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, 8], 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

Joint Workshops at 49th International Conference on Very Large Data Bases (VLDBW'23) — Workshop on Applied AI for Database Systems and Applications (AIDB'23), August 28 - September 1, 2023, Vancouver, Canada

rcmarcus@seas.upenn.edu (R. Marcus)

0000-0002-1279-1124 (R. Marcus)

Table 1
The number of query templates executed on a commercial workload, 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.

Duration	# tmpl.	% time	P50 # execs
$<$ 1 week	52	3%	$<$ 1000
1 - 4 weeks	181	5%	$<$ 1000
4 - 12 weeks	1092	6%	40900
12 - 24 weeks	540	19%	8700
24 - 52 weeks	10983	31%	108600
Total	12848	64%	$\approx$ 100000

cause of poor query plans [16]. While inaccurate, cardinality estimates are fast to compute – but what if we took a different approach, and designed a query optimizer that intrusively looked at data and executed queries to get around inaccurate cost estimates?

Program synthesis (Section 4) While DBMSes seek to provide a fast implementation of the relational model to users, the abstractions of the relational model often seep into DBMS design. Bespoke systems (e.g., Millwheel [17], Bigtable [18]) fully discard the generality of the relational model, which allows them to use custom implementations that often outperform anything a general DBMS could offer. Expanding on [19], what if program synthesis techniques we could *automatically* generate a bespoke data system for a user?

To begin, Section 2 will show summary statistics from a large, highly-repetitive workload that motivates the idea of query superoptimization. Then, Section 3 and 4 will discuss future directions for learned query superoptimization work.

2. Repetition in the real world

In modern analytics systems, repetitive parameterized queries are common. Dashboards, along with weekly and monthly reports, are regularly implemented using parameterized queries.

Of course, modern analytics systems also get a good number of ad-hoc queries. While collaborating with a large corporation, I analyzed a year of query logs from an on-premise data warehouse to determine how often parameterized queries were executed. The results are summarized in Table 1.

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 precomputed *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

is using an evolutionary algorithm to search for better plans, as in [14], to use on-the-fly data sampling for join planning as in [21], or to use various forms of cardinality injection [15].

But we can do much better: it may be possible to build query superoptimizers that interleave machine learning and query execution. Next, I propose two possible systems using this paradigm.

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, but this procedure also gathers experience faster, allowing the underling reinforcement learning agent to learn faster.

An additional benefit to a reinforcement-learning powered superoptimizer is that many of the features and safety mechanisms built into current learned systems could be removed. For example, Bao [23] provides a number of tools, each using significant resources, to ensure regressions do not occur. With a superoptimizer, no such advanced regression-avoiding tooling is needed, since the empirical quality of the query plan can be observed before optimization finishes.

Tipping the balance towards exploration is far from an optimal solution. Since even a superoptimizer must operate in a limited time budget, choosing exactly which plans to test is an optimization problem on its own. Simply selecting the first k exploratory plans might be far from ideal, since all k plans might explore the same part of the search space. Active learning techniques, like those used in Datafarm [24], might be applicable here. This particular problem — choosing the ideal k explorations before 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 $E : Q \to \mathbb{R}^n$ 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 $D : \mathbb{R}^n \to Q$ 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

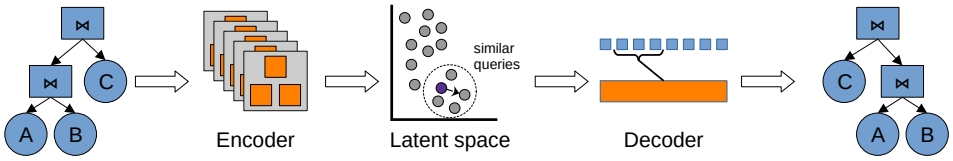

Figure 1: An encoder-decoder design for query optimization. A query plan is ran through a neural network (e.g., tree convolution [25]), which transforms the query into a point in a latent space. The encoder is trained to map queries with similar performance properties together. A Bayesian optimizer takes a step in the latent space, moving from the purple circle to the pointed-to circle. The decoder uses generates the new query plan tree, which is executed. The performance of the resulting query plan is given to the Bayesian optimizer, which can then take another step in the latent space.

problem. Let the query plan p_1 be the plan generated by a traditional query optimizer, which will be the initial condition for our optimization. Further, let $L(p)$ be the latency of p, determined by executing the plan. The Bayesian optimization algorithm then searches for a vector $\hat{v}$ such that:

$$\underset{\hat{v}}{\text{argmin}} \quad L(D(E(p_1) + \hat{v}))$$

The point $E(p_1) + \hat{v}$ can then be decoded into a query plan using the decoder D.

Bayesian optimization of expensive functions with continuous inputs is a well-studied problem, with many algorithms available [31]. Reinforcement learning powered query optimizers had to invent customized algorithms to deal with the specific challenges of query optimization. By tapping into a preexisting field like Bayesian optimization (which have already had proven success in drug discovery [27], hardware design [32], and even urban planning [33]), a learned query superoptimizer may be in reach in the near future.

4. Program synthesis

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

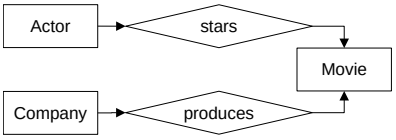

Figure 2: ER diagram of actors, movies, and production companies (attributes omitted). The standard realization into a physical schema would contain five relations.

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]

[1] One 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.

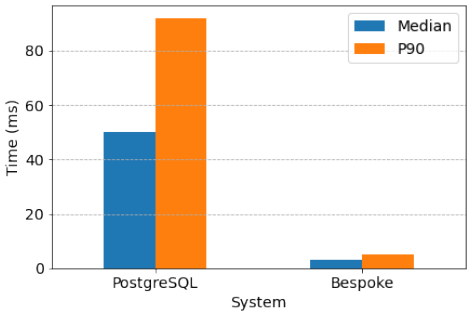

Figure 3: Performance of PostgreSQL vs. the bespoke two-hashmap system for Q_1. The bespoke system performs significantly better at both the 50th (median) and 90th percentiles.

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: How many movies were produced by C and starred actor A with a rating R 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 cover 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

[1] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, T. G. Price, Access Path Selection in a Relational Database Management System, in: J. Mylopolous, M. Brodie (Eds.), SIGMOD '79, SIGMOD '79, Morgan Kaufmann, San Francisco (CA), 1979, pp. 511–522. doi:10.1016/B978-0-934613-53-8.50038-8.

[2] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, N. Tatbul, Neo: A Learned Query Optimizer, PVLDB 12 (2019) 1705–1718. doi:10.14778/3342263.3342644.

[3] Z. Yang, W.-L. Chiang, S. Luan, G. Mittal, M. Luo, I. Stoica, Balsa: Learning a Query Optimizer Without Expert Demonstrations, in: Proceedings of the 2022 International Conference on Management of Data, SIGMOD '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 931–944. doi:10.1145/3514221.3517885.

[4] R. Marcus, O. Papaemmanouil, Deep Reinforcement Learning for Join Order Enumeration, in: First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM @ SIGMOD '18, Houston, TX, 2018.

[5] R. Zhu, W. Chen, B. Ding, X. Chen, A. Pfadler, Z. Wu, J. Zhou, Lero: A Learning-to-Rank Query Optimizer (2023). doi:10.48550/ARXIV.2302.06873.

[6] M. Stillger, G. M. Lohman, V. Markl, M. Kandil, LEO - DB2's LEarning Optimizer, in: VLDB, VLDB '01, 2001, pp. 19–28.

[7] P. Negi, M. Interlandi, R. Marcus, M. Alizadeh, T. Kraska, M. Friedman, A. Jindal, Steering Query Optimizers: A Practical Take on Big Data Workloads, in: Proceedings of the 2021 International Conference on Management of Data, SIGMOD '21, ACM, Virtual Event China, 2021, pp. 2557–2569. doi:10.1145/3448016.3457568.

[8] W. Zhang, M. Interlandi, P. Mineiro, S. Qiao, N. Ghazanfari, K. Lie, M. Friedman, R. Hosn, H. Patel, A. Jindal, Deploying a Steered Query Optimizer in Production at Microsoft, in: Proceedings of the 2022 International Conference on Management of Data, SIGMOD '22, ACM, Philadelphia PA USA, 2022, pp. 2299–2311. doi:10.1145/3514221.3526052.

[9] K. Vaidya, A. Dutt, V. Narasayya, S. Chaudhuri, Leveraging query logs and machine learning for parametric query optimization, Proceedings of the VLDB Endowment 15 (2022) 401–413. doi:10.14778/3494124.3494126.

[10] P. Bizarro, N. Bruno, D. J. DeWitt, Progressive Parametric Query Optimization, IEEE Transactions on Knowledge and Data Engineering 21 (2009) 582–594. doi:10.1109/TKDE.2008.160.

[11] Y. E. Ioannidis, R. T. Ng, K. Shim, T. K. Sellis, Parametric query optimization, The VLDB Journal 6 (1997) 132–151. doi:10.1007/s007780050037.

[12] T. Boggiano, G. Fritchey, What Is Query Store?, in: T. Boggiano, G. Fritchey (Eds.), Query Store for SQL Server 2019: Identify and Fix Poorly Performing Queries, Apress, Berkeley, CA, 2019, pp. 1–29. doi:10.1007/978-1-4842-5004-4_1.

[13] G. Aluç, D. E. DeHaan, I. T. Bowman, Parametric Plan Caching Using Density-Based Clustering, in: 2012 IEEE 28th International Conference on Data Engineering, ICDE '12, 2012, pp. 402–413. doi:10.1109/ICDE.2012.57.

[14] L. Doshi, V. Zhuang, G. Jain, R. Marcus, H. Huang, D. Altinbuken, E. Brevdo, C. Fraser, Kepler: Robust Learning for Faster Parametric Query Optimization, Proceedings of the 2023 ACM SIGMOD Conference 1 (2023) 109. doi:10.1145/3588963.

[15] S. Chaudhuri, Query optimizers: Time to rethink the contract?, in: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, SIGMOD '09, Association for Computing Machinery, New York, NY, USA, 2009, pp. 961–968. doi:10.1145/1559845.1559955.

[16] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, T. Neumann, How Good Are Query Optimizers, Really?, PVLDB 9 (2015) 204–215. doi:10.14778/2850583.2850594.

[17] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, S. Whittle, MillWheel: Fault-tolerant stream processing at internet scale, Proceedings of the VLDB Endowment 6 (2013) 1033–1044. doi:10.14778/2536222.2536229.

[18] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, R. E. Gruber, Bigtable: A Distributed Storage System for Structured Data, ACM Transactions on Computer Systems 26 (2008) 4:1–4:26. doi:10.1145/1365815.1365816.

[19] J. Feser, S. Madden, N. Tang, A. Solar-Lezama, Deductive optimization of relational data storage, Proceedings of the ACM on Programming Languages 4 (2020) 170:1–170:30. doi:10.1145/3428238.

[20] K. Ono, G. M. Lohman, Measuring the Complexity of Join Enumeration in Query Optimization, in: VLDB, VLDB '90, 1990, pp. 314–325.

[21] T. Neumann, M. J. Freitag, Umbra: A Disk-Based System with In-Memory Performance, in: 10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings, CIDR '20, www.cidrdb.org, 2020.

[22] R. S. Sutton, A. G. Barto, Introduction to Reinforcement Learning, 1st ed., MIT Press, Cambridge, MA,

USA, 1998.

[23] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, T. Kraska, Bao: Making Learned Query Optimization Practical, in: Proceedings of the 2021 International Conference on Management of Data, SIGMOD '21, China, 2021. doi:10.1145/3448016.3452838.

[24] R. Van De Water, F. Ventura, Z. Kaoudi, J.-A. Quiané-Ruiz, V. Markl, Farming Your ML-based Query Optimizer's Food, in: 2022 IEEE 38th International Conference on Data Engineering (ICDE), ICDE '22, 2022, pp. 3186–3189. doi:10.1109/ICDE53745.2022.00294.

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

[26] N. Maus, H. T. Jones, J. Moore, M. Kusner, J. Bradshaw, J. R. Gardner, Local Latent Space Bayesian Optimization over Structured Inputs, in: Advances in Neural Information Processing Systems, NeurIPS '22, 2022.

[27] D. Eriksson, M. Pearce, J. R. Gardner, R. Turner, M. Poloczek, Scalable global optimization via local Bayesian optimization, in: Proceedings of the 33rd International Conference on Neural Information Processing Systems, NeurIPS '19, Curran Associates Inc., Red Hook, NY, USA, 2019, pp. 5496–5507.

[28] O. Shamir, S. Sabato, N. Tishby, Learning and generalization with the information bottleneck, Theoretical Computer Science 411 (2010) 2696–2711. doi:10.1016/j.tcs.2010.04.006.

[29] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is All you Need, in: Advances in Neural Information Processing Systems, volume 30 of *NeurIPS '17*, Curran Associates, Inc., 2017.

[30] S. Hochreiter, J. Schmidhuber, Long Short-Term Memory, Neural Computation 9 (1997) 1735–1780. doi:10.1162/neco.1997.9.8.1735.

[31] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, N. de Freitas, Taking the Human Out of the Loop: A Review of Bayesian Optimization, Proceedings of the IEEE 104 (2016) 148–175. doi:10.1109/JPROC.2015.2494218.

[32] J. Wang, S. C. Clark, E. Liu, P. I. Frazier, Parallel Bayesian Global Optimization of Expensive Functions, Operations Research 68 (2020) 1850–1865. doi:10.1287/opre.2019.1966.

[33] J. T. Springenberg, A. Klein, S. Falkner, F. Hutter, Bayesian Optimization with Robust Bayesian Neural Networks, in: Advances in Neural Information Processing Systems, volume 29 of *NeurIPS '16*, Curran Associates, Inc., 2016.

[34] D. Lemire, G. Ssi-Yan-Kai, O. Kaser, Consistently faster and smaller compressed bitmaps with Roaring, Software—Practice & Experience 46 (2016) 1547–1569. doi:10.1002/spe.2402.

[35] X. Wu, D. Theodoratos, A. Kementsietsidis, Configuring bitmap materialized views for optimizing XML queries, World Wide Web 18 (2015) 607–632. doi:10.1007/s11280-013-0272-y.

[36] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. Dhoot, A. R. Kumar, A. Agiwal, S. Bhansali, M. Hong, J. Cameron, M. Siddiqi, D. Jones, J. Shute, A. Gubarev, S. Venkataraman, D. Agrawal, Mesa: A geo-replicated online data warehouse for Google's advertising system, Communications of the ACM 59 (2016) 117–125. doi:10.1145/2936722.

[37] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, J. Zhou, SCOPE: Easy and efficient parallel processing of massive data sets, Proceedings of the VLDB Endowment 1 (2008) 1265–1276. doi:10.14778/1454159.1454166.

[38] J. Gottschlich, A. Solar-Lezama, N. Tatbul, M. Carbin, M. Rinard, R. Barzilay, S. Amarasinghe, J. B. Tenenbaum, T. Mattson, The three pillars of machine programming, in: Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL 2018, Association for Computing Machinery, Philadelphia, PA, USA, 2018, pp. 69–80. doi:10.1145/3211346.3211355.

[39] A. Solar-Lezama, Program Synthesis by Sketching, Ph.D. thesis, University of California, Berkeley, United States – California, 2008.

[40] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, S. Amarasinghe, Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines, ACM SIGPLAN Notices 48 (2013) 519–530. doi:10.1145/2499370.2462176.

[41] M. B. S. Ahmad, J. Ragan-Kelley, A. Cheung, S. Kamil, Automatically translating image processing libraries to halide, ACM Transactions on Graphics 38 (2019) 204:1–204:13. doi:10.1145/3355089.3356549.

[42] M. Vechev, E. Yahav, G. Yorsh, Abstraction-guided synthesis of synchronization, in: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '10, Association for Computing Machinery, New York, NY, USA, 2010, pp. 327–338. doi:10.1145/1706299.1706338.

[43] S. Idreos, K. Zoumpatianos, B. Hentschel, M. S. Kester, D. Guo, The Data Calculator: Data Structure Design and Cost Synthesis from First Princi-

ples and Learned Cost Models, in: Proceedings of the 2018 International Conference on Management of Data, SIGMOD '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 535–550. doi:10.1145/3183713.3199671.

[44] Immanuel Trummer, GenesisDB: Synthesizing Customized SQL Execution Engines from Natural Language Instructions Using GPT-3 Codex, Technical Report, Cornell, Ithaca. NY, 2022.

[45] I. Trummer, CodexDB: Synthesizing code for query processing from natural language instructions using GPT-3 codex, Proceedings of the VLDB Endowment 15 (2022) 2921–2928. doi:10.14778/3551793.3551841.

[46] X. Zhou, G. Li, C. Chai, J. Feng, A learned query rewrite system using Monte Carlo tree search, Proceedings of the VLDB Endowment 15 (2021) 46–58. doi:10.14778/3485450.3485456.

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

[48] B. Hilprecht, C. Binnig, Zero-shot cost models for out-of-the-box learned cost prediction, Proceedings of the VLDB Endowment 15 (2022) 2361–2374. doi:10.14778/3551793.3551799.

[49] J. Sun, G. Li, An end-to-end learning-based cost estimator, Proceedings of the VLDB Endowment 13 (2019) 307–319. doi:10.14778/3368289.3368296.

[50] S. Hasan, S. Thirumuruganathan, J. Augustine, N. Koudas, G. Das, Deep Learning Models for Selectivity Estimation of Multi-Attribute Queries, in: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 1035–1050. doi:10.1145/3318464.3389741.

[51] R. Heinrich, M. Luthra, H. Kornmayer, C. Binnig, Zero-shot cost models for distributed stream processing, in: Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems, DEBS '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 85–90. doi:10.1145/3524860.3539639.

[52] H. Behr, V. Markl, Z. Kaoudi, Learn What Really Matters: A Learning-to-Rank Approach for ML-based Query Optimization, in: B. König-Ries, S. Scherzinger, W. Lehner, G. Vossen (Eds.), Database Systems for Business, Technology, and the Web 2023, BTW '23, Gesellschaft für Informatik e.V., 2023. doi:10.18420/BTW2023-25.

[53] X. Yu, C. Chai, G. Li, J. Liu, Cost-Based or Learning-Based? A Hybrid Query Optimizer for Query Plan Selection, Proceedings of the VLDB Endowment 15 (2022) 3924–3936. doi:10.14778/3565838.3565846.

[54] L. Cen, R. Marcus, H. Mao, J. Gottschlich, M. Alizadeh, T. Kraska, Learned Garbage Collection, in: Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL @ PLDI '20, ACM, 2020. doi:10.1145/3394450.3397469.

[55] N. Hasabnis, J. Gottschlich, ControlFlag: A self-supervised idiosyncratic pattern detection system for software control structures, in: Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming, MAPS '21, Association for Computing Machinery, New York, NY, USA, 2021, pp. 32–42. doi:10.1145/3460945.3464954.

[56] M. Alam, J. Gottschlich, N. Tatbul, J. Turek, T. Mattson, A. Muzahid, A zero-positive learning approach for diagnosing software performance regressions, in: Proceedings of the 33rd International Conference on Neural Information Processing Systems, NeurIPS '19, Curran Associates Inc., Red Hook, NY, USA, 2019, pp. 11627–11639.

[57] R. Snodgrass, Ergalics: A Natural Science of Computation, Position, University of Arizona, Tucson, AZ, 2010.

[58] S. Currim, R. T. Snodgrass, Y.-K. Suh, R. Zhang, M. W. Johnson, C. Yi, DBMS metrology: Measuring query time, in: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD '13, Association for Computing Machinery, New York, NY, USA, 2013, pp. 421–432. doi:10.1145/2463676.2465331.

[59] H. Massalin, Superoptimizer: A look at the smallest program, ACM SIGARCH Computer Architecture News 15 (1987) 122–126. doi:10.1145/36177.36194.