Database Workload Characterization with Query Plan Encoders

Debjyoti Paul[†]
School of Computing, University of Utah
Salt Lake City, Utah
deb@cs.utah.edu

Jie Cao[†]
School of Computing, University of Utah
Salt Lake City, Utah
jcao@cs.utah.edu

Feifei Li
School of Computing, University of Utah
Salt Lake City, Utah
lifeifei@cs.utah.edu

Vivek Srikumar
School of Computing, University of Utah
Salt Lake City, Utah
svivek@cs.utah.edu

ABSTRACT

Smart databases are adopting artificial intelligence (AI) technologies to achieve *instance optimality*, and in the future, databases will come with prepackaged AI models within their core components. The reason is that every database runs on different workloads, demands specific resources, and settings to achieve optimal performance. It prompts the necessity to understand workloads running in the system along with their features comprehensively, which we dub as workload characterization.

To address this workload characterization problem, we propose our query plan encoders that learn essential features and their correlations from query plans. Our pretrained encoders captures the *structural* and the *computational performance* of queries independently. We show that our pretrained encoders are adaptable to workloads that expedites the transfer learning process. We performed independent assessments of structural encoder and performance encoders with multiple downstream tasks. For the overall evaluation of our query plan encoders, we architect two downstream tasks (i) query latency prediction and (ii) query classification. These tasks show the importance of feature-based workload characterization. We also performed extensive experiments on individual encoders to verify the effectiveness of representation learning, and domain adaptability.

PVLDB Reference Format:
Debjyoti Paul, Jie Cao, Feifei Li, Vivek Srikumar. Database Workload Characterization with Query Plan Encoders. PVLDB, 15(4): 923 - 935, 2022.
doi:10.14778/3503585.3503600

1 INTRODUCTION

Database Management Systems (DBMS) are general-purpose systems that aim to provide solutions to as many applications as possible. Database designers expose many configuration settings to facilitate end-users in managing complex workloads efficiently. However, there is no single configuration that works for all workloads, and finding the optimal configuration setting is very dependent on the workload characteristics.

In the usual process, DBAs first need to learn about the database queries that frequently run on their database system and then dig deeper to characterize these queries. It requires in-depth knowledge and a robust understanding of the queries and their execution features. It is a challenging and laborious task for DBAs to comprehend the execution features of queries and their relations with configuration knobs. Furthermore, the large number of possible DBMS configurations settings make it a daunting task for DBAs. Advanced DBAs apply simple data mining techniques and hand-tweaked feature engineering to understand the nature of the workload, but this requires domain expertise, which is rare.

Nowadays, many small to medium businesses (SMBs) manage their databases with cloud services. Cloud database service providers can now obtain and analyze large amounts of anonymized workload data. Managing database resources efficiently is indispensable for providing quality services. Each database instance runs a different workload. Applying data science can help identify workloads with similar characteristics, and then it can be used in downstream tasks, e.g., query optimization, configuration recommendation, and index recommendation. Essentially, it raises a requirement of database workload characterization, i.e., the ability to describe the distinctive nature and features of queries in a workload.

Previous work [32] shows with TPC-H benchmarks how each database query behaves differently with changes in database configuration settings. For example, query Q18 and query Q7 in TPC-H benchmark responds to knob changes `shared_buffers` vs. `effective_cache_size` very differently w.r.t. *query latency*. Each query possesses distinct features, and the demands for computational resources are also different. It suggests that each query needs to be treated uniquely and based on their characteristic. Recent research works [10, 11, 17] leverages query plans as the *feature description of queries* and use it for tasks like index recommendation [10, 11] and configuration knob tuning [17].

In the natural language domain, a word is a structural and functional unit of a meaningful sentence. Similarly, in the database domain, a query is the structural unit, and a database query plan is the functional unit of a workload. With the advancement in the distributed representation of words, the downstream tasks like sentence similarity, question answering, and textual entailment have improved dramatically [9, 21, 39]. Likewise, we foresee that downstream tasks like workload similarity, index recommendation, and

[†]These authors contributed equally.

Proceedings of the VLDB Endowment, Vol. 15, No. 4 ISSN 2150-8097.
doi:10.14778/3503585.3503600

database configuration recommendation can benefit from the study of workload characterization.

We propose a scalable data-driven artificial intelligence (AI) approach for workload characterization with a distributed representation of query plans. One of the benefits of AI deep learning models is automatic feature engineering and auto-correlation among features. It is a non-trivial arduous task and possesses many challenges in achieving the aim of workload characterization. Some of the challenges that make it very different from other entity representation learning are *Query Independence, Diverse Query Structure, Modeling Computational Complexity,* and *Data Dependence.* We present a constructive detail for each aforementioned challenges in §2.3.

Our Approach. In our work, we first propose a query plan distributed representation model that captures the inherent characteristics such as *structure, computational demand*, and *feature manifests* embedded within a query plan structure. Hence, we created two parts for query plan representation, *(i) Structure Representation, (ii) Computational Performance Representation.* The two representations, either separately or collectively, can be used in downstream tasks to understand a query comprehensively. As an example, we demonstrate an approach to perform query latency prediction with the help of query representations. It can help in offline profiling of workloads and aid in tuning database settings. We believe that instance optimality of a database can only be achieved with the in-depth understanding of queries running in a system, and suggests the introduction of workload characterization component for it.

In our choice of design for distributed representation, we can either use a *fixed-embedding* or a *pretrained encoder* approach. Fixed embedding is useful where the set of elements is complete, and after model training, we get a fixed representation for all the elements in the set. This approach is instrumental in domains like graph embedding. On the other hand, a pretrained encoder is a learned model that can output embedding on receiving the input by featurizing the input attributes and learned weights from previous observations. We follow the pretrained encoder approach for adaptability and transfer of knowledge.

Furthermore, we follow a bidirectional encoder strategy with both *feature-based* and *finetuning-based* approach inspired by the language models [9]. In this approach, the embedding obtained from the pretrained encoder is trained to learn features, and then the feature embedding output can be fed to multiple task-specific models. The approach aims to alleviate the requirement of task-specific representation and facilitate the reuse of already learned features from the encoder to multiple domain-specific tasks. A pretrained plan representation model also simplifies the transfer learning process when trained on a large dataset and fine-tuned for a specific data and problem set.

We summarize the contribution of this paper.

- We propose plan encoders for distributed representation of query plans. The general feature-based encoders capture inherent characteristics of query plans.
- We capture two aspects of the query plans independently with two classes of encoders. The *structure* , and the *computation* of query plans.
- The *structure* encoder is inspired by the natural language model, representing a tree structure of heterogeneous operators in a latent multidimensional space. Consequently, we evaluate our structure encoder model with similar query classification and regression tasks on multiple datasets.
- Our *computational* encoder is a collection of encoder instances. Each encoder corresponds to a database operator such as scan, join, sort, aggregate, etc., optimizing for multiple metrics to capture the computational features. The encoder uses statistical information and data distribution of the underlying relational data along with the explicitly specified plan features and database configurations.
- We suggest a pretraining approach for our encoders with a large dataset of diverse query plans and database benchmarks. We then introduce a finetuning-based approach that can quickly adapt to new data distribution with limited data resources. It is essential for increment learning and fast domain adaptation with new workloads.
- To show the overall effectiveness of our encoders, we performed query latency prediction and query classification tasks. In query latency prediction, given a query plan and a database configuration setting, the downstream model predicts the query latency using our plan encoders. In the query classification task, we use our plan encoders to classify closely related queries.

The rest of the paper is organized as follows. §2 provides background and challenges we face while performing query plan representation, respectively. In §3, we present our structure encoder and performance encoder, followed by downstream tasks using plan encoders in §4. We present experiments and results of our downstream tasks with plan encoders in §5, and analysis of individual encoders in §6. We present a brief section on related works in §7, followed by conclusion in §8.

2 PRELIMINARIES

Recently we are noticing a trend of utilizing the power of Artificial Intelligence (AI) in buffer resource tuning, indexing, and query optimizer [15, 19, 31]. Soon, we expect database systems packaged with pretrained AI models and dedicated cloud servers with embedded AI accelerators to facilitate the processing. Our proposed workload characterization with a distributed representation of query plans can empower database core components to operate efficiently with in-depth insights on workloads.

2.1 Workload, Query and Query Plan.

We define a database workload as $W = \{(q_1, \theta_1), (q_2, \theta_2), .., (q_n, \theta_n)\}$, where q_i is the database query, and θ_i is a normalized weight of importance of q_i in workload W such that $\sum_{i=1}^{n} \theta_i = 1$. The weight θ_i can be as simple as the frequency of appearance of q_i in W or can be arbitrarily decided by the DBA. Generally, database users mostly run a set of predefined template queries with seldom ad-hoc queries on databases. A data-driven smart database should collect query frequencies and resource usages (e.g., memory, latency, cost, blocks read/write, etc.) to determine popular (transactional/analytics intensive) workloads for choosing optimal database configuration.

For each query q_i, one can obtain the corresponding query plan p_i from the database system. Also, to note that a query with a similar template can generate a different query execution plan or *query-plan* based on the meta-information of a table in a database. Let us

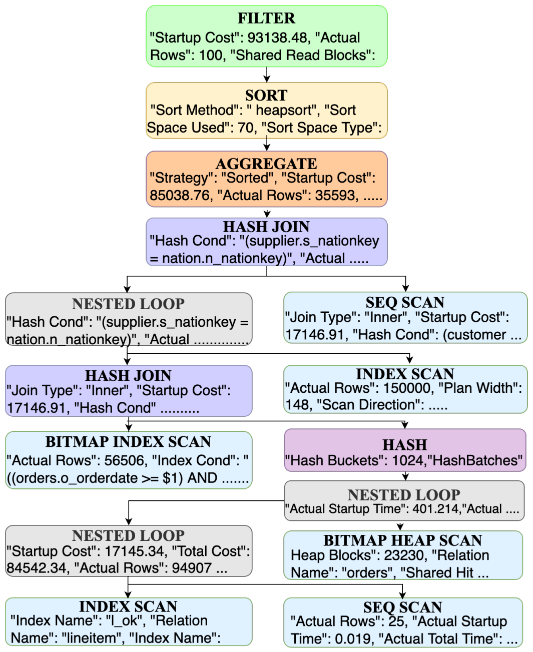

FILTER
"Startup Cost": 93138.48, "Actual Rows": 100, "Shared Read Blocks":
SORT
"Sort Method": " heapsort", "Sort Space Used": 70, "Sort Space Type":
AGGREGATE
"Strategy": "Sorted", "Startup Cost": 85038.76, "Actual Rows": 35593, ...
HASH JOIN
"Hash Cond": "(supplier.s_nationkey = nation.n_nationkey)", "Actual .....
NESTED LOOP
"Hash Cond": "(supplier.s_nationkey = nation.n_nationkey)", "Actual ....
SEQ SCAN
"Join Type": "Inner", "Startup Cost": 17146.91, "Hash Cond": (customer ...
HASH JOIN
"Join Type": "Inner", "Startup Cost": 17146.91, "Hash Cond" ....
INDEX SCAN
"Actual Rows": 150000, "Plan Width": 148, "Scan Direction": .....
BITMAP INDEX SCAN
"Actual Rows": 56506, "Index Cond": " ((orders.o_orderdate >= $1) AND ....
HASH
"Hash Buckets": 1024,"HashBatches"
NESTED LOOP
"Actual Startup Time": 401.214,"Actual ....
NESTED LOOP
"Startup Cost": 17145.34, "Total Cost": 84542.34, "Actual Rows": 94907 ...
BITMAP HEAP SCAN
Heap Blocks": 23230, "Relation Name": "orders", "Shared Hit ...
INDEX SCAN
"Index Name": "l_ok", "Relation Name": "lineitem", "Index Name":
SEQ SCAN
"Actual Rows": 25, "Actual Startup Time": 0.019, "Actual Total Time": ...

Figure 1: A query execution plan from TPC-H[33].

Table 1: The properties from query execution plan that are common to all the operators and a few specific to major operators like Scan, Join, Sort and Aggregate.

Operator	Plan Properties or Features
All	Actual Loops , Actual Rows , Local Dirtied Blocks , Local Hit Blocks , Local Read Blocks , Local Written Blocks , Plan Rows , Plan Width , Shared Dirtied Blocks , Shared Hit Blocks , Shared Read Blocks , Shared Written Blocks , Temp Read Blocks , Temp Written Blocks , Parent Relationship , Plan Buffers
Scan	Relation Name, Scan Direction, Index Name, Index Condition, Scan Condition, Filter, Rows Removed, Heap Blocks, Parallel, Recheck Condition
Join	Join Type, Inner Unique, Merge Condition, Hash Condition, Rows Removed by Join Filter, Parent Relationship, Hash Algorithm, Hash Algo, Hash Buckets, Hash Batches, Peak Memory
Sort	Sort Type, Sort Method, Sort Space, Sort Key, Sort Space Type, Sort Space Used, Peak Memory
Aggregate	Strategy, Hash Algo, Hash Buckets, Hash Batches, Parallel Aware, Partial Mode, Peak Memory

say, q_i generates two query plan p_k and query plan p_l on different instances. It is safe to assume and treat both the queries differently from the functional point of view in our approach. Hence, there can be a one-to-many mapping from queries to query plans.

Alternatively, we can now define workload as $W = \{(p_1, \theta_1), (p_2, \theta_2), \ldots, (p_m, \theta_m)\}$, where p_i is the database query-plan, and θ_i is a normalized weight of importance of p_i in workload W such that $\sum_{i=1}^{n} \theta_i = 1$. For readability, we will refer a query-plan as a plan in the paper from now.

A plan is a tree structure with heterogeneous functional operator nodes like *Seq Scan, Index Scan, Bitmap Heap Scan, Nested Loop, Hash Join, Aggregate, Sort, Filter* etc. Each operator node contains a set of execution properties. We present an example of a plan in Figure 1 of query Q5 from the TPC-H benchmark with operator types. All operators have a set of common properties, and in addition, a few contain specific properties based on their functions. These operator properties carry valuable information about their execution. Based on the functions of each operator, we grouped all operators into five exclusive groups, i.e., *Scan, Join, Aggregate, Join* and *others.* In Table 1, we lay out the properties common to all groups as 'All' and the properties exclusive to Scan, Join, Sort, and Aggregate operators. These operator properties are used for computational performance representation of the plan. Please note that we do not use properties like Total Cost, Actual Total Time, Actual Startup Cost because we use them as labels in our prediction tasks. We describe it in §3.2.

For any plan p_i as input to our *Structural Encoder* and *Computational Encoder*, the models outputs the structural embedding $\mathcal{S}(p_i)$ and the computational performance embedding $C(p_i)$, respectively. These embeddings are used by downstream models for different fine-tuning tasks.

2.2 Deep Neural Networks (DNNs)

DNNs are widely used computational frameworks for many AI applications. DNNs are layers of neuron thoughtfully structured that performs a weighted sum computation of the input values at each neuron. A structure of DNNs or *model* is also an instance of a machine learning algorithm that learns patterns from data with inferences and then by readjusting weights to minimize error. DNNs are very efficient in reducing high dimensional data into low dimensional code, or features [14]. DNN hardly requires feature engineering and can learn complex relations among multiple features. In our paper, we are specifically interested in the entity representation learning capability of DNNs. Moreover, we focus our attention on Autoencoder (an Encoder-Decoder approach) for learning the *structural* representation model. A particular kind of Autoencoder called Denoising Autoencoder can capture robust generalized features from original data [37]. We applied an advanced feature-based encoding and learning technique inspired by natural language models. Recent applications of encoder architectures on language models are very successful in capturing structural and statistical properties [9, 39]. Query plans are structurally complex, and properties of plan operators are implicitly correlated. Hence, we adapted the autoencoder approach in our representation models. For the *computational performance* representation, we used a supervised learning approach to learn metrics from features.

2.3 Challenges and Mitigation Strategies

Traditional machine learning approaches encode entities into a fixed-length features before feeding them into any model for prediction tasks. We provide a consolidated set of challenges we face while performing workload characterization with plan encoders because of heterogeneous nature, diverse shape, and varying depth of plans.

- *Query Independence:* Each query is unique and independent. Even if the queries are from the same benchmark or workload; they are seldom similar in structural and computational complexity. Unlike other entity embeddings where contextual appearances of entities play pivotal importance (such as word embedding), in workload contextual or temporal appearance of queries are not related.
- *Diverse Query Structure:* The structure of query plans is represented as a tree of functional operator nodes, e.g., *scan, join,*

sort etc. It is a non-trivial task to represent a tree structure containing attribute features at every node.

- *Modeling Computational Complexity:* Each query has a specific demand for computational resources based on their functional operations. Moreover, the resource demand of each functional operator is different. An open question arises whether to implement an operator-level model or a single primary model for encoding.
- *Data Dependence:* In databases, the generation of query plans from a query depends on many factors, such as index availability, statistical information on data. A complete query plan can only capture basic information about underlying data. It raises the question of whether it is enough or we need to incorporate more information.
- *Encoding Multiple Properties:* Database plans contain interrelated properties and information that give hints about query performance and their execution metrics such as latency and throughput. It is a challenge to unify and discover complex correlations among the properties and features explicitly obtained from plans.
- *Domain Adaptation:* The encoder models trained on a set of workloads are likely to encounter a different unseen workload in the prediction phase. It is a challenge to quickly adapt to a new workload setting (with less training data) using the prior pretrained weights of the models.

We adopted specific strategies in our approach addressing the above challenges. We purposefully design a feature-based query plan encoder for learning the individual characteristics from different query plans. For modeling the performance complexity, we incorporate meta-information (e.g., data distribution, selectivity, cardinality) of database tables and attributes used in queries providing a detailed picture of the data access pattern.

It is a not trivial attempt to incorporate all the relevant meta-information and capture relevant features in our plan performance representation. Still, it is reasonable to assume that if we can incorporate all the required information to the encoders, we might be able to learn the influencing factors contributing to the evaluation metrics of query plans. After all, query optimizers are universally designed logical components that generate query plans. The encoders producing a distributed representation of query plans can facilitate many downstream tasks and enhance the performance of core components. It encourages us to keep the encoder as general as possible and capture the correlation among properties well enough in the query plan representation. We aim to create a pretrained encoder model that learns from large and diverse datasets to learn plan features with a data-driven approach. In the ideal scenario, we want pretrained encoders to quickly adapt to new domains with less dataset, expediting domain transfer.

3 QUERY PLAN REPRESENTATION

In this section, we present our *Structure Encoder* and *Computational Performance Encoder* for plans. Each node in the tree is a functional operator with multiple properties, and nodes are ordered and connected via unlabeled edges depicting the dependence relation. For structural representation, we mainly study the operator type of each node and leaving the performance-related properties for *computational performance representation* in §3.2. When sketching our encoders, we realize that keeping separate *structure*, and *computational performance* representation enables downstream tasks to choose and weigh each representation independently in their model and introduces modular design. It also helps us in evaluating the structure and performance encoders separately.

For both Structure Encoder and Computational Encoder, we aim that our pretrained model can easily be adapted by new applications. Hence, we study both of them on a two-stage framework: pretraining and finetuning. In this section, we mainly introduce the pretraining tasks and model architectures for them. Then we outline our finetuning evaluation in §3.3.

3.1 Structure Encoder

We first try to give a clear picture of the diverse types of operators in plans and how we define a taxonomy for them. Same functional operators can use different strategies to fulfill their operations. There are multiple types of Scan operators like Sequential Scan, Index Scan, Bitmap Heap Scan, etc. Again, the same strategy is often used in multiple functional operators, like, Hash Join and Hash Aggregate use Hash strategy. We organized each type of operator into three sub-level types as a taxonomy of operators. The top-level Level 1 mostly suggest functional properties such as Sort, Insert, Union, Scan, Join, etc. FLevel 2 and Level 3 are grouped based on mutually exclusive strategy types such as Hash, Index, Heap, etc. Table 2 shows all three levels of operator sub-type for defining a real operator. We define all operator with three sub-type as ⟨Level 1⟩-⟨Level 2⟩-⟨Level 3⟩. For example, operator *Bitmap Heap Scan* and *Left Merge Join* is represented as Scan-Heap-Bitmap and Join-Merge-Left, respectively. All these operator types form the tree structure as shown in Figure 1, we need to find a way to encode the tree. Notice that workload analysis based on similar query plans can help DBAs in optimal utilization of database resources, e.g., buffers and configuration, by utilizing historical experiences from other databases. Furthermore, encoders enable the clustering of similar-featured queries learned from a large set of queries without actually sharing any private/sensitive query information. Inspired by this goal, we propose a plan-pair similarity regression task to guide structural representation learning.

3.1.1 Plan-pair Similarity Regression. For pretraining our structure plan encoder, we need a dataset of plan pairs with their similarity scores, but obtaining such a dataset is challenging because this is a graph similarity matching and scoring problem [40]. We came up with a method to generate a bootstrapping training dataset, using a widely used graph similarity metric for natural language representation domain: *Smatch* [6]. It calculates the degree of overlap between two graph structures, defined as the maximum F1-score obtainable via a one-to-one matching of each node in two graphs. Hence it is a value from 0 to 1, 0 means very different, while 1 means exactly the same. In this task, we treat the optimal Smatch score as the similarity of the two plans. The Smatch score between two tree-structure plans can be computed by graph matching optimization algorithm, such as Integer Linear Programming (ILP) or Hill-climbing methods. After we get the *Smatch* scores s_{ij} of each plan-pair $\langle p_i, p_j \rangle$, this can easily form a large dataset with *Smatch*

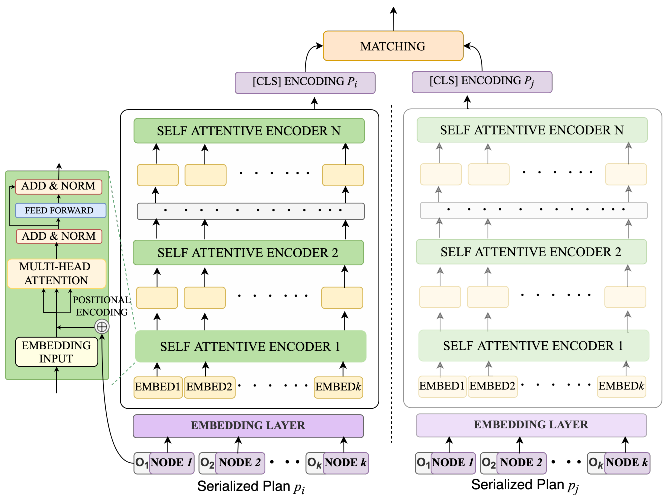


Figure 2: Plan Structure Encoder Model. Serialized plan p_i and p_j with node positional information denoted with O.

score as the similarity supervision. We first train our structure encoder to predict the *Smatch* score of each plan pair. To note that the idea is not to learn the Smatch but to learn contrast features from plans. Later in our experiments on the downstream applications, we show that the structure encoder pretrained from this task can be easily finetuned for a new task or domain.

3.1.2 Model Architecture. We can keep the plan tree structure intact and use tree-encoding architecture (such as, tree-LSTMs [30]) or use serialize methods to treat it as a sequence encoding problem with positional encodings. In tree-LSTMs information, flows are only through immediate neighbors, and it needs separate attention mechanism for contexts among the nodes of sibling subtrees [1, 26]. For query plans (with many join and select permutations), we encourage keeping wider contexts from a neighbor sub-tree siblings, and that's why find self-attention model with positional encodings a simple and better approach.

We use the depth root first traversal to serialize plans, with a simple yet ingenuity hack by adding hierarchical brackets for each non-terminal node in the tree. An open bracket always encapsulates sub-trees at the start and a closing bracket at the end; this is less ambiguous than simple *BFS* and *DFS* tree traversal strategies. These brackets preserve positional information of the structure and are then utilized inherently by our self-attentive encoder with positional encoding. We present a running example of our *DFS-Bracket* strategy in Table 3 for the plan in Figure 1.

Self-attentive Encoder Layer. We employ the multi-head, multi-hop attention mechanism used in Transformer networks [36] pictorially presented in Figure 2. Due to space constraints, we refer readers to the original work for details. We use same **Q**: attention query[1], **K**: key, **V**: value matrices notation from the original paper.

The multi-head attention is defined as,

$$\text{Multihead}(\mathbf{Q},\mathbf{K},\mathbf{V}) = [\text{head}_1 \circ \ldots \circ \text{head}_h]\mathbf{W}^O \tag{1}$$

$$\text{head}_i = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{W}_i^Q\left(\mathbf{K}\mathbf{W}_i^K\right)^T}{\sqrt{d}}\right)\mathbf{V}\mathbf{W}_i^V \tag{2}$$

[1]Note that attention query Q is different from query plan p_i

Table 2: The taxonomy of operator types for every node

Level	Operator Sub-types
Level 1	Aggregate, Append, Count, Delete, Enum, Gather, Aggregate (Group, GroupAggregate), Hash, Insert, Intersect, Join (Nested Loop), Limit, LockRows, Loop, ModifyTable, Network, Result, Scan, Sequence, Set(SetOp), Sort, Union, Unique, Update, Window, WindowAgg, Materialize
Level 2	And, CTE (Common Table Expressions), Except, Exists, Foreign, Hash, Heap, Index, IndexOnly, LoopHash, Merge, Or, Query, Quick, Seq, SetOp, Subquery, Table, WorkTable
Level 3	Anti, Bitmap, Full, Left, Parallel, Partial, Partition, Right, Semi, XN (parallel operators)

Table 3: Running examples for DFS-Bracket traversal Strategies. We use hyphens to connect 3 subtypes. When no subtype for the node, we denote it as NIL **type, here we use blank space for it to save table space. For example, the first node** Filter- **actually means the first subtype is** Filter**, the second and the third subtype is** NIL**.**

Strategy	Node Sequence
DFS Bracket	(Filter–, (Sort–, (Aggregate–, (Join-Hash-, (Loop–Nested, (Join-Hash-, (Hash–, (Loop–Nested, (Loop–Nested, Scan-Index-, Scan-Seq-) Scan-Heap-Bitmap)) Scan-Index-Bitmap) Scan-Index-) Scan-Seq-))))

The $\mathbf{W}_i$'s refer to projection matrices for the three inputs and the final $\mathbf{W}^o$ projects the concatenated heads into a single vector, and $\frac{1}{\sqrt{d}}$ is scaling factor where d is the dimension of $\mathbf{Q}, \mathbf{K}$, and $\mathbf{V}$. $\circ$ means concatenating the encoding attended by multiple heads.

The choices of the attention query, key, and value define the attention mechanism. In our work, we use *self-attention*, defined by setting all three matrices to $[\mathbf{n}_{j1} \ldots \mathbf{n}_{jk}]$, where $\mathbf{n}_{jk}$ is the input encoding of the jth self-attentive layer, which is corresponding the encoding of the kth node in the serialized version along with positional information.

Input Embedding Layer. We represent every plan operator node as a concatenation of embedding of the three subtypes as given in Table 2. Besides these regular nodes, we also added four special nodes CLS, SEP, BR_OPEN, and BR_CLOSE for a start, end, bracket open, and close in the serialized plan sequence, respectively. For positional encoding in a self-attentive layer, we keep track of bracket states for any input sequence with a list. This list counts the number of opening brackets for all the levels till that node in the serialized plan. It is a simpler tree position encoding scheme inspired by the work of Shiv et al [27]. We restrict our discussion due to space constraints, though we present a few examples of list states and the encoded positional information here.

a. (((→ 1,1,1 → [0,0,1,0,0,1,0,0,1,0,0,0],
b. (()((→ 1,2,1 → [0,0,1,0,1,0,0,0,1,0,0,0],
c. ((())((→ 1,1,2,2 → [0,0,1,0,0,1,0,1,0,0,1,0].

Matching Layer The output of the self-attention encoder is a sequence of vector for each nodes, we use the output encoding of CLS node as the encoding of the plan p_i, because it aggregates the weighted sum of all other nodes in the self-attentive layer. We denote the plan encoding for p_i as $P_i \in \mathbb{R}^d$. After encoding the plan-pair $<p_i, p_j>$into vectors $<P_iP_j>$, then we use a matching layer to compute the similarity as

$$\sigma(W * [P_i \circ P_j \circ (P_i - P_j) \circ (P_iP_j)] + b)$$

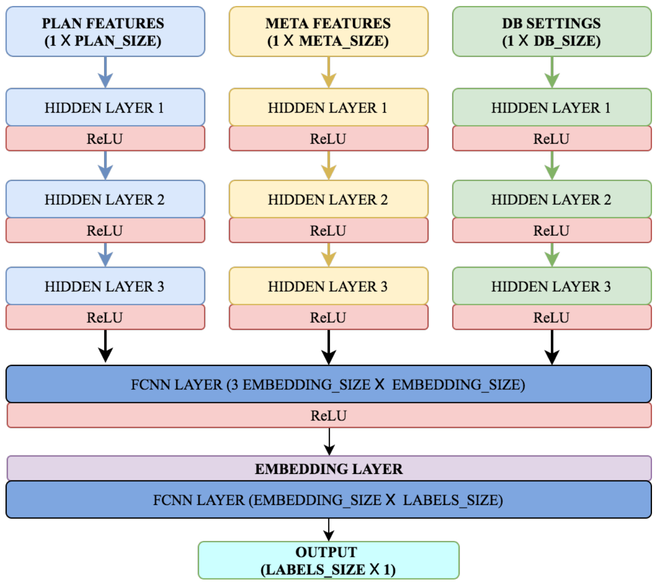

PLAN FEATURES (1 X PLAN_SIZE)
META FEATURES (1 X META_SIZE)
DB SETTINGS (1 X DB_SIZE)
HIDDEN LAYER 1
ReLU
HIDDEN LAYER 2
ReLU
HIDDEN LAYER 3
ReLU
FCNN LAYER (3 EMBEDDING_SIZE X EMBEDDING_SIZE)
ReLU
EMBEDDING LAYER
FCNN LAYER (EMBEDDING_SIZE X LABELS_SIZE)
OUTPUT (LABELS_SIZE X 1)

Figure 3: The multi-column deep neural network(DNN) for our computational performance encoder.

where σ denotes the sigmoid activation function, $W \in \mathbb{R}^{4d}$, b is bias, and $\circ$ are the concat operators on four vectors.[2]

3.2 Computational Performance Encoder

This section presents our computational performance encoder, describing the pretraining task to supervise the encoder learning and our proposed model architectures and intuitions.

3.2.1 Performance Attribute Prediction. The properties mentioned in Table 1 for each type of broadly classified operator in a plan give an ample hint on its computational demand. These properties are either derived from complex logical inferences by a plan optimizer or actual output from the query execution. In previous works [10, 11, 19], we notice the use of Total Cost, Total Time, Startup Time properties as a measure of performance. We strongly agree with previous research works on using the properties above-mentioned as measures of computational performance. Moreover, in our encoder, we use these attributes as labels for prediction to encode the underlying features. We use properties explicitly mentioned in nodes (an instance of an operator in a plan), meta-information from databases, and database configuration settings to predict these labels. In the process of learning the labels, we learn the implicit features as embedding with our computational performance encoder.

We first create encoders, each for (i) Scan (ii) Join (iii) Sort (iv) Aggregate functional operators, these four operators are the most frequently used in query plans. The nodes with operator type Hash Join, Merge Join, Nested Loop, Left/Right/ Inner/Outer Merge Join, Nested Loop is mapped to Join; similarly, Seq Scan, Index Scan, Heap Scan, Bitmap Heap Scan is mapped to Scan. From the properties of each node, we also extract the relation names and attribute names from which it is accessing the data from node properties such as Relation Name, Hash/Join/Merge/Index Condition, Filter,

[2]Other match function exists, e.g. bilinear similarity $P_i M P_j^T$, $M \in \mathbb{R}^{d \times d}$. However, we found that this contanated matching similarity can largely reduce the parameters size from d^2 to $4d$ and achieve better performance.

Table 4: Meta Features and DB Settings used as input features to Computational Performance Encoder

Features Type	Feature Attributes
Meta Features	rel_name, att_name, rel_tuples, rel_pages, rel_file_node, rel_access_method , n_distinct, distinct_values, selectivity, avg_width, correlation
DB Settings	bgwriter_delay, shared_buffers, bg-writer_lru_maxpages, wal_buffers, random_page_cost, bgwriter_lru_multiplier, checkpoint_completion_target, checkpoint_timeout, cpu_tuple_cost, max_stack_depth, deadlock_timeout, default_statistics_target, work_mem effective_cache_size, effective_io_concurrency, join_collapse_limit, from_collapse_limit, maintenance_work_mem

Output. We map them with the meta-information collected from the database. In Table 4, we show the meta-information attributes we use as input to the model used by the node. This information can be easily extracted from DBMS system tables e.g., PostgreSQL [28].

We also use a set of database configuration setting values of the running database as input features to the model. These configuration settings are selected based on their importance for performance tuning as described in [24, 35]. The approach of training our models with diverse configuration settings also sets us apart from other plan-metric prediction works [19, 29].

Altogether, we have three types of input features,

- *(a) Plan Features, f_{node}:* Features obtained from a plan operator nodes, see Table 1 for feature list.
- *(b) Meta Features, f_{meta}:* Meta-information about data and its distribution, see Table 4.
- *(c) DB Settings, f_{db}:* Handful number of database configuration settings, see Table 4.

With a triplet feature tuple as input $(f_{node}, f_{meta}, f_{db})$ our performance computation encoder aims to learn latent features while optimizing for Total Time, Total Cost, or Startup Time. In our joint training optimization approach as described in §3.2.3, we make use of these three metrics to capture better encoder features and avoid overfitting.

3.2.2 Model Architecture. We now present the deep neural network (DNN) architecture of the encoder with a pictorial representation in Figure 3. It is a three-column DNN on the top each for Plan features, Meta features, and DB features, respectively, with another fully connected NN layer merging the three parts and producing the embedding layer. The last fully-connected NN component takes the output of the embedding layer to predict the metric labels, i.e., Total Cost, Total Time, Startup Time. Also, each NN layer is followed by an activation function layer of ReLU (Rectified Linear Unit), Sigmoid or Tanh functions. As mentioned earlier, we create multiple instances of this supervised regression model, each for a functional operator.

A NN layer can efficiently represent or capture complex relations among input features by applying an affine transformation of the input. With multiple feed-forward NN layers, number of recursive affine transformations with weight matrices and non-linear activation functions are applied to the input features to produce an output. Then the difference between the desired output and the predicted output is calculated based on some metric functions dubbed as loss.

A gradient descent based technique is applied to tweak the weights on each layer used to perform the optimize affine transformation weights minimizing the loss. It allows the model to learn non-linear and polynomial order complex functions, automatically identifying the relevant features.

One of the ingenuity of this model is three-column multi-layered feature approach on Plan, Meta, and DB features, respectively, allows the model to find correlation among the same type of features first. Then transformed weighted features from each part can correlate effectively. As a preliminary attempt, we train an alternate model with a standard (single-column) DNN with all the input features together. In §6.2, we provide a comparative study to evaluate both the models.

3.2.3 Joint Training. A general rule of thumb for any model is that the distribution of predicted data remains the same as training data. But, in our case, the data distributions change with new workload. When the model learns from a single or small workload benchmark, the model overfits to the training set. With an assumption that if enough information on the data distribution is provided for training the model, the model may learn the factors governing the performance metrics for each operator (Scan, Join, Sort, Aggregate, etc.). Also, the fact that a general query plan optimizer (which is a logical component) uses the same statistical information we use as input to our model encourages us. The trick is to learn a generalized pretrained model that can adapt to an unseen workload with small data from the new domain. Hence, the pretrained models should utilize already learned parameters to adapt with the new workload.

We utilize a *joint training* approach for training the encoders. We train each operator model on multiple workloads on different data distributions and multiple database configuration settings. In joint training approach, we perform multiple metric tasks, each task optimizes for each label, i.e. Total Cost, Total Time and Startup Time. The difference in each of these models is the last NN-layer, which uses the embedding layer as input. Since the top level of the model remains unchanged, the weights are naturally tweaked to learn features based on multiple tasks.

We evaluate our performance encoder models on two criteria, (i) the model uses less data from a new domain to adapt, and (ii) the model error on validation and test data converges. We provide a detailed evaluation results on our pretrained computational performance encoder in §5.

3.3 Finetuning Evaluation

Given the above pretraining for learning structure and computational performance encoders, we hope that our learned model can be easily used in other unseen applications. We conduct two groups of finetuning evaluation for them:

Domain Adaptation. For both the structure encoder and computational encoder, they are trained from a source distribution on plan-pair similarity regression and performance attribution prediction tasks. *Domain Adaptation* aims at that these models can be easily finetuned on a different target data distribution. Hence, we finetuning them on different benchmark workloads on the same tasks, such as TPC-H and TPC-DS , and Spatial benchmarks. For plan-pair similarity regression task, we generate a collection of plan pairs for each new benchmark, and then calculating the *Smatch* scores for evaluation. For the performance attribute prediction task, we collect the new dataset by running workloads on different database configurations. More details about those datasets is introduced in §5.1, and the results on domain adaptation for each encoder in §6.

Transfer Learning to New Tasks Besides the ability of domain adaptation, we also define two new tasks to evaluate whether our pretrained plan encoder can be easily used for other tasks rather than our pretraining task in §4.

4 DOWNSTREAM TASKS

In this section, we show two downstream tasks that use our proposed plan structure and performance encoders. We present a bird-eye view model architecture, common to both the downstream tasks in Figure 4. For a given query plan input, meta information of database, and database configuration, the plan encoders (structure and performance encoder) produce respective representations as output. This output is then fed to the downstream task-specific model. Note that for generating the computational performance representation, we group plan nodes based on the type of functional operator and then pass it to the corresponding performance encoder to obtain representation.

The downstream task model is a standard multilayer-DNN taking three inputs, (a) structure embedding,(b) computational performance embedding, and (c) the database settings. The properties of database settings are real numbers. They can have an arbitrarily large value, which hinders learning a better model. We overcame the problem by scaling each database setting with a logarithmic function and using them as added features along with the actual numbers. Furthermore, we added a flexible design of reshaping the dimension of the structure or performance representation in the downstream task model for obtaining better accuracy.

4.1 Query Latency Prediction

The first downstream task is a real-world task of predicting query latency for an input query plan on a given database knob configuration settings utilizing our plan encoders. Formally, we define the query latency prediction problem as follows.

Problem 1 (Query Latency Prediction:). *Given a query plan p, meta-features f_{meta} of the database, and a database configuration settings f_{db}, the model predicts the latency of the query.*

For generating the training data for the latency model, we created an automated workload running scripts[3] that runs on cloud server instances and uploads executed plans along with the meta-features and database settings to our data repository. The script generates a new database configuration and configures the database automatically for each run. These new database configurations are generated based on the Latin Hypercube Sampling method [3, 20] for the properties mentioned in Table 4. This method for generating database settings has been used earlier [12, 35].

4.2 Query Classification

A smart database could use the knowledge of workload/ query distribution to set an optimal database configuration. An important step towards it is to learn the features of similar queries, and cluster/classify them. We conducted a query template prediction task

[3]https://github.com/debjyoti385/workload_scripts

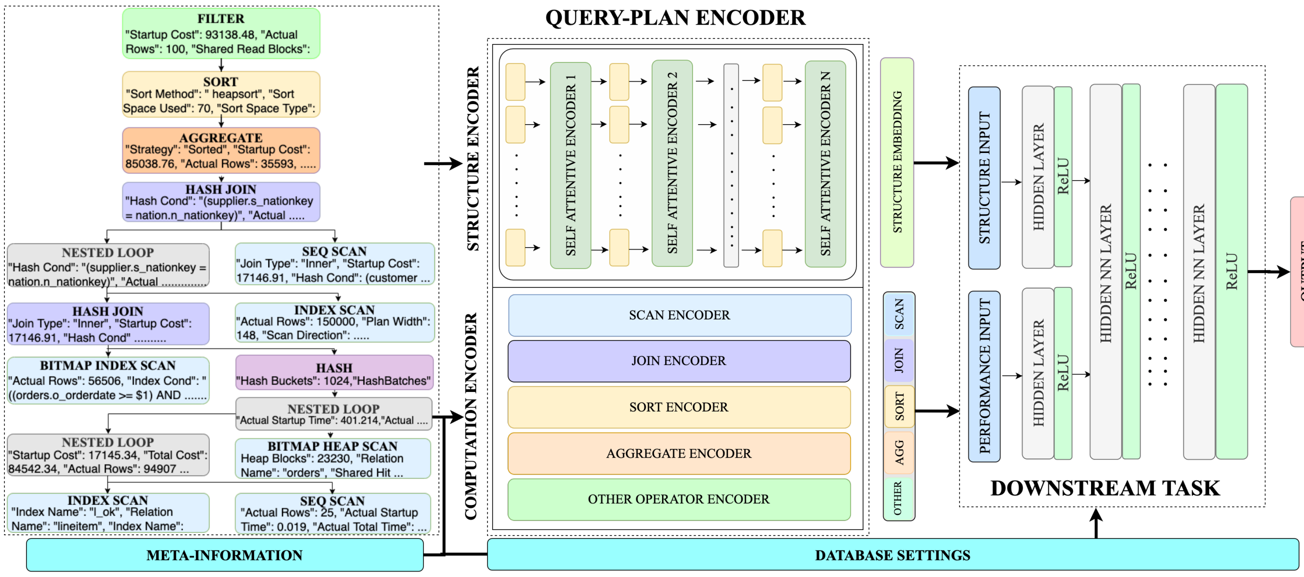

FILTER
"Startup Cost": 93138.48, "Actual Rows": 100, "Shared Read Blocks":
SORT
"Sort Method": " heapsort", "Sort Space Used": 70, "Sort Space Type":
AGGREGATE
"Strategy": "Sorted", "Startup Cost": 85038.76, "Actual Rows": 35593, ...
HASH JOIN
"Hash Cond": "(supplier.s_nationkey = nation.n_nationkey)", "Actual .....
NESTED LOOP
"Hash Cond": "(supplier.s_nationkey = nation.n_nationkey)", "Actual ....
SEQ SCAN
"Join Type": "Inner", "Startup Cost": 17146.91, "Hash Cond": (customer ...
HASH JOIN
"Join Type": "Inner", "Startup Cost": 17146.91, "Hash Cond" ....
INDEX SCAN
"Actual Rows": 150000, "Plan Width": 148, "Scan Direction": .....
BITMAP INDEX SCAN
"Actual Rows": 56506, "Index Cond": " ((orders.o_orderdate >= $1) AND ....
HASH
"Hash Buckets": 1024,"HashBatches"
NESTED LOOP
"Actual Startup Time": 401.214,"Actual ....
NESTED LOOP
"Startup Cost": 17145.34, "Total Cost": 84542.34, "Actual Rows": 94907 ...
BITMAP HEAP SCAN
Heap Blocks": 23230, "Relation Name": "orders", "Shared Hit ...
INDEX SCAN
"Index Name": "l_ok", "Relation Name": "lineitem", "Index Name":
SEQ SCAN
"Actual Rows": 25, "Actual Startup Time": 0.019, "Actual Total Time": ...
META-INFORMATION
QUERY-PLAN ENCODER
STRUCTURE ENCODER
SELF ATTENTIVE ENCODER 1
SELF ATTENTIVE ENCODER 2
SELF ATTENTIVE ENCODER N
COMPUTATION ENCODER
SCAN ENCODER
JOIN ENCODER
SORT ENCODER
AGGREGATE ENCODER
OTHER OPERATOR ENCODER
STRUCTURE EMBEDDING
SCAN
JOIN
SORT
AGG
OTHER
STRUCTURE INPUT
PERFORMANCE INPUT
HIDDEN LAYER
ReLU
HIDDEN NN LAYER
ReLU
HIDDEN NN LAYER
ReLU
OUTPUT
DOWNSTREAM TASK
DATABASE SETTINGS

Figure 4: A bird-view diagram, showing the role of plan encoders for a downstream task.

with our pretrained plan structure and performance encoders. We aim to show that our plan encoders can efficiently project query plans in latent dimensions finding similar query plans. We formally define the problem statement as follows.

Problem 2 (Query Classification:). *Given a query plan p, meta-features f_{meta} of the database, and a database configuration settings f_{db}, the model predicts the predefined class for the query plan based on feature similarities.*

We conducted an experiment with join order benchmark [16] containing 113 interesting query templates and 33 clusters of similar query templates. Due to the variable cardinality of the database tables and query predicates, the query plans generated from a query optimizer can differ from one another. It also makes the classification task challenging to cluster the query features accordingly. We include the performance encoder in classification tasks as queries even with similar plan structures can differ in performance features. We present detail of this experiment and the role of individual encoders in § 5.3.

5 EXPERIMENTS AND RESULTS

This section first describes the datasets we used in our experiments. We then present evaluation methods with experimental results for latency prediction and query template classification tasks.

5.1 Datasets

Crowdsourced Plan Dataset. We collected this dataset containing PostgreSQL queries along with its execution plans from a crowdsourced website[4][8]. We used this dataset for pretraining our structure encoder model. After pruning the plans with more than 200 nodes, we generate 57430/2871/2871 plan-pairs for training/dev/test and then calculate the Smatch score as their similarity score.

Industry Standard Benchmarks. We have used two industry-standard TPC-H [33] and TPC-DS [34] benchmarks as workloads with different scale factors (SF), and execute them with different database settings with an automated script[5]. We used a part of this dataset for pretraining our performance encoders. Table 5 shows statistics of the explored database settings from prepared datasets.

Spatial Benchmark. Spatial queries are notorious for hogging resources and need a proper database configuration for optimal performance. PostGIS, the spatial object extension for PostgreSQL, admits the configuration tuning requirement based on workload type in their documentation [24]. We use the two following spatial benchmarks in our experiments.

Jackpine: Jackpine [25] benchmark contains diverse spatial queries on spatial join with multi-polygons, lines, points and combination of them. We revised[6] the original benchmark with recently available shape datafiles, PostGIS extension, and made publicly available.

Open Street Map (OSM): The Open Street Map(OSM) workload has spatial overlap, distance, and routing queries. This dataset is created[7] with inspiration from work [4]. Due to sparsity, it is difficult to understand the underlying data distribution, which makes it an inviting benchmark for the experiment. We used the OSM map of New York and Los Angeles county.

Join Order Benchmark. It contains 113 different queries, which can be grouped into 33 clusters due to the similar SQL queries with different join orders. We run those queries on different database configurations and then collect the 16229 different plans. We split that into 13505, 1362, 1362 as training, dev, and test, respectively.

5.2 Results on Query Latency Prediction

We first evaluate our query latency prediction model with multiple experiments to project the overall effectiveness of using our plan encoders. We used pretrained structure and performance plan encoders trained on the Crowdsourced dataset and multiple TPC-H and TPC-DS workloads, respectively. A detailed analysis of our pretrained encoders is given in §6.1 and 6.2.

Ablation Studies. *(a) Spatial Benchmark:* We first present an ablation study on individual queries. The aim of this study is to measure

[4]https://explain.depesz.com
[5]https://github.com/debjyoti385/workload_scripts
[6]https://github.com/debjyoti385/jackpine
[7]https://github.com/debjyoti385/osm_benchmark

Table 5: Statistics on configuration settings generated for training data.

Database Setting	Unit	Median	95th Percentile	5th percentile
bgwriter_delay	ms	4,860.00	9,421.05	456.00
bgwriter_lru_maxpages	integer	515.00	958.05	55.00
checkpoint_timeout	ms	300.00	540.00	60.00
deadlock_timeout	ms	300,000.00	540,000.00	26,000.00
default_statistics_target	integer	4,827.50	9,563.00	454.85
effective_cache_size	bytes	1,048,576.00	1,966,080.00	131,072.00
effective_io_concurrency	integer	52.00	96.00	6.00
maintenance_work_mem	bytes	7,340,032.00	15,728,640.00	876,953.60
max_stack_depth	integer	3,072.00	5,120.00	417.95
random_page_cost	number	5,028.60	9,507.39	560.40
shared_buffers	bytes	2,097,152.00	3,932,160.00	131,072.00
wal_buffers	bytes	130,624.00	131,072.00	12,416.00
work_mem	bytes	15,728,640.00	31,457,280.00	1,048,576.00

the error relative to the variability of query latency. For initial training of the latency prediction model, we used plans from spatial benchmark [5, 22, 25] executed on 120 different database configurations. The trained model then predicts query latency for spatial queries on different database configurations. To prepare our test datasets, we ran each benchmark 50 times with very different database configuration settings.

Figure 5 shows the query latency statistics of query templates with median query latency greater than 500 milliseconds from spatial benchmark; Jackpine (with prefix Q), and OSM benchmark (with prefix OSM). The blue bars in the chart show the median of the query latency for all the query execution with different database settings. The orange line shows the *query latency variability* due to change of database settings. The bottom point of the orange line represents the 5th percentile, and the highest point marks the 95th percentile of query latencies. We present a complimentary Figure 6 along with Figure 5 that pictorially shows the mean absolute error for all the query templates from the spatial benchmark. The orange line is the measure of the time difference between 95th percentile and 5th percentile of a query latency in milliseconds, depicting the extent of the *query latency variability* for the particular query. To note, vertical axes on both figures i.e. Figure 5 and 6 are presented on a logarithmic scale with milliseconds as unit. It shows that at least 68% of the queries have MAE less than 10% of *variability*, and 90% of the queries have MAE less than 30% of *variability*.

Query latency prediction on the spatial benchmark is challenging because of the sparse geospatial data distribution from two areas contributing towards large variability. Furthermore, the performance of spatial queries is easily affected by database configurations. Significantly less mean absolute error from the latency prediction model shows that pretrained encoders helped the model.

(b) TPC-DS SF-100 Benchmark: In this experiment, we compare our latency model with state-of-the-art latency prediction models for each query template from the TPC-DS benchmark for a scale factor of 100 (i.e., 100 GB). A recent study by Marcus et al.[19] shows TPC-DS query ablation study with TAM [38], SVM [2], RBF [18] and QPP Net [19]. It is to note that we used the same TPC-DS plan dataset used by the study [19], and we split our dataset in 80:20 ratio for use as training and test data. In Figure 7, we show an ablation study of mean absolute errors of the predicted latencies for all the TPC-DS query templates for different models. We find that 25 (36%) query templates showing at least 10% better MAEs than the best baselines, 33 (48%) query templates within ±10% MAEs of best baseline, and only 11 (16%) query templates with MAEs greater

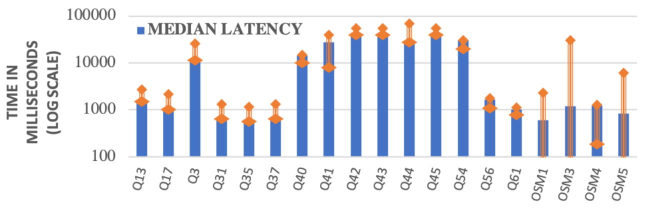


Figure 5: Statistics on latency of spatial queries (> 500 ms) from Jackpine [25] and OSM benchmark, where the blue bar represents median, the orange line represents the *query latency variability* with 5th and 95th percentile of query latency for different database configuration.

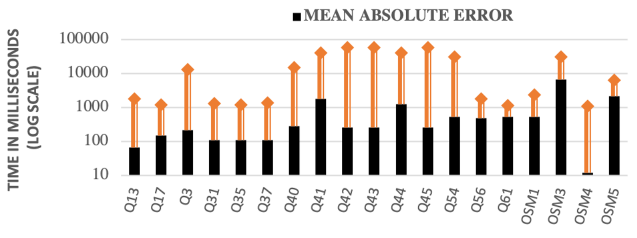


Figure 6: The black bar represents mean absolute error (MAE) (in ms) for spatial Jackpine and OSM queries, the orange line represents the *query latency variability* i.e. the measure of time difference between 95th percentile and 5th percentile (same as the orange line from Figure 5), a smaller black bar on a larger orange-line bar means better results.

than 10% of the baseline values. It is also worth mentioning that 22 (31%) and 12 (17%) of those queries templates show MAEs reduction of at least 25%, 50% over the best baseline.

We performed another analysis with the relative error factor R of the predicted latency from the ground truth for all the models, calculated as follows.

$$R(q) = \max\left(\frac{predicted(q)}{original(q)}, \frac{original(q)}{predicted(q)}\right)$$

We present the percentage of the queries with less than $1.5R$, between $1.5R$ and $2R$, and greater than $2R$ in Table 6 for TPC-DS dataset. Our result shows that our Plan Encoder has an edge over the QPP Net and other baselines. More than 91% of the queries are within $1.5R$ factor with Plan Encoder which is 2%, 6%, 23%, 40% better than QPP Net, RBF,SVM, and TAM respectively. The number of queries with more than $2R$ factor also reduced to 2%. Furthermore, we find that 74% of the queries are within $1.25R$ factor of the original latency. With a high percentage of the prediction within $1.5R$ and $1.25R$ factor, it can be said that Plan Encoder performed quite well.

As our encoders are first pretrained with general datasets, we expect them to perform well in general, which it did. But there is a small percentage of queries (from a few query templates) where the predictions are off by a considerable factor contributing to a higher mean value for errors on those query templates, shown in the ablation study with Figure 7. We investigated it and noticed that for some query templates database metadata (e.g. set of indexed/non-indexed columns) and configuration settings (e.g. shared buffers, working memory) largely contributes to latency output. Since, plan encoder takes database configuration as input as well (unlike baselines), we tried our best to match the configuration for those with baselines runs for TPC-DS dataset. Overall, it is still perceptible that our pretrained plan encoder approach works well in general on two spatial benchmarks (Jackpine, OSM) and TPC-DS.

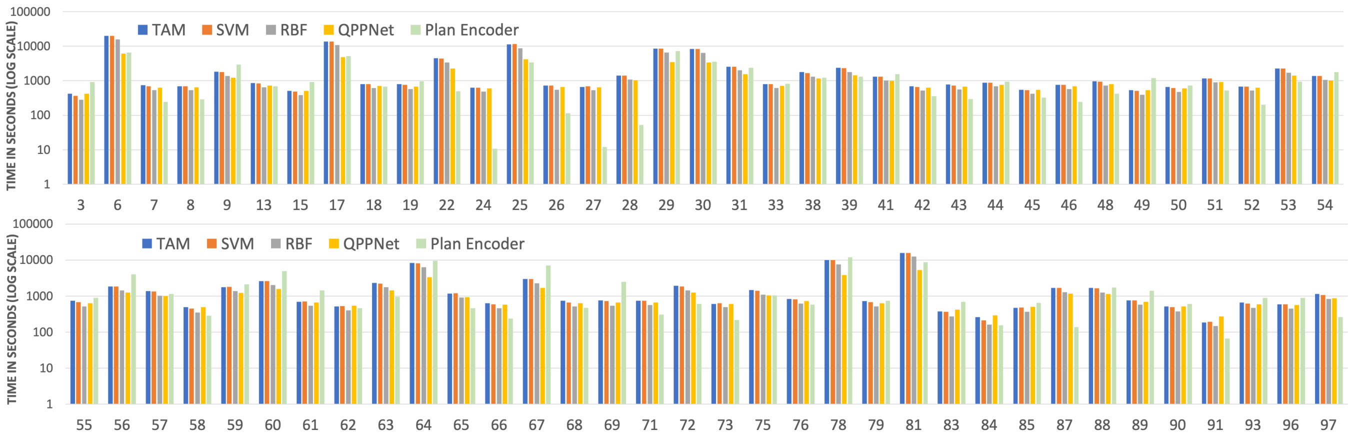


Figure 7: Ablation study of mean absolute error (MAE) (y-axis in logarithmic scale) for the all the TPC-DS query templates (x-axis) with scale factor 100.

Table 6: Queries from TPC-DS SF-100 test set binned based on R factor for all the models.

Model	$R \leq 1.5$	$1.5 < R \leq 2.0$	$R > 2.0$
TAM	51%	22%	27%
SVM	68%	15%	17%
RBF	85%	6%	9%
QPPNet	89%	**7%**	4%
Plan Encoder	**91%**	**7%**	**2%**

Discussion on Embedding Sizes; Structure Encoder vs. Performance Encoder: We performed another experiment finding the optimal embedding size for structure encoder w.r.t. the performance encoder. First, we found that using only structure encoder vs. only performance encoder yields 5 times the latency error of the latter. As a follow-up experiment, we kept performance encoder embedding fixed to 300 and varied structure encoder embedding sizes from 8 to 320. We used five TPC-DS SF-10 test datasets and found that the average MAEs dropped till embedding size 160 and then increased; we got the best MAEs with embedding sizes for structure vs. performance encoders as 160:300. It also confirms that features from the performance encoder dominate the structure encoders features, which is relatively low importance but still significantly impacts the latency prediction task.

5.3 Results on Query Classification

We conducted the query classification experiment with join-order benchmark, we fuse our pretrained structure and performance encoder to classify a plan with a template-id. The join order benchmark has 113 query templates and 33 clusters, and it is not trivial to classify queries from this dataset as join orders can change arbitrarily in plans. Our classifier aims to predict both the template id and cluster id. Our query classification model is similar to the latency prediction model but with a batch normalization layer and multi-classification cross-entropy loss. To understand how structure and performance encoder performs in the task, we performed an ablation study using structure-only, performance-only, and both in our experiments. The results in Table 7 show structure encoder plays main role in this task. Without it, the performance-only performs badly. Adding the performance encoder boosts f1-scores by 0.058 (29%) on template and 0.08 (21%) on cluster classifications.

Table 7: F1-scores of models for template and cluster query classification task on development and test set.

Models	Development		Test	
	template	cluster	template	cluster
Structure only	0.2452	0.4670	0.1946	0.3847
Performance only	0.1645	0.2973	0.0977	0.1769
Both encoders	**0.2783**	**0.5573**	**0.2518**	**0.4647**
Both encoders 10% data	0.2000	0.4927	0.151	0.334
Both encoders 30% data	0.2555	0.5228	0.1843	0.3855

We also found, when the models are finetuned on only 10% and 30% of data, i.e., rows with *Both 10% data* and *Both 30% data*, the models still performed reasonably well, which indicates that our pretrained encoder can boost learning for domain adaptation.

6 ANALYSIS

6.1 Structure Encoder

As described in §3.1, our structure encoder is pretrained on plan-pair similarity regression task with the self-attention encoder. We use a large amount of dataset from the Crowdsourced Plan dataset for pretraining. In this paper, we first prune those extremely large plans with more than 200 nodes. Then randomly select 63172 pairs of plans to form the dataset for our plan-pair regression task and calculating all the Smatch scores of those pairs.

Baseline Models For our plan-pair similarity regression (PPSR) task, we compare our Plan Encoder (Encoder-PPSR) with other self-supervised encoders such as Sparse Autoencoders (Sparse-AE), LSTM encoders (LSTM-PPSR) as baselines. All these baselines learn to represent input plans into a latent multidimensional space.

Results on Finetuning After completing the pretraining on Crowdsourced dataset for three models: Sparse AE, LSTM-PPSR, Encoder-PPSR. We investigate the domain adaptation capability of these models with finetuning. We randomly selected 11126, 55498, 60000 plan-pairs with plans from TPC-H, TPC-DS, and SPATIAL datasets, then created the training, dev, test splits with a ratio of $20 : 1 : 1$. We opted for three different strategies to test domain adaptation finetuning, *(a) Scratch-* Without pretraining, *(b) Fixed-* Keeping pretrained encoder in eval mode (fixed embedding) and train only the prediction layers, *(c) Fine-* Train both encoder and prediction layers together in finetuning procedure. Figure 8 show the *Smatch*

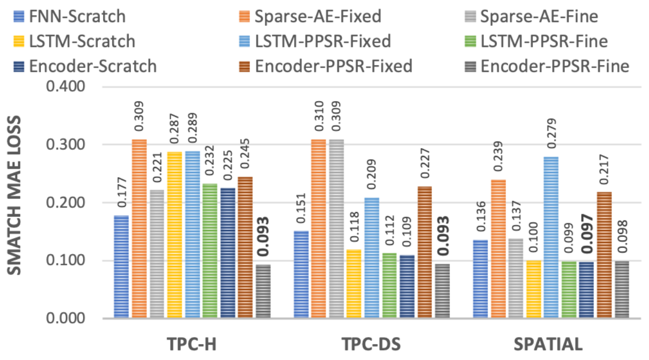


Figure 8: Main Results of finetuning structure encoder on TPC-H, TPC-DS, and SPATIAL

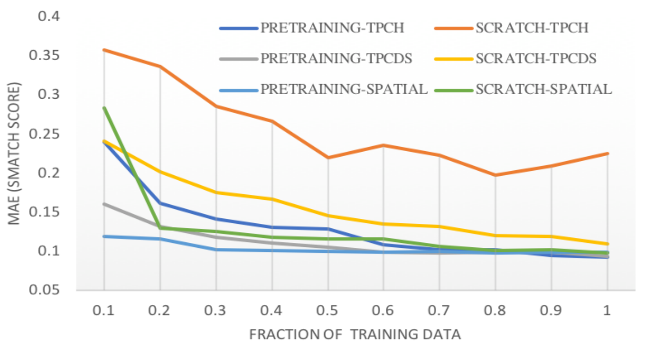
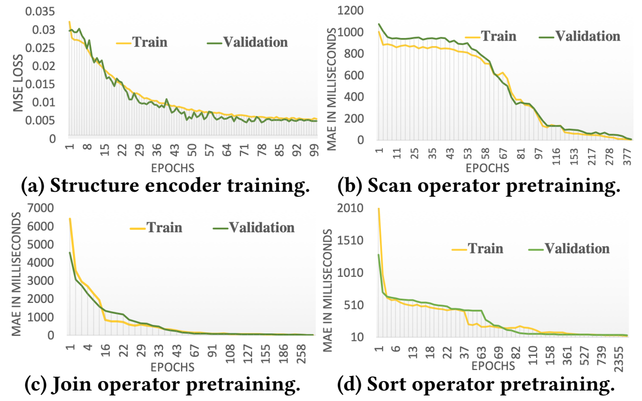


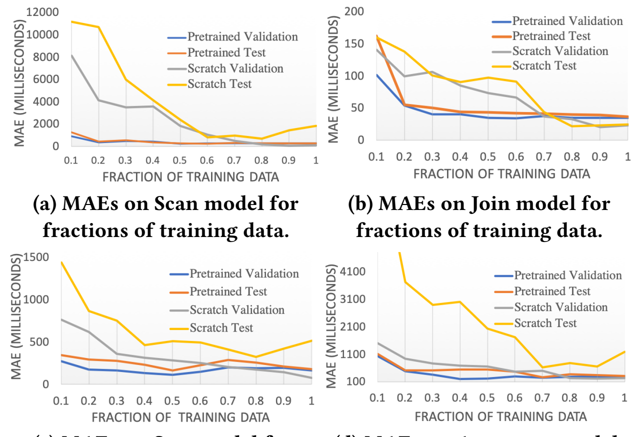
Figure 9: Plan-pair Regression: MAEs of Smatch score on fractions of training data



(a) Structure encoder training. (b) Scan operator pretraining.

(c) Join operator pretraining. (d) Sort operator pretraining.

Figure 10: Showing convergence of Mean absolute errors(MAE) (in seconds) for the validation, test and train datasets, while pretraining all the computational performance encoders.



(a) MAEs on Scan model for fractions of training data. (b) MAEs on Join model for fractions of training data.

(c) MAEs on Sort model for fractions of training data. (d) MAEs on Aggregate model for fractions of training data.

Figure 11: The effect of dataset size for finetuning with pretrained vs scratch(non-pretrained) models on TPC-DS SF-8, showing ≥ 0.3 fraction of dataset is enough for finetuning a pretrained model.

score's mean absolute error (MAE), the absolute difference between predicted and the actual Smatch score. The trend among *Scratch, Fixed,* and *Fine* strategies on all three domains shows similar MAE behavior. Note that both LSTM and self-attention scratch models performed at par on the spatial dataset, using pretraining did not improve the result by a lot in this case. Overall, in all the three domains, TPC-H, TPC-DS, and Spatial; Encoder-PPSR-Fine did well, which signifies that our self-attentive encoder can adapt better to a new domain.

In Figure 9, we compare pretraining and no-pretraining (scratch) method with different amount of training data. For all 3 benchmarks, especially TPC-H and TPC-DS, our pretrained method can achieve small MAE of Smatch score on less amount of data. On spatial data, our pretrained method only slightly better than no-pretraining one.

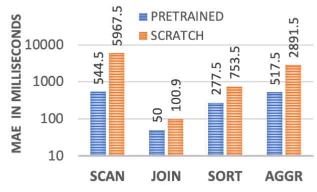
6.2 Computational Performance Encoder

We now perform local probe on computational performance encoder with a set of experiments evaluating the pretrained encoders for Scan, Join, Sort, and Aggregate operator. For pretraining, we used TPC-H and TPC-DS, both with scale factors 1,2,3 and 5 were executed on at least 20 different configuration settings randomly generated via Latin Hypercube Sampling method [3, 20].

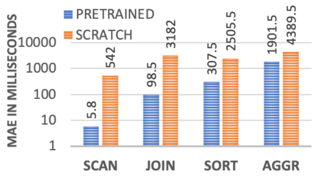
Pretraining: We first illustrate the training procedure and a few learnings from it. We split the dataset into 8:1:1 ratio for train, validation, and test for pretraining of all the four operators. Figure 10 shows the Mean Absolute Error (MAE) on latency (Actual Total Time) label for train, validation, and test data for scan, join, and sort operator. In all the cases along with aggregate (not shown in Figure 10), the train, validation, and test MAE converges below 1 second and stays around tens of milliseconds. The MAE on test data is calculated based on the epoch with the best validation model seen while training. We stop the training when the MAE on validation does not improve more than 5 milliseconds in the last 100 epochs. With a 12 GB GPU on a Ubuntu 18.04 operating system, each model takes around 6-8 hours to train.

Key insight on training the models is that the best MAE varies based on operators. The best MAE for the Scan model on test data is 12 milliseconds, where the validation MAE is 7 milliseconds. In the Join and Sort models, the test MAEs reach a low of 3.42 milliseconds and 44 milliseconds, respectively. It is to note that we performed pretraining on all the three labels Actual Total Time, Total Cost and Startup Time but for brevity we could only report the Actual Total Time metric in our figures.

Finetuning with pretrained models. The goal of having a pretrained model is to expedite the domain adaptability with less data. In many cases, obtaining adequate training data is challenging and time-consuming. In this set of experiments, we perform finetuning tasks on a new dataset of TPC-DS with scale factors 8 (SF-8). We also performed the same experiment on the spatial dataset, showing a similar result. Due to space constraints, we could not add the result of the spatial dataset.



(a) TPC-DS SF-8 benchmark.



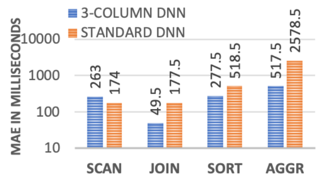
(b) Spatial benchmark.

Figure 12: Comparison of MAEs for pretrained vs scratch models with 0.3 fraction of finetuning data.

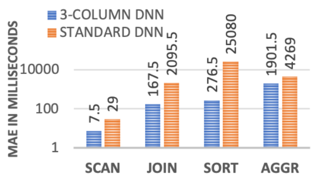
To show the effectiveness of pretraining models over scratch or non-pretrained model, we orchestrated a comparative experiment where the performance of models trained on fractions of training data. We limit the full training dataset to randomly chosen 2000 plans and test dataset to 500 plans for both TPC-DS and Spatial datasets. We run each model for 100 epochs which take around 10 minutes to train. In all the line charts from Figure 11, we notice that as the amount of training data increases, the MAE decreases on all the models, but the validation MAEs of scratch models is only comparable with the pretrained models when trained with 0.5 to 0.7 fractions of training data. The critical observation is that pretrained test seldom improves beyond 0.3 fractions of training data for our workloads.

To make the clear distinction between pretrained and scratch models, we show the MAE on the test dataset for each operator and dataset with 0.3 fractions of training data in Figure 12 for TPC-DS SF-8 and Spatial workloads. We report the test MAE for the best validation model obtained in 100 epochs. In all the cases, the pretrained model beats the scratch model by a considerable margin. Conclusively, it confirms that our pretrained encoders are beneficial and adapts to a new workload quickly.

Multi-column vs Standard DNN This experiment performs a comparative evaluation between our three-column DNN and a standard (single-column) DNN for the performance encoder. Similar to the previous finetuning experiment, we pretrained both models with the same workloads. After that, we finetuned each model with 0.3 fractions of training data from TPC-DS SF-8 and Spatial workloads independently to obtain multiple evaluation models. Figure 13a and 13b shows the Mean Absolute Error(MAE) obtained from the three-column DNN and the standard DNN models for an unseen TPC-DS SF-8 and Spatial benchmark dataset, respectively. With the TPC-DS workload, Figure 13a shows MAE for the three-column DNN model is better than standard DNN for all the operators except the *scan* operator. Whereas the MAE for three-column DNN is significantly less than standard DNN for the spatial workload. It suggests that keeping the performance features ($f_{node}, f_{meta}, f_{db}$) independent for the first few layers helps the model. Different features might get intertwined in the early stage in the standard single-column model, impeding its learnability.



(a) TPC-DS SF-8 benchmark.



(b) Spatial benchmark.

Figure 13: Comparison of MAEs for multi-column vs standard DNN models with 0.3 fraction of finetuning data.

In summary, our experiments present plan encoders' effectiveness in learning query plans characteristics through downstream tasks and domain adaptation probes. The results suggest the requirement of pretrained models to characterize unseen queries. Other database core systems certainly can leverage the plan encoders to increase their effectiveness and achieve *instance optimality.*

7 RELATED WORKS

There exists a few research work that uses data-driven analysis on query plans and its features to comprehend workload characteristic [2, 10, 13, 18, 19, 29]. Early research works [13, 18, 41], focuses on feature engineering with data mining techniques like k-NN[7] on high-dimensional features. The initial works show the importance of feature engineering, which encourages follow up research works using neural networks for workload related prediction tasks (metrics, resource demands, indexing, etc.) [10, 11, 17, 19].

All these methods learn models from input features of query plans for a specific task. In our paper, we show an approach to learn pretrained query plan encoders that can be used for many downstream tasks. Currently, database researchers are proposing prepackaged AI learned models for core components of databases [15, 29, 35]. Our work on query plan encoders bridges the gap between query input and prediction tasks.

Database tuning is an interesting problem to achieve instance optimality and closely relates to query performance prediction tasks. An earlier work, Ituned [32] uses a feature-based approach for tuning databases. Recently published work, QTune [17] uses query plans and reinforcement learning for tuning databases. In both approaches, query plans are essential. Our attempt to create a pretrained encoder for query plans is relevant to database tuning and other similar tasks. We show its relevancy with a latency prediction over a different configuration and different data. An earlier work by Popescu et al. [23] shows it is feasible to accomplish performance prediction tasks on new data distribution for the same query. One of the significant contributions of our pretrained encoders is the adaptability of the models with new data and queries.

8 CONCLUSION

In this work, we study a method of featurizing database workloads with AI based encoders that helps in understanding database queries under structural and performance properties. We followed a pretrained encoder based approach for our models that learns weights from diverse training dataset and then use the learned model in downstream tasks like query latency prediction. We performed multiple probes on structural encoder and performance plan encoders, to prove their learning capability and efficacy. We also present an in-depth ablation study on query latency prediction for multiple benchmark workload proving the usefulness of workload characterization with plan encoders. Our approach of studying database workloads with pretrained encoder models can pave a new direction in this field.

ACKNOWLEDGMENTS

We would also like to show our gratitude to Hubert Lubaczewski for providing us access to the crowdsourced plan dataset.

REFERENCES

[1] Mahtab Ahmed, Muhammad Rifayat Samee, and Robert E Mercer. 2019. Improving tree-LSTM with tree attention. In *2019 IEEE 13th International Conference on Semantic Computing (ICSC)*. IEEE, 247–254.

[2] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B Zdonik. 2012. Learning-based query performance modeling and prediction. In *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 390–401.

[3] P Audze and V Eglājs. 1977. New approach to the design of multifactor experiments. Problems of Dynamics and Strengths. 35. *Zinatne Publishing House* (1977), 104–107.

[4] BLP Baas. 2012. *Nosql spatial–neo4j versus postgis*. Master's thesis.

[5] Albert-Laszlo Barabasi. 2005. The origin of bursts and heavy tails in human dynamics. *Nature* 435, 7039 (2005), 207–211.

[6] Shu Cai and Kevin Knight. 2013. Smatch: an evaluation metric for semantic feature structures. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. 748–752.

[7] Thomas Cover and Peter Hart. 1967. Nearest neighbor pattern classification. *IEEE transactions on information theory* 13, 1 (1967), 21–27.

[8] Explain Depesz. 2019. PostgreSQL's explain analyze made readable. https://explain.depesz.com/ [Online; accessed 16-Dec-2019].

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[10] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R Narasayya. 2019. Ai meets ai: Leveraging query executions to improve index recommendations. In *Proceedings of the 2019 International Conference on Management of Data*. 1241–1258.

[11] Bailu Ding, Sudipto Das, Wentao Wu, Surajit Chaudhuri, and Vivek Narasayya. 2018. Plan stitch: Harnessing the best of many plans. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1123–1136.

[12] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning database configuration parameters with iTuned. *Proceedings of the VLDB Endowment* 2, 1 (2009), 1246–1257.

[13] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L Wiener, Armando Fox, Michael Jordan, and David Patterson. 2009. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *2009 IEEE 25th International Conference on Data Engineering*. IEEE, 592–603.

[14] Geoffrey E Hinton and Ruslan R Salakhutdinov. 2006. Reducing the dimensionality of data with neural networks. *science* 313, 5786 (2006), 504–507.

[15] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. Sagedb: A learned database system. (2019).

[16] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.

[17] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2118–2130.

[18] Jiexing Li, Arnd Christian König, Vivek Narasayya, and Surajit Chaudhuri. 2012. Robust estimation of resource consumption for sql queries using statistical techniques. *Proceedings of the VLDB Endowment* 5, 11 (2012), 1555–1566.

[19] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-structured deep neural network models for query performance prediction. *arXiv preprint arXiv:1902.00132* (2019).

[20] Michael D McKay, Richard J Beckman, and William J Conover. 2000. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics* 42, 1 (2000), 55–61.

[21] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.

[22] OpenStreetMap contributors. 2017. Planet dump retrieved from https://planet.osm.org . https://www.openstreetmap.org.

[23] Adrian Daniel Popescu, Vuk Ercegovac, Andrey Balmin, Miguel Branco, and Anastasia Ailamaki. 2012. Same queries, different data: Can we predict runtime performance?. In *2012 IEEE 28th International Conference on Data Engineering Workshops*. IEEE, 275–280.

[24] PostGIS. 2019. Spatial and Geographic Objects for PostgreSQL. https://postgis.net/workshops/postgis-intro/tuning.html [Online; accessed 16-Dec-2019].

[25] Suprio Ray, Bogdan Simion, and Angela Demke Brown. 2011. Jackpine: A benchmark to evaluate spatial database performance. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 1139–1150.

[26] Yusuke Shido, Yasuaki Kobayashi, Akihiro Yamamoto, Atsushi Miyamoto, and Tadayuki Matsumura. 2019. Automatic source code summarization with extended tree-lstm. In *2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.

[27] Vighnesh Shiv and Chris Quirk. 2019. Novel positional encodings to enable tree-based transformers. *Advances in Neural Information Processing Systems* 32 (2019), 12081–12091.

[28] Michael Stonebraker, Lawrence A Rowe, and Michael Hirohama. 1990. The implementation of POSTGRES. *IEEE transactions on knowledge and data engineering* 2, 1 (1990), 125–142.

[29] Ji Sun and Guoliang Li. 2019. An end-to-end learning-based cost estimator. *arXiv preprint arXiv:1906.02560* (2019).

[30] Kai Sheng Tai, Richard Socher, and Christopher D Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075* (2015).

[31] Jian Tan, Tieying Zhang, Feifei Li, Jie Chen, Qixing Zheng, Ping Zhang, Honglin Qiao, Yue Shi, Wei Cao, and Rui Zhang. 2019. iBTune: individualized buffer tuning for large-scale cloud databases. *Proceedings of the VLDB Endowment* 12, 10 (2019), 1221–1234.

[32] Vamsidhar Thummala and Shivnath Babu. 2010. iTuned: a tool for configuring and visualizing database parameters. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 1231–1234.

[33] TPC Benchmark H: Standard Specification v2.17.3. [n.d.]. TPC Benchmark H: Standard Specification v2.17.3. http://www.tpc.org/tpch/default.asp

[34] TPC Benchmark DS: Standard Specification v2.6.0. [n.d.]. TPC Benchmark DS: Standard Specification v2.6.0. http://www.tpc.org/tpcds/

[35] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 1009–1024.

[36] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[37] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. 2008. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*. ACM, 1096–1103.

[38] Wentao Wu, Yun Chi, Shenghuo Zhu, Junichi Tatemura, Hakan Hacigümüs, and Jeffrey F Naughton. 2013. Predicting query execution time: Are optimizer cost models really unusable?. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 1081–1092.

[39] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V Le. 2019. XLNet: Generalized Autoregressive Pretraining for Language Understanding. *arXiv preprint arXiv:1906.08237* (2019).

[40] Laura A Zager and George C Verghese. 2008. Graph similarity scoring and matching. *Applied mathematics letters* 21, 1 (2008), 86–94.

[41] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. 2017. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 338–350.