AlphaJoin: Join Order Selection à la AlphaGo

Ji Zhang[†‡]
Supervised by Ke Zhou[†] and Sebastian Schelter[‡]
[†]Hua Zhong University of Science and Technology, [‡]University of Amsterdam
jizhang@hust.edu.cn

ABSTRACT

Query optimization remains a difficult problem, and existing database management systems (DBMSs) often miss good execution plans. Identifying an efficient join order is key to achieving good performance in database systems. A primary challenge in join order selection is enumerating a set of candidate orderings and identifying the most effective ordering. Searching in larger candidate spaces increases the potential of finding well-working plans, but also increases the cost of query optimization.

Inspired by the success of *AlphaGo* for the game of Go. In this Ph.D. work, we propose an optimization approach referred to as `AlphaJoin`, which applies AlphaGo's techniques, namely Monte Carlo Tree Search (MCTS), to the join order selection problem. Preliminary results indicate that our approach consistently outperforms a state-of-the-art method and the PostgreSQL's optimizer on its own respective execution engine. Our approach is open-sourced and publicly available on Github[1].

1. INTRODUCTION

Database tuning is vital for optimizing the performance of a database management system (DBMS) [1, 2, 3, 4, 5]. Query optimization is one of the most well-studied issues in this field. Identifying an efficient join order is key to achieving good performance in database systems. A primary challenge in join order selection is to minimize the number of execution plans to enumerate, as well as the runtime of the final chosen plan [6]. Traditional database systems employ a variety of heuristical methods (dynamic programming, greedy approaches, genetic algorithms and simulated annealing) as a join order selection policy. However, these traditional query optimizers rely on internal static information and hence do not learn from historic experiences. Because of the lack of feedback, these methods select a query plan, execute it and then forget this selection; thus they never learn from previous experiences.

In the face of the recent success of machine learning (ML) for various computer science problems, it is very natural to think about the idea of using ML for join order selection in the optimizer [6, 7, 8]. Marcus et al. [6] proposed a proof-of-concept join order enumerator named `ReJOIN` entirely driven by deep reinforcement learning to learn from previously e executed plans. They provide preliminary results that indicate that their approach outperforms PostgreSQL's join enumeration process in terms of effectiveness and efficiency. Unfortunately, `ReJOIN` and the traditional heuristics methods all assume a cost-based approach (search a subspace of all possible join orderings and select the "cheapest" order according to the cost model based on statistical information) to join order selection optimization. These approaches still require a human-designed cost model and might be not accurate when the data in the database changes dynamically [8]. In other words, although PostgreSQL's execution engine chooses the join order with the lowest cost, its actual execution time may not be the lowest. This illustrates that the cost model of PostgreSQL might not really reflect the execution time of the query plan [15].

Marcus et al. [8] presented a learning optimizer called `NEO` that generates highly efficient query execution plans using deep neural networks, based on the actual execution time instead of a cost-based model, which achieves similar or improved performance compared to state-of-the-art commercial optimizers on their respective query execution engines. However, these methods (traditional database execution engines, `ReJOIN` and `NEO`) are based on some simple search strategies which search unevenly (randomly) result in some plans that were never tried (evenly a part of the complete plan) and have a possibility of falling in a local optimum.

Inspired by the impressive search capability of Monte Carlo Tree Search (MCTS [9]), which is at the core of the *AlphaGo* [10] system for playing the game of Go, we explore the benefits of MCTS for the join order selection. We call our approach `AlphaJoin`. MCTS is a search method usually used in games to predict the set of moves that should be taken to reach a final winning solution with high likelihood. The main idea is to simulate many possible join orders in one tree structure which is efficient to learn for searching from an even manner. and to apply MCTS to select the order to execute with the highest estimated performance. We hope that `AlphaJoin` inspires many other database researchers to experiment with combining query optimizers in new ways.

In this Ph.D. work, we make the following contributions:

- To the best of our knowledge, `AlphaJoin` is the first approach to use MCTS to learn and generate a highly efficient join order in a query optimizer. We design a neural network (*Order Value Network*, OVN) to predict the query execution time of a given plan, and leverage this network within MCTS to score candidate query plans. (We refer to this as `AlphaJoin 1.0`)
- Based on `AlphaJoin 1.0`, we design another neural network called *Adaptive Decision Network* (ADN) to choose between our `AlphaJoin 1.0` and the PostgreSQL optimizer for a given query which further improves the optimization performance. (We refer to this approach as

[1]https://github.com/HustAIsGroup/AlphaJoin

Proceedings of the VLDB 2020 PhD Workshop, August 31st, 2020. Tokyo, Japan.

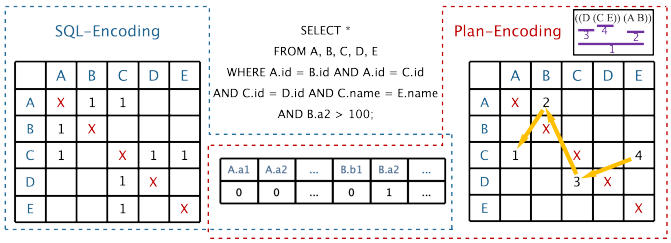


Figure 1: The encoding methods in `AlphaJoin`: SQL-encoding and Plan-encoding.

`AlphaJoin 2.0`)

- Our experimental results demonstrate that `AlphaJoin` can generate efficient join orders with improved performance compared to the state-of-the-art optimization tool *NEO* and PostgreSQL's query optimizer on its own respective execution engine.

2. APPROACH

In this section, we first describe two encoding methods (SQL-encoding and Plan-encoding) and then provide an overview of our proposed approach `AlphaJoin`. Note that `AlphaJoin 2.0` is the extended version of `AlphaJoin 1.0` to further improve the optimization performance.

2.1 Encodings

`AlphaJoin` uses two encodings: SQL-Encoding, which encodes information regarding the SQL query, but is independent of the query plan, and a plan-encoding, which represents the execution plan.

SQL-encoding encodes the table and attribute information contained in the SQL query. Similar to previous work [11], the representation of each query consists of two components: the first component encodes the join graph of the query in an adjacency matrix. A "1" in the matrix corresponds to the join predicate connecting two tables, e.g. in Figure 1, the "1" in the first row, second column corresponds to the join predicate connecting A and C. The second component is a simple "one-hot encoding" of the attributes involved in contained SQL predicates.

Plan-encoding In addition to the SQL encoding, we also require a representation of a partial or complete query execution plan. There needs to be a consistent one-to-one match between each encoding and the corresponding join order. In other words, the plan encoding method must be both encodable and decodable at the same time. However, the execution plan encoding in the proposed method `ReJOIN` [6] uses Huffman coding which is hard to represent as a dense tree and cannot distinguish between the left child and right child, driving table and driven table. Inspired by the encoding method for the state of the Go board in AlphaGo, we designed a new plan-encoding method that also consists of two components. The only difference to SQL-encoding is that we represent the join order instead of just the join graph in the encoding matrix (using different order numbers instead of only "1"). The larger a number in the matrix, the higher priority of a join operation for two corresponding tables. For example, in the red dotted frame of Figure 1, the "4" in the third row and fifth column corresponds to the join predicate connecting C and E at the first of all. Then, the "3" in the fourth row and third column corresponds to the join predicate connecting the result of (C E) and D. The "2" in the first row and second column corresponds to the

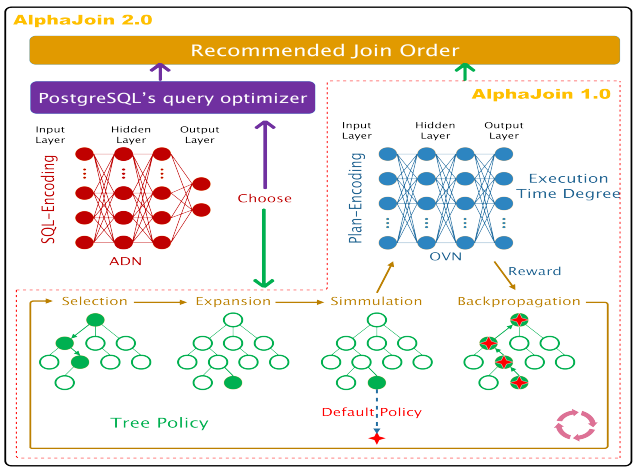


Figure 2: Overview of our proposed `AlphaJoin` which includes two encoding methods (SQL-encoding and Plan-encoding), two trained neural networks OVN and ADN, and an optimizer which applies MCTS.

join predicate connecting A and B. Lastly, the "1" in the third row and first column corresponds to the join predicate connecting the result of (D (C E)) and the result of (A B). Note that this process of execution plan encoding is similar to the encoding of moves in AlphaGo.

2.2 `AlphaJoin 1.0`

Next, we introduce `AlphaJoin 1.0`, which consists of two components: the Order Value Network and MCTS. Afterwareds, we show some preliminary results of `AlphaJoin 1.0`.

2.2.1 *Order Value Network*

The order value network (OVN) is a deep neural network to predict the best-possible query execution time degree for a partial execution plan. The architecture of the OVN is shown in Figure 2. It consists of an input layer I, three hidden layers H (2048, 512, 128-dimensional units) with ReLU activation, and an output layer O. As the goal of this neural network is to estimatewhich query plans are fast or slow, the data we feed to this network is the plan-encoding of the query plan and the output is the result of a multi-label classification, where the $K = 4$ possible labels indicate the execution time degree (from 0 to 4, the lower the degree is, the lower the execution time is) of the entered query plan. Note that we tried other setups, but achieved the best result with $K = 4$. We employ a $softmax$ output to produce a proper probability distribution over the execution time degree and additionally dropout regularization [12] to prevent overfitting. We train our network to minimize the cross entropy [13] between the historical query plans and their corresponding predicted execution time (the standard loss for multiclass classification problems) which is a measure between distributions. It is interesting to study other types of neural networks to improve the prediction result, and we explored convolutional networks, but found no significant performance improvements. Our model achieves about 60.9% accuracy on Join Order Benchmark (JOB). Although the predictive results are not on par with the results for most predictive tasks, in the process of MCTS, a large number of simulations will be performed to select the appropriate join order, and these simulations will make up for the accuracy of the network. Note that the value network of AlphaGo only achieves 50% accuracy but still exhibits good performance.

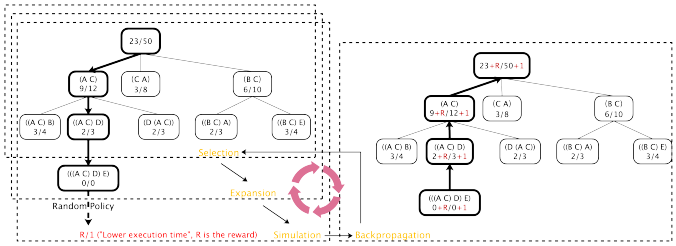


Figure 3: The correspondence between the four distinct steps in MCTS and join order selection.

2.2.2 MCTS for AlphaJoin

We first introduce the UCT algorithm, reward function in MCTS and then discuss the MCTS for join order selection.

UCT Algorithm. The "Upper Confidence Bounds applied to Trees (UCT)" [14] algorithm is a game tree search algorithm to solve the problem of which tree node should be selected. This algorithm adopts the well-known exploration & exploitation scheme, which not only gives full search (the learned experience) to the ability of the model but also explores more tree nodes that were previously never tried, to reduce the possibility of falling into a local optimum. The detailed UCT calculation formula is $UCT(v_i, v) = \frac{Q(v_i)}{N(v_i)} + C\sqrt{\frac{logN(v)}{N(v_i)}}$ where v_i is the current node, v is its parent node, $Q(v_i)$ refers to as the number of times to gain an advantage at the current node, $N(v_i)$ $(N(v))$ is the total number the current nodes (parent nodes) were accessed, and C is a parameter to adjust the sensitivity of exploration. Taking AlphaGo as an example, the first part of the formula refers to exploitation which determines the probability of "win" after selecting this node. The second part of the formula refers to exploration, and there is a high probability of exploring other untouched nodes (instead of v_i) if $N(v_i)$ is large. Note that the path from each child node to the root node in the tree represents a complete join order.

Reward Function. Analogous to AlphaGo, we need to define the "win" in `AlphaJoin` after a complete join order was searched by MCTS. In the query optimization problem, our goal is to generate a query plan that results in the lowest execution time. Therefore, the lower the execution time, the higher the probability of "win". We designed a simple reward function which is $R_j = \frac{K-k_j}{K}$ for the reward R_j of the evaluated join order j by MCTS. K is the predefined execution time degree in our OVN (Section 2.2.1); k_j (range from 0 to K) is the predicted execution time degree of the tried join order j. The reward achieves the largest value 1 if the execution time degree of the plan was predicted as the lowest degree 0.

MCTS for Join Order Selection. The MCTS algorithm is a decision-making algorithm that applies the Monte Carlo method for tree search. MCTS expands the tree through simulations when searching the space until making a decision, and feeds the final result of the decision ("win" or not) back to the nodes in the tree for updating. After a large number of simulations, the information of each node represents the ratio of the number of correct decisions to the total number of simulations at this node, which indicates the value of this node. In order to decrease the search space from the root node to the leaf node, we define the number of simulations required for each node as $S_n = N_C * F_s$ where N_C is the number of child nodes under the current node and F_s is a search factor to control the whole simulation time. The larger F_s is, the more search time MCTS will spend (we set it to 15 and analyze the impact of different values on the optimization performance in Section 2.2.3). The total number of simulations for each SQL query is $\sum_1^J S_n$, where J is the number of joins in the query. This process will continue until all possible join orders are searched or the predefined maximum number of simulations is reached. Each simulation of MCTS can be broken down into four distinct steps: selection, expansion, simulation and backpropagation. Each of these steps for join order selection is shown in Figure 3 and explained in details below:

1. Selection- We apply the UCT algorithm to select the join order from the root node to the leaf node. Once a child node which is also a leaf node is encountered during travel, MCTS jumps into the expansion step.

2. Expansion- In this process, a new child node (join order) is added to the tree to the node which was optimally reached during the selection process.

3. Simulation- A simulation is performed by randomly choosing moves or strategies until all the tables are joined.

4. Backpropagation- The backpropagation process is performed from the new node to the root node. During this process, the total number of simulations stored in each node is incremented (add "1"). If the new node's simulation results in a lower execution time, these nodes are also incremented (add reward "R").

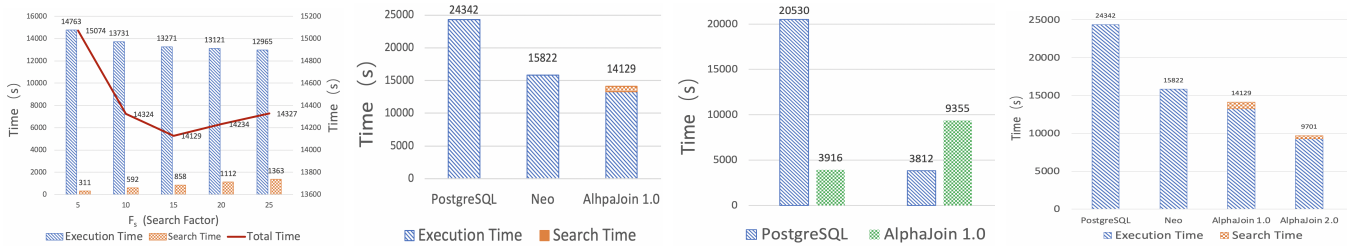
2.2.3 Preliminary Results for AlphaJoin 1.0

We first explore the performance impact of the search factor F_s on join order selection by `AlphaJoin` and then investigate the performance of `AlphaJoin 1.0`, PostgreSQL's join enumeration process and `NEO` via the Join Order Benchmark, a set of queries used in previous assessments of query optimizers [8, 15]. Figure 4(a) shows the impact of different search factors F_s on the optimized execution time and search time. As the search factor F_s increases from 5 to 25, the optimized execution time by `AlphaJoin 1.0` is continuously decreased but the search time of our method is increased. This is because a larger F_s causes more simulations on each node using MCTS. Therefore, a tradeoff is required to select an appropriate F_s between the optimized execution time and search time, thus we set F_s to 15.

Figure 4(b) shows the overall optimized performance of `AlphaJoin 1.0`, PostgreSQL's query optimizer and `NEO`. `AlphaJoin 1.0` outperforms other candidates even when we include the search time. This experiment demonstrates that our method using MCTS efficiently selects appropriate join orders.

2.3 AlphaJoin 2.0

In order to further improve the optimized performance of the `AlphaJoin 1.0`, it is interesting to compare the execution time of each SQL performed by PostgreSQL's query optimizer and `AlphaJoin 1.0`. Not all the join orders recommended by `AlphaJoin 1.0` are better than the ones generated by PostgreSQL's query optimizer. According to the statistics, still about 48% (almost half of the cases) of queries optimized by PostgreSQL's query optimizer on its own respective execution engine achieve a better performance than `AlphaJoin 1.0` (note that `NEO` achieves this in 41% of the cases). We attributes this to the uncertainty of the UCT algorithm in our method. Figure 4(c) shows the total execution time of those queries which achieve a better per-



(a) (b) (c) (d)

Figure 4: (a) The impact of different search factors F_s on the optimized execution time and search time. (b) The overall optimized performance `AlphaJoin 1.0`, PostgreSQL's query optimizer and `NEO`. (c) The total execution time of queries which achieve better performance via `AlphaJoin 1.0` (left) and PostgreSQL's query optimizer (right). (d) The overall optimized performance `AlphaJoin 2.0`, `AlphaJoin 1.0`, PostgreSQL's query optimizer and `NEO`.

formance via `AlphaJoin 1.0` (left) and PostgreSQL's query optimizer (right). An interesting finding is that although these two methods are similar in the number of preferred queries, `AlphaJoin 1.0` achieves a greatly improved performance compared to the PostgreSQL's join enumeration process. In other words, `AlphaJoin 1.0` performs better for slow queries (queries with large execution time). This is because these slow queries have more space for optimization using our method. In contrast, for the optimization of fast queries, our method is not suitable. In order to further improve the optimized performance of `AlphaJoin 1.0`, we introduce `AlphaJoin 2.0`. Note that we could also explore other methods to decrease the search time of MCTS, e.g., by parallelizing MCTS [16].

In order to alleviate the situation we analyzed above, we train another neural network referred to as the Adaptive Decision Network (ADN, shown in Figure 2) to choose between our `AlphaJoin 1.0` and the PostgreSQL optimizer for a given query. ADN is learned from a labeled dataset of historical execution times of the optimizer in PostgreSQL and our `AlphaJoin 1.0`. The structure of ADN is similar to OVN, the only difference are the inputs and outputs. The input is the SQL-encoding of the query and the output is a binary classification result which indicates which optimizer should be chosen to perform. We refer to this architecture as `AlphaJoin 2.0`.

3. PRELIMINARY RESULTS

We use the same benchmark and experimental setup to evaluate the performance between PostgreSQL's optimizer, `NEO`, `AlphaJoin 1.0` and `AlphaJoin 2.0`. We present preliminary experiments in Figure 4(d) that indicate that `AlphaJoin 2.0` can generate better join orders with lower execution time compared to the ones generated by PostgreSQL's optimizer (decreased by 60.1%), the state-of-the-art method `NEO` [8] (decreased by 38.7%) and our previously proposed version `AlphaJoin 1.0` (decreased by 31.3%). Moreover, we also decrease the ratio of the queries for which PostgreSQL's optimizer performed better from 48% to 9%.

4. CONCLUSION AND NEXT STEPS

In this Ph.D. work, we present `AlphaJoin`, the first MCTS-based query optimizer that generates highly efficient join orders for database optimizer. `AlphaJoin` iteratively improves its performance through a combination of two neural networks and MCTS. Preliminary results show `AlphaJoin` consistently outperforms the state-of-the-art method `NEO` and the PostgreSQL's optimizer.

As a next step, we plan to investigate methods for improving the prediction accuracy of two networks OVN and ADN to further optimize the search efficiency of MCTS. Besides, our method still performs worse than the optimizer in PostgreSQL on a set of fast queries, which we need to investigate to further improve the overall performance and propose "`AlphaJoin 3.0`".

5. REFERENCES

[1] Benoit Dageville et al. Automatic sql tuning in oracle 10g. In *VLDB*, pages 1098–1109, 2004.

[2] Surajit Chaudhuri et al. Self-tuning database systems: A decade of progress. In *VLDB*, pages 3–14, 2007.

[3] Dana Van Aken et al. Automatic dbms tuning through large-scale machine learning. In *SIGMOD*, 2017.

[4] Ji Zhang et al. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *SIGMOD '19*, page 415–432, 2019.

[5] Guoliang Li et al. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proc. VLDB Endow.*, 12(12):2118–2130, 2019.

[6] Ryan Marcus et al. Deep reinforcement learning for join order enumeration. aiDM'18. ACM, 2018.

[7] Ryan Marcus. Towards a hands-free query optimizer through deep learning. In *CIDR*, pages 1–8, 2019.

[8] Ryan Marcus et al. Neo: A learned query optimizer. *Proc. VLDB Endow.*, 12(11):1705–1718, 2019.

[9] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Computers and Games*, volume 4630, pages 72–83. Springer, 2006.

[10] David Silver et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 2016.

[11] Jennifer Ortiz et al. Learning state representations for query optimization with deep reinforcement learning. In *DEEM'18*. ACM, 2018.

[12] Nitish Srivastava et al. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014.

[13] Ian Goodfellow et al. *Deep Learning*. MIT Press, 2016.

[14] Sylvain Gelly et al. Exploration exploitation in go: Uct for monte-carlo go. In *NIPS Workshop OTEE*, 2006.

[15] Viktor Leis et al. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, 2015.

[16] Anji Liu. Watch the unobserved: A simple approach to parallelizing monte carlo tree search, 2018.