Learning a Query Plan Optimizer with Reinforcement Learning

Jeff Sheldon jeffsh@cs.washington.edu

Optimizing query plans is an exercise in estimating the cardinality cost of executing a query, whereby the plan with the lowest estimated cost will be executed. Planners are typically given a plan in linear algebraic form and cardinality costs for the low-level components that form the basis of the query. Using dynamic programming, a series of complete plans are formed, and their costs compared to determine the optimal plan to execute. The entire process relies on the accuracy of cardinality estimates to make the proper plan determination. In practice, these costs may vary significantly from the true cost of executing a query plan. Given the inaccuracy of cardinality estimations, this project considers the task of estimating query plan costs to be an unknowable function.

Machine learning may be employed to learn an estimated version of a function. This project applies deep learning techniques to learn a function capable of accurately estimating query plan costs. By producing a model capable of inferring accurate costs of proposed plans, we may select the optimal plan to execute. A neural network architecture will be implemented that, when given a series of query plans and related cardinality estimations, will produce an estimated cost for each query plan, thus allowing the optimal plan to be selected.

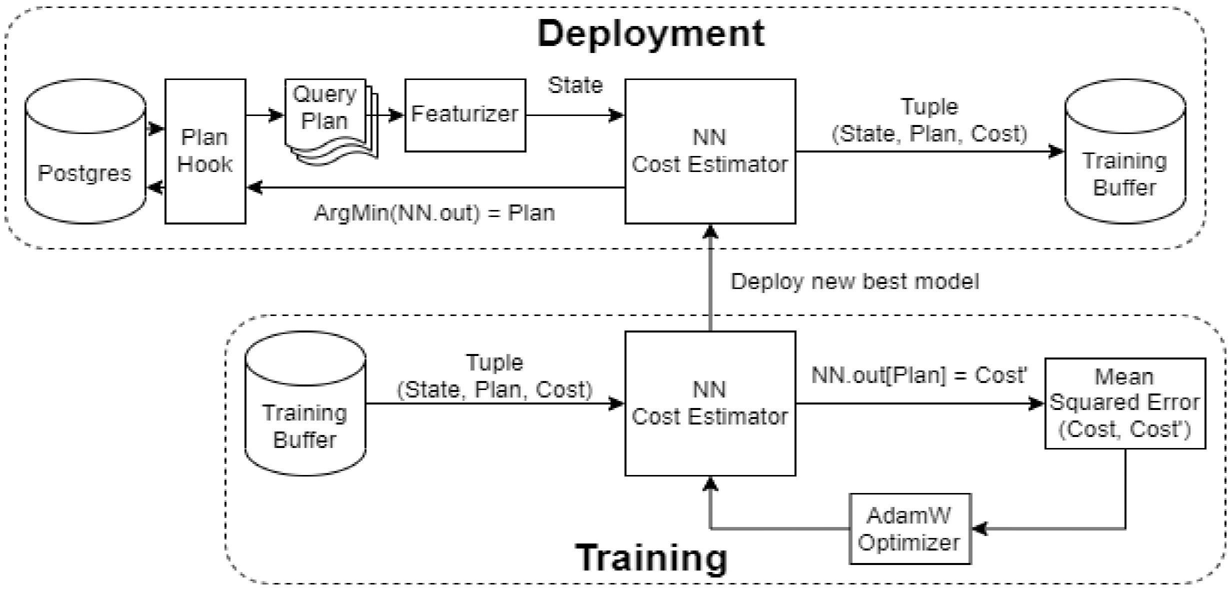
System Architecture

Supervised learning is a poor fit for a query cost estimator for two reasons. Primarily, the dataset used to train such a system is static and will not evolve with the system. Secondly, such a dataset, containing query proposals and their true execution cost is not known to exist.

Instead, an architecture is proposed that treats this as a reinforcement learning problem. A hypothetical agent will observe the current state of its environment, which is the combination of query plans to estimate and cardinality estimates. Given the observed state, it will select what it believes is the most efficient plan to execute. It will then observe the true cost of executing that plan. The result of that process is a tuple composed of the initial state, the selected plan, and the observed cost. This tuple may then be used as a learning experience for the agent. A dataset is constructed from all such tuples, which will allow for a supervised training process to produce a new model for the agent to query when selecting a plan.

A deployment pipeline is used by a database engine to select a query plan to execute. The database engine will propose a series of query plans. The deployment pipeline will estimate the cost of each plan and instruct the database engine to execute the plan with the lowest estimated cost. The pipeline will record the state representation of the proposed plans and cardinality estimates, along with the plan it selected, and the observed cost of executing the plan, for future training.

A training pipeline will use the recorded states, selected plans, and observed costs for model training. This is a form of Q-learning, where a reward value (observed cost) is associated with a state-action pair (featurized query plans and cardinality estimates + plan selected). As the deployment pipeline continues to run, the set of data from which the training pipeline may train a model will continue to grow, leading to better models, which will be deployed back to the deployment pipeline.



Project Implementation

Prior work[i] has been done to integrate Postgres with Python, using function hooks written in C. This enables the use of Python deep learning frameworks. This project uses those C libraries[1] with no alteration. As those libraries are responsible for the data made available to Python, this project is limited to the raw feature data produced by those libraries. That data consists of two components: an array of proposed query plans and a buffer of cardinality, index, and primary key metrics.

A server is implemented in Python[ii], to which the function hook sends proposed plans and observed costs. Given an array of plans to select from, the Python server passes the plans and buffer information through a featurizer, to construct inputs into a neural network for cost inference. The neural network outputs a regression for each plan in the input array. The plan associated with the lowest regression is returned to Postgres for execution. On completion of plan execution, the actual cost of executing the plan is sent to the server. The observed cost, the featurized state from the prior step, and the plan that was selected to the training buffer.

The training pipeline is executed separately from the deployment pipeline. A neural network is produced for all possible hyperparameter setting. The best model is selected according to Mean Squared Error loss. As the neural network outputs a set of regressions, the regression used to measure loss is that which is tied to the plan that was selected when the sample was created by the deployment pipeline.

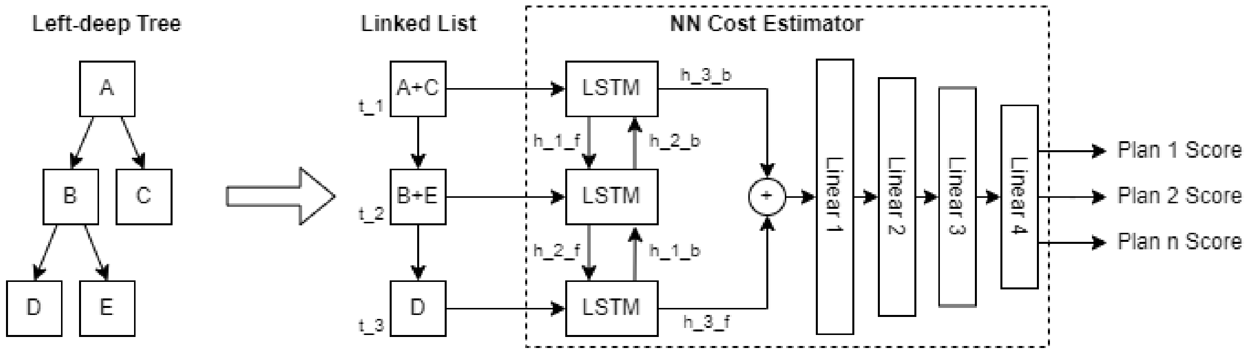
Featurization & Model Architecture

For each query executed, the deployment pipeline is passed an array of plans to estimate. The Postgres function hook ensures that these plans are always left-deep join plans. Natively, this structure is problematic for neural networks. Typical neural architectures rely on input matrices of one or two dimensions.

This project introduces a new method of shaping query plans to a data structure that is inherent to neural networks. Given a left-deep tree, we can rely on the following characteristics. First, any node that is not a leaf will have at least one and no more than two child nodes. Second, given a pair of sibling nodes, only

[1] https://github.com/learnedsystems/BaoForPostgreSQL/tree/master/pg_extension

one sibling may have descendants. Given these rules, we can imagine a small modification to the tree structure that results in a linked list. Given a node with two child nodes, we will merge the properties of the childless descendant into the parent. We are now guaranteed to never have more than a single descendant node from any parent node. Each node within the linked list is converted from a set of properties to a feature vector. This is done for all candidate query plans and the resulting vectors for each node are concatenated together.



An LSTM recurrent neural network is used to process each feature vector in the linked list. In the diagram above, the RNN is unwound and shown processing each element in the list at subsequent timesteps. From step to step, a hidden state is passed through the LSTM, encoding information from the current vector and all prior vectors. The LSTM is bidirectional, so that information is encoded in both directions. The output of the LSTM is two hidden states, one for each direction. These vectors are concatenated, capturing signal information from both ends of the input linked list that may have been diminished within the LSTM. The resulting vector is passed through four linear layers, the output of which is an estimated cost for each plan the network was asked to predict.

Dataset & Challenges

A version of the IMDB JOB[2] dataset is used for both training and evaluation. This dataset consists of 177 SQL queries and a 1.27GB database. The dataset is static. Because of the small size of this dataset, there is risk that the model may not be trained well enough to generalize to unseen data. To mitigate this, the dataset is synthetically expanded. The Postgres function hook is configured to produce five candidate query plans. For every query run, the neural network server is forced to propose each candidate, thus producing five samples for each query.

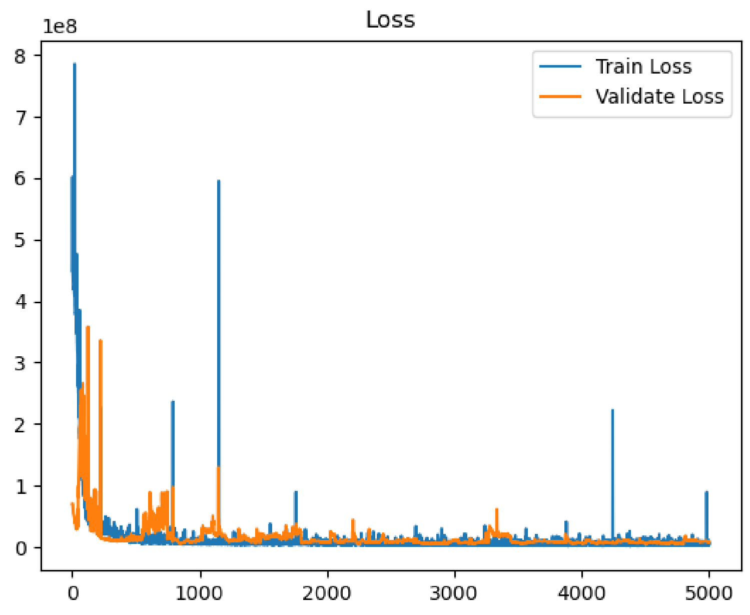
The 177 queries are split into two sets: a training set of 144 queries and a hold-out test set of 33 queries. Using the method described above, these sets are expanded to 720 and 165, respectively. The training set is further split into training data (648) and validation data (72) for hyperparameter tuning.

Evaluation

Dozens of models were produced, using different hyperparameters and neural architectures, and evaluated against each other to determine the correct model to use. Model evaluation was performed using mean squared error and training time as the metrics for comparison.

[2] https://github.com/RyanMarcus/imdb_pg_dataset

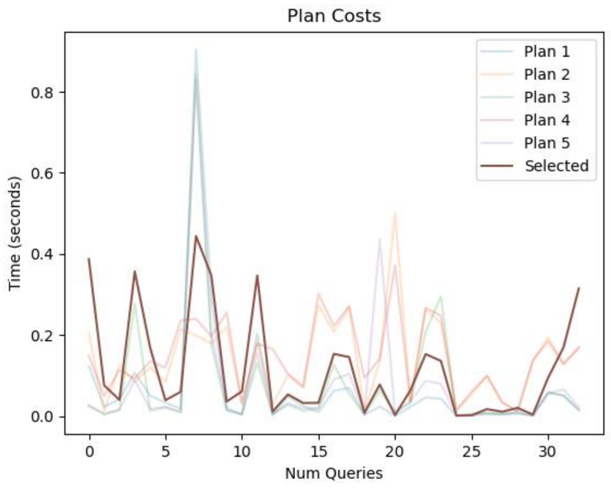
The final model was trained for 5000 epochs. The learning rate used was 0.00005. Batches of 32 were trained at a time. To the right is the comparison of loss between the training and validation datasets, over the 5000 epochs.



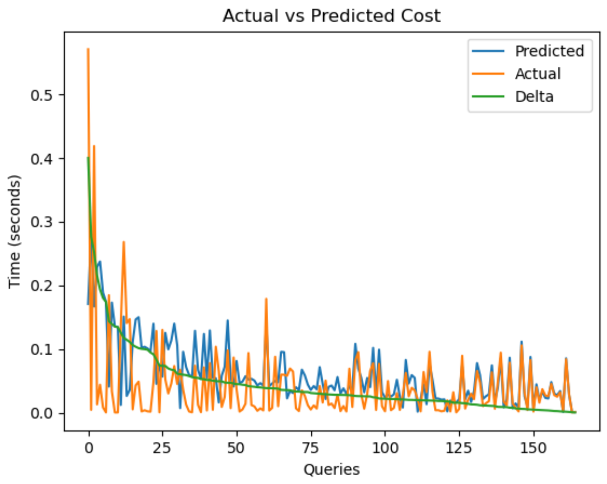
The final model is evaluated against the hold-out data, to determine the model's ability to generalize to unseen data. It should be noted that to properly evaluate the generalized performance of the model, it should be tested against an entirely different database. Due to time constraints on this project, that was not possible.

The model produced does well to generalize to the hold-out data but fails to outperform the Postgres query optimizer. From the comparison of all candidate plan costs, the model typically selects cheaper plans, but rarely selects the cheapest. Comparing the actual and predicted costs of all query plan candidates (five per query), we see that the model does well to predict the actual cost. In this metric, a perfect model would result in a delta of zero. While close, the final model fails to perfectly predict any costs.

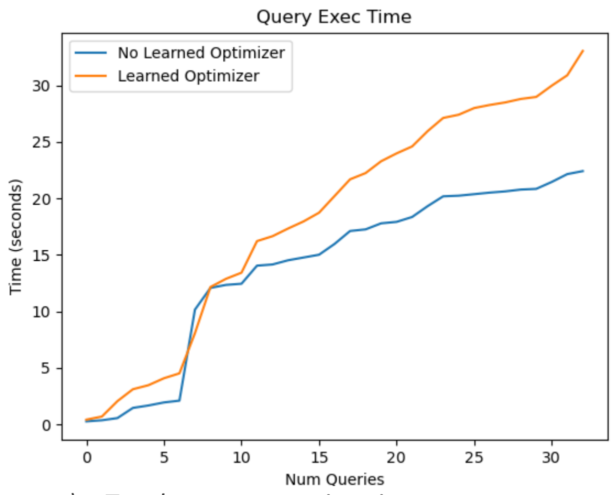
Because the model is incapable of reliably predicting accurate query plan costs, it is incapable of selecting the optimal query plan to perform. The total query execution time graph demonstrates this. For this evaluation, all 33 test queries were executed against the database initially, to induce the database to perform any cache optimizations it deemed appropriate. The 33 queries were then executed a second time, using the default Postgres optimizer and the runtime results recorded. The queries were then executed a third time, using the model to determine the plan to execute.



a) Observed cost of candidates and selected



b) Delta between normalized costs



c) *Total query execution time*

a) *The observed cost of all query plan candidates is depicted for each query. The darked line is the plan selected.*

b) *The predicted and observed costs for all plans are normalized and compared. The delta is depicted in green, and data sorted, from highest delta to lowest.*

c) *The total time to execute all SQL queries is shown, comparing the overall performance of the Postgres optimizer to the neural network.*

Conclusion

In this project, I have demonstrated that a recurrent neural network may be used to estimate the costs of query plans, thereby allowing for the selection of a query plan. When applied to a reinforcement learning setting, the model can train itself by observing the resulting cost of the query plan it selects. In a production environment, queries run over time will produce richer models.

The final model produced fit the training data very well but is incapable of outperforming the default Postgres optimizer. A lack of adequate training data is the likely cause.

This architecture shows promise and would benefit from more time. Given a far richer dataset from which to learn, it is believed that a model may be produced that is better able to generalize to unseen data. More time and more compute power would enable a better search for proper hyperparameters and neural network architectures. The architectures tested for this project were limited to variations of one another, with layers added and removed and density of layers increased and decreased.

---

[i] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, Tim Kraska. *Bao: Making Learned Query Optimization Practica*l. SIGMOD/PODS 2021 https://dl.acm.org/doi/10.1145/3448016.3452838

ii Project source code: https://github.com/j-confusatron/DBQueryOptimizer