x'_p:

[2] TreeConvolution utilizado en Neo, simplificado para árboles binarios. Código fuente original: https://github.com/RyanMarcus/TreeConvolution

$$x'_p = \sigma(\mathbf{e}_p \cdot \mathbf{x}_p + \mathbf{e}_l \cdot \mathbf{x}_l + \mathbf{e}_r \cdot \mathbf{x}_r). \tag{3.4}$$

El símbolo $\sigma(.)$ es una transformación no lineal (por ejemplo, ReLU [65]), $(\cdot)$ es un producto escalar (o producto punto) y x'_p es la salida del filtro. Cada filtro combina así información de la vecindad local de un nodo de árbol. El mismo filtro se 'desliza' a través de cada árbol en un plan de ejecución, lo que permite aplicar un filtro a planes de tamaño arbitrario. Se puede aplicar un conjunto de filtros a un árbol para producir otro árbol con la misma estructura, pero con vectores de tamaños potencialmente diferentes que representen cada nodo. En la práctica, se aplican cientos de filtros.

Dado que la salida de una convolución de árbol es otro árbol, se pueden 'apilar' varias capas de filtros de convolución de árbol. La primera capa de filtros de convolución de árbol accederá al árbol del plan de ejecución aumentado (es decir, cada filtro se deslizará sobre cada triángulo padre / hijo izquierdo / hijo derecho del árbol aumentado). La cantidad de información que ve un filtro en particular se denomina campo receptivo del filtro [66]. La segunda capa de filtros se aplicará a la salida de la primera, por lo tanto, cada filtro en esta segunda capa verá información derivada de un nodo n en el árbol aumentado original, los hijos de n y los nietos de n: cada capa de convolución de árbol tiene un campo receptivo más grande que el anterior. Como resultado, la primera capa de convolución de árbol aprende características simples, mientras que la última capa de convolución de árbol aprende características complejas (por ejemplo, reconocer una secuencia de 'left-deep joins'[3]).

Presentamos un ejemplo concreto que muestra como la primera capa de convolución de árbol puede detectar patrones interesantes en planes de ejecución de consultas. En el ejemplo de la Figura 3.6, mostramos dos planes de ejecución que difieren solo en el operador de join superior. Como se muestra, el tipo de unión (left o inner) está codificado en las dos primeras entradas del vector de características en cada nodo. Un filtro de convolución de árbol, compuesto por tres vectores de peso con $\{1, -1\}$ en las dos primeras posiciones y ceros para el resto, servirá como 'detector' para planes de consulta con dos combinaciones de joins secuenciales. Obsérvese como el nodo raíz del plan con dos combinaciones de INNER_JOIN secuenciales recibe una salida de 2 de este filtro, mientras que el nodo raíz del plan con una combinación LEFT_JOIN en la parte superior de un INNER_JOIN

[3]Operaciones de joins entre los elementos de un árbol en el que los nodos profundos y a la izquierda se ejecutan primero.

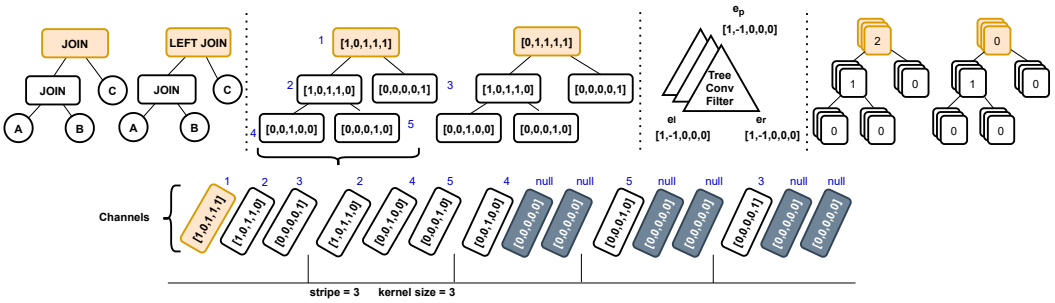


Figura 3.6: Intuición detrás de los filtros en las convoluciones de árboles. La cadena de la parte inferior representa las características generadas para el árbol de la izquierda. Los índices en azul definen el orden de los nodos recorridos en preorden (pero se agrega el nodo raíz, luego el izquierdo y derecho en cada paso) y los índices null corresponde a los vectores de los hijos de los nodos hojas del árbol.

recibe una salida de 0. Las capas de convolución de árbol posteriores pueden usar esta información para formar detectores más complejos, en dependencia de la granularidad de las operaciones físicas implementadas en el motor y disponibles en la codificación[4].

Generar un vector fijo con Dynamic pooling

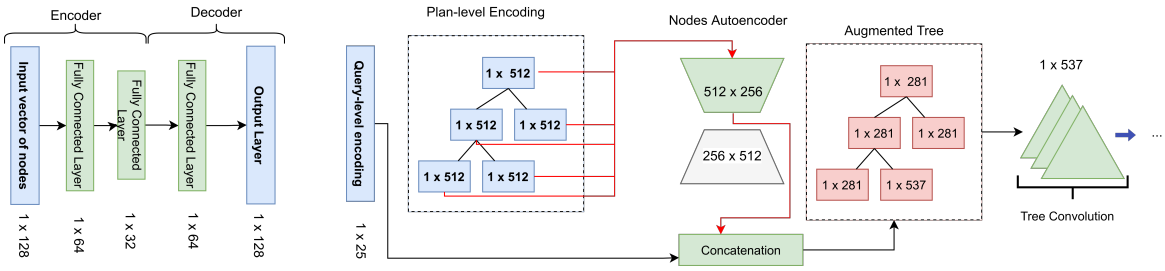
Después de aplicar las capas convolucionales sobre los planes, se extraen las características identificadas por los filtros. El nuevo árbol tiene exactamente la misma forma y tamaño que el original, que varía entre los diferentes planes, pero con nodos de menor dimensión. Por lo tanto, las características extraídas no se pueden enviar directamente a una capa neuronal de tamaño fijo. Debido a esto, se aplica una capa de pooling dinámico [23] para abordar este problema. Concretamente, se toma el valor máximo en cada dimensión de las características que son detectadas por la convolución basada en árboles. Después de la agrupación, el vector fijo que resume la información del plan está apto para ser procesado por capas densas ocultas, hasta que finalmente la capa con 1 unidad de salida permite predecir la latencia de la consulta. Finalmente, la red puede entrenarse en un entorno supervisado utilizando 'backpropagation'.

[4]En [13] los ejemplos presentados están orientados a un nivel de granularidad mayor en la operaciones de joins como las combinaciones, entre Merge Joins, Hash Joins, entre otros. Sin embargo, estos operadores físicos no están disponibles en el motor de ejecución de JENA.

3.3.3. Pre-entrenamiento con autoencoders para nodos hojas.

En las RDBMS las operaciones de JOIN se realizan entre un conjunto reducido y fijo de tablas, sin embargo, el escenario en las GDB es más complejo. Como ya mencionamos, las operaciones de JOIN en las GDB pudiéramos definirlo como 'dinámicas'. Esto se debe a que los JOIN en el lenguaje SPARQL, se definen entre los predicados mediante la coincidencia de variables entre dos triples. Lo anterior implica que en dependencia del tamaño y la diversidad de la ontología de una base de datos, los vectores de nodos del árbol del plan de ejecución pueden llegar a contener miles de predicados. Con el objetivo de abordar esta problemática se propone realizar un preentrenamiento sobre los vectores de nodos utilizando autoencoders. El uso de este permite reducir la dimensión de los vectores de nodos a la vez que conserva la información de los nodos.

El autoencoder propuesto consiste en una red neuronal y se compone de capas densamente conectadas que van reduciendo la dimensión de salida a medida que van siendo apiladas en el 'encoder' y luego aumentan en las capas del 'decoder'. La Figura



(a) Autoencoder (b) Parte la arquitectura que interactúa con las capas del 'encoder'

Figura 3.7: En (a) la arquitectura del autoencoder. En (b) se especifica el uso del autoencoder para preprocesar la información de los nodos del plan.

3.4. Diseño experimental

El entrenamiento de redes neuronales sigue comúnmente el mismo orden: (i) primeramente limpiar los datos y separarlos en conjuntos de *entrenamiento, validación y test*, (ii) luego extraer vectores de características, (iii) entrenar los modelos(seleccionando la mejor combinación de hiperparámetros de acuerdo al conjunto de validación) y seleccionar finalmente el modelo que mejor minimice el error, (iv) probar esos modelos en el conjunto de test.

Para completar el paso (iii), generalmente se utiliza una búsqueda en grilla, lo que permite identificar la mejor combinación de hiperparámetros de manera exhaustiva y utilizando algunas restricciones. Este método puede tomar un tiempo significativo, sin embargo, entrega buenos resultados. La cantidad de épocas, el número de capas, número de unidades por capas, funciones de activación y la taza de aprendizaje son ejemplos de algunos hiperparámetros que ajustaremos.

3.4.1. Selección de la función de activación

Exploraremos la selección de las funciones de activación de la siguiente forma:

- Ejecutaremos 3 iteraciones sobre la misma selección de datos entrenamiento y validación.
- Se entrenarán 100 épocas de una arquitectura formada solo por la capas convolucionales y capas densas finales. Para esta prueba se evitaron los datos y enfoques a nivel de consulta.
- En cada iteración se utilizarán una de las funciones de activación siguientes: Relu, LeakyRelu, Tanh en las activaciones aplicadas luego de las capas convolucionales sobre los árboles.
- Se fijará el optimizador Adam con 0.00015.

En la Figura 3.8, no se muestra una diferencia significativa entre las funciones de activación utilizadas. Se obtienen valores similares en cuanto a las métricas analizadas en los conjuntos de entrenamiento y validación. Se observan variaciones marcadas en algunas épocas del conjunto de validación para todas las métricas, especialmente en el uso de la TanH. Seleccionamos la función de activación Leaky ReLU, debido a que presenta relativamente menos inestabilidad durante el entrenamiento y en los conjuntos de validación entregó ligeramente menores errores para algunas épocas.

3.4.2. Selección de la función de optimización

En esta sección exploraremos la selección de las funciones de optimización de la siguiente forma:

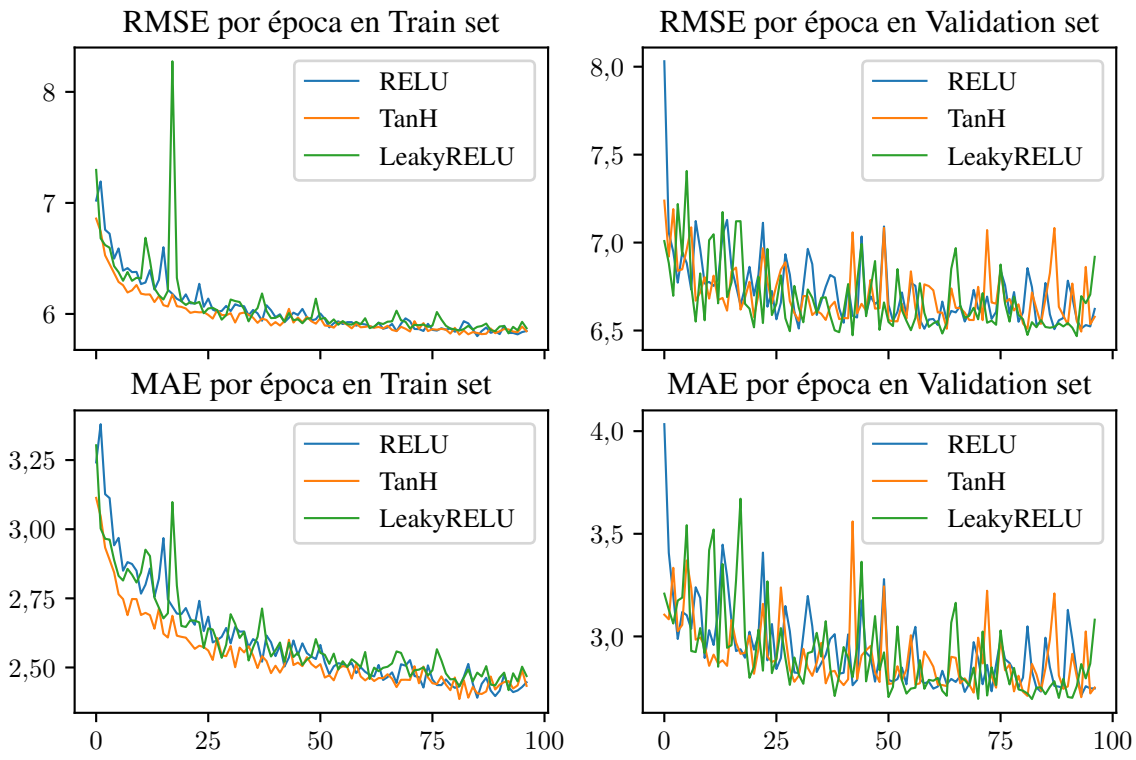


Figura 3.8: Gráficas de las funciones de activación RELU, TanH y LeakyRelu. Métricas Root Mean Squared Error y Mean Absolute Error en los conjuntos Train y Validation.

- Pasos (1) y (2) de la sección anterior 3.4.1.
- Se fijará la función de activación Leaky ReLU.
- En cada iteración se utilizarán una de los optimizadores siguientes: SGD, Adam, Adagrad. Se seleccionará el optimizador que menor error y estabilidad en el entrenamiento obtenga.

La Figura 3.9, muestra el desempeño para los 3 optimizadores probados. Adam y Adagrad aceleran la convergencia a los mejores resultados desde las primeras épocas a diferencia del SGD. Adagrad brinda más estabilidad en el entrenamiento que Adam, sin embargo, con este último se obtienen los menores errores respecto a las métricas analizadas en el conjunto de validación. Por lo anterior, se decide seleccionar Adam como optimizador.

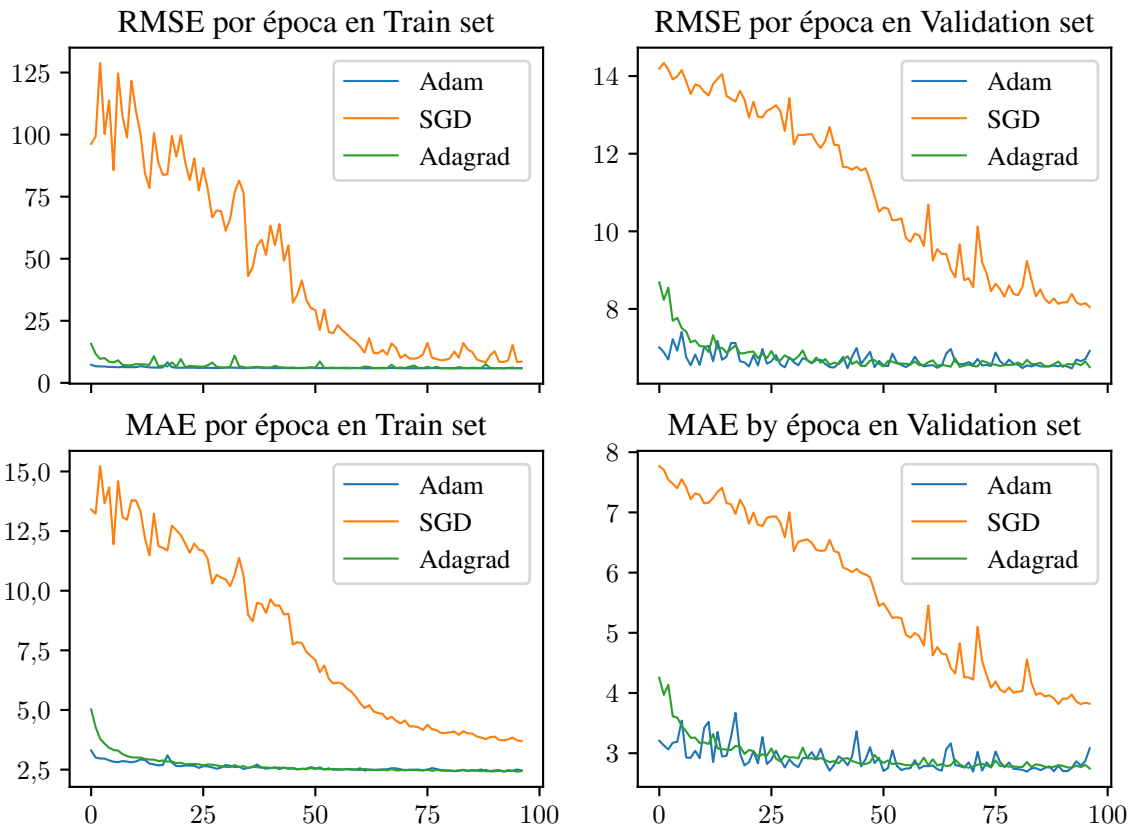


Figura 3.9: Desempeño de las funciones de optimización para 100 épocas sobre las características de los planes de ejecución.

3.4.3. Selección capas de convoluciones sobre árboles

A continuación exploraremos las consecuencias de la profundidad del uso de capas convoluciones sobre la estructura de árboles y la cantidad de unidades por capas.

- Pasos (1) y (2) de la sección 3.4.1.
- Se fijará la función de activación obtenida en la sección anterior.
- Se fijará el optimizador seleccionado en la sección anterior.
- Se ejecutarán 4 redes utilizando las características a nivel de plan y apilando capas de convoluciones sobre árboles de la siguiente forma:
  - Red con 4 capas convolucionales apiladas: [1024, 512, 256, 128]
  - Red con 3 capas convolucionales apiladas: [512, 256, 128]
  - Red con 2 capas convolucionales apiladas: [256, 128]
  - Red con 1 capa convolucional: [256]

En la Figura 3.10, se observa como los modelos con capas de profundidad 3 y 4, obtienen mejores desempeños en cuanto a las métricas en los conjuntos de entrenamiento y validación. Específicamente, el modelo de 3 capas con 512, 256 y 128 unidades en las capas convolucionales de árboles presenta los mejores desempeños.

3.4.4. Selección características a nivel de consulta

En la sección 3.1.1, se propusieron técnicas de extracción de características a nivel de consulta. A continuación exploraremos la efectividad de extender la codificación a nivel de plan con estas:

- Pasos (1) y (2) de la sección 3.4.1.
- Se fijará la función de activación obtenida en la sección anterior.
- Se fijará el optimizador seleccionado en la sección anterior.
- Se fijarán las capas a nivel de plan con 512, 256, 128 unidades por capas.

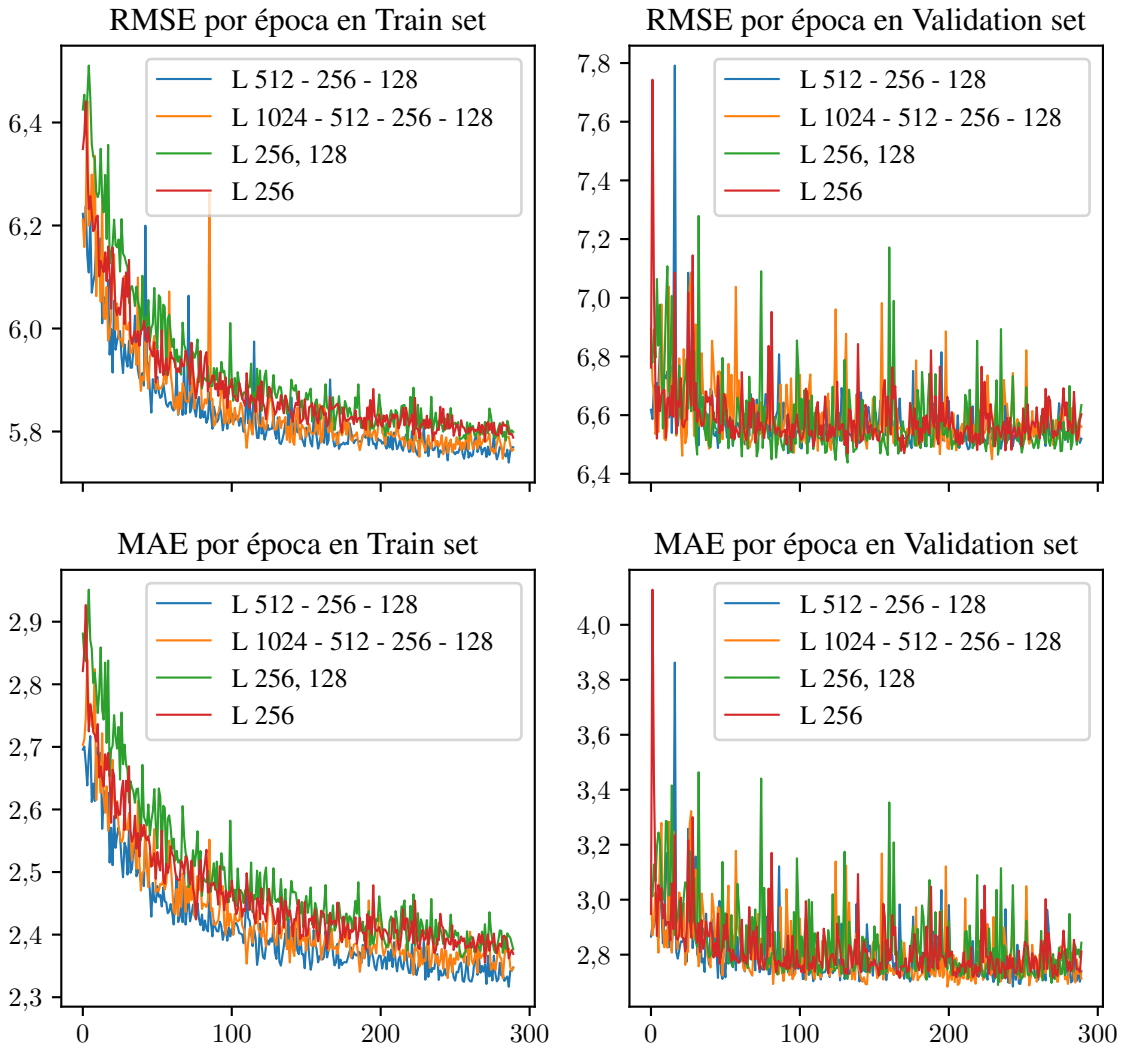


Figura 3.10: Entrenamiento a 300 épocas (datos de la 10 en adelante) de arquitecturas con distintos niveles de profundidad y unidades por capa.

- Se ejecutarán 3 redes utilizando las características a nivel de consulta siguientes para extender las características a nivel de plan en la primera capa nivel de plan:

    1. Red con 3 capas densas que recibe como entrada características algebraicas y de patrones de grafo.
    2. Red con 3 capas densas que recibe como entrada características de filtros y cardinalidades por predicado.
    3. Red con 3 capas densas que recibe como entrada características algebraicas, patrones de grafo, filtros y cardinalidades de predicados.

En la Figura 3.11, se observa el desempeño de las codificaciones a nivel de consulta. El desempeño de la red con características extraídas de los filtros y las cardinalidades estáticas obtenidas del optimizador de Jena no resultaron muy efectivas por si solas. Sin embargo, se observa que la combinación de estas con las características del álgebra y patrones de grafos obtienen los mejores resultados. El uso de características a nivel de consulta mejora estabilidad para el conjunto de validación, respecto a los resultados obtenidos del uso de información a nivel de plan.

3.4.5. Efectividad del uso de autoencoders

El la sección 3.3.3, se propuso un autoencoder como técnica de preentrenamiento para reducir la dimensionalidad en los vectores de los nodos del plan de consulta.

A continuación se agregan algunos detalles del entrenamiento del autoencoder:

- La entrada y salida de la red corresponde a vectores de tamaño fijo, teniendo en cuenta la información de los nodos de todos los planes que forman el dataset para el conjunto de *training*.
- Se propone 1 capa oculta para el encoder y 1 para el decoder.
- Las unidades por capa se seleccionarán de manera que la última capa del 'encoder' resuma al menos 4 veces la cantidad de características de entrada.
- Se utilizará Adagrad como optimizador con un learning rate en el orden de 0.0001.

A continuación exploramos la efectividad del preentrenamiento con autoencoder para resumir la información a nivel de plan. La arquitectura probada tiene las siguientes características:

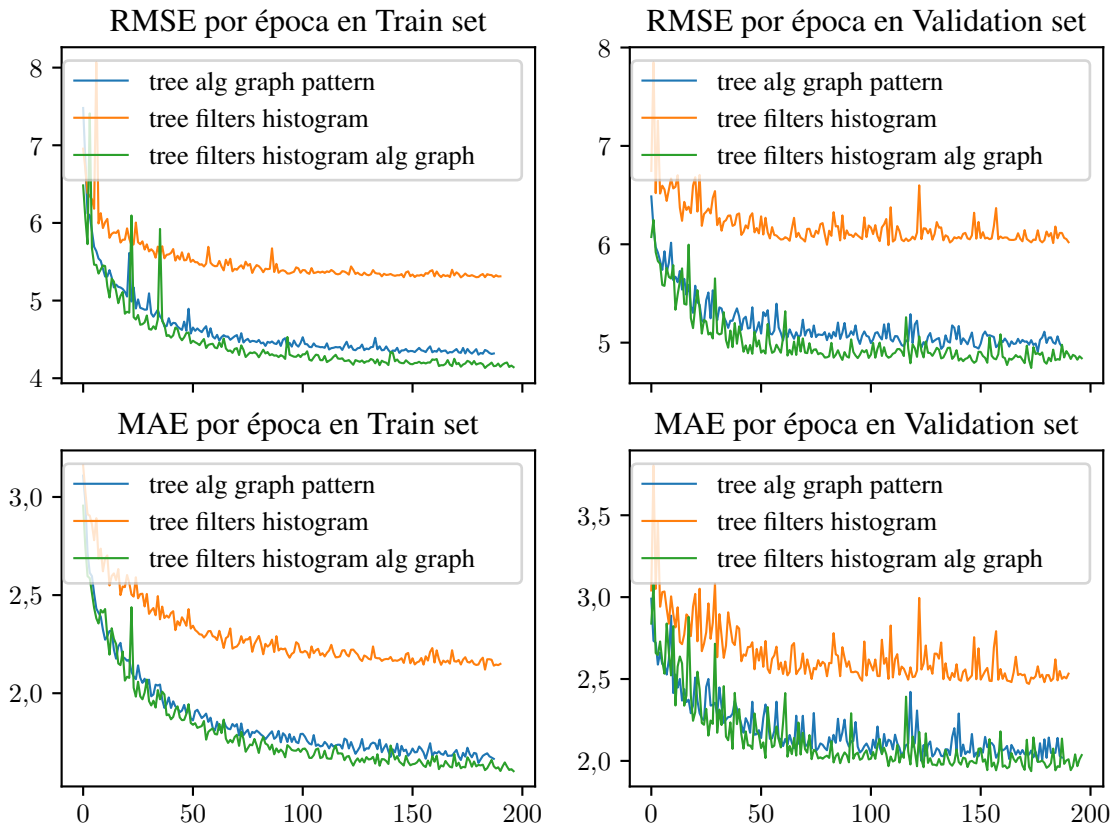


Figura 3.11: Entrenamiento a 300 épocas con EarlyStopping de arquitecturas con distinta información a nivel de consulta y 3 capas de convoluciones sobre árboles fijas.

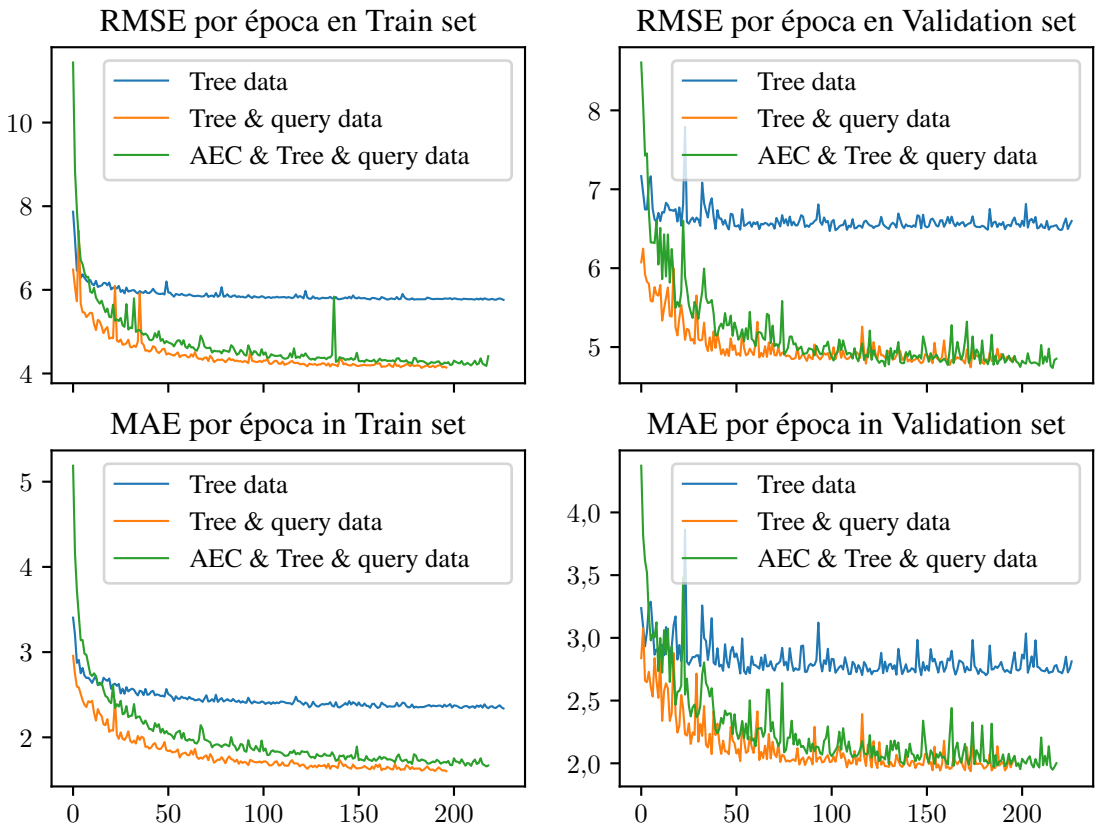


Figura 3.12: Comparación del desempeño de la arquitectura utilizando datos preprocesados o no con el autoencoder propuesto.

- Pasos (1) y (2) de la sección 3.4.1.
- Se fijará la función de activación LeakyReLU.
- Se fijará el optimizador seleccionado en la sección un learning rate de 0.00015.
- Se fijarán las capas a nivel de plan con 512, 256, 128 unidades por capas.
- Se ejecutará 1 red utilizando la mejor configuración de características a nivel de consultas obtenidas en la sección anterior y las características a nivel de plan que serán previamente preprocesadas por las capas del 'encoder' extraídas del autoencoder.

La Figura 3.12, muestra el desempeño en el conjunto de entrenamiento y validación de: una red con información a nivel de planes de ejecución; una red con información a nivel de consulta y de planes de ejecución; y una red similar a la anterior con los datos a nivel del nodos del plan previamente preprocesados por el autoencoder. Se observa como el uso del autoencoder no introduce mejoras significativas en cuanto a la generalización del model. Sin embargo, si tiene un impacto positivo en el tiempo de entrenamiento y los recursos de hardware necesarios para entrenar el modelo. Esto se debe a que disminuye significativamente la cantidad de parámetros de la red.

3.4.6. Ajuste de hiperparámetros de la línea base

En la siguiente sección realizaremos una comparación de la arquitectura propuesta respecto a las propuestas del estado del arte. En concreto se realiza la comparación respecto a la NuSVR propuesta en [27] y una red neuronal densa similar a la propuesta para las características a nivel de consulta. A continuación se especifican los detalles de los modelos de la linea base y el ajuste de hiperparámetros realizado.

Ajuste de hiperparámetros de la NuSVR

La Máquina de Soporte Vectorial con kernel NuSVR propuesta en [27], utiliza las características Algebraicas y patrones de grafos especificados en las secciones 3.1.1 y 3.1.1 respectivamente. En este caso es necesario ajustar los hiperparámetros:

- Nu: límite superior en la fracción de errores de entrenamiento y un límite inferior de la fracción de vectores de soporte. Debe estar en el intervalo (0,1].

C	nu	mse	rmse	mae	mseval	rmseval	maeval
260	0.25	107.056	10.347	5.745	112.255	10.595	5.913
260	0.3	105.753	10.284	5.549	114.383	10.695	5.739
220	0.35	105.309	10.262	5.363	115.603	10.752	5.583
220	0.35	105.309	10.262	5.363	115.603	10.752	5.583
200	0.4	105.193	10.256	5.214	115.687	10.756	5.465
240	0.35	105.291	10.261	5.359	115.929	10.767	5.581
280	0.2	111.243	10.547	5.999	116.067	10.773	6.154
220	0.4	105.227	10.258	5.211	116.183	10.778	5.466
220	0.2	111.343	10.552	6.004	116.259	10.782	6.161
220	0.15	119.526	10.933	6.378	124.364	11.151	6.512

Tabla 3.1: Ajuste de hiperparámetros del modelo NuSVR.

- C: Parámetro de penalización C del término de error.

Algoritmo 1 Selección de hiperparámetros Nu y C de la NuSVR

procedure TESTHYPERPARAMSNUSVR($x\_train, y\_train, x_val, y\_val, runs$)
    $results \leftarrow []$
    for `i =0; i <runs; i++` **do**
        $C \leftarrow random(start = 200; stop = 300; step = 20)$ ▷ Int en rango.
        $nu \leftarrow random(start = 0{,}1; stop = 0{,}5; step = 0{,}05)$ ▷ Float en rango.
        $model \leftarrow NuSVR(nu, C))$
        $model.fit(train\_data))$
        $predictions \leftarrow model.predict(x\_val)$
        $stats \leftarrow rmse(preditions, y\_val)$
        $results \leftarrow results.append([C, nu, stats])$
    return $hparams\_min(results)$ ▷ Obtener C y nu de modelo con min RMSE registrado.

El Algoritmo 1, muestra el procedimiento para la selección de los hiperparámetros de la NuSVR. El procedimiento realiza 10 iteraciones seleccionando valores aleatorios acotados en determinados rangos de los hiperparámetros C y nu. Luego seleccionamos la combinación de hiperparámetros que mejor rendimiento obtenga en el conjunto de validación. La tabla 3.1 muestra como el menor RMSE se obtiene utilizando un valor $C = 260$ con un $nu = 0{,}25$, para los cuales se obtiene un RMSE de 10.595 en el conjunto de validación.

Ajuste de hiperparámetros del modelo neuronal denso

La segunda arquitectura propuesta como línea base corresponde a un modelo neuronal como capas densas con las siguiente características:

- Utilizar como vectores de entrada las codificaciones detalladas en 3.1.1,3.1.1 y 3.1.2
- Modelo de 3 capas ocultas con activaciones ReLU.
- Luego de las activaciones en las capas L1 y L2, y L3 se aplicó una capa de Dropout con una probabilidad de abandono de 0.25.
- La capa de salida contiene 1 sola unidad lineal que corresponde a la predicción de la red.

Algoritmo 2 Selección de hiperparámetros L1, L2, L3 del modelo neuronal denso

procedure TESTHYPERPARAMSDENSEMODEL($x\_train, y\_train, x_val, y\_val, runs$)
    $results \leftarrow []$
    $L1\_list \leftarrow range(200, 800, 120)$ ▷ Lista elementos en rango con paso de 120.
    $L2\_list \leftarrow range(200, 600, 100)$ ▷ Lista elementos en rango con paso de 100.
    $L3\_list \leftarrow range(100, 400, 100)$ ▷ Lista elementos en rango con paso de 100.
    for `L1 in L1_list` **do**
        for `L2 in L2_list` **do**
            for `L3 in L3_list` **do**
                $model \leftarrow DenseModel(L1, L2, L3))$
                $model.fit(train\_data))$
                $predictions \leftarrow model.predict(x\_val)$
                $stats \leftarrow rmse(preditions, y\_val)$
                $results \leftarrow results.append([L1, L2, L3, stats])$
    return $hparams\_min(results)$ ▷ Obtener unidades por capa con min RMSE registrado.

El Algoritmo 2, muestra el procedimiento para la selección de los hiperparámetros del modelo denso correspondiente a la segunda linea base(LB_DENSEMODEL). El procedimiento realiza una grilla sobre los rangos de unidades para las 3 capas ocultas del modelo. Luego seleccionamos la combinación de unidades por capa que mejor rendimiento obtenga en el conjunto de validación. La tabla 3.2 muestra como el menor RMSE se obtiene utilizando $L1 = 440$, $L2 = 300$ y $L3 = 400$, para los cuales se obtiene un RMSE de $10{,}595$ en el conjunto de validación.

L1	L2	L3	maeval	mseval	rmseval
440.0	300.0	400.0	5.35	104.23	**10.21**
440.0	500.0	300.0	5.37	105.52	10.27
680.0	500.0	300.0	5.38	105.72	10.28
560.0	500.0	400.0	5.39	106.15	10.30
320.0	500.0	300.0	5.39	106.80	10.33
560.0	300.0	400.0	5.39	106.86	10.34
320.0	300.0	400.0	5.39	106.89	10.34
680.0	600.0	400.0	5.41	106.92	10.34
440.0	400.0	400.0	5.40	107.05	10.35
320.0	600.0	400.0	5.41	107.25	10.36

Tabla 3.2: Ajuste de hiperparámetros del modelo neuronal denso.

Capítulo 4

Resultados y discusión

En este capítulo presentaremos el proceso que fue llevado a cabo para validar la arquitectura y comprobar si la hipótesis y objetivos planteados fueron alcanzados.

La sección 4.1 presenta detalladamente el proceso de validación seguido con la arquitectura propuesta, en particular se explican las métricas de evaluación elegidas y se introducen los resultados esperados. La sección 4.2 profundiza en los resultados obtenidos y efectúa un análisis estadístico de los mismos. En dicha sección se muestran los datos que acompañan los resultados más importantes del trabajo y que son la base para las conclusiones presentadas en la sección 5.

4.1. Métricas de evaluación

La métricas a utilizar para validar la propuesta fueron seleccionadas teniendo en cuenta la naturaleza de la tarea de aprendizaje y las métricas utilizadas en investigaciones anteriores similares.

El error cuadrático medio (MSE) es una métrica común para comparar estimadores en tareas de regresión. También es usualmente utilizada como función de pérdida en modelos de redes neuronales. Como función de pérdida minimiza las diferencias cuadráticas entre el valor estimado y el valor real. Una variante del MSE calcula la raíz cuadrada del error cuadrático medio (RMSE). Esta es una de las métricas propuestas en el trabajo de Hasan y Gandon [27]. Una característica es que tiende a penalizar los errores muy grandes.

$$MSE_{(test)} = \frac{1}{m}\sum_{i=1}^{m}(y_i^{(test)} - \hat{y}_i^{(test)})^2, \tag{4.1}$$

$$RMSE_{(test)} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(y_i^{(test)} - \hat{y}_i^{(test)})^2}. \tag{4.2}$$

Otra métrica de error y función de pérdida muy común en problemas de regresión es el Error Absoluto Medio (MAE). Como función de pérdida minimiza las diferencias absolutas entre el valor estimado y el valor real. Esta tiende a ser más robusta a valores atípicos que el MSE.

$$MAE_{(test)} = \frac{1}{m}\sum |y_i^{(test)} - \hat{y}_i^{(test)}|. \tag{4.3}$$

En las métricas anteriores es necesario destacar el superíndice $test$. Este indica que para evaluar la generalización de un algoritmo de aprendizaje estas deben ser aplicadas sobre el conjunto de datos de *test*. A diferencia de los conjuntos de entrenamiento y validación, el conjunto de test debe estar totalmente aislado del proceso de entrenamiento y selección de hiperparámetros.

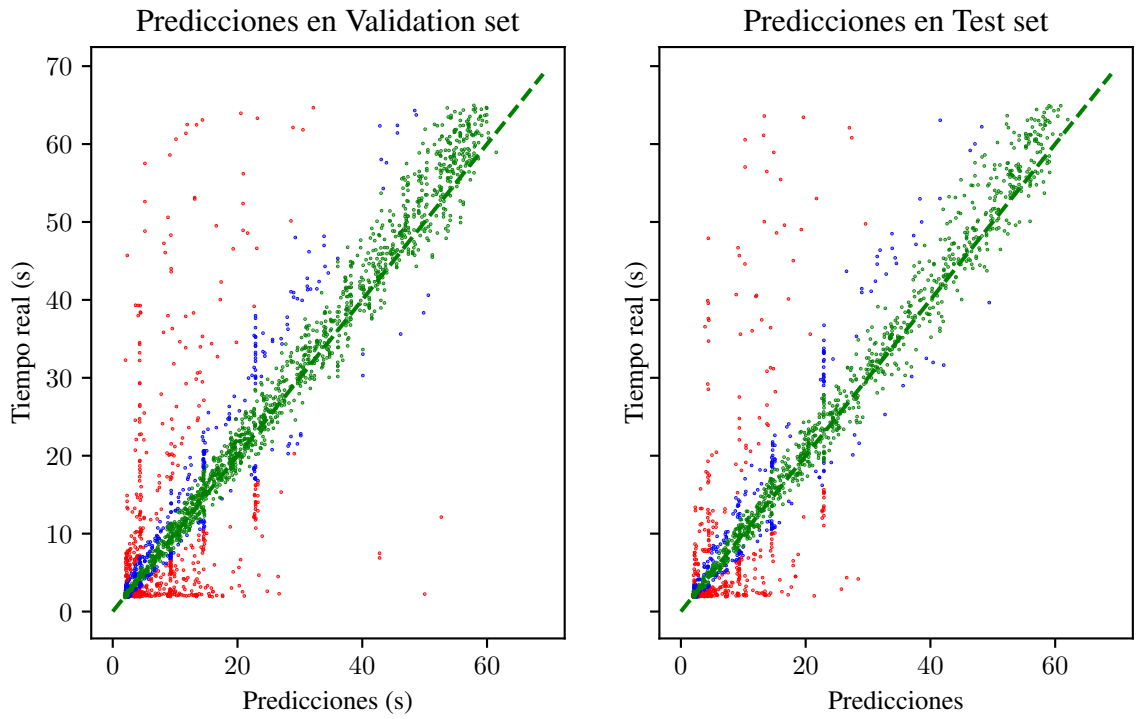


Figura 4.1: Predicciones de la arquitectura propuesta para los conjuntos de validación y test.

La Figura 4.1, visualiza las predicciones realizadas utilizando la arquitectura propuesta para los conjuntos de validación (izquierda) y test (derecha). Cada punto representa una consulta en el conjunto de datos. El valor correspondiente a cada punto en el eje horizontal corresponde con la predicción realizada en la escala de segundos y su valor en el eje vertical corresponde al tiempo de ejecución real.

- Los puntos en color verde representan consultas cuya predicción se considera buena: $(|y - \hat{y}| <= y * 0{,}2)$.
- Los puntos en color azul representan consultas cuya predicción se considera aceptable $y * 0{,}2 < (|y - \hat{y}| <= y * 0{,}4)$.
- El resto de puntos rojos corresponde con predicciones malas.

4.2. Validación mediante prueba de hipótesis

En la sección anterior concluimos aportando evidencias sobre la superioridad de la propuesta de solución respecto a los modelos de la línea base. Sin embargo, es necesario validar que la mejora obtenida se mantiene para distintos conjuntos de datos y no es fruto del azar. A continuación se detalla la aplicación y resultados de aplicar la 'Prueba t Student para muestras dependientes' con el objetivo de garantizar una comparación estadísticamente significativa.

4.2.1. Prueba t Student para muestras dependientes

La prueba t Student para muestras dependientes o emparejadas, se utiliza para comparar las medias de una muestra o de dos muestras emparejadas o dependientes. En el caso de utilizar solo una muestra, la prueba se realiza comparando las medias resultantes dos *eventos* que ocurren sobre la misma muestra. De manera análoga, es posible aplicar este test comparando las medias de dos eventos que ocurren sobre dos muestras, siempre y cuando se garantice que dichas muestras son dependientes. El objetivo es verificar si los registros del evento 1 cambian significativamente respecto a los del evento 2.

Concretamente en nuestro caso, aplicamos la prueba t sobre muestras dependientes para verificar si la diferencia de medias entre los resultados obtenidos con la solución

propuesta y los resultados de cada uno de los modelos de la línea base) es estadísticamente significativa. En total se aplicará el test entre los 3 pares de modelos siguientes:

- Test aplicado entre el modelo de convoluciones sobre árboles (TreeConvModel) y el modelo neuronal de capas densas (DenseModel 3.4.6) utilizando la métrica RMSE.
- Test aplicado entre el modelo de convoluciones sobre árboles (TreeConvModel) y el modelo NuSVR (NuSVR 3.4.6) utilizando la métrica RMSE.
- Test aplicado entre el modelo DenseModel y el modelo NuSVR utilizando la métrica RMSE.

A continuación detallamos el procedimiento:

1. Utilizar las consultas extraídas y procesadas de la base de datos Wikidata (sección 3.2).
2. Seleccionar 5 subconjuntos aleatorios del 85 % de los datos de entrenamiento separados en la sección 3.2, el otro 15 % se utilizará como conjunto de validación.
3. Entrenar el modelo propuesto TreeConvModel, DenseModel 3.4.6) y el modelo NuSVR 3.4.6 sobre cada subconjunto de entrenamiento del paso (1).
4. Evaluar cada modelo sobre el conjunto de *test* (separado en la sección 3.2) obteniendo en cada caso el RMSE.

	Tree_Conv		Dense_model		NuSvr	
	RMSE	MAE	RMSE	MAE	RMSE	MAE
1	5.257	2.068	10.354	5.403	17.787	5.912
2	5.020	2.007	10.409	5.424	15.196	5.799
3	4.967	2.008	10.337	5.361	15.914	5.861
4	4.911	1.989	10.364	5.363	12.192	5.670
5	5.148	2.185	10.410	5.403	11.823	5.645
Media	5.06	2.05	10.37	5.39	14.58	5.78
Std	0.13	0.07	0.03	0.02	2.27	0.1

Tabla 4.1: Valores de RMSE y MAE para los 3 modelos en los 5 particionamientos para el conjunto de *test*.

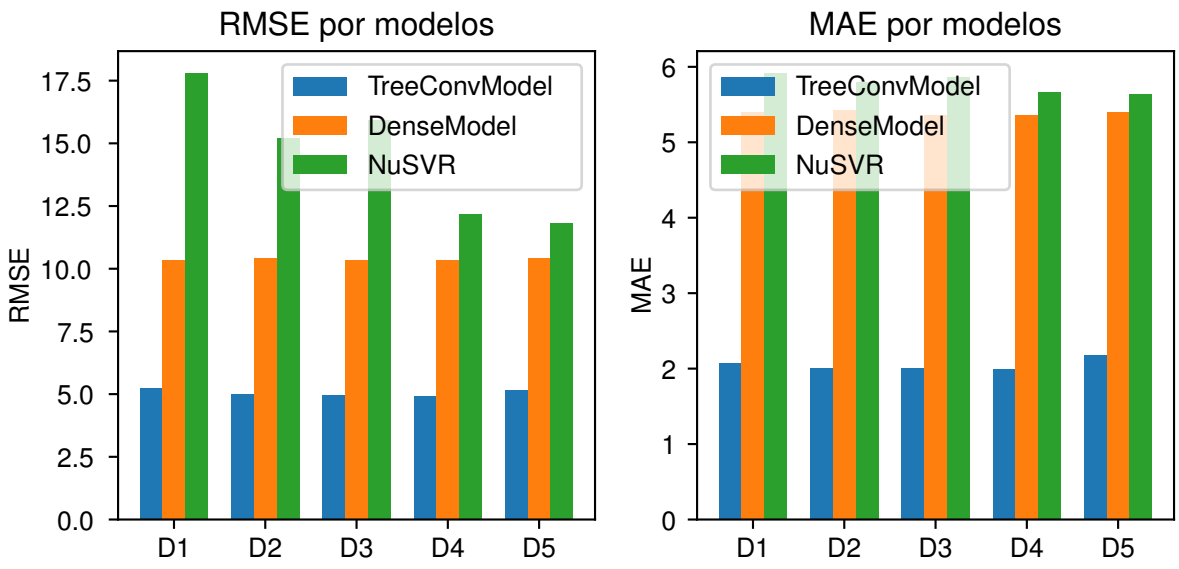


Figura 4.2: Valores de Root Mean Squared Error (gráfica a la izquierda) y de Mean Absolute Error (derecha) obtenidos luego de aplicar los modelos en análisis sobre los 5 particionamientos aleatorios y evaluar cada modelo en el conjunto de Test.

En la figura 4.2 se observan los resultados obtenidos para la métrica RMSE en cada uno de los particionamientos realizados para la prueba de hipótesis. Se puede observar como el modelo propuesto mejora ampliamente los modelos de la línea base en cuanto a la métrica RMSE. Sin embargo, es necesario determinar si la diferencia en las medias sobre la muestra (5 subconjuntos aleatorios) para cada modelo representa una verdadera diferencia en la población de la que proviene la muestra.

Para hacer esta comparación, se realiza la prueba t utilizando la diferencia entre las dos medias y dividiendo por el error estándar de la diferencia entre las dos medias muestrales dependientes:

$$t = \frac{\bar{A} - \bar{B}}{S_{\bar{D}}}, \tag{4.4}$$

donde $\bar{A}$ es la media de los RMSE para la propuesta de solución, $\bar{B}$ la media de los RMSE para la NuSVR, y $S_{\hat{D}}$ el error estándar de las diferencias entre ambas medias.

El error estándar se obtiene:

$$S_{\bar{D}} = \frac{S_D}{\sqrt{N}}, \tag{4.5}$$

$$S_D = \sqrt{\frac{\sum D^2 - \frac{(\sum D)^2}{N}}{N-1}}, \tag{4.6}$$

donde: $S_{\bar{D}}$ es el error estándar de las diferencias entre las medias muestrales, S_D representa la desviación estándar de las diferencias de medias (para una muestra poblacional), D es la diferencia entre los RMSE de cada par de las muestras para los eventos A y B, N es la cantidad de individuos en la muestra (5 en este caso).

Dado lo anterior, el valor t observado puede ser utilizado para contrastar la siguiente hipótesis:

$$\begin{aligned} H_0 &: \mu_D = 0. \\ H_A &: \mu_D > 0, \end{aligned} \tag{4.7}$$

donde μ_D representa la diferencia de las medias muestrales. La hipótesis H_0 plantea que no existe diferencia entre las medias de los rendimientos entre los modelos, y se rechaza en favor de $H_A :> 0$ cuando $|t| > t_\alpha$, siendo t_α el valor crítico se obtiene de una distribución t Student para $N - 1 = 4$ grados de libertad. Por ejemplo, el t obtenido en la comparación del modelo propuesto con respecto a la NuSVR arroja un $t = -8{,}579828$ y para 4 grados de libertad y un $\alpha = 0{,}05$ obtenemos un valor crítico $t_\alpha = 2{,}776$, por lo que $|t| > t_\alpha$ y en consecuencia se rechaza la hipótesis H_0.

También es posible interpretar el test utilizando el *p-value*. Si el *p-value* $> \alpha$, entonces se acepta la hipótesis H_0 de que no existe diferencias entre las medias muestrales. En caso contrario, se rechaza en favor de la hipótesis alternativas H_A.

	TreeConvModel_DenseModel	TreeConvModel_NuSvr	DenseModel_NuSvr
t_α	2.776445e+00	2.776445	2.776445
p-value	1.144593e-07	0.001014	0.021120
t	-8.506974e+01	-8.579828	-3.684497

Tabla 4.2: Resultados del estadístico t y p-value obtenidos por pares de modelos analizados. Por ejemplo, la columna 'TreeConv_DenseModel' contiene los resultados entre los pares de modelos TreeConvModel y DenseModel, obteniéndose para estos un t de $85{,}069$ y un p-value de $1{,}144593e-07$.

La Tabla 4.2, incluye los resultados de aplicar el procedimiento anterior sobre las combinaciones entre la propuesta de solución y los modelos de la línea base.

- TreeConvModel_DenseModel: un *p-value* con valor $1{,}144593e-07 < 0{,}05$ indica que se rechaza la hipótesis H_0, por lo que se puede afirmar que si existe una diferencia estadísticamente significativa entre los RMSE del modelo propuesto y el modelo neuronal denso.

- TreeConvModel_NuSvr: un *p-value* con valor $0{,}001014 < 0{,}05$, indica que se rechaza la hipótesis H_0, por lo que se puede afirmar que si existe una diferencia estadísticamente significativa entre los RMSE del modelo propuesto y el modelo neuronal denso.

- DenseModel_NuSVR: un *p-value* con valor $0{,}021120 < 0{,}05$, indica que se rechaza la hipótesis H_0, por lo que se puede afirmar que si existe una diferencia estadísticamente significativa entre los RMSE el modelo neuronal denso.

Dado que si existe una diferencia significativa entre los rendimientos de los modelos en análisis, podemos afirmar que la superioridad del modelo propuesto respecto a las líneas base para el dataset analizado son significativas. A continuación agregamos las conclusiones finales de la investigación.

Capítulo 5

Conclusiones y Trabajo Futuro

Con la propuesta realizada hemos cumplido tanto el objetivo general como los objetivos específicos propuestos para esta investigación. Los resultados obtenidos sustentan la efectividad del uso de características a nivel de planes de ejecución de las consultas SPARQL. Lo anterior, conjuntamente con la aplicación de técnicas de convoluciones sobre árboles en una arquitectura de redes neuronales profundas, superaron los rendimientos de los modelos de la línea base en la predicción de rendimiento en consultas SPARQL.

5.1. Conclusiones

A partir de los resultados presentados, se puede concluir:

- Respecto a los modelos analizados, los resultados mostraron la superioridad de las arquitecturas de redes neuronales sobre la NuSVR utilizada en otras propuestas del estado del arte para la predicción de rendimiento de consultas SPARQL. Tanto el modelo neuronal denso, como la arquitectura propuesta de convoluciones sobre árboles presentaron mejor rendimiento y estabilidad en el entrenamiento respecto a las métricas analizadas. Sin embargo, el uso de las convoluciones de árboles para extraer patrones de los planes de ejecución de las consultas resultaron mejorar significativamente los modelos que solo utilizan características estructurales de la consulta.

- Un reto no menor para el cumplimiento de los objetivos propuestos, lo constituye la falta de disponibilidad de conjuntos de datos reutilizables. Las fuentes de datos

identificadas no satisfacían requerimientos fundamentales como:

  - Registros de los tiempos de ejecución para cada consulta en un entorno controlado.
  - Conjunto de consultas con un balance adecuado respecto al tiempo de ejecución asociado.
  - Disponibilidad del plan de ejecución asociado a cada consulta.

- Dadas las limitaciones anteriores, fue necesario generar un dataset nuevo conformado por registros de consultas reales y otras generadas. Las consultas se ejecutaron sobre la base de datos de Wikidata (versión 10-2020) en un entorno local y cuenta con más de 24600 consultas SPARQL. Este dataset es uno de los resultados obtenidos en esta investigación y quedará disponible para uso de la comunidad en futuras investigaciones en **??**.

- Respecto a las características utilizadas, los resultados mostraron la efectividad de codificar las operaciones de joins de los planes de ejecución de las consultas utilizando árboles binarios. Los mejores resultados sin embargo, se obtienen incluyendo además características generales de las consultas como las características algebraicas y patrones de grafos.

- El proceso de selección de características realizado permitió el ajuste de algunos hiperparámetros necesarios para evidenciar la efectividad de la extracción de características y la arquitectura propuesta. Sin embargo, se recomienda realizar un proceso más exhaustivo para el uso en un entorno real con el objetivo de mejorar aun más las predicciones.

Con respecto a los objetivos propuestos, se puede concluir que:

- Se generó un dataset de consultas asociadas a la base de datos de grafos Wikidata lo suficientemente grande para entrenar y evaluar los modelos de aprendizaje.

- Se definieron e implementaron las técnicas para la extracción de características (disponibles en **??**) para las consultas SPARQL. Las técnicas implementadas incluyen características a nivel de planes de ejecución: operaciones de joins, tipos de triples, predicados y características a nivel de consulta como: características algebraicas, patrones de grafos, histogramas y cardinalidades.

- Se definió e implementó una arquitectura de redes neuronales profunda que incluye capas convoluciones sobre árboles para la información a nivel de plan, y capas densas para las características a nivel de consulta. La arquitectura propuesta supera los modelos del estado del arte identificados para la tarea de predicción de rendimiento en consultas SPARQL.

- Se evaluó la efectividad del modelo utilizando métricas acordes a la tarea como RMSE y MAE. Finalmente se validó la solución a partir de una prueba de hipótesis de diferencias de medias pareadas, lo que nos permitió asegurar que la propuesta supera los modelos del estado del arte en el dataset analizado.

5.2. Trabajo Futuro

Los resultados obtenidos en esta investigación dejan abiertos algunos caminos para futuras investigaciones. Uno de los enfoques no explorados respecto a las técnicas de extracción de características es el uso de incrustaciones de la información interna de las bases de datos. Creemos que en el contexto de las bases de datos de grafos, el modelo propuesto podría mejorar su rendimiento si aprovechara incrustaciones de los predicados que se suceden en las operaciones de join.

Respecto al aprendizaje de estructuras más complejas como los árboles del plan de ejecución de las consultas, otros modelos como las 'Tree-Structured LSTM' han demostrado su efectividad en tareas del área de Procesamiento de Lenguaje Natural. En esta investigación no se cubrió este tipo de arquitecturas, por lo que sería interesante explorar la efectividad de estas en la predicción de rendimiento de consultas SPARQL.

Como mencionamos en la problemática planteada, la predicción de rendimiento es un paso necesario para abordar la optimización de sistemas y servicios de bases de datos de grafos. Por lo anterior, un primer paso natural a futuro es la implementación de un optimizador que utilice modelos como el propuesto para mejorar la selección de planes de ejecución.

Bibliografía

[1] "Cisco Annual Internet Report - Cisco Annual Internet Report (20182023) White Paper."

[2] D. Reinsel, J. Gantz, and J. Rydning, "The Digitization of the World from Edge to Core," p. 28.

[3] R. V. Guha, D. Brickley, and S. Macbeth, "Schema. org: Evolution of structured data on the web," vol. 59, no. 2, pp. 44–51.

[4] "Resource Description Framework (RDF) Model and Syntax Specification."

[5] "SPARQL Query Language for RDF."

[6] "Blazegraph Database."

[7] "Virtuoso Open Source."

[8] "Apache Jena - Apache Jena Fuseki."

[9] A. Abele, J. P. McCrae, P. Buitelaar, A. Jentzsch, and R. Cyganiak, "Linking open data cloud diagram 2017,"

[10] C. Ghidini, O. Hartig, M. Maleshkova, V. Svatek, I. Cruz, H. Aidan, J. Song, M. Lefrançois, and F. Gandon, *The Semantic Web-ISWC 2019*. Springer.

[11] R. Marcus and O. Papaemmanouil, "Towards a Hands-Free Query Optimizer through Deep Learning."

[12] W. Wang, M. Zhang, G. Chen, H. V. Jagadish, B. C. Ooi, and K.-L. Tan, "Database Meets Deep Learning: Challenges and Opportunities," vol. 45, no. 2, pp. 17–22, 2016-09-28, 2016.

[13] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul, "Neo: A learned query optimizer," vol. 12, no. 11, pp. 1705–1718.

[14] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?," vol. 9, no. 3, pp. 204–215.

[15] C. Buil-Aranda, A. Hogan, J. Umbrich, and P.-Y. Vandenbussche, "Sparql web-querying infrastructure: Ready for action?," in *International Semantic Web Conference*, pp. 277–293, Springer.

[16] H. Liu, M. Xu, Z. Yu, V. Corvinelli, and C. Zuzarte, "Cardinality estimation using neural networks," in *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*, pp. 53–59.

[17] A. Kipf, T. Kipf, B. Radke, V. Leis, P. Boncz, and A. Kemper, "Learned Cardinalities: Estimating Correlated Joins with Deep Learning."

[18] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi, "Learning State Representations for Query Optimization with Deep Reinforcement Learning."

[19] Y. Park, S. Zhong, and B. Mozafari, "Quicksel: Quick selectivity learning with mixture models," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pp. 1017–1033.

[20] S. Krishnan, Z. Yang, K. Goldberg, J. Hellerstein, and I. Stoica, "Learning to Optimize Join Queries With Deep Reinforcement Learning," 2019-01-10, 2019.

[21] R. Marcus and O. Papaemmanouil, "Deep Reinforcement Learning for Join Order Enumeration," in *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management - aiDM'18*, pp. 1–4, ACM Press.

[22] R. Marcus and O. Papaemmanouil, "Plan-Structured Deep Neural Network Models for Query Performance Prediction,"

[23] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI16, pp. 1287–1293, AAAI Press.

[24] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, and T. Zhang, "Self-driving database management systems," in *CIDR 2017, Conference on Innovative Data Systems Research*.

[25] J. Duggan, U. Cetintemel, O. Papaemmanouil, and E. Upfal, "Performance prediction for concurrent database workloads," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pp. 337–348, Association for Computing Machinery.

[26] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, and S. B. Zdonik, "Learning-based Query Performance Modeling and Prediction," in *2012 IEEE 28th International Conference on Data Engineering*, pp. 390–401, IEEE.

[27] R. Hasan and F. Gandon, "A machine learning approach to SPARQL query performance prediction," in *2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT)*, vol. 1, pp. 266–273, IEEE.

[28] T. Berners-Lee, "Linked data-design issues (2006)," *URL* `http://www.w3.org/DesignIssues/LinkedData.html`, 2011. [Revisado el 08/01/2016].

[29] Tim Berners-Lee, "Semantic web architecture." URL `https://www.w3.org/2000/Talks/1206-xml2k-tbl/slide10-0.html`, 2014. [Revisado el 08/01/2016].

[30] A. Gangemi and V. Presutti, "The bourne identity of a web resource," in *Proceedings of Identity Reference and the Web Workshop (IRW) at the WWW Conference*.

[31] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform resource identifier (uri): Generic syntax," *URL* `http://www.rfc-editor.org/rfc/rfc3986.txt`, 2004.

[32] Bikakis, Tsinaraki, Gioldasis, Stavrakantonakis, and Christodoulakis, "The XML and semantic web worlds: Technologies, interoperability and integration. A survey of the state of the art," in *Semantic Hyper/Multi-Media Adaptation: Schemes and Applications*, Springer.

[33] D. Brickley and R. V. Guha, "Rdf schema 1.1," *W3C Recommendation PER-rdf-schema-20140109*, 2014.

[34] D. Beckett, T. Berners-Lee, E. Prudhommeaux, and G. Carothers, "Rdf 1.1 turtle–terse rdf triple language," *W3C Recommendation REC-turtle-20140225*, 2014.

[35] D. Beckett, "Rdf 1.1 n-triples," *World Wide Web Consortium, Recommendation REC-n-triples-20140225*, 2014.

[36] D. Beckett and B. McBride, "Rdf/xml syntax specification (revised)," *W3C Recommendation REC-rdf-syntax-grammar-20040210*, vol. 10, 2004.

[37] T. W. S. W. Group *et al.*, "Sparql 1.1 overview," *W3C Recommendation REC-sparql11-overview-20130321*, 2013.

[38] E. Prudhommeaux and A. Seaborne, "Sparql query language for rdf," 2008.

[39] J. Perez, M. Arenas, and C. Gutierrez, "Semantics and {{Complexity}} of {{SPARQL}}," p. 14.

[40] R. Angles, H. Thakkar, and D. Tomaszuk, "Mapping RDF databases to property graph databases," vol. 8, pp. 86091–86110.

[41] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor, "Cypher: An evolving query language for property graphs," in *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pp. 1433–1445, Association for Computing Machinery.

[42] J. L. Reutter and D. Srivastava, eds., *Proceedings of the 11th Alberto Mendelzon International Workshop on Foundations of Data Management and the Web, Montevideo, Uruguay, June 7-9, 2017*, vol. 1912 of *CEUR Workshop Proceedings*. CEUR-WS.org.

[43] M. Besta, E. Peter, R. Gerstenberger, M. Fischer, M. Podstawski, C. Barthels, G. Alonso, and T. Hoefler, "Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries," vol. abs/1910.09017.

[44] D. Kossmann, "The state of the art in distributed query processing," vol. 32, no. 4, pp. 422–469.

[45] C. Buil-Aranda, "Federated Query Processing for the Semantic Web."

[46] T. Mitchell, *Machine Learning*. Mcgraw-Hill International Edit, McGraw Hill Higher Education, 1st ed.

[47] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press.

[48] "Apache Jena - ARQ - A SPARQL Processor for Jena."

[49] W. E. Zhang, Q. Z. Sheng, Y. Qin, K. Taylor, and L. Yao, "Learning-based SPARQL query performance modeling and prediction," vol. 21, no. 4, pp. 1015–1035.

[50] M. H. Namaki, K. Sasani, Y. Wu, and A. H. Gebremedhin, "Performance prediction for graph queries," in *Proceedings of the 2nd International Workshop on Network Data Analytics*, NDA'17, Association for Computing Machinery.

[51] M. Zaheer, S. Kottur, S. Ravanbakhsh, B. Poczos, R. R. Salakhutdinov, and A. J. Smola, "Deep sets," in *Advances in Neural Information Processing Systems 30* (I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds.), pp. 3391–3401, Curran Associates, Inc.

[52] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, *et al.*, "Relational inductive biases, deep learning, and graph networks."

[53] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space."

[54] D. Duvenaud, D. Maclaurin, J. Aguilera-Iparraguirre, R. Gómez-Bombarelli, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams, "Convolutional networks on graphs for learning molecular fingerprints," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, pp. 2224–2232, MIT Press.

[55] W. Fan, Y. Ma, Q. Li, J. Wang, G. Cai, J. Tang, and D. Yin, "A graph neural network framework for social recommendations," pp. 1–1.

[56] K. S. Tai, R. Socher, and C. D. Manning, "Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks," in *Proceedings of the*

53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), pp. 1556–1566, Association for Computational Linguistics.

[57] H. Purwins, B. Li, T. Virtanen, J. Schlüter, S. Chang, and T. Sainath, "Deep learning for audio signal processing," vol. 13, no. 2, pp. 206–219.

[58] L. Kaufman and P. J. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*, vol. 344. John Wiley & Sons.

[59] A. Krizhevsky, V. Nair, and G. Hinton, "CIFAR-10 (Canadian Institute for Advanced Research),"

[60] Y. LeCun, C. Cortes, and C. Burges, "MNIST handwritten digit database," vol. 2.

[61] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR09*.

[62] M. Saleem, M. I. Ali, A. Hogan, Q. Mehmood, and A.-C. N. Ngomo, "LSQ: The linked SPARQL queries dataset," in *The Semantic Web - ISWC 2015* (M. Arenas, O. Corcho, E. Simperl, M. Strohmaier, M. d' Aquin, K. Srinivas, P. Groth, M. Dumontier, J. Heflin, K. Thirunarayan, and S. Staab, eds.), pp. 261–269, Springer International Publishing.

[63] A. Hogan, C. Riveros, C. Rojas, and A. Soto, "A worst-case optimal join algorithm for SPARQL," in *The Semantic Web ISWC 2019* (C. Ghidini, O. Hartig, M. Maleshkova, V. Svátek, I. Cruz, A. Hogan, J. Song, M. Lefrançois, and F. Gandon, eds.), pp. 258–275, Springer International Publishing.

[64] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi, "A survey of deep neural network architectures and their applications," vol. 234, pp. 11–26.

[65] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pp. 315–323.

[66] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3431–3440.