QPSeeker: An Efficient Neural Planner combining both data and queries through Variational Inference*

Christos Tsapelas
Athena Research Center
Athens, Greece
ctsapelas@athenarc.gr

Georgia Koutrika
Athena Research Center
Athens, Greece
georgia@athenarc.gr

ABSTRACT

Recently, deep learning methods have been applied on many aspects of the query optimization process, such as cardinality estimation and query execution time prediction, but very few tackle multiple aspects of the optimizer at the same time or combine both the underlying data and a query workload. QPSeeker takes a step towards a neural database planner, encapsulating the information of the data and the given workload to learn the distributions of cardinalities, costs and execution times of the query plan space. At inference, when a query is submitted to the database, QPSeeker uses its learned cost model and traverses the query plan space using Monte Carlo Tree Search to provide an execution plan for the query.

1 INTRODUCTION

Cost-based query optimization is the process where a database system determines the optimal execution plan for a query. Cardinality estimation, join order selection and computational cost estimation of a (sub)plan highly affect the decisions of the planner during the construction of the execution plan. Even though most databases use hand-crafted heuristics, which encompass many years of research, they do not scale well to modern analytical workloads. Towards this direction, recent efforts have turned their attention to deep neural networks and aim at substituting traditional components of the planner with neural approximators [14, 20, 26]. Despite promising results so far, we observe three aspects of the optimization process that most state-of-the-art methods do not tackle in their entirety:

– *Optimization based either only on data or queries.* Most approaches address query optimization from a workload-driven point of view [9, 20, 39]. Few efforts focus on data distribution approximation, mostly for the cardinality/selectivity estimation problem [5, 26], taking into account only the underlying data. We observe that a traditional query optimizer calculates and internally stores statistics regarding the underlying data, which are used for the cost estimation of a plan operator, while it uses the information provided from the workload for query caching or optimization of similar queries, in other words it leverages information from both data and the queries. We believe a ML-based optimizer should do the same too.

– *Optimization of a single task during query planning.* When a query is posed to the database system, a traditional query optimizer must estimate the selectivities of the query filters, the cardinalities of join operators, and form an optimal join ordering for the query. Very few ML-based methods tackle the aforementioned set of tasks at once [6], while most focus either only on cardinality estimation [29], query latency estimation [39] or join order selection [37] by trying either to approximate the data distributions or come up with rich query representations. All these tasks are mutually dependent, hence optimizing only for one makes the process inefficient.

– *Training on only one plan of the query plan space per query provided by the DB optimizer.* The proposed methods tackling the join ordering problem or taking into consideration the execution plan of the query for a particular task, rely heavily on the execution plan provided by the database optimizer and do not traverse the query plan space at all. As a result, the biases being present in the optimizer's logic form a biased dataset and get transferred into the model's weights [4]. A neural network has the tendency to amplify the biases present in the training set. In query optimization, and especially in production environments, this side effect can have catastrophic results.

Motivated from the above limitations, we propose *QPSeeker* [1] (Query Plan Seeker), a novel *end-to-end neural database planner* that (*a*) simultaneously learns to perform all basic tasks of a traditional optimizer, such as join order selection, cardinality/selectivity estimation and execution time prediction, (*b*) leverages queries and data for training and inference, (*c*) samples the query space of each training query to generate an enriched training set, and (*d*) uses its rich learned model for query planning.

Overview. In QPSeeker's core is a model that learns to *approximate the distributions of the cardinality, computational cost and runtime* of the plans in the workload. Our approach assumes that queries with *similar characteristics* (e.g., number of tables, number of joins, filters applied, etc.) and *complexity* in terms of execution time will be close to each other in a latent space, and we *use variational inference* to learn this space. Hence, at the heart of our system lies a Variational Autoencoder (VAE) [13], whose latent space is enforced to follow a Gaussian structure, where each latent dimension represents a latent feature of the data. At the end of training, similar queries and, particularly, similar query execution plans will fall close to each other in the learned latent space.

To jointly learn from both data and queries, information about data and data distributions is used for training, along with features extracted from the query. We address several challenges. One challenge is to capture the data distributions from the table data and provide a rich representation for further processing. Moreover, the query/plan representations inside the system play an important role in order to give the model the ability to capture the correlations between the query, its complexity, and the data. Furthermore, one important question is how we associate the query and the execution plan we are investigating. For this purpose, our approach comprises the following novel components.

First, for the representation of table data in the model, we choose TaBERT [36], a language model for tabular data. Trained on millions of tables from the WDC corpus [17], it learns much

*Produces the permission block, and copyright information

[1] https://github.com/athenarc/QPSeeker

richer data representations than creating database table embeddings from scratch. Moreover, the pretraining tasks applied to TaBERT, i.e., the datatype and cell value prediction, help TaBERT to learn information about table data types and distributions.

Second, QPSeeker extracts the sets of relations, joins and predicates from the query (as in [14]), and subsequently *learns the mapping between these three sets and the query plan statistics.* In this way, it is able to capture the distributions of various instances of the above sets (in terms of which elements are present) paired with the particular physical operators in the query plan.

Third, QPSeeker employs a *rich, tree-like, query plan representation* that: (*a*) captures *data distributions using TaBERT*; (*b*) encodes each node in the plan using *information about this node* (including the relations involved, the physical operation, and the contextual representation of the table data) *as well as the impact of the previous operations* in the subplan; and (*c*) computes an embedding vector that contains the prediction of the values for the cardinality, cost, and runtime for each node, as well as a data vector that captures the impact of the node's children.

Fourth, we observe that each plan node does not have the same impact on the final runtime of the query and its computational cost differs from the cost of the other nodes in the plan. For example, an early decision of the planner for an Index Scan over a table, which may seem promising at early stages, may lead to bad paths (join orderings). Therefore, QPSeeker associates a query and a plan by *considering the impact of each plan node on the query estimations through a cross-attention mechanism.* In particular, we apply attention between the query embedding vector and the embedding vector of each node in the plan, to score which nodes have the most impact on the final estimations.

For training, we generate sample plans for each query instead of relying on a single, 'best', plan provided by the DB optimizer. In this way, we "mimick" a traditional optimizer that traverses the plan space for a given query and estimates the query execution cost and runtime along with the cardinalities of the intermediate results. For each query, we create samples from the plan space, by taking different join orderings and different methods for the operators of the query. The rationale for sampling the plan space is that the DB optimizer relies on internal statistics and formulas to make its estimations and come up with the best plan. If we just rely on this plan to train our optimizer, we will not be able to acquire such broad knowledge. Furthermore, we can either use the internal cost model of the DBMS or a user-defined one to generate the training data. Using sampling coupled with variational inference allows us to train our model not to directly learn a mapping of the workload to the target values, but to approximate the distributions of the cardinality, computational cost and runtime of the execution plans per query in the workload.

At inference, we use Monte Carlo Tree Search [15] along with our learned cost model, which combines information from both the data and the query, to traverse the query plan space.

Contributions. The contributions of this paper are:

- We introduce QPSeeker, a novel neural planner that simultaneously learns to perform all basic tasks of a traditional optimizer, such as join ordering, cardinality/selectivity estimation and execution time prediction.
- We cast our learning problem to a variational inference problem. We train our VAE-based model to approximate the distributions of the cardinality, computational cost and runtime of the execution plans. Our model captures hidden commonalities between the queries and the data into a latent space.
- We leverage both data and queries. We employ a rich query plan representation that captures the correlations between the query, its complexity, and the data. For the representation of table data, we choose TaBERT that captures the data distributions and provides a rich data representation.
- We calculate the impact each plan node has on the query's estimations through an attention mechanism.
- For training, we generate sample plans from the query space of each training query to generate an enriched training set. In this way, we train our model not to directly learn a mapping of a workload to the target values, but to approximate the distributions of the cardinality, computational cost and runtime of the execution plans per query in the workload.
- At inference, we use the learnt model and Monte Carlo Tree Search for query planning.
- We present detailed experimental results. Our experiments show that QPSeeker achieves being an all-in-one planner that performs all tasks of a query optimizer in an effective way outperforming competitors. Especially, for complex queries, it outperforms PostgreSQL. Furthermore, it learns better using complex workloads, and it shows excellent adaptability to different workloads, where competitors cannot cope.

2 RELATED WORK

In the last years, there has been significant efforts into the integration of machine learning models into query optimizers.

Learning Cardinality Estimation. For the regression problems of cardinality and selectivity estimation, many (un)supervised methods have been proposed. MSCN [14] is a supervised method that uses set extraction of the basic elements (relations, joins and predicates) from each query. Following the same rationale, a new heuristic metric called *Plan-Error* is proposed for cost-guided cardinality estimation [27]. *These approaches neglect the presence of the underlying data and their effect on query performance.*

There are approaches that try to capture the underlying data. Flow-Loss [26] defines a metric, where the query plan is formulated as an electric circuit, and the model estimates the cheapest path. DQM [5] faces the cardinality estimation task as both (un-)supervised problem, by estimating distribution densities. Naru [35] and its predecessor, UAE [31], use autoregressive models to approximate joint distributions over the database tables. NeuroCard [34] estimates the cardinalities over extracted samples from full outer joins of the database tables. DeepDB [8] introduced relational sum product networks, which are tree-structured to capture the data distributions using several local PDFs, and as we go up the tree, each node stores cumulative PDFs. *These approaches can approximate quite accurately the table distributions for a small number of joins, but they do not scale well for many joins.* FLAT [41] uses another type of network, called factorize-sum-split-product network (FSPN), to capture the underlying data calculating the level of dependency of column via conditional factorization. Finally, Fauce [19] introduced a model incorporating uncertainty in its predictions.

Learning Join Orders. Another line of research has used reinforcement learning (RL) to find plans with low cost (e.g., [16], ReJOIN [22]). Neo [21] proposes a learnable query optimizer that incrementally searches and builds the physical query plan. Despite the significant training time, the produced query plans were very competitive compared to the plans of a commercial optimizer. Neo also introduced the formulation of the query plan as a Tree-Convolution, also used in Bao [20]. Bao uses RL to learn hints at the plan operator level to advise the PostgreSQL query optimizer. Bao's approach was adopted to shrink the very large search space of the SCOPE optimizer [2] to make it work in a cloud environment [25]. RTOS [38] introduced a Tree-LSTM structure used with Q-Learning to tackle the join order selection task, but also needed large number of episodes to achieve comparable results. Balsa [33] used RL to produce query plans by applying query plans to PostgreSQL and evaluating its choices by trial and error. Toulouse [12] applies RL to learn join orders from an expert cost-based optimizer and transfers this knowledge into rule-based optimizers.

Learning Cost Estimation. Plan cost estimation is a critical task of query optimization. E2E-Cost [29] and QPPNet [23] featurize the physical query plan as a tree and propose to train a regression model to predict the cost of a physical plan. E2E-Cost mimics the tree-structure of the query plan into a Tree-LSTM model. Moreover, it introduced a new approach to create a neural representation keeping the logical semantics of a predicate as well as a new method to create embeddings for string values. QPPNet also follows the tree structure of the plan, associating each plan operator to a small MLP. It proposed a plan-structured deep neural network, i.e., a neural network model specifically designed to predict the latency of query execution plans by dynamically assembling neural units in a network isomorphic to a given query plan. Zero-Shot [6] aims at generalizing learned cost estimation to unseen databases. In contrast to workload-driven approaches, zero-shot cost models suggests a new learning paradigm based on pre-trained cost models.

Comparison. QPSeeker is an end-to-end neural database planner that *performs all basic tasks of an optimizer*, i.e., join order selection, cardinality/selectivity estimation, cost and execution time prediction (while most works focus on a single task). Furthermore, existing approaches are haunted by complex designs and significant training times. For example, Neo [21] reported 24h for training, while QPPNet [23] used a network of 8 layers, each additional hidden layer adding on the order of 2^{14} additional weights, that did not converge until epoch 1000 (28 hours). By leveraging Variarional Inference [1] boosted by a language model (TaBERT), QPSeeker is *considerably leaner, just 10.8M parameters in total*, and *can be trained in a short time*, (less than 1h) as we will see in the experiments, making it the first viable solution that brings deep learning inside the query optimizer.

Furthermore, most approaches do not leverage both queries and data. E2ECost [29] includes a small sample from each table in the encoding similar to [14]. QPSeeker *combines information from the queries and the data*. It employs a *rich query plan representation* approach that (a) *captures data distributions using TaBERT* [36], a language model suited for tabular data, and (b) uses *attention to weigh in the impact of each query plan node* on the query estimations. Note that recent works have focused on query plan representations. For example, QueryFormer [40] proposes a different scheme based on Transformers that integrates histograms obtained from database systems into query plan encoding.

3 THE PROPOSED FRAMEWORK

3.1 Problem Statement

Given a query, the goal of the planner is to come up with a good execution plan. An execution plan is represented as a tree whose internal nodes are operators and its leaves correspond to the input tables of the query. During query planning, the accurate

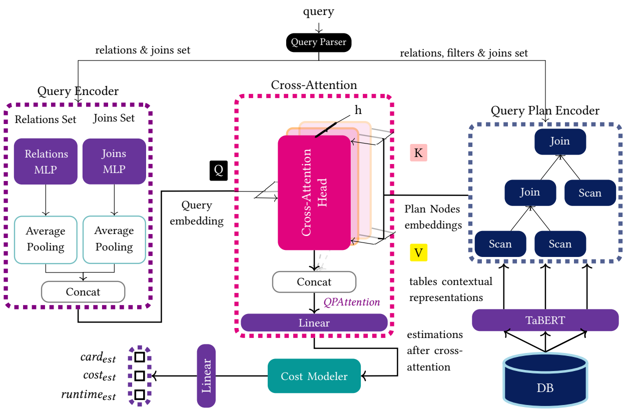

Figure 1: QPSeeker's architecture. The relations and joins sets are encoded from the Query Encoder resulting to the query embedding (left). TaBERT provides the encodings for base relations and the whole plan is encoded by the Plan Encoder (right). QPAttention scores the impact of each node in the plan and encodes the input QEP. The Cost Modeler (VAE) makes the estimates of target values for each QEP (bottom).

estimations of cardinalities, costs, and runtimes of the execution plan nodes give the ability to the optimizer to build good plans.

We start with a workload W of queries. For each query in W *a)* we extract the execution plan produced from the optimizer, or *b)* we extract a sample of execution plans per query (more details in Section 5.1). In the former case, our training set consists of a one-to-one mapping between the queries and the execution plans, while in the latter a one-to-many. Each unique pair of query and execution plan is called *QEP*. Each *QEP* is characterized by its cardinality, computational cost, and runtime.

We aim to construct a model, which associates the table data with the physical operations in the plan to predict the resulting cardinalities, computational costs and runtimes for W, and further to approximate the data distributions and the cardinality, computational cost and runtime distributions from the available workload, and make predictions for unseen queries. During inference, when a query is posed to the database system, QPSeeker uses the trained model and traverses the query plan space to evaluate candidate plans and suggest the one to be executed.

3.2 QPSeeker Pipeline

We provide an overview of QPSeeker's pipeline (Figure 1). From each query, we extract three sets: (*a*) the set T_{q} of query relations, (*b*) the set J_{q} of joins, and (*c*) the set P_{q} of conditions over the database relations. The relations and joins are one-hot encoded based on the database schema and these encodings are passed to the *Query Encoder* to build the query embedding vector.

From the execution plan of each *QEP*, the *Plan Encoder* encodes each physical operator and computes the values of the plan in a bottom-up fashion, based on features present in the query. Each plan node takes as input (*a*) the sum of one-hot encodings of the relations being present at each level of the subplan, (*b*) the physical operation applied also in one-hot encoding, and (*c*) the contextual representation of the table data extracted from TaBERT. The output of each node is an embedding vector, where the last dimensions are the estimations of the cardinality, cost and latency of the plan at this level. The root node holds these values for the entire plan.

Next, we combine the outputs of the *Query Encoder* and *Plan Encoder*, i.e., the query and the plan embedding vectors, using Attention (*QPAttention*) to score which nodes of the plan affect the most the given query, followed by a dense layer, with output size equal to the sum of the query and plan embedding vectors. The above lead to the encoding of each *QEP* to apply the Variational Inference model.

The final component of QPSeeker is the *Cost Modeler*, responsible to capture the distributions of the available *QEPs* in our training set. The goal of the *Cost Modeler* is to approximate the posterior distributions of the target values, which is accomplished through a variational inference model. The purpose of this model is to introduce a set of latent Gaussian variables to approximate the desired distributions of *QEPs* in the training set. More details in Section 4.4. Finally, the reconstructed vector is passed to a linear layer to get the estimates of QPSeeker.

4 QEPS ENCODING

4.1 Query Encoding

The query encoder is responsible for providing a rich representation of the query. Its output will be used to compute the association between the query and the execution plan.

We follow the feature extraction process described in MSCN [14]. Each relation in T_q is mapped to a one-hot vector of size N, where N is the number of database relations. Similarly, each join in J_q is transformed into a one-hot vector with length M equal to the number of all possible joins in the database. Next, we transform each set of vectors into a fixed-size input for further processing. We map T_q and J_q to two fixed size arrays, NxN and MxM, respectively. The NxN (MxM resp.) array contains all the one-hot vectors for T_q (J_q, resp.) at the first rows and the rest are all zeros.

We feed each matrix, along with a column vector that serves as mask to filter the non-zero rows, into a feed forward network with five hidden layers (i.e., a Multi-layer Perceptron, MLP). Finally, the encoder applies mean pooling among the elements of each set to derive one representation for each set and concatenates the two representations to form the query embedding vector. We use this encoding method, which is based on sets, because we wish to be able to approximate the distributions of the queries containing a particular combination of relations and joins, rather than learning a query specific encoding. This approach, along with the *Cost Modeler's* functionality, allows QPSeeker to be able to associate and group alternative ways to execute a query, as shown in Figure 5 (and will be discussed in Section 4.4).

4.2 Query Plan Encoding

The plan encoder aims at capturing the result of the interactions of the physical operators over the tables learning from the operators and the structure of each query plan. In this way, the model can learn, for example, the cost of applying a Hash Join over two tables, where the outer table is accessed via an index.

Generally, the performance of each plan operator is highly correlated with that of its children in the execution plan. During the flow of computation performed inside the plan encoder, we wish to capture this interaction between the nodes at the operator level. Hence, at each node, we compute an embedding vector that contains the prediction of the values for the cardinality, cost, and runtime for this node, as well as the interaction between the nodes. To capture the correlations between the nodes, we need somehow to inform the parent about the output of its children.

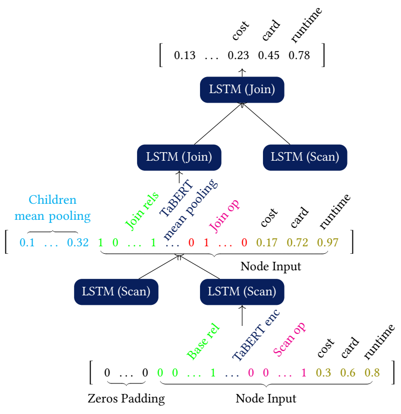

Figure 2: Plan Tree Encoding.

Hence, apart from the current state of the subplan, which will be discussed shortly in node input, each node in the plan encoder passes its output to its parent.

Figure 2 depicts our plan tree encoding. *Plan Encoder* assembles the plan operators in a tree, having the same tree structure as the execution plan provided by the optimizer.

Node Encoding. A query plan consists of two types of nodes: (*a*) the leaf nodes that correspond to the scan operations over the base tables of the database, and (*b*) the intermediate nodes that correspond to the join operations. In our configuration, each plan node is modeled as an LSTM cell [10]. Similar to the query plans produced by a database system, where each node in a plan is affected only by its children, the input of each LSTM cell can come only from its children. Additionally, the architecture of the LSTM cell suits very well the query plan encoding process, as it can capture information over long sequences (its inputs) and hence it can decide which information from its ancestors is useful and which not.

Node Output. Each node of the plan outputs a vector (of size 1500) that contains useful knowledge about the interactions of the operators in the query plan. An example vector is seen at the output of the root node in Figure 2. The last three dimensions of this vector are the estimations of the cardinality, cost, latency of the node in the plan. The remaining dimensions comprise a data vector that captures the interactions between the nodes of the (sub-)plan under this node. As estimates for the whole query plan, we consider the output of the root node of the plan.

Node Input. The input of a node is a fixed size (2048) vector that combines several types of information. On the right side of Figure 2, we see the input of a leaf node, and on the left side, we see the input of an intermediate node. They share a similar structure but they also have some differences as we explain below.

The input of a leaf is the concatenation of the following vectors (looking at the example vector from right to left):

a. Estimations for the cardinality, cost and runtime for the operation of this node. For a given query plan, we use EXPLAIN to get this information from the DB optimizer.
b. The physical operator applied to this node one-hot encoded.
c. The representation of the data processed. If there is a filter in the set P_q of predicates over a column of the table, we take the representation of this column filtered based on this predicate, otherwise the table representation. In both cases, the representation is provided by TaBERT.

d. The table accessed in this leaf node in one-hot encoding.

e. Zeros for padding. The input of each node consists of two parts. One part comes from its children nodes and one concerns the operation of the node per se. Since leaf nodes do not have children, they only encode information regarding the node and the first part is padded with zeros to tell the plan encoder that there are no children for this operator of the plan, hence there is no information from a predecessor node to affect the node.

The input of a non-leaf node is the concatenation of the following vectors:

a. Estimations of the cardinality, cost and runtime for the operation of the node computed by mean pooling the last three dimensions of the output vectors of the node's children.

b. The physical operator applied to the node one-hot encoded.

c. The representation of the data processed, which comes from the result of mean pooling over the output from the [CLS] token of each joined relation. This token has a special functionality as it holds information over the entire table. More details about the [CLS] token are provided below.

d. The relation encoding is the sum of one-hot vectors of all relations joined up to this level of the plan. Providing this encoding to the LSTM cell, we inform the plan which relationships are present in the subplan and which are not.

e. The information about the interaction between the children and parent. Instead of zeros in leaf nodes indicating the absence of an ancestor, we desire that features from children nodes are passed up the tree. Hence, we provide the result of mean pooling from the data vectors of the node's children.

TaBERT - Table Data Representation. While the query and the plan representations are crucial, the representation of the table data and their distributions are also very important. One approach would be to create embedding vectors from scratch for each database like Neo [21] and TLSTM [29], but such a strategy has limitations if the table data changes, because the model has to be retrained again. To override the above restrictions, we reap the benefits of transfer learning properties found in large pretrained language models. TaBERT [36] is a special case of BERT [3] for tabular data, and provides much richer and robust tabular data representation, unbounded from the strict assumptions regarding their datatypes and their prior distributions as in a RDBMS.

We use TaBERT as follows: for each *QEP* in the workload, it tokenizes the columns of the table, and for each column, it creates (name, datatype, value) triplets separated by a special symbol. Each value is extracted from the top-K rows of the table with the biggest n-gram overlap with the query. Then, these triplets are concatenated with the query and served as input to a BERT model. After the initial BERT encoding, TaBERT needs to gather the information from all row-level encodings into one vector containing an output for each column. Consequently, it calculates cell-wise attention over all rows, called vertical attention. Each cell contains the output of TaBERT for each column in the table. Finally, as in language modeling, where the [CLS] special token at the start of each sentence holds information about the whole sentence, similarly this token in the output of TaBERT holds information for the whole table.

TaBERT is trained on *Masked Column Prediction (MCP)* and *Cell Value Recovery (CVR)* objectives. The former encourages the model to recover the name and the datatype of the masked column from its contexts, hence learning, in this way, the correlation between the masked column and the other columns in the table. With this task, we can pass to our plan encoder information about the datatype of a column. The latter objective encourages the model to predict the values of the masked columns. More precisely, after column masking and the extraction of top-K rows from the table, TaBERT is tasked to predict the values of the masked cells. In this way, TaBERT captures information about the column distribution, along with its context within the rows. The developers provide three different models for $K = [1, 2, 3]$. With the use of TaBERT, this information is also inferred in QPSeeker's plan encoder.

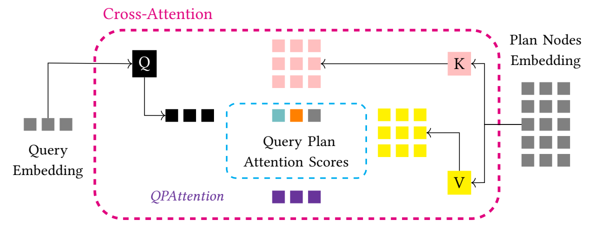

Figure 3: Attention between the query embedding and the plan nodes' embeddings.

Hence, for each condition in P_{q} and relation present in the query, we use the latent representation extracted from TaBERT by passing the query and the corresponding table where the condition applies to. We extract the representation of the respective column in the condition and the table representation to be used in the inputs of the plan encoder as described earlier. The table representation is extracted for all tables in the query, through the [CLS] token.

4.3 Attending the queries to the query plans

When the plan encoding phase is finished, QPSeeker combines the query and the plan embedding vectors into one embedding vector, as shown in Figure 3. However, the simple approach of concatenating these two vectors into one common vector does not have any semantic value, as they represent two different sources, i.e., the query and the query plan along with table data. Instead, we apply cross-attention inspired by the Perceiver architecture [11]. Furthermore, we observe that each node does not have the same impact on the plan in terms of the execution time and computational cost for the complete plan. For example, the selection of Sequential Scan instead of the use of an index over a large table with a high selective filter affects more the final execution time of the plan. Or the selection of an operator requiring more memory and hence more computational cost, like a Hash Join, will have a higher value for its cost, than an Index Scan. Therefore, we desire to give a score to each plan node and measure which nodes in the plan have the higher impact on the final estimations. To this direction, we make use of cross-attention between the query and the output of each node in the plan.

For the implementation of cross-attention, we follow the standard notation and create three matrices, the query, key and value, (QKV), which are used as projection matrices for each *QEP*. These matrices project the query and plan embeddings into a latent space, where both projections have the same dimensions. The query and plan nodes embeddings are multiplied with $Q \in \mathbb{R}^{q \times c}$ and $K, V \in \mathbb{R}^{d \times c}$ respectively, where q and d are the output sizes of the query encoder and the plan Encoder, while c is the latent space dimension. The above attention formulation corresponds to one attention head. In QPSeeker, we use multi-head attention with four attention heads. On each head, we calculate the scaled dot-product of Q, K matrices and apply softmax to the result. In this way, we calculate the attention scores between the query and the plan nodes. These scores capture the nodes in the plan with the most impact with respect to the query. Then, *QPAttention* is calculated by:

$$QPAttention = \frac{softmax(QK^T)V}{\sqrt{c}}, \qquad \text{where} \tag{1}$$

$$Q = QueryEncoder(T_q, J_q) \tag{2}$$

$$K, V = PlanEncoder(T_q, J_q, table\_data) \tag{3}$$

Finally, the output of each attention head is concatenated and passed to a dense layer, resulting into a vector with size, equal to the query embedding vector.

For queries containing no joins, the calculation of *QPAttention* does not add any value, as the query plan will contain only one operator. In this case, *QPAttention* is equal to the concatenation of the query and plan embedding vectors.

4.4 Cost Modeler

So far, we have encoded the query, the table data associated with the query plan, and for each *QEP*, we have calculated how the plan is associated with the query by weighing in the impact of each plan node on the estimations for the query through an attention mechanism. However, for each query, the space of possible plans is huge, and each plan has different execution time and computational cost. Our goal is to capture the distributions of the cardinalities, costs and execution times for the plans in the space of a query, and to generalise for the entire workload.

For this purpose, at the heart of QPSeeker lies a variational autoencoder, acting as the cost modeler. The objective of the cost modeler is not only to approximate the target distributions but also to be able to generalise on unseen queries, by providing accurate estimates for each plan node statistics, and consequently, for a whole execution plan suggested by QPSeeker, through variational inference [1]. Our belief is that execution plans with analogous complexity, in terms of runtime and execution cost, or with similar characteristics, such as relations and filters applied, if projected to a structured latent space, will have representations close to each other, depicting these similarities. The use of the VAE aims at the formation of such a latent space.

More precisely, our approach for the *Cost Modeler* is based on the following framework: Given a set W of observed variables, i.e., in our case, a workload W of *QEPs*, where each *QEP* is characterized by its cardinality, cost, and runtime, infer a latent variable z, which generates the initial observations. The described conditional probabilty can be written as:

$$p(z|W) = \frac{p(W|z)p(z)}{p(W)}$$

where the density of workload W can be computed as:

$$p(W) = \int p(W|z)p(z)\,dz$$

As we observe, the calculation of the density function for our workload W demands the computation of $p(z)$, which we do not have access to, as it is a latent variable, hence the above integral is intractable. Despite that, we can approximate the above value by applying variational inference [1].

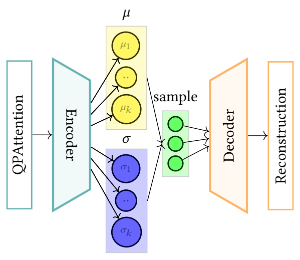

Figure 4: VAE's architecture. The latent space z formulates a mixture of univariate Gaussian distributions.

In variational inference, we specify a family of densities over latent variable z, with the purpose to find the best candidate approximation to the exact conditional. Hence, let $q(z|W)$ be the approximation of the latent variable generating the values for cardinalities, costs and runtimes for the *QEPs* in our given workload W. Since we want to find the best candidate to approximate the latent variable, the optimization problem is to minimize the error between the latent and our approximation. Since both values are density functions, our goal is to minimize the Kullback-Leibler (KL) divergence, which can be written as follows:

$$KL_{min}(q(z|W), p(z|W)) \tag{4}$$

All we need is to specify the form of the latent variable.

Forcing structure into the latent space. VAE implements the above model, and it consists of three parts: (a) the encoder, encoding the input into the latent space, (b) the sampler, sampling from the latent distribution, and (c) the decoder, which receives the sample from the latent and decodes it to the initial input.

Initially, the encoder receives the result of *QPAttention* and encodes it into a latent space, serving as the $p(W|z)$ in our framework. Then, the sampler samples a data point from this latent distribution, and finally, the decoder receives this vector from this distribution and outputs the reconstructed vector, serving as the approximation $q(z|W)$ in our framework. Finally, the reconstructed vector is passed to a linear layer, to get the estimates for a particular QEP.

As described above, in order for VAE to conform with our described framework, its latent space must describe a distribution, thus it is forced to have a structure. QPSeeker forces this structure to be a mixture of univariate Gaussians, and the latent space represents the parameters of the distributions mixture. The first half represents the mean and the other half the variance of the latent distributions, as shown in Figure 4. Finally, during training, QPSeeker minimizes: *a)* the reconstruction loss of *QPAttention* and the KL divergence described in equation 4 and *b)* the mean squared error (MSE) between the true values of QEPs and QPSeeker estimates. The loss can be written as:

$$QPSeeker_{\text{loss}} = ||x - \hat{x}||^2 + \beta * KL[N(\mu_i, \sigma_i), N(0, 1)] \tag{5}$$

where x are the estimates of QPSeeker and $\hat{x}$ the true values of a particular QEP. For $\beta > 1$, we emphasize on the KL, encouraging QPSeeker to learn broader distributions. More on the effect of β in the experiments section.

Figure 5 shows how QPSeeker has organized its latent space for QEPS produced by sampling from the *JOB* Benchmark [18]. We used the t-SNE method [30] to project the 32 latent features into 2-d plane. The color codes indicate that these QEPs have been produced from the same query template. QPSeeker has

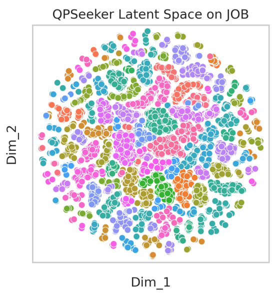

Figure 5: t-SNE projection of Cost Modeler's (VAE) latent space on JOB. The colour codes indicate query plan samples produced from the same query template.

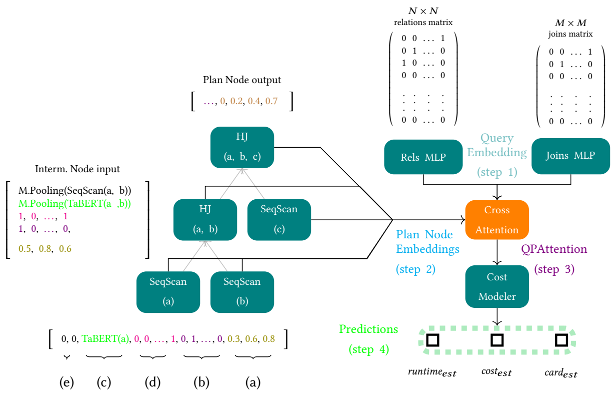

Figure 6: Running example of an input *QEP*.

organized its latent space, not only in a way that query plan samples from the same query template are close to each other, but also samples from different queries.

5 TRAINING LOOP & INFERENCE

We train under two settings: (*a*) for each query, we use the query plan provided by the DB optimizer, and (*b*) we enumerate the query plan space and extract a sample for training (Section 5.1). In all cases, the first step is the extraction of the query representation as described in Section 4.1. For example, in our training set exists a *QEP* containing the following query and execution plan:

```
query: select * from a, b, c where a.a1=b.b1 and b.b2=c.c1
       and a.a2>1

plan: 1.[SeqScan(a),0.3,0.6,0,8],    2.[SeqScan(b),0.2,0.4,0.1]
      3.[HashJoin(a, b),0.5,0.8,0.6], 4.[SeqScan(c),0.3,0.7,0.8]

      5.[HashJoin(a, b, c),0.4,0.6,0.9]
```

Figure 6 shows the running example. *Query Encoder* takes as input the relation and join matrices and produces the query embedding vector (step 1). Then, we encode each execution plan (Section 4.2), and we apply the cross-attention mechanism (Section 4.3) to create the input for the cost modeler. For each *QEP*, we get the base table representations from TaBERT and construct the input of each node. The input of each node is passed through the *Plan Encoder* resulting to the embedding vector of each node (step 2). The next step is the calculation of *QPAttention* between the query embedding and the five nodes in the plan (step 3). Then, we pass this joint embedding vector of the QEP to the variational autoencoder, and finally, (step 4) we pass the reconstructed vector into a dense layer to get the prediction for the cardinalities, costs and latencies of the encoded QEP (Section 4.4).

5.1 Query Plan Set Selection

In this section, we describe our approach for generating training data from samples of the query plan space. In order to learn the distributions of the cardinalities, costs and execution times of the query plans set, the naïve approach is to enumerate the query plan space per query and construct all possible query plans per query. As the number of relations and joins increases, the plan space is growing exponentially and the time to enumerate and execute all these plans is prohibitive.

Hence, from the query graph, we enumerate all the possible join orderings. Then, we transform each join order into the corresponding binary left-depth query plan tree, and we randomly select an operation from PostgreSQL for each node of the plan. All leaf nodes refer to table scans and all intermediate nodes are the join operations between two tables.

For each plan we construct, we calculate their corresponding measures, by a simple yet effective user-defined cost model, then we sort them based on the cost and pick the first 15% as the query plan set for a particular query. Our cost model is defined below:

```
1. Seq Scan = tbl_blocks / block_size +
           tbl_rows * cpu_tuple_cost
2. Index Scan = index_height * random_page_cost +
           index_leaf_pages / 2 * cpu_tuple_cost
3. BitmapIndexScan = index_height *
            random_page_cost + log(tbl_blocks / block_size)
4. Merge Join = (|relA| * log(|relA|) +
      |relB| * log(|relB|) + |relA| + |relB|) * cpu_tuple_cost
5. HashJoin = (|relA| + 2 * |relB|) * cpu_tuple_cost

6. NestedLoops = (|relA| * relA_blocks + relB_blocks) *
           cpu_tuple_cost
```

Our choice of left-depth trees is based on both traditional optimizers and learned approaches like Flow-Loss [26] and MTMLF [32]. We consider different join orderings because we wish our cost model to be aware of the impact of alternate join orderings on runtime and cost of a plan. This is achieved through the plan encoding process, as the relations present in a subplan are given as input both from the 1-hot encoding and from TaBERT embeddings. Additionally, even plans with the same join orderings but with different operators, are very useful, since QPSeeker takes into account all values and not just the cardinalities. This allows us to fuse the complexity of each operator into the learned cost model, as the estimates are given as input to each node and give the ability to the model to learn an internal representation of query plans with similar features, as shown in Figure 5.

Finally, all produced query plans are submitted to PostgreSQL for execution. In order to inject our plan in the optimizer, we use the PgCuckoo [9] extension with some modifications, which forces the optimizer to use our hand-crafted query plan at runtime. Moreover, we use the EXPLAIN ANALYZE functionality of the database system to get the statistics from the execution of our plan. In order to reduce the range of values among all the plans in the query workload, making the predictions for QPSeeker easier, we apply Min-Max scaling on the cardinalities of the queries, their execution times and the cost per physical node in the plans. We also apply the same process for the intermediate cardinalities of all subplans.

5.2 Inference - Monte Carlo Tree Search

After training the cost model of QPSeeker, the planner can be used for planning new queries. As mentioned before, as the number of relations increases, the number of possible execution plans grows exponentially making the plan space intractable. In order to traverse the search space fast and find a good execution plan, we use the Monte Carlo Tree Search (MCTS) [15] algorithm. In its basis, MCTS uses randomness to select the next plan operator using sampling, thus it can estimate a near optimal action in current state with low computation effort. Moreover, the fact that it chooses the best action based on long-term rewards, makes it very appealing for the query plan decision.

We use vanilla MCTS to traverse the query plan space in a bottom-up fashion. We start from base relations and apply one join at a time until all relations are present in the final plan. As a reward function, we use the Upper Confidence bounds for Trees (UCT) formula proposed be the authors:

$$\frac{r_i}{n_i} + C\sqrt{\frac{lnt}{n_i}} \tag{6}$$

where for the i-th node, r_i is its reward and calculates how many times the node is present in the best plan so far during the simulations. Next, n_i is the number of rollouts, t is the number of rollouts of the parent node, and C is the exploration coefficient parameter, ranging between [0, 1].

For the evaluation of each plan node, we use QPSeeker's internal cost model. Finally, the execution plan with the least estimated execution time is considered the best plan. The reward for each node being present in the best plan discovered so far in the simulation is one unit. For each query, we set a planning time cut-off of *200ms* and if the agent has not finished traversing the space in this time budget, we select the best plan found so far. MCTS consists of four steps:

1. *Selection.* Based on the current state of the selected subplan, the agent chooses the new plan operator in the plan with highest value, based on our policy, forming the new state of the plan.
2. *Expansion.* The agent generates all possible nodes of the query plan, based on the previously selected action in the plan.
3. *Rollout.* We start a simulation from the current state of the plan by randomly selecting the next operator to be applied, until the plan is complete.
4. *Backpropagation.* Based on the played simulation, we estimate the execution time of the simulated plan using QPSeeker's cost model and update the rewards.

6 EXPERIMENTAL SETUP

In this section, we describe our experimental setup, and then in Section 7, we present our experimental results. We evaluated QPSeeker using 3 different workloads (Table 1).

1. The Synthetic used in MSCN [14]. This workload consists of $100k$ queries with 0-2 joins per query.
2. The JOB workload is an augmentation of the Join Order Benchmark [18]. For each query, we extract sample plans from the query plan space of each query, as described in Section 5.1, resulting to $50k$ *QEPs*.

Workload	Queries	QEPs	Plan Source	Database
Synthetic	$100K$	$100K$	DB optimizer	IMDb
JOB	113	$50K$	sampling	IMDb
Stack	$6.2K$	$6.2K$	DB optimizer	Stack
JOB-Light	70	70	-	IMDb
JOB-Ext.	24	24	-	IMDb

Table 1: Evaluation workloads, queries and plan generation process. Light and Extended versions of JOB were used only for evaluation.

3. The Stack workload used in Bao [20], which contains over 18 million questions and answers from StackExchange webistes. The workload consists of $6.2K$ queries.

All workloads used in our evaluation are proposed by previous approaches and are well-studied. *Synthetic* and *JOB* are related to the *IMDb* database, with size equal to $7.2GB$, while *Stack* is equal to $100GB$. More specifically, *Synthetic* is the easiest among the workloads, as the execution of the $100K$ queries lasted almost 3 days, meaning $1.3sec$ per query on average, while *JOB* and *Stack* took 9 and 1 days, with $15.7sec$ and $2.5sec$ per query, respectively. The choice of the above workloads is based on some interesting characteristics found in their distributions. *Synthetic's* distributions for runtime and cost have the lowest values among the three workloads as expected, but shows very wide range in the cardinality distribution. It contains from highly selective queries with 1-tuple result, to queries with up to approximately 460M rows. Based on our encoding method, *Synthetic* has the most sparse representation, as the 25% of the queries consist of only one scan over a table, translating to a *QEP* containing only one 1-hot vector from the query and only one plan node. On the other hand, *JOB* and *Stack* consist of queries with up to 16 and 18 joins respectively, resulting to a much more dense input. In *JOB's* case, the runtime distribution has the highest values between the three workloads, making it the slowest one, while cardinality and cost distributions appear to have completely different distributions from the other workloads, as they are multimodal distributions. Finally, while *Stack* is connected to the largest database in our setup, we do not observe any anomalies and all values follow normal distributions with high variance. In this case, we aim to test how QPSeeker works under a much bigger database.

6.1 Competitors

We first compare QPSeeker's cost model predictions against dedicated systems on each task. Then, we compare the query plans produced by QPSeeker with two other optimizers:

1. Cost Model performance

- *Cost Estimation.* We compare our predictions on plan costs with *Zero-shot Cost Estimator* [7]. It is a db-agnostic cost estimator using features extracted from the execution plan that are common across different databases. We train Zero-Shot Cost Estimator over the databases/workloads provided by the authors and we use QPSeeker workloads/databases as inference.
- *Cardinality Estimation.* We compare our query cardinality predictions with *MSCN* [14]. It is a cardinality estimator using the relation, join and filter sets present in the query. We transformed our input workloads to be suitable for input to MSCN.

- *Runtime Prediction.* We compare QPSeeker's runtime predictions with *QPPNet* [23], which is a plan-based runtime estimator. It constructs a network similar to the tree structure of the execution plan, assigning a different MLP for each plan operator. We extended their dataset creation process to include QPSeeker workloads.

2. Query Optimization We use PostgreSQL as our baseline system. Furthermore, we use *Bao* [20], which is a RL-based optimizer providing hints to PostgreSQL planner to deactivate certain plan operators per query. We trained Bao, by letting it to gain experience through the execution of the training set of QPSeeker. Then, we use Bao as an advisor for the execution of the evaluation set.

6.2 Hyperparameters

The output size for the relations and joins MLP in *Query Encoder* is 256 each, resulting in a 512-dimensional vector. The hidden layer size of each MLP is 256. The output size of each plan node in *Plan Encoder* is equal to 950 and we extract the hidden state of the LSTM cell for each plan node. The *Cross-Attention* mechanism has $h = 4$ attention heads with size 256. The output of each head is concatenated and given as input to a linear layer with output size equal to the sum of the two MLPs from the *Query Encoder* and the output from the *Plan Encoder*. For the VAE, both the encoder and the decoder are feed forward networks consisting of 5 hidden layers each. The output of each hidden layer is cut down to the half and doubled in the decoder case similarly. We tested various sizes for all components of the architecture, where the increase of parameters in the model did not result in significant boost in predictions accuracy. We set the latent space of VAE to represent 32 latent features. For TaBERT, we extract the representation of the tuple with the highest n-gram overlap with the query, hence we set $K = 1$. The authors implemented instances for $K \in [1, 3]$ and trained both base and large instances of BERT[3]. We show the impact of each configuration in Section 7.2. We did not observe any significant difference in prediction accuracy, but higher numbers of K as using the large instance, have a noticeable impact on computation cost and response time from TaBERT. During training, we freeze TaBERT weights, to keep low the computational cost of training. For each model instance, we set the batch size equal to 16 with learning rate to 0.001.

6.3 Training Setup

In all training setups, we split the available workload into 80% - 20% training and evaluation *QEPs* sets, respectively. Especially, in the JOB training setup, where the query plans are sampled, we split the available *QEPs* at query level, thus we evaluate QPSeeker on queries never seen before. We experiment with the effect of β parameter on QPSeeker's distribution approximation (Formula (5)). For each workload, we train 3 instances of QPSeeker with β values in $[100, 200, 300]$. The β values, were extracted after monitoring the gradients of the network and the values between the KL divergence and the reconstruction loss. For inference, we experimented with the tunable bias exploration parameter $C \in [0.25, 0.5, 0.75]$, without noticing any significant difference on the final plan produced, so we set the exploration parameter $C = 0.5$. All experiments are performed on a Macbook Pro, M1 Pro and 32*GB* RAM, using PyTorch [28].

As mentioned before, we use pgCuckoo to inject our execution plans to PostgreSQL. We implemented a plan rewriter, where the produced plan from QPSeeker is rewritten, following the PgCuckoo's Haskell library format. Then, our plan is compiled into algebraic code used by the PostgreSQL executor. In this way, we are able to control the plan generation granularity at plan-operator level. This process is used for both the generation of plans through sampling using our user-defined cost model for training, as well as the plans produced during inference.

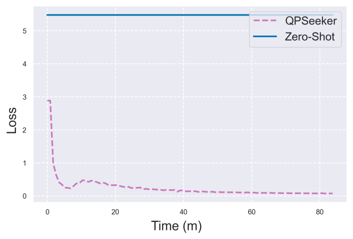

Figure 7: Train Loss through time of QPSeeker against Zero-Shot.

7 EXPERIMENTAL RESULTS

Initially, we evaluate the ability of QPSeeker to approximate the distribution of *QEPs* for each evaluation workload and we provide Q-Error percentiles on each instance. Q-Error [24] essentially measures the deviation between the predicted and true value, in orders of magnitude. Next, using the best instance per workload, we compare QPSeeker's cost model with state-of-the-art systems per task and report again Q-Error percentiles. These are presented in Section 7.1. Following, we evaluate the performance of our cost model, by executing JOB with query plans produced by a cost model trained on a completely different workload, like Synthetic. Finally, we train different instances of our cost model under different samples produced by our plan sampling method and compare the impact of different instances of TaBERT. Both of these evaluations are discussed in Section 7.2.

7.1 Cost Model Performance

First, we report Q-Error percentiles for cardinality, cost and runtime prediction of QPSeeker for diferent values of parameter β. Next, we use the best instance per workload, based on predicted runtime, and compare Q-Error percentiles with each competitor on all workloads.

7.1.1 Parameter β effect. Table 2 shows the performance of our model compared with true values for each quantity. For each workload, we highlight the model instance with the best performance regarding the runtime prediction, as this prediction is used during inference as the scoring model for MCTS. Generally, we observe that, in both JOB and Stack workloads, the smallest value of $\beta = 100$, has the best results, while for Synthetic, it is close to QPSeeker instance with $\beta = 200$. This result can be explained from the formulation of our loss function. Keeping the value of β low, favors the reconstruction loss, in other words, it focuses more on correct predictions, making QPSeeker to be stricter to its predictions.

Among datasets, we observe QPSeeker adapts really well on the "complex" workloads, but it falls short on the "easy one" (i.e., the Synthetic). This difference between the Synthetic and the other two workloads comes from the fact that the input to the Query Encoder in the former case is much sparser than the latter ones. A large subset of Synthetic queries involve only one table. For these queries, the Relations MLP gets a matrix containing the one-hot encoding in only one cell and the rest of the *NxN*

Table 2: QPSeeker Cost Model for different values of β. Best model extracted using median Q-Error (50th percentile).

		Cardinality			Cost			Runtime		
Dataset	**Perc**	$\beta = 100$	$\beta = 200$	$\beta = 300$	$\beta = 100$	$\beta = 200$	$\beta = 300$	$\beta = 100$	$\beta = 200$	$\beta = 300$
Synthetic	50%	23.72	18.49	21.02	5.31	4.20	5.11	4.20	**3.79**	4.15
	90%	1440.46	1712.01	1477.18	30.75	40.86	31.82	58.35	**71.87**	58.389
	95%	7332.00	7736.28	9047.75	2580.73	3654.32	2753.45	243.23	**323.01**	248.76
	99%	9268.34	10 025.61	1148.05	3742.72	5299.70	3993.21	302.49	**401.70**	309.37
	std	3196.24	3571.49	3893.63	827.55	1172.52	883.07	69.24	**92.47**	70.84
JOB	50%	2.40	5.79	6.23	122.56	160.80	94.62	**1.97**	2.05	2.12
	90%	77.25	95.08	100.39	122.66	160.90	150.51	**7.02**	5.75	5.31
	95%	1563.37	2137.53	2267.75	407.87	314.62	297.73	**14.83**	14.73	13.96
	99%	1570.83	2275.12	2285.32	980.30	747.23	802.18	**48.31**	59.41	64.37
	std	435.31	596.74	567.28	1734.80	1362.93	1396.26	**24.28**	25.25	27.89
Stack	50%	10.85	10.68	10.95	1.16	1.48	1.56	**2.76**	2.77	2.91
	90%	268.71	275.21	253.63	1.52	1.93	1.96	**17.44**	15.51	19.59
	95%	471.00	577.00	499.00	1.66	2.37	2.85	**39.07**	37.13	37.81
	99%	1031.86	842.3	973.96	147.81	246.28	250.31	**125.65**	142.21	109.42
	std	302.53	264.69	289.82	12.29	37.24	35.23	**22.05**	22.76	22.76

input contains zeros while the Joins MLP gets as input a matrix fulled with zeros.

This observation also gives us food for thought regarding how to train a neural model. A workload like Synthetic that contains very simple queries may not help a neural model acquire good knowledge of the complexity of the underlying schema and hence the query complexity. This is an interesting research direction.

Table 3: Cost Estimation Q-Error percentiles

W	Perc	QPSeeker	Zero-shot	PostgreSQL
Synth.	50%	4.20	1.83	4.71
	90%	40.86	26.28	18.06
	95%	3654.32	106.51	30.28
	99%	5299.70	282.174	115.34
	std	1172.52	49.54	522.61
JOB	50%	122.56	2.75	13.56
	90%	122.66	11.86	401.91
	95%	407.87	20.58	1316.60
	99%	980.30	46.16	2961.72
	std	1734.8	139.39	559.03
Stack	50%	1.16	2.52	596.91
	90%	1.52	175.74	6050.96
	95%	1.66	175.73	12 247.22
	99%	147.81	1817.71	38 145.16
	std	12.29	246.57	8395.04

7.1.2 Cost Estimation. For the Zero-Shot model, we had to extend its parser to be compatible with our workload format. We performed the evaluation suggested and implemented by the authors. We trained Zero-shot model on 19 different databases and 77 workloads (approximately 3 per database), the same used by the authors. All hyperparameters remained unchanged and we trained the model with the default setup proposed by the authors. The evaluation results are shown in Table 3.

First, we observe that for the Synthetic workload, Postgres gave significantly better predictions from the other two competitors. Next, Zero-Shot outperformed QPSeeker on JOB and achieved the best results among all systems. Finally, QPSeeker outperformed by orders of magnitude both systems on Stack. These results are very interesting, as each competitor is better at exactly one workload. By an analysis of the workloads, we observe QPSeeker could not capture the complex distributions of the first two workloads, as they form distributions with more than two modes.

Table 4: Cardinality Estimation Q-Error percentiles

W	Perc	QPSeeker	MSCN	PostgreSQL
Synth.	50%	18.49	1.22	2.07
	90%	1712.01	3.80	13.00
	95%	7736.28	7.96	27.52
	99%	10 025.61	31.59	154.66
	std	3571.49	25.34	2042.65
JOB	50%	2.40	10.42	30.46
	90%	77.25	634.92	1570.66
	95%	1563.37	2128.60	3473.50
	99%	1570.83	26 089.85	13 077.50
	std	435.31	7879.18	2294.43
Stack	50%	10.85	3.71	257.00
	90%	268.71	58.40	15 015.50
	95%	471.00	216.98	37 465.50
	99%	1031.86	940.38	275 255.55
	std	302.53	3990.69	79 522.75

7.1.3 Cardinality Estimation. For cardinality estimation, we compare our system against MSCN. For each workload, we train it with the default setup suggested by the authors. For MSCN to be compatible with Stack and JOB workloads, we had to remove any alphanumerical filters per query, as it accepts only numerical ones. The results of our evaluation are shown in Table 4. On the Synthetic workload, MSCN gives better results, as expected and is accurate until the $90th$ percentile. Next, on JOB, we observe QPSeeker to provide the best estimates, while MSCN seems to be unable to adapt to this workload from the $50th$ percentile and above. Finally, on Stack workload, QPSeeker provides decent estimates for half of the queries, while both systems perform close to each other as we go to the $99th$ percentile. Interestingly, on both complex workloads, QPSeeker outperforms PostgreSQL, with the latter having the worst performance among all three on Stack. Both Stack and JOB have complex queries with many joins, where PostgreSQL makes bad estimations.

Table 5: Execution Time Estimation Q-Error percentiles

W	Perc	QPSeeker	QPPNet	PostgreSQL
Synth.	50%	3.79	1.41	1.68
	90%	71.87	8.57	5.36
	95%	323.01	17.35	13.03
	99%	401.70	633.14	356.34
	std	92.47	115.13	514.05
JOB	50%	1.97	8.89	116.98
	90%	7.02	181.13	47 392.97
	95%	14.83	575.79	297 577.39
	99%	48.31	2682.05	3 646 587.51
	std	24.28	2165.01	624 869.24
Stack	50%	2.76	3.99	1.17
	90%	17.44	19.09	4.18
	95%	39.07	31.50	4521.73
	99%	125.65	70.04	103 700.91
	std	22.05	14.97	363 394.66

7.1.4 Execution Time Prediction. For runtime prediction of queries, we compare our system with QPPNet. Despite QPPNet's small size (approx. 4.5 MB), its complexity and the fact that each neural unit needs an optimizer separately makes the training process to be prohibitive for real-life scenarios. The results of our evaluation are shown in Table 5. QPSeeker shows that it can learn better when trained on complex workloads such as JOB and Stack. For JOB, QPSeeker provides accurate estimates up to the 90*th* percentile, while the value 7.02 on the 99*th* percentile is very satisfying. On the other hand, QPPNet manages to adapt for the majority of the queries but not close to QPSeeker's performance. Finally, on the Stack workload, both systems are very competitive with each other, with QPSeeker achieving to provide better results for the vast majority of the queries. Again, we observe that PostgreSQL does not cope well with the complex workloads.

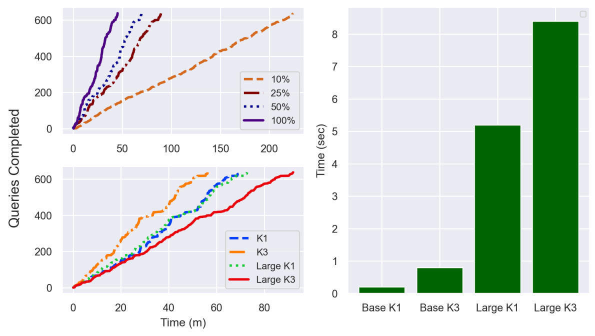

Figure 8: (Left) Sample size impact and TaBERT size impact on performance. (Right) Average Time spent on TaBERT.

In the above evaluation, we observe that PostgreSQL is competitive or better compared with the competitors in *Synthetic's* case. Before executing any queries to the system, we have updated the internal statistics using the ANALYZE command. From the insights we provided in Section 6, the input to QPSeeker is very sparse for *Synthetic* as the input for single scan queries is only 1-hot vector and a plan node for the Query and Plan Encoder, while the input to the *Joins MLP* for the entire dataset will contain at most two 1-hot vectors as input. Similarly, Zero-Shot and QPPNet rely heavily on the query plan face similar issues. On the other hand, on the other two workloads, which contain a significant bigger number of joins, we observe that PostgreSQL shows poor performance even from the *50th* percentile, while the learned approaches shine.

7.2 Query Optimization

In this section, we evaluate the performance of query plans produced by QPSeeker in comparison with PostgreSQL and Bao on two available workloads, Stack and JOB.

For Stack, QPSeeker and Bao were both trained and tested on query sets coming from the same workload. As mentioned in the training setup, test queries are never seen during training. For the JOB workload and its variations, we wish to test how well QPSeeker adapts to query workloads having completely different distributions. Hence, we train QPSeeker on Synthetic and test the query plans provided for all instances of JOB. From our workload discussion, Synthetic is a simple workload, which covers a small subset of database tables. For fair comparison, we use the instance of Bao trained on the same training set with QPSeeker. In all cases, QPSeeker consumed the time budget for planning evaluating 772 plans for Stack, 377 for JOB and 1342 for Synthetic.

7.2.1 Cost Model performance using sampling. In this section, we evaluate the performance of our cost model trained with *QEPs* produced by our proposed sampling method. We extract three random subsets from the Stack workload containing a sample of 10%, 25% and 50% of the available queries and each sample is overset of the previous (the 10% exists in both 25% and 50%). From each subset of queries, we sample query plans until we reach the initial number of available *QEPs* and train our cost model. In Figure 8, we observe that the cost model trained on the 10% of the queries is not competitve at all, while both the 25% and 50% produce execution plans having similar performance, as the one trained on the entire workload.

7.2.2 TaBERT impact. We evaluate the impact of TaBERT in our cost model. While keeping the best configuration based on Table 2, we test the performance of our cost model for K=1 and K=3, while for both base and large instances of BERT model. In Figure 8, we observe there is not a significant impact in performance, but the we observe a significant increase in time spent to extract the tables representations from TaBERT between different values of K and for different instances of BERT. This increase is expected because for K=3, TaBERT applies a rowwise Attention, which is an intense computation and finally, the large instance has 3× more parameters than base.

7.2.3 Queries executed through time. In Figure 10, we demonstrate the number of queries executed during execution per workload. For the Stack and JOB workloads, QPSeeker is very close with PostgreSQL, having very small variance in query runtime, while in the extended version, it manages to outperform all competitors. On the other hand, QPSeeker had the worst performance in JOB-Light, by a large margin. This result produced by large regressions on two memory-demanding queries in the workload. Generally, Bao did not manage to adapt to any new workloads having the worst performance except JOB-Light, where it needs the double time in comparison with PostgreSQL to execute it.

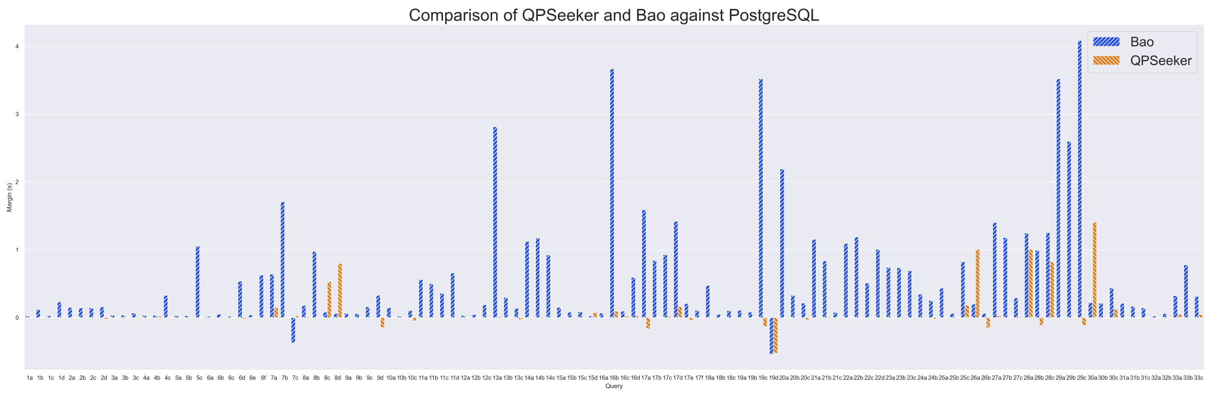

Figure 9: Query Runtimes margins of Bao (blue) and QPSeeker (orange) compared with PostgreSQL on JOB (lower is better).

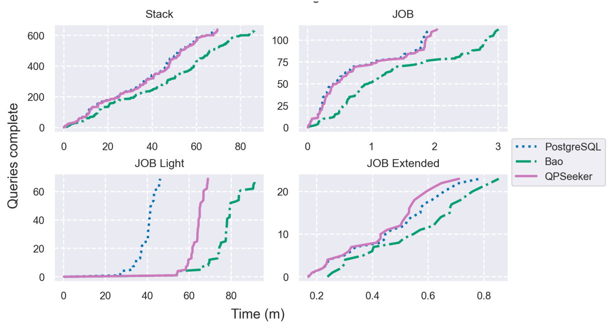

Figure 10: Number of queries completed through time.

7.2.4 JOB comparison. In Figure 9, we showcase the margin between the runtimes of QPSeeker and Bao plans, when trained on the Synthetic workload, compared with PostgreSQL on JOB. We want to check if there is any speed-up for all 113 queries in the workload.

First, we observe that Bao could not adapt to the new workload and provides a worst execution plan for the majority of the queries. In total, Bao was a minute slower than PostgreSQL across all queries in JOB, providing a better plan only on two queries. On the other hand, QPseeker is on par with PostgreSQL for the majority of the workload, performing better on some queries, and only being worse on 4 queries. This result is very encouraging, as the majority of tables being present in JOB queries, are not present in the Synthetic ones. Thus, QPSeeker not only performed well on queries never seen before, but adapted also on parts of the database that were never seen during training phase.

7.3 Discussion

We make the following observations that touch upon performance, training workloads, and research directions.

- QPSeeker can approximate quite well query cardinalities and runtimes outperforming dedicated competitors and PostgreSQL, while for cost, it could not capture complex distributions well (which points to a possible direction for future work). Furthermore, it manages to perform well for query planning (outperforming Bao).
- Furthermore, training on one workload and evaluating on a different one, we saw that QPSeeker can be as good as PostgreSQL, while Bao could not adapt to the new workload.
- QPSeeker learns better using complex workloads. A workload like Synthetic that contains very simple queries may not help a neural model acquire good knowledge of the complexity of the underlying schema and the queries. Designing appropriate training sets for neural models for databases is required.
- QPSeeker outperforms PostgreSQL (and competitors) in complex queries. That highlights a possible direction towards hybrid optimizers where a neural planner kicks in for complex queries where traditional optimizers have trouble handling.

8 CONCLUSIONS

QPSeeker is a novel database planner that combines the database data along with queries, to simultaneously learn to perform all basic tasks of a traditional optimizer, i.e., estimate the running time, computational cost and cardinality of a query, using variational inference, and it uses its rich learned model for query planning.

We showed that QPSeeker organizes its latent space in a way, where QEPs generated not only from the same but also from different queries, have latent representations close to each other. Moreover, we showed the formulation of such a cost model can provide good estimates for the majority of queries in the workload and β parameter significantly affects the final results. We showed that its cost model often outperforms its competitors, and that QPSeeker can achieve comparable or better performance, on complex workloads, like *JOB* and its variations, even when trained on a non-complex workload which can be easily constructed, like *Synthetic*. Our work opens up several interesting research directions, including work on hybrid optimizers and benchmarks.

ACKNOWLEDGMENTS

This work was supported by computational time granted from the National Infrastructures for Research and Technology S.A. (GRNET) in the National HPC facility - ARIS. This research has been funded by the European Union through the Horizon.1.3 Research and Innovation Programme, in the context of the FAIRCORE4EOSC project under grant agreement No.GA 101057264.

REFERENCES

[1] David M. Blei, Alp Kucukelbir, and Jon D. McAuliffe. 2017. Variational Inference: A Review for Statisticians. *J. Amer. Statist. Assoc.* 112, 518 (apr 2017), 859–877. https://doi.org/10.1080/01621459.2017.1285773

[2] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. 2008. SCOPE: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.* 1 (2008), 1265–1276.

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/v1/n19-1423

[4] Melissa Hall, Laurens van der Maaten, Laura Gustafson, and Aaron Adcock. 2022. A Systematic Study of Bias Amplification. https://doi.org/10.48550/ARXIV.2201.11706

[5] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. 2019. Multi-Attribute Selectivity Estimation Using Deep Learning. https://doi.org/10.48550/ARXIV.1903.09999

[6] Benjamin Hilprecht and Carsten Binnig. 2022. Zero-Shot Cost Models for Out-of-the-box Learned Cost Prediction. *Proc. VLDB Endow.* 15, 11 (2022), 2361–2374. https://www.vldb.org/pvldb/vol15/p2361-hilprecht.pdf

[7] Benjamin Hilprecht and Carsten Binnig. 2022. Zero-Shot Cost Models for Out-of-the-box Learned Cost Prediction. https://doi.org/10.48550/ARXIV.2201.00561

[8] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, Not from Queries! *Proc. VLDB Endow.* 13, 7 (mar 2020), 992–1005. https://doi.org/10.14778/3384345.3384349

[9] Denis Hirn and Torsten Grust. 2019. PgCuckoo: Laying Plan Eggs in PostgreSQL's Nest. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1929–1932. https://doi.org/10.1145/3299869.3320211

[10] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[11] Andrew Jaegle, Felix Gimeno, Andrew Brock, Andrew Zisserman, Oriol Vinyals, and Joao Carreira. 2021. Perceiver: General Perception with Iterative Attention. https://doi.org/10.48550/ARXIV.2103.03206

[12] Antonios Karvelas, Yannis Foufoulas, Alkis Simitsis, and Yannis Ioannidis. 2023. Toulouse: Learning Join Order Optimization Policies for Rule-based Data Engines. (2023).

[13] Diederik P Kingma and Max Welling. 2013. Auto-Encoding Variational Bayes. https://doi.org/10.48550/ARXIV.1312.6114

[14] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677* (2018).

[15] Levente Kocsis and Csaba Szepesvári. 2006. Bandit Based Monte-Carlo Planning. *Machine Learning: ECML* 2006, 282–293. https://doi.org/10.1007/11871842_29

[16] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. 2018. Learning to Optimize Join Queries With Deep Reinforcement Learning. *ArXiv* abs/1808.03196 (2018).

[17] Oliver Lehmberg, Dominique Ritze, Robert Meusel, and Christian Bizer. 2016. A Large Public Corpus of Web Tables Containing Time and Context Metadata. In *Proceedings of the 25th International Conference Companion on World Wide Web* (Montréal, Québec, Canada) *(WWW '16 Companion)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 75–76. https://doi.org/10.1145/2872518.2889386

[18] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 204–215. https://doi.org/10.14778/2850583.2850594

[19] Jie Liu, Wenqian Dong, Qingqing Zhou, and Dong Li. 2021. Fauce: Fast and Accurate Deep Ensembles with Uncertainty for Cardinality Estimation. *Proc. VLDB Endow.* 14, 11 (jul 2021), 1950–1963. https://doi.org/10.14778/3476249.3476254

[20] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. *Bao: Making Learned Query Optimization Practical.* Association for Computing Machinery, New York, NY, USA, 1275–1288. https://doi.org/10.1145/3448016.3452838

[21] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.* 12, 11 (jul 2019), 1705–1718. https://doi.org/10.14778/3342263.3342644

[22] Ryan Marcus and Olga Papaemmanouil. 2018. Deep Reinforcement Learning for Join Order Enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management* (Houston, TX, USA) *(aiDM'18)*. Association for Computing Machinery, New York, NY, USA, Article 3, 4 pages. https://doi.org/10.1145/3211954.3211957

[23] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *Proc. VLDB Endow.* 12, 11 (jul 2019), 1733–1746. https://doi.org/10.14778/3342263.3342646

[24] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors. *Proc. VLDB Endow.* 2, 1 (aug 2009), 982–993. https://doi.org/10.14778/1687627.1687738

[25] Parimarjan Negi, Matteo Interlandi, Ryan Marcus, Mohammad Alizadeh, Tim Kraska, Marc Friedman, and Alekh Jindal. 2021. Steering Query Optimizers: A Practical Take on Big Data Workloads. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2557–2569. https://doi.org/10.1145/3448016.3457568

[26] Parimarjan Negi, Ryan Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-Loss: Learning Cardinality Estimates That Matter. *Proc. VLDB Endow.* 14, 11 (jul 2021), 2019–2032. https://doi.org/10.14778/3476249.3476259

[27] Parimarjan Negi, Ryan Marcus, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2020. Cost-Guided Cardinality Estimation: Focus Where it Matters. *2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW)* (2020), 154–157.

[28] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

[29] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-Based Cost Estimator. *Proc. VLDB Endow.* 13, 3 (nov 2019), 307–319. https://doi.org/10.14778/3368289.3368296

[30] Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing Data using t-SNE. *Journal of Machine Learning Research* 9 (2008), 2579–2605. http://www.jmlr.org/papers/v9/vandermaaten08a.html

[31] Peizhi Wu and Gao Cong. 2021. A Unified Deep Model of Learning from both Data and Queries for Cardinality Estimation. In *Proceedings of the 2021 International Conference on Management of Data.* 2009–2022.

[32] Ziniu Wu, Pei Yu, Peilun Yang, Rong Zhu, Yuxing Han, Yaliang Li, Defu Lian, Kai Zeng, and Jingren Zhou. 2021. A Unified Transferable Model for ML-Enhanced DBMS. https://doi.org/10.48550/ARXIV.2105.02418

[33] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a Query Optimizer Without Expert Demonstrations. In *Proceedings of the 2022 International Conference on Management of Data*. ACM. https://doi.org/10.1145/3514221.3517885

[34] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2021. NeuroCard: One Cardinality Estimator for All Tables. *Proceedings of the VLDB Endowment* 14, 1, 61–73.

[35] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep Unsupervised Cardinality Estimation. *Proceedings of the VLDB Endowment* 13, 3, 279–292.

[36] Pengcheng Yin, Graham Neubig, Wen tau Yih, and Sebastian Riedel. 2020. TaBERT: Pretraining for Joint Understanding of Textual and Tabular Data. In *Annual Conference of the Association for Computational Linguistics (ACL).*

[37] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement Learning with Tree-LSTM for Join Order Selection. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1297–1308. https://doi.org/10.1109/ICDE48307.2020.00116

[38] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement Learning with Tree-LSTM for Join Order Selection. *2020 IEEE 36th International Conference on Data Engineering (ICDE)* (2020), 1297–1308.

[39] Ji Zhang. 2020. AlphaJoin: Join Order Selection à la AlphaGo. In *PhD@VLDB.*

[40] Yue Zhao, Gao Cong, Jiachen Shi, and Chunyan Miao. 2022. QueryFormer: A Tree Transformer Model for Query Plan Representation. *Proc. VLDB Endow.* 15, 8 (2022), 1658–1670. https://www.vldb.org/pvldb/vol15/p1658-zhao.pdf

[41] Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. 2021. FLAT: Fast, Lightweight and Accurate Method for Cardinality Estimation. *Proc. VLDB Endow.* 14, 9 (may 2021), 1489–1502. https://doi.org/10.14778/3461535.3461539