Neo: A Learned Query Optimizer

Ryan Marcus[1], Parimarjan Negi[2], Hongzi Mao[2], Chi Zhang[1],
Mohammad Alizadeh[2], Tim Kraska[2], Olga Papaemmanouil[1], Nesime Tatbul[23]
[1]Brandeis University [2]MIT [3]Intel Labs
[1]{ryan, chi, olga}@cs.brandeis.edu [2]{pnegi, hongzi, alizadeh, kraska, tatbul}@mit.edu

ABSTRACT

Query optimization is one of the most challenging problems in database systems. Despite the progress made over the past decades, query optimizers remain extremely complex components that require a great deal of hand-tuning for specific workloads and datasets. Motivated by this shortcoming and inspired by recent advances in applying machine learning to data management challenges, we introduce *Neo* (*Neural Optimizer*), a novel learning-based query optimizer that relies on deep neural networks to generate query executions plans. Neo bootstraps its query optimization model from existing optimizers and continues to learn from incoming queries, building upon its successes and learning from its failures. Furthermore, Neo naturally adapts to underlying data patterns and is robust to estimation errors. Experimental results demonstrate that Neo, even when bootstrapped from a simple optimizer like PostgreSQL, can learn a model that offers similar performance to state-of-the-art commercial optimizers, and in some cases even surpass them.

PVLDB Reference Format:
Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, Nesime Tatbul. Neo: A Learned Query Optimizer. *PVLDB*, 12(11): 1705-1718, 2019.
DOI: https://doi.org/10.14778/3342263.3342644

1. INTRODUCTION

In the face of a deluge of machine learning success stories, every database researcher has likely wondered if it is possible to *learn* a query optimizer. Query optimizers are key to achieving good performance in database systems, and can speed up query execution by orders of magnitude. However, building a good optimizer today takes thousands of person-engineering-hours, and is an art only a few experts fully master. Even worse, query optimizers need to be tediously maintained, especially as the system's execution and storage engines evolve. As a result, none of the freely available open-source query optimizers come close to the performance of commercial optimizers offered by IBM, Oracle, or Microsoft.

Due to the heuristic-based nature of query optimization, there have been many attempts to apply learning to query optimizers. For example, almost two decades ago, Leo, DB2's LEarning Optimizer, was proposed [53]. Leo learns from its mistakes by adjusting its cardinality estimations over time. However, Leo still requires a human-engineered cost model, a hand-picked search strategy, and a lot of developer-tuned heuristics. Importantly, Leo only improves its cardinality estimation model, and cannot further optimize its search strategy based on data (e.g., to account for uncertainty in cardinality estimates for join order selection).

More recently, the database community has started to explore how neural networks can be used to improve query optimizers [36, 60]. The majority of this work has focused on replacing a component of the optimizer with learned models. For example, DQ [25] and ReJOIN [35] use reinforcement learning combined with traditional human-engineered cost models to automatically learn search strategies and explore the space of possible join orderings. These papers show that learned search strategies can outperform conventional heuristics on a given cost model. Moreover, in addition to the cost model, these systems still rely on heuristics for cardinality estimation, physical operator selection, and index selection.

Other approaches demonstrate how machine learning can be used to achieve better cardinality estimates [22, 28, 43, 44]. However, none demonstrate that their improved cardinality estimations *actually lead to better query plans.* It is relatively easy to improve the average error of cardinality estimates, but much harder to improve estimations for the cases that actually improve query plans [27]. Furthermore, unlike join order selection, selecting join operators (e.g., hash join, merge join) and choosing indexes cannot be entirely reduced to cardinality estimation. SkinnerDB [56], showed that adaptive query processing strategies can benefit from reinforcement learning, but it requires a specialized (adaptive) query execution engine and cannot benefit from operator pipelining.

In this paper, we present *Neo* (*Neural Optimizer*), a learned query optimizer that achieves similar or improved performance compared to state-of-the-art commercial optimizers (Oracle and Microsoft) *on their own query execution engines*. Given a set of query rewrite rules to ensure semantic correctness, Neo learns to make decisions about join order, operator, and index selection. Neo optimizes these decisions using reinforcement learning, tailoring itself to the user's database instance and basing its decision on actual query latency.

Neo's design blurs the boundaries between the main components of a traditional query optimizer: cardinality estimation, the cost model, and the plan search algorithm. Neo does not explicitly estimate cardinalities or rely on hand-crafted cost models. Neo combines these two functions in a *value network*, a neural network that takes a partial query plan and *predicts* the best expected runtime that could result from completing this partial plan. Guided by the value network, Neo performs a simple search over the query plan space to make decisions. As Neo discovers better query plans,

Proceedings of the VLDB Endowment, Vol. 12, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3342263.3342644

Neo's value network improves, focusing the search on better plans. This subsequently leads to further improvements to the value network, resulting in even better plans, and so on. This *value iteration* [7] reinforcement learning procedure continues until Neo's decision-making policy has converged.

Neo required overcoming several key challenges. First, to automatically capture intuitive patterns in tree-structured query plans, we designed a *value network*, a deep neural network model, using tree convolution [40]. Second, to ensure the value network understands the semantics of a given database, we developed *row vectors*, a featurization which represent query predicate semantics automatically by using data from the underlying database. Third, we overcame reinforcement learning's infamous *sample inefficiency* by using a technique known as *learning from demonstration* [18,36]. Finally, we integrated these approaches into an end-to-end reinforcement learning system capable of building query execution plans.

While we believe Neo represents a significant step forward, Neo still has many important limitations. First, Neo requires a-priori knowledge about query rewrite rules (to guarantee correctness). Second, we restrict Neo to select-project-equijoin-aggregate queries. Third, our optimizer does not yet generalize from one database to another, as our features are specific to a schema — however, Neo *does* generalize to unseen queries (containing any number of known tables). Fourth, Neo requires a traditional query optimizer to bootstrap its learning process (although this optimizer can be simple).

Interestingly, Neo automatically adapts to changes in the accuracy of its inputs. Further, Neo can be tuned depending on the customer preferences (e.g., trade off worst-case performance vs. average performance), adjustments which are not trivial to achieve with more traditional query optimizers.

We argue that Neo *represents a step forward in building an entirely learned optimizer.* To the best of our knowledge, Neo is the first fully-learned system (modulo query rewrite rules) to construct query execution plans in an end-to-end fashion (i.e., from query latency). Neo can already be used to improve the performance of thousands of applications which rely on PostgreSQL and other open-source database systems (e.g., SQLite). We hope that Neo inspires many other database researchers to experiment with combining query optimizers and learned systems in new ways.

In summary, we make the following contributions:

- Neo, an end-to-end learning approach to query optimization, including join order, index, and physical operator selection.
- We show that, after training with a sample query workload, Neo is able to generalize even to queries it has not encountered before.
- We evaluate query encoding techniques and propose a new one, which implicitly represents correlations within the database.
- We show that, after a short training period, Neo is able to achieve performance comparable to Oracle's and Microsoft's query optimizers *on their own respective execution engines.*

Next, in Section 2, we provide an overview of Neo's learning framework. Section 3 describes how queries and query plans are represented by Neo. Section 4 explains Neo's value network, the core learned component of Neo. Section 5 describes row vectors, an optional learned representation of the underlying database that helps Neo understand correlation within the user's data. We present an experimental evaluation of Neo in Section 6, discuss related works in Section 7, and offer concluding remarks in Section 8.

2. LEARNING FRAMEWORK OVERVIEW

We next discuss Neo's system model, depicted in Figure 1, and overall reinforcement learning strategy. Neo operates in two phases: an initial phase, in which expertise is collected from an expert optimizer, and a runtime phase, where queries are processed.

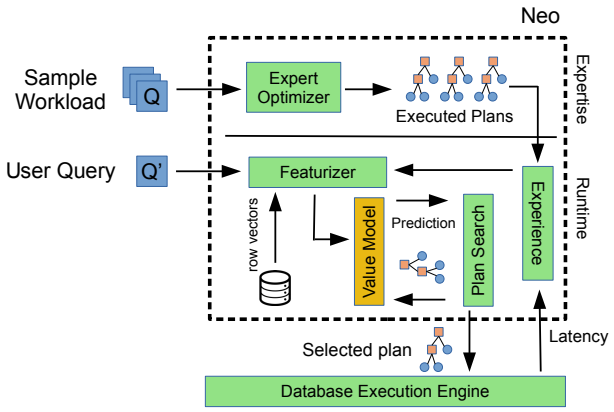


Figure 1: Neo system model

Expertise Collection In the first phase, labeled *Expertise*, Neo generates experience from a traditional query optimizer, as proposed in [36]. Neo assumes the existence of a *Sample Workload* consisting of queries representative of the user's total workload and of the underlying engine's capabilities (i.e., exercising a representative set of operators). Additionally, we assume Neo has access to a simple, traditional rule- or cost-based *Expert Optimizer* (e.g., Selinger [51], PostgreSQL [3]). Neo uses this optimizer *only* to create query execution plans (QEPs) for each query in the sample workload. These QEPs, along with their latencies, are added to Neo's *Experience* (a set of plan/latency pairs), which are used as a starting point in the model training phase. Note that the expert optimizer can be *unrelated* to the underlying execution engine.

Model Building With the collected experience, Neo builds an initial *Value Model*. The value model is a deep neural network designed to predict the *final* execution time of a given partial or complete plan. We train the value network using the collected experience in a supervised fashion. This process involves transforming each collected query into features (*Featurizer*). These features contain query-level information (e.g., join graph) and plan-level information (e.g., join order). Neo can work with a number of different featurizations, ranging from simple one-hot encodings to more complex embeddings (Section 5). Neo's value network uses tree convolution [40] to process the tree-structured QEPs (Section 4.1).

Plan Search Once query-level information has been encoded, Neo uses the value model to search over the space of QEPs (i.e., selection of join orderings, join operators, and indexes) and discover the plan with the minimum predicted execution time (i.e., value). Since the space of all execution plans for a particular query is far too large to exhaustively search, Neo uses the learned value model to guide a best-first search of the space (Section 4.2). A complete plan created by Neo, which includes a join ordering, join operators (e.g. hash, merge, loop), and access paths (e.g., index scan, table scan) is sent to the underlying execution engine, which is responsible for applying semantically-valid query rewrite rules (e.g., inserting necessary sort operations) and executing the final plan. This ensures the correctness of the generated execution plans.

Model Retraining As Neo optimizes more queries, the value model is iteratively improved and custom-tailored to the user's database. This is achieved by incorporating newly collected experience regarding each executed QEP. Specifically, once a QEP is chosen for a particular query, it is sent to the underlying execution engine, which processes the query and returns the result to the user. Additionally, Neo records the final execution latency of the QEP, adding the plan/latency pair to its *Experience*. Then, Neo retrains the value model based on this experience, iteratively improving its estimates.

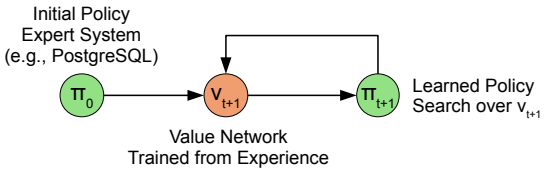


Figure 2: Value iteration

Discussion This process – searching and model retraining – is repeated for each query sent by the user. Neo's architecture is designed to create a corrective feedback loop: when Neo's learned cost model guides Neo to a query plan that Neo predicts will perform well, but then the resulting latency is high, Neo's cost model learns to predict a higher cost for the poorly-performing plan. Thus, Neo is less likely to choose plans with similar properties to the poorly-performing plan in the future. As a result, Neo's cost model becomes more accurate, effectively *learning from its mistakes.*

Neo represents query optimization as an Markov decision process (MDP, formalized in Section 3.1), in which each *state* corresponds to a partial query plan, each *action* corresponds to a step in building a query plan in a bottom-up fashion, and a *reward* is given only at the final (terminal) state based on the plan's latency. Neo's approach to navigating this MDP is called *value iteration* [7]. As depicted in Figure 2, a function is trained to approximate the *utility* (value) of a particular state based on previous experience. This function, which we call the *value network*, is then used to create a policy. Traditionally, the created policy is simple, like greedily selecting actions based on the value network.

Neo builds on the traditional value iteration model in two ways. First, Neo does not greedily follow the suggestions of the value network: it has recently been shown [33, 52] that using the trained value network as a heuristic to guide a search can improve results. Second, Neo does not "start from scratch," but rather bootstraps from a dataset of query execution plans built by a traditional query optimizer (which was designed by human experts). This avoids reinforcement learning's infamous *sample inefficiency* [18,48]: without bootstrapping, reinforcement learning algorithms may require millions of iterations [38] before becoming competitive with systems built manually by human experts. Intuitively, bootstrapping from an expert source (learning from demonstration) mirrors how young children acquire language or learn to walk by imitating adults (experts), and has been shown to drastically reduce the time required to learn a good policy [18, 49]. This is especially critical for database management systems: each iteration requires a query execution, and users are likely unwilling to execute millions of queries before achieving performance on-par with current optimizers. Worse yet, executing a poor query plan takes *longer* than executing a good plan, so the initial iterations would take an infeasible amount of time to complete [36].

An important aspect of any reinforcement learning system is *balancing exploration and exploitation*. Neo exploits knowledge through its plan search procedure, leaning heavily on the value network to guide its best-first search. As in value iteration [38], Neo ensures that new policies are explored through model retraining: each time the value network is retrained, its weights are reset to random values, and the entire network is trained against the collected experience. This ensures that the value network's prediction for unseen query plans have a high degree of stochasticity (as unseen query plans are "off manifold" [10, 33]). We also note that the architecture of Neo closely mirrors that of AlphaGo [52], a reinforcement learning system created to play the game Go. Due to space constraints, a detailed comparison between Neo and AlphaGo is available in Section 2 of the online appendix [34].

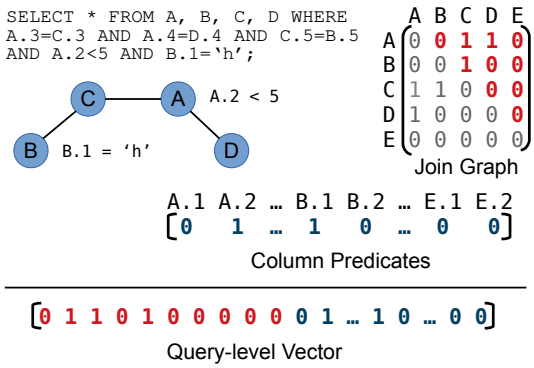


Figure 3: Query-level encoding

3. QUERY FEATURIZATION

In this section, we describe how query plans are represented as vectors, starting with some necessary notation.

3.1 Notation

For a query q, we define the set of base relations used in q as $R(q)$. A partial execution plan P for a query q (denoted $Q(P) = q$) is a forest of trees representing an execution plan that is still being built. Each internal (non-leaf) tree node is a join operator $\bowtie_i \in J$, where J is the set of possible join operators (e.g., hash $\bowtie_H$, merge $\bowtie_M$, loop $\bowtie_L$) and each leaf node is either a table scan, an index scan, or an *unspecified* scan over a relation $r \in R(q)$, denoted $T(r)$, $I(r)$, and $U(r)$ respectively.[1] An unspecified scan is a scan that has not been assigned as either a table or an index scan yet. For example, a partial query execution plan could be denoted as:

$$[(T(D) \bowtie_M T(A)) \bowtie_L I(C)], \quad [U(B)] \tag{1}$$

Here, the type of scan for B is unspecified, and no join has been selected to link B with the rest of the plan. The plan does specify a table scan of table D and A, which feed into a merge join, whose result will then be joined using a loop join with C.

A *complete* execution plan is a plan with only a root and no unspecified scans; all decisions on how the plan should be executed have been made. We say that one execution plan P_i is a *subplan* of another execution plan P_j, written $P_i \subset P_j$, if P_j could be constructed from P_i by (1) replacing unspecified scans with index or table scans, or (2) combining subtrees in P_i with a join operator.

Building a complete execution plan can be viewed as a Markov decision process (MDP). The initial state of the MDP is a partial plan where every scan is unspecified and there are no joins. Each action involves either (1) fusing together two roots with a join operator or (2) turning a unspecified scan into a table or index scan. More formally, every action transforms the current plan P_i into a any plan P_j such that $P_i \subset P_j$. The reward of every action is zero, except for the final action, which has a reward equal to the latency of the produced execution plan. Like prior work [25, 35], this formulation has the advantage of being "loopless": one always arrives at a complete query execution plan after a finite number of actions.

3.2 Encodings

Neo uses two encodings: a *query encoding*, which encodes information regarding the query, but is independent of the query plan, and a *plan encoding*, which represents the partial execution plan.

Query Encoding The representation of query-dependent but plan-independent information is similar to previous work [25, 35, 43],

[1]Neo can trivially handle additional scan types, e.g., bitmap scans.

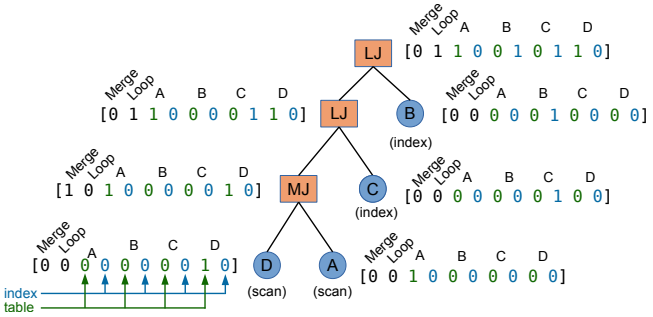


Figure 4: Plan-level encoding

and consists of two components. The first component encodes the query's join graph as an adjacency matrix, e.g. in Figure 3, the 1 in the first row, third column corresponds to the join predicate connecting A and C. Both the row and column corresponding to the relation E are empty, because E is not involved in the example query. For simplicity, we assume that at most one foreign key exists between each relation. However, the representation can easily be extended to include multiple foreign keys (e.g., by using the index of the relevant key instead of "1"). Furthermore, since this matrix is symmetrical, we only encode the upper triangular portion (red).

The second component of the query encoding is the column predicate vector. In Neo, we currently support three increasingly powerful variants, with varying levels of precomputation requirements:

1. `1-Hot` (*existence* of a predicate): a simple "one-hot" encoding of which attributes are involved in any query predicate. The length of the one-hot encoding vector is the number of attributes over all database tables. For example, Figure 3 shows the "one-hot" encoded vector with the positions for attribute $A.2$ and $B.1$ set to 1, since both attributes are used as part of predicate. Join predicates are not considered here. The learning agent *only* knows whether an attribute is present in a predicate or not. While naive, the `1-Hot` representation can be built without any access to the underlying database.
2. `Hist` (*selectivity* of a predicate): an extension of the `1-Hot` encoding which replaces "0" or "1" with the predicted selectivity of that predicate (e.g., $A.2$ could be 0.2, if we predict a selectivity of 20%). For predicting selectivity, we use an off-the-shelf histogram approach with uniformity assumptions.
3. `R-Vector` (*semantics* of a predicate): the most advanced encoding, using *row vectors*. Based on word2vec [37], a natural language processing model, each entry in the column predicate vector is replaced with a vector containing semantic information related to the predicate. This encoding requires building a model over the data in the database, and is the most expensive option. We discuss row vectors in Section 5.

More powerful the encodings provide more degrees of freedom for the model to learn complex relationships. However, this does not mean that simpler encodings preclude the model from learning complex relationships. For example, even though `Hist` does not encode correlations between tables, the model might still learn about them and accordingly correct the cardinality estimations internally, e.g. from repeated observation of query latencies. But the `R-Vector` encoding make Neo's job easier by providing a semantically-enhanced representation of the query predicate.

Plan Encoding In addition to the query encoding, we also require a representation of partial or complete query execution plan. While prior works [25, 35] have flattened the tree structure of each partial execution plan, our encoding *preserves the inherent tree structure* of execution plans. We transform each node of the partial execution plan into a vector, creating a tree of vectors, as shown in Figure 4. While the number of vectors (i.e., number of tree nodes) can increase, and the structure of the tree itself may change (e.g., left deep or bushy), every vector has the same number of columns.

This representation is created by transforming each node into a vector of size $|J| + 2|R|$, where $|J|$ is the number of join types, and $|R|$ is the number of relations. The first $|J|$ entries of each vector encode the join type (e.g., in Figure 4, the root node uses a loop join), and the next $2|R|$ entries encode which relations are used, and the associated scan type (table, index, or unspecified). For leaf nodes, this subvector is a one-hot encoding, unless the leaf represents an unspecified scan, in which case it is treated as though it were both an index scan and a table scan (a 1 is placed in both the "table" and "index" columns). For internal nodes, these entries are the union of the corresponding children nodes. For example, the bottom-most loop join in Figure 4 has 1s in the positions corresponding to table scans over A and D and an index scan over C.

Note that this representation can contain two partial query plans (i.e., several roots) which have yet to be joined, e.g. to represent partial plan in Equation 1, when encoded, the $U(B)$ root node would be encoded as: [0000110000]. The purpose of these encodings is merely to provide a representation of execution plans to Neo's value network, described next.

4. VALUE NETWORK

Next, we present Neo's *value network*, a neural network which is trained to predict the *best-possible query latency* for a partial execution plan P_i: in other words, the best-possible query latency achievable by a complete execution plan P_f such that $P_i \subset P_f$. Since knowing the best-possible execution plan for a query ahead of time is impossible, we approximate the best-possible query latency with the best query latency *seen so far by the system*.

Let Neo's *experience* E be a set of complete query execution plans $P_f \in E$ with known latency $L(P_f)$. We train a model M to approximate, for all P_i that are a subplan of any $P_f \in E$:

$$M(P_i) \approx \min\{C(P_f) \mid P_i \subset P_f \wedge P_f \in E\}$$

where $C(P_f)$ is the *cost* of a complete plan. The user can change the cost function to alter the behavior of Neo. For example, if the user is concerned only with minimizing total query latency across the workload, the cost could be defined as the latency, i.e., $C(P_f) = L(P_f)$. However, if instead the user prefers to ensure that every query q in a workload performs better than a particular baseline, the cost function can be defined as

$$C(P_f) = L(P_f)/Base(P_f),$$

where $Base(P_f)$ is latency of plan P_f with that baseline. Regardless of how the cost function is defined, Neo will attempt to minimize it over time. The model is trained by minimizing a loss function [50]. We use a simple L2 loss function:

$$(M(P_i) - \min\{C(P_f) \mid P_i \subset P_f \wedge P_f \in E\})^2.$$

The same query plan may exhibit different latencies depending on external state (cache, concurrent transactions). By default, Neo's value model will try to predict the final *average* latency of a query plan (this minimizes the L2 loss). However, depending on the user's requirements, the loss function could be modified to encourage the value network to predict the final *worst* observed latency (e.g., choose query plans that are robust to cache state), or to predict the *best* observed latency (e.g., choose query plans that assume the correct data is currently cached). If desired, one could even use a piecewise loss function to favor the worst, average, or best case for different queries in the user's workload.

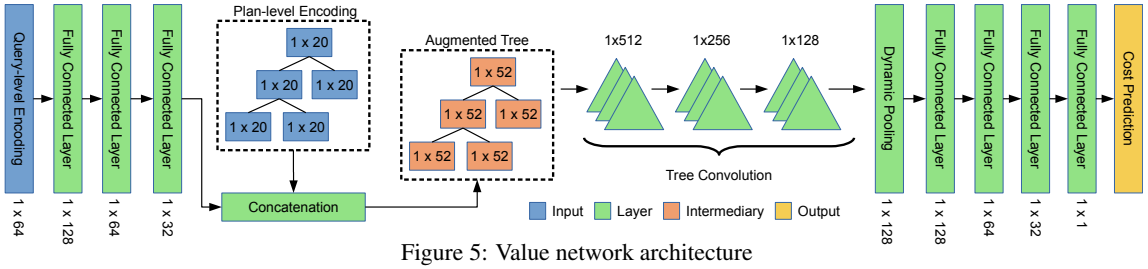


Figure 5: Value network architecture

Network Architecture The architecture of the Neo value network is shown in Figure 5.[2] The architecture was designed to create an *inductive bias* [33] suitable for query optimization: the structure of the neural network itself is designed to reflect an intuitive understanding of what causes query plans to be fast or slow. Humans studying query plans learn to recognize suboptimal or good plans by pattern matching: a merge join on top of a hash join with a shared join key is likely inducing a redundant sort or hash; a loop join on top of two hash joins is likely highly sensitive to cardinality estimation errors; a hash join using a fact table as the "build" relation likely incurs spills; a series of merge joins that do not require re-sorting is likely to perform well, etc. Our insight is that all of these patterns can be recognized by analyzing subtrees of a query execution plan. Neo's model architecture is essentially a large bank of these patterns that are learned *automatically, from the data itself*, by taking advantage of a technique called *tree convolution* [40].

As shown in Figure 5, when a partial query plan is evaluated by the model, the query-level encoding is fed through a number of fully-connected layers, each decreasing in size. The vector outputted by the third fully connected layer is concatenated with the plan-level encoding, i.e., each tree node (the same vector is added to all tree nodes). This is a standard technique, known as "spatial replication" [52, 62], for combining fixed-size data (query-level encoding) and dynamically-sized data (plan-level encoding). Once each tree node vector has been augmented, the forest of trees is sent through several tree convolution layers [40], an operation that maps trees to trees. Afterwards, a dynamic pooling operation [40] is applied, flattening the tree structure into a single vector. Several additional fully connected layers are used to map this vector to a single value, used as the model's prediction for the inputted plan. A formal description of the value network model is given in [34].

4.1 Tree Convolution

Neural network models like CNNs [29] take input tensors with a fixed structure, such as a vector or an image. For Neo, the features embedded in each execution plan are structured as nodes in a tree (e.g., Figure 4). Thus, we use tree convolution [40], an adaption of traditional image convolution for tree-structured data.

Tree convolution is a natural fit for Neo. Similar to the convolution transformation for images, tree convolution slides a set of *shared* filters over each part of the plan tree. Intuitively, these filters can capture a wide variety of local parent-children relations. For example, filters can look for hash joins on top of merge joins, or a join of two relations when a particular predicate is present. The output of these filters provides signals utilized by the final layers of the value network; filter outputs could signify relevant factors such as when the children of a join operator are sorted (suggesting a merge join), or a filter might estimate if the right-side relation of a join will have low cardinality (suggesting that an index may be useful). We provide two concrete examples later in this section.

[2]We omit activation functions, present between each layer, from our diagram and our discussion.

Since each node of the query tree has exactly two child nodes, each filter consists of three weight vectors, e_p, e_l, e_r. Each filter is applied to each local "triangle" formed by the vector x_p of a node and two of its left and right child, x_l and x_r ($\vec{0}$ if the node is a leaf), to produce a new tree node x'_p:

$$x'_p = \sigma(e_p \odot x_p + e_l \odot x_l + e_r \odot x_r).$$

Here, $\sigma(\cdot)$ is a non-linear transformation (e.g., ReLU [16]), $\odot$ is a dot product, and x'_p is the output of the filter. Each filter thus combines information from the local neighborhood of a tree node. The same filter is "slid" across each tree in a execution plan, allowing a filter to be applied to plans of arbitrary size. A set of filters can be applied to a tree in order to produce another tree with the same structure, but with potentially different sized vectors representing each node. In practice, hundreds of filters are applied.

Since the output of a tree convolution is another tree, multiple layers of tree convolution filters can be "stacked." The first layer of tree convolution filters will access the augmented execution plan tree (i.e., each filter will be slid over each parent/left child/right child triangle of the augmented tree). The amount of information seen by a particular filter is called the filter's *receptive field* [31]. The second layer of filters will be applied to the output of the first, and thus each filter in this second layer will see information derived from a node n in the original augmented tree, n's children, and n's grandchildren: each tree convolution layer thus has a larger receptive field than the last. As a result, the first tree convolution layer learns simple features (e.g., recognizing a merge join on top of a merge join), whereas the last tree convolution layer learns complex features (e.g., recognizing a left-deep chain of merge joins).

We present two concrete examples that show how the first layer of tree convolution can detect interesting patterns in query execution plans. In Example 1 of Figure 6a, we show two execution plans that differ only in the topmost join operator (a merge join and hash join). As depicted in the top portion of Figure 6b, the join type (hash or merge) is encoded in the first two entries of the feature vector in each node. A tree convolution filter (Figure 6c top), comprised of three weight vectors with $\{1, -1\}$ in the first two positions and zeros for the rest, will serve as a "detector" for query plans with two sequential merge joins. This can be seen in Figure 6d (top): the root node of the plan with two sequential merge joins receives an output of 2 from this filter, whereas the root node of the plan with a hash join on top of a merge join receives an output of 0. Subsequent tree convolution layers can use this information to form more complex detectors, like to detect three merge joins in a row (a pipelined query execution plan), or a mixture of merge joins and hash joins (which may induce re-hashing or re-sorting).

In Example 2, Figure 6, suppose tables A and B are sorted on the same key, and are thus ideally joined together with a merge join, but that C is not sorted. The filter shown in Figure 6(c, bottom) serves as a detector for query plans that join A and B with a merge join, behavior that is likely desirable. The top weights (e_p) recognize the merge join, and the right weights (e_r) recognize table B over all other tables. The result of this convolution (Figure 6d, bottom)

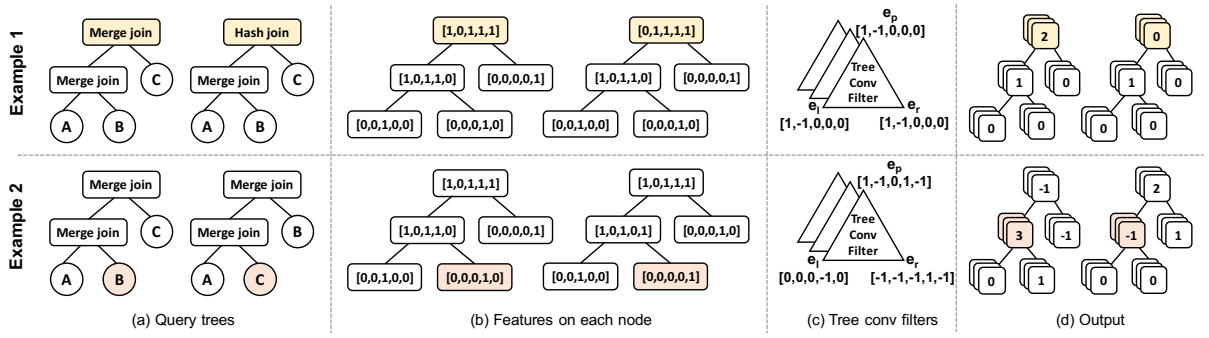


Figure 6: Tree convolution examples

shows its highest output for the merge join of A and B (first plan), and a negative output for the merge join of A and C (second plan).

In practice, filter weights are *learned* over time, and not configured by hand. Performing gradient descent to update filter weights will cause filters that correlate with latency (helpful features) to be rewarded (remain stable), and filters with no clear relationship to latency to be penalized (pushed towards more useful values). This creates a corrective feedback loop, resulting in the development of filterbanks which extract useful features [29].

4.2 DNN-Guided Plan Search

The value network predicts the quality of an execution plan, but does not directly give an execution plan. Following recent works [4, 52], we combine the value network with a search technique to generate plans, resulting in a *value iteration technique [7]*.

Given a trained value network and an incoming query q, Neo performs a search of the plan space for a given query. In some ways, this search mirrors the search process used by traditional database optimizers, with the trained value network taking on the role of the database cost model. However, unlike these traditional systems, the value network *does not predict the cost of a subplan*, but rather *the best possible latency achievable from an execution plan that includes a given subplan*. This difference allows us to perform a best-first search [12] to find an execution plan with low expected cost. Essentially, this amounts to repeatedly exploring the candidate with the best predicated cost until a halting condition occurs.

The search process for query q starts by initializing an empty min heap to store partial execution plans. This min heap is ordered by the value network's estimation of each partial plan's cost. Initially, a partial execution plan with an unspecified scan for each relation in $R(q)$ is added to the heap. For example, if $R(q) = \{A, B, C, D\}$, then the heap is initialized with P_0:

$$P_0 = [U(A)], \quad [U(B)], \quad [U(C)], \quad [U(D)].$$

Each search iteration begins by removing the subplan P_i at the top of the min heap. We enumerate P_i's children, $Children(P_i)$, scoring each child using the value network and adding them to the min heap. Intuitively, the children of P_i are all the plans creatable by specifying a scan in P_i or by joining two trees of P_i with a join operator. Formally, we define $Children(P_i)$ as the empty set if P_i is a complete plan, and otherwise as the set of available actions at this state of the MDP (see Section 3.1). Once each child is scored and added to the min heap, another search iteration begins, exploring the next most promising plan. Each step of search operation takes $O(\log n)$ time, where n is the size of the min heap.

While this process could be terminated when a leaf (a complete plan) is found, this search procedure can easily be transformed into a *anytime search algorithm* [63]: an algorithm that continues to find better results until a fixed time cutoff. In this variant, Neo continues exploring the most promising nodes from the heap until a time threshold is reached, at which point the most promising complete execution plan is returned. This gives the user control over the tradeoff between planning time and execution time. Users could select a different time cutoff for different queries depending on their needs. In the event that the time threshold is reached before a complete execution plan is found, Neo's search procedure enters a "hurry up" mode [55], and greedily explores the most promising children of the last plan explored until a leaf is reached. The cutoff time should be tuned on a per-application bases. We find that 250ms is sufficient for a wide variety of workloads (Section 6.6).

5. ROW VECTOR EMBEDDINGS

Neo can represent query predicates in a number of ways, including a simple one-hot encoding (`1-Hot`) or a histogram-based representation (`Hist`), as described in Section 3.2. Here, we motivate and describe *row vectors*, Neo's most advanced option for representing query predicates (`R-Vector`).

While cardinality estimation is critical to the success of traditional query optimizers [26, 30], database systems often make simplifying assumptions, such as uniformity, independence, and/or the principle of inclusion that often undermine this goal [27]. Neo, takes a different approach: instead of making simplifying assumptions about data distributions and attempting to directly estimate predicate cardinality, we build a semantically-rich, vectorized representation of query predicates that can serve as an input to Neo's value model, enabling the network to learn *generalizable* insights into data correlations. Following recent work in semantic querying [9], entity matching [41], data discovery [14], and error detection [17], we build a vectorized representation of each query predicate *based on data in the database itself*.

Our row vector approach is based on the popular and well-studied word2vec algorithm [37], a way of transforming natural language words (e.g., English words) into vectors. While these vectors are meaningless on their own, the *distances between them have semantic meaning*: for example, the distance between "spaghetti" and "pasta" will be small, whereas the distance between "banana" and "doorknob" will be large. Intuitively, word2vec works by taking advantage of a word's context: words that frequently appear nearby in text are assigned similar vector representations, and words that rarely do so are assigned dissimilar vectors (e.g. "At the Italian restaurant, I ordered..."). In Neo, we treat each row of each table in a database as a sentence, and we treat each column value of a table row as a word. Thus, values that frequently co-occur in rows are mapped to similar vectors. We call these vectors *row vectors*. Neo's value network can take these row vectors as inputs, and use them to identify correlations within the data and predicates with syntactically-distinct but semantically-similar values (e.g., both "action" and "adventure" frequently co-occur with "superhero").

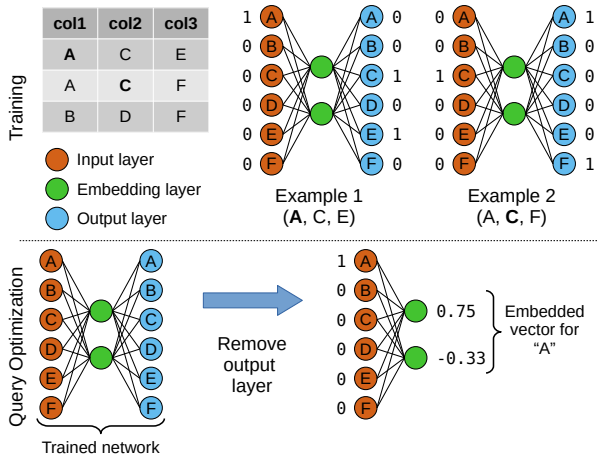


Figure 7: Row vector embedding process

The remainder of this section first gives a high-level overview of how Neo's row vectors are built, and then explores why row vectors are effective at capturing correlations in real-world data. For details, see the online appendix [34].

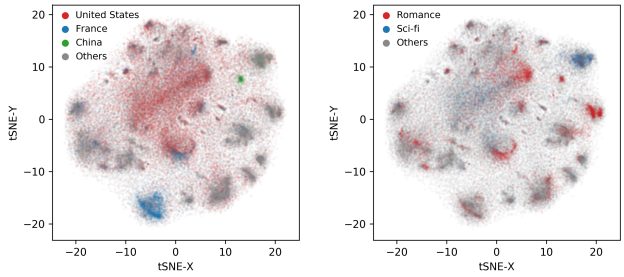
5.1 R-Vector Featurization

At a high level, our goal is to build a semantically rich representation of a query predicate which Neo can use as an input. For example, if a query over the IMDB movie dataset / JOB dataset [26] looks for all actors in movies tagged with "marvel-comics", the query will return many actors who play superheros. Similarly, if a query looks for all actors in movies tagged with "avengers", the query will also return many actors who play superheros. However, a query for all actors in movies tagged with "romance" is unlikely to return many superhero actors. Thus, we want to create a vectorized representation of "marvel-comics" that is similar to "avengers" but dissimilar to "romance". Given such a vectorization, Neo will have a better chance of making good predictions about a query for "avengers" movies after having seen a query for "marvel-comics" movies, thus giving Neo more opportunities to *generalize*.

Neo's row vector encoding requires two steps (Figure 7). Before query optimization, a *training* step learns an embedding with a specialized neural network. During query optimization, the output layer of the specialized neural network is removed, creating a truncated network which maps inputs to an embedded vector [34].

Training To generate row vectors, we use word2vec — a natural language processing technique for embedding contextual information about collections of words [37]. We build an embedding of each value in the database using an off-the-shelf word2vec implementation [47]. We depict this process in the top half of Figure 7.

We first construct a three-layer neural network, called the embedding network, with equally-sized input and output layers. The neural network will be trained to map each one-hot encoded value in the database to an output vector representing the value's context. For example, the top half of Figure 7, Example 1, shows how the embedding network is trained to map an input of "A" to an output vector representing "C" and "E", corresponding to the first row in the example table. For this first row, the embedding network is also trained to map "C" to an output vector representing "A" and "E", as well as to map "E" to an output vector representing "A" and "C". This procedure is repeated for each row in the database (e.g., Example 2). Note that the embedding network will never achieve a high level of accuracy: "A" may appear in multiple contexts, making this impossible. The goal of the algorithm is to capture statistical relationships between database values and their context.



(a) Birthplace of each actor

(b) Top actors in each genre

Figure 8: The same t-SNE projection (each axis is a unitless quantity) of embedded actor names, colored by (a) birthplace and (b) genre: the same embedding automatically captures multiple correlations. Correlations appear as semantically meaningful clusters.

Query optimization The bottom half of Figure 7 depicts how Neo builds row vector encodings during query optimization. After the embedding network is trained, the output layer is removed, resulting in a two layer network (the weights representing the transformation from the embedding layer to the output layer may also be discarded). This truncated network can be used by Neo to build a vectorized representation of a database value by passing it through the input layer and recording the value of the embedding layer.

To encode a query predicate, we combine information about the predicate operator (e.g., `LIKE` or `!=`) with the embedded vector. In the simplest case, a query predicate is in the form of `tbl.attr OP VALUE`, for example, `actor.name = "Robert Downey Jr"`. For these simple cases, the query predicate can be encoded by concatenating a one-hot encoding of the predicate operator (e.g., `=`) with the embedded vector the predicate value (e.g., `"Robert Downey Jr"`). This concatenated vector replaces the simple 0 or 1 used in the `1-Hot` encoding (Section 3.2).

Embedded vectors can be combined and searched to handle wildcard `LIKE` queries or complex logical queries (e.g., *AND*s, *OR*s). For example, Neo handles wildcard queries by searching for an example of a match in the database, and then using the embedded value of that match [34]. The embeddings can be improved by partially denormalizing the database, allowing the word2vec model to capture cross-table correlations. Our word2vec training process is open source, and available on GitHub [1].

Example Next, we explore an example trained word2vec model on the IMDB / JOB dataset [26]. After training a row vector model on the entire IMDB dataset, we used t-SNE[3] to project the embedded vectors of actor names space into two-dimensional space for plotting [58]. The results plotted in Figure 8 present a visual example of how row vectors capture semantic correlations across database tables. As shown, various semantic groups (e.g., Chinese actors, Sci-fi movie actors) are clustered together. Intuitively, this provides helpful signals to estimate query latency given similar predicates: as many of the clusters in Figure 8 are linearly separable, their boundaries can be learned by machine learning algorithms. In other words, since predicates with similar *semantic* values (e.g., two American actors) are likely to have similar *correlations* (e.g., be in American films), representing the semantic value of a query predicate allows the value network to recognize similar predicates and thus better generalize to unseen predicates.

[3]The t-SNE algorithm finds low-dimensional embeddings of high-dimensional spaces that maintain distances between pairs of points: points that are close together (far apart) in the low-dimensional space are close together (far apart) in the high-dimensional space.

6. EXPERIMENTS

We evaluated Neo's performance using both synthetic and real-world datasets to answer the following questions: (1) how does the performance of Neo compare to commercial, high-quality optimizers, (2) how well does Neo generalize to new queries, (3) how much overhead does Neo's training and execution incur, (4) how do the different encoding strategies impact query latency, (5) how do other parameters (e.g., search time or loss function) impact the overall performance, and finally, (6) how robust is Neo to estimation errors. Unless otherwise stated, queries are executed on a server with 32GB of RAM, an Intel Xeon CPU E5-2640 v4, and a solid-state drive. Each DBMS was configured according to the "best practices" guide provided by the distributing organization.

6.1 Setup

We evaluate Neo across a number of different database systems, using three different benchmarks:

1. `JOB`: the join order benchmark [26], with a set of queries over the Internet Movie Data Base (IMDB) consisting of complex predicates, designed to test query optimizers.

2. `TPC-H`: the standard TPC-H benchmark [45], using a scale factor of 10.

3. `Corp`: a 2TB dataset together with 8,000 unique queries from an internal dashboard application, provided by a large corporation (on the condition of anonymity).

Unless otherwise stated, all experiments are conducted by randomly placing 80% of the available queries into a training set, and using the other 20% of the available queries as a testing set. In the case of TPC-H, we generated 80 training and 20 test queries based on the benchmark query templates without reusing templates between training and test queries.

Each result presented is the median of 50 randomly initialized runs. Neural networks are trained with Adam [21]. Layer normalization [5] is used for training stability. Activation functions are "leaky ReLUs" [16]. We use a search time cutoff of 250ms. The network architecture follows Figure 5, which we selected after testing several variants on a small subset of `JOB`. except the size of the plan-level encoding is dependent on the encoding strategy selected. Row vectors are build using partial denormalization [34].

We compare Neo against two open-source (PostgreSQL 11.2, SQLite 3.27.1), and two commercial (Oracle 12c, Microsoft SQL Server 2017 for Linux) database systems; specifically, we train Neo to build query plans for each of these systems, and then compare Neo's query plans against those produced by each system's query optimizer. Due to the license terms [46] of Microsoft SQL Server and Oracle, we can only show performance in relative terms.

For initial experience collection for Neo, we always used the PostgreSQL optimizer as the expert. We define the *target system* as the system Neo is creating query plans for: that is, if Neo is building plans to execute on Oracle, we refer to Oracle as the target system. To train Neo, we first use the PostgreSQL optimizer to create a query plan for every query in the training set. We then measured the execution time of this plan on the targeted execution engine (e.g., Oracle) by forcing the target system, through query hints, to obey the proposed query plan. Next, we begin training: Neo trains a value network to predict the latency of the complete and partial plans in its experience set, and then uses that value network to build new query plans. These new query plans are then executed by the underlying DBMS, and their resulting latencies are added to Neo's experience. We repeat this process 100 times.

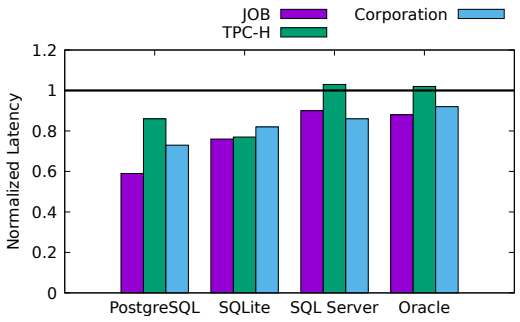


Figure 9: Latency of test query plans created by Neo after 100 episodes of training, normalized to plans created by the target system's corresponding optimizer for different workloads.

6.2 Overall Performance

Figure 9 shows the relative performance of Neo after 100 training iterations on each test workload, using the `R-Vector` encoding over the holdout dataset (lower is better). For example, with PostgreSQL and the `JOB` workload, Neo produces queries that take only 60% of average execution time than the ones created by the original PostgreSQL optimizer. Since the PostgreSQL optimizer is used to gather initial expertise for Neo, this demonstrates Neo's ability to improve upon an existing open-source optimizer.

Moreover, for SQL Server and the `JOB` and `Corp` workloads, the query plans produced by Neo are also 10% faster than the plans created by the SQL Server commercial optimizer (note that these plans are executed on SQL Server). Importantly, the SQL Server optimizer, which includes a multi-phase search procedure and a hundred-input dynamically-tuned cost model [15, 42], is expected to be substantially more advanced than PostgreSQL's optimizer. Yet, by bootstrapping only with PostgreSQL's optimizer, Neo is able to eventually outperform or match the performance of the SQL Server optimizer on its own platforms. Similar results were found for Oracle. Note that the faster execution times are solely based on better query plans (i.e., there are no modifications to the underlying execution engines). The only exception where Neo does not outperform the two commercial systems is for the TPC-H workload. We suspect that both SQL Server and Oracle have been tuned towards TPC-H, as it is one of the most common benchmarks.

Overall, this experiment demonstrates that *Neo is able to create plans, which are as good as, and sometimes even better than, open-source optimizers and their significantly superior commercial counterparts.* However, Figure 9 only compares the median performance of Neo after the 100th training episode. This naturally raises the following questions: (1) how does the performance compare with a fewer number of training episodes and how long does it take to train the model to a sufficient quality (answered in the next subsection), and (2) how robust is the optimizer to various estimation errors (answered in Section 6.4).

6.3 Convergence Time

To analyze the convergence time, we measured the performance after every training iteration, for a total of 100 complete iterations. We first report the learning curves in terms of training iterations to facilitate comparisons between different systems (e.g., a training episode with MS SQL Server might run much faster than PostgreSQL, simply because the MS SQL Server execution engine is better tuned). Afterwards, we report the wall-clock time to train the models on the different systems. Finally, we answer the question of how much our bootstrapping method helped with the training time.

6.3.1 Learning Curves

We measured the performance of Neo on each dataset with respect to the targeted system's optimizer (i.e., in each plot, a performance of 1 is equivalent to the target engine's optimizer) for every episode: a full pass over the set of training queries (retraining the network from the experience, choosing a plan for each training query, executing that plan, and adding the result to Neo's experience). Figure 10 depicts 50 runs: the solid line represents the median, and the shaded region represents the minimum and maximum values. For all DBMSes except for PostgreSQL, we additionally plot the relative performance of the plans generated by the PostgreSQL optimizer when executed on the target engine (e.g., executing the PostgreSQL plan on Oracle).

Convergence Each figure demonstrates a similar behavior: after the first iteration, Neo's performance is poor (nearly 2.5 times worse than the target system's optimizer). Then, for several iterations, the performance of Neo improves sharply, until it levels off. We note that Neo is able to improve on the PostgreSQL optimizer in as few as 9 training iterations (i.e., the number of training iterations until the median run crosses the line representing PostgreSQL). It is not surprising that matching the performance of a commercial optimizer (MS SQL Server or Oracle) requires significantly more training iterations, as commercial systems are much more sophisticated.

Variance The variance between the different training iterations is small for all workloads, except for TPC-H. We hypothesize that TPC-H's uniform data distribution renders the `R-Vector` embeddings less useful, and thus it takes the model longer to adjust accordingly. This behavior is not present in the non-synthetic datasets.

6.3.2 Wall-Clock Time

So far, we analyzed how long it took Neo to become competitive in terms of *training iterations*; next, we analyze the time it takes for Neo to become competitive in terms of *wall-clock time* (real time). We analyzed how long it took for Neo to reach two milestones: a policy producing query plans on-par with (1) the plans produced by PostgreSQL, but executed on the target execution engine, and (2) the plans produced by the target system's optimizer and executed on the target system's execution engine. The results are plotted in Figure 11a: the left and right bars represent milestone 1 and 2, respectively), split into time spent training the neural network and time spent executing queries. Note that the query execution step is parallelized, executing queries on different nodes simultaneously.

Unsurprisingly, it takes longer for Neo to become competitive with the more advanced, commercial optimizers. However, for every engine, learning a policy that outperforms the PostgreSQL optimizer consistently takes less than two hours. Furthermore, Neo was able to *match or exceed the performance of every optimizer within half a day*. Note that this time does not include the time for training the query encoding, which in the case of the `1-Hot` and `Histogram` are negligible. However, this takes longer for `R-Vector` (see Section 6.7).

6.3.3 Is Demonstration Even Necessary?

Since gathering demonstration data introduces additional complexity, it is natural to ask if demonstration is necessary at all: is it possible to learn a good policy from zero knowledge? While previous work [35] showed that an off-the-shelf deep reinforcement learning technique can learn to find query plans *that minimize a cost model* without demonstration data, learning a policy *based on query latency* (i.e., end-to-end) is difficult because a bad plan can take *hours* to execute. Unfortunately, randomly chosen query plans behave exceptionally poorly (i.e., 100x to 1000x worse [26]), potentially increasing the training time of Neo by a similar factor [36].

We attempted to work around this problem by selecting an ad-hoc query timeout t (e.g., 5 minutes), and terminating query executions when latencies exceed t. However, this technique destroys a good amount of the signal that Neo uses to learn: join patterns resulting in a latency of 7 minutes get the same reward as join patterns resulting in a latency of 1 week, and thus Neo cannot learn that the join patterns in the 7-minute plan are an improvement over the 1-week plan. As a result, even after training for over three weeks, we did not achieve results even on par with the PostgreSQL optimizer.

6.4 Robustness

Here, we test the efficacy of alternative query encoding (e.g., `1-Hot`), Neo's ability to handle unseen queries invented specifically to exhibit novel behavior, and Neo's resilience to noisy inputs.

6.4.1 Query Encoding

Figure 11b shows the performance of Neo across each DBMS for the `JOB` dataset, varying the query encoding. Here, we include two `R-Vector` encodings: partial denormalization [34], in which `R-Vector` are trained on a partially denormalized database, and a variant without any denormalization (suffixed with "no joins"). As expected, the `1-Hot` encoding consistently performs the worst, as the `1-Hot` encoding contains minimal information about predicates. The `Hist` encoding, while making naive uniformity assumptions, provides enough information about predicates to improve Neo's performance. In each case, the `R-Vector` encodings produce the best overall performance, with the "no joins" variant lagging slightly behind. We hypothesize that this is because the `R-Vector` encoding contains more semantic information about the underlying database than other encodings.

6.4.2 On Entirely New Queries

Previous experiments demonstrated Neo's ability to generalize to queries in a randomly-selected, held-out test set drawn from the same workload as the training set. While this shows that Neo can handle previously-unseen predicates and modifications to join graphs, it does not necessarily demonstrate that Neo will be able to generalize to a completely new query. To test Neo's behavior on new queries, we created a set of 24 additional queries, which we call `Ext-JOB` [2], that are semantically distinct from the original `JOB` workload (no shared predicates or join graphs).

After training Neo for 100 episodes on the `JOB` queries, we evaluated the performance of Neo on the `Ext-JOB` queries. Figure 12a shows the results: the height of the solid bar represents the average normalized latency of the plans produced Neo on the unseen queries. First, we note that with the `R-Vector` featurization, the execution plans chosen for the entirely-unseen queries in the `Ext-JOB` dataset still outperformed or matched the target system's optimizer. We hypothesize that the larger gap between the `R-Vector` featurizations and the `Hist`/ `1-Hot` featurizations is due to `R-Vector` capturing information about query predicates that generalizes to entirely new queries.

Learning new queries Since Neo is able to progressively learn from query executions, we evaluated Neo's performance on the `Ext-JOB` queries after 5 additional training iterations (which included experience from the `Ext-JOB` queries), depicted by the patterned bars in Figure 12a. Once Neo has seen each new query a handful of times, Neo's performance increases, having learned how to handle the new patterns introduced by the previously-unseen queries. While the performance of Neo initially degrades when confronted by new queries, Neo adapts to suit these new queries. This showcases the potential for a deep-learning powered query optimizer to keep up with changes in real-world query workloads.

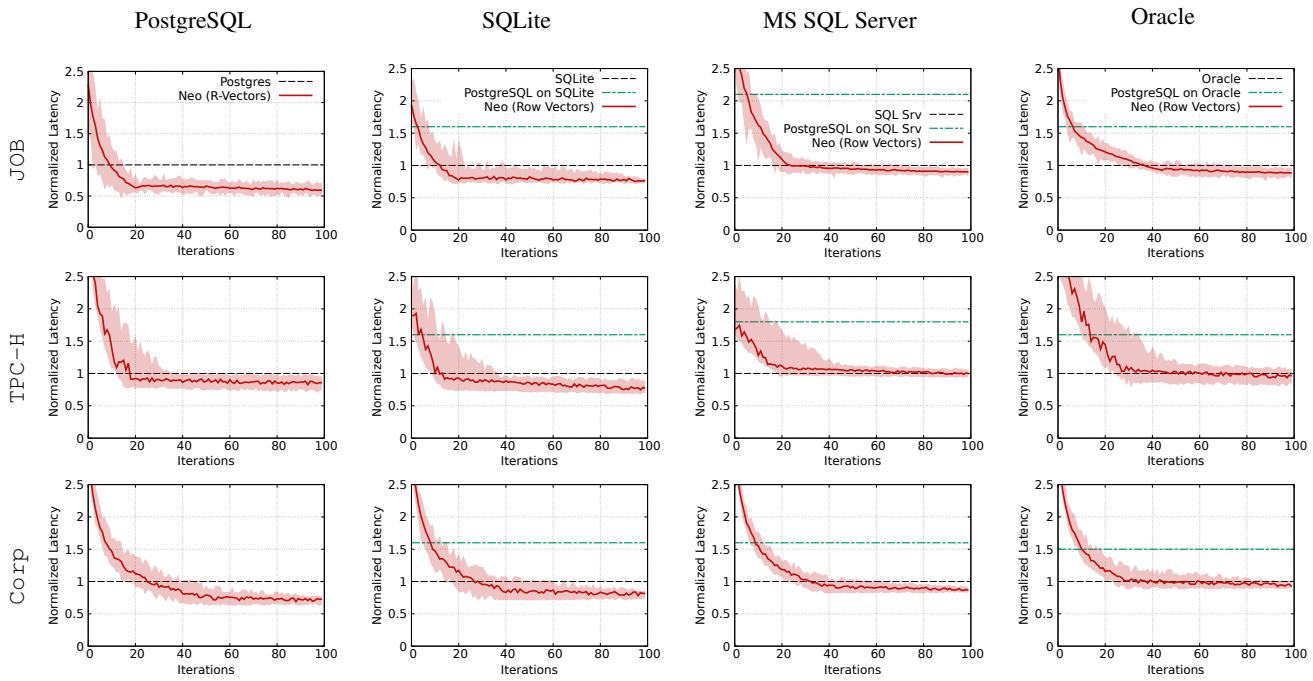


Figure 10: Learning curves (normalized latency over time) with variance. For each DBMS and dataset, we measure the latency of plans created by the DBMS' corresponding optimizer, the PostgreSQL optimizer, and Neo. Latencies are normalized to the latencies of the plans produced by the optimizer of the corresponding DBMS (e.g., all values in the fourth column are normalized to the latencies of the plans created by the Oracle optimizer). Shaded area spans minimum to maximum across fifty runs with different random seeds. Central line is the median. For a plot with all featurizations, please visit: `http://rm.cab/l/lc.pdf`

6.4.3 *Cardinality Estimates*

The strong relationship between cardinality estimation and query optimization is well-studied [6, 39]. However, effective query optimizers must take into account that most cardinality estimates tend to become significantly less accurate as the number of joins increases [26]. While deep neural networks are generally regarded as black boxes, here we show that Neo is capable of learning when to trust cardinality estimates and when to ignore them.

To measure the robustness of Neo to cardinality estimation errors, we trained two Neo models with an additional feature at each tree node. The first model received the PostgreSQL optimizer's cardinality estimation (PostgreSQL), and the second model received the true cardinality (True cardinality). We then plotted a histogram of both model's outputs across every state encountered while optimizing queries in the JOB workload when the number of joins was ≤ 3 and > 3, introducing artificial error.

Figure 13a and 13b shows the histogram of value network predictions for the PostgreSQL model for states with ≤ 3 or > 3 joins, respectively. Figure 13a shows that, when there are at most 3 joins, an increase in cardinality estimation error from zero orders of magnitude to two and five orders of magnitude causes an increase in the variance of the distribution: when the number of joins is at most 3, Neo learns a model that varies with the PostgreSQL cardinality estimate. However, in Figure 13b, we see that the distribution of network outputs hardly changes at all when the number of joins is greater than 3: when the number of joins is greater than 3, Neo learns to ignore the PostgreSQL cardinality estimates all together.

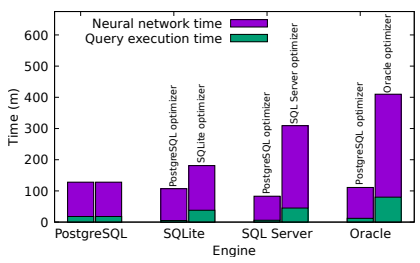
Figure 13c and 13d show that when Neo's value model is trained with true cardinalities as inputs, Neo learns a model that varies its prediction with the cardinality regardless of the number of joins. In other words, when provided with true cardinalities, Neo learns to rely on the cardinality information regardless of the number of joins. This demonstrates that Neo is capable of learning which input features are reliable, even when the reliability of those features is dependent on factors such as the number of joins.

6.4.4 *Per Query Performance*

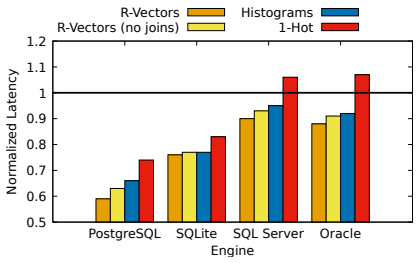
Next, we analyze Neo's performance at the query level. The absolute performance improvement (or regression) in seconds for each query in the JOB workload between the Neo and PostgreSQL plans (executed on PostgreSQL) are shown in Figure 14 (purple). While Neo improves the execution time of some queries, by up to 40 seconds, Neo also worsens the execution time of a few of queries (e.g., query 24a becomes 8.5 seconds slower).

In contrast to a traditional optimizer, Neo's optimization goal can easily be changed. So far, we always aimed to optimize the total workload cost, i.e., the total latency across all queries. However, we can also change the optimization goal to optimize for the *relative* improvement per query (green bars in Figure 14), as discussed in Section 4. This implicitly penalizes changes in the query performance from the baseline (e.g., PostgreSQL). When trained with this optimization goal, the total workload time is still accelerated (by 289 seconds, as opposed to nearly 500 seconds), *and* all but one query sees improved performance from the PostgreSQL baseline (29b regresses by 43 milliseconds). This provides evidence that Neo responds to different optimization goals, allowing it to be customized for different scenarios.

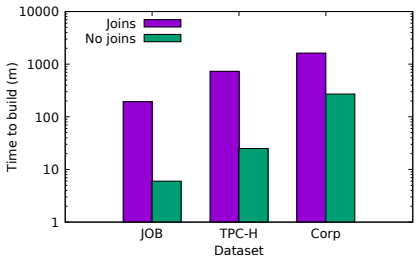
It is possible that Neo's loss function could be further customized to weigh queries differently depending on their importance to the user, i.e. query priority. It may also be possible to build an optimizer that is directly aware of service-level agreements (SLAs). We leave such investigations to future work.



(a) For each engine, training time for Neo to match the performance of the plans generated by the PostgreSQL optimizer and each engine's corresponding optimizer (identical for PostgreSQL). JOB.

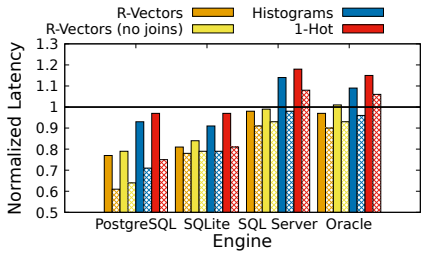


(b) Effect of different featurizations on latency of Neo's query plans after 100 training iterations (JOB dataset). Normalized to the latency of each DBMS' corresponding optimizer's query plans.

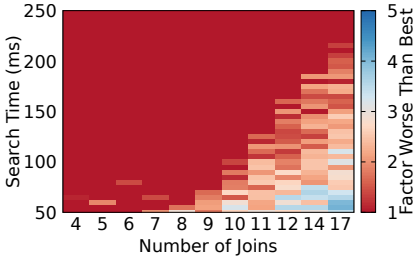


(c) Row vector training time for all three datasets. The "join" variant performs partial denormalization, which is included in the measured time. The "no join" variant performs no denormalization.

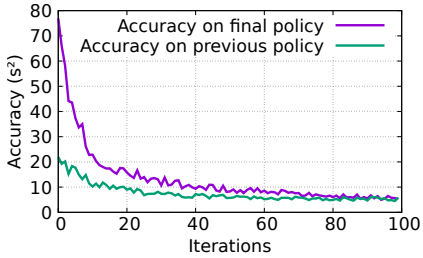
Figure 11



(a) After training on JOB, new queries (Ext-JOB) are introduced. Solid bars represent the average normalized latency of Neo's initial plans for the Ext-JOB queries. Patterned bars show the average normalized latency of Neo's plans after seeing Ext-JOB queries 5 times. Normalized to plans created by each DBMS' optimizer.

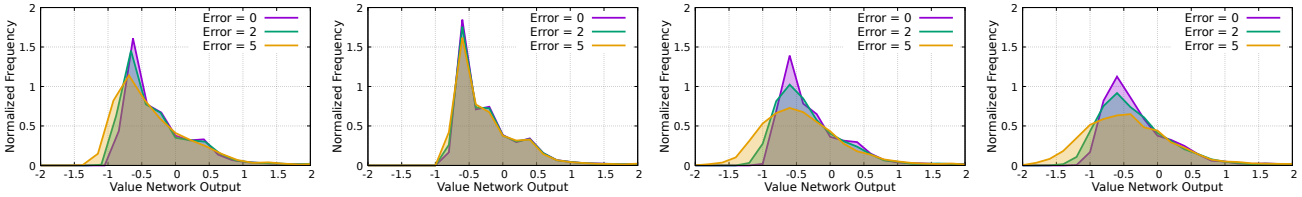


(b) After training, we plot the average query performance normalized to the best seen performance based on the number of joins in a query and the time spent in search. For queries with ≤ 9 joins, the best seen performance is achieved with only 100ms of search time. Queries with more joins needed up to 230ms of search time.



(c) Accuracy (MSE) of the value network per training iteration. Blue shows the accuracy of the value network at iteration i (x-axis) at predicting the latency of plans produced at iteration 100. Green shows the accuracy of the value network at iteration i at predicting the latency of plans produced at the previous iteration (iteration $i-1$).

Figure 12: Robustness and accuracy (all using Neo trained on JOB with PostgreSQL for 100 iterations)



(a) PostgreSQL, ≤ 3 joins (b) PostgreSQL, > 3 joins (c) True cardinality, ≤ 3 joins (d) True cardinality, > 3 joins

Figure 13: Histograms of Neo's normalized value network outputs for the JOB dataset when trained with PostgreSQL's cardinality estimates (Figure 13a and 13b) and with true cardinalities (Figure 13c and 13d). Each plot shows histograms representing artificially adding 0, 2, and 5 orders of magnitude of random noise (error) to the cardinality estimates given to Neo. For query plans with 3 or fewer joins (13a and 13c), Neo's predictions vary when error is added. However, for query plans with more than 3 joins (13b and 13d), the model trained with PostgreSQL's estimates (13b) shows significantly less variance than the model trained with true cardinalities (13d). This is evidence that Neo can learn to "trust" its inputs conditionally. Note that Neo does not receive explicit cardinality estimates in any other experiment.

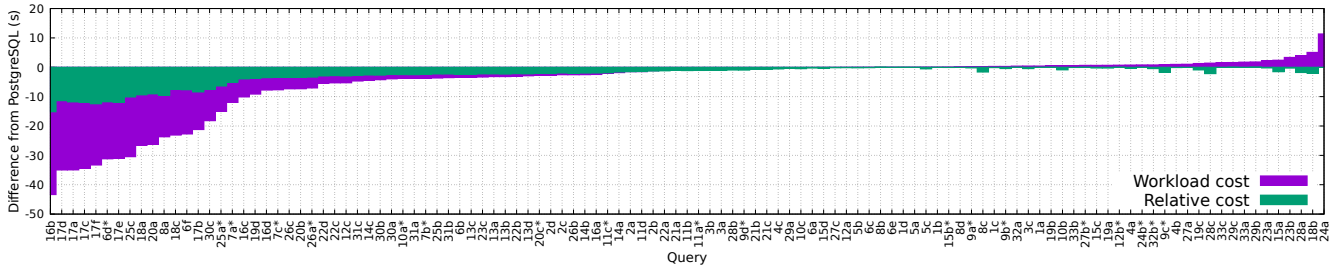


Figure 14: Absolute difference in time for Neo and PostgreSQL plans for each JOB query after training for 100 iterations with workload vs. relative cost functions (lower is better). Queries suffixed with $\star$ are part of the test set, and are never added to Neo's experience.

6.5 Value Network Accuracy

Neo's value network is responsible for accurately predicting the final latency of partial and complete query plans. We evaluated the value network's accuracy during training on the JOB dataset using PostgreSQL. After each iteration, we measured the mean squared error (MSE) of the value network's prediction vs. the true latencies of the plans produced (1) in the previous iteration and (2) in the final iteration. Figure 12c shows the results. Initially, the value network does a relatively poor job estimating the latencies of both the previous iteration and the final iteration. However, as training continues, the two curves converge. The convergence of the two curves – the value network's accuracy on the most recent iteration vs. the last iteration – is indicative that the policy is becoming *stable* [54], a desirable property that generally (but not nessecarily) correlates with decreased runtime variance.

6.6 Search

Neo uses the trained value network to search for query plans until a fixed-time cutoff (Section 4.2). Figure 12b shows how the performance of a query with a particular number of joins (selected randomly from the JOB dataset, executed on PostgreSQL) varies as the search time is changed. Note that the x-axis skips some values (the JOB dataset has no queries with 13 joins). Here, query performance is given relative to the best observed performance. For example, when the number of joins is 10, Neo found the best-observed plan whenever the cutoff time was greater than 120ms. We also tested significantly extending the search time (to 5 minutes), and found that such an extension did not change performance regardless of the number of joins in the query (up to 17 in the JOB dataset).

The relationship between the number of joins and sensitivity to search time is unsurprising: queries with more joins have a larger search space, and thus require more time to optimize. While 250ms to optimize a query with 17 joins is acceptable in many scenarios, other options [59] may be more desirable when this is not the case.

6.7 Row Vector Training Time

Neo builds its R-Vector encoding using the open source gensim package [47]. Figure 11c shows the time taken to train row vectors on each dataset, for both the "joins" (partially denormalized) and "no joins" (normalized) variants [34]. The time to train a R-Vector encoding is related to the size of the database. For the JOB dataset ($\approx$ 4GB), the "no joins" variant trains in under 10 minutes, whereas the "no joins" variant for the Corp dataset ($\approx$ 2TB) requires two hours to train. The "joins" variant takes significantly longer to train, e.g. three hours (JOB) to over a day (Corp).

Building row vectors may be prohibitive in some cases. However, compared to Hist, we found that the "joins" variant (on average) resulted in 5% faster query times and that the "no joins" variant (on average) resulted in 3% faster query times. Depending on the multiprocessing level, query arrival rate, etc., row vectors may "pay for themselves" quickly: for example, the training time for the "joins" variant on the Corp dataset is "paid for" after 540 hours of query processing, since the row vectors speed up query processing by 5% and require 27 hours to train. As the corporation constantly executes 8 queries simultaneously, this amounts to just three days. The "no joins" variant (improves performance by 3%, takes 217 minutes to train) is "paid for" after just 15 hours.

We do not analyze the behavior of row vectors on a changing database. It is possible that, depending on the database, row vectors quickly become "stale", or remain relevant for long periods of time. New techniques [13,61] suggest that retraining word vector models when the underlying data has changed can be done quickly, but we leave investigating these methods to future work.

7. RELATED WORK

Query optimization has been studied for more than forty years [11, 51]. Yet, query optimization is still an unsolved problem [30], especially due to the difficulty of accurately estimating cardinalities [26, 27]. The LEO optimizer was the first to introduce the idea of a query optimizer that learns from its mistakes [53]. In follow-up work, CORDS [19] proactively discovered correlations between any columns using data samples in advance of query execution.

Since proposed [60], deep learning is seeing traction in databases research. For example, recent work [20, 57] showed how to exploit reinforcement learning for Eddies-style, fine-grained adaptive query processing. The SkinnerDB system [56] shows how regret-bounded reinforcement learning can be applied to dynamically improve the execution of an individual query in an adaptive query processing system [56]. [43] used reinforcement learning to build state representations of traditional optimizers. [44] offered query-driven mixture models as an alternative to histograms and sampling for selectivity learning. [22, 28] proposed a deep learning approach to cardinality estimation, specifically designed to capture join-crossing correlations. Word2vec-style embeddings have been applied to data exploration [14] and error detection [17]. The closest works to ours are [25, 35], which proposed a learning based approach exclusively for join ordering, and only for a given cost model. The key contribution of Neo is that it provides an end-to-end, continuously learning solution to the database query optimization problem. Our solution does not rely on any hand-crafted cost model or data distribution assumptions.

This paper builds on recent progress from our own team. ReJOIN [35] proposed a deep reinforcement learning approach for join order enumeration, which was generalized into a broader vision in [36]. Decima [32] proposed a reinforcement learning-based scheduler, utilizing a graph neural network. SageDB [23, 24] laid out a vision towards building a new type of data processing system that makes heavy used of learned components. This paper is one of the first steps to realizing this overall vision.

8. CONCLUSIONS

This paper presents Neo, the first end-to-end learning optimizer that generates highly efficient query execution plans using deep neural networks. Neo iteratively improves its performance through a combination of reinforcement learning and a search strategy. On four database systems and three query datasets, Neo consistently outperforms or matches existing commercial query optimizers (e.g., Oracle's and Microsoft's) which have been tuned over decades.

In the future, we plan to investigate methods for generalizing a learned model to unseen schemas (using e.g. transfer learning [8]). We are interested in measuring the performance of Neo when bootstrapping from both more primitive and advanced commercial optimizers. Critically, Neo ignores many pieces of database state that are critical to achieving optimal query performance: cache state, concurrent queries, other applications on the same server, etc. While traditional optimizers tend to ignore these factors as well, Neo lays a foundation for building query optimizers that *automatically* adapt to such external factors – doing so may only require finding appropriate ways of encoding these factors as inputs to the value network, or may require significantly more research.

9. ACKNOWLEDGMENTS

This research is supported by Google, Intel, and Microsoft as part of the MIT Data Systems and AI Lab (DSAIL), NSF IIS 1815701, NSF IIS Career Award 1253196, and an Amazon Research Award. We also thank Tim Mattson (Intel) for his valuable feedback.

10. REFERENCES

[1] Embedding tools, https://github.com/parimarjan/db-embedding-tools.

[2] Ext-JOB queries, https://git.io/extended_job.

[3] PostgreSQL database, http://www.postgresql.org/.

[4] T. Anthony, Z. Tian, and D. Barber. Thinking Fast and Slow with Deep Learning and Tree Search. In *Advances in Neural Information Processing Systems 30*, NIPS '17, pages 5366–5376, 2017.

[5] J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer Normalization. *arXiv:1607.06450 [cs, stat]*, July 2016.

[6] B. Babcock and S. Chaudhuri. Towards a Robust Query Optimizer: A Principled and Practical Approach. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 119–130, New York, NY, USA, 2005. ACM.

[7] R. Bellman. A Markovian Decision Process. *Indiana University Mathematics Journal*, 6(4):679–684, 1957.

[8] Y. Bengio. Deep Learning of Representations for Unsupervised and Transfer Learning. In *Proceedings of ICML Workshop on Unsupervised and Transfer Learning*, ICML WUTL '12, pages 17–36, June 2012.

[9] R. Bordawekar and O. Shmueli. Using Word Embedding to Enable Semantic Queries in Relational Databases. In *Proceedings of the 1st Workshop on Data Management for End-to-End Machine Learning (DEEM)*, DEEM '17, pages 5:1–5:4, 2017.

[10] P. P. Brahma, D. Wu, and Y. She. Why Deep Learning Works: A Manifold Disentanglement Perspective. *IEEE Transactions on Neural Networks and Learning Systems*, 27(10):1997–2008, Oct. 2016.

[11] S. Chaudhuri. An Overview of Query Optimization in Relational Systems. In *ACM SIGMOD Symposium on Principles of Database Systems*, SIGMOD '98, pages 34–43, 1998.

[12] R. Dechter and J. Pearl. Generalized Best-first Search Strategies and the Optimality of A*. *J. ACM*, 32(3):505–536, July 1985.

[13] M. Faruqui, J. Dodge, S. K. Jauhar, C. Dyer, E. H. Hovy, and N. A. Smith. Retrofitting Word Vectors to Semantic Lexicons. In *The 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, NAACL '15, pages 1606–1615, 2015.

[14] R. C. Fernandez and S. Madden. Termite: A System for Tunneling Through Heterogeneous Data. In *AIDM @ SIGMOD 2019*, aiDM '19, 2019.

[15] L. Giakoumakis and C. A. Galindo-Legaria. Testing SQL Server's Query Optimizer: Challenges, Techniques and Experiences. *IEEE Data Eng. Bull.*, 31:36–43, 2008.

[16] X. Glorot, A. Bordes, and Y. Bengio. Deep Sparse Rectifier Neural Networks. In G. Gordon, D. Dunson, and M. Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *PMLR '11*, pages 315–323, Fort Lauderdale, FL, USA, Apr. 2011. PMLR.

[17] A. Heidari, J. McGrath, I. F. Ilyas, and T. Rekatsinas. HoloDetect: Few-Shot Learning for Error Detection. *arXiv:1904.02285 [cs]*, Apr. 2019.

[18] T. Hester, M. Vecerik, O. Pietquin, M. Lanctot, T. Schaul, B. Piot, D. Horgan, J. Quan, A. Sendonaris, G. Dulac-Arnold, I. Osband, J. Agapiou, J. Z. Leibo, and A. Gruslys. Deep Q-learning from Demonstrations. In *Thirty-Second AAAI Conference on Artifical Intelligence*, AAAI '18, New Orleans, Apr. 2017. IEEE.

[19] I. F. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga. CORDS: Automatic Discovery of Correlations and Soft Functional Dependencies. In *ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 647–658, 2004.

[20] T. Kaftan, M. Balazinska, A. Cheung, and J. Gehrke. Cuttlefish: A Lightweight Primitive for Adaptive Query Processing. *arXiv preprint*, Feb. 2018.

[21] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. In *3rd International Conference for Learning Representations*, ICLR '15, San Diego, CA, 2015.

[22] A. Kipf, T. Kipf, B. Radke, V. Leis, P. Boncz, and A. Kemper. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *9th Biennial Conference on Innovative Data Systems Research*, CIDR '19, 2019.

[23] T. Kraska, M. Alizadeh, A. Beutel, Ed Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. SageDB: A Learned Database System. In *9th Biennial Conference on Innovative Data Systems Research*, CIDR '19, 2019.

[24] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 489–504, New York, NY, USA, 2018. ACM.

[25] S. Krishnan, Z. Yang, K. Goldberg, J. Hellerstein, and I. Stoica. Learning to Optimize Join Queries With Deep Reinforcement Learning. *arXiv:1808.03196 [cs]*, Aug. 2018.

[26] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How Good Are Query Optimizers, Really? *PVLDB*, 9(3):204–215, 2015.

[27] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *The VLDB Journal*, pages 1–26, Sept. 2017.

[28] H. Liu, M. Xu, Z. Yu, V. Corvinelli, and C. Zuzarte. Cardinality Estimation Using Neural Networks. In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*, CASCON '15, pages 53–59, Riverton, NJ, USA, 2015. IBM Corp.

[29] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi. A survey of deep neural network architectures and their applications. *Neurocomputing*, 234:11–26, Apr. 2017.

[30] G. Lohman. Is Query Optimization a '"Solved" Problem? In *ACM SIGMOD Blog*, ACM Blog '14, 2014.

[31] J. Long, E. Shelhamer, and T. Darrell. Fully Convolutional Networks for Semantic Segmentation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, CVPR '15, June 2015.

[32] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh. Learning Scheduling Algorithms for Data Processing Clusters. *arXiv:1810.01963 [cs, stat]*, 2018.

[33] G. Marcus. Innateness, AlphaZero, and Artificial Intelligence. *arXiv:1801.05667 [cs]*, Jan. 2018.

[34] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. Neo: Towards A Learned Query Optimizer. *arXiv:1904.03711 [cs]*, Apr. 2019.

[35] R. Marcus and O. Papaemmanouil. Deep Reinforcement Learning for Join Order Enumeration. In *First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, aiDM '18, Houston, TX, 2018.

[36] R. Marcus and O. Papaemmanouil. Towards a Hands-Free Query Optimizer through Deep Learning. In *9th Biennial Conference on Innovative Data Systems Research*, CIDR '19, 2019.

[37] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient Estimation of Word Representations in Vector Space. *arXiv:1301.3781 [cs]*, Jan. 2013.

[38] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, and G. Ostrovski. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[39] G. Moerkotte, T. Neumann, and G. Steidl. Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors. *PVLDB*, 2(1):982–993, 2009.

[40] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI '16, pages 1287–1293, Phoenix, Arizona, 2016. AAAI Press.

[41] S. Mudgal, H. Li, T. Rekatsinas, A. Doan, Y. Park, G. Krishnan, R. Deep, E. Arcaute, and V. Raghavendra. Deep Learning for Entity Matching: A Design Space Exploration. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 19–34, New York, NY, USA, 2018. ACM.

[42] B. Nevarez. *Inside the SQL Server Query Optimizer*. Red Gate books, Mar. 2011.

[43] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi. Learning State Representations for Query Optimization with Deep Reinforcement Learning. In *2nd Workshop on Data Managmeent for End-to-End Machine Learning*, DEEM '18, 2018.

[44] Y. Park, S. Zhong, and B. Mozafari. QuickSel: Quick Selectivity Learning with Mixture Models. *arXiv:1812.10568 [cs]*, Dec. 2018.

[45] M. Poess and C. Floyd. New TPC Benchmarks for Decision Support and Web Commerce. *SIGMOD Records*, 29(4):64–71, Dec. 2000.

[46] A. G. Read. DeWitt clauses: Can we protect purchasers without hurting Microsoft. *Rev. Litig.*, 25:387, 2006.

[47] R. Řehůřek and P. Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, LREC '10, pages 45–50. ELRA, May 2010.

[48] S. Schaal. Learning from Demonstration. In *Proceedings of the 9th International Conference on Neural Information Processing Systems*, NIPS'96, pages 1040–1046, Cambridge, MA, USA, 1996. MIT Press.

[49] M. Schaarschmidt, A. Kuhnle, B. Ellis, K. Fricke, F. Gessert, and E. Yoneki. LIFT: Reinforcement Learning in Computer Systems by Learning From Demonstrations. *arXiv:1808.07903 [cs, stat]*, Aug. 2018.

[50] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, Jan. 2015.

[51] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In J. Mylopolous and M. Brodie, editors, *SIGMOD '89*, SIGMOD '89, pages 511–522, San Francisco (CA), 1989. Morgan Kaufmann.

[52] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, Jan. 2016.

[53] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *VLDB*, VLDB '01, pages 19–28, 2001.

[54] R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[55] N. Tran, A. Lamb, L. Shrinivas, S. Bodagala, and J. Dave. The Vertica Query Optimizer: The case for specialized query optimizers. In *2014 IEEE 30th International Conference on Data Engineering*, ICDE '14, pages 1108–1119, Mar. 2014.

[56] I. Trummer, S. Moseley, D. Maram, S. Jo, and J. Antonakakis. SkinnerDB: Regret-bounded Query Evaluation via Reinforcement Learning. *PVLDB*, 11(12):2074–2077, 2018.

[57] K. Tzoumas, T. Sellis, and C. Jensen. A Reinforcement Learning Approach for Adaptive Query Processing. *Technical Reports*, June 2008.

[58] L. van der Maaten and G. Hinton. Visualizing Data using t-SNE. *Journal of Machine Learning Research*, 9(Nov):2579–2605, 2008.

[59] F. Waas and A. Pellenkoft. Join Order Selection (Good Enough Is Easy). In *Advances in Databases*, BNCD '00, pages 51–67. Springer, Berlin, Heidelberg, July 2000.

[60] W. Wang, M. Zhang, G. Chen, H. V. Jagadish, B. C. Ooi, and K.-L. Tan. Database Meets Deep Learning: Challenges and Opportunities. *SIGMOD Rec.*, 45(2):17–22, Sept. 2016.

[61] L. Yu, J. Wang, K. R. Lai, and X. Zhang. Refining Word Embeddings Using Intensity Scores for Sentiment Analysis. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 26(3):671–681, Mar. 2018.

[62] J.-Y. Zhu, R. Zhang, D. Pathak, T. Darrell, A. A. Efros, O. Wang, and E. Shechtman. Toward Multimodal Image-to-Image Translation. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, NIPS '17, pages 465–476. Curran Associates, Inc., 2017.

[63] S. Zilberstein. Using Anytime Algorithms in Intelligent Systems. *AI Magazine*, 17(3):73–73, Mar. 1996.