Learned Query Superoptimization (Extended Abstract)

Ryan Marcus

University of Pennsylvania

Abstract

Traditional query optimizers are designed to be fast and stateless: each query is quickly optimized using approximate statistics, sent off to the execution engine, and promptly forgotten. Recent work on learned query optimization have shown that it is possible for a query optimizer to "learn from its mistakes," correcting erroneous query plans the next time a plan is produced. But what if query optimizers could avoid mistakes entirely? This paper presents the idea of learned query superoptimization. A new generation of query superoptimizers could autonomously experiment to discover optimal plans using exploration-driven algorithms, iterative Bayesian optimization, and program synthesis. While such superoptimizers will take significantly longer to optimize a given query, superoptimizers have the potential to massively accelerate a large number of important repetitive queries being executed on data systems today.

1. Introduction

Traditional cost-based query optimizers [1] are designed to be fast. When a new query arrives, the optimizer performs some computations using statistical cardinality estimates, and produces a query plan. In this sense, traditional query optimizers can be viewed as cheap (compared to query execution) stateless functions. Each new query is a blank slate, representing a "fire and forget" system.

Past work on learned query optimization [2, 3, 4, 5, 6] have addressed the "forget" component of "fire and forget" optimizers. Since every optimizer processes queries with similar qualities (all querying the same database), one can say there is cross entropy (shared information) between each query. By learning a model of query performance that takes advantage of this cross entropy, learned query optimizers "learn from their mistakes." Since learned query optimizers are starting to see early commercial adoption [7, 5], we can conclude that the database community has made some progress on the "forget" element of traditional optimizers.

But what about the "fire" element? Query optimizers are built so that optimization time is low compared to query execution time, so the cost of query optimization can be seen as amortized over the query’s execution. The simplifying assumption at work here is that since each incoming query could be unlike anything seen before, the time spent optimizing any specific query might not be useful for optimizing other queries. When we view each query as arbitrary, this assumption seems valid.

But in many real applications, query workloads are highly repetitive. The same query template – and often the exact same query – might be executed many times. In fact, for many analytic dashboarding systems, the majority of cluster resources might go to executing a set of highly-repetitive queries (e.g., the hourly sales dashboard executes nearly-identical queries every hour).

Given the shape of these analytics workloads, does it make sense to view each query optimization as a cheap stateless function that only relies on optimizer statistics? This issue has been partially addressed by past work on parameterized or parametric query optimization (PQO) [9, 10, 11] and query plan caching [12, 13], which has mostly focused on further reducing optimization times (as opposed to query latency) by avoiding repetitive work, although there are some exceptions [14].

Here, I propose learned query superoptimization. Following the “anytime” model proposed in [15], this paper considers: what if instead of bounding the query optimizer to a few hundred milliseconds, we instead used hours or even days of computation to optimize a query? For queries that execute thousands or even millions of times a year (as may be the case in large dashboard applications), the additional optimization time could possibly "pay for itself."

The thought of using significant resources on query optimization may be viscerally unappealing to many in the database community. To assuage these feelings, note that "superoptimization" is a common concept in program compilation: since some programs are executed so often, spending a significant amount of time optimizing a program may be worthwhile even if the program’s execution time only improves marginally. If we are willing to spend extra time compiling our DBMSes to eke-out every last drop of performance, why not give DBMS users the same option for their queries?

This paper presents two orthogonal directions for future research on learned query superoptimization. Exhaustive plan search (Section 3) It is well known that cardinality estimates are often poor, and often the
will discuss future directions for learned query superoptimization work, grouped by their lifespan. The P50 (median) number of executions per template is rounded to the nearest thousand. Query templates used for at least half a year to a full year (the last row) represents 31% of the cluster’s total compute time.

<table>
<thead>
<tr>
<th>Duration</th>
<th># tmpl.</th>
<th>% time</th>
<th>P50 # execs</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 1 week</td>
<td>52</td>
<td>3%</td>
<td>&lt; 1000</td>
</tr>
<tr>
<td>1 - 4 weeks</td>
<td>181</td>
<td>5%</td>
<td>&lt; 1000</td>
</tr>
<tr>
<td>4 - 12 weeks</td>
<td>1092</td>
<td>6%</td>
<td>40900</td>
</tr>
<tr>
<td>12 - 24 weeks</td>
<td>540</td>
<td>19%</td>
<td>8700</td>
</tr>
<tr>
<td>24 - 52 weeks</td>
<td>10983</td>
<td>31%</td>
<td>108600</td>
</tr>
<tr>
<td>Total</td>
<td>12848</td>
<td>64%</td>
<td>(\approx 100000)</td>
</tr>
</tbody>
</table>

There were nearly 11,000 query templates that appeared in the logs for six months to a full year, and execution of those templates occupied 31% of the cluster’s resources throughout the 12-month period. It is worth noting that these queries are executed quite often: the median query template that lived for six months to one year was executed over a 100,000 times. With such a large percentage of cluster resources going to executing these repetitive queries, it seems reasonable to try and optimize repetitive queries specifically.

### 3. Exhaustive plan search

For queries joining \(n\) relations, optimizers must search \(O(3^n)\) \([20]\) plans when considering just join ordering, access paths (e.g., index vs. full scan), and operator selection (e.g., hash vs. merge). The key design principles of most traditional optimizers are (1) quickly eliminate large unpromising parts of the plan space, narrowing the search space to a manageable size, and (2) use pre-computed statistics to find an optimal query plan without significant data processing (i.e., without scanning the underlying data). Unfortunately, these key design principles are also often the root cause of suboptimal query plans:

- **A narrow search space** Traditional optimizers must use heuristics to exclude parts of the exponentially-large search space. Unfortunately, these heuristics can exclude the optimal plan. For example, the PostgreSQL optimizer excludes any plan containing cross joins from consideration, but if two dimension tables are sufficiently small, a cross join could be optimal. The reason traditional optimizers exclude cross joins is because plans with cross joins are *almost* always suboptimal, and even when a cross join is optimal, there is normally a near-optimal plan without a cross join. Nevertheless, while unusual, such heuristics can exclude optimal query plans without finding a near-optimal one \([2]\).

- **Optimizer statistics** Traditional query optimizers are at the mercy of their cost models and cardinality estimators. While cardinality estimators achieve reasonably good accuracy for table scans, estimator errors often reach catastrophic levels after only a few joins \([16]\). Thus, the optimizer must plan the final joins of a query plan (which are often the most significant) with virtually no statistics. Even learned query optimization \([4, 2, 3, 5]\) follow this paradigm: learned query optimizers still use heuristics to prune the plan space (e.g., the anytime search in \([2]\), and still only operate with pre-computed statistics: the only difference is that the pruning heuristic and statistics are learned.

What if we built a query superoptimizer that ignored both of these fundamental design principles? The simplest possible superoptimizer escaping this paradigm might "wrap" a traditional optimizer: execute the top \(k\) queries produced by the traditional optimizer and pick the best one, budgeting for the highest \(k\) possible. Such an optimizer would verify the quality of \(k\) query plans on actual data, albeit in a crude fashion. Another possibility
3.1. Design 1: repeated reinforcement learning

Reinforcement learning powered optimizers like Neo [2], Balsa [3], and Lero [5] work in episodes. Each episode corresponds to building a complete execution plan for a given query. Each episode also ends with a reward, the signal that the learned optimizer uses to adjust future actions.

Reinforcement learning algorithms must navigate the exploration-exploitation tradeoff. In short, a learning agent must choose to either (1) explore, trying something new and potentially gaining valuable information, or (2) exploit, try something that is known to work already and reap a reward. Existing learned query optimizers seek to perfectly balance exploration and exploitation, maximizing long-term rewards [22].

A learned query superoptimizer could tilt the scale significantly towards exploration. Doing so would cause the underlying reinforcement learning algorithm to produce more diverse and risky plans. The superoptimizer could then execute each plan, possibly on a sample of the database, and return the plan that works the best. This procedure not only produces a higher quality plan, settling on a final, reward-granting decision — is, to the best of my knowledge, yet unstudied by the database or reinforcement learning community.

3.2. Design 2: latent space optimization

A second possibility is inspired by recent work [26] applying Bayesian techniques to molecular optimization, the task of searching for molecules with particular properties. Translating from finding molecules to finding query plans, the fundamental idea is to encode query plans into a latent space, use Bayesian optimization techniques to optimize the plan within the latent space, and then decode the point in the latent space back into a query plan. Figure 1 shows an illustration of this process.

The motivation behind this idea is to translate the problem of query optimization from a "structural optimization" problem (i.e., a problem where the output must have a particular structure, like a query plan) into a "continuous optimization" problem (i.e., a problem in which the output is continuous). Once we have a continuous problem, a large number of well-studied continuous optimization algorithms can be applied (e.g., [27]).

**Encoder & latent space** The encoder’s $E : Q \rightarrow \mathbb{R}^n$ goal is to map a query plan $q \in Q$ into a $n$-dimensional latent space $\mathbb{R}^n$ using a neural network. We want positions in the latent space to be semantically relevant, meaning that we want similar points in the latent space to represent similar query plans and have similar performance properties (i.e., latency, IOs). One way to do this is with an information bottleneck [28]: we train a model to predict the performance properties of a query, but we make one of the last layers (possibly the second or third from the end) intentionally small. This forces the model to learn a compact representation of the query plans in the intentionally small layer, but also requires that the intentionally small layer is organized in such a way that allows for predicting the performance properties of the query. Such a model will not achieve high accuracy compared to models without an information bottleneck, but accuracy is not our goal. We can now chop off the layers after the information bottleneck, and use the resulting network as an encoder: the information bottleneck layer serves as our latent space.

**Decoder** The decoder’s $D : \mathbb{R}^n \rightarrow Q$ job is to take a point in the latent space created by the encoder and transform the point back into a query plan. The decoder can be trained by differentiating through the encoder, freezing the weights of the encoder during the process. Architecturally, the decoder can be anything that produces a sequence, but a reasonable choice may be a transformer model [29] or an LSTM [30].

**Bayesian optimization** The core of a latent space query superoptimizer is Bayesian optimization. The query optimization problem can be cast as a black-box optimization...
The previous proposals for query superoptimizers have all assumed that the final output of a query optimizer should be an executable query plan. In this section, we propose a “rethinking of the query optimization contract” [15] – specifically, we propose a query optimizer that simultaneously optimizes query execution plans and the underlying physical layout of the data. To illustrate this potential, imagine writing a program that answers queries about the relationships between actors, movies, and production companies. The system only needs to be able to answer questions in the form of $Q_1$: 

$Q_1$: How many movies were produced by company $C$ and starred actor $A$?

You could implement this system using a relational database, containing five relations, as shown in the ER diagram in Figure 2. Answering the query for any $A$ and $C$ can now be done with a simple SQL query. The underlying optimizer will join the five relations together, possibly even changing the join order based on $A$ and $C$. Note that, because both stars and produces are many-to-many, using fewer than 5 relations would violate second normal form.

Do we think this implementation is optimal? Of course not. Sure, using a relational database has great engineering benefits, such as (1) the reuse of existing components, (2) the ability to extend the system to other queries, and (3) most organizations already know how to maintain a DBMS. But, if our data system’s job is to answer $Q_1$ as fast as possible, then we know that the algorithm executed by the relational database will not be optimal.

Instead of four joins, imagine we store two hashmaps: one mapping each actor to a bitmap, where that bitmap stores a 1 if the actor appears in that movie, and a second mapping each company to a bitmap, where that bitmap stores a 1 if the company produces that movie. Now, answering $Q_1$ is as simple as two hashmap lookups and a bitmap count-intersect. With a cuckoo hashmap and smart prefetching, this could be implemented with only 6 cache misses.

Figure 3 compares the performance of PostgreSQL and an implementation of the bespoke system implemented in Rust using the standard library’s hashmap and Roaring [34] bitmaps. The bespoke solution is an order of magnitude faster than the RDBMS. Further, a traditional RDBMS has no chance of replicating the execution strategy of the custom solution.1

1One could manually create bitmap indexes and query the count-intersection of them, but this is essentially just creating the customized solution inside of the DBMS, instead of allowing the DBMS optimizer to find an optimal plan.
The above question may seem trivial: a smart DBA could likely create a similar effect using clever materialized views [35]. To see how program synthesis could help in less trivial cases, imagine designing a system to answer $Q_2$:

$$Q_2: \text{How many movies were produced by } C \text{ and starred actor } A \text{ with a rating } R \text{ s.t. } R_1 < R \leq R_2.$$  

The rating of a movie can be stored as a computed attribute of Movie in Figure 2. Assuming a 5-star review system, a synthesized system similar to the system for $Q_1$ could store a bitmap for each discrete rating (i.e., a bitmap for 1 star films, 2 star films, 3 star films, etc.), and then queries could be answered by counting the number of bits in the appropriate bitmaps.

What the customized hashmap solution gains in speed, it loses in generality. Adding additional query types would require a significant re-write. Implementing transaction semantics for adding new actors, movies, or companies may be challenging. But, we know that bespoke data management systems are occasionally created for important applications (e.g., Google Ads [36], Microsoft’s SCOPE [37]), often at a high cost. Depending on the business value of answering $Q_1$ or $Q_2$ quickly, a more bespoke solution may be worthwhile.

How can we make DBMSes capable of automatically generating the customized hashmap solution for answering $Q_1$ or $Q_2$, while simultaneously providing the generality and ease-of-use of SQL? Even capturing 50% of the bespoke system’s performance in a generic RDBMS would be a huge win.

One possible way to make that happen is machine programming [38], or getting computers to program themselves. Imagine that we can map out the myriad of potential optimizations, from bitmap intersections to augmented binary trees. It is possible that program synthesis techniques (e.g., [39]), taking only the schema and SQL query as an input, could be used to construct provably-correct, custom-tailored data systems like the hashmap solution described above. Combined with the right learning technique to guide the search, a synthesis approach might be able to create data structures and algorithms that outperform human experts. Similar advancements have already occurred in graphics processing [40, 41] and concurrent algorithms [42].

Building a mapping or library of potential optimizations may seem challenging, but today there are many examples of such libraries and even techniques to synthesize the library itself. Data Calculator [43] showed how different data structures can be composed together in a provably-correct way to build a variety of key-value stores. Castor [19] makes progress in terms of defining a formal language that could potentially express more complex data layouts. GenesisDB [44] and CodexDB [45] showed how large language models might be able to create customized database components. LearnedRewrite [46] shows how search techniques can be covering a wide space when rewriting SQL queries.

5. Related work

Query optimization is one of the most well-studied problems in the database community, with a history spanning 50 years [1]. A change to the fundamental “contract” of the query optimizer was proposed in [15]. More recently, we have seen applications of machine learning techniques [47, 48, 49, 50, 51], especially reinforcement learning [2, 23, 3, 5, 52, 53], to the problem of query optimization. [19, 43] show how program synthesis techniques can be used to create custom-tailored relational layouts or key-value stores.

In general, machine programming [38] represents a distinct field of research, and includes work about garbage collection [54], static analysis [55], and program regression detection [56]. Bayesian optimization is a well-studied area, and [31] provides a survey of this area. [26] inspired the latent space optimization proposed here. The idea of autonomous experimentation on top of a database system has also been previously explored [57, 58]. To the best of my knowledge, the term “superoptimization” was coined in [59].

6. Conclusions

This paper has proposed learned query superoptimization, an opportunity for the database community to redefine the task of query optimization for increasingly-common repetitive analytic workloads. Many of the ideas mentioned here will be pursued in the coming years. Collaborators are welcome!
References


