Semantic Web 0 (2023) 1–0
IOS Press

SPARQL query execution time prediction using Deep Learning

Daniel Casals [a], Carlos Buil-Aranda [a,*], Carlos Valle [b]
[a] *IMFD; Informatics Department, Universidad Técnica Federico Santa María, Chile*
[b] *Departamento de Ciencia de Datos e Informática, Universidad de Playa Ancha*

Abstract. Nowadays new graph-oriented database engines (GDBMS) are capable of managing large volumes of graph data. These engines, like any other database engine, have an internal query optimizer that proposes a query execution plan (QEP) for each query they process. This step is entirely necessary since without it queries to large amounts of data may need hours to terminate. Selecting a "good enough" query plan is a difficult task as there may be thousands of possible query plans to consider just for a single query. The development of a good query planner is considered the "Holy Grail" in database management systems.

Recent results show how neural network models can help in the optimization process in Relational Database Management System (RDBMS), however, performance prediction in the area of graph databases is still in its infancy. There are two challenges in predicting graph query performance using machine learning models, yet to be addressed: the first is to find the most suitable query representation in a format that can be interpreted by Machine Learning (ML) algorithms; the second (which depends on the first challenge), is the learning algorithm used to predict performance. Herein we propose a method that actually fills in such a gap, by providing first a query characterization that (second) allows us to develop a deep learning model to estimate query latency later on. We adapt an existing model using convolutions over trees to generate a model that SPARQL queries and evaluate it against Wikidata, using their query logs and data. Our model correctly predicts the latency for queries in 87% of the validation set and we provide an explanation to those queries that we fail to predict the latency.

Keywords: Tree convolution, database optimization, query latency prediction, graph databases

1. Introduction

Nowadays new graph-oriented database engines (GDBMS) are capable of managing large volumes of graph data. Commercial solutions (from Oracle, Amazon Neptune or Neo4J), as well as open source solutions like Virtuoso [1] or Apache Jena Fuseki [2] or the recent MillenniumDB [24], are examples of them. These engines, like any other database engine, have an internal query optimizer that proposes a query execution plan (QEP) for each query they process. This step is entirely necessary since without it queries to large amounts of data may need hours to terminate (query complexity is at its best linear in terms of the data queried). Selecting a "good enough" query plan is a difficult task as there may be thousands of possible query plans to consider just for a single query [12]. The development of a good query planner is considered the "Holy Grail" in database management systems.

Recent results show how neural network models can help in the optimization process in Relational Database Management System (RDBMS). These models estimate intermediate results from query executions for later use within the query optimizer [9], help predicting database loads [6] or can replace conventional query optimizers within relational database engines, such as in [13, 15], predicting the entire query plan latency.

As stated in [19] "latency is the most important metric in a DBMS, as it captures all aspects of performance". Query performance estimation is also of interest for workload allocation in more complex systems such as cloud database services and the growing in-

*Corresponding author: cbuil@inf.utfsm.cl
[1] Virtuoso Open Source Edition: http://vos.openlinksw.com/owiki/wiki/VOS.
[2] Apache Jena Fuseki: https://jena.apache.org/documentation/fuseki2/.

terest in implementing concepts such as "Self-Driving Database Management Systems" [19]. In general, latency is used as a metric to meet Quality-of-Service objectives [3, 6].

In contrast to the DBMS area, performance prediction in the area of graph databases is still in its infancy. For query prediction in graph databases, the latest results come from the use of machine learning techniques such as SVRs or clustering algorithms such as k-nearest neighbor (K-nn) [7]. There are two challenges in predicting graph query performance using machine learning models. The first is to find the most suitable query representation in a format that can be interpreted by Machine Learning (ML) algorithms. The second (which depends on the first challenge), is the learning algorithm used to predict performance.

To the best of our knowledge, there is no solution using Deep Learning techniques for latency prediction of SPARQL queries on graph databases. Finally, the gap in performance prediction hinders progress in optimizing SPARQL queries and other applications mentioned above.

Herein we propose a method that actually fills in such a gap, by providing first a query characterisation that allows us to develop a deep learning model to estimate query latency later on.

The rest of the paper is organized as follows: Section 2.1 introduces the SPARQL and RDF concepts necessary in the paper while Section~ 3 introduces the related work so far, focusing especially on Database Query Processing. Section 4 presents the Deep Learning architecture we propose for solving the SPARQL query latency prediction problem and in Section 5 we evaluate the proposal. In Section 6 discuss the results we obtained and finally, we conclude in 7. We also provide the code and the data for our model and experiments in the Github companion repository[3].

This work is the extension of the research presented in [4]. The main differences are:

- A new Related Work section, which presents to the reader the state of the art in using Machine Learning techniques in database systems optimization.
- A new Architecture Section that introduces our new model for SPARQL query latency prediction, which is based on Tree Convolution Deep Neural Networks [18] and Autoencoders. This approach is totally different from our previous work and is the core of the current contribution.
- A new Experiments Section, in which we use data from the Wikidata [23] dataset and query logs.

[3] https://github.com/cbuil/sparql-latency-prediction

2. Preliminaries

In this Section we introduce some brief preliminaries for RDF, SPARQL and Database Query Processing used throughout.

2.1. RDF and SPARQL

RDF RDF is the graph-based data model at the heart of the Semantic Web. RDF terms can be IRIs (**I**), literals (**L**) or blank nodes (**B**). A triple $(s, p, o) \in \mathbf{I} \cup \mathbf{B} \times \mathbf{I} \times \mathbf{I} \cup \mathbf{B} \cup \mathbf{L}$ is called an RDF triple, where s is called the subject, p the predicate, and o the object. An RDF graph is a set of RDF triples.

SPARQL SPARQL is the standard query language for RDF [2]. Let V be a set of variables. A tuple $t \in \mathbf{I} \cup \mathbf{B} \times \mathbf{I} \times \mathbf{I} \cup \mathbf{B} \cup \mathbf{L}$ is called a triple pattern. Blank nodes in triple patterns can be considered as query variables for our purposes. A set of triple patterns is called a basic graph pattern. We denote by $var(t)$ and $var(P)$ the set of variables found in a triple pattern t and basic graph pattern P, respectively. We call a variable $?x \in var(P)$ a *join variable* if it appears in two or more triple patterns of P, and a lonely variable otherwise.

We define the semantics of SPARQL queries in terms of solution mappings. A mapping μ is a partial function $\mu : \mathbf{V} \rightarrow I \cup B \cup L$ where the domain of μ ($dom(\mu)$), is the set of variables on which μ is defined. Consider a triple pattern t, we denote by $\mu(t)$ the image of the triple pattern t under μ, i.e. $\mu(t)$ is the triple obtained by replacing the variables in t according to μ. We say that two mappings μ_1 and μ_2 are compatible, denoted $\mu_1 \sim \mu_2$, $\mu_1(?x) = \mu_2(?x)$ for every $?x \in dom(\mu_1) \cup dom(\mu_2)$. Given sets of mappings Ω_1 and Ω_2, we then define their join as We can now define the evaluation of a triple pattern and a basic graph pattern over an RDF graph G (the latter being defined as a join over its triple patterns):

Letting $\mu(P)$ denote the image of P under μ, with respect to the latter definition, we can equivalently say that

SPARQL offers a wide range of query operators that can be used to combine or modify the results of ba-

sic graph patterns, such as union, optional, filters, aggregates (counts, averages, etc.), property paths, etc. In this paper, we focus on predicting the performance of queries having only projections (`SELECT`) join and optional operations between basic graph patterns, which form the core of SPARQL queries.

2.2. Database systems and implementation plans

In [10] the authors present a general architecture (Figure 2.2) for query processing: (1) query parser, (2) query rewriter, (3) query optimizer, (4) plan refinement and query execution engine. The parser reads the query and transforms it into the system's internal representation. Then, in (2) the query is rewritten by creating a logical query plan from this internal representation. The query optimizer is in charge of applying different optimizations depending on the type of system, the physical state, the indexes to be used, the nodes to send the query to, etc. As a result, the query optimizer generates an optimized query plan that specifies how the query will be executed. This plan is refined and transformed into an executable plan by the plan refinement component which will finally be executed by the query execution engine.

Figure 2 shows some details of the optimization process executed for the query on the left side of that Figure. The order of the triples is determined by the cardinality estimates for predicates and types of triples. Note that in this case the triple (`?person dbo:birthPlace :Cuba`) is executed first because the predicate `dbo:birthPlace` is the lowest cardinality predicate defined in the statistics file. Note also that the triple (`?person dbo:birthDate ?birth`) executed is the third one executed because it is a triple of type `var_uri_var` so it is usually more expensive during execution. An inappropriate selection of the order in which the intermediate results of a query are executed can cause significant impacts on the execution time of a query, as well as on the hardware resources.

3. Related Work

Before stepping into describing our contribution we present the existing approaches for query optimization in both, relational and graph database systems. In the former, we divide the approach in those works that predict the cardinalities used to optimize a query and those works that focus on predicting the performance of queries.

3.1. Cardinality prediction in RDBMS

Akdere et al. [3] use Reinforcement Learning (RL) combined with traditional human-designed cost models to automatically learn optimized query plans. They use dense neural networks to minimize the cost (intermediate results cardinality) of a subplan resulting from a join operation. The authors encode each intermediate table, generated after a join operation, using `1-hot` vectors which are then concatenated. The RL algorithm iterates over the predictions of possible joins choosing at each step the least costly one until the complete plan until the algorithm obtains the optimal plan.

Kipf et al. [9] proposed a deep learning cardinality estimator called the Multi-Set Convolutional Network (MSCN). It predicts co-relationships of joins within the data, addressing some weaknesses in techniques such as sampling basic statistics by table. MSCN uses Deep Sets [26] which allows expressing query features using sets, which can then be used as input for deep learning models. The authors extract each query feature using the following sets:

1. Table Set: samples information from each table, using one-hot vectors as their internal representation.
2. Join Set: similarly as in the "Table Set", the authors use one-hot vectors for representing joins between tables in the query.
3. Predicate Set: the authors represent predicates in triples of the form (col, op, val), using one-hot vectors for the predicate (col), the operator (op) and for the value a normalized number between $[0:1]$.

In general, these cardinality estimators rely on the use of predicate statistics and operators, which are typically represented in trees that encode the query structure. However, these MLP (fully connected neural nets) models may not have the appropriate inductive bias for them [5]. In despite of that, there are some works like in [13], which conclude that cardinality prediction does not necessarily lead to the optimisation of a good plan.

3.2. Performance prediction in RDBMS with Machine Learning

One of the pioneering works in performance prediction using machine learning techniques is [3]. The authors encode SQL queries using operator-level features in conjunction with query execution plan information:

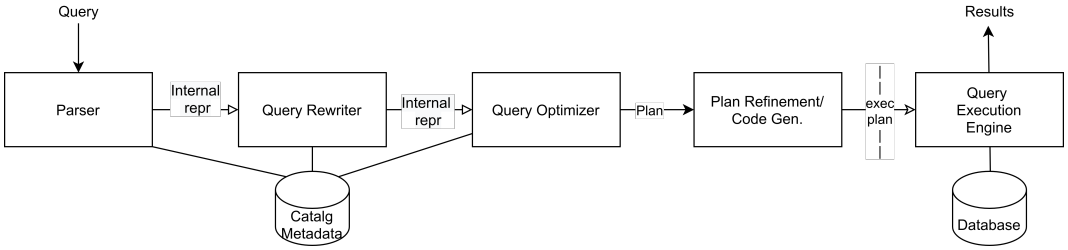

Fig. 1. (1) query parser, (2) query rewriter, (3) query optimizer, (4) plan refinement component and query execution engine.

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX : <http://dbpedia.org/resource/>
SELECT  ?person ?birth ?death
FROM <http://dbpedia.org>  WHERE {
  {
  ?person dbo:birthPlace :Cuba ;
  tdf:type dbo:Athlete .
  ?person dbo:birthDate ?birth ;
     FILTER (isLITERAL (?birth ))
     FILTER ( ?birth < "1900-01-01"^^xsd:date )
  }
  OPTIONAL {  ?person dbo:deathDate ?death . }
}
LIMIT  100
```

(a) SPARQL query.

```
(stats
  (meta
    ...
    (count 89337663))
  ...
  ((VAR rdf:type dbo:Athlete>) 114848)
  (dbp:birthPlace 41949)
  ...
  (dbo:birthDate 232640)
  ...
  (dbo:deathDate 69706)
  ...
  (dbo:federalState 81)
  ...
```

(b) Query statistics used by the optimizer

Fig. 2. On the left a SPARQL query and the statistics used for its optimization within the query planner on the right.

– At the plan level: they extract features from the query optimizer estimates, such as operator cardinalities and plan execution costs, together with the occurrence count of each type of operator in the query plan. A single model is used for these types of features (a Nu-SVR and Kernel Canonical Correlation Analysis (KCCA)).
– At the operator level: they extract characteristics that apply to all operators, such as: I/O estimates, number of tuples, selectivity, and the execution times of each operator. In this case, the authors use multivariate linear models for each operator. These are nested according to the order of execution provided by the engine, using the predictions of the lowest level operators as inputs for the next operator in the tree until the algorithm returns the complete plan prediction.

From this work several others followed, using Deep Neural Networks to predict query performance. Marcus et al. [15] propose a deep neural network which focuses on training query plan structures. It follows

a similar concept to the one described in [18] nesting small modules or units of dense neural networks (nodes) to build a tree-structured network. Each neural unit processes one operation in the plan, the leaf units correspond to scan operations on tables. The query characteristics they use are, cost estimates, I/O, intermediate results, join type, hash algorithms, sorting method, among other data. In addition to the intermediate latency of the operator, each leaf unit also predicts a vector representative of the processed input. This vector is then used as input for the next node that receives as input the two nodes (the left and right) plus the features inherent to the join type, until it reaches the root of the query and the algorithm outputs the query plan latency. Because the features used are only those delivered by the PostgreSQL optimizer this solution does not need an expertly defined feature design. However, its success depends on the optimizer delivering relevant information from each operator of the query plan. This affects its re-usability in the context of graph database engines where it is difficult to obtain such descriptive information from the optimizer beyond the optimized tree.

Neo [14] is one of the most recent works using neural network models and reinforced learning to build an end-to-end solution for query optimization in RDBMS systems. Neo learns to make optimization decisions by taking into account the order of joins, physical operators and index selection. The performance obtained is competitive with optimizers for enterprise RDBMS systems such as Microsoft SQL Server and Oracle. Neo brings together some of the most useful techniques previously seen for vector coding of SQL queries. The query-level features are divided into 2 components. The first uses the top half of an adjacency matrix to encode cross joins, whose vectors are then concatenated. The second component encodes the predicate columns using one of 3 methods:

- One-hot vector marking the presence or absence of a predicate.
- Histograms: similar to above but the 1 in the predicate position is replaced by the selectivity of the predicate (normalized to [0, 1]) using histograms.
- R-vector: rescues semantic information in predicate embeddings using word2vec-based models [17].

At the level of execution plans, characteristics are encoded preserving their inherent structure as trees. For each plan or sub-plan the model creates a vector tree where each node contains information about: types of joins, type of search operations on the data (table, indexes, etc.). Next, Deep Neural Network called "Value Network" processes the previous code above using tree-based convolutions (TBCNN) [18], and can predict the execution performance of queries or subqueries. Similarly to CNN networks for computer vision problems, the Neo architecture (especially the use of Tree-Based CNN), is designed to create a suitable inductive bias for query optimization.

3.3. SPARQL Performance prediction

As we have mentioned, SPARQL query performance prediction has not been widely addressed. To our knowledge, the most relevant work is the one proposed by Hasan and Gandon in 2014 [7]. The authors use Support Vector Machine for regression (Nu-SVR) and K-Nearest Neighbors (Knn) to predict query execution time. SPARQL queries are encoded as input vectors of the algorithms by mixing the following approaches:

- Algebraic: the authors extract the frequency of the operators present in the body of the query.
- Graph patterns: queries are represented as graphs in order to extract their BGPs. Using that graph representation the authors apply the K-medoids algorithm on the query dataset [7]. The intuition is related to the fact that SPARQL queries are in essence formed by patterns of triples that search for matches in the data stored in the database. Graph patterns are shaped by these triple patterns, so if the graph patterns of two queries are very similar they may have similar execution times associated with them.

In [1], the authors use similar feature and model extraction approaches as in [7]. In addition to execution time, the authors predict other performance metrics such as CPU and memory usage. Unlike in [7], each type of triple is modeled as a structurally different subgraph in order to further differentiate the training examples. The computation of the edit distance between all queries is replaced by randomly selecting the K centroids of the queries generated in each template used in [7].

While there are a variety of systems implementing a query optimizer using Deep Learning in the relational database management systems area, in the graph database world these optimizers are scarce. We learned from them the techniques to develop our Deep Learning for graph databases, which we describe in the next Section.

4. Tree-Based Convolutional Neural Networks for SPARQL query performance prediction

Based on the problem raised in Section 1 and the advances in the field described in Section 3, we describe our solution below. As mentioned previously, a prediction model based on deep networks faces two main challenges for query execution time prediction: (i) selecting the right set of features to encode a query; (ii) the definition of the network architecture that exploits the features defined in (i) to perform predictions. In particular, we will take advantage of the SPARQL query structure to train a tree-based convolutional neural network.

Section 4.1 details the feature extraction techniques explored, which we will then apply to the data sources identified in the same Section 4.1. Section 4.2 details the proposed neural network architecture that uses the extracted features for training and subsequent evaluation of the proposal.

4.1. Feature extraction

Feature extraction from SPARQL queries is one of the two fundamental challenges for this research. For some learning tasks there is a consensus on the forms of data representation used. For example, images are generally represented using a matrix in which each cell represents the intensity of a pixel on the scale [0, 255]. For color images the RGB format is used, which is represented by 3 matrices (one for each channel). In other learning tasks related to audio signal processing it is common to use codification such as: 'waveforms' (waveform) and spectrograms [20].

Unlike these tasks, SPARQL queries do not have a *community-accepted* codification for the execution time prediction task. In this research, we follow the methodology for feature extraction in SQL queries defined in Neo, which includes the definition of general query features and execution plan-level features. In the following, we propose the feature extraction that we use.

4.1.1. Query-level characteristics

For query-level features we use two approaches, algebraic feature extraction and graph pattern feature extraction.

Algebraic Characteristics The algebraic characteristics record the frequencies of SPARQL operators present or not in the queries as characteristics. More in detail, we selected as features the occurrence of operators: BGPs, joins, `OPTIONAL`, `UNION` and `FILTER`, among others. We also include as query features the number of triple patterns (bgps) in the queries and the depth of the tree. Additionally, instead of including only the occurrences of joins between BGPs, we also include joins between triple patterns. This allows us to include more information in the model, since most joins occur between triple patterns in the query. If the query contains the `LIMIT` operator, we use its numeric value.

Figure 3 (left side) shows an example of algebraic feature extraction. The procedure followed is as follows:

1. Use Apache Jena ARQ to extract the tree algebra from the SPARQL query.
2. Count frequency of terms used per query.
3. Produce a vector of n features (one per term) per query.
4. Add the depth of the tree and the amount of bgps.

Graph patterns The second approach benefits from the ability to model SPARQL queries as graphs to extract patterns from them. This uses similarity patterns between queries, represented as graphs using a vector of K_{gp} dimensions, where the value of each feature is the structural similarity between that query pattern and any of the other representative patterns in the dataset.

Figure 3 (right side) shows an example of extracting features from graph patterns for a SPARQL query. We describe the process as follows:

The first step is to build a graph representative of the SPARQL query. For this, we join the triples taking into account the nodes that are common variables between them. Finally, we replace both variables and literals by a node with a common symbol (Figure 3 with "?").

The second step is to make clusters from the queries using the k-medioids algorithm [8]. K-medioids choose the centroids (Kgp centroids) using a distance function as the Edit Distance. In Figure 3 (right side) each circle represents a cluster of queries in which we identify each centroid by a color.

Finally, to obtain the vector representation of K_{gp} (the query pattern), we compute the structural similarity between a query graph p_j and the center k_i of the cluster query graph $C(k)$. The term $d(p_i, C(k))$ is the edit distance between the query graphs p_i and $C(k)$, obtaining a normalized similarity between 0 and 1 (0 being the most different and 1 being the most equal).

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

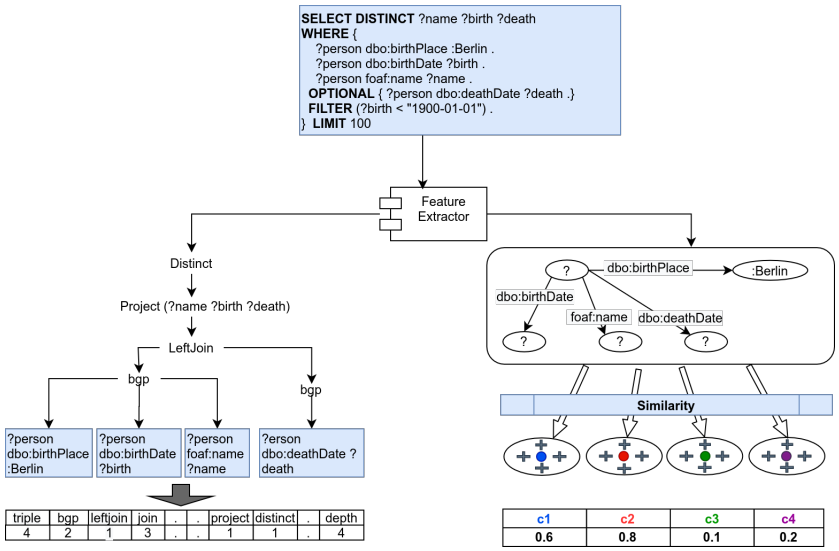

Fig. 3. Query level feature extraction methods: algebra feature encoding (left side), graph pattern encoding (right side).

The results of [7] propose the combination of both approaches (Algebra Features and Graph Pattern Features) to obtain better performance. We use that combination to extract query-level dataset features. In the next Section we will define the features at the execution plan level.

Histograms of predicates The Jena optimizer proposes a cardinality estimate for each triple. Jena calculates that value depending on the type of triple and the predicate included in the query. For example, a triple of type `VAR_URI_VAR`, where the predicate used is `wdt:P31` ('instance of') has an associated cardinality of more than 51 million results. However, if it is present in a triple of type `VAR_URI_URI` the optimizer delivers a cardinality of '4'. We will use the cardinalities of the traditional optimizer normalized to [0, 1] as the value associated with each predicate, with 1 being the maximum cardinality among all predicates sampled.

4.1.2. *Characteristics at the implementation plan level*

Regardless of the operators in a SPARQL query, performance is largely determined by the order in which the engine executes the query patterns. An early triple pattern with a bounded variable execution may save minutes of query processing. Such order is determined by the predicate selectivity within the query. The query engine by first selects the triples whose predicate has lower selectivity according to a file storing the per-predicate statistics, codifying its execution plan as a binary tree. We extract such binary tree and codify it for later use within our model as shown in Figure **??**. The leaf nodes of the tree represent the triples that execute SCAN operations on the database. The intermediate nodes represent the JOIN operations between the triples. The execution plan has to be analyzed in a bottom-up fashion, with the lower left nodes being the first to be executed.

We enrich each node's information by adding node semantic information such as the estimated cardinality for the node according to the optimizer, cost information, etc. This information is provided by the query optimizer, and also introduces a high dimensionality in the node vectors due to the fact that the databases are composed of thousands of predicates. In a single query hundreds of predicates may be present, needing a large vector to codify it. This problem is not common in the context of relational systems, whose joins are executed between a relatively small number of tables.

4.2. *Deep neural network architecture*

We now present our architecture for predicting SPARQL query execution times. Similarly to the work in Neo [13], we propose a neural network trained to predict query latency for a complete execution plan $P^{(f)}$. Our goal (as opposed to Neo) is not to learn from partial query plans $P^{(i)} \subset P^{(f)}$, since that research is bounded to latency prediction. However, a proposed optimizer using this architecture in the context of SPARQL should be trained on $P^{(i)}$ sub-plans

to approximate the best possible latency estimate such that $P^{(i)} \subset P^{(f)}$.

We feed this architecture with a training set A composed by N execution plans $\{P_n^{(f)}\}_{n=1}^{N}$ each of them with a known latency $L(P_n^{(f)})$).

Our goal is to find a parametrized function h that maps $P^{(f)} \longrightarrow L(P^{(f)})$. We choose h that minimizes the error of estimating query latency using a quadratic loss function (L2) over the training set A :

$$\theta^* = \text{argmin}_\theta \text{MSE}\left(L(P^{(f)}, h(P^{(f)}, \theta)\right), \qquad (2)$$

where θ^* is the set of model parameters and *MSE* is the mean square error defined as

$$\text{MSE}\left(L(P^{(f)}), h(P^{(f)}, \theta)\right) = \frac{1}{N}\sum_{n=1}^{N}(L(P_n^{(f)}) - h(P_n^{(f)}, \theta))^2.$$

4.2.1. Architecture

Figure 4 shows our generic architecture. We designed it to create an inductive bias suitable for query latency prediction: the network design encodes the intuition details of the causes that produce fast or slow query execution. Experts studying query execution plans learn to recognize sub-optimal or good plans by tree pattern matching. The intuition behind is that the architecture can recognize these patterns by analyzing the subtrees from a query execution plan, similarly to what experts do. Consequently, the model is essentially an expert at recognizing query execution trees, learning them automatically, from the data itself, using a technique called tree convolution [18].

As shown in Figure 4, the first step is to introduce the query-level encoding into the model through several fully-connected layers. (i) the architecture concatenates the vector produced by the last dense layer with the plan-level coding, i.e., the same vector is added to each node in the tree. This is a technique known as "spatial replication" [13], to combine fixed-size data (query-level encoding) and dynamically sized data (plan-level encoding). (ii) Next, we feed the plan-level encoding into our Autoencoder architecture (Section 4.2.3 to reduce the high dimensionality produced by the large amount of graph edges. (iv) we concatenate the autoencoder output with the query level encoding and feed with it the tree convolution layers [18], an operation that has as input and output a tree with the same structure but different node-level dimension. (v) Subsequently, we apply a dynamic clustering operation [18], as explained in Section 4.2.4. This architecture has many similarities with the architecture proposed by Marcus et al. [13], however we adapted it to deal with a graph query language (i.e. no fixed schema, thousands of different edges, etc.).

4.2.2. Convolutions on trees

Convolutional Neural Networks (CNN) are a very popular option when the learning task is done over spatial hierarchies [25], like images [11] or time series [22]. As in Neo [4], we use a tree-based convolutional architecture [18] since we want to learn the hierarchy of a SPARQL query plan, which is very similar to a SQL query plan. Thus, we naturally process the plan structure [16] by sliding a set of shared filters over each part of the plan tree, capturing a wide variety of local relationships between parent and child nodes.

These filters search for combinations of the `JOIN` type between triple patterns, depending on the specific predicates present in these triple patterns. These output filters provide the signals used by the final layers of the network and highlight relevant query elements. Some of these query elements are the implications of join order predicate or the implications of using a very common edge in the RDF graph (such as the type predicate) on the left or right side of a JOIN. For example, executing a triple of type `VAR_URI_URI` or `VAR_URI_LITERAL` first significantly decreases the cardinality of intermediate and final results and hence execution time, whereas a triple of type `VAR_URI_VAR` is usually associated with high cardinality and hence high-latency queries, specially if the previous `URI` represents a very common edge in graph database.

Following Neo's approach, the tree query is generated with a pre-order path, where at each step not only the root, but also the left and then adding the right child of the tree, generates at each step the locality of a tree (parent / left child / right child). Then, we use a traditional 1D convolution operation applied on the tree to learn the queries.

Since query trees has exactly two child nodes, each convolutional filter has three weighting vectors: $\mathbf{e}_p, \mathbf{e}_l, \mathbf{e}_r$, one of the parent node, one for the left child and another for the right child of the tree, forming the vector $\mathbf{x}_p$ of a node and its left and right children, $\mathbf{x}_l$ and $\mathbf{x}_r$ respectively. We set to $\vec{0}$ left and right children

[4] TreeConvolution used in Neo, simplified for binary trees. https://github.com/RyanMarcus/TreeConvolution

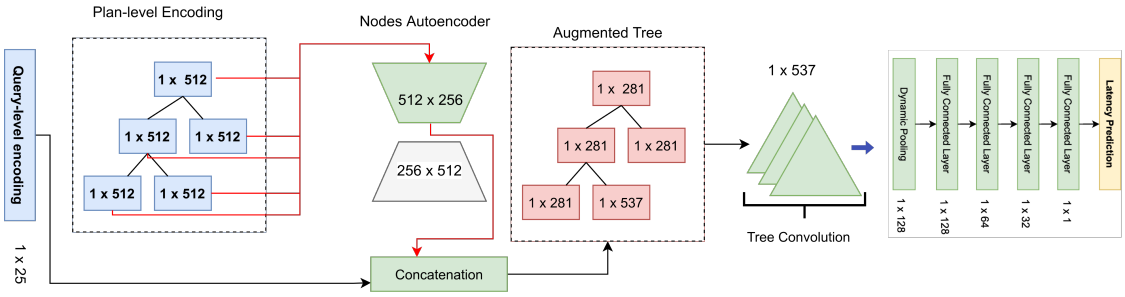

Fig. 4. Architecture of the proposed neural network including query-level and execution plan-level features, autoencoder and dynamic pooling

when they are leaves. Finally, output of a filter x'_p of a node $\mathbf{x}_p$, is computed as follows:

$$x'_p = \sigma(\mathbf{e}_p \cdot \mathbf{x}_p + \mathbf{e}_l \cdot \mathbf{x}_l + \mathbf{e}_r \cdot \mathbf{x}_r). \quad (3)$$

Equation 3 shows how the weights learn the local hierarchies of a tree node by using $\sigma(\cdot)$ as *activation function*, i.e. a nonlinear transformation, and $(\cdot)$ as dot product (or scalar product). This transformation propagates through the nodes of the tree up to the root, independently of the size of that tree. In order to extract different and useful features from a tree, we apply a set of filters adding also multiple tree convolutional layers. By means of these layers we learn the relationships between nodes from different parts of the tree. Hence, the first tree convolution layer learns local features between a node and its children. While the last tree convolution layer learns complex features (e.g., recognizing a sequence of left-deep joins[5].

The intuition behind the tree convolution filters we apply is to encode the size of the query plan intermediate results and propagate them through the query operators represented by nodes in the trees. Since the amount of intermediate results is directly proportional to the selectivity of the predicates in the triple patterns (which retrieves the query data from disk), and the amount of properties is large we use an autoencoder to reduce the query dimensionality from leaf nodes.

4.2.3. *Pre-training with autoencoders for leaf nodes*

"In RDBMSs, JOIN operations are performed between a relatively reduced and fixed set of tables, however, the scenario in GDBs is more complex. As already mentioned, JOIN operations in GDBs could be described as 'dynamic'. This is because JOINs in the SPARQL language are defined between predicates by matching variables between two triples. The above implies that depending on the size and diversity of the ontology of a database, the vectors of nodes in the execution plan tree may contain thousands of predicates (around 10,000 of different properties in Wikidata[6], which map to predicates in the query). In order to address this problem, we propose to perform a pre-training on the node vectors using autoencoders. The use of autoencoders allows to reduce the dimension of the node vectors while preserving the node information.

The autoencoder consists of a neural network composed of densely connected layers that reduce the output dimension. They do that by stacking layers in the "encoder" and then increasing in the "decoder" layers. We train first with the node vectors of all the trees in the training set. Next, only the encoder layers are reused in the architecture.

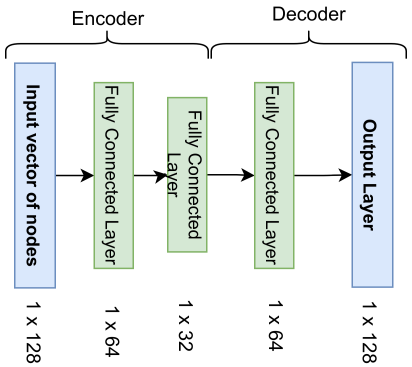

Fig. 5. Autoencoder architecture that reduces the dimensionality generated by the large amount of properties (edges) in a graph database

[5]Join operations are ordered starting on a left side leaves.

[6]https://www.wikidata.org/wiki/Wikidata:List_of_properties

4.2.4. Fixed vector with dynamic pooling

After applying the convolutional layers on the plans, we follow the approach in Neo by extracting the features identified by the tree convolution filters. The new tree has exactly the same shape and size as the original one, which varies among the different plans, but with nodes of smaller dimension. Therefore, the extracted features cannot be sent directly to a neural layer of fixed size. To address this problem we apply a dynamic pooling layer [18]. Specifically, we use the maximum value in each dimension of the features detected by the tree-based convolution. After pooling, the fixed vector summarizing the plan information is suitable to be processed by dense hidden layers, until finally the layer with 1 output unit allows predicting the query latency. Finally, the network can be trained in a supervised environment using backpropagation.

5. Experiments

In the previous Sections we proposed our architecture to solve the query latency prediction problem for querying RDF graph databases. We now present the experiments to assess our architecture against other state-of-the-art approaches. We will see how our Tree-Based Convolutional Neural Network statistically outperforms the other algorithms.

5.1. Method

Neural network training commonly follows the same order: (i) first clean the data and separate them into training, validation and test sets, (ii) extract feature vectors, (iii) train the models (selecting the best combination of hyperparameters according to the validation set) selecting the model that best minimizes the error, (iv) test those models on the test set.

Wikidata In the evaluation we used real world data and queries from Wikidata [23]. Wikidata is a collaboratively edited knowledge base hosted by the Wikimedia Foundation. It is a common source of data for Wikimedia projects such as Wikipedia, and it has been made available to the general public under a public domain license. Wikidata stores 86,671,701 items (RDF resources), and 1,084,935,969 statements (triples[7]).

For (i) we use the Wikidata log files from a specific time period[8]. We filtered these query logs keeping only the queries that returned results, removing those with the `SERVICE` keyword. We divided the remaining queries in training, validation and testing sets, with a distribution of 20% for the testing queries, while we used 70% and 30% for training and validation sets respectively. We run all these queries in the Wikidata snapshot latest truthy[9].

In Section A.1 we describe the process we followed to (ii), extracting the query characteristics (i.e. extracting the features vector).

To complete step (iii), we use a grid search, since it allows us to identify the best combination of hyperparameters exhaustively and using some constraints. We selected the number of layers, number of units per layer, activation functions, optimizer and learning rate as hypeparameters as shown in Table 1. To select the best combination of them we use 70% of the queries as training data and the remaining 30% as validation set (excluding 20% of the entire dataset used for testing). Best model is obtained with LeakyReLU as activation function, Adam optimizer, learning rate = 0.00015 and three convolution layers of 512,256,128 units in each layer.

Table 1
Hyperparameters values for gridsearch

Hyperparameter	values
Activation function	ReLU, Leaky ReLU, tanh
Optimizer	Adam, Adagrad, SGD
Learning rate	{0,0015,0.00015 }
Units number of the first Tree convolution layer	{1024, 512, 256, 128}
Units number of the second Tree convolution layer	{512, 256, 128,0}
Units number of the third Tree convolution layer	256, 128,0
Units number of the fourth Tree convolution layer	256,0

5.1.1. Baseline Hyperparameters Tuning

We now compare the architecture we propose with other state-of-the-art proposals. Specifically, we compare the NuSVR approach in [7] and a dense neural network similar to the one proposed for query-level features. The details of the baseline models and the hyperparameter tuning performed are specified below.

[7] https://tools.wmflabs.org/wikidata-todo/stats.php
[8] All intervals from Wikidata truthy logs
[9] https://dumps.wikimedia.org/wikidatawiki/entities/

Tuning of NuSVR hyperparameters The Support Vector Machine with NuSVR kernel proposed in [7], uses the Algebraic features and graph patterns specified in Section 4.1.1. In this case it is necessary to adjust the hyperparameters:

- Nu: upper bound on the fraction of training errors and a lower bound on the fraction of support vectors. Must be in the interval (0,1].
- C: Penalty parameter C of the error term.

For selecting NuSVR hyperparameters we run a grid search. That procedure performs 10 iterations by selecting random values bounded in certain ranges of the hyperparameters *C* and *nu*. We then select the combination of hyperparameters that performs best on the validation set. The lowest RMSE is obtained using a value $C = 200$ with a $nu = 0.4$.

5.1.2. Hyper Parameter Tuning

The second architecture proposed as a baseline corresponds to a neural model as dense layers (DenseModel hereafter) with the following characteristics:

Use as input vectors the query-level features: algebraic features, graph patterns and predicate cardinalities, detailed in 3.1.1, 3.1.1 and 3.1.2 respectively. Hidden 3-layer model with ReLU activations. After the activations in layers L1 and L2, and L3, a Dropout layer was applied with a dropout probability of 0.25. The output layer contains 1 single linear unit corresponding to the network prediction.

After selecting the best hyperparameters and running our model with them, in the next Section we present the results for our work.

6. Results and discussion

In this Section we present the process for validating the architecture and verify whether we fulfill our hypotheses and goals or not. In particular we explain the evaluation metrics we selected and the expected results. This section shows the data that accompany the most important results of the work and are the basis for the conclusions presented in Section 7.

6.1. Metrics

We use the Mean Square Error (MSE) as the loss function to evaluate our neural network models. MSE minimizes the quadratic differences between the estimated value and the true value. Hasan and Gandon [7] use the Root Mean Square Error (RMSE) which is a variant of MSE, however it penalizes excessive large errors.

$$MSE_{(test)} = \frac{1}{m}\sum_{i=1}^{m}(y_i^{(test)} - \hat{y}_i^{(test)})^2, \tag{4}$$

$$RMSE_{(test)} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(y_i^{(test)} - \hat{y}_i^{(test)})^2}. \tag{5}$$

Another common error metric and loss function in regression problems is the Mean Absolute Error (MAE). As a loss function it minimizes absolute differences between the estimated value and the true value. It tends to be more robust to outliers than the MSE. We use all these three metrics to validate our experiments.

$$MAE_{(test)} = \frac{1}{m}\sum |y_i^{(test)} - \hat{y}_i^{(test)}|. \tag{6}$$

Figure 6 shows that our full model and the same model using auto-encoders perform similarly. When using only trees the model clearly performs worse. This poses a very interesting situation, in which the auto encoder seems to not provide any help in the query latency prediction. The auto encoder helps to reduce the dimensionality when having a large amount of different RDF predicates (present in all queries in both training and validation data sets), lowering the model's training time. However, the dimensionality seems already incorporated in the model without the auto encoder.

Figure 7 shows the predictions our deep learning architecture made for the validation (left) and test (right) sets. Each point in the plot represents a query in the dataset, showing the value on the *X* axis the model's query execution prediction and on the *Y* axis the actual query execution time.

- Green dots represent queries whose prediction are considered good: $(|y - \hat{y}| <= y * 0.2)$.
- Blue dots represent queries whose prediction are considered acceptable $y * 0.2 < (|y - \hat{y}| <= y * 0.4)$.
- Red red dots correspond to bad predictions.

We consider most of the predictions good since the difference between the predicted query execution time

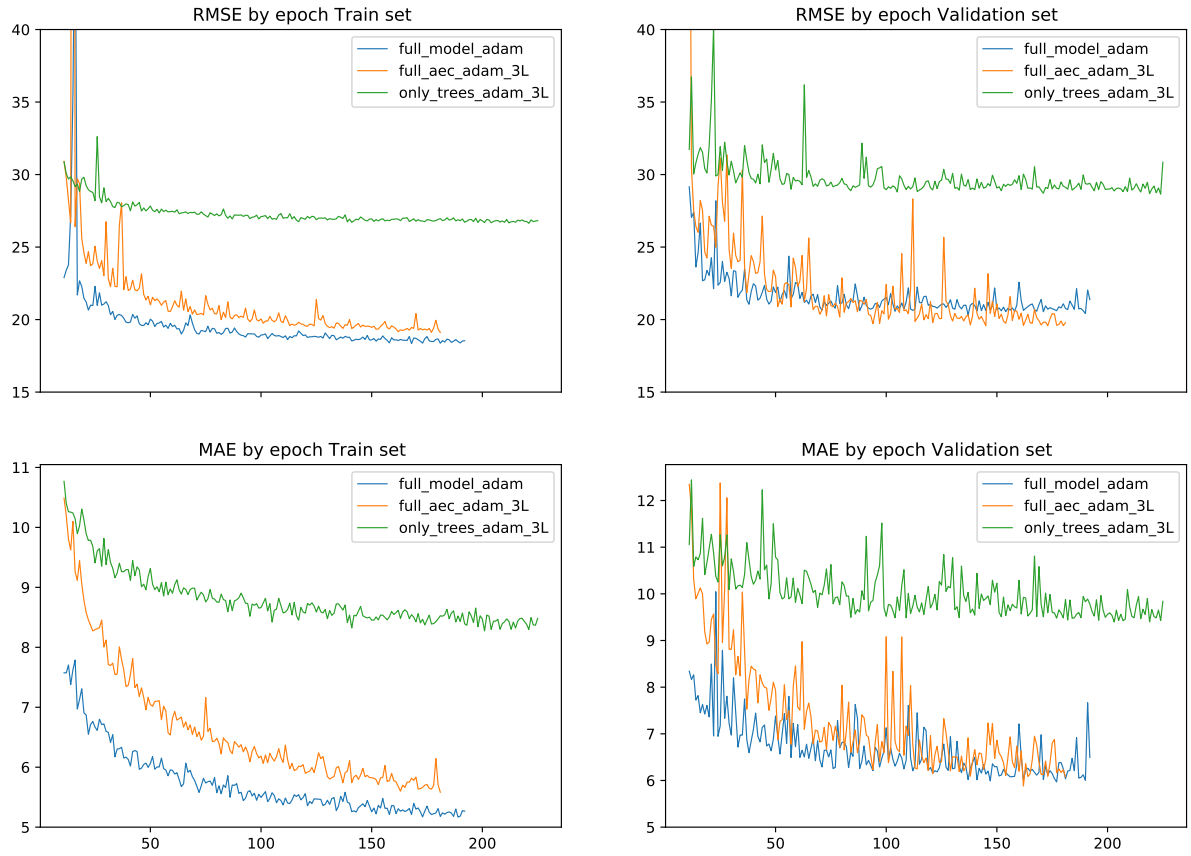

Fig. 6. Figure shows the model convergence according to RME and MAE metrics. X axis represents the amount of epochs while the Y axis presents the metrics used. According to the hyperparameters grid search we executed, "full model adam" is the best model using three TreeConv layers (feeding it with the query and query execution tree characteristics). We also evaluated the use of autoencoders to reduce the amount of parameters added by the properties within the data, using the same "full model adam". We represented that in the box "full aec adam 3L", however we did not obtain better results. Finally, we also show our model using only tree characteristics in the plot labelled "only trees adam 3"

and the actual time the query engine needed to run the same query has a difference below 20%. For instance, a query needing 0.5 seconds to be executed, our model predicted that it would need between 0.59 or 0.41 seconds, which is reasonable for the query engine to make a decision about what plan to chose to execute the query. For larger query execution times (25 seconds or more), our model predicts correctly most of the query execution times. Moreover, a 20% in a query of 30 seconds would be around 5 seconds up or down, which allows more informed decisions to the query optimizer.

The amount of bad predictions is minimal, however most of these bad predictions are in those queries that are faster to terminate. This poses problems since the model may predict that a query takes 10 seconds in while it only needs 0.5 for being executed. We look more in detail to these queries that our model failed to correctly predict its execution time in Section 6.2.

6.2. *Discussion*

After completing the query latency prediction we find it mandatory to look at those queries that the model incorrectly predicted their latency. The goal is to identify query patterns that may have been overlooked by our statistical approach. All the queries used in the evaluation along with their codification results, and predicted latency are available in the Github repository for our project[10]. These queries are marked as *red* in Figure 7 (1,743 out of 20,632, 7% of the total). To these queries we run a canonicalization soft-

[10]https://github.com/cbuil/sparql-latency-prediction

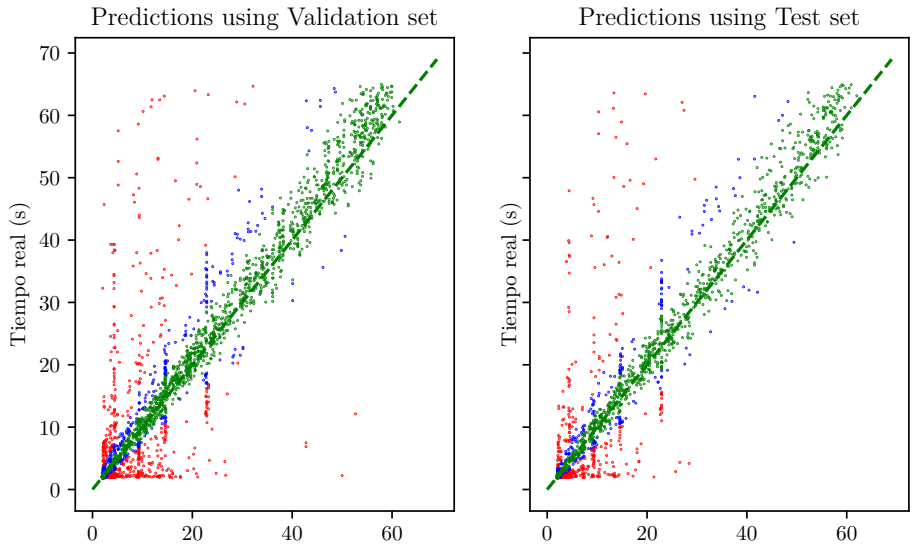

Fig. 7. Predictions using the proposed architecture for validation and test suites. On the left the predictions using our model over the validation set, on the right the predictions using our model over the test set. Green dots represent those queries that our model correctly predicted its execution time.

ware [21] for finding common SPARQL queries (variable normalization, translating queries to a common representation, etc.). That software found 366 repeated queries, which we remove from the analysis, leaving 1377 queries for our analysis.

Missing query operators In the remaining queries there are 174 `COUNT` queries, 301 queries having a property path operator, 112 `VALUES` operator, as well as 151 queries using vocabularies different from Wikidata, which we do not support. That makes 738 queries not supported by our model, 53% of the queries in the dataset.

Single triple pattern queries The largest group of queries misclassified by our model is the group of queries with a single triple pattern having the `P31` Wikidata property (*type of*), with a total of 408 queries, around 30% of the wrongly classified queries. We hypothesize that this is the largest set of queries since there are 117,967 different types in Wikidata[11]. Our model that seems to consider in a similar way data with type *Human* (`Q5`) with thousands of instances and *television network* (`Q1254874`), with less than 1,000.

[11] https://www.wikidata.org/wiki/Property_talk:P31

7. Conclusions

In this paper we presented a Deep Learning model to predict SPARQL query latencies. This model contributes to the solution of the open problem of finding an optimal query plan to a database query, in our case an optimal query plan to a RDF graph database using SPARQL queries. To find such an optimal query plan is an open research problem in all types of database systems, most notably on Relational Database Systems, which have several frameworks approaching it. We start our model by looking at one of these solutions, and we build a reusable characterisation of SPARQL queries that feed our model. We adapt the existing model from Neo process SPARQL queries and evaluate it against Wikidata, using their query logs and data. Our model correctly predicts the latency for queries in an 87% of the validation set and we provide an explanation to those queries that we fail to predict the latency.

As future work we plan to improve our model to correctly predict the query latencies from our analysis in Section 6.2 by adding more operators to our query characterisation. We will also develop a new query optimizer for MillenniumDB [24], based on the current model.

8. Acknowledgements

Daniel Casals has been funded by ANID – Millennium Science Initiative Program – Code ICN17 002 and by USM Postgraduate grant. Carlos Buil-Aranda has been funded by ANID – Millennium Science Initiative Program – Code ICN17 002 and Proyecto Fondecyt Iniciación 11170714.

References

[1] , . Dacasals/Sparql-Query-Exec-Prediction-Model. URL: https://github.com/dacasals/sparql-query-exec-prediction-model.

[2] , . SPARQL Query Language for RDF. URL: https://www.w3.org/TR/rdf-sparql-query/.

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

[4] Amat, D.A.C., Aranda, C.B., Valle-Vidal, C., 2021. A neural networks approach to SPARQL query performance prediction, in: XLVII Latin American Computing Conference, CLEI 2021, Cartago, Costa Rica, October 25-29, 2021, IEEE. pp. 1–9. URL: https://doi.org/10.1109/CLEI53233.2021.9639899, doi:.

[5] Battaglia, P.W., Hamrick, J.B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., et al., . Relational inductive biases, deep learning, and graph networks. arXiv:1806.01261.

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

[7] Hasan, R., Gandon, F., . 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), IEEE. pp. 266–273.

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

[9] Kipf, A., Kipf, T., Radke, B., Leis, V., Boncz, P., Kemper, A., . Learned Cardinalities: Estimating Correlated Joins with Deep Learning. URL: http://arxiv.org/abs/1809.00677, arXiv:1809.00677.

[10] Kossmann, D., . The state of the art in distributed query processing 32, 422–469. doi:.

[11] LeCun, Y., Cortes, C., Burges, C., . MNIST handwritten digit database 2.

[12] Leis, V., Gubichev, A., Mirchev, A., Boncz, P., Kemper, A., Neumann, T., . How good are query optimizers, really? 9, 204–215.

[13] Marcus, R., Negi, P., Mao, H., Zhang, C., Alizadeh, M., Kraska, T., Papaemmanouil, O., Tatbul, N., . Neo: A learned query optimizer 12, 1705–1718. doi:.

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

[15] Marcus, R., Papaemmanouil, O., b. Plan-Structured Deep Neural Network Models for Query Performance Prediction doi:.

[16] Marcus, R., Papaemmanouil, O., c. Towards a Hands-Free Query Optimizer through Deep Learning. URL: http://arxiv.org/abs/1809.10212, arXiv:1809.10212.

[17] Mikolov, T., Chen, K., Corrado, G., Dean, J., . Efficient estimation of word representations in vector space. arXiv:1301.3781.

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

[19] Pavlo, A., Angulo, G., Arulraj, J., Lin, H., Lin, J., Ma, L., Menon, P., Mowry, T., Perron, M., Quah, I., Santurkar, S., Tomasic, A., Toor, S., Aken, D.V., Wang, Z., Wu, Y., Xian, R., Zhang, T., . Self-driving database management systems, in: CIDR 2017, Conference on Innovative Data Systems Research. URL: https://db.cs.cmu.edu/papers/2017/p42-pavlo-cidr17.pdf.

[20] Purwins, H., Li, B., Virtanen, T., Schlüter, J., Chang, S., Sainath, T., . Deep learning for audio signal processing 13, 206–219. doi:.

[21] Salas, J., Hogan, A., 2018. Canonicalisation of monotone SPARQL queries, in: Vrandecic, D., Bontcheva, K., Suárez-Figueroa, M.C., Presutti, V., Celino, I., Sabou, M., Kaffee, L., Simperl, E. (Eds.), The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, Monterey, CA, USA, October 8-12, 2018, Proceedings, Part I, Springer. pp. 600–616. URL: https://doi.org/10.1007/978-3-030-00671-6_35, doi:.

[22] Tang, W., Long, G., Liu, L., Zhou, T., Jiang, J., Blumenstein, M., 2020. Rethinking 1d-cnn for time series classification: A stronger baseline. CoRR abs/2002.10061. URL: https://arxiv.org/abs/2002.10061, arXiv:2002.10061.

[23] Vrandečić, D., Krötzsch, M., 2014. Wikidata: a free collaborative knowledgebase. Communications of the ACM 57, 78–85.

[24] Vrgoc, D., Rojas, C., Angles, R., Arenas, M., Arroyuelo, D., Buil-Aranda, C., Hogan, A., Navarro, G., Riveros, C., Romero, J., 2021. Millenniumdb: A persistent, open-source, graph database. arXiv:2111.01540.

[25] Yamashita, R., Nishio, M., Do, R.K.G., Togashi, K., 2018. Convolutional neural networks: an overview and application in radiology. Insights into Imaging 9, 611 – 629.

[26] Zaheer, M., Kottur, S., Ravanbakhsh, S., Poczos, B., Salakhutdinov, R.R., Smola, A.J., . Deep sets, in: Guyon, I., Luxburg, U.V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., Garnett, R. (Eds.), Advances in Neural Information Processing Systems 30. Curran Associates, Inc., pp. 3391–3401. URL: http://papers.nips.cc/paper/6931-deep-sets.pdf.

Appendix A. Appendix

A.1. Validation by means of hypothesis testing

In this Section we validate that we can generalize the results we obtain to different datasets. To do that we use the 'Student's t-test for dependent samples', process that we describe in the following lines.

A.1.1. Student's t-test for dependent samples

The Student's t-test for dependent or paired samples is generally used to compare the means of one sample or two paired or dependent samples. In the case of using only one sample, the test is performed by comparing the resulting means of two events occurring in the same sample. Similarly, it is possible to apply this test by comparing the means of two events occurring over two samples, as long as it is guaranteed that these samples are dependent. The objective is to verify whether the records of event 1 change significantly with respect to those of event 2.

In our case, we apply the t-test on dependent samples to verify whether the difference in means between the results (using the RMSE metric) obtained with the proposed solution and the results of the baseline model (NuSVR 3.4.6) is statistically significant or not.

To apply the test we execute the following steps:

1. Use the queries extracted and processed from the Wikidata database (Section 4.1.2).
2. Select 5 random subsets from 85% of the training data separated in Section 4.1.2. The other 15% will be used as validation set.
3. Train the proposed TreeConvModel, DenseModel and NuSVR model on each training subset of step (1).
4. Evaluate each model on the test set obtaining in each case the RMSE.

Table 2
RMSE and MAE values for the 2 models in the 5 partitions for the *test* set.

	Tree_Conv		NuSvr	
	mae	rmse	mae	rmse
0	5.098	18.695	18.455	42.059
1	5.302	19.177	18.594	42.402
2	5.377	19.129	18.627	42.190
3	5.622	19.241	18.622	42.273
4	5.376	19.156	18.778	46.934
mean	5.355	19.08	18.615	43.171
std	0.188	0.219	. 0.115	2.107

Figure 8 shows the results obtained for the RMSE metric in each of the testing partitions. We see how the proposed model outperform the baseline model. We now determine whether the difference in the means over the sample (5 random subsets) for each model represents a true difference in the population from which the obtain the sample. To make this comparison, we use a t-test using the difference between the two means and dividing by the standard error of the difference between the two dependent sample means:

$$t = \frac{\bar{A} - \bar{B}}{S_{\bar{D}}}, \tag{7}$$

where $\bar{A}$ is the mean RMSE for the proposed solution, $\bar{B}$ the mean RMSE for the NuSVR, and $S_{\bar{D}}$ the standard error of the differences between the two means.

We obtain the standard error:

$$S_{\bar{D}} = \frac{S_D}{\sqrt{N}}, \tag{8}$$

$$S_D = \sqrt{\frac{\sum D^2 - \frac{(\sum D)^2}{N}}{N-1}}, \tag{9}$$

where: $S_{\bar{D}}$ is the standard error of the differences between sample means, S_D represents the standard deviation of the mean differences (for a population sample), D is the difference between the RMSE of each pair of samples for events A and B, N is the number of individuals in the sample (5 in this case).

Given the above, the observed t-value can be used to test the following hypothesis:

$$\begin{aligned} H_0 &: \mu_D = 0. \\ H_A &: \mu_D > 0, \end{aligned} \tag{10}$$

where μD represents the difference in sample means. Hypothesis H_0 states that there is no difference between the means of the returns between the models, which is rejected in favor of $H_A :> 0$ when $|t| > t_\alpha$, where t_α the critical value is obtained from a t Student distribution for $N - 1 = 4$ degrees of freedom.

We can also interpret the test using a p-value: if the *p-value* $> \alpha$, then the hypothesis H_0 that there is no difference between the sample means is accepted. Otherwise, it is rejected in favor of the alternative hypothesis H_A.

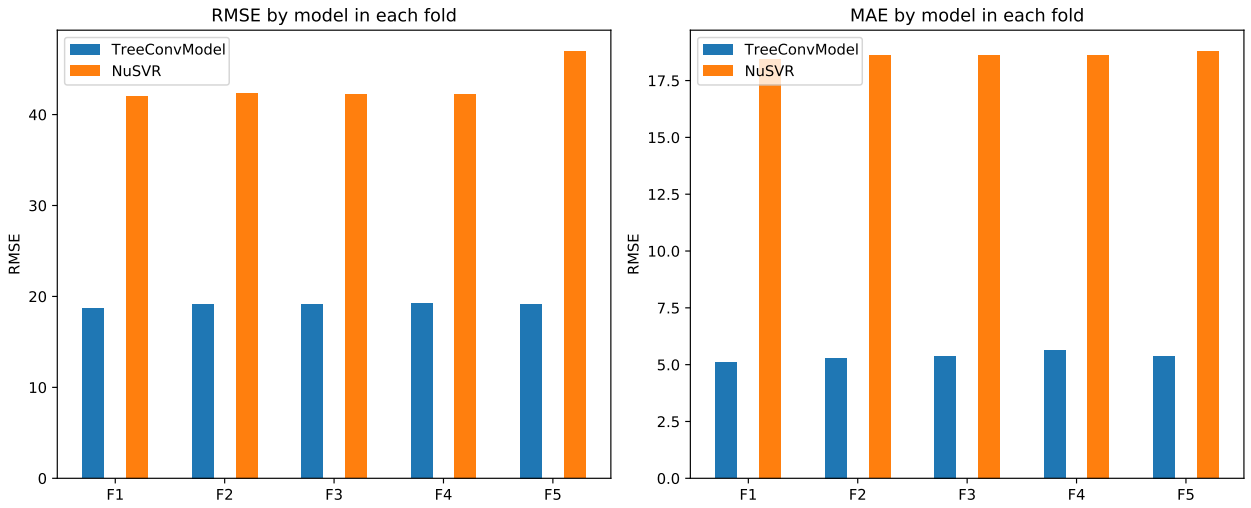

Fig. 8. This Figure shows RMSE (left) and MAE (right) metrics for 5-folds (random selection of 5 training and validation sets) evaluated within the two model's test set: "full model adam" (best model according to the grid search in Figure 6) and NuSVR using as parameters $C = 200$ and $\mu = 0.4/$.

	TreeConvMolde vs NuSvr
t_α	0.05
p-value	0.0000128
t	-26.090

Table 3

Results of the *t* and *pvalue* statistical tests from pairs of TreeConvModel and NuSVR model.

Table 3 includes the results of applying the above procedure. A *p-value* of $0.0000128 < 0.05$ indicates that the hypothesis H_0 is rejected, so it can be concluded that there is a statistically significant difference between the RMSE of the proposed model and the baseline.

Since there is a significant difference between the performances of the models under analysis, we can affirm that the superiority of the proposed model with respect to the baseline for the analyzed dataset are significant. Next, we present the final conclusions of the research.

To conclude this Section, we can safely say that our prediction model outperforms the existing state of the art models. This is however no surprise since we are using Deep Learning models, and more modern architecture. The most important outcome we want to highlight are the precision of our predictions. These may allow a query planner to take more informed decisions about how to process a query, even though we still need calculate better those query that execute faster than predicted.

Appendix B. Effectiveness of the use of autoencoders

We also studied the effect of using an autoencoder as a pre-training technique to reduce the dimensionality in the query plan node vectors.

Some details of the autoencoder training are added below:

- The input and output of the network corresponds to vectors of fix size, taking into account the information of the nodes of all the plans that form the dataset for the training set.
- We propose 1 hidden layer for the encoder and 1 for the decoder.
- The units per layer shall be selected so that the last layer of the "encoder" summarizes at least 4 times the number of input features.
- Adagrad will be used as optimizer with a learning rate in the order of 0.0001.

We next explore the effectiveness of pretraining with autoencoder to summarize plan-level information. The architecture tested has the following characteristics:

- Grid Search as described in Section 5.

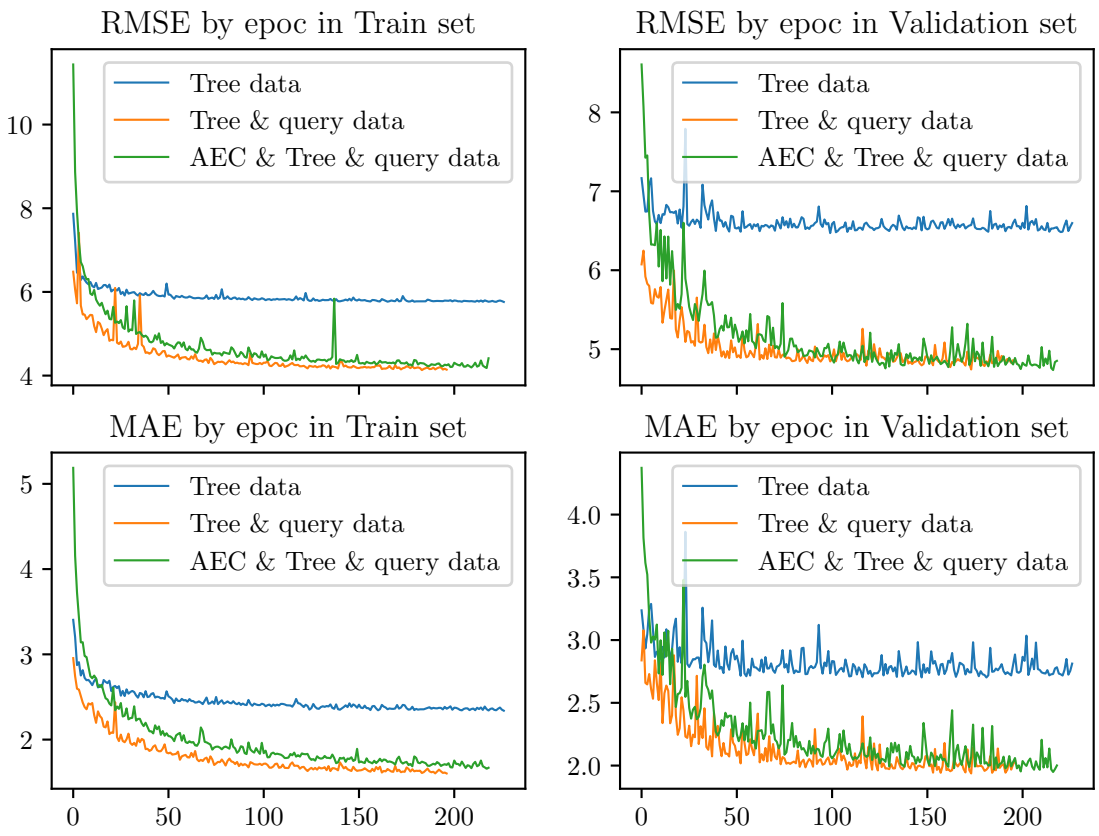

Fig. 9. Comparison of architecture performance using preprocessed and non-preprocessed data with the proposed autoencoder.

- We fix the LeakyReLU activation function.
- We set the learner's learning rate to 0.00015.
- We fix layers at plan level with 512, 256, 128 units per layer.
- We run one network using the best configuration using the best query-level features obtained previously

Figure 9, shows the performance on the training and validation sets of: a network with execution plan level information; a network with query and execution plan level information; and a network similar to the previous one with the plan node level data previously preprocessed by the autoencoder. We see how the use of the autoencoder does not introduce meaningful improvements in terms of model generalization. However, it does have a positive impact on the training time and hardware resources needed to train the model. This is because it significantly decreases the number of network parameters.