Query Optimization in the Wild: Realities and Trends

Yuanyuan Tian
Gray Systems Lab, Microsoft
yuanyuantian@microsoft.com

ABSTRACT

For nearly half a century, the core design of query optimizers in industrial database systems has remained remarkably stable, relying on foundational principles from System R and the Volcano/Cascades framework. However, the rise of cloud computing, massive data volumes, and unified data platforms has exposed the limitations of this traditional, monolithic architecture. Taking an industrial perspective, this paper reviews the past and present of query optimization in production systems and identifies the challenges they face today. Then this paper highlights three key trends gaining momentum in the industry that promise to address these challenges. First, a tighter feedback loop between query optimization and query execution is being used to improve the robustness of query performance. Second, the scope of optimization is expanding from a single query to entire workloads through the convergence of query optimization and workload optimization. Third, and perhaps most transformatively, the industry is moving from monolithic designs to composable architectures that foster agility and cross-engine collaboration. Together, these trends chart a clear path toward a more dynamic, holistic, and adaptable future for query optimization in practice.

1 Introduction

Query optimization (QO) is one of the most enduring and formidable challenges in the field of data management. For nearly half a century, since the dawn of relational databases, it has been a cornerstone of academic research, generating a vast body of literature that explores everything from plan enumeration and cost modeling to advanced machine learning based techniques (see surveys [25, 26]). Yet, despite decades of brilliant work, query optimization is far from a solved problem. The gap between theoretical possibilities and production reality remains significant, as industrial database systems often favor stability, predictability, and incremental progress over disruptive, high-risk innovation.

If one were to look under the hood of most modern databases today, they would find a design paradigm that has stood the test of time: a cost-based optimizer, built on the foundational frameworks pioneered by System R [61], Volcano [29], Cascades [28], and Starburst [59]. The longevity of this architecture is a powerful testament to its robustness and the foresight of its creators. However, the ground beneath these systems is shifting. The explosion of data volume, the rise of the cloud, the decoupling of storage and compute, and the convergence of once-disparate data engines into unified lakehouse ecosystems present challenges that this classic architecture was never designed to handle. The need for manual tuning, the brittleness of cost models, and the monolithic nature of traditional optimizers are becoming critical bottlenecks.

This paper takes a distinctly industrial perspective, focusing on the past and present of QO as practiced in real-world, production-level database systems. We will examine the challenges these systems face today and argue that the next wave of innovation will not come from replacing the core optimizer, but from augmenting and evolving it in practical, impactful ways. We highlight three promising and pragmatic trends that are already gaining momentum in the industry, each addressing critical limitations of the traditional QO model.

These trends represent a natural evolution, moving from a narrow, static view of optimization to a more dynamic, holistic, and composable one. First, we will explore the collaboration between QO and query execution (QE) to ensure robust performance. Second, we will discuss the convergence of QO with workload optimization (WO), expanding the scope from a single query to optimizing the entire workload. Finally, we will analyze what is perhaps the most transformative shift: the move away from monolithic, customized optimizers toward a more agile and composable QO architecture, enabling faster innovation and cross-engine collaboration in modern data platforms.

2 QO Evolution: The Past and Present

The evolution of QO is best understood through a historical lens. The history of QO is deeply intertwined with the evolution of relational databases. As shown in Figure 1, over the past five decades, relational systems have progressed through various architectural shifts, from data warehouses and MPP (Massively Parallel Processing) databases to big data platforms, cloud databases, data lakes, and modern lakehouse architectures. QO has evolved alongside these trends. However, interestingly, if you look closely, many of today's optimizers still rely on foundational QO frameworks introduced

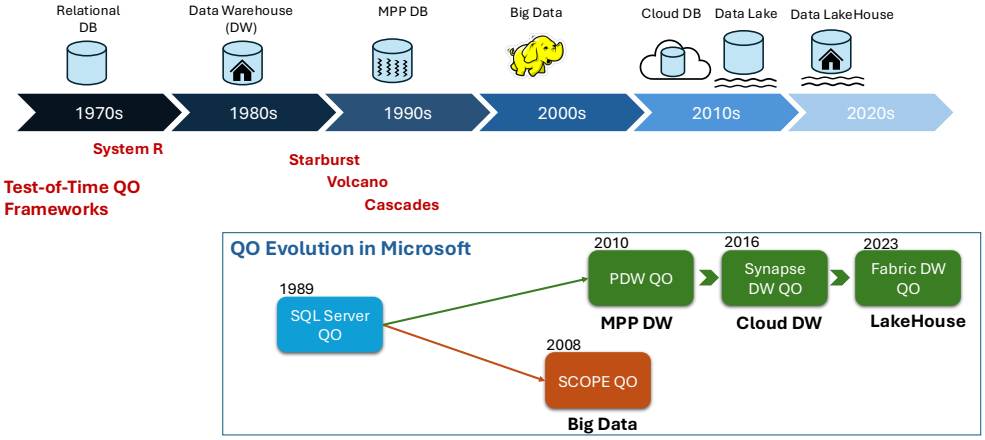

Figure 1: The history of relational databases and QO

decades ago. Four frameworks in particular, System R [61], Starburst [59], Volcano [29], and Cascades [28] have stood the test of time. System R pioneered the work on query optimization. Starburst's optimizer remains to be the QO for IBM Db2 [31] today. Volcano and Cascades introduced extensible and modular architectures that are still widely used in modern database systems [17, 18, 21, 38, 48, 51, 62]. The enduring success of these QO architectures is due to their powerful extensibility and adaptability. Although recent years have seen a surge in new relational query engines, their query optimizers are all derived from the same time-tested architectures, primarily the Cascades/Volcano framework, and thus share significant similarities. Below, we outline the standard Cascades/Volcano-based QO architecture adopted by the majority of today's database systems [17, 18, 21, 38, 48, 51, 62].

Reference QO Architecture: A Cascades/Volcano optimizer generally proceeds through four major phases.

- Parsing/Algebrization: The input SQL query is parsed into an Abstract Syntax Tree (AST) and then converted into an initial logical algebraic tree composed of relational operators such as Join, Project, and Scan.
- Simplification/Normalization: The logical tree is iteratively rewritten into a canonical, semantically equivalent form using transformation rules such as predicate pushdown, join reordering, and projection pruning.
- Cost-Based Exploration: Transformation and implementation rules are applied to explore alternative logical and physical plans, all of which are organized in a MEMO structure that groups logically equivalent expressions; for each group and required physical property, the optimizer recursively computes and memoizes the lowest-cost plan, adding enforcer operators (e.g., Sort, Repartition) when necessary to meet property requirements such as ordering or data distribution.
- Post-Optimization: Additional rewrites and property adjustments are applied to the selected plan, such as adding execution annotations or final physical tweaks, producing the fully optimized execution plan ready for the execution engine.

Implementations of this high-level architecture, however, vary widely. Systems like Spark prioritize rule-based simplification with limited cost-based exploration, while mature engines employ sophisticated cost-based exploration. A common practical strategy in these advanced systems is to break cost-based exploration into sub-stages, each running a full Cascades pass with a subset of rules over a subset of conditions along with a timeout. This strategy helps manage the search space and balance plan quality with compilation time, though the specific stage design highly depends on the workload requirements and system implementation. For instance, SQL Server QO has three stages (OLTP, quick, and full optimization), whereas UQO [20] (the QO for Fabric DW) roughly divides exploration into two stages: single-node optimization followed by distributed planning.

2.1 Example: QO Evolution in Microsoft

To understand the powerful adaptability of the existing QO frameworks, let's use a concrete example to illustrate how an optimizer based on the Cascades framework has evolved to handle new requirements over time (see the lower half of Figure 1).

In 1989, the first version of SQL server [48] was shipped with its Cascades-based optimizer. Over the years, Microsoft has successfully evolved the SQL Server QO across multiple database products within the company.

First, to adapt itself to handle modern analytical workloads, the SQL Server QO was evolved to support column-oriented storage and batch-based query processing [33]. This was achieved by introducing a new index type called *columnstore index* and a new *Columnstore Index Scan* operator, adding new implementation rules, and defining a new physical property to manage "row mode" versus "batch mode", along with with enforcer rules to switch between them.

In 2008, SCOPE [21], an internal big data query engine in Microsoft, forked the SQL Server QO as its base and underwent significant adaptations to suit its new processing environment [60]. First, to support distributed execution, the optimizer was enhanced with new data exchange operators. It also introduced and reasoned about physical properties like partitioning, sorting, and grouping, using enforcer rules to satisfy operator requirements for these properties [70]. Second, to prevent runaway jobs that could consume excessive resources, the team embedded pragmatic safeguards directly into the opti-

mizer. For instance, rules were added to disallow cross products entirely and to limit the results of internal joins to less than 1000x1000 per matching key to prevent cost explosions from small but problematic inputs [60].

Simultaneously, Parallel Data Warehouse or PDW [50], an MPP data warehouse, evolved its optimizer from the SQL Server QO as well. To support distributed processing, PDW's optimization pipeline employs multiple optimizers. A front-end optimizer first generates a search space (MEMO) using the standard SQL Server stack. A Distributed Query Optimizer (DQO) then refines this by adding data movement operators, which break the plan into sub-plans, called "steps". These steps are translated back into SQL and executed by worker nodes, each optimizing its local query fragment independently.

This optimization lineage continued as PDW transitioned into its cloud-based incarnation, Synapse DW [52], which largely reused the multi-optimizer approach. More recently, as Synapse DW was integrated into the Microsoft Fabric lakehouse ecosystem [51], the optimizer evolved again. The new Unified Query Optimizer (UQO) [20] refines the design by merging the multiple optimizers back into a single, cost-based optimizer for generating distributed plans. It achieves this by incorporating distribution-related physical properties directly into the MEMO, along with implementation and enforcer rules that manage data distribution requirements for operators.

3 Is QO a "Solved" Problem?

If today's databases still use QO frameworks from decades ago, does that mean QO is a "solved" problem? Guy Lohman, the QO expert behind Starburst, addressed this very question in his 2014 ACM blog post [36] and again in his 2017 talk at BTW [37], concluding that the answer is no.

The core issue lies in the cost model, which serves as the optimizer's foundation but is built on a series of outdated assumptions (as detailed in Table 1). It assumes that queries are executed one at a time, the costs of different operators can be simply aggregated, predicates are independent of each other, all joins are primary key-foreign key joins, certain types of costs, such as I/O, dominate others, and that a more detailed model enhances accuracy. However, in reality, queries run concurrently, operations interleave, predicates are often correlated, there are many-to-many joins, hardware and database architecture have undergone significant changes over the years, and increased detail introduces more assumptions into the cost model, ultimately making the QO even more brittle. The shift to the cloud has particularly stressed traditional QO. Cost models must now incorporate the complexities of multi-tier storage (SSD, HDD, cloud object stores) and network costs, making them even more brittle.

This view is strongly supported by empirical evidence. A study by Leis et al. [35] empirically evaluated the main components of a classic QO including cardinality estimation, the cost model, and plan enumeration techniques, and came to the same conclusion. This study found that all tested database systems routinely produce large errors in their cardinality estimates. The join size estimators based on the independence assumption are particularly fragile. In addition, the study found that simple cost models were found to be sufficient, as their errors were dwarfed by the errors in cardinality estimation.

4 Practical Trends in QO

This section will discuss three trends in the industry aimed at addressing some of the QO problems in practice.

4.1 Collaboration between QO and QE

Given that query optimization can often be error-prone, the goal of the QO has shifted, from always trying to find the absolute best plan to primarily avoiding really bad ones. As a result, QO and QE now need to work hand-in-hand to ensure consistently good performance.

One common strategy is to build robust query execution engines that are more resilient to suboptimal plans. For example, most modern databases employ techniques like Bloom filters or bitmap filters to pre-filter tables and reduce the size of join inputs [16, 31, 48, 55]. This helps the execution engine tolerate poor join orderings decided by the QO and still deliver acceptable performance. In particular, the authors of [69] proved that the synergistic interaction between SQL Server's bitmap filters, its pull-based execution model, and the Cascades optimizer allows the system to generate *instance-optimal* plans for acyclic joins, thereby achieving results equivalent to those of Yannakakis's algorithm.

Another approach where QO and QE collaborate to ensure query performance is adaptive query optimization. In this strategy (see Figure 2), the QE gathers runtime statistics and based on this feedback, can trigger re-optimization to adjust the execution plan dynamically.

Many modern databases implement some form of adaptive optimization. For instance, the adaptive join feature of SQL server [46] and Oracle [54] can switch from a hash join to a nested loops join at runtime, depending on observed cardinalities. Google BigQuery defaults to a shuffle join but may switch to a broadcast join during execution if the data volume allows [41]. More recently, Databricks introduced adaptive query execution [67], which uses runtime statistics from completed stages to choose between shuffle and broadcast joins, adjust the degree of parallelism, and restart scans with dynamically generated Bloom filters.

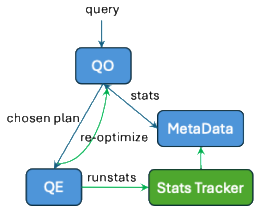

Figure 2: The adaptive QO architecture

4.2 Converging QO with WO

The second trend is the convergence of QO and WO. Traditional QO is memoryless. It optimizes each query without considering past executions. On the other hand,

Table 1: The assumptions for QO vs the reality

	Assumption	Reality
Workload	One query at a time	Queries run simultaneously
Additivity	Costs can be simply added together	Operations interleave
Independence	Predicates are independent	Predicates are often correlated
Subsumption	Joins are PK-FK joins	Many-to-many joins
Weighting	Certain types of cost (I/Os, memory, CPU) dominate	Hardware & architecture have changed dramatically
Model Detail	Detailed models are more accurate	More details add more assumptions!

WO takes in query traces and optimize for the entire workload, such as creating indexes or materialized views. Modern database systems are converging QO and WO by utilizing workload-level insights to optimize individual queries more effectively (see Figure 3).

For example, SQL Server uses the Query Store to persist historical execution plans for a query and monitors query performance to detect plan changes or regressions over time [49]. It can also apply Automatic Plan Correction to fall back to a last known best plan if regression is detected [42]. Similarly, Oracle's SQL Plan Management feature [56] maintains a set of valid plans for each query. The optimizer can then choose from these baseline plans, usually the one with the lowest cost.

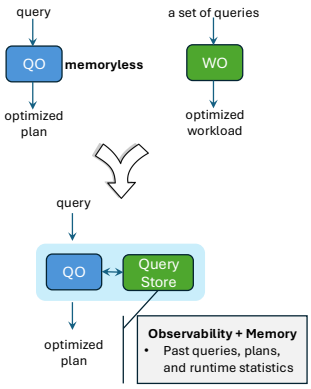

Figure 3: The convergence of QO and WO

4.2.1 About Learned QO

In recent years, there has been a significant increase in the application of Machine Learning (ML) techniques to enhance query optimization, a research area now commonly referred to as "learned QO" [30, 32, 39, 40, 64–66]. The core premise behind all learned QO approaches is that queries are often recurrent and similar, enabling these methods to learn from the past to improve the future. So, essentially all learned QO approaches are embodiments of the convergence of QO and WO.

Despite the rapidly growing body of research, the industry remains hesitant to deploy learned QO techniques in real-world production systems. As discussed in [71], this gap between academia and industry is rooted in several key factors: production systems are far more complex than research prototypes, and they have strict requirements for explainability and debuggability that many ML models cannot meet; furthermore, there is a significant risk of performance regressions when models trained on historical data encounter evolving workloads, and the high cost of training and inference remains a major barrier.

Nevertheless, there are some industrial efforts in applying the learned QO approaches to production systems [43, 44]. *In general, the "pacemaker approach" of using ML to enhance existing QO is often more feasible than the "heart transplant approach" of replacing the QO entirely with a learned model.*

Let's illustrate examples of pacemaker approaches. SQL Server has a number of Intelligent Query Processing (IQP) features [45] where the system learns from its mistakes during runtime and automatically applies corrective measures to improve future performance. For example, the cardinality estimation (CE) Feedback feature [43] observes runtime differences between estimated and actual row counts and automatically adjusts the optimizer's CE model assumptions (fully independent vs. partially correlated vs. fully correlated) for future executions of repeating queries. In a similar vein, its Degree of Parallelism (DoP) Feedback feature learns and enforces the optimal DoP for repeating queries [44]. A recent work [13] improved the SQL Server cost model by using ML to automatically tune the constant parameters in the cost model offline for a target workload. Adapting from techniques in Bao [39], QOAdvisor [53, 68] recommends rule hints for Microsoft Scope QO. It had to make practical adjustments to manage the system complexity, like incremental plan changes, cost-efficient experimentation with a contextual bandit model, and a validation model to prevent regressions. Google BigQuery recently introduced History-Based Optimizations [27], which learn patterns and characteristics from prior query executions to generate efficient execution plans. Key optimizations include join pushdown, semi-join reduction, build/probe side swapping for hash joins, and dynamic adjustment of DoP. Redshift leverages its Automated Materialized Views (AutoMV) feature [15] to boost query performance, employing ML and workload monitoring to automate the creation and management of views. Databricks' recent Predictive Optimization feature [24] uses ML to collect and maintain table statistics in a smart way.

These above methods share a common strategy: they preserve the existing QO with minimal modification, while using ML to find the optimal settings and "knobs" for it to operate under. Furthermore, a successful learned QO implementation typically relies on two critical components: comprehensive history tracking (e.g., query logs, runtime statistics) and robust feedback mechanisms to continuously refine performance.

4.2.2 Related Directions Beyond Learned QO

While not intrinsic to QO, **ML-driven storage optimization** is increasingly used to improve performance. Systems like Databricks (Predictive I/O [22], Liquid Clustering [23]) and Amazon Redshift [14] use ML to automate physical layout decisions, such as clustering keys, sort orders, and data distribution, creating data organizations that the QO can exploit for faster execution. Regarding **Generative AI**, while LLMs have revolutionized user interfaces (e.g., text-to-SQL), their adoption in core QO remains experimental at best. The strict requirements for determinism, low latency, and absolute correctness in plan generation currently conflict with the probabilistic nature and high inference costs of LLMs, preventing their use in the critical path of production QO.

4.3 From Customized to Composable QO

The third major trend is the transition from customized to composable QO. This transition is primarily influenced by two significant, industry-wide developments.

The Convergence of Data Platforms: The first driver is the industry's shift away from siloed data engines toward highly integrated and converged solutions. Microsoft Fabric [47] is a notable example, creating a unified lakehouse where different compute engines operate on the same copy of data in OneLake stored as Delta Parquet tables. Within this ecosystem, engines like Fabric Spark and Fabric DW share the same physical data, have similar architectures and usage patterns, yet differ significantly in optimizer maturity, with DW's QO being far more advanced. This specific combination makes a shared QO far more valuable than in a single engine system where decoupling the QO offers lower ROI.

The Rise of Composable Database Architectures: Concurrently, there has been a fundamental move from monolithic to composable database designs [58], spurred by cloud adoption and the decoupling of storage and compute. This modularity is further enabled by the widespread adoption of open standards such as Parquet [5], Arrow [1], and Substrait [7]. In the Lakehouse ecosystem, this trend is evident in the proliferation of open table formats (Delta Lake [9], Hudi [3], and Iceberg [4]) and interoperability layers like XTable [8] and UniForm [10], alongside unified catalog initiatives such as OpenHouse [11], Polaris [6], and Unity Catalog [12]. In addition, there has been a surge in open-source projects focused on reusable database components. On the QE side, projects like Velox [57] and Datafusion [2] exemplify this, while on the QO side, general-purpose libraries like Calcite [19] and Orca [63] are being developed to unify and share core QO components.

Both Orca and Calcite represent "QO-as-a-Library" design philosophy, where a modular, cost-based optimizer can be embedded within and customized for different host systems. Built upon the Volcano/Cascades paradigm, they each provide a foundation for adding custom rewrite rules and operators. While Orca and Calcite may not directly solve today's core optimizer challenges, their modular and extensible architectures are crucial enablers. By fostering faster innovation, improving engineering efficiency, and reducing time-to-market for new engines, they create the foundation needed to develop and deploy the next generation of QO technologies that can address these modern problems.

Recent work [13] further elevates composability with the proposal of "QO-as-a-Service" (QOaaS) for unified ecosystems like Microsoft Fabric. By externalizing the QO into an independent service that communicates with engines via RPC, QOaaS retains the benefits of QO libraries while enabling new capabilities like independent scaling, workload-aware tuning, and cross-engine optimization. The architecture ensures interoperability by using a standard plan specification like Substrait [7] and establishes a closed-loop system. To achieve this, the QOaaS proposal introduces three key components: a Query Insight Store to capture historical plans and runtime statistics; an External Tuner Plugin to enable data-driven tuning by external processes; and a Config/Action Store to feed optimized configurations back into the core optimizer for continuous improvement.

5 Conclusion and Discussion

In conclusion, the foundational architecture of QO in modern databases remains largely unchanged from the frameworks proposed five decades ago. Yet, as we have discussed, QO is far from a solved problem. The way forward is not to endlessly polish the existing model, but to address its foundational limitations with practical, forward-looking solutions. This paper has highlighted three pragmatic trends gaining momentum in the industry: the collaboration between QO and QE; the convergence of QO with WO; and the transformative shift from monolithic to composable QO architectures.

While feedback loops and workload-level optimization inherently introduce telemetry and storage overheads, production systems mitigate these costs through adaptive sampling (logging only identifying/expensive queries) and asynchronous processing (performing heavy analysis off the critical path). Furthermore, this "tax" is viewed as an essential insurance policy; the cost of storing historical statistics is negligible compared to the resource waste and SLA violations caused by a single catastrophic regression. In modern cloud architectures, this trade-off is increasingly favorable, prioritizing system resilience over raw minimal overhead.

Finally, as noted in [34], a major challenge for the progression of an industry QO is performance regression. No matter how well a method improves average workload performance, a small number of slower queries can be a deal-breaker for adoption. To varying degrees, the three trends highlighted in this paper serve as an antidote to this stagnation. QO and QE collaboration acts as the airbag, resilient enough to handle, and even correcting, QO mistakes at runtime. By merging QO and WO, we move from stateless guessing to stateful learning, allowing the system to remember what works and avoid repeating mistakes (but falling back to safe defaults for unobserved queries). A composable QO service, empowered by a Query Insight Store and pluggable interfaces, creates the infrastructure necessary to test and integrate innovations safely, making the progression of QO both safer and faster.

6 References

[1] Apache arrow. https://arrow.apache.org/, 2025.

[2] Apache datafusion. https://datafusion.apache.org/, 2025.

[3] Apache hudi. https://hudi.apache.org, 2025.

[4] Apache iceberg. https://iceberg.apache.org, 2025.

[5] Apache parquet. https://parquet.apache.org/, 2025.

[6] Apache polaris (incubating). https://polaris.apache.org, 2025.

[7] Apache substait. https://substrait.io, 2025.

[8] Apache xtable. https://xtable.apache.org, 2025.

[9] Delta lake. https://delta.io, 2025.

[10] Delta uniform: a universal format for lakehouse interoperability. https://www.databricks.com/blog/delta-uniform-universal-format-lakehouse-interoperability, 2025.

[11] Openhouse. https://www.openhousedb.org, 2025.

[12] Unity catalog. https://www.unitycatalog.io, 2025.

[13] R. Alotaibi, Y. Tian, S. Grafberger, J. Camacho-Rodriguez, N. Bruno, S. M. Brian Kroth, A. Agrawal, M. Behera, A. Gosalia, C. Galindo-Legaria, M. Joshi, M. Potocnik, B. Sezgin, X. Li, and C. Curino. Towards Query Optimizer as a Service (QOaaS) in a Unified LakeHouse Ecosystem: Can One QO Rule Them All? In *CIDR '25'*, 2025.

[14] Amazon. Automatic table optimization. https://docs.aws.amazon.com/redshift/latest/dg/t_Creating_tables.html, 2020.

[15] Amazon. Automated materialized views. https://docs.aws.amazon.com/redshift/latest/dg/materialized-view-auto-mv.html, 2025.

[16] Apache Spark. Spark: Unified Engine for Large-scale Data Analytics. https://spark.apache.org, 2025.

[17] D. F. Bacon, N. Bales, N. Bruno, B. F. Cooper, A. Dickinson, A. Fikes, C. Fraser, A. Gubarev, M. Joshi, E. Kogan, A. Lloyd, S. Melnik, R. Rao, D. Shue, C. Taylor, M. van der Holst, and D. Woodford. Spanner: Becoming a SQL system. In *ACM SIGMOD*, 2017.

[18] E. Begoli, J. Camacho-Rodríguez, J. Hyde, M. J. Mior, and D. Lemire. Apache Calcite: A Foundational Framework for Optimized Query Processing over Heterogeneous Data Sources. In *ACM SIGMOD*, 2018.

[19] E. Begoli, J. Camacho-Rodríguez, J. Hyde, M. J. Mior, and D. Lemire. Apache Calcite: A Foundational Framework for Optimized Query Processing over Heterogeneous Data Sources. In *ACM SIGMOD*, 2018.

[20] N. Bruno, C. Galindo-Legaria, M. Joshi, E. Calvo Vargas, K. Mahapatra, S. Ravindran, G. Chen, E. Cervantes Juárez, and B. Sezgin. Unified query optimization in the fabric data warehouse. In *ACM SIGMOD*, 2024.

[21] R. Chaiken, B. Jenkins, P.-r. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. In *PVLDB*, 2008.

[22] Databricks. Announcing the General Availability of Predictive I/O for Reads. https://www.databricks.com/blog/announcing-general-availability-predictive-io-reads.html, 2024.

[23] Databricks. Announcing Automatic Liquid Clustering. https://www.databricks.com/blog/announcing-automatic-liquid-clustering, 2025.

[24] Databricks. Introducing predictive optimization: Faster queries, cheaper storage, no sweat. https://www.databricks.com/blog/introducing-predictive-optimization-faster-queries-cheaper-storage, 2025.

[25] B. Ding, V. Narasayya, and S. Chaudhuri. Extensible query optimizers in practice. *Foundations and Trends in Databases*, 14(3-4):186–402, 2024.

[26] B. Ding, R. Zhu, and J. Zhou. Learned query optimizers. *Foundations and Trends in Databases*, 13(4):250–310, 2024.

[27] Google. Get up to 100x query performance improvement with bigquery history-based optimizations. https://cloud.google.com/blog/products/data-analytics/new-bigquery-history-based-optimizations-speed-query-performance, 2025.

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

[29] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE '93*, page 209–218, 1993.

[30] B. Hilprecht, A. Schmidt, M. Kulessa, A. Molina, K. Kersting, and C. Binnig. Deepdb: learn from data, not from queries! *Proc. VLDB Endow.*, 13(7):992–1005, 2020.

[31] IBM. IBM Db2. https://www.ibm.com/products/db2, 2025.

[32] A. Kipf, T. Kipf, B. Radke, V. Leis, P. A. Boncz, and A. Kemper. Learned cardinalities: Estimating correlated joins with deep learning. 2019.

[33] P.-r. Larson, C. Clinciu, E. N. Hanson, A. Oks, S. L. Price, S. Rangarajan, A. Surna, and Q. Zhou. Sql server column store indexes. In *SIGMOD '11*, page 1177–1184, 2011.

[34] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. Still asking: How good are query optimizers, really? *Proc. VLDB Endow.*, 18(12):5531–5536, 2025.

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

[36] G. Lohman. Is query optimization a solved problem? ACM Blog, 2014.

[37] G. Lohman. Query optimization: Are we there yet? In *Proceedings of BTW 2017 (Datenbanksysteme für Business, Technologie und Web)*, BTW, 2017.

[38] Z. Lyu, H. H. Zhang, G. Xiong, G. Guo, H. Wang, J. Chen, A. Praveen, Y. Yang, X. Gao, A. Wang, et al. Greenplum: a hybrid database for transactional and analytical workloads. In *ACM SIGMOD*, 2021.

[39] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska. Bao: Making learned query optimization practical. *SIGMOD Rec.*, 51(1):6–13, June 2022.

[40] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. Neo: a learned query optimizer. *Proc. VLDB Endow.*, 12(11):1705–1718, July 2019.

[41] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, T. Vassilakis, H. Ahmadi, D. Delorey, S. Min, M. Pasumansky, and J. Shute. Dremel: a decade of interactive sql analysis at web scale. *Proc. VLDB Endow.*, 13(12):3461–3472, 2020.

[42] Microsoft. Automatic tuning). http://learn.microsoft.com/en-us/sql/relational-databases/automatic-tuning/automatic-tuning?view=sql-server-ver17, 2025.

[43] Microsoft. Cardinality estimation (CE) feedback. https://learn.microsoft.com/en-us/sql/relational-databases/performance/intelligent-query-processing-cardinality-estimation-feedback, 2025.

[44] Microsoft. Degree of parallelism (DOP) feedback. https://learn.microsoft.com/en-us/sql/relational-databases/performance/intelligent-query-processing-degree-parallelism-feedback, 2025.

[45] Microsoft. Intelligent query processing in SQL databases. https://learn.microsoft.com/en-us/sql/relational-databases/performance/intelligent-query-processing?view=sql-server-ver17, 2025.

[46] Microsoft. Join (SQL Server). https://learn.microsoft.com/en-us/sql/relational-databases/performance/joins?view=sql-server-ver17, 2025.

[47] Microsoft. Microsoft fabric. https://www.microsoft.com/en-us/microsoft-fabric, 2025.

[48] Microsoft. Microsoft SQL Server. https://microsoft.fandom.com/wiki/Microsoft_SQL_Server, 2025.

[49] Microsoft. Monitor performance by using the Query Store. https://learn.microsoft.com/en-us/sql/relational-databases/performance/monitoring-performance-by-using-the-query-store?view=sql-server-ver17, 2025.

[50] Microsoft. Parallel Data Warehouse. https://learn.microsoft.com/en-us/sql/analytics-platform-system/parallel-data-warehouse-overview, 2025.

[51] Microsoft. Synapse Data Warehouse in Microsoft Fabric. https://learn.microsoft.com/en-us/fabric/data-warehouse/, 2025.

[52] Microsoft. Synapse Data Warehouse in Microsoft Fabric. https://azure.microsoft.com/en-us/products/synapse-analytics, 2025.

[53] P. Negi, M. Interlandi, R. Marcus, M. Alizadeh, T. Kraska, M. Friedman, and A. Jindal. Steering query optimizers: A practical take on big data workloads. In *SIGMOD*, page 2557–2569, 2021.

[54] Oracle. The Optimizer In Oracle Database 19c. Technical report, Oracle, 2019.

[55] Oracle. Oracle Database. https://www.oracle.com/database, 2025.

[56] Oracle. Overview of SQL Plan Management. https://docs.oracle.com/en/database/oracle/oracle-database/19/tgsql/overview-of-sql-plan-management.html, 2025.

[57] P. Pedreira, O. Erling, M. Basmanova, K. Wilfong, L. Sakka, K. Pai, W. He, and B. Chattopadhyay. Velox: meta's unified execution engine. In *PVLDB*, 2022.

[58] P. Pedreira, O. Erling, K. Karanasos, S. Schneider, W. McKinney, S. R. Valluri, M. Zait, and J. Nadeau. The composable data management system manifesto. In *PVLDB*, 2023.

[59] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in starburst. In *SIGMOD '92*, page 39–48, 1992.

[60] C. Power, H. Patel, A. Jindal, J. Leeka, B. Jenkins, M. Rys, E. Triou, D. Zhu, L. Katahanas, C. B. Talapady, J. Rowe, F. Zhang, R. Draves, I. Santa, and A. Kumar. The cosmos big data platform at microsoft: Over a decade of progress and a decade to look forward. *Proc. VLDB Endow.*, 14(12):3148–3161, 2021.

[61] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD '79*, page 23–34, 1979.

[62] M. A. Soliman, L. Antova, V. Raghavan, A. El-Helw, Z. Gu, E. Shen, G. C. Caragea, C. Garcia-Alvarado, F. Rahman, M. Petropoulos, et al. Orca: a modular query optimizer architecture for big data. In *ACM SIGMOD*, 2014.

[63] M. A. Soliman, L. Antova, V. Raghavan, A. El-Helw, Z. Gu, E. Shen, G. C. Caragea, C. Garcia-Alvarado, F. Rahman, M. Petropoulos, et al. Orca: a modular query optimizer architecture for big data. In *ACM SIGMOD*, 2014.

[64] I. Trummer, J. Wang, Z. Wei, D. Maram, S. Moseley, S. Jo, J. Antonakakis, and A. Rayabhari. Skinnerdb: Regret-bounded query evaluation via reinforcement learning. *ACM Trans. Database Syst.*, 46(3), 2021.

[65] L. Woltmann, J. Thiessat, C. Hartmann, D. Habich, and W. Lehner. Fastgres: Making learned query optimizer hinting effective. *Proc. VLDB Endow.*, 16(11):3310–3322, 2023.

[66] P. Wu and G. Cong. A unified deep model of learning from both data and queries for cardinality estimation. In *SIGMOD '21*, page 2009–2022, 2021.

[67] M. Xue, Y. Bu, A. Somani, W. Fan, Z. Liu, S. Chen, H. van Hovell, B. Samwel, M. Mokhtar, R. Korlapati, A. Lam, Y. Ma, V. Ercegovac, J. Li, A. Behm, Y. Li, X. Li, S. Krishnamurthy, A. Shukla, M. Petropoulos, S. Paranjpye, R. Xin, and M. Zaharia. Adaptive and robust query execution for lakehouses at scale. *Proc. VLDB Endow.*, page 3947–3959, 2024.

[68] W. Zhang, M. Interlandi, P. Mineiro, S. Qiao, N. Ghazanfari, K. Lie, M. Friedman, R. Hosn, H. Patel, and A. Jindal. Deploying a steered query optimizer in production at microsoft. In *SIGMOD*, page 2299–2311, 2022.

[69] H. Zhao, Y. Tian, R. Alotaibi, B. Ding, N. Bruno, J. Camacho-Rodriguez, V. Papadimos, E. C. Juarez, C. Galindo-Legaria, and C. Curino. I Can't Believe It's Not Yannakakis: Pragmatic Bitmap Filters in Microsoft SQL Server. In *CIDR '26'*, 2026.

[70] J. Zhou, P.-A. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the scope optimizer. In *ICDE '10*, pages 1060–1071, 2010.

[71] Y. Zhu, Y. Tian, J. Cahoon, S. Krishnan, A. Agarwal, R. Alotaibi, J. Camacho-Rodriguez, B. Chundatt, A. Chung, N. Dutta, A. Fogarty, A. Gruenheid, B. Haynes, M. Interlandi, M. Iyer, N. Jurgens, S. Khushalani, B. Kroth, M. Kumar, J. Leeka, S. Matusevych, M. Mittal, A. Mueller, K. Muthyala, H. Nagulapalli, Y. Park, H. Patel, A. Pavlenko, O. Poppe, S. Ravindran, K. Saur, R. Sen, S. Suh, A. Tarafdar, K. Waghray, D. Wang, C. Curino, and R. Ramakrishnan. Towards building autonomous data services on azure. In *SIGMOD '23*, page 217–224, 2023.