Learning to Optimize Join Queries With Deep Reinforcement Learning

Sanjay Krishnan[1,2], Zongheng Yang[1], Ken Goldberg[1], Joseph M. Hellerstein[1], Ion Stoica[1]
[1]RISELab, UC Berkeley [2]Computer Science, University of Chicago
skr@cs.uchicago.edu {zongheng, goldberg, hellerstein, istoica}@berkeley.edu

ABSTRACT

Exhaustive enumeration of all possible join orders is often avoided, and most optimizers leverage heuristics to prune the search space. The design and implementation of heuristics are well-understood when the cost model is roughly linear, and we find that these heuristics can be significantly suboptimal when there are non-linearities in cost. Ideally, instead of a fixed heuristic, we would want a strategy to guide the search space in a more data-driven way— tailoring the search to a specific dataset and query workload. Recent work in deep reinforcement learning (Deep RL) may provide a new perspective on this problem. Deep RL poses sequential problems, like join optimization, as a series of 1-step prediction problems that can be learned from data. We present our deep RL-based DQ optimizer, which currently optimizes select-project-join blocks, and we evaluate DQ on the Join Order Benchmark. We found that DQ achieves plan costs within a factor of 2 of the optimal solution on all cost models and improves on the next best heuristic by up to 3×. Furthermore, DQ executes 10,000× faster than exhaustive enumeration and more than 10× faster than left/right-deep enumeration on the largest queries in the benchmark.

1. INTRODUCTION

Query optimization has been studied in database research and practice for almost 40 years [31]. The algorithmic problem of join optimization is a core component of almost all query optimizers, and new algorithmic techniques for reducing planning latency and scaling to larger queries remains an active area of research [29,36]. The classic problem is, of course, NP-hard, and practical algorithms leverage *heuristics* to make the search for a good plan efficient.

The design and implementation of heuristics are well-understood when the cost model is roughly linear, i.e., the cost of a join is linear in the size of its input relations. This assumption underpins many classical techniques as well as recent work [21,29,31,36]. However, many practical systems have relevant non-linearities in join costs. For example, an intermediate result exceeding the available memory may trigger partitioning, or a relation may cross a size threshold that leads to a change in physical join implementation.

It is not difficult to construct reasonable scenarios where classical heuristics dramatically fail. Consider the query workload and dataset in the Join Order Benchmark [23]. A popular heuristic is pruning the search space to only include left-deep join orders. Prior work showed that left-deep plans are extremely effective on this benchmark for cost models that prefer index joins [23]. Experimentally, we found this

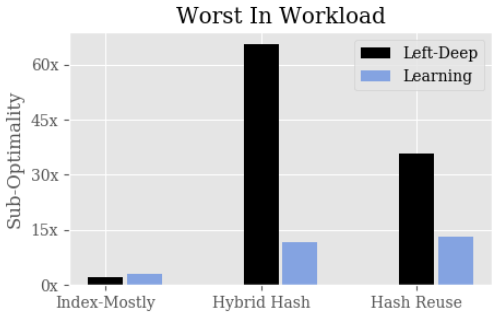

Figure 1: We consider 3 cost models for the Join Order Benchmark: (1) one with inexpensive index lookups, (2) one where the only physical operator is a hybrid hash join with limited memory, and (3) one that allows for the reuse of previously built hash tables. The figure plots the suboptimality of left-deep planning for each. The classical left-deep dynamic program fails on the latter two scenarios. We propose a reinforcement learning based optimizer, DQ, which can adapt to a specific cost model given appropriate training data.

to be true as well: the worst-case cost over the entire workload is only 2× higher than the true optimum (for an exponentially smaller search space). However, when we simply change the cost model to be more non-linear, consisting of (1) hybrid hash join operators that partition data when it exceeds limited available memory, or (2) hash join operators that can re-use previously built hash tables, suddenly the left-deep is no longer as effective—almost 50× more costly than the true optimum (Figure 1).

These results illustrate that practically, the search problem is unforgiving: different heuristics have weak spots where they fail by orders of magnitude relative to optimal. Success on such atypical or non-linear cost models may require searching over "bushy" plans, not just left-deep ones. With new hardware innovations [4] and a move towards serverless RDBMS architectures [1], it is not unreasonable to expect a multitude of new query cost models that significantly differ from existing literature; which might require a complete redesign of standard pruning heuristics. Ideally, instead of a fixed heuristic, we would want a strategy to guide the search space in a more data-driven way—tailoring the search to a specific dataset, query workload, and observed join costs. This sets up the main premise of the paper: would it be possible to use data-driven machine learning methods to identify such a heuristic from data?

The main insight of this paper is that *join optimization is a sequential learning problem* and there is an elegant way to integrate classical join optimization algorithms with statis-

tical machine learning techniques. To understand how, let us consider the hypothetical problem of exhaustive enumeration including "bushy" join trees and Cartesian products. The principle of optimality leads to a dynamic programming algorithm that incrementally builds a plan from optimal subplans. The algorithm re-uses previously enumerated subplans through exact memoization in a lookup table, and uses this table to construct a sequence of 1-step optimal decisions. In the abstract, the algorithm has a data structure that summarizes previous enumerations and makes a future decision using this data structure. We can cast this as a generalized prediction problem: given the costs of previously enumerated subplans, which 1-step decision is most likely optimal? The implications of this formulation is that to reach optimality, we may not have to exhaustively enumerate the plan space if costs of certain subplans can be extrapolated.

Algorithmically, this is the problem statement of the Artificial Intelligence field of reinforcement learning (RL) [34]. RL algorithms use sampling and statistical machine learning to estimate the long-term benefit of decisions. Concretely, this corresponds to setting up a regression problem between the decision to join a particular pair of relations and the observed benefit of making that decision in past data. Since this regression problem can be very non-linear in query optimization, we need a sufficiently rich class of functions, like neural networks, to parameterize the regression model. With this formulation, we can apply standard RL algorithms, similar to those used to play Atari games [28] or the game of Go [32], to join optimization. This formulation is particularly compelling if a single model can capture predictions for an entire workload, thus sharing learned experiences across planning instances.

We believe that (deep) reinforcement learning provides a new algorithmic perspective for thinking about join enumeration, and particularly a data-driven way of thinking about enumeration heuristics. Rather than the standard tunable parameters of a query optimizer, we now have to control what training data the model sees and how that data is featurized. The algorithm makes few assumptions about the structure of the cost model or the topology of the search space. Our deep RL-based optimizer, DQ, is built on Apache Calcite [2]. We show DQ optimizes plans well across many different cost models for a relatively modest set of training queries. Figure 1 illustrates that DQ significantly reduces the worst-case performance of a workload by adapting to the structure of the problem.

Our evaluation uses the recently proposed Join Order Benchmark (JOB) [23]. Its 33 templates and 113 queries in total contain between 4 and 15 relations, with an average of 8 relations per query. To summarize the results, we constructed three different cost models and evaluated 6 heuristic baselines. We found that DQ achieves plan costs within a factor of 2 of the optimal solution on all cost models. On the two cost models with significant non-linearities DQ improves on the next best heuristic by 1.7× and 3×. DQ executes 10,000× faster than exhaustive enumeration, over 1,000× faster than zig-zag tree enumeration, and more than 10× faster than left/right deep enumeration on the largest queries in the benchmark. We also evaluate executing the optimized plans on a real PostgreSQL database for a selected set of queries.

In summary, this paper makes the following contributions:

- We formulate join optimization as a Markov Decision Process, and we propose a deep reinforcement learning solution.
- We build a deep RL-based optimizer, DQ, with a flexible architecture allowing for tunable featurization and data collection schemes.
- We conduct evaluation against a range of classical heuristic optimizers and find DQ competitive in terms of plan quality and latency.

2. BACKGROUND

Traditional dynamic programs re-use previously computed results through memoization. In contrast, Reinforcement Learning (RL) represents the information from previously computed results with a learned model. We apply an RL model with a particular structure, a regression problem that associates the downstream value (future cumulative cost) with a decision (a join of two relations for a given query). Training this model requires observing join orders and their costs sampled from a workload. By projecting the effects of a decision into the future, the model allows us to dramatically prune the search space without exhaustive enumeration.

2.1 Problem Setting

We make the classical assumption of searching for a query plan made up of binary join operators and unary selections, projections, and access methods. We present our method for conjunctions of binary predicates and foreign key equi-joins.

We will use the following database of three relations denoting employee salaries as a running example throughout the paper:

Emp($id, name, rank$) Pos($rank, title, code$) Sal($code, amount$)

Consider the following join query:

```
SELECT *
 FROM Emp, Pos, Sal
 WHERE Emp.rank = Pos.rank AND
 Pos.code = Sal.code
```

There are many possible orderings to execute this query. For example, one could execute the example query as $Emp \bowtie (Sal \bowtie Pos)$, or as $Sal \bowtie (Emp \bowtie Pos)$.

2.2 Introduction to Reinforcement Learning

Bellman's "Principle of Optimality" and the characterization of dynamic programming is one of the most important results in computing [8]. It has a deep connection to a class of stochastic processes called Markov Decision Processes (MDPs), which formalize a wide range of problems from path planning to scheduling. In an MDP model, an agent makes a sequence of decisions with the goal of optimizing a given objective (e.g., improve performance, accuracy). Each decision is dependent on the current state, and typically leads to a new state. The process is "Markovian" in the sense that the system's current state completely determines its future progression. Formally, an MDP consists of a five-tuple:

$$\langle S, A, P(s,a), R(s,a), s_0 \rangle$$

where S describes a set of states that the system can be in, A describes the set of actions the agent can take, $s' \sim P(s,a)$

describes a probability distribution over new states given a current state and action, and s_0 defines a distribution of initial states. $R(s, a)$ is the reward of taking action a in state s. The reward measures the performance of the agent. The objective of an MDP is to find a decision policy $\pi : S \mapsto A$, a function that maps states to actions, with the maximum expected reward:

$$\begin{aligned} \arg\max_{\pi} \quad & \mathbf{E}\left[\sum_{t=0}^{T-1} R(s_t, a_t)\right] \\ \text{subject to} \quad & s_{t+1} = P(s_t, a_t), a_t = \pi(s_t). \end{aligned}$$

As with dynamic programming in combinatorial problems, most MDPs are difficult to solve exactly. Note that the greedy solution, eagerly maximizing the reward at each step, might be suboptimal in the long run. Generally, analytical solutions to such problems scale poorly in the time horizon.

Reinforcement learning (RL) is a class of stochastic optimization techniques for MDPs [34]. An RL algorithm uses sampling, taking randomized sequences of decisions, to build a model that correlates decisions with improvements in the optimization objective (cumulative reward). The extent to which the model is allowed to extrapolate depends on how the model is parameterized. One can parameterize the model with a table (i.e., exact parameterization) or one can use any function approximator (e.g., linear functions, nearest neighbors, or neural networks). Using a neural network in conjunction with RL, or Deep RL, is the key technique behind recent results like learning how to autonomously play Atari games [28] and the game of Go [32].

2.3 Markov Model of Enumeration

To pose the construction of a query plan tree as an MDP problem, it will be useful in subsequent discussion to view join nesting as a sequence of *graph contractions*. We briefly introduce the formalism here.

DEFINITION 1 (QUERY GRAPH). *Let G define an undirected graph called the query graph, where each relation R is a vertex and each join predicate ρ defines an edge between vertices. The number of connected components of G are denoted by κ_G.*

Each possible join is equivalent to a combinatorial operation called a graph contraction.

DEFINITION 2 (CONTRACTION). *Let $G = (V, E)$ be a query graph with V defining the set of relations and E defining the edges from the join predicates. A contraction c is a function of the graph parameterized by a pair of vertices $c = (v_i, v_j)$. Applying c to the graph G defines a new graph with the following properties: (1) v_i and v_j are removed from V, (2) a new vertex $(v_i + v_j)$ is added to V, and (3) the edges of $(v_i + v_j)$ are the union of the edges incident to v_i and v_j.*

Each contraction reduces the number of vertices by 1. Each plan can be described as a sequence of such contractions $c_1 \circ c_2 ... \circ c_T$ until $|V| = \kappa_G$. Going back to our running example, suppose we start with a query graph consisting of the vertices (Emp, Pos, Sal). Let the first contraction be $c_1 = (Emp, Pos)$; this leads to a query graph where the new vertices are $(Emp + Pos, Sal)$. Applying the only remaining possible contraction, we arrive at a single remaining vertex $Sal + (Emp + Pos)$ corresponding to the join plan $Sal \bowtie (Emp \bowtie Pos)$.

Symbol	*Definition*
G	A query graph. This is a *state* in the MDP.
c	A join (contraction on the graph). This is an *action* in the MDP.
G'	The resultant query graph after applying a join.
$J(c)$	A cost model that scores joins.

Table 1: Notation used throughout the paper.

Hereafter, we will use the term *join* as synonymous with *contraction*. The join optimization problem is to find the best possible join sequence—i.e., the best query plan. Also note that this model can be simply extended to capture physical operator selection as well. The set of allowed joins can be typed with an eligible join type, e.g., $c = (v_i, v_j, \mathsf{HashJoin})$ or $c = (v_i, v_j, \mathsf{IndexJoin})$.

We assume access to a cost model J, a function that estimates the incremental cost of a particular join, i.e., $J(c) \mapsto \mathbb{R}_+$.

PROBLEM 1 (JOIN OPTIMIZATION PROBLEM). *Let G define a query graph and J define a cost model. Find a sequence $c_1 \circ c_2 ... \circ c_T$ terminating in $|V| = \kappa_G$ to minimize:*

$$\begin{aligned} \min_{c_1,\ldots,c_T} \quad & \sum_{i=1}^{T} J(c_i) \\ \textit{subject to} \quad & G_{i+1} = c(G_i). \end{aligned}$$

Note how this problem statement exactly defines an MDP (albeit by convention a minimization problem rather than maximization). G is a representation of the **state**, c is a representation of the **action**, the graph contraction process defines $P(G, c)$, and the reward function is the negative cost $-J$. The output of an MDP is a function that maps a given query graph to the best next join.

Before proceeding, we summarize notation that will be used throughout the remainder of our paper in Table 1.

2.4 Long Term Reward of a Join

To introduce how RL gives us a new perspective on this classical database optimization problem, let us first examine the greedy solution. A naive solution is to optimize each c_i independently (also called Greedy Operator Optimization in recent literature [29]). The algorithm proceeds as follows: (1) start with the query graph, (2) find the lowest cost join, (3) update the query graph and repeat until only one vertex is left.

The greedy algorithm, of course, does not consider how local decisions might affect future costs. For illustration, consider our running example query with the following simple costs (assume a single join method with symmetric cost):

$$J(EP) = 100,\ J(SP) = 90,\ J((EP)S) = 10,\ J((SP)E) = 50$$

The greedy solution would result in a cost of 140 (because it neglects the future effects of a decision), while the optimal solution has a cost of 110. However, there is an upside: this greedy algorithm has a computational complexity of $O(|V|^3)$, despite the super-exponential search space.

The greedy solution is suboptimal because the decision at each index fails to consider the long-term value of its action. One might have to sacrifice a short term benefit for

a long term payoff. Consider the optimization problem for a particular query graph G:

$$V(G) = \min_{c_1,\ldots,c_T} \sum_{i=1}^{T} J(c_i) \quad (1)$$

In classical treatments of dynamic programming, this function is termed the *value function*. It is noted that optimal behavior over an entire decision horizon implies optimal behavior from any starting index $t > 1$ as well, which is the basis for the idea of dynamic programming. So, $V(G)$ can then be defined recursively for any subsequent graph G' generated by future joins:

$$V(G) = \min_{c}\{\ J(c) + \cdot V(G')\ \} \quad (2)$$

We can rewrite this value recursion in the following form:

$$Q(G, c) = J(c) + \cdot V(G')$$

leading to the following recursive definition of the *Q-function* (or cost-to-go function):

$$Q(G, c) = J(c) + \min_{c'} Q(G', c') \quad (3)$$

Intuitively, the Q-function describes the long-term value of each join: the cumulative cost if we act optimally for all subsequent joins after the current join decision. Knowing Q is equivalent to solving the problem since local optimization $\min_{c'} Q(G', c')$ is sufficient to derive an optimal sequence of join decisions.

If we revisit the greedy algorithm, and revise it hypothetically as follows: (1) start with the query graph, (2) find the lowest *Q-value* join, (3) update the query graph and repeat, then this algorithm has the same computational complexity of $O(|V|^3)$ but is provably optimal. To sketch out our solution, we will use Deep RL to approximate a global Q-function (one that holds for all query graphs in a workload), which gives us a polynomial-time algorithm for join optimization.

2.5 Applying Reinforcement Learning

An important class of reinforcement learning algorithms, called Q-learning algorithms, allows us to approximate the Q-function from samples of data [34]. What if we could regress from features of (G, c) to the future cumulative cost based on a small number of observations? Practically, we can observe samples of decision sequences containing $(G, c, J(c), G')$ tuples, where G is the query graph, c is a particular join, $J(c)$ is the cost of the join, and G' is the resultant graph. Such a sequence can be extracted from any final join plan and evaluating the cost model on the subplans.

Let's further assume we have a parameterized model for the Q-function, Q_θ:

$$Q_\theta(f_G, f_c) \approx Q(G, c)$$

where f_G is a *feature vector* representing the query graph and f_c is a feature vector representing a particular join. θ is the model parameters that represent this function and is randomly initialized at the start. For each training tuple i, one can calculate the following label, or the "estimated" Q-value:

$$y_i = J(c) + \arg\max_{c'} Q_\theta(G', c')$$

The $\{y_i\}$ can then be used as labels in a regression problem. If Q were the true Q-function, then the following recurrence would hold:

$$Q(G, c) = J(c) + \arg\max_{c'} Q_\theta(G', c')$$

So, the learning process, or *Q-learning*, defines a loss measuring how close the previous equation is to a fixed-point at each iteration:

$$L(Q) = \sum_i \|y_i - Q_\theta(G, c)\|_2^2$$

Then parameters of the Q-function can be optimized with gradient descent until a fixed-point is reached by recalculating the labels at the next iteration after each gradient step. When the model being optimized is a deep neural network, this algorithm is called the Deep Q Network algorithm [28].

Q-learning yields two key benefits: (1) the search cost for a single query relative to traditional query optimization is radically reduced, since the algorithm has the time-complexity of greedy search, and (2) the parameterized model can potentially learn across queries that have "similar" but non-identical subplans. This is because the similarity between subplans are determined by the query graph and join featurizations, f_G and f_c; thus if they are designed in a sufficiently expressive way, then the neural network can be trained to extrapolate the Q-function estimates to an entire workload of queries.

3. LEARNING TO OPTIMIZE

To apply Q-learning to the join optimization MDP, we need two pieces: a featurized representation for the arguments (G and c) and a way of collecting training data. The model takes in a subplan G and predicts which next join is likely to be optimal.

3.1 Featurizing the Join Decision

Before we get into the details, we will give a brief motivation of how we should think about featurization in a problem like this. The features should be sufficiently rich that they capture all relevant information to predict the future cumulative cost of a join decision. This requires knowing what the overall query is requesting, the tables on the left side of the proposed join, and the tables on the right side of the proposed join. It also requires knowing how single table predicates affect cardinalities on either side of the join.

Participating Relations: The first step is to construct a set of features to represent which relations are participating in the query and in the particular join. Let A be the set of all attributes in the database (e.g., $\{Emp.id, Pos.rank, ..., Sal.code, Sal.amount\}$). Each relation *rel* (including intermediate join results) has a set of *visible attributes*, $A_{rel} \subseteq A$, the attributes present in the output. So every query graph G can be represented by its visible attributes $A_G \subseteq A$. Each join is a tuple of two relations (L, R) and we can get their visible attributes A_L and A_R. Each of the attribute sets A_G, A_L, A_R can then be represented with a *binary 1-hot encoding*: a value 1 in a slot indicates that particular attribute is present, otherwise 0 represents its absence. Using $\oplus$ to denote concatenation, we obtain the query graph features, $f_G = A_G$, and the join decision features, $f_c = A_L \oplus A_R$, and, finally, the overall featurization for a particular (G, c) tuple is simply $f_G \oplus f_c$.

```
SELECT *
 FROM Emp, Pos, Sal
 WHERE Emp.rank =
       Pos.rank
 AND Pos.code = Sal.code
```

(a) Example query

A_G = [E.id, E.name, E.rank, P.rank, P.title, P.code, S.code, S.amount]
= [1 1 1 1 1 1 1 1]

(b) Query graph featurization

A_L = [E.id, E.name, E.rank]
= [1 1 1 0 0 0 0 0]
A_R = [P.rank, P.title, P.code]
= [0 0 0 1 1 1 0 0]

(c) Features of $E \bowtie P$

A_L = [E.id, E.name, E.rank, P.rank, P.title, P.code]
= [1 1 1 1 1 1 0 0]
A_R = [S.code, S.amount]
= [0 0 0 0 0 0 1 1]

(d) Features of $(E \bowtie P) \bowtie S$

Figure 2: **A query and its corresponding featurizations (§3.1).** One-hot vectors encode the visible attributes in the query graph (A_G), the left side of a join (A_L), and the right side (A_R). Such encoding allows for featurizing both the query graph and a particular join. A partial join and a full join are shown. The example query covers all relations in the schema, so $A_G = A$.

Query:
<example query>
AND Emp.id > 200

Selectivity(Emp.id>200) = 0.2

$f_G = A_G$ = [E.id, E.name, $\cdots$]
= [1 1 1 1 1 1 1 1]
$\rightarrow$ [.2 1 1 1 1 1 1 1]

(a) Selectivity scaling in query graph features

Query:
<example query>

feat_vec(IndexJoin($E \bowtie P$))
$= A_L \oplus A_R \oplus$ [1 0]

feat_vec(HashJoin($E \bowtie P$))
$= A_L \oplus A_R \oplus$ [0 1]

(b) Concatenation of physical operators in join features

Figure 3: **Accounting for selections and physical operators.** Simple changes to the basic form of featurization are needed to support selections (left) and physical operators (right). For example, assuming a system that chooses between only IndexJoin and HashJoin, a 2-dimensional one-hot vector is concatenated to each join feature vector. Discussion in §3.1.

Figure 2 illustrates the featurization of our example query. The overall intuition behind this scheme is to use each column name as a feature, because it identifies the distribution of that column.

Selections: Selections can change said distribution, i.e., (col, sel-pred) is different than (col, TRUE). To handle single table predicates in the query, we have to tweak the feature representation. As with most classical optimizers, we assume that the optimizer eagerly applies selections and projections to each relation. Next, we leverage the table statistics present in most RDBMS. For each selection σ in a query we can obtain the selectivity δ_σ, which estimates the fraction of tuples present after applying the selection[1]. To account for selections in featurization, we simply scale the slot in f_G that the relation and attribute σ corresponds to, by δ_r. For instance, if selection Emp.id > 200 is estimated to have a selectivity of 0.2, then the Emp.id slot in f_G would be changed to 0.2. Figure 3a pictorially illustrates this scaling.

Physical Operator: The next piece is to featurize the choice of physical operator. This is straightforward: we add another one-hot vector that indicates from a fixed set of implementations the type of join used. Figure 3b shows an example.

Extensibility: In this paper, we focus only on the basic form of featurization described above and study foreign key equality joins. An ablation study as part of our evaluation (Table 9) shows that the pieces we settled on all contribute to good performance. That said, there is no architectural limitation in DQ that prevents it from utilizing other features. Any property believed to be relevant to join cost prediction can be added to our featurization scheme. For example, we can add an additional binary vector f_{ind} to indicate which attributes have indexes built. Likewise, physical properties like sort-orders can be handled by indicating which attributes are sorted in an operator's output. Hardware environment variables (e.g., available memory) can be added as scalars if deemed as important factors in determining the final best plan. Lastly, more complex join conditions such as inequality conditions can also be handled (§7).

[1]We consider selectivity estimation out of scope for this paper. See discussion in §4 and §6.

3.2 Generating the Training Data

DQ uses a multi-layer perceptron (MLP) neural network to represent the Q-function. It takes as input the final featurization for a (G, c) pair, $f_G \oplus f_c$. Empirically, we found that a two-layer MLP gave the best performance under a modest training time constraint. We implemented a standard DQN algorithm (§2.5) in DL4J, a popular deep learning framework in Java.

Now, we describe what kind of training data is necessary to learn a Q-function. In supervised regression, we collect data of the form `(feature, values)`. The learned function maps from feature to values. One can think of this as a *stateless* prediction, where the underlying prediction problem does not depend on some underlying process state. On the other hand, in the Q-learning setting, there is state. So we have to collect training data of the form `(state, decision, new state, cost)`. Therefore, a training dataset consists of tuples of the following format:

```
List<Graph, Join, Graph', Cost> dataset;
```

or `(G, c, G', J)` for short.

This defines a stochastic gradient descent iteration until convergence:

$$\theta^{(i+1)} \leftarrow \theta^{(i)} - \alpha \cdot [(J + \min_{c'} Q_\theta(G', c')) - Q_\theta(G, c)] \cdot \nabla_\theta Q_\theta(G, c)$$

In many cases like robotics or game-playing, RL is used in a live setting where the model is trained on-the-fly based on concrete moves chosen by the policy and measured in practice. Q-learning is known as an "off-policy" RL method. This means that it can be trained on experimental data from other problem instances, as long as the training data sufficiently covers the decisions to be made. Its training is independent of the data collection process and can be suboptimal—as long as the data collection process sufficiently covers relevant scenarios.

Such off-policy data can be acquired via a system's native optimizer and cost model. In fact, useful data is *automatically generated* as a consequence of running classical planning algorithms. For each join decision that the optimizer makes, we can get the incremental cost of the join. Furthermore, if we run a classical dynamic programming algorithm to optimize a k-way join, we not only get a final plan but also data from all of the samples along the final join path but every single subplan enumerated along the way. Any heuristic optimizer search can be used to generate this information: bushy trees, zig-zag trees, left-deep, or right-deep. Randomized algorithms such as QuickPick [38] can also be used.

In our experiments, we bootstrap planning with a bushy dynamic program until the number of relations in the join exceeds 10 relations. Then, the data generation algorithm switches to a greedy scheme for efficiency for the last $K-10$ joins. Ironically, the data collected from such an optimizer might be "too good" because it does not measure or learn from a diverse enough space of subplans. If the training data only consisted of optimal sub-plans, then the learned Q-function may not accurately learn the downside of poor subplans. Likewise, if the training purely sampled random plans, it might not see very many instances of good plans.

To encourage more "exploration", during data collection noise can be injected into the optimizer to force it to enumerate more diverse subplans. We control this via a parameter ϵ, the probability of picking a random join as opposed to a join with the lowest cost. As the algorithm enumerates subplans, if $\mathsf{rand}() < \epsilon$ then a random (valid) join is chosen on the current query graph; otherwise it proceeds with the lowest-cost join as usual. This is an established technique to address such "covariate shift", a phenomenon that has been extensively studied in prior work [22].

3.3 Execution after Training

After training, we obtain a parameterized estimate of the Q-function, $Q_\theta(f_G, f_c)$. For execution, we simply go back to the standard algorithm as in the greedy method but instead of using the local costs, we use the learned Q-function: (1) start with the query graph, (2) featurize each join, (3) find the join with the lowest **estimated Q-value** (i.e., output from the neural net), (4) update the query graph and repeat.

This algorithm has the time-complexity of greedy enumeration except in greedy, the cost model is evaluated at each iteration, and in our method, a neural network is evaluated. One pleasant consequence is that DQ exploits the abundant vectorization opportunities in numerical computation. In each iteration, instead of invoking the neural net sequentially on each join's feature vector, DQ *batches* all candidate joins (of this iteration) together, and invoke the neural net once on the batch. Modern CPUs, GPUs, and specialized accelerators (e.g., Tensor Processing Unit [19]) all offer optimized instructions for such single-instruction multiple-data (SIMD) workloads. The batching optimization amortizes each invocation's fixed overheads and has the most impact on large joins.

3.4 Feedback From Execution

A learned Q-function also allows us to leverage feedback from real query executions. Readers might be familiar with the concept of fine-tuning in the neural network literature [41], where a network is trained on one dataset and transferred to another with minimal retraining. We can apply a very similar principle to accounting for feedback.

The challenge is that when we execute a plan, we do not observe all of the intermediate costs without significant instrumentation of the database. We may only observe a final runtime. Executing subplans can be very expensive (queries can take several minutes to run). However, we can still leverage a final runtime to adjust plans that optimize runtime, while still leveraging what we have learned from the cost-model. One can think about the fine-tuning process as first using the cost model to learn relevant features about the structure of subplans (i.e., which ones are generally beneficial). After this is learned, those features are fed into a predictor to project the effect of that decision on final runtime.

The Q-function relates subplans to future projected costs. As with any cost model, the cumulative cost is a proxy for runtime. We can define a new cost function where the immediate cost of all joins are 0 but the cost of the final plan is its runtime. We can generate a fine-tuning dataset $\{x_i\}_1^N$ with:

$$x_i = \begin{cases} \texttt{(G, c, G', 0)} & \text{if not final} \\ \texttt{(G, c, G', runtime)} & \text{if final} \end{cases}$$

The training procedure first pre-trains the network on a large amount of samples from the optimizer's cost model (inexpensive compared to execution). Once it's trained to convergence, we freeze the weights of the first two layers. When data is collected from real execution, we re-initialize the last (output) layer and restart training on these data samples.

4. OPTIMIZER ARCHITECTURE

Selinger's optimizer design separated the problem of plan search from cost/selectivity estimation [31]. This insight allowed independent innovation on each topic over the years. In our initial work on using Deep Learning for query optimization, we follow this lead, and intentionally focused on learning a search strategy only. For cost estimation, we lean on traditional selectivity and cost estimation techniques. This allows us to study the effectiveness of Deep RL on optimizer search specifically, without confounding factors from cost estimation.

Even within the search problem, we focus narrowly on the classical select-project-join kernel. This is traditional in the literature, going back to Selinger [31] and continuing as recently as Neumann et al.'s very recent experimental work [29]. It is also particularly natural for illustrating the connection between dynamic programming and Deep RL and implications for query optimization. We intend for our approach to plug directly into a Selinger-based optimizer architecture like that of PostgreSQL, DB2 and many other systems. Our approach, DQ, is simply a learning-based replacement for prior algorithms for searching a plan space. Like any non-exhaustive query optimization technique, our results are heuristic. The new concerns raised by our approach have to do with limitations of training, including overfitting and avoiding high-variance plans. We use this section to describe the extensibility of our approach and what design choices the user has at her disposal.

4.1 Architecture and API

First, we describe the information that DQ requires to train, learn, and execute. The algorithm has a training phase in which it observes join costs and an execution phase where it evaluates the learned model. DQ makes relatively minimal assumptions about the structure of the optimizer. Below are the API hooks that we require implemented:

Workload Generation. A function that returns a list of training queries of interest. DQ requires a relevant workload for training. In our experiments, we show that this workload can be taken from query templates or sampled from the database schema.

```
sample(): List<Queries>
```

Cost Sampling. A function that given a query returns a list of join actions and their resultant costs. DQ requires the system to have its own optimizer to generate training data. This means generating feasible join plans and their associated costs. Our experiments evaluate integration with both deterministic enumeration algorithms (bushy and left-deep) and a randomized algorithm called QuickPick-1000.

```
train(query): List<Graph, Join, Graph', Cost>
```

Predicate Selectivity Estimation. A function that returns the selectivity of a particular single table predicate. DQ leverages the optimizer's own selectivity estimate for featurization (§3.1).

```
selectivity(predicate): Double
```

In our evaluation (§5.6), we will vary these exposed hooks to experiment with different implementations for each (e.g., comparing training on highly relevant data from a desired workload vs. randomly sampling join queries directly from the schema).

4.2 Implementation

Our prototype implementation is integrated with Apache Calcite [2]. Apache Calcite provides libraries for parsing SQL, representing relational algebraic expressions, and a Volcano-based query optimizer [15,16]. Calcite does not handle physical execution or storage and uses JDBC connectors to a variety of database engines and file formats. We implemented a package inside Calcite that allowed us to leverage its parsing and plan representation, but also augment it with more sophisticated cost models and optimization algorithms. We implemented an internal simulator to simulate cost estimates based on histograms, true cardinalities, and other approximations. All of our code is written in single-threaded Java.

5. EXPERIMENTS

To set up the experiments, we revisit a motivating claim from the introduction: the design and implementation of heuristics is well-understood when the cost model is linear and non-linearities can lead to significant suboptimality. The experiments intend to illustrate that DQ offers a form of *robustness to cost model*, meaning, that it prioritizes plans tailored to the structure of the cost model, workload, and physical design—even when these plans are bushy. We also report how DQ is affected by various extensible knobs it exposes, namely varying the relevance of the training data and how the data is collected.

Our evaluation uses the recently proposed "Join Order Benchmark" (JOB) [23]. This benchmark is derived from the Internet Movie Data Base (IMDB). It contains information about movies and related facts about actors, directors, production companies, etc. The dataset is 3.6 GB in size and consists of 21 relational tables. The largest table has 36 million rows. The benchmark contains 33 templates and 113 queries in total. The joins have between 4 and 15 relations, with an average of 8 relations per query. In our evaluation, we do not explicitly study the problems of selectivity/cost estimation or avoiding high-variance plans. To summarize the results, we found that:

- DQ achieves plan costs within a factor of 2 of exhaustive enumeration on all cost models (§5.3.1, §5.3.2, §5.3.3).
- On the two cost models with significant non-linearities DQ improves on the next best heuristic (zig-zag) by a factor of up to 3 (§5.3.2, §5.3.3).
- When planning the largest queries in JOB, DQ executes 10,000× faster than exhaustive enumeration, over 1,000× faster than zig-zag tree enumeration, and more than 10× faster than left/right-deep enumeration (§5.4). It outperforms all baselines on queries with 11 or more relations.

5.1 Robustness to Cost Model

In the first set of experiments, we show how the same query workload with different cost models can lead to very different performance in heuristics. The cost model described in Leis et al. [23] is inspired by an in-memory system. The cost of a hash join is linear in the size of the input relations, and the cost of an index join is essentially the cost of streaming the left side of the join. This cost model greatly rewards index-usage (by convention the right relation is used for index lookup), and left-deep strategies are very strong in this setting. One of the experimental conclusions is that there is little benefit of bushy plans. This leads us to a natural question—where are bushy plans useful?

Many systems have piecewise cost models with regimes determined by the size of a relation (see work on Parametric Query Optimization [17,35]). When the relation exceeds a size threshold, either because of additional partitioning due to memory constraints or because of a switch to a different physical operator, non-linear costs can be incurred. We argue that in these non-linear regimes classical heuristics experience significant difficulty. We found that size thresholds lead to a phenomenon that we call *packing*, where the optimal plans tend to be bushy to strategically create intermediate results with sizes under the thresholds. We illustrate these behaviors with three different cost models motivated by different underlying DBMS architectures.

CM1: In the first cost model (inspired by [23]), we model a main-memory database that performs two types of joins: index joins and in-memory hash joins. Let O describe the current operator, O_l be the left child operator, and O_r be the right child operator. The costs are defined with the following recursions:

$$\mathsf{c}_{ij}(O) = \mathsf{c}(O_l) + \mathsf{match}(O_l, O_r) \cdot |O_l|$$

$$\mathsf{c}_{hj}(O) = \mathsf{c}(O_l) + \mathsf{c}(O_r) + |O|$$

where c denotes the cost estimation function, $|\cdot|$ is the cardinality function, and match denotes the expected cost of

an index match, i.e., fraction of records that match the index lookup (always greater than 1) multiplied by a constant factor λ (we chose 1.0). We assume indexes on the primary keys. In this cost model, if an eligible index exists it is generally desirable to use it, since $\mathsf{match}(O_l, O_r) \cdot |O_l|$ rarely exceeds $\mathsf{c}(O_r) + |O|$ for foreign key joins. Even though the cost model is nominally "non-linear", primary tradeoff between the index join and hash join is due to index eligibility and not dependent on properties of the intermediate results. For the JOB workload, unless λ is set to be very high, hash joins have rare occurrences compared to index joins.

CM2: In the next cost model, we remove index eligibility from consideration and consider only hash joins and nested loop joins with a memory limit M. The model charges a cost when data requires additional partitioning, and further falls back to a nested loop join when the smallest table exceeds the squared memory:

$$\mathsf{c}_{join} = \begin{cases} \mathsf{c}(O_l) + \mathsf{c}(O_r) + |O| & \text{if } |O_r| + |O_l| \leq M \\ \mathsf{c}(O_l) + \mathsf{c}(O_r) + 2(|O_r| + |O_l|) + |O| & \text{if } \min(|O_r|, |O_l|) \leq M^2 \\ \mathsf{c}(O_l) + \mathsf{c}(O_r) + (|O_r| + \left\lceil \frac{|O_r|}{M} \right\rceil |O_l|) \end{cases}$$

The non-linearities in this model are size-dependent, so controlling the size of intermediate relations is important in the optimization problem. We set the memory limit M to 10^5 tuples in our experiments. This limit is low in real-world terms due to the small size of the benchmark data. However, we intend for the results to be illustrative of what happens in the optimization problems.

CM3: In the next cost model, we model a database that accounts for the reuse of already-built hash tables. We use the Gamma database convention where the left operator as the "build" operator and the right operator as the "probe" operator [14]. If the previous join has already built a hash table on an attribute of interest, then the hash join does not incur another cost.

$$\mathsf{c}_{nobuild} = \mathsf{c}(O_l) + \mathsf{c}(O_r) - |O_r| + |O|$$

We also allow for index joins as in CM1. This model makes hash joins substantially cheaper in cases where re-use is possible. This model favors some subplans to be right-deep plans which maximize the reuse of the built hash tables. Therefore, optimal solutions have both left-deep and right-deep segments.

In our implementation of these cost models, we use *true* cardinalities on single table predicates, and we leverage standard independence assumptions to construct more complicated cardinality estimates. The goal of this work is to evaluate the join ordering process independent of the strength or weakness of the underlying cardinality estimation.

5.2 Baseline Algorithms

We consider the following baseline algorithms. These algorithms are not meant to be a comprehensive list of heuristics but rather representative of a class of solutions.

1. Exhaustive (**EX**): This is a dynamic program that exhaustively enumerates all join plans avoiding Cartesian products.
2. left-deep (**LD**): This is a dynamic program that exhaustively enumerates all left-deep join plans.
3. Right-Deep (**RD**): This is a dynamic program that exhaustively enumerates all right-deep join plans.
4. Zig-Zag (**ZZ**): This is a dynamic program that exhaustively enumerates all zig-zag trees (every join has at least one base relation, either on the left or the right) [42].
5. IK-KBZ (**KBZ**): This algorithm is a polynomial time algorithm that decomposes the query graph into chains and orders the chains based on a linear approximation of the cost model [21].
6. QuickPick-1000 (**QP**): This algorithm randomly selects 1000 join plans and returns the best of them. 1000 was selected to be roughly equivalent to the planning latency of DQ [38].

5.3 Plan Cost

We now evaluate all of the baseline algorithms against DQ on the three cost models. We use the same cost model for optimization as well as scoring the final plans. All of the algorithms consider join ordering without Cartesian products, so **EX** is an optimal baseline. We report results in terms of the suboptimality w.r.t. **EX**, namely $cost_{algo}/cost_{\mathbf{EX}}$.

We present results on all 113 JOB queries. We train on 80 queries and test on 33 queries. We do 4-fold cross validation to ensure that every test query is excluded from the training set at least once. The performance of DQ is only evaluated on queries not seen in the training workload.

5.3.1 Cost Model 1

First, we consider optimizing CM1. These results reproduce the conclusions of [23], where left-deep plans are generally very good (utilize indexes very well) and there is little need for zigzag or exhaustive enumeration. DQ is competitive with these optimal solutions without *a priori* knowledge of the index structure. In fact, DQ significantly outperforms the other heuristic solutions **KBZ** and **QP**. While it is true that **KBZ** also restricts its search to left-deep plans, it is suboptimal for cyclic join graphs—its performance is hindered since almost all of the queries in JOB contain cycles. We found that **QP** struggles with the physical operator selection, and a significant number of random samples are required to find a very narrow set of good plans (ones the use indexes effectively).

	Min	Mean	Max
QP	1.0	23.87	405.04
KBZ	1.0	3.45	36.78
RD	4.70	53.25	683.35
LD	1.0	1.08	2.14
ZZ	1.0	1.07	1.87
EX	1.0	1.0	1.0
DQ	1.0	1.32	3.11

Table 2: Cost relative to optimal plan under CM1.

These results are not surprising: they show that DQ, a learning-based solution, reasonably matches performance on cases where good heuristics exist. On average DQ is within 22% of the LD solution and in the worst case it is only 1.45× worse than LD.

5.3.2 Cost Model 2

By simply changing the cost model, we can force the left-deep heuristics to perform poorly. CM2 accounts for disk

usage in hybrid hash joins. In this cost model, none of the heuristics match the exhaustive search over the entire workload. Since the the costs are largely symmetric for small relation sizes there is little benefit to either left-deep or right-deep pruning. Similarly zig-zag trees are only slightly better, and the heuristic methods fail by orders of magnitude on their worst queries.

	Min	Mean	Max
QP	7.43	51.84	416.18
KBZ	5.21	29.61	106.34
RD	1.93	8.21	89.15
LD	1.75	7.31	65.45
ZZ	1.0	5.07	43.16
EX	1.0	1.0	1.0
DQ	1.0	1.68	11.64

Table 3: Cost relative to optimal plan under CM2.

DQ still comes close to exhaustive enumeration. It does not perform as well as in the previous experiment (with its worst query about 12× the optimal cost) but on average it is still significantly better than the alternatives. Our experiments suggest that as the memory limit becomes so small that plans are affected, the heuristics begin to diverge from the optimal solution, as seen in Table 4 below.

	$M = 10^8$	$M = 10^6$	$M = 10^4$	$M = 10^2$
KBZ	1.0	3.31	30.64	41.64
LD	1.0	1.09	6.45	6.72
EX	1.0	1.0	1.0	1.0
DQ	1.04	1.42	1.64	1.56

Table 4: Mean relative cost vs. memory limit (# tuples in memory).

5.3.3 *Cost Model 3*

Finally, we illustrate results on the CM3 that allows for the reuse of hash tables. Right-deep plans are no longer inefficient in this model as they facilitate reuse of the hash table (note right and left are simply conventions and there is nothing important about the labels). The challenge is that now plans have to contain a mix of left-deep and right-deep structures. Zig-Zag tree pruning heuristic was exactly designed for cases like this. Surprisingly, DQ is actually significantly better than zig-zag enumeration. We observed that bushy plans were necessary in a small number of queries and DQ found lower cost solutions.

	Min	Mean	Max
QP	1.43	16.74	211.13
KBZ	2.21	14.61	96.14
RD	1.83	5.25	69.15
LD	1.35	4.21	35.91
ZZ	1.0	3.41	23.13
EX	1.0	1.0	1.0
DQ	1.0	1.91	13.14

Table 5: Cost relative to optimal plan under CM3.

In summary, these results suggest that DQ is robust against different cost model regimes, since it can learn to adapt to workloads.

5.4 Planning Latency

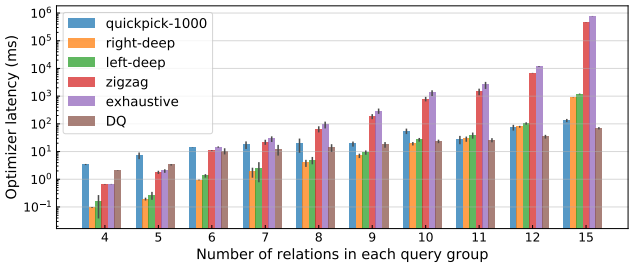

Figure 4: Optimizer latency on all JOB queries, grouped by the number of relations in the query graph. Error bars represent ± standard deviation around the mean; a total of 5 trials were run. Discussion in §5.4.

Next, we report the planning time of DQ and several other optimizers across the entire 113 JOB queries. The same model in DQ is used to plan all queries. Implementations are written in Java, single-threaded[2], and reasonably optimized at the algorithmic level (e.g., QuickPick would short-circuit a partial plan if it's already estimated to be more costly than the current best plan)—but no significant efforts are spent on low-level engineering (e.g., details such as bit twiddling tricks [37] or optimized JVM GC behavior). Hence, the relative magnitudes are more meaningful than the absolute values. Experiments were run on an AWS EC2 c5.9xlarge instance with a 3.0GHz Intel Xeon Platinum CPU and 72GB memory.

Figure 4 groups the run times by number of relations in each query graph. In the small-join regime, the overheads of DQ are mostly attributed to interfacing with DL4J (creating and filling the featurization buffers; JNI overheads due to native CPU backend execution). These could have been optimized away by targeting a non-JVM engine and/or GPUs, but we note that when the number of joins is small, exhaustive enumeration would be the ideal choice.

In the large-join regime, DQ achieves drastic speedups: for the largest joins DQ wins by up to 10,000× compared to exhaustive enumeration. The gain mostly comes from the batching optimization (§3.3), which upper-bounds the number of neural net invocations by the number of relations in a query. We believe this is a profound performance argument for such a learned optimizer—it would have an even more unfair advantage when applied to larger queries or executed on specialized accelerators [19].

5.4.1 *Training Overhead*

Of course, there is an overhead to collecting the data and training a model. Sampling training plans from **EX** (§3.2) on 80 queries took 30× more time than actually training the model (154 minutes and under 5 minutes on a laptop, respectively). We envision a real-world deployment strategy in which, data collection is a byproduct of a normal optimizer that runs in the background, and DQ acts an additional learning layer.

5.5 Micro-benchmarks

In the subsequent experiments, we try to characterize when DQ is expected to work and how efficiently.

5.5.1 *Difficulty of Handling Selections*

[2]To ensure fairness, for DQ we configure the underlying BLAS library to use 1 thread. No GPU is used.

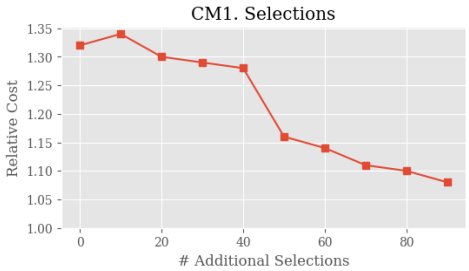

Figure 5: Accurately modeling the effects of single table predicates requires more training data. Generating a modest number of synthetic single table equality predicates can help. We plot the mean relative cost of DQ as a function of the number of synthetic predicates.

Next, we report how DQ's performance changes due to handling single-table selections. We run the 33 JOB query templates (1) with only the join predicates (i.e., ignoring the effects of single-table predicates) and (2) with selections included. Results are averaged over 5-fold cross-validation so all of the reported numbers are when the query is not seen during training. We report the min, mean, and max cost of the optimized plans relative to exhaustive enumeration.

Table 6 shows the results. It highlights an interesting difference between learning-based and enumeration-based optimization. Classical optimizers have an internal cost model that relates predicate selectivities to join costs. On the other hand, in DQ, this model is learned from data. Implicitly learning this relationship can be difficult for a neural network, if it has not observed a sufficient amount of distinct predicates spaced at a variety of selectivities. In Table 6, we show that if we ignore the effects of single table predicates, DQ actually comes close to matching exhaustive enumeration; when selections are handled, difficulties arise for a learning-based optimizer.

	CM1	CM2	CM3
No selections	1.07	1.13	1.01
With selections	1.32	1.68	1.91

Table 6: Effect of single table predicates on DQ's mean relative cost.

It is easy to generate a large dataset with a lot of different join orders, but generating random predicates on the tables can be more challenging. One approach to address this issue is synthetic data augmentation (a technique popularized by deep learning models in computer vision): we randomly sample equality conditions on each of the tables and include these synthetic queries in the training dataset. Specifically, we generate random equality predicates on the columns that have actual selections in the original workload; these predicates are added to the join queries. Figure 5 shows how the costs change when up to 100 random single-table selections are included. With more data, DQ can better attribute the effects of selections on join costs.

Selections are interesting because they have threshold effects on cost. Once a selectivity of a predicate on one of the relations drops beyond a certain point, the set of low-cost plans might drastically change. These results are both positive and negative. First, they illustrate that learning-based approaches can easily be made robust to different phenomena by manipulating the training data and/or the featurization. On the other hand, the system may not accurately be able to assess its own confidence on previously unseen plans. It might make predictions assuming smooth behavior without knowledge of a drastic change due to selectivity.

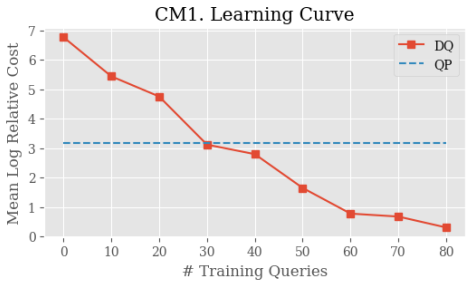

Figure 6: The mean relative cost (in log-scale) as a function of the number of training queries seen by DQ. We include QuickPick-1000 as a baseline.

5.6 Data Quantity/Relevance

We further evaluate the nature of training data required by DQ, in terms of raw quantity (number of training queries seen) and relevance (the relevance to the test workload).

5.6.1 *Quantity*

We consider the optimization with CM1. We vary the number of training queries given to DQ and plot the mean relative cost to optimal using the cross validation technique described before. Figure 6 illustrates the relationship. DQ requires about 30 training queries to match the performance of QuickPick-1000.

We found that this break-even point roughly corresponds to seeing all of the relations in a query at least once. In fact, one can train the model on small queries and test it on larger ones—as long as the relations are covered well. To investigate its generalization power, we trained DQ on all of the queries with $\leq$ 9 and 8 relations, respectively, and tested on the remaining queries (out of a total of 113). For comparison we include a baseline scheme of training on 80 random queries and testing on 33. Results (costs relative to optimal) are shown in Table 7.

	# Training Queries	Mean Relative Cost
Random	80	1.32
Train $\leq$ 9-way	82	1.61
Train $\leq$ 8-way	72	9.95

Table 7: DQ trained on small joins and tested on larger joins.

The results show that even when trained on subplans, DQ performs relatively well and generalizes to larger joins (recall, the workload contains up to 15-way joins). This indicates that DQ indeed learns *local structures*—efficient joining of small combinations of relations. When those local structures do not sufficiently cover the cases of interest during deployment, we see degraded performance.

5.6.2 *Relevance of Training Data*

Next, we consider the effect of different training data sampling schemes. We consider the optimization with single table predicates and use CM1. Figure 7 plots the performance of different data sampling techniques each with 80 training queries. The more relevant the training queries can be made

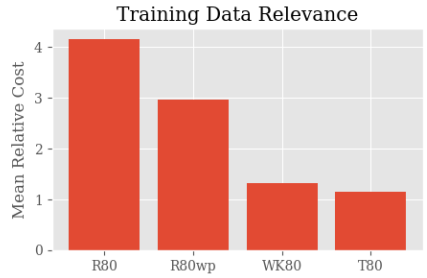

Figure 7: We plot the relative performance of DQ with different training workloads. R80 is a dataset sampled independent of the JOB workload with random joins from the schema and random equality predicates. R80wp has random joins as before but contains single table predicates from the workload. WK80 includes 80 actual queries sampled from the workload. T80 describes a scheme where each of the 33 query templates is covered at least once in sampling.

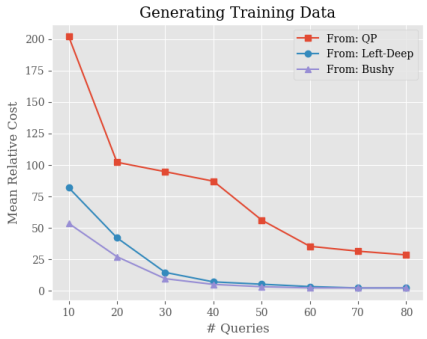

Figure 8: DQ trained on training data collected by QuickPick-1000 , left-deep, or the bushy optimizer. Data variety boosts convergence speed and final quality. We report the average relative cost w.r.t. **EX**.

towards the test workload, the less data is required for good performance.

It also shows that the performance does not completely suffer with random queries. DQ still achieves a lower cost compared to QuickPick-1000 even with random queries (4.16 vs. 23.87). This experiment illustrates that DQ does not actually require *a priori* knowledge of the workload.

5.6.3 *Training Data Generation*

Next, we evaluate using different baseline optimizers for collecting the training dataset. We consider the optimization with CM1. We collect a varying amount of data sampled from queries in the workload. We compare using a QuickPick-1000 optimizer, left-deep optimizer, and the bushy optimizer described earlier (Figure 8). All methods allow DQ to quickly converge to good solutions. The dynamic programming-based methods, left-deep and bushy, converge faster as they produce subplans as well as final plans in the training data. In contrast, the QuickPick optimizer yields only 1000 random full plans per query. The subplan structures offer variety valuable for training, and they better cover the space of different relation combinations that might be seen in testing.

5.6.4 *Model Choices*

In theory, with enough data, DQ should approach optimal performance. The natural next question is where the gap comes from in practice—is it the neural network's ability to represent the cost function (model-limited), or is it due to a lack of training data (data-limited)? We ran an experiment on the first 20 training queries. We intentionally feed DQ the same queries during training and testing, with the goal of measuring its ability to memorize training data. We train the model to convergence, then measure the percentage of 20 queries where the learned optimizer exactly matched the cost from the optimal optimizer.

	CM1	CM2	CM3
With Predicates	90%	75%	90%
Without Predicates	100%	80%	95%

Table 8: % of queries where DQ exactly matches the optimal optimizer.

We see that there is some inherent imprecision in the neural network, even if the exact testing query has been seen before. We find that this has less to do with the linearity or non-linearity of the cost function, but rather how "close" good plans are in terms of cost. If there are a lot of roughly similar good plans the neural network may not have enough precision to differentiate between them (despite its vast representational capacity). It is not simply an issue of undertraining: we found that increasing the number of epochs from 10k to 100k had little effect on the final training error. This does raise the question about our modeling choices. Table 9 reports an ablation study of the featurization described earlier (§3.1):

	Graph Features	Sel. Scaling	Loss
No Predicates	No	No	0.087
	Yes	No	0.049
	Yes	Yes	0.049
Predicates	No	No	0.071
	Yes	No	0.051
	Yes	Yes	0.020

Table 9: Feature ablation.

Without features derived from the query graph (Figure 2b) and selectivity scaling (Figure 3a) the training loss is 3.5× more. These results suggest that all of the different features contribute positively for performance.

5.6.5 *Sensitivity to Training Data*

Classically, join optimization algorithms have been deterministic. Except for **QP**, all of our baselines are deterministic as well. Randomness in DQ (besides floating-point computations) stems from what training data is seen. We run an experiment where we provide DQ with 5 different training datasets and evaluate on a set of 20 hold-out queries. We report the max range (worst factor over optimal minus best factor over optimal) in performance over all 20 queries in Table 10. For comparison, we do the same with **QP** over 5 trial samples.

	CM1	CM2	CM3
QP	2.11×	1.71×	3.44×
DQ	1.59×	1.13×	2.01×

Table 10: Plan variance over trials.

We found that while the performance of DQ does vary due

to training data, the variance is relatively low. Even if we were to account for this worst case, DQ would still be competitive in our macro-benchmarks. It is also substantially lower than that of **QP**, a true randomized algorithm.

5.6.6 *Sensitivity to Faulty Cardinalities*

In general, the cardinality/selectivity estimates computed by the underlying RDMS do not have up-to-date accuracy. All query optimizers, to varying degrees, are exposed to this issue since using faulty estimates during optimization may yield plans that are in fact suboptimal. It is therefore worthwhile to investigate this sensitivity and try to answer, "is the neural network more or less sensitive than classical dynamic programs and heuristics?"

In this microbenchmark, the optimizers are fed *perturbed* base relation cardinalities (explained below) during optimization; after the optimized plans are produced, they are scored by an *oracle* cost model. This means, in particular, DQ only sees noisy relation cardinalities during training and is tested on true cardinalities. The workload consists of 20 queries randomly chosen out of all JOB queries; the join sizes range from 6 to 11 relations. The final costs reported below are the average from 4-fold cross validation.

The perturbation of base relation cardinalities works as follows. We pick N random relations, the true cardinality of each is multiplied by a factor drawn uniformly from $\{2, 4, 8, 16\}$. As N increases, the estimate noisiness increases (errors in the leaf operators get propagated upstream in a compounding fashion). Table 11 reports the final costs with respect to estimate noisiness.

	$N = 0$	$N = 2$	$N = 4$	$N = 8$
KBZ	6.33	6.35	6.35	5.85
LD	5.51	5.53	5.53	5.60
EX	5.51	5.53	5.53	5.60
DQ	5.68	5.70	5.96	5.68

Table 11: Costs ($\log_{10}$) when N relations have perturbed cardinalities.

Observe that, despite a slight degradation in the $N = 4$ execution, DQ is not any more sensitive than the **KBZ** heuristic. It closely imitates exhaustive enumeration—an expected behavior since its training data comes from **EX**'s plans computed with the faulty estimates.

5.7 Real Execution

Lastly, we use our Apache Calcite connector to execute the learned plans on a real Postgres database. We force join plans constructed by our optimizer suite by setting join_collapse_limit = 1 in the database and we also set enable_material = false to disable any intra-query materialization. We load the IMDB data into Postgres and create primary key indexes. We assume a cost model that uses index joins when available and nested loop joins otherwise. All experiments here were run on an EC2 t2.xlarge instance with PostgreSQL 9.2.

Figure 9 illustrates the results on 4 queries from JOB. In this physical design, left-deep plans (KBZ, ZZ, LD) are preferred and are almost as effective as exhaustive plans. RD plans are predictably slow as they don't take advantage of the indexes. QP plans are good only when they by chance sample a left-deep strategy. Learning is competitive with the good plans in all of the real queries.

Additionally, we evaluate fine-tuning the network based on past execution data (§3.4). We execute 50 random join plans and collect data from their execution; the network is then fine-tuned (bars denoted as "DQF"). On two of the queries (10a, 21a), we see significant speedups of up to 2×; in the other two, there was no statistically significant change due to fine-tuning.

6. RELATED WORK

Applications of machine learning in database internals is still the subject of significant debate this year and will continue to be a contentious question for years to come [7,20,24,27]. An important question is what problems are amenable to machine learning and AI solutions. We believe that query optimization is one such sub-area. The problems considered are generally hard and orders of magnitude of performance are at stake. In this setting, poor learning solutions will lead to slow but not incorrect execution, so correctness is not a concern.

Cost Function Learning Admittedly, we are not the first to consider "learning" in the query optimizer and there are a number of alternative architectures that one may consider. The precursors to this work are attempts to correct query optimizers through execution feedback. One of the seminal works in this area is the LEO optimizer [25]. This optimizer uses feedback from the execution of queries to correct inaccuracies in its cost model. The underlying cost model is based on histograms. The basic idea inspired several other important works such as [10]. The sentiment in this research still holds true today; when Leis et al. extensively evaluated the efficacy of different query optimization strategies they noted that feedback and cost estimation errors are still challenges in query optimizers [23]. A natural first place to include machine learning would be what we call *Cost Function Learning*, where statistical learning techniques are used to correct or replace existing cost models. This is very related to the problem of performance estimation of queries [3,39,40].

We actually started with this model, where a neural network learns to predict the selectivity of a single relation predicate. Results were successful, albeit very expensive from a data perspective. To estimate selectivity on an attribute with 10k distinct values, the training set had to include 1000 queries. This architecture suffers from the problem of *featurization of literals*; the results are heavily dependent on learning structure in literal values from the database that are not always straightforward to featurize. This can be especially challenging for strings or other non-numerical data types. A recent workshop paper does show some promising results in using Deep RL to construct a good feature representation of subqueries but it still requires > 10k queries to train [30].

Adaptive Query Optimization Adaptive query processing [5,12] as well as the related techniques to re-optimize queries during execution [6,26] is another line of work that we think is relevant to the discussion. Reinforcement learning studies sequential problems and adaptive query optimization is a sequential decision problem over tuples rather than subplans. We focus our study on optimization in fixed databases and the adaptivity that DQ offers is at a workload level. Continuously updating a neural network can be challenging for very fine-grained adaptivity, e.g., processing different tuples in different ways.

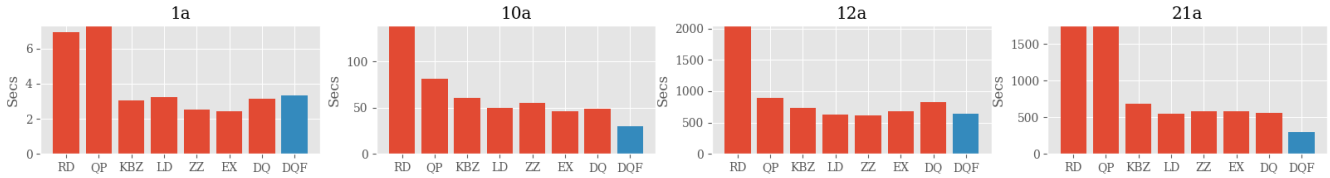

Figure 9: We execute the plans on a PostgreSQL database and measure the runtimes. DQF denotes a learned plan with 50 queries of fine-tuning. Query 1a, 10a, 12a, and 21a from JOB are shown. Runtimes that greatly exceeded the best optimizers are clipped.

Robustness There are a couple of branches of work that study robustness to different parameters in query optimization. In particular, the field of "parametric query optimization" [17,35], studies the optimization of piecewise linear cost models. The interesting part about DQ is it is agnostic to this structure. It learns a heuristic from data identifying different regimes where different classes of plans work. We hope to continue experiments and attempt to interpret how DQ is partitioning the feature space into decisions. There is also a deep link between this work and least expected cost (LEC) query optimization [11]. Markov Decision Processes (the main abstraction in RL) are by definition stochastic and optimize the LEC objective.

Join Optimization At Scale Scaling up join optimization has been an important problem for several decades, most recently [29]. At scale, several randomized approaches can be applied. There is a long history of randomized algorithms (e.g., the QuickPick algorithm [38]) and genetic algorithms [9,33]. These algorithms are pragmatic and it is often the case that commercial optimizers will leverage such a method after the number of tables grows beyond a certain point. The challenge with these methods is that their efficacy is hard to judge. We found that QuickPick often varied in performance on the same query quite dramatically.

Another heuristic approach is relaxation, or solve the problem exactly under simplified assumptions. One straightforward approach is to simply consider greedy search avoiding Cartesian products [13], which is also the premise of the IK-KBZ algorithms [18,21]. Similar linearization arguments were also made in recent work [29,36]. Existing heuristics do not handle all types of non-linearities well, and this is exactly the situation where learning can help. Interestingly enough, our proposed technique has a $O(n^3)$ runtime, which is similar to the *linearizedDP* algorithm described in [29].

7. EXTENSIONS

This work is meant to be an initial study of Deep Reinforcement Learning in the context of join optimization. We list additional features that are easy to implement but felt they were out of scope for this paper.

Sort Orders: We do not consider cost models for sorting, sort-merge joins, or interesting orders. Keeping track of interesting sort orders in subplans is actually not difficult in our framework. They can be tracked as a set of additional features.

Non-Foreign Key Inner Joins: We presented our method with a featurization designed for inner joins over foreign key relations. It is relatively straightforward to extend this model to join conditions composed of conjunctions of binary expressions. Assume the maximum number of expressions in the conjunction is capped at $\mathcal{N}$. As before, let A be the set of all attributes in the database. Each expression has two attributes and an operator. As with featurizing the vertices we can 1-hot encode the attributes present. We additionally have to 1-hot encode the binary operators $\{=, \neq, <, >\}$. For each of the expressions in the conjunctive predicate, we concatenate the binary feature vectors that have its operator and attributes. Since the maximum number of expressions in the conjunction capped at $\mathcal{N}$, we can get a fixed sized feature vector for all predicates.

Re-training: It is relatively straightforward to periodically retrain the model as new data is collected to adjust to changing tables and physical properties. Model training takes a matter of minutes. We avoided this discussion as evaluating this property requires a realistic benchmark that shows data growing over time and the queries evolving accordingly.

8. DISCUSSION

Even today many database systems favor "exact" solutions when the number of relations are small but switch to approximations after a certain point. The algorithmic connection between reinforcement learning and classical join optimization algorithms can provide a new perspective on join optimization, with a polynomial time algorithm that can approach optimal performance given the right set of training data and features. We studied the search problem factored away from other concerns in query optimization. However, this is not a fundamental architectural decision. It is popular in recent AI research to try "end-to-end" learning, where problems that were traditional factored into subproblems (e.g., self-driving cars involve separate models for localization, obstacle detection and lane-following) are learned in a single unified model. One can imagine a similar architectural ambition for an end-to-end learning query optimizer, which simply maps subplan features to measured runtimes. This would require a significant corpus of runtime data to learn from, and changes to the featurization and perhaps the deep network structure we used here. But the fundamental architecture would be quite similar. Exploring the extremes of learning and query optimization in future work may shed insights on pragmatic middle grounds.

As illustrated by the Cascades optimizer [15] and follow-on work, cost-based dynamic programming—whether bottom up or top-down with memoization—need not be restricted to select-project-join blocks. Most query optimizations can be recast into a space of algebraic transformations amenable to dynamic programming, including asymmetric operators like outer joins, cross-block optimizations including order optimizations and "sideways information passing", and even non-relational operators like PIVOT. Of course this blows

up the search space, and new heuristic decisions come into play for search. Large spaces are ideal for solutions like the one we proposed.

9. REFERENCES

[1] Amazon Aurora Serverless. https://aws.amazon.com/rds/aurora/serverless/.

[2] Apache Calcite. https://calcite.apache.org/.

[3] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, and S. B. Zdonik. Learning-based query performance modeling and prediction. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 390–401. IEEE, 2012.

[4] J. Arulraj and A. Pavlo. How to build a non-volatile memory database management system. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1753–1758. ACM, 2017.

[5] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *ACM sigmod record*, volume 29, pages 261–272. ACM, 2000.

[6] S. Babu, P. Bizarro, and D. DeWitt. Proactive re-optimization. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 107–118. ACM, 2005.

[7] P. Bailis, K. S. Tai, P. Thaker, and M. Zaharia. Don't throw out your algorithms book just yet: Classical data structures that can outperform learned indexes. https://dawn.cs.stanford.edu/2018/01/11/index-baselines/, 2017.

[8] R. Bellman. *Dynamic programming*. Princeton University Press, 1957.

[9] K. Bennett, M. C. Ferris, and Y. E. Ioannidis. *A genetic algorithm for database query optimization*. Computer Sciences Department, University of Wisconsin, Center for Parallel Optimization, 1991.

[10] S. Chaudhuri, V. Narasayya, and R. Ramamurthy. A pay-as-you-go framework for query execution feedback. *Proceedings of the VLDB Endowment*, 1(1):1141–1152, 2008.

[11] F. Chu, J. Halpern, and J. Gehrke. Least expected cost query optimization: what can we expect? In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 293–302. ACM, 2002.

[12] A. Deshpande, Z. Ives, V. Raman, et al. Adaptive query processing. *Foundations and Trends® in Databases*, 1(1):1–140, 2007.

[13] L. Fegaras. A new heuristic for optimizing large queries. In *International Conference on Database and Expert Systems Applications*, pages 726–735. Springer, 1998.

[14] R. H. Gerber. Data-flow query processing using multiprocessor hash-partitioned algorithms. Technical report, Wisconsin Univ., Madison (USA), 1986.

[15] G. Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[16] G. Graefe and W. McKenna. The volcano optimizer generator. Technical report, COLORADO UNIV AT BOULDER DEPT OF COMPUTER SCIENCE, 1991.

[17] A. Hulgeri and S. Sudarshan. Parametric query optimization for linear and piecewise linear cost functions. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 167–178. VLDB Endowment, 2002.

[18] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM Transactions on Database Systems (TODS)*, 9(3):482–502, 1984.

[19] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pages 1–12. IEEE, 2017.

[20] 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*, pages 489–504. ACM, 2018.

[21] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *VLDB*, volume 86, pages 128–137, 1986.

[22] M. Laskey, J. Lee, R. Fox, A. Dragan, and K. Goldberg. Dart: Noise injection for robust imitation learning. *Conference on Robot Learning 2017*, 2017.

[23] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, 2015.

[24] L. Ma, D. Van Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data*, pages 631–645. ACM, 2018.

[25] V. Markl, G. M. Lohman, and V. Raman. Leo: An autonomic query optimizer for db2. *IBM Systems Journal*, 42(1):98–106, 2003.

[26] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic. Robust query processing through progressive optimization. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 659–670. ACM, 2004.

[27] M. Mitzenmacher. A model for learned bloom filters and related structures. *arXiv preprint arXiv:1802.00884*, 2018.

[28] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. In *Nature*. Nature Research, 2015.

[29] T. Neumann and B. Radke. Adaptive optimization of very large join queries. In *Proceedings of the 2018 International Conference on Management of Data*, pages 677–692. ACM, 2018.

[30] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi. Learning state representations for query optimization with deep reinforcement learning. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*, DEEM'18, pages 4:1–4:4, New York, NY, USA, 2018. ACM.

[31] 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 *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34. ACM, 1979.

[32] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. In *Nature*. Nature Research, 2016.

[33] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and randomized optimization for the join ordering problem. *The VLDB JournalâĂŤThe International Journal on Very Large Data Bases*, 6(3):191–208, 1997.

[34] R. S. Sutton, A. G. Barto, et al. *Reinforcement learning: An introduction*. MIT press, 1998.

[35] I. Trummer and C. Koch. Multi-objective parametric query optimization. *Proceedings of the VLDB Endowment*, 8(3):221–232, 2014.

[36] I. Trummer and C. Koch. Solving the join ordering problem via mixed integer linear programming. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1025–1040. ACM, 2017.

[37] B. Vance and D. Maier. Rapid bushy join-order optimization with cartesian products. In *ACM SIGMOD Record*, volume 25, pages 35–46. ACM, 1996.

[38] F. Waas and A. Pellenkoft. Join order selection (good enough is easy). In *British National Conference on*

Databases, pages 51–67. Springer, 2000.

[39] W. Wu, Y. Chi, H. Hacígümüş, and J. F. Naughton. Towards predicting query execution time for concurrent and dynamic database workloads. *Proceedings of the VLDB Endowment*, 6(10):925–936, 2013.

[40] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacigümüs, and J. F. Naughton. Predicting query execution time: Are optimizer cost models really unusable? In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 1081–1092. IEEE, 2013.

[41] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson. How transferable are features in deep neural networks? In *Advances in neural information processing systems*, pages 3320–3328, 2014.

[42] M. Ziane, M. Zaït, and P. Borla-Salamet. Parallel query processing with zigzag trees. *The VLDB JournalâĂŤThe International Journal on Very Large Data Bases*, 2(3):277–302, 1993.