# AI-DRIVEN DATABASE QUERY OPTIMIZATION

## From Cost Models to Neural Optimizers

### A Practical Engineering Guide

PostgreSQL Internals

ML for Cardinality & Cost Estimation

Learned Indexes

Reinforcement Learning Planners

Production Deployment

MARTIN ELLWOOD

# AI-Driven Database Query Optimization

## From Cost Models to Neural Optimizers — A Practical Engineering Guide

Steve Publications

This book is available at
https://leanpub.com/ai-drivendatabasequeryoptimization

This version was published on 2026-09-17

# Contents

# From Cost Models to Neural Optimizers — A Practical Engineering Guide

This book takes you from the fundamentals of traditional database query optimization through the frontier of AI-driven techniques that are reshaping how databases execute queries. You will learn how to understand PostgreSQL's optimizer internals, build machine learning models for cardinality estimation and cost prediction, design learned index structures, implement reinforcement learning-based query planners, and deploy these systems in production with rigor, safety, and measurable performance gains. If you are a software engineer, database engineer, ML engineer, or systems architect working with relational databases at scale, this is the guide you need to build or evaluate AI-assisted query optimization systems.

# Introduction

A slow query can bring an entire system to its knees. You have probably seen it firsthand: a dashboard freezes for thirty seconds while a poorly planned join spins through billions of row combinations, the error budget evaporates, users complain, and the on-call engineer races to kill the runaway process. Behind every such incident lies a silent decision made milliseconds earlier by a tiny piece of database software known as the query optimizer.

The query optimizer is the brain of any relational database. When you submit a SQL statement, the optimizer translates your declarative intent into a concrete execution plan. It decides which table to scan first, which indexes to use, which join algorithm to apply at each step, how much memory to allocate, and whether to parallelize. A good plan might execute in two milliseconds. A bad plan for the same query might take twenty minutes. The optimizer does not know the right answer. It guesses, using statistics, cost models, and heuristics, and that guess determines everything.

For decades, query optimizers have relied on handcrafted techniques. Histograms estimate the number of rows that satisfy a condition. Cost formulas weigh CPU time against disk I/O. Dynamic programming algorithms search for the best join ordering. Genetic algorithms help when the search space gets too large. These methods are battle-tested, explainable, and generally reliable. They are also fundamentally limited. They make simplifying assumptions that real-world data violates. They treat each query in isolation. They cannot learn from past mistakes or adapt to evolving workloads.

Ten years ago, the database research community began asking a provocative question: what if we replace some of these handcrafted components with machine learning models? What if we could train a model to estimate join cardinalities more accurately than a histogram, or to predict query execution times more faithfully than a cost formula, or even to explore the space of query plans like a reinforcement learning agent learning to play a game?

The field has exploded. Since 2015, hundreds of papers have explored applying machine learning to every stage of query optimization. Some approaches have shown remarkable promise: neural networks that capture complex data

correlations, reinforcement learning optimizers that discover plans the traditional optimizer misses, large language models that reason about query structure in entirely new ways. Others have stumbled on practical barriers: training overhead, inference latency, brittleness under workload shifts, the risk of a learned model choosing a catastrophically bad plan.

This book is not a survey of those papers. You can find surveys that list and summarize them. This is an engineering guide. It starts from first principles: how traditional optimizers actually work, why they work the way they do, and where their limitations come from. It then walks you through building ML-driven alternatives and augmentations, step by step, with code that you can run, modify, and extend. It ends with the hard questions that only matter when you are responsible for production systems: how do you deploy a learned optimizer safely, how do you know it is actually helping, and what do you do when it goes wrong.

By the time you finish this book, you will be able to read and critique research papers in the field with a critical eye. You will understand the mathematical foundations behind learned cardinality estimation, cost models, and plan selection. You will have built end-to-end systems that integrate machine learning into query optimization pipelines. You will know which techniques have proven practical value in production and which remain primarily academic. And you will have a realistic understanding of both the power and the limits of AI in database optimization.

## What This Book Covers

We organize the material in three progressive parts.

The first part establishes the foundations. We explore how traditional database query optimizers work, using PostgreSQL as our primary reference. You will learn about SQL parsing, logical and physical query plans, relational algebra, cardinality estimation using histograms, cost-based optimization, join ordering algorithms, indexing strategies, and the execution engine that turns plans into results. We compare PostgreSQL with MySQL, DuckDB, ClickHouse, and other significant systems so you understand the broader landscape. This background is essential: you cannot intelligently improve or replace an optimizer component if you do not understand what it does, why it was designed that way, and what constraints it operates under.

The second part introduces machine learning into the optimization pipeline. We start with learned cardinality estimation, the most mature and widely studied application of ML to query optimization. You will learn how to build random forest estimators, multi-set convolutional networks, and neural density estimators like NeuroCard. We then move to learned cost models that predict query execution time, learned index structures that replace or augment traditional B-trees, and neural query optimizers that use graph neural networks and reinforcement learning to select execution plans. Throughout, we use PostgreSQL as the practical anchor but discuss implementations and approaches across multiple database systems. We include complete, runnable code examples: data generation scripts, model training pipelines, integration with database planners, and benchmarking frameworks.

The third part deals with production deployment and advanced topics. We cover workload-aware optimization that adapts to changing query patterns, automatic index and materialized view selection, adaptive query optimization that adjusts plans during execution, and the emerging role of large language models in query reasoning and optimization. We build an end-to-end AI-assisted optimizer system that integrates multiple learned components. We establish rigorous evaluation methodology so you can measure whether your AI-driven approaches actually improve performance. Finally, we tackle the operational challenges of production deployment: model serving, latency, retraining, drift detection, observability, safety mechanisms, and rollback procedures.

## Who This Book Is For

This book assumes you are already comfortable with SQL and relational databases. You should know how to write and read query plans, understand basic indexing concepts, and have some experience with PostgreSQL or another major database system. You should also have programming proficiency, preferably in Python, and familiarity with basic machine learning concepts: supervised learning, neural networks, loss functions, and evaluation metrics.

If you are a database engineer or DBA, this book will give you a deep understanding of optimizer internals and practical techniques for tuning and improving query performance. If you are a software engineer working with databases at scale, you will learn how to build ML-assisted tools that enhance

your database performance. If you are a machine learning engineer interested in systems applications, you will see how to apply ML techniques to a concrete, high-stakes domain with rigorous evaluation criteria. If you are a systems architect evaluating AI-driven database solutions, you will have the knowledge to separate genuine advances from hype.

## How to Use This Book

Read the book in order if you want a complete foundation. The early chapters on traditional optimization provide context that makes the later ML chapters much more meaningful. However, if you are already comfortable with query optimizer internals, you can jump to the ML chapters that interest you most. Each chapter is self-contained in its treatment of the topic, though some cross-references to earlier material are inevitable.

Code examples are designed to be practical. They are complete enough to run as-is (with documented prerequisites) and include annotations explaining the key design decisions. You should experiment with the code, modify it, and try applying similar approaches to your own schemas and workloads. The code repositories referenced in the book will be kept available where open source licenses permit.

Throughout the book, we distinguish clearly between what is established practice, what is emerging but promising, and what is purely experimental. We cite primary sources: peer-reviewed papers, database documentation, official benchmarks, and source code repositories. Where there is genuine disagreement in the research community, we present multiple perspectives with supporting evidence. We do not shy away from limitations, failures, or cases where traditional approaches remain superior.

## Why PostgreSQL?

PostgreSQL serves as our primary practical database for three reasons. First, it is the most widely used open-source relational database with a sophisticated cost-based optimizer that rivals commercial systems. Second, it is extensively documented, and its source code is freely available, making it an ideal learning vehicle for understanding optimizer internals. Third, the ML-for-databases research community has frequently used PostgreSQL as the evaluation platform, so a large body of published work and open-source code builds on it.

That said, the techniques discussed in this book apply broadly. Many concepts transfer directly to MySQL, MariaDB, SQLite, Oracle, SQL Server, and cloud-native databases like Snowflake and BigQuery. Where the optimizer design differs significantly between systems, we explain the differences and their implications for ML-driven optimization.

## Acknowledgments and Limitations

The field of AI-driven query optimization evolves rapidly. Papers published today may be superseded within a year. Production systems adopt and adapt research at different paces. Some approaches that look promising on paper fail in production; some that seem limited on paper find niche applications where they excel. This book captures the state of the field as of mid-2026, drawing on research, documentation, and real-world experience accumulated up to that point.

We make an effort to verify factual claims about software versions, APIs, benchmarks, and project capabilities against primary sources. However, software changes, projects may become abandoned or transformed, and benchmarks may be re-evaluated. If you find information that is outdated or incorrect, we encourage you to report it so future editions can be corrected.

We also cannot cover everything. The literature on ML for databases is vast and growing. We focus on query optimization specifically; related topics like automatic database tuning, workload classification, data layout optimization, and approximate query processing receive attention where they intersect with query optimization but are not treated as comprehensively. We hope this book serves as both a definitive guide to the core topic and a gateway to the broader field.

Let us begin.

# Chapter 1: The Problem of Query Optimization

## The Cost of Bad Query Plans

A bad query plan is rarely the fault of the person who wrote the SQL. More often, it is the result of a subtle interaction between data distribution, query structure, and optimizer assumptions that produces a plan that is dramatically slower than alternatives the optimizer considered but rejected.

Consider this real-world scenario from an e-commerce platform. A reporting query joins five tables: orders, order_items, products, categories, and users. On a Monday morning, the query executes in 3.2 seconds. On Tuesday morning, after a routine database maintenance window that included running ANALYZE to update statistics, the same query takes 17 minutes. The data has not changed significantly. The query has not changed. Only the statistics have changed, and that change caused the optimizer to select a different plan.

Specifically, the updated statistics altered the estimated cardinality of one intermediate join. Before the maintenance, the optimizer estimated that the join between orders and order_items would produce about 500,000 rows, leading it to choose a hash join that built a hash table in memory. After the maintenance, a more accurate (but unlucky) estimate suggested 2 million rows, so the optimizer chose a nested loop join with an index scan on order_items. The nested loop worked fine for the first few thousand rows but then degraded catastrophically as the outer table grew, resulting in billions of index lookups. The query did not fail. It just ran for seventeen minutes until someone noticed and killed it.

This pattern is not rare. Viktor Leis and colleagues from the Technical University of Munich documented a systematic version of this problem in their 2015 paper “How Good Are Query Optimizers, Really?” [1]. They created the Join Order Benchmark, consisting of 113 complex multi-join queries run against the IMDB dataset. On average, PostgreSQL selected plans that were about 3 times slower than the optimal plans they found through exhaustive search. For

some queries, the gap was 100x or more. These gaps did not arise from bugs in PostgreSQL. They arose from the inherent difficulty of optimization under uncertainty.

The optimizer makes decisions with incomplete information. It cannot execute every possible plan and compare their actual execution times; that would be prohibitively expensive, especially for queries that might take hours. Instead, it estimates. It estimates how many rows each operation will produce. It estimates the cost of scanning each table with or without each index. It estimates whether a join will fit in memory or spill to disk. All of these estimates feed into a cost model, and the optimizer selects the plan with the lowest estimated cost.

The problem is that all of these estimates can be wrong. Sometimes slightly wrong, sometimes dramatically wrong. When estimates are off by a small margin, the optimizer still picks a good plan. When they are off by orders of magnitude, it can pick a disaster.

## Why Query Optimization Is NP-Hard

At the heart of query optimization lies the join ordering problem. Given a query that joins multiple tables in which order should the joins be performed?

For a simple join of three tables A, B, and C, there are only six possible orders: (A join B) join C, (A join C) join B, (B join A) join C, (B join C) join A, (C join A) join B, and (C join B) join A. For each order, there are multiple join algorithms (nested loop, hash join, merge join), multiple scan strategies for each table (sequential scan, various index scans), and decisions about memory allocation and parallelization. Even for three tables, the search space is substantial.

For n tables, the number of possible left-deep join trees alone is n factorial, which grows as:

- n = 5: 120 possible orders
- n = 10: 3,628,800 possible orders
- n = 15: over 1.3 trillion possible orders

This combinatorial explosion makes exhaustive search impractical for queries with more than about ten to twelve joins. The problem was proven NP-hard by Ibaraki and Kameda in 1981 [2], meaning that there is no known

algorithm that can always find the optimal join order in polynomial time (unless P equals NP, which most researchers believe is false).

Modern optimizers use dynamic programming to solve the join ordering problem for left-deep trees in time that grows exponentially but with a smaller constant than brute force. For example, the classic approach used by System R, the first commercial optimizer to use dynamic programming, has a time complexity of O(n * 2^n) [3]. For a query with 12 joins, this is about 50 million subproblems, which is manageable on modern hardware. But at some point, the planner must stop searching.

PostgreSQL uses a configurable threshold called geqo_threshold, which defaults to 12. For queries involving fewer than 12 FROM items, PostgreSQL uses exhaustive dynamic programming to find the optimal left-deep join order. For queries with 12 or more, it switches to GEQO, the Genetic Query Optimizer, which uses evolutionary algorithms to search for a good join order without guaranteeing optimality.

The NP-hardness of join ordering is not just a theoretical curiosity. It directly shapes the optimizer architecture. Because the optimizer cannot explore the entire space, it must rely on estimates to prune the search. If cardinality estimates are inaccurate, the pruning can eliminate the truly optimal plan early, leaving the optimizer with a good-enough plan that is still significantly slower.

## The Gap Between Theoretical Optimality and Practical Constraints

Even if the optimizer could evaluate every possible plan, several practical constraints prevent it from guaranteeing optimality.

First, the cost model is approximate. PostgreSQL's cost model assigns abstract cost values to operations. A sequential page read costs seq_page_cost, which defaults to 1.0. A CPU operation to process one tuple costs cpu_tuple_cost, which defaults to 0.01. These values are dimensionless – they are not measured in milliseconds. They represent relative costs calibrated so the optimizer can compare plans. The problem is that these costs do not capture real-world hardware behavior accurately. They do not account for CPU cache effects, branch prediction, NUMA topology, or storage tier

differences. A plan that looks cheaper in abstract cost units may actually be slower when executed on specific hardware [4].

Second, cardinality estimates are based on limited statistics. PostgreSQL collects statistics about each column: the number of distinct values, a histogram of value distribution, and a list of most common values. These statistics are sampled randomly from the table, not computed over the entire data set. They are updated lazily, typically by autovacuum when enough rows have changed. When statistics are stale, estimates are inaccurate. Even when statistics are fresh, they are inherently limited. Histograms with 100 buckets cannot capture fine-grained distribution details. Most common value lists with 100 entries cannot represent all skew. Column statistics do not capture correlations between columns by default.

Third, the optimizer has a time limit. Planning a query must be fast. If a query takes 10 milliseconds to execute, spending 5 seconds to plan it is absurd. In practice, planning time is typically constrained to a small fraction of expected execution time. This means the optimizer must make trade-offs: it may choose a simpler plan search strategy, reduce the depth of exploration, or use coarser estimates to stay within the time budget. For ad-hoc analytics queries that run for minutes, spending a second or two on planning is acceptable. For OLTP queries that execute in microseconds, planning overhead must be negligible.

Fourth, the optimizer cannot account for runtime dynamics. It does not know that another query is currently scanning the same table and competing for I/O bandwidth. It does not know that the hash table for a join will spill to disk because the memory allocation turned out to be too small. It does not know that an index will cause page splitting that slows down concurrent updates. All of these factors affect actual performance but are invisible to the planner.

These constraints mean that the optimizer is always working with an incomplete and imperfect view of the optimization landscape. Machine learning can potentially help in several ways: by learning more accurate cardinality estimates from actual execution data, by learning cost models that better reflect real hardware behavior, by learning to prioritize exploration in the most promising regions of the search space, and by learning to adapt to changing workloads and data distributions. But none of these approaches can eliminate the fundamental uncertainty that makes optimization hard. They can only reduce it.

## A Motivating Example: Cardinality Estimation Failure in Multi-Join Queries

To understand how estimation errors propagate and cause real problems, let us walk through a concrete example based on the Join Order Benchmark. Consider a query that joins six tables from the IMDB database:

```
SELECT COUNT(*)
FROM title AS t
JOIN cast_info AS ci ON t.id = ci.movie_id
JOIN movie_info AS mi ON t.id = mi.movie_id
JOIN movie_keyword AS mk ON t.id = mk.movie_id
JOIN name AS n ON ci.person_id = n.id
JOIN title_type AS tt ON t.id = tt.id
WHERE mi.info LIKE '%Los Angeles%'
AND mk.keyword LIKE '%sequel%'
AND n.gender = 'm'
```

This query is designed to find male actors who appeared in sequels produced in Los Angeles. It is structurally simple: a set of joins with three filter predicates. But the joins span tables of very different sizes and selectivities:

- title: approximately 6 million rows
- cast_info: approximately 100 million rows
- movie_info: approximately 50 million rows
- movie_keyword: approximately 40 million rows
- name: approximately 10 million rows
- title_type: approximately 20 rows

The optimal join order depends critically on the selectivity of each filter. The filter on title_type is extremely selective and reduces the result to a tiny subset of titles. If the optimizer joins title_type first, the intermediate result is small, and subsequent joins are cheap. If it delays title_type, the intermediate results grow large, and the final join becomes expensive.

Now imagine that the histogram for mi.info is coarse, with only 100 buckets, and the pattern '%Los Angeles%' falls in a bucket whose boundaries do not align well with the actual distribution of values. PostgreSQL might estimate that 5% of movie_info rows satisfy the filter, when the actual selectivity is 0.1%.

This is an estimation error of 50x. When multiplied across joins, this error compounds.

In the original Join Order Benchmark study, such queries frequently caused the optimizer to choose plans that were dramatically suboptimal. Leis et al. documented cases where cardinality estimation errors of 100x or 1000x led to join order choices that increased execution time by factors of 10, 50, or even 1000. The optimizer was not broken. It was simply optimizing the wrong thing: the estimated cost based on faulty cardinality estimates, rather than the actual cost.

This is precisely the gap that machine learning approaches target. If we could train a model to learn the actual distribution of values in mi.info, including correlations between mi.info, mk.keyword, and n.gender, we could produce more accurate estimates and guide the optimizer toward better plans. This is the fundamental motivation behind learned cardinality estimation, which we explore in depth in later chapters.

## Overview of the Optimizer Pipeline and Where AI Can Intervene

A modern query optimizer consists of several stages. Each stage is a candidate for ML-driven enhancement or replacement:

1. SQL Parsing and Analysis: The optimizer takes raw SQL, parses it into a parse tree, resolves table and column references, and produces a logical query tree. This stage is largely rule-based and well-understood. Recent work explores using large language models to assist with query rewriting, hint generation, and natural language to SQL translation, but parsing itself is not a major ML target.
2. Logical Optimization: The optimizer applies algebraic transformations to the logical query tree: pushing selections and projections down toward the leaves, eliminating redundant operations, flattening unnest operations, and applying other equivalence rules. Some of these transformations are deterministic; others involve choices. ML could potentially learn which transformations are most beneficial for specific query patterns and workloads.

3. Cardinality Estimation: For each subquery in the plan, the optimizer estimates the number of rows it will produce. This is the most studied ML application area. Learned cardinality estimators can dramatically improve estimate accuracy, especially for correlated joins and skewed data distributions. We cover this extensively in Chapters 4 and 11.
4. Plan Enumeration: The optimizer generates candidate execution plans by exploring different join orders, join algorithms, and scan strategies. This is where the NP-hard join ordering problem lives. ML can help by learning to guide the search toward promising plans, by learning cost functions that prune the search more effectively, or by learning to select plans directly without explicit enumeration. We cover these approaches in Chapters 10, 14, and 18.
5. Cost Modeling: For each candidate plan, the optimizer estimates its cost using a cost model that combines CPU time, I/O time, and memory usage. ML can learn more accurate cost models that capture hardware-specific behavior, runtime dynamics, and workload interactions. We cover this in Chapter 12.
6. Plan Selection: The optimizer selects the plan with the lowest estimated cost. ML can augment this decision by learning to predict which plans will perform well based on historical execution data, or by using reinforcement learning to explore alternative plans and learn from their outcomes.
7. Index Selection and Physical Design: The optimizer uses existing indexes but does not create new ones. ML can recommend indexes that would improve the workload, either reactively by analyzing query patterns or proactively by simulating the impact of candidate indexes. We cover this in Chapter 16.
8. Execution and Adaptive Optimization: During execution, the optimizer can receive feedback about actual performance. Adaptive query optimization uses this feedback to adjust plans in-flight or to re-optimize on subsequent executions. ML can enhance adaptivity by learning from feedback to improve future plans. We cover this in Chapter 17.

The key insight is that ML is not a monolithic replacement for the optimizer. It is a collection of techniques that can improve individual components within the existing optimization pipeline. Some approaches augment traditional techniques: a learned cardinality estimator feeds estimates into the existing cost model. Others replace components entirely: a neural network predicts plan execution time instead of using handcrafted cost formulas. The most

practical approaches are usually hybrid: they use ML where it adds the most value and fall back on traditional techniques elsewhere.

## Scope and Roadmap of This Book

This book is organized to take you from foundations to advanced practice.

Chapters 2 through 9 cover traditional query optimization. We use PostgreSQL as our primary example because it is well-documented, widely used, and representative of the cost-based optimization approach used by most major databases. You will learn how to read and interpret EXPLAIN output, how PostgreSQL estimates cardinalities, how it searches for optimal join orders, how it evaluates different indexes and join algorithms, and how plans are actually executed. This material is essential context for understanding what ML techniques are improving and why.

Chapters 10 through 17 cover the core ML techniques for query optimization. Chapter 10 establishes the ML foundations: supervised learning, feature engineering, evaluation metrics. Chapter 11 covers learned cardinality estimation in depth, including random forest approaches, multi-set convolutional networks, neural density estimators, and loss functions designed for optimization quality. Chapter 12 covers learned cost models. Chapter 13 covers learned indexes. Chapter 14 covers neural query optimizers and plan selection, including graph neural networks and reinforcement learning. Chapter 15 covers workload-aware optimization. Chapter 16 covers automatic index and materialized view selection. Chapter 17 covers adaptive query optimization.

Chapter 18 provides a dedicated, critical treatment of large language models for query optimization. This is a rapidly evolving area with both genuine potential and significant hype. We cover what LLMs can and cannot do, how to represent queries and plans for LLMs, prompting and fine-tuning strategies, retrieval-augmented approaches, agent architectures, and practical limitations.

Chapters 19 through 22 cover production deployment and evaluation. Chapter 19 walks through building a complete end-to-end AI-assisted optimizer system. Chapter 20 establishes rigorous evaluation methodology, including benchmarks, metrics, and testing for robustness. Chapter 21 covers production deployment: model serving, latency management, retraining, drift

detection, observability, and safety mechanisms. Chapter 22 looks forward to open problems and future directions.

By the end, you will have a comprehensive understanding of both the theoretical foundations and practical implementation of AI-driven query optimization. You will be equipped to evaluate new research, build ML-assisted optimization tools for your own systems, and make informed decisions about which techniques to adopt in production.

## References

[1] Leis V, Gubichev A, Mirchev A, Boncz P, Kemper A, Neumann T. "How good are query optimizers, really?" Proceedings of the VLDB Endowment, 9(3): 204–215, 2015. URL: https://www.vldb.org/pvldb/vol9/p204-leis.pdf

[2] Ibaraki T, Kameda T. "On the complexity of finding an optimal join order." In: Proc. Intl. Conf. Very Large Data Bases, pages 158–166, 1981.

[3] Selinger PG, Astrahan MM, Chamberlin DD, Lorie RA, Price TT. "Access path selection in a relational database management system." In: Proceedings of the ACM SIGMOD Conference, pages 23–34, 1979.

[4] Leis V, Radke B, Gubichev A, Kemper A, Neumann T. "Cardinality estimation done right: index-based join sampling." In: Proc. CIDR, 2017.

# Chapter 2: Anatomy of a Traditional Query Optimizer

## SQL Parsing and the Parse Tree

Every query optimization journey begins with a SQL string. When you write SELECT * FROM users WHERE age > 25, you are expressing a declarative intent: you want all columns from rows in the users table where the age column is greater than 25. You are not specifying how to retrieve those rows, which index to use, or in what order to scan the table. That is the optimizer's job.

PostgreSQL's query processing pipeline starts with parsing. The SQL string is tokenized into lexical tokens: keywords, identifiers, literals, operators, and punctuation. The parser then assembles these tokens into a parse tree that represents the syntactic structure of the query. At this stage, no semantic checking has occurred. The parser does not verify that the table users exists, that age is a column in users, or that 25 is a valid comparison value for the age column.

The parse tree is an internal representation used during parsing. It is quickly transformed into a query tree, which is a more structured representation that the rest of the pipeline uses. In PostgreSQL source code, this transformation happens in the analyzer module, located in src/backend/parser/analyze.c.

Consider our example query. The parse tree captures:

- SELECT list: asterisk, meaning all columns
- FROM clause: single relation named users
- WHERE clause: binary comparison, greater than, between column age and integer literal 25

The query tree representation is similar but more structured, with typed fields for the select target list, range table entries, and qualification expressions. This representation is what the optimizer starts from.

## From Parse Tree to Logical Query Representation

After parsing comes analysis. The analyzer resolves all names: table names are matched against the database catalog, column names are matched against the resolved tables, and type checking is performed. If the analyzer discovers an error – such as a reference to a non-existent table – it reports it and the query processing stops. No optimization occurs.

Once analysis is complete, the query tree is rewritten if necessary. Views are expanded: the analyzer substitutes the view's underlying query for the view reference. Rules defined on tables are applied. For example, if you query a view, PostgreSQL does not have a separate optimization phase for the view; it expands the view definition inline and optimizes the combined query as a whole. This is one of PostgreSQL's strengths: views are not materialized unless explicitly defined as materialized views, so the optimizer can push predicates through view definitions, use indexes on underlying tables, and produce efficient plans that treat the view as transparent.

After rewriting, the query tree is converted into a logical query representation that the optimizer uses as input. This representation is still relational-algebra-like: it describes what operations must be performed (scan these tables, join them on these conditions, filter on these predicates, project these columns, aggregate by these groups) but not how they will be performed physically. The distinction between logical and physical plans is crucial and deserves careful attention.

A logical plan is an abstract sequence of operations. It might say "join users with orders on users.id = orders.user_id." It does not specify whether to use a nested loop join, a hash join, or a merge join. It does not specify whether to scan users sequentially or using an index. It might say "filter on age > 25" but does not say whether the filter is applied before or after the join.

A physical plan is concrete. It specifies exactly which operator is used for each operation, which index is scanned, in what order joins are performed, how memory is allocated, and whether operations are parallelized. The optimizer's job is to choose among the many possible physical plans that correctly implement the logical plan, selecting the one that it estimates will execute fastest.

## The Query Optimization Pipeline in PostgreSQL

PostgreSQL's optimizer is a classic cost-based optimizer. It generates multiple candidate plans and selects the one with the lowest estimated cost. The process follows a well-defined sequence:

First, the optimizer generates scan paths for each individual table in the query. A scan path describes a way to read rows from a table. The most basic scan path is a sequential scan, which reads every page of the table in order. If indexes exist on the table, the optimizer also generates index scan paths. For a B-tree index on age, for example, the optimizer generates an index scan path that uses the index to find rows where age > 25. For each scan path, the optimizer estimates the cost based on the statistics about the table and the index.

Next, the optimizer combines these base scan paths into join paths for each pair of tables that must be joined. For each pair, it considers different join algorithms:

- Nested loop join: for each row from the outer input, scan the inner input looking for matching rows
- Hash join: build a hash table from one input, then probe the hash table with rows from the other input
- Merge join: sort both inputs on the join key and then merge-scan them simultaneously

The choice of join algorithm depends on many factors: the estimated sizes of the inputs, whether the inputs are already sorted on the join key, whether there is sufficient memory to build a hash table, and the relative costs of different operations.

For queries with more than two tables, the optimizer must also decide the join order: which pair to join first, what to join next, and so on. As discussed in Chapter 1, this is the NP-hard join ordering problem. PostgreSQL uses dynamic programming to explore left-deep join trees exhaustively up to a threshold (geqo_threshold, default 12 tables), then switches to a genetic algorithm for larger joins.

Throughout this process, the optimizer applies algebraic transformations to explore alternative plans. It might push a filter down to be applied before

a join (reducing the join input size), or it might pull a filter up to be applied after a join (because the filter predicate references columns from both sides). It might choose to sort data early so a merge join can be used later, or it might avoid sorting and use a hash join instead. These transformations are governed by algebraic equivalences: two plans that are algebraically equivalent always produce the same result, but they may have very different costs.

The optimizer explores a huge space of possible plans. Each plan is represented as a path structure in PostgreSQL's internal representation. Each path has an associated cost (startup cost plus total cost) and an estimated number of output rows. The optimizer uses these estimates to prune paths: if it has already found a path that produces all the rows needed at a lower cost than a new path under consideration, it discards the new path. This pruning is aggressive and essential for keeping the search tractable.

Finally, the optimizer selects the best path for the overall query — the one with the lowest total cost — and converts it into a physical plan. The physical plan is a tree of executor nodes, each of which corresponds to a specific operator that the executor will run. The executor then walks this tree, fetching rows from child nodes and applying operators to produce results for parent nodes, until the root node produces the final result set.

This entire process happens in a fraction of a second for most queries. For complex queries with many joins, it can take milliseconds or even seconds. The optimizer is a sophisticated piece of software that embodies decades of database research.

## EXPLAIN and EXPLAIN ANALYZE — Reading Plan Output

The primary tool for understanding how PostgreSQL optimizes queries is the EXPLAIN command. When you run EXPLAIN SELECT ..., PostgreSQL shows the plan it chose without actually executing the query. When you run EXPLAIN ANALYZE SELECT ..., PostgreSQL executes the query and shows both the plan and actual execution statistics.

Let us look at a concrete example. Suppose we have a PostgreSQL database with a table orders:

```
CREATE TABLE orders (
    id SERIAL PRIMARY KEY,
    customer_id INTEGER NOT NULL,
    product_id INTEGER NOT NULL,
    quantity INTEGER NOT NULL,
    order_date DATE NOT NULL
);
```

After populating this table with data and running ANALYZE to collect statistics, we can examine plans:

```
EXPLAIN SELECT * FROM orders WHERE customer_id = 42;
```

PostgreSQL might return:

```
Seq Scan on orders  (cost=0.00..45800.00 rows=100 width=24)
  Filter: (customer_id = 42)
```

This plan says: perform a sequential scan of the entire orders table, then filter rows where customer_id equals 42. The cost is estimated at 45,800 units, and the optimizer expects 100 output rows, each 24 bytes wide.

Now suppose we create an index:

```
CREATE INDEX idx_orders_customer_id ON orders (customer_id);
```

Running EXPLAIN again on the same query:

```
EXPLAIN SELECT * FROM orders WHERE customer_id = 42;
```

PostgreSQL now returns:

```
Index Scan using idx_orders_customer_id on orders  (cost=0.43..13.88 rows=100
↪  width=24)
  Index Cond: (customer_id = 42)
```

The optimizer has switched to an index scan. The estimated cost is now about 14 units instead of 45,800. The index allows PostgreSQL to jump directly

to rows where customer_id equals 42, avoiding the need to scan the entire table.

The key to reading EXPLAIN output is understanding the cost fields. Each plan node shows cost=startup..total. The startup cost is the cost to produce the first row. The total cost is the cost to produce all rows. For a sequential scan, the startup cost is zero because you start reading immediately. For an index scan, the startup cost is slightly higher because you must first navigate the index tree. The total cost for an index scan can be much lower if the index dramatically reduces the number of pages scanned.

EXPLAIN ANALYZE adds actual execution statistics:

```
EXPLAIN ANALYZE SELECT * FROM orders WHERE customer_id = 42;
```

The output now includes actual time, actual rows, and loops:

```
Index Scan using idx_orders_customer_id on orders  (cost=0.43..13.88 rows=100
↪  width=24)
  (actual time=0.023..0.156 rows=95 loops=1)
  Index Cond: (customer_id = 42)
Planning Time: 0.245 ms
Execution Time: 0.198 ms
```

The actual time shows that the first row was produced in 0.023 milliseconds and all rows were produced by 0.156 milliseconds. The actual rows (95) is close to the estimated rows (100), which indicates that the cardinality estimate was reasonably accurate. Planning time was 0.245 milliseconds – negligible compared to execution time.

EXPLAIN ANALYZE is the most valuable tool for diagnosing query performance problems. By comparing estimated rows and costs to actual values, you can identify where the optimizer's assumptions are wrong. Large discrepancies between estimated and actual rows suggest inaccurate statistics or problematic data distributions – exactly the cases where learned estimators might help.

For machine learning-based optimizer research and development, EXPLAIN FORMAT JSON is particularly useful. It provides a structured, parseable representation of the plan:

```
EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON) SELECT * FROM orders WHERE customer_id
↪  = 42;
```

The JSON output contains all plan node details in a machine-readable format, including buffer usage statistics. This structured output is what training data collection pipelines parse to extract features for ML models. Every learned optimizer that integrates with PostgreSQL relies on this JSON output as its ground truth source.

## Cost Model Components: CPU, I/O, Memory

PostgreSQL's cost model is the engine that drives all optimization decisions. It assigns abstract cost values to different types of operations so the optimizer can compare plans. Understanding this cost model is essential for understanding what ML techniques might improve.

The cost model has several configurable parameters:

- seq_page_cost: Cost of reading a 8-kilobyte page sequentially. Default is 1.0. This is the reference cost against which all other costs are calibrated.
- random_page_cost: Cost of reading a page randomly. Default is 4.0. This reflects the assumption that random I/O is four times slower than sequential I/O, which was historically true for spinning hard drives. On modern SSDs or when data is cached in memory, this assumption is often wrong.
- cpu_tuple_cost: Cost of processing one tuple (row). Default is 0.01.
- cpu_index_tuple_cost: Cost of processing one index tuple. Default is 0.005, reflecting that index tuples are typically smaller and faster to process than table tuples.
- cpu_operator_cost: Cost of evaluating one operator (such as a comparison or arithmetic operation). Default is 0.0025.
- effective_cache_size: An estimate of how much memory PostgreSQL can use for caching data pages. Default is 4 gigabytes. This parameter does not control memory allocation; it influences cost estimation. A larger effective_cache_size tells the optimizer that data is more likely to be in cache, reducing the estimated cost of random I/O operations.

Let us work through a concrete cost calculation. Consider a sequential scan of a table with 1 million rows and 5,406 pages (each 8 kilobytes):

```
disk_cost = pages * seq_page_cost = 5,406 * 1.0 = 5,406
cpu_cost = rows * cpu_tuple_cost = 1,000,000 * 0.01 = 10,000
total_cost = disk_cost + cpu_cost = 15,406
```

Now consider an index scan that uses an index on the filter column. Suppose the filter selects 100 rows, and the index has 500 pages:

```
index_disk_cost = index_pages * seq_page_cost = 500 * 1.0 = 500
heap_random_cost = rows_selected * random_page_cost = 100 * 4.0 = 400
cpu_index_cost = rows_selected * cpu_index_tuple_cost = 100 * 0.005 = 0.5
cpu_heap_cost = rows_selected * cpu_tuple_cost = 100 * 0.01 = 1.0
total_cost = 500 + 400 + 0.5 + 1.0 = 901.5
```

The index scan is much cheaper: 901 versus 15,406. That is why the optimizer chooses the index.

But notice the assumptions embedded in this calculation. The cost model assumes that each of the 100 heap pages accessed by the index scan is a random read at cost 4.0. In reality, if those pages are already in the buffer cache, the cost is essentially zero. If the index entries are clustered such that the heap pages are accessed in order, the cost is closer to sequential read cost. The cost model does not know any of this. It uses fixed parameters and crude estimates.

This is a fundamental limitation. Cost models must be simple and fast to compute during optimization. They cannot run detailed simulations of cache behavior, CPU branch prediction, or storage controller behavior. They use heuristics and approximations that work well on average but fail in specific situations.

Machine learning can help by learning cost models from actual execution data. Instead of using fixed formulas, a learned cost model can predict execution time based on features of the plan and the query, potentially capturing hardware-specific behaviors and runtime dynamics that handcrafted formulas miss. We explore this in Chapter 12.

## Planner Configuration Parameters and Their Effects

PostgreSQL exposes numerous configuration parameters that influence optimizer behavior. Some are boolean flags that enable or disable specific plan types; others are numeric parameters that tune cost estimation; still others control how aggressively the optimizer searches for plans.

The enable_* parameters are particularly important for understanding optimizer decisions:

- enable_seqscan: Controls whether sequential scans are considered. Default is on.
- enable_indexscan: Controls whether index scans are considered. Default is on.
- enable_bitmapscan: Controls whether bitmap heap scans are considered. Default is on.
- enable_nestloop: Controls whether nested loop joins are considered. Default is on.
- enable_hashjoin: Controls whether hash joins are considered. Default is on.
- enable_mergejoin: Controls whether merge joins are considered. Default is on.

Setting any of these to off does not completely eliminate that plan type. It increases the cost of plans using that type by a factor of 10,000, making them unlikely to be chosen unless they are the only option. These parameters are primarily useful for experimentation and debugging: you can force the optimizer to consider or avoid specific strategies to understand their impact.

The geqo_* parameters control the Genetic Query Optimizer:

- geqo: Enables GEQO entirely. Default is on.
- geqo_threshold: Number of FROM items at which GEQO is used instead of dynamic programming. Default is 12.
- geqo_effort: Controls how hard GEQO tries to find a good plan, on a scale of 1 to 10. Default is 5. Higher values increase planning time but may find better plans.
- geqo_pool_size, geqo_generations, geqo_selection_bias: Lower-level genetic algorithm parameters that control population size, number of generations, and selection pressure.

The statistics parameters affect cardinality estimation:

- default_statistics_target: Number of histogram buckets collected per column by ANALYZE. Default is 100. Increasing this improves estimation accuracy for skewed distributions but increases ANALYZE time and statistics storage.

- plan_cache_mode: Controls how prepared statements are planned. Default is auto, where PostgreSQL decides whether to use a generic plan (independent of parameter values) or custom plans (optimized for specific parameter values). Setting it to force_custom_plan can help with queries whose optimal plan is highly parameter-sensitive.

Understanding these parameters is important for two reasons. First, they provide knobs that you can tune when you encounter optimizer problems. Second, they reveal the optimizer's assumptions and decision points – exactly the places where learned components might improve behavior.

For example, a learned optimizer might dynamically adjust the effective value of random_page_cost based on observed cache hit rates for specific tables. Or it might learn when a generic plan is actually suboptimal for a parameterized query and trigger re-planning. Or it might learn to guide GEQO's search more intelligently than a generic genetic algorithm. These are all research directions that build on understanding how the traditional optimizer uses these parameters.

## Comparing PostgreSQL with Other Systems

While PostgreSQL is our primary reference, understanding how other major databases approach optimization provides useful perspective on design choices and alternatives.

MySQL's optimizer uses a similar cost-based approach but with different implementation details. MySQL historically had weaker cardinality estimation, especially for joins, though recent versions have improved significantly. MySQL uses index condition pushdown (ICP) to push filter predicates into the index scan, reducing the number of heap lookups. MySQL 8.0 introduced subquery materialization and lateral derived tables, expanding the optimization opportunities.

SQLite uses a highly simplified optimizer. It primarily uses rule-based optimization with limited cost estimation. SQLite's optimizer is designed for simplicity and correctness rather than performance on complex queries, which is appropriate for its embedded use cases. The lack of sophisticated optimization in SQLite makes it less relevant as a target for ML-driven optimization, though learning-based enhancements for simple optimizers are an interesting research direction.

DuckDB takes a very different approach. It is an in-process analytical database designed for OLAP workloads. Its optimizer uses the DPhyp algorithm (dynamic programming with hypergraphs) for join ordering, based on the paper "Dynamic Programming Strikes Back" by Moerkotte and Neumann [5]. DuckDB's execution engine is vectorized: it processes data in batches (vectors) of 1024 or 2048 rows, rather than one row at a time. This design choice fundamentally changes the cost model: vectorized execution is much more efficient for analytical queries that scan large amounts of data, but less efficient for point queries. DuckDB also uses columnar storage, which further biases the cost model toward full scans and vectorized operations.

ClickHouse is another analytical database with a strong emphasis on columnar storage and vectorized execution. Its optimizer is less cost-based and more rule-based, relying heavily on data layout optimizations (primary key indexes, data skipping indexes, materialized views) rather than sophisticated plan search. ClickHouse's optimization philosophy is that the right storage layout makes most queries fast enough that complex optimization is unnecessary. This is a different design paradigm than the traditional cost-based optimizer, and it has implications for how ML techniques might be applied.

Oracle, SQL Server, and commercial databases have optimizers that are broadly similar in approach to PostgreSQL: cost-based optimization with dynamic programming join ordering, cardinality estimation using statistics, and cost models parameterized by hardware characteristics. They also have more sophisticated features: Oracle has histogram types for various data distributions, SQL Server has adaptive query processing features that adjust plans during execution, and both have extensive online tuning capabilities. The core principles are the same across all cost-based optimizers, which means that ML techniques that work for PostgreSQL often transfer to other systems with some adaptation.

## References

[5] Moerkotte G, Neumann T. "Dynamic programming strikes back." Proceedings of the VLDB Endowment, 1(2): 539–549, 2008.

# Chapter 3: Relational Algebra and Query Plans

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Basic Relational Operators

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Logical vs Physical Plans

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Algebraic Equivalences and Transformation Rules

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Left-Deep vs Bushy Join Trees

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## PostgreSQL's Join Tree Representation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Building Alternative Plans Through Transformations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Code Example: Parsing EXPLAIN JSON and Extracting Plan Structure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

# Chapter 4: Statistics and Cardinality Estimation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Table Statistics: Column Histograms, Most-Common-Values

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Correlation Handling and Dependency Statistics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Join Selectivity Estimation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## PostgreSQL Statistics Collection: ANALYZE, Histogram Buckets

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Failure Modes: Skew, Outliers, Correlated Columns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Impact of Estimation Errors on Plan Choice

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

# Chapter 5: Join Ordering and Search Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## The NP-Hard Join Ordering Problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Dynamic Programming for Left-Deep Trees

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Greedy Search and Branch-and-Bound

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## PostgreSQL's GEQO Genetic Algorithm

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Query Decomposition and Join Graph Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Trade-offs: Optimality vs Planning Time

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

# Chapter 6: Indexing and Physical Design

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## B-Tree Indexes: Structure, Cost, and Usage

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Hash Indexes and Bitmap Indexes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Partial Indexes and Covering Indexes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## PostgreSQL: GiST, GIN, BRIN, SP-GiST

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Index Selection as Part of Query Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Multi-Dimensional Indexes and Composite Keys

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

# Chapter 7: Cost Models and Plan Selection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## The Total Cost Formula: CPU, I/O, Memory

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Cost Parameters in PostgreSQL

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Selectivity-Based Cost Computation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Pipeline vs Materialization Costs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Memory Grant Estimation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Plan Selection: Choosing the Minimum-Cost Plan

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

# Chapter 8: Query Execution Engines

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Volcano Iterator Model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Vectorized Execution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Hash-Based Execution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Nested Loop, Sort-Merge, Index Scan Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## PostgreSQL's Executor Internals

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## The Plan Execution Feedback Loop

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

# Chapter 9: Caching, Reuse, and Parameterized Plans

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Prepared Statements and Plan Caching

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## PostgreSQL: Prepared Statements and Plan Inference

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Parameter Sniffing and Plan Stability Issues

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Adaptive Plans and Parameter-Sensitive Plans

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Plan Reuse Trade-offs: Overhead vs Consistency

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Query Fingerprinting and Workload Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

# Chapter 10: Machine Learning Foundations for Database Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Supervised Learning: Regression and Classification for Cost Estimation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Feature Engineering for Database Queries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Model Families: Linear Models, Random Forests, Gradient Boosting

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Deep Learning Fundamentals Relevant to Query Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Graph Representations for Query Plans

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Evaluation Metrics: Prediction Accuracy vs Optimization Quality

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

# Chapter 11: Learned Cardinality Estimation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Failure Modes of Traditional Histogram-Based Estimation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Designing Features for Query Representation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Random Forest Cardinality Estimators

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Deep Learning Approaches: RNNs, Transformers for Query Encoding

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Learned Histograms and Sampling-Based Methods

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Building a Learned Cardinality Estimator in Python

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Integration with PostgreSQL Optimizer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

# Chapter 12: Learned Cost Models

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## The Structure of Cost Models and Their Assumptions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Learning Cost from Execution Traces

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Feature Engineering for Plan Cost Prediction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Regression Models for Operator Cost Prediction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Handling Distributional Shifts Across Hardware

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## End-to-End Cost Model Training Pipeline

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Accuracy Requirements: What Margin of Error Matters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

# Chapter 13: Learned Indexes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## The Learned Index Paper and Core Ideas

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Piecewise Linear Regression Models for Index Functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Adaptive Radix Trees and Learned Variants

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Learned Index Performance: Query Latency, Memory, Insertions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Practical Implementations: Lucene, SQLite Extensions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## When Learned Indexes Help and When They Do Not

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

# Chapter 14: Neural Query Optimizers and Plan Selection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Representing Queries and Plans as Graphs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Graph Neural Networks for Plan Scoring

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Reinforcement Learning for Plan Search

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## End-to-End Optimizers That Bypass Traditional Search

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Hybrid Approaches: ML-Guided Search Within Traditional Optimizers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Training Data Generation and Reward Design

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

# Chapter 15: Workload-Aware Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Workload Classification and Clustering

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Temporal Patterns and Query Mixing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Learning Workload-Specific Cost Parameters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Adaptive Tuning Based on Workload Shifts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## PostgreSQL: Autovacuum and Workload Statistics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Multi-Tenant and OLTP vs OLAP Workload Challenges

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

# Chapter 16: Automatic Index and Materialized View Selection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## The Index Recommendation Problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Greedy and Genetic Algorithm Approaches

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## ML-Based Index Selection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Materialized View Recommendation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## PostgreSQL: pg_hint_plan and Index Usage Feedback

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Cost-Benefit Trade-offs: Space vs Query Speed

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

# Chapter 17: Adaptive Query Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Progressive Feedback During Execution

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## PostgreSQL: Adaptive Hash Joins

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Re-optimization and Query Reshaping

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Online Learning for Cardinality Estimation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Plan Guidance and Forced Plans

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Safety: Preventing Adaptive Failures from Cascading

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

# Chapter 18: LLM-Based Query Reasoning and Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## What LLMs Can and Cannot Do for Database Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Representing SQL and Query Plans for LLMs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Prompting Strategies for Query Optimization Assistance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Fine-Tuning LLMs on Plan Data

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Retrieval-Augmented Generation for Plan Recommendations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Tool-Using Agents and Optimizer as LLM Tool

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Hallucination, Correctness, and Evaluation Risks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Practical Architectures Combining LLMs with Traditional Optimizers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

# Chapter 19: Building an AI-Assisted Optimizer: End-to-End System

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## System Architecture: Data Collection, Model Serving, Optimizer Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Designing the Training Data Pipeline from PostgreSQL

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Model Selection for Each Optimization Stage

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## A/B Testing New Plans Against Baseline

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Gradual Rollout and Safety Mechanisms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Complete Code Implementation Walkthrough

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

# Chapter 20: Evaluation, Benchmarking, and Correctness



## Benchmarks: TPC-H, TPC-DS, JOB, Custom Workloads



## Metrics: Execution Time, Latency Percentiles, Throughput



## Planning Overhead and Inference Latency



## Model Accuracy vs Optimization Quality: The Critical Distinction

## Regression Detection and Robustness Testing



## Generalization Across Schemas and Workloads



## References

# Chapter 21: Production Deployment and Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Model Serving Architecture: Latency, Throughput, Reliability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Continuous Training and Drift Detection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Observability: Logging, Metrics, Tracing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Rollback Strategies and Plan Safety

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Handling Cold Starts and Data Scarcity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Privacy, Security, and Access Control

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Team Skills and Organizational Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

# Chapter 22: Frontiers and Open Problems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Open Research Problems in AI-Driven Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Scaling to Large Schemas and Complex Queries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Cross-Database and Cloud-Native Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Federated and Distributed Query Optimization with ML

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## The Path from Research to Production: Lessons Learned

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## Future Directions and a Realistic Assessment

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

## References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

# Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.

# References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/ai-drivendatabasequeryoptimization.