NEURON: Query Execution Plan Meets Natural Language Processing For Augmenting DB Education

Siyuan Liu Sourav S Bhowmick Wanlu Zhang Shu Wang
Wanyi Huang Shafiq Joty
School of Computer Science & Engg., Nanyang Technological University, Singapore
sliu019|assourav|zh0012lu|wang1004|hu0011yi|srjoty@ntu.edu.sg

ABSTRACT

A core component of a database systems course at the undergraduate level is the design and implementation of the query optimizer in an RDBMS. The query optimization process produces a *query execution plan* (QEP), which represents an execution strategy for an SQL query. Unfortunately, in practice, it is often difficult for a student to comprehend a query execution strategy by perusing its QEP, hindering her learning process. In this demonstration, we present a novel system called NEURON that facilitates natural language interaction with QEPs to enhance its understanding. NEURON accepts an SQL query (which may include joins, aggregation, nesting, among other things) as input, executes it, and generates a *simplified* natural language description (both in text and voice form) of the execution strategy deployed by the underlying RDBMS. Furthermore, it facilitates understanding of various features related to a QEP through a natural language-based question answering framework. We advocate that such tool, world's first of its kind, can greatly enhance students' learning of the query optimization topic.

ACM Reference Format:
Siyuan Liu, Sourav S Bhowmick, Wanlu Zhang, Shu Wang, Wanyi Huang, Shafiq Joty. 2019. NEURON: Query Execution Plan Meets Natural Language Processing For Augmenting DB Education. In 2019 International Conference on Management of Data (SIGMOD '19), June 30–July 5, 2019, Amsterdam, Netherlands. ACM, NewYork, NY, USA, 4 pages. https://doi.org/10.1145/3299869.3320213

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3320213

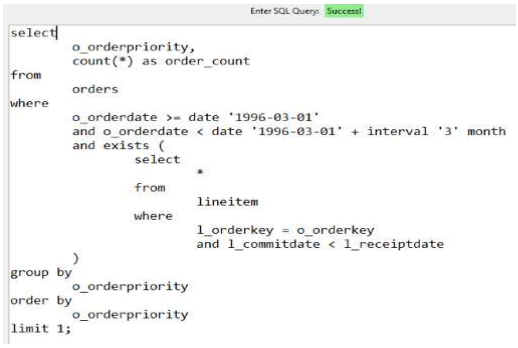

Figure 1: Query 4 in TPC-H benchmark dataset.

1 INTRODUCTION

The database systems course is widely offered in major universities as part of the undergraduate computer science degree program. A core component of this course is the topic of query optimization. Specifically, the query optimization process produces a *query execution plan* (QEP), which represents an execution strategy of an SQL query. Given an SQL query, a student enrolled in a database systems course would typically like to understand how it is executed on the underlying RDBMS by studying the associated QEP. Unfortunately, every commercial database vendor has its own secret sauce for the implementation of the query optimizer. Consequently, comprehension of a QEP not only demands deep knowledge of various query optimization-related concepts but also vendor-specific implementation details. We advocate that this is an unrealistic expectation from an undergraduate student learning database systems for the first time.

Example 1.1. Bob is an undergraduate student majoring in computer science and is currently enrolled in a database course, which uses PostgreSQL 9.6 to teach various concepts. He wishes to understand the QEP of the SQL query in Figure 1 on a TPC-H benchmark dataset[1]. Figure 2 (partially) depicts the QEP generated by PostgreSQL for this query. Unfortunately, Bob finds the textual description of the QEP is not only verbose but it also contains unfamiliar terms (*e.g.*, hash semijoin, bucket, width). Hence, he decided to switch to

[1] http://www.tpc.org.

	QUERY PLAN text
1	Limit (cost=283240.33..283261.81 rows=1 width=24) (actual time=2517.131..2517.131 rows=1 loops=1)
2	-> GroupAggregate (cost=283240.33..283347.75 rows=5 width=24) (actual time=2517.129..2517.129 rows=1 loops=1)
3	Group Key: orders.o_orderpriority
4	-> Sort (cost=283240.33..283276.12 rows=14316 width=16) (actual time=2515.763..2516.253 rows=10570 loops=1)
5	Sort Key: orders.o_orderpriority
6	Sort Method: quicksort Memory: 3811kB
7	-> Hash Semi Join (cost=220326.10..282252.14 rows=14316 width=16) (actual time=1734.275..2498.630 rows=52164 loops=1)
8	Hash Cond: (orders.o_orderkey = lineitem.l_orderkey)
9	-> Seq Scan on orders (cost=0.00..48594.99 rows=56557 width=20) (actual time=0.045..226.711 rows=56965 loops=1)
10	Filter: ((o_orderdate >= '1996-03-01'::date) AND (o_orderdate < '1996-06-01 00:00:00'::timestamp without time zone))
11	Rows Removed by Filter: 1443034
12	-> Hash (cost=187509.83..187509.83 rows=2000182 width=4) (actual time=1733.473..1733.473 rows=3793295 loops=1)
13	Buckets: 131072 (originally 131072) Batches: 64 (originally 32) Memory Usage: 3098kB
14	-> Seq Scan on lineitem (cost=0.00..187509.83 rows=2000182 width=4) (actual time=0.037..1246.522 rows=3793295 loops=1)
15	Filter: (l_commitdate < l_receiptdate)
16	Rows Removed by Filter: 2207919
17	Planning time: 4.656 ms
18	Execution time: 2517.741 ms

Figure 2: A QEP in PostgreSQL (Enlarged view at [5]).

the visual tree representation of the QEP [5] for better comprehension. Although relatively succinct visually, it simply depicts the sequence of operators (*e.g.,* hash → hash semi join → sort → aggregate → limit) used for processing the query, hiding additional details about the query execution. In fact, Bob needs to manually delve into details associated with each node in the tree for further information. ∎

Clearly, an easy and intuitive natural language-based interface can greatly enhance Bob's comprehension of QEPs for SQL queries. However, the majority of natural language interfaces for RDBMS [2–4, 7] have focused either on translating natural language sentences to SQL queries or narrating SQL queries in natural language to naïve users. Scant attention has been paid for the natural language understanding of QEPs.

In this demonstration, we present a novel system called NEURON (**N**atural Languag**E** **U**nderstanding of Que**R**y Execuți**O**n Pla**N**) to facilitate natural language interaction with QEPs in PostgreSQL. Given the QEP of an SQL query, NEURON analyzes it to automatically generate a *simplified* natural language description (both text and voice form) of the key steps undertaken by the underlying RDBMS to execute the query. Furthermore, it supports a question-answering system that allows a user to seek answers to a variety of concepts and features associated with a QEP in natural language.

In this demonstration, we will present a walk-through of the NEURON tool, and explain how it provides a natural language interface to understand QEPs of an RDBMS. We will then show how it can be used to facilitate understanding of various concepts related to QEPs through a natural language-based question answering framework. For example, one may ask questions such as *"What is a hash semi join?"*, *"How many tuples are left after Step 5?"*, and *"What is the most expensive operation?"*.

2 SYSTEM OVERVIEW

NEURON is implemented using Python on top of PostgreSQL 9.6. Figure 3 depicts the architecture of NEURON and mainly consists of the following modules.

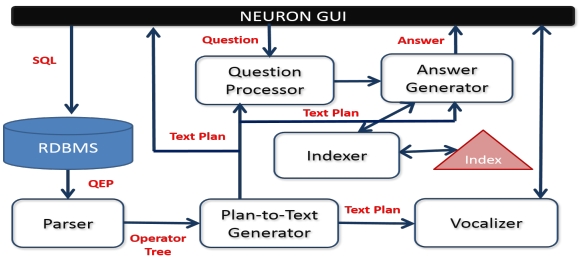

Figure 3: Architecture of NEURON.

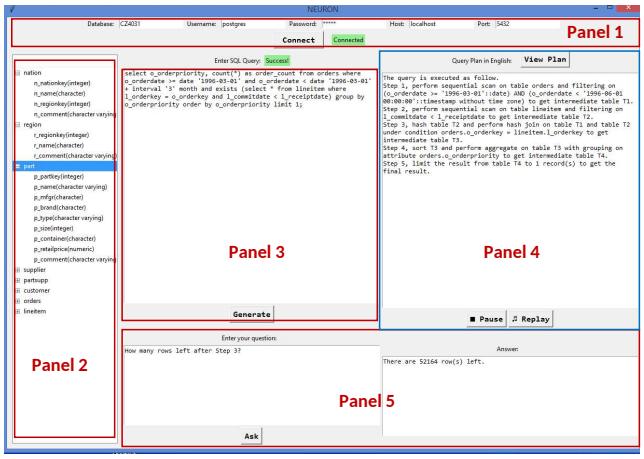

Figure 4: The GUI of NEURON (Enlarged view at [5]).

The GUI module. Figure 4 is a screenshot of the visual interface of NEURON. Panel 1 enables a user to connect to the underlying relational database. Panel 2 shows the schema of the underlying database. A user formulates an SQL query (which may include aggregation, nesting, joins, among other things) in textual format on this database in Panel 3. Upon clicking the `Generate` button, the query is executed and the corresponding execution plan is generated in a natural language (*i.e.,* English) and displayed in Panel 4. Note that NEURON generates both textual as well as vocal forms of an execution plan. A user can click on the `Pause` or the `Replay` buttons to interact with the vocalized form of a plan. Clicking on the `View Plan` button, retrieves the original QEP as generated by PostgreSQL. Panel 5 allows a user to pose questions related to a QEP in a natural language.

The Parser module. The goal of this module is to parse and transform the QEP of an SQL query into an *operator tree*, which is exploited by subsequent modules. Once a user formulates and executes an SQL query in Panel 3, it first invokes the PostgreSQL API (using the *Psycopg adapter*) to obtain the corresponding QEP in JSON format. Then, the plan is parsed and an *operator tree* is constructed. Specifically, each node in an operator tree contains relevant information associated with a plan such as the operator type (*e.g.,* hash join), name of the relation being processed by the node, alias given to intermediate results (*e.g.,* subqueries), column(s) used for

grouping or sorting, name of the index being processed by the node, subplan ids, filtering conditions used during a join or a table scan, conditions used for index-based search, and the number of rows left after an operation. Note that this module ignores information in the original QEP that is not useful for realizing the NEURON framework such as *plan width* and whether a node is *parallel aware*.

The Plan-to-Text Generator module. This module takes an operator tree as input and generates a textual description of the QEP represented by a sequence of steps (*e.g.,* Panel 4 in Figure 4). At first glance, it may seem that we may simply perform a postorder traversal on the operator tree and transform the information contained in each node into a natural language format. However, this naïve approach may generate a verbose description of a QEP containing irrelevant and redundant information. This is because some nodes in an operator tree may not carry any meaningful information as far as textual description of a QEP is concerned. For instance, the node Result is used in PostgreSQL to represent an intermediate relation for storing temporary results. Hence, this module first removes Result nodes from an operator tree.

The modified operator tree contains now two categories of nodes, namely *critical* and *non-critical* nodes. The former nodes represent important operations (*e.g.,* hash join, sort) in a QEP and may contain a large amount of information. The latter nodes are located near critical nodes (*e.g.,* parent, child) but do not carry important information on its own in comparison to the critical ones. Hence, we reduce the modified operator tree further by *merging* non-critical nodes with corresponding critical nodes. Some examples of such merge operation are as follows: (a) The Hash Join node and its child Hash are merged. (b) The Merge Join node and its child Sort are merged. (c) The Bitmap Heap Scan node and its child Bitmap Index Scan are merged. (d) The Aggregate node and its child Sort are merged. (e) The Unique node and its child Sort are merged.

An important issue here is the handling of subqueries in an SQL query. PostgreSQL creates a corresponding subplan for each subquery in a QEP whose return value can be referred to from other parts of a plan. It assigns a temporary name to this subplan for future referral. However, such name should not appear in the natural language representation of a QEP. Thus, we use a dictionary to keep track of subplan names and their corresponding relation names so that when other steps mention the output of a subquery, the referred name will be replaced by the corresponding relation name(s).

Based on the aforementioned strategies, this module traverses the tree in a postorder fashion to generate a sequence of steps (identified by *step id*s) describing a QEP. Each node in the reduced operator tree generates a step and each step is represented as a text description of the node's content based on its type. Specifically, we leverage different *natural language templates* for different node types to generate meaningful statements. In this context, each intermediate result is assigned an identifier to ensure unambiguous reference from a parent operator to its children's results. Filter and join conditions are parsed and converted to human-readable natural language representations. For example, an Index Scan node is converted to the following step: "*Perform index scan on table X (and filtering on X.b = 1) to get intermediate table A*". Figure 4 depicts an example output of this module (in Panel 4) for the QEP in Figure 2.

The Vocalizer module. The goal of this module is to vocalize the natural language description of a QEP by first performing text-to-speech conversion utilizing Google's *Text-to-Speech* (GTTS) API and then playing it using the *Pygame* package (https://www.pygame.org/).

The Indexer module. This module is exploited by the question-answering (QA) framework of NEURON. The QA subsystem accepts a user query as input and returns an answer as output (Panel 5). Note that not all queries related to a QEP can be answered by analyzing a QEP. For example, *"what is a bitmap heap scan?"* cannot be answered simply by analyzing a QEP. To address this challenge, this module first extracts definitions of SQL keywords and query plan operators from relevant Web sources[2] as well as comments associated with the source code of PostgreSQL (https://github.com/postgres/postgres/blob/master/src/include/nodes/plannodes.h). Then a set of documents containing these definitions is indexed using an inverted index (we use the *Whoosh* Python library) where each document contains the definition of a single SQL keyword or a query operator. The words in the documents are lemmatized and stop words are removed during this process.

The Question Processor module. Once a user enters a question related to a QEP through Panel 5, the goal of this module is to *classify* the question, and extract the part-of-speech (POS) tags and keywords in it. Consequently, it consists of the following three submodules.

The Question Classifier submodule. The current implementation of NEURON supports five categories of questions: (a) definitions of various SQL keywords and query plan operators; (b) the number of tuples generated at a specific step; (c) the list of operators used to evaluate a query; (d) the amount of time taken by specific step(s) in a QEP; and (e) finding the *dominant* (*i.e.,* most expensive) operator in a QEP. Hence, given a user's question, its category needs to be identified first before it can be answered. The goal of this submodule is to classify a user's question into one of these five categories. To this end, it adopts a Naive Bayes classifier. A set of training questions (67 questions) is prepared manually together

[2] https://www.postgresql.org/docs/10/static/sql-commands.html, http://use-the-index-luke.com/sql/explain-plan/postgresql/operations, https://www.postgresql.org/message-id/12553.1135634231@sss.pgh.pa.us

with their true categories. The features used in the classifier are the unigrams (bag of words).

Given a user's question, the unigram features are generated and the category is determined by applying the classifier.

The Part-of-speech (POS) Tagger submodule. This submodule extracts the part-of-speech (POS) tags in a question (using the *TextBlob* Python library). POS tags are used to find the *step id* (*i.e., id* of a step in Panel 4) inside a question related to Categories (b) and (d).

The Keyword extractor submodule. To answer questions related to Category (a), it is paramount to identify keywords in the question so that we know what is being asked. This submodule extracts the keywords by first removing stop words. The list of English stop words is obtained from the NLTK Python library (http://www.nltk.org/). The word "only" is excluded as it is one of the keywords for query operators (*e.g.,* Index Only Scan). The remaining words are lemmatized and duplicate words are eliminated.

The Answer Generator module. This module aims to retrieve the correct answer based on the question category by exploiting the following different submodules.

The Concept Definition submodule. If a question belongs to Category (a) then it uses keywords extracted from it to retrieve the relevant document containing the definition using the index.

The Row Count submodule. To answer questions regarding the number of rows after a certain step (Category (b)), the *step id* must be supplied to the question. Note that questions in the form of "*number of rows left after joining relations A and B*" (*i.e.,* without step id) are not supported as two or more joins on same relations but different columns may be performed in a single query, leading to ambiguity.

The submodule extracts the step id by finding word with the POS tag CD (cardinal number) in a question. After that, the operator tree is traversed to find the node that the step id belongs to. The number of rows is then retrieved from the Actual Rows element associated with this node.

The Operator List submodule. The operator tree is traversed to retrieve the distinct list of operators used in a QEP (Category (c)).

The Total Time submodule. To answer questions regarding Category (d), similar to Category (b) questions, the *step id* must be supplied to a question. It traverses the operator tree to retrieve the total time of a specific step, which is calculated based on the Actual Total Time element of the corresponding node and its children.

The Dominant Operator submodule. To find the most expensive operator in a QEP (Category (e)), NEURON computes the total time taken by each operator and returns the one with longest time.

Note that the answers are formatted using natural language templates to generate meaningful statements.

3 RELATED SYSTEMS AND NOVELTY

Natural language interfaces to relational databases have been studied for several decades [1, 3, 4, 6, 7]. Given a logically complex English language sentence as query input, the goal of majority of these work is to translate it to SQL. On the other hand, frameworks such as Logos [2] explain SQL queries to users using a natural language. NEURON compliments these efforts by providing a natural language explanation of the QEP of a given SQL query. It further supports a natural language-based QA framework that enables users to ask questions related to the plan.

4 DEMONSTRATION OBJECTIVES

Our demonstration will be loaded with TPC-H benchmark (we use the TPC-H v2.17.3) and DBLP datasets. For DBLP, we download the XML snapshot of the data and then store them in 10 relations. Example SQL queries on these datasets will be presented. Users can also write their own ad-hoc queries.

The audience will be requested to formulate a SQL query or select one from the list of benchmark queries using the NEURON GUI. Upon execution of the query, one will be able to view as well as hear the natural language description of the QEP. She may pause and replay the natural language description as she wishes. By clicking on the `View Plan` button, one can view the original QEP generated by PostgreSQL and appreciate the difficulty in perusing and comprehending plan details, highlighting the benefits of natural language interaction brought by NEURON. Lastly, the audience can pose the aforementioned types of questions related to a QEP through the NEURON GUI and get accurate answers in real-time. Such QA session aims to facilitate further natural language-based clarification regarding the execution strategy deployed by the underlying query engine. A short video to illustrate these features of NEURON is available at https://youtu.be/wRIWuYbU2F0.

Acknowledgments. Sourav S Bhowmick and Shafiq Joty are supported by AcRF Tier-1 Grant 2018-T1-001-134.

REFERENCES

[1] F. Basik, et al. DBPal: A Learned NL-Interface for Databases. *In SIGMOD*, 2018.
[2] A. Kokkalis, P. Vagenas, A. Zervakis, A. Simitsis, G. Koutrika, Y. E. Ioannidis. Logos: A System for Translating Queries into Narratives. *In SIGMOD*, 2012.
[3] F. Li, H. V. Jagadish. NaLIR: An Interactive Natural Language Interface for Querying Relational Databases. *In SIGMOD*, 2014.
[4] F. Li, H. V. Jagadish. Constructing an Interactive Natural Language Interface for Relational Databases. *PVLDB*, 8(1), 2014.
[5] S. Liu, et al. NEURON: Query Optimization Meets Natural Language Processing For Augmenting Database Education. https://arxiv.org/pdf/1805.05670.pdf.
[6] A.-M. Popescu, O. Etzioni, H. A. Kautz. Towards a Theory of Natural Language Interfaces to Databases. *In IUI*, 2003.
[7] D. Saha, A. Floratou, et al. ATHENA: An Ontology-Driven System for Natural Language Querying over Relational Data Stores. *PVLDB*, 9(12), 2016.