People's Democratic Republic of Algeria
الجمهورية الجزائرية الديمقراطية الشعبية
Ministry of Higher Education and Scientific Research
وزارة التعليم العالي و البحث العلمي

كلية الرياضيات والاعلام الالي
قسم الاعلام الالي
Faculty of Mathematics and Computer Science
Department of Computer Science

Final Year Project Report

For the attainment of the Master's degree in Computer Science

Specialization : Business Intelligence

AI-Driven Index Recommendation for SQL Query Optimization in PostgreSQL Databases

Prepared by:
AYATA HabibEllah
BENSADI Said Aymene

Supervised by:
Pr. ATTIA Abdelouaheb

Defended on 11 June, 2025, before the jury composed of:

Dr. Adel SAHA
Dr. Meriem REGOUID

Graduating Class: 2024/2025

Dedication

“

To all those who have illuminated my path, I dedicate this humble work with heartfelt gratitude.

To my mother, the embodiment of tenderness and patience, and to my father, Abdelhalim, whose strength guides me.

To my sister Leila, and my brothers Walid, Abderaouf, and Mohamed-Tayeb — your support and sacrifices mean everything.

To Walid's wife, Kenza, and their child Zaid — for your support and assistance.

To my uncles, aunts, and extended family — thank you for your presence and support.

To my dear companions and friends — Azzedine, Mounim, Aymen, Rachid, Abdeslam, and Youcef — your presence and friendship have been a true blessing on this journey.

To the souls of my late grandparents — may Allah have mercy on them — and to my grandmothers — may Allah protect them.

To my university colleagues and professional teammates — loyal companions throughout this journey.

To all who contributed, quietly or openly — my deepest respect and gratitude.

”

- Said Aymen

Dedication

“

I want to thank everyone — those who know me and those who don't.

”

- Habib Ellah

Acknowledgment

First and foremost, we thank God Almighty for granting us health, strength, and the determination to begin and successfully complete our studies,

We would like to express our deepest gratitude to our supervisor, **Pr. ATTIA Abdelouahab**, for giving us the opportunity to carry out this research, and for his invaluable guidance and advice throughout this work. His generosity, kindness, knowledge, and constant availability greatly facilitated our progress,

We are also sincerely thankful to the members of the jury, **Dr. Meriem REGOUID** and **Dr. Adel SAHA**, for honoring us with their presence, and for enriching our work with their insightful suggestions,

Our heartfelt thanks go as well to our parents, for their unconditional love, care, prayers, and sacrifices. Their unwavering support has been a pillar throughout our academic journey,

Finally, we wish to thank all those who, directly or indirectly, contributed to the completion of this thesis.

Abstract

Modern database management systems (DBMSs) like PostgreSQL offer several types of indexing mechanisms to accelerate query processing. However, determining the optimal index structure for diverse SQL queries remains a challenging task, particularly as data complexity and volume grow. This thesis presents a machine learning-driven framework for intelligent index recommendation aimed at improving query performance in PostgreSQL databases.

The methodology involves generating a synthetic dataset using a faker python script, constructing a relational schema, and populating it with realistic data. Diverse SQL queries are generated using Gemini AI to cover a wide range of complexity and patterns. These queries are preprocessed and transformed into feature vectors using techniques like TF-IDF, tokenization, and normalization. Each query is then labeled with the "best" index type—such as B-Tree, Hash, GIN, or Bitmap—based on performance metrics observed in PostgreSQL.

A supervised learning model, specifically a Random Forest classifier, is trained on this labeled dataset to predict optimal index types for unseen queries. The model is evaluated using standard metrics such as accuracy, confusion matrix analysis. Additionally, a user-friendly web interface is developed to facilitate real-time query input and indexing recommendations.

This work contributes to the ongoing research in automated database tuning and demonstrates how machine learning can be effectively applied to solve complex database optimization problems. The approach enhances database performance and reduces the manual effort required for index tuning.

Keywords : PostgreSQL, Index Recommendation, Machine Learning, Query Optimization, Random Forest, TF-IDF, Supervised Learning.

Résumé

Les systèmes modernes de gestion de bases de données tels que PostgreSQL offrent plusieurs mécanismes d'indexation pour accélérer le traitement des requêtes. Cependant, déterminer la structure d'index optimale pour une grande diversité de requêtes SQL reste une tâche complexe, notamment avec l'augmentation de la complexité et du volume des données. Cette thèse présente un cadre basé sur l'apprentissage automatique pour la recommandation intelligente d'index, visant à améliorer les performances des requêtes dans les bases de données PostgreSQL.

La méthodologie consiste à générer un ensemble de données synthétiques à l'aide d'un script Python Faker, à construire un schéma relationnel, puis à le remplir avec des données réalistes. Des requêtes SQL diverses sont générées via Gemini AI pour couvrir une large gamme de complexités et de motifs. Ces requêtes sont prétraitées et transformées en vecteurs de caractéristiques en utilisant des techniques telles que TF-IDF, la tokenisation et la normalisation. Chaque requête est ensuite étiquetée avec le type d'index « optimal » — tel que B-Tree, Hash, GIN ou Bitmap — sur la base des mesures de performance observées dans PostgreSQL.

Un modèle d'apprentissage supervisé, spécifiquement un classificateur Random Forest, est entraîné sur cet ensemble de données étiquetées afin de prédire le type d'index optimal pour des requêtes inconnues. Le modèle est évalué à l'aide de métriques standard telles que la précision et l'analyse de la matrice de confusion. De plus, une interface web conviviale a été développée pour faciliter la saisie en temps réel des requêtes et fournir des recommandations d'indexation.

Ce travail contribue à la recherche continue sur l'optimisation automatique des bases de données et démontre comment l'apprentissage automatique peut être efficacement appliqué pour résoudre des problèmes complexes d'optimisation des bases de données. Cette approche améliore les performances des bases de données et réduit l'effort manuel nécessaire à l'ajustement des index.

Mots-clés : PostgreSQL, recommandation d'index, apprentissage automatique, optimisation des requêtes, Random Forest, TF-IDF, apprentissage supervisé.

ملخص

توفر أنظمة إدارة قواعد البيانات الحديثة مثل بوستجريس كيو إل عدة آليات للفهرسة لتسريع تنفيذ الاستعلامات. ومع ذلك، فإن تحديد بنية الفهرسة المثلى لمجموعة متنوعة من استعلامات سي كيو إل لا يزال يمثل تحديًا كبيرًا، خاصة مع زيادة تعقيد البيانات وحجمها. تقدم هذه الأطروحة إطارًا مدعومًا بتقنيات التعلم الآلي للتوصية الذكية بأنواع الفهارس بهدف تحسين أداء الاستعلامات في قواعد بيانات بوستجريس كيو إل.

تشمل المنهجية إنشاء مجموعة بيانات تركيبية باستخدام مكتبة بايثون فاكر، وإنشاء مخطط علائقي، وتعبئته ببيانات واقعية. يتم إنشاء استعلامات سي كيو إل متنوعة باستخدام جيميني أيه آي لتغطية مجموعة واسعة من الأنماط والتعقيدات. تُعالَج هذه الاستعلامات وتُحوّل إلى متجهات خصائص باستخدام تقنيات مثل ت أف-آي دي أف، والتجزئة، والتطبيع. ثم يتم تصنيف كل استعلام حسب نوع الفهرس "الأمثل" مثل بي-تري أو هاش أو جين أو بيت ماب، بناءً على مقاييس الأداء في بوستجريس كيو إل.

يُدرَّب نموذج تعلم خاضع للإشراف، وهو مصنف الغابة العشوائية، على هذه البيانات المصنفة بهدف التنبؤ بنوع الفهرس الأمثل للاستعلامات الجديدة غير المرئية. يتم تقييم النموذج باستخدام مقاييس معيارية مثل الدقة وتحليل مصفوفة الالتباس. بالإضافة إلى ذلك، تم تطوير واجهة ويب سهلة الاستخدام لتسهيل إدخال الاستعلامات في الوقت الفعلي وتقديم توصيات الفهرسة.

تساهم هذه الدراسة في البحث المستمر في ضبط قواعد البيانات تلقائيًا وتُظهر كيف يمكن لتقنيات التعلم الآلي أن تُطبق بفعالية لحل مشاكل تحسين قواعد البيانات المعقدة. تعزز هذه الطريقة من أداء قواعد البيانات وتقلل من الجهد اليدوي المطلوب لضبط الفهارس.

كلمات مفتاحية :

بوستجريس كيو إل، توصية الفهرسة، التعلم الآلي، تحسين الاستعلامات، الغابة العشوائية، وزن المصطلحات مع التردد العكسي، التعلم الخاضع للإشراف.

Contents

List of Figures

List of Tables

List of acronyms

AI *Artificial Intelligence*

API *Application Programming Interface*

CBO *Cost-Based Optimizer (PostgreSQL context)*

CPU *Central Processing Unit*

CSV *Comma-Separated Values*

CTEs *Common Table Expressions (SQL context)*

DBAs *Database Administrators*

DBMSs *Database Management Systems*

GIN *Generalized Inverted Index (PostgreSQL index type)*

GiST *Generalized Search Tree (PostgreSQL index type)*

ML *Machine Learning*

NLP *Natural Language Processing*

OLAP *Online Analytical Processing*

RDBMS *Relational Database Management Systems*

SQL *Structured Query Language*

TF-IDF *Term Frequency-Inverse Document Frequency*

TIDs *Tuple IDs (PostgreSQL context)*

WAL *Write-Ahead Logging (PostgreSQL context)*

General Introduction

The contemporary digital era is characterized by an unprecedented generation and accumulation of data, rendering effective data management and retrieval paramount for organizational success.[4] Relational Database Management Systems (RDBMS) form the backbone of data storage for countless applications, with PostgreSQL standing out as a powerful, extensible, and widely adopted open-source solution[13]. However, ensuring optimal performance, particularly efficient query execution, remains a persistent challenge as databases scale in size and complexity.[6]
Database indexing is a fundamental technique employed to accelerate data access, yet the selection of the most appropriate indexing strategy within PostgreSQL's rich ecosystem of index types (including B-Tree, GIST, Hash, Bitmap, and potential variants like reverse key) constitutes a non-trivial task demanding significant expertise.[17]

This thesis explores the application of Artificial Intelligence (AI), specifically machine learning classification techniques, to automate and enhance the process of PostgreSQL index type recommendation. By analyzing the features of SQL queries, this research aims to develop an intelligent model capable of suggesting the optimal index type, thereby contributing to improved database performance and streamlined administration. This introductory chapter lays the groundwork by establishing the context, defining the problem, outlining the objectives and scope, and detailing the contributions of this research endeavor.[25]

Background and Motivation

In the contemporary digital landscape, data is undeniably a critical asset for organizations across all sectors. The ability to efficiently store, retrieve, and analyze vast quantities of information underpins core business operations, strategic decision-making, and the delivery of responsive user experiences. Relational Database Management Systems (RDBMS) remain the cornerstone for managing structured data, with PostgreSQL emerging as a highly popular, robust, and feature-rich open-source option known for its standards compliance, extensibility, and reliability.

The Importance of Database Performance Optimization

The sheer volume and velocity of data generated today present significant challenges to database performance. As databases grow, query execution times can degrade substantially, impacting application responsiveness, user satisfaction, and overall system efficiency. Slow queries can lead to bottlenecks, increased operational costs due to higher resource consumption (CPU, I/O, memory), and missed opportunities in time-sensitive business intelligence scenarios. Consequently, database performance optimization is not merely a technical refinement but a crucial activity directly influencing business outcomes. Effective optimization ensures that data can be accessed quickly and reliably, enabling real-time analytics, supporting high-throughput transactional systems, and maintaining a competitive edge. A fundamental technique within database performance optimization is the strategic implementation of indexes. Indexes are specialized data structures designed to accelerate data retrieval operations on database tables, analogous to an index in a book, allowing the database engine to locate specific rows rapidly without scanning the entire table.

The Potential of Artificial Intelligence in Database Management

The inherent complexity and data-driven nature of the index selection problem make it a prime candidate for the application of Artificial Intelligence (AI) and Machine Learning (ML) techniques. AI/ML models excel at identifying intricate patterns and relationships within large datasets, learning from experience, and making predictions or classifications based on that learning. In the context of database management, AI has shown promise in various areas, including automated parameter tuning, as demonstrated by systems like OtterTune which leverage machine learning to optimize database configurations.[15] Specifically for index selection, AI offers the potential to automate or significantly augment the decision-making process. By analyzing features extracted from SQL queries (such as keywords, operators, involved tables and columns, and clause structures) and correlating them with the performance characteristics of different index types under specific conditions, an ML model can learn to recommend the most suitable index. This approach can potentially:

- Reduce the reliance on scarce and expensive human expertise.
- Provide consistent and data-driven indexing recommendations.
- Adapt more readily to evolving workloads compared to manual tuning.
- Analyze query patterns at a scale and speed infeasible for human administrators.

This thesis explores this potential by leveraging AI to build an intelligent index recommendation system specifically tailored for PostgreSQL queries. The motivation stems from the clear need for more efficient, automated methods to tackle the persistent challenge of database performance optimization through effective index selection, thereby unlocking the full potential of powerful database systems like PostgreSQL.

Problem Statement

Efficient query processing is paramount for the performance of relational database systems. While indexing is a fundamental technique for accelerating data retrieval in PostgreSQL, the manual selection of the most effective index type among the diverse options available (such as standard B-Tree, GIST, Hash, Bitmap, and reverse) presents a significant challenge. Determining the optimal index requires a nuanced understanding of the specific query structure, the underlying data distribution and cardinality, the operational characteristics and overheads of each index type, and the potential impact on write operations. This complex decision-making process is often:

- **Time-Consuming:** Analyzing workloads and experimenting with different index configurations manually is laborious.
- **Error-Prone:** Incorrect index choices can lead to negligible performance gains, wasted storage resources, and increased maintenance overhead, potentially even degrading performance in some cases.
- **Difficult to Scale:** Manually reassessing and adapting indexing strategies for large, complex databases with dynamic query workloads and evolving data schemas is often impractical.

The lack of automated, intelligent tools specifically designed to recommend the most appropriate PostgreSQL index type based on query analysis leads to suboptimal database operation performance, inefficient resource utilization, and increased operational burden on database administrators and developers. Consequently, there exists a distinct need for a data-driven approach that can systematically analyze SQL queries and provide reliable recommendations for the best-suited indexing method from PostgreSQL's repertoire, thereby simplifying the optimization process and enhancing overall system efficiency. This thesis addresses the problem of automating the selection of optimal PostgreSQL index types for given SQL queries using an AI-driven recommendation model.

Research Objectives

Based on the identified problem of complex manual index selection in PostgreSQL and the potential of AI to address this challenge, this thesis aims to achieve the following primary objective: To fulfill this primary objective, the following specific research objectives have been defined:

- **To develop a methodology for generating synthetic data:** This involves creating a large-scale (over 1.5 million rows) synthetic PostgreSQL database schema and populating it with data using a Faker Pytohn script [8] for generating a diverse and representative set of SQL queries using Large Language Model capabilities, specifically the Gemini AI API (Google, n.d.-b)[10].
- **To engineer features from SQL queries:** This objective focuses on pre processing and cleaning the generated SQL queries and investigating effective techniques

for extracting meaningful features that capture the query's structure and intent. Specifically, this research will utilize the Term Frequency-Inverse Document Frequency (TF-IDF) method for vector representation.

- **To construct a labeled dataset for supervised learning:** This involves establishing a reliable process to determine the ground-truth optimal index type for each generated query within the context of the synthetic database and chosen index candidates. This labeled dataset will serve as the basis for training the predictive model.
- **To train and evaluate a machine learning classification model:** This objective involves selecting, implementing, and training a suitable machine learning classifier, specifically Random Forest, to learn the mapping between SQL query features and the optimal index type. The model's predictive performance will be rigorously evaluated using standard classification metrics (e.g., accuracy, confusion matrix).
- **To implement a prototype system with a user interface:** This involves developing a basic user interface that allows a user to input an SQL query and receive the index type recommendation generated by the trained AI model, demonstrating the practical application of the research.

Achieving these objectives will result in a functional prototype system and empirical evidence regarding the feasibility and effectiveness of using an AI-driven approach for PostgreSQL index type recommendation.

Scope and Limitations

To ensure clarity and focus, it is essential to define the boundaries of this research endeavor.

Scope:

- **Database System:** This study is specifically focused on the PostgreSQL Relational Database Management System. The findings and the developed model are tailored to its specific indexing capabilities and query processing characteristics.
- **Index Types:** The investigation is confined to a predefined set of common and distinct PostgreSQL index types: Generalized Search Tree (GIST), reverse key , Hash, Bitmap, and the standard B-Tree index. Other specialized or extension-provided index types are outside the scope.
- **Recommendation Focus:** The primary goal is to recommend the most appropriate type of index for a given SQL query, not to determine the optimal set of columns to include in the index or the precise physical design parameters.
- **Input:** The model takes an individual SQL query as input for analysis and recommendation.

- **Methodology:** The approach employs TF-IDF for feature extraction from SQL queries and Random Forest as the classification algorithm for index type prediction.
- **Data Source:** The model development and evaluation rely on a synthetically generated database (using Python script) and synthetically generated SQL queries (using the Gemini AI API).
- **Output:** The final output includes the trained AI model and a prototype user interface demonstrating its query-input and recommendation-output functionality.

Limitations:

- **Reliance on Synthetic Data:** The use of synthetic data and AI-generated queries, while enabling control and scale, may not perfectly replicate the complexity, distributions, correlations, and specific patterns found in real-world production database schemas and application workloads. The generalizability of the findings to diverse real-world scenarios needs further validation.
- **Limited Feature Representation:** Relying solely on TF-IDF for feature extraction might not capture all semantic or structural nuances of SQL queries relevant to index selection. More sophisticated NLP techniques or query plan analysis were not explored within this scope.
- **Model Specificity:** The results are specific to the chosen Random Forest algorithm. Other machine learning models (e.g., Gradient Boosting, Neural Networks) might yield different performance characteristics.
- **Static Environment Assumption:** The model is developed based on a static snapshot of the database schema and workload. It does not inherently account for schema evolution, data drift, or changes in query patterns over time.
- **Write Performance Impact:** The primary focus is on optimizing query retrieval performance (SELECT statements). The potential impact of the recommended indexes on the performance of data modification operations (INSERT, UPDATE, DELETE) is not explicitly evaluated.

Contribution of the Thesis

This Master's thesis contributes to the field of Business Intelligence and AI-driven database optimization by addressing the specific challenge of index type selection in PostgreSQL. The key contributions of this research are:

- **Development and Evaluation of an AI-based Index Type Recommender for PostgreSQL:**
  The primary contribution is the design, implementation, and empirical evaluation of a novel machine learning model specifically aimed at recommending the most suitable index type (GIST, reverse key, hash, bitmap, or B-Tree) for given SQL queries within a PostgreSQL context.

- **Novel Application of LLM for Query Generation in Database Optimization Research:**
  This work demonstrates a methodological contribution through the application of a large language model (Gemini AI) to systematically generate a diverse and large-scale set of SQL queries. This technique addresses a common bottleneck in database optimization research – the need for varied and representative query workloads for training and testing AI models, especially when real-world logs are unavailable or insufficient.

- **Application of NLP Techniques for SQL Query Feature Engineering:**
  The thesis investigates and demonstrates the effectiveness of using Term Frequency-Inverse Document Frequency (TF-IDF), a common Natural Language Processing technique, for extracting relevant features directly from raw SQL query strings to predict optimal index types. This contributes empirical evidence regarding the suitability of TF-IDF for this specific database optimization task.

- **Creation of a Synthetic Data Generation and Evaluation Framework:**
  A framework combining synthetic database population (script Python Faker) with AI-driven query generation and automated labeling (based on performance within the synthetic environment) is established. This framework provides a reproducible methodology for generating the necessary datasets to train and evaluate AI models for database optimization tasks in controlled environments.

- **Empirical Performance Benchmarks:**
  The research provides quantitative performance results (accuracy,confusion matrix) for the Random Forest classifier using TF-IDF features in the context of PostgreSQL index type recommendation on a large synthetic dataset. These benchmarks serve as a reference point for future research in this specific area.

- **Proof-of-Concept Prototype:**
  The development of a user interface connected to the trained model serves as a tangible proof-of-concept, illustrating the practical potential of integrating such an AI recommender into the workflow of database administrators or developers.

Collectively, these contributions advance the understanding of how AI, particularly NLP and classification techniques, can be effectively applied to automate and improve the complex task of index type selection in modern relational database systems like PostgreSQL.

Thesis Structure

This thesis is organized into three main chapters, preceded by this General Introduction and followed by a general conclusion and perspectives.

Chapter 1, "Literature Review and State of the Art," delves into the foundational concepts of database indexing, detailing various types prevalent in PostgreSQL such as B-Tree, Hash, GiST, and GIN, alongside index selection strategies. It further explores

established query optimization techniques and the applications of machine learning, including Natural Language Processing for SQL query feature extraction and the use of Random Forests for recommendation systems.

Chapter 2, "Methodology," outlines the systematic approach undertaken for this research, covering the generation of a synthetic PostgreSQL database, the AI-driven creation of a diverse SQL query set, the feature engineering process using TF-IDF, the empirical labeling of optimal index types, the development and training of a Random Forest classifier, and the design of the user interface.

Chapter 3, "Experiments and Results," presents the findings from the implementation, including the characteristics of the generated data and queries, the outcomes of feature engineering, the performance evaluation of the trained model through metrics like accuracy and confusion matrix analysis, and a showcase of the developed user interface.

Finally, the thesis concludes with a summary of contributions and outlines potential avenues for future work.

Chapter 1

Literature Review and State of the Art

1.1 Introduction

Optimizing query performance through effective index selection is a cornerstone of modern database administration. While indexes are vital structures for accelerating data retrieval, choosing the right strategy is a complex and resource-intensive task that demands a deep understanding of query patterns, data distributions, and system internals. This chapter provides a critical review of the existing literature and state-of-the-art tools to build a comprehensive foundation for our research. We begin by dissecting the fundamentals of database indexing, exploring the distinct characteristics and trade-offs of various types available in PostgreSQL. From there, we examine the evolution of index selection techniques, tracing the path from traditional manual tuning to the sophisticated, automated systems powered by machine learning and artificial intelligence. By critically analyzing prominent tools and research frameworks, this review will not only illuminate the current advancements in the field but also identify the existing gaps and opportunities for improvement, thereby contextualizing the novel approach this thesis proposes.

1.2 Fundamentals of Database Indexing

Efficient data retrieval lies at the heart of modern database systems, and indexing is one of the most powerful techniques employed to accelerate query performance. An index in a database is a data structure that improves the speed of data retrieval operations on a database table at the cost of additional writes and storage space. Indexes are especially vital in large-scale systems, where full table scans can be computationally expensive and time-consuming.

The concept of indexing in databases is closely analogous to an index in a book — rather than reading every page to find a particular topic, the index directs the reader to the precise location. Similarly, database indexes allow the query processor to quickly locate and access the desired data without scanning every row in the table. Most modern relational database management systems (RDBMSs), including PostgreSQL, MySQL, SQL Server, and Oracle, implement multiple types of indexing strategies tailored to various query patterns and data types.

Indexes are typically built on one or more columns of a table and serve as auxiliary data structures. Among the common indexing structures are B-Trees, hash indexes, bitmap indexes, GiST (Generalized Search Trees), and GIN (Generalized Inverted Indexes). Each index type offers specific advantages and is suitable for particular use cases depending on query predicates, data cardinality, and access patterns.

The use of indexes involves a critical trade-off between read performance and write overhead. While indexes significantly enhance read operations such as *SELECT* queries, they can impose overhead during *INSERT, UPDATE, and DELETE* operations, as the index structures must be maintained synchronously with the data.
Therefore, intelligent index design requires a deep understanding of query workloads

and update frequencies. Poorly chosen indexes may lead to degraded performance or excessive storage usage [3].

In PostgreSQL, for example, indexes can be explicitly created by users using the CREATE INDEX statement, or automatically via primary key and unique constraints. Moreover, the PostgreSQL query planner uses the cost-based optimizer (CBO) to decide when and how to use indexes, taking into account statistics and cost models [13]. Understanding the internal mechanics of different index types is crucial for database administrators and developers to achieve optimal performance.

It is also important to consider the implications of indexing in terms of storage space. While indexes improve retrieval time, they occupy disk space that scales with the volume of indexed data. Consequently, large tables with multiple indexes can suffer from increased storage and I/O overhead. This underscores the need for selective indexing and periodic index maintenance strategies such as REINDEX or VACUUM.

Furthermore, composite indexes (indexes on multiple columns) and partial indexes (indexes that include only a subset of rows) provide additional flexibility. These advanced techniques allow for further optimization tailored to specific workloads and query conditions.

In summary, indexing remains a fundamental and indispensable component of relational database systems. Its intelligent application significantly boosts performance and ensures scalability in high-demand environments. However, the design and maintenance of indexes require careful balancing of retrieval benefits and write costs, making them a central topic in the field of database performance tuning.

1.2.1 Basic Concepts and Trade-offs:

Database indexing is fundamentally concerned with accelerating query performance by organizing data in auxiliary structures that enable rapid access. At its core, an index is a physical structure associated with a database table or view that allows the database management system (DBMS) to locate records more efficiently, typically without scanning the entire dataset. The concept is conceptually similar to a book's index, which helps readers quickly find relevant pages based on keywords.

Indexes are built upon the idea of reducing the search space when querying large volumes of data. In an unindexed table, searching for a value typically involves a sequential scan (also called a full table scan), which has a time complexity of O(n), where n is the number of rows. With an appropriate index, the DBMS can reduce search complexity to O(log n) or even O(1) in specific cases [21]. This can drastically improve the performance of read-heavy operations, such as those found in OLAP (Online Analytical Processing) systems.

However, this performance gain does not come without cost. One of the principal trade-offs of using indexes lies in write performance degradation. Since indexes must be updated whenever the underlying data changes (e.g., through INSERT, UPDATE, or DELETE operations), they introduce additional write and maintenance overhead. This makes indexing a classic case of the read-write trade-off: optimizing for fast reads may slow down writes, and vice versa.

Moreover, indexes consume additional disk space, which may become substantial for large datasets or when multiple indexes are maintained on the same table. For example, in PostgreSQL, creating a B-tree index on a large column can result in significant memory usage, especially when index rebalancing is triggered by heavy insertions [13].

Another important concept is index selectivity, which refers to how well an index distinguishes between rows. Highly selective indexes (those where the indexed column has many distinct values) are typically more useful for queries. Conversely, indexing columns with low selectivity (such as boolean flags or gender fields) may result in poor performance gains and could even harm performance due to wasted I/O and planner misestimation [3].

PostgreSQL offers features like partial indexes and expression indexes to help mitigate some of these trade-offs. A partial index, for example, indexes only a subset of rows that meet a specific predicate, thus saving storage and reducing maintenance costs while still supporting relevant queries. Likewise, multicolumn or composite indexes allow developers to optimize queries that involve multiple predicates efficiently, but they must be carefully ordered to match the most frequent query patterns.

In practice, the decision to create, retain, or drop an index should be driven by a comprehensive understanding of query patterns, update frequencies, and storage constraints. Modern systems also incorporate index usage statistics, helping database administrators decide which indexes are effective and which are unused or costly to maintain.

In conclusion, the use of indexes in relational databases involves a delicate balance between read efficiency, write overhead, storage utilization, and administrative complexity. Mastery of indexing principles is therefore essential for designing scalable and responsive database applications.

1.2.2 B-Tree Indexing (Standard)

B-tree indexes (short for balanced tree) are the default and most commonly used indexing structures in modern relational database management systems (RDBMS), including PostgreSQL. Introduced by Bayer and McCreight in 1972,and further popularized and analyzed by Comer[5], the B-tree is a self-balancing tree data structure that maintains

sorted data and allows searches, insertions, deletions, and sequential access in logarithmic time (O(log n)) [1].

In PostgreSQL, the B-tree index is the default index type because it provides an optimal balance of performance across a broad range of queries, especially those involving comparisons such as $<$, $<=$, $=$, $>=$, and $>$ [13]. A B-tree maintains its balance by keeping all leaf nodes at the same depth and by splitting and merging nodes as needed during inserts and deletes, ensuring predictable performance.
Each node in a B-tree index contains keys and pointers (also known as references) to child nodes or tuples. During query execution, PostgreSQL traverses the B-tree from the root to the appropriate leaf node, narrowing the search range at each level. This structure makes B-tree indexes particularly efficient for:

- **Equality searches :**

```sql
SELECT * FROM orders WHERE order_id = 1001;
```

- **Range queries :**

```sql
SELECT * FROM customers WHERE age BETWEEN 30 AND 50;
```

- **Sorted scans :**
  ORDER BY or GROUP BY clauses.

PostgreSQL Example:

```sql
-- Creating a B-tree index on the "last_name" column
CREATE INDEX idx_customers_lastname ON customers (last_name);
```

This index improves the performance of queries such as:

```sql
SELECT * FROM customers WHERE last_name = 'Smith';
SELECT * FROM customers ORDER BY last_name;
```

Benefits of B-Tree Indexes:

- **Balanced performance:** They offer consistent O(log n) access time across different types of operations (search, insert, delete).
- **Support for sorting and range scans:** Unlike hash indexes, B-trees can efficiently support range conditions.
- **Broad operator support:** B-tree indexes are compatible with most standard comparison operators in SQL.
- **Partial and multicolumn support:** PostgreSQL supports partial B-tree indexes and composite indexes, which can be tailored to specific queries for further optimization.

Trade-offs:

- **Update cost:** B-tree indexes must be rebalanced upon insertions and deletions, which adds overhead.
- **Space usage:** Although more compact than some other index types, B-trees still consume additional disk space, especially when indexing large text fields or multiple columns.
- **Inefficiency with non-scalar data types:** B-trees are not optimal for indexing unstructured or multidimensional data, such as geometric types or full-text search.

PostgreSQL Internal Behavior:

PostgreSQL uses write-ahead logging (WAL) to ensure durability of B-tree modifications. Additionally, it incorporates autovacuum and reindexing tools to manage index bloat, which can occur over time due to tuple dead versions (Tuples are not deleted immediately after update or delete operations).

Variants:

- **Unique B-tree indexes:** Enforce uniqueness across the indexed column(s).
- **Expression indexes:** Allow creating B-tree indexes on computed expressions.

```
1 SELECT * FROM customers WHERE last_name = 'Smith';
2 SELECT * FROM customers ORDER BY last_name;
```

Performance Considerations: Query planners in PostgreSQL automatically decide when to use a B-tree index based on selectivity estimates, cost models, and statistics collected via ANALYZE. The user can also inspect index usage through the EXPLAIN or EXPLAIN ANALYZE command.

1.2.3 Hash Indexes:

Hash indexes are specialized data structures designed to support rapid equality lookups using a hash function. Unlike B-tree indexes that preserve order and support range queries, hash indexes focus on direct, exact-match queries by converting key values into a fixed-size hash code using a deterministic algorithm. In PostgreSQL, hash indexes are implemented as static hashing mechanisms and are optimized for queries using the equality (=) operator.

A hash index works by applying a hash function to the key and mapping it to a specific location (bucket) in the index. This structure is particularly effective for queries of the form:

```
SELECT * FROM employees WHERE employee_id = 101;
```

When a hash index is used, the engine computes the hash code for employee_id = 101 and retrieves the corresponding bucket containing matching rows, thus significantly reducing lookup time in large datasets.

```
-- Creating a hash index on employee_id
CREATE INDEX idx_employee_hash ON employees USING hash (employee_id);
```

To ensure the use of the hash index during execution, the query planner considers statistics and available operators. For example:

```
SELECT * FROM employees WHERE employee_id = 1002;
```

The above query is a candidate for optimization using the hash index, provided that the planner determines it to be more efficient than a sequential or B-tree scan.

Characteristics of Hash Indexes in PostgreSQL:

- **Equality-based indexing:**Only queries using = can leverage hash indexes; they are not suitable for <, >, BETWEEN, or LIKE operators.
- **Fast constant-time performance:** On average, they provide O(1) access time for exact matches, assuming minimal hash collisions.
- **No ordering guarantee:** Unlike B-trees, hash indexes do not maintain any ordering of keys.
- **Lesser versatility:**They do not support index-only scans or partial indexes as flexibly as B-trees.

Benefits:

- **High performance for exact-match queries:** Particularly useful when querying on keys with high cardinality where range queries are unnecessary.

- **Memory-efficient lookup structure:** Less overhead for maintaining node pointers and tree balance.
- **Simplicity of implementation and predictability of access time:** especially useful in in-memory or low-disk I/O workloads.

Limitations and Considerations:

- **No support for range queries or sorting.**
- **Historically non-durable:** Prior to PostgreSQL 10, hash indexes were not WAL-logged and hence unsafe for crash recovery. This limited their use in production systems. Since version 10, hash indexes are fully WAL-logged, making them crash-safe and durable .
- **Higher collision risk:** Poor choice of hash functions or non-uniform distribution of data may lead to clustering and performance degradation.

Example Use Case: Hash indexes are well-suited for indexing lookup tables or columns frequently queried using equality conditions, such as user authentication systems or exact-match filters in data warehouses.

```
-- Example in a user login system
CREATE INDEX idx_login_hash ON users USING hash (username);

SELECT * FROM users WHERE username = 'john.doe';
```

In this scenario, a hash index allows PostgreSQL to retrieve the row associated with 'john.doe' in constant time, assuming low collision.

Table 1.1: Comparison of B-tree and Hash Index Types

Index Type	Lookup (Equality)	Lookup (Range)	Ordering Support	Index-only Scan
B-tree	O(log n)	Yes	Yes	Yes
Hash	O(1)	No	No	Limited

1.2.4 Generalized Search Tree (GiST) Indexes

The Generalized Search Tree (GiST) is a flexible and extensible indexing framework in PostgreSQL that allows developers to define custom indexing strategies for complex data types beyond the traditional numeric or text-based fields. Introduced to handle a broader range of queries, GiST indexes are not bound by the constraints of B-tree or hash indexes and can be adapted to support various types of queries such as nearest-neighbor search, full-text search, geometric queries, and more .

Unlike B-tree or hash indexes that are tightly coupled to specific operations (e.g., ordering or equality), GiST is operator-agnostic, making it suitable for any domain where a notion of proximity, containment, or overlap is defined.

Internal Design and Operation:

GiST follows a tree-based structure similar to B-trees but allows developers to customize the behavior of internal nodes, leaf pages, and traversal strategies. To achieve this flexibility, GiST requires a set of user-defined methods, including:

- **Consistent() -** Determines whether a key satisfies a predicate.
- **Union()** – Calculates a bounding representation for a node.
- **Compress() and Decompress()** – Used for space optimization.
- **Penalty()** – Guides insertion into the most appropriate subtree.
- **PickSplit()** – Defines how to split nodes when overflowing.

These extensibility points allow the GiST framework to be adapted for diverse data types and access patterns.
Example Use Case in PostgreSQL: A common application of GiST is in geospatial indexing, particularly when working with the PostGIS extension for spatial data:

```
-- Create a GiST index on a geometry column
CREATE INDEX idx_location_gist ON landmarks USING gist (location);
```

This index supports queries such as:

```
-- Find all landmarks within a specific bounding box
SELECT * FROM landmarks
WHERE location && ST_MakeEnvelope(3.0, 36.0, 6.0, 39.0, 4326);
```

The && operator checks for bounding box overlap, and the GiST index enables efficient spatial filtering based on these geometric conditions.

Key Applications of GiST Indexes:

- **Geometric and spatial data -** e.g. point-in-polygon, intersection, proximity queries.
- **Full-text search** – GiST can index tsvector columns in text search.
- **Network and range types** – such as IP addresses and numeric ranges.
- **Custom data types** – Developers can define custom behaviors for user-defined types.

Performance Considerations:

GiST indexes, due to their general-purpose nature, do not offer the same level of performance as specialized indexes like B-trees for standard operations. However, their adaptability makes them essential in domains requiring complex filtering logic or multidimensional queries.

Table 1.2: Features Supported by GiST Indexes

Feature	Supported by GiST
Equality Matching	Yes
Range Queries	Yes
Nearest-Neighbor Searches	Yes
Ordered Retrieval	Not guaranteed
Index-only Scans	No (usually)

Limitations:

- **More disk space:** e.g. point-in-polygon, intersection, proximity queries.
- **Slower insertions and updates:** GiST can index tsvector columns in text search.
- **No guaranteed order:** such as IP addresses and numeric ranges.
- **Complexity in definition:** Developers can define custom behaviors for user-defined types.

1.2.5 Generalized Inverted Index (GIN) / Reverse Indexes (Conceptual Overlap)

The Generalized Inverted Index (GIN) is a powerful indexing mechanism in PostgreSQL, particularly effective for indexing composite types like arrays, full-text documents (tsvector), and JSONB fields where items within a column value need to be indexed individually [17]. According to the official PostgreSQL documentation, GIN indexes are designed for cases where you want to query for rows containing specific elements within such composite values, making them highly suitable for full-text search or querying array elements [17]. This contrasts with B-tree or Hash indexes which typically operate on the entire column value.

Internal Structure and Behavior

In contrast to B-tree or GiST indexes, GIN builds a reverse mapping from elements to their containing rows. For example, when indexing a text column using GIN and full-text search, each unique word (lexeme) is associated with a list of rows (posting list) where that word appears. This design is especially suitable when a column stores composite or decomposable values, such as:

- Arrays (e.g., text[], int[])
- Full-text data (tsvector)
- JSON/JSONB structures
- Range types

In essence, the GIN index maintains a large number of keys (tokens or elements), each associated with a list of positions (TIDs – tuple IDs) where that token occurs.

Avantages (Benefits)

- **Efficient multi-key searches:**Excellent for querying composite data types *(arrays, tsvector, JSONB)* where multiple elements inside a column need to be searched simultaneously.[16]
- **Ideal for semi-structured data:**Well-suited for indexing JSONB and full-text data, supporting complex search scenarios.[26]
- **Reverse index structure:** Implements an inverted index mapping tokens to tuple IDs, similar to information retrieval systems used in search engines.[26]
- **Fast read/query performance:**Queries that check for the presence of multiple tokens are highly optimized compared to B-tree indexes.[16]
- **Supports full-text search natively:**Integral to PostgreSQL's full-text search capabilities with native support for *tsvector and tsquery.*[16]

Inconvénients (Trade-offs)

- **Slower index creation:**Building GIN indexes is more time-consuming compared to B-tree due to the complexity of maintaining posting lists for each token.[16]
- **Higher space usage:**GIN indexes consume more disk space because they store detailed mappings of tokens to row locations (posting lists).[16]
- **Costly updates and inserts:**Maintaining posting lists incurs higher overhead during data modifications; PostgreSQL mitigates this via a *"fastupdate"* option that delays merging changes.[16]
- **No native prefix or partial match support:**Unlike some specialized extensions *(e.g., pg_trgm)*, GIN does not support prefix or approximate matching by default.[16] [26]

GIN Example in PostgreSQL: Consider an example where documents are stored with their full-text representation:

```
-- Create a GIN index on a tsvector column for full-text search
CREATE INDEX idx_docs_gin ON documents
USING gin(to_tsvector('english', content));
```

Queries using the index might look like:

```
-- Full-text search using a GIN index
SELECT * FROM documents
WHERE to_tsvector('english', content)
@@ to_tsquery('artificial & intelligence');
```

Internally, the GIN index maps each token (e.g., 'artificial', 'intelligence') to a list of rows that include the token. This is functionally similar to a reverse index in information retrieval systems, where words are mapped to document IDs.

Performance Characteristics

Table 1.3: Features of GIN Indexes

Feature	GIN Index
Indexing speed	Slower (especially for large datasets)
Query performance (read)	Fast for multi-key lookups
Space consumption	Higher than B-tree
Supports partial match / prefix	No (except with some extensions)
Update cost	Higher than B-tree or hash

GIN supports fast read performance, especially when queries need to check for the presence of multiple elements. However, insertion and update costs are typically higher due to the maintenance of large posting lists.

To alleviate this, PostgreSQL offers the option to build GIN indexes with fastupdate enabled (default), which stores recent updates in a pending list and merges them later:

```
CREATE INDEX idx_jsonb_gin ON data USING gin(jsonb_column)
WITH (fastupdate = on);
```

1.2.6 Bitmap Indexes:

Bitmap indexes are an alternative indexing structure that is especially efficient for columns with low cardinality — that is, columns with a relatively small number of distinct values, such as gender, status flags, or boolean fields. While not natively implemented in PostgreSQL as a user-defined index type, bitmap indexes are utilized internally by the

query planner through bitmap index scans, particularly when combining multiple indexes in a single query. Bitmap indexes have been extensively studied and applied in data warehousing, decision support systems, and OLAP environments due to their ability to facilitate fast, set-based operations on large data volumes.

Concept and Structure

A bitmap index represents each distinct value in a column by a bit vector, where each bit corresponds to a row in the table. If a row contains that value, the corresponding bit is set to 1; otherwise, it is 0. For example, in a column status with possible values 'active', 'inactive', and 'pending', the system constructs three bitmaps: This binary

Table 1.4: Status Flags per Row

Row ID	active	inactive	pending
1	1	0	0
2	0	1	0
3	0	0	1
4	1	0	0

representation allows bitwise operations (AND, OR, NOT) to efficiently resolve complex queries such as multiple conditions, range filtering, or aggregation.

Internal Use in PostgreSQL

Although PostgreSQL does not expose a dedicated CREATE INDEX ... USING bitmap syntax, the bitmap index scan strategy is automatically chosen by the query planner when multiple simple indexes are combined. This is most evident when querying with multiple conditions:

```
-- Suppose there are B-tree indexes on both columns
SELECT * FROM users
WHERE gender = 'F' AND status = 'active';
```

If each condition uses an index on its respective column, PostgreSQL may create bitmap index scans and then AND the results:

```
EXPLAIN ANALYZE
SELECT * FROM users
WHERE gender = 'F' AND status = 'active';
```

Expected output:

```
Bitmap Heap Scan on users
  Recheck Cond: ((gender = 'F') AND (status = 'active'))
  -> BitmapAnd
    -> Bitmap Index Scan on idx_gender
       Index Cond: (gender = 'F')
    -> Bitmap Index Scan on idx_status
       Index Cond: (status = 'active')
```

This plan shows that PostgreSQL internally transforms B-tree indexes into bitmaps for fast combination and filtering.

Advantages of Bitmap Indexes:

- Efficient for low-cardinality attributes – especially in analytical workloads.
- Excellent performance in complex queries involving multiple conditions.
- Compact storage due to the binary representation.
- Set-based operations (e.g., OR, AND) are extremely fast using bitwise logic.

Limitations

- **Not efficient for high-cardinality attributes,** where the number of distinct values is close to the number of rows — the bitmaps become large and sparse.
- **Frequent updates and inserts** can be expensive, as they require modifying potentially many bitmaps.
- **Not available as a first-class index in PostgreSQL;** requires implicit use via B-tree indexes and planner logic.

In some columnar databases (e.g., ClickHouse, Vertica, Amazon Redshift) or dedicated data warehouse engines, bitmap indexes are implemented natively and optimized for large-scale analytical queries.

1.2.7 Index Selection Strategies in PostgreSQL

Indexing plays a critical role in optimizing query performance in PostgreSQL, as it facilitates faster data retrieval by reducing the number of rows scanned. The choice of index and the indexing strategy to adopt depend heavily on the query patterns, table size, and column characteristics, which are intricately tied to PostgreSQL's query processing and optimization mechanisms [14]. This section explores the strategies employed by PostgreSQL to select the most suitable index and highlights the factors influencing index selection, particularly its cost-based optimizer, and the ways in which PostgreSQL decides when and which index to use.

PostgreSQL Cost-Based Optimization

PostgreSQL's query planner relies on a cost-based optimizer, a concept pioneered in early relational systems [20], that evaluates various execution plans by assigning costs to them. These costs are based on factors such as:

- **Disk I/O:** Index scans typically reduce disk I/O by accessing fewer rows.
- **CPU Utilization:** Operations such as sorting and joining can add computational overhead.
- **Memory Usage:** Queries that involve large intermediate results may require more memory, impacting performance.

The planner considers both sequential scans and index scans. A sequential scan involves reading the entire table row by row, while an index scan allows faster access to rows that match specific criteria. For most situations, the planner can choose between these two options, and the decision largely depends on the estimated cost of each approach.

In situations where multiple indexes are available, the planner evaluates Bitmap Index Scans, which can combine results from multiple indexes to create a more efficient query plan. The planner also evaluates whether covering indexes (indexes that contain all the columns required by the query) can be used to avoid additional table lookups.

1.3 Query Optimization Techniques

Query optimization in relational database systems is a critical process aimed at improving the performance of SQL queries by finding the most efficient execution plan. This typically involves transforming a declarative SQL query into an imperative execution plan that can be run by the database engine. As established in foundational database literature [11], the optimizer explores a space of possible plans, estimating the cost of each based on data statistics and system parameters, and selects the one with the lowest estimated cost . PostgreSQL, like other relational database management systems, incorporates a sophisticated query planner and optimizer that analyzes queries and determines the optimal strategy for data retrieval. The efficiency of a query plan can significantly impact the system's overall performance, particularly in applications where large volumes of data are involved.

The optimization process begins with query parsing and transformation into an internal representation, followed by the generation of multiple alternative execution plans. These plans are evaluated based on cost estimates, which are calculated using statistical information about the data, such as table sizes, index availability, data distribution, and join selectivity. The plan with the lowest estimated cost is typically chosen for execution.

One of the fundamental optimization strategies involves the use of indexes, which allow the database to quickly locate specific rows without scanning entire tables. Indexes can drastically reduce I/O operations and query response times, especially for large datasets. However, the effectiveness of an index depends on the nature of the query, the type of index used, and how well it aligns with the query predicates. PostgreSQL supports a

variety of index types, such as B-tree, Hash, GIN, and GiST, each suited for specific use cases.

Another important technique is predicate pushdown, where filtering conditions are applied as early as possible during query execution to minimize the amount of data processed in later stages. This technique reduces intermediate result sizes and helps streamline operations such as joins and aggregations.

The optimizer also applies join reordering, which rearranges the sequence of join operations to reduce computational costs. By prioritizing joins with smaller intermediate results or higher selectivity, the database can avoid unnecessary computations and memory usage. PostgreSQL employs both heuristic and cost-based strategies to determine the optimal join order.

In complex queries involving subqueries or common table expressions (CTEs), subquery flattening and CTE inlining are used to simplify query structures and integrate them into the main execution plan. This enhances the optimizer's ability to evaluate different execution paths more effectively.

Moreover, PostgreSQL utilizes statistics and histograms to estimate the selectivity of expressions, which directly affects the cost calculations of execution plans. Accurate and up-to-date statistics allow the optimizer to make better decisions. Regularly running ANALYZE ensures that the planner has current information about data distributions.

In summary, PostgreSQL's query optimization techniques rely on a combination of cost estimation, execution plan generation, and heuristic rules to enhance performance. These techniques aim to reduce resource consumption, improve execution speed, and ensure scalability across different workloads. Understanding and leveraging these optimization mechanisms is essential for database administrators and developers seeking to maximize query efficiency and overall system performance.

1.4 Machine Learning Applications in Database Systems:

Machine learning (ML) is transforming database management by enhancing performance, efficiency, and scalability, leading to the broader concept of "learned database systems" [23]. Key applications within this domain include:

- **Query Optimization:** ML techniques improve upon traditional rule/cost-based optimizers by learning from historical execution data to select optimal query plans, particularly beneficial for complex queries where conventional heuristics fall short.

- **Automated Database Tuning:** ML algorithms automate the fine-tuning of database parameters (buffer sizes, memory allocation) through reinforcement learning and genetic algorithms, reducing manual intervention while maximizing performance for specific workloads.

- **Intelligent Index Recommendation:** ML models analyze query patterns and workload characteristics to recommend optimal indexing strategies, balancing read performance improvements against write operation costs and maintenance overhead.
- **Anomaly Detection:** Unsupervised learning algorithms identify abnormal patterns in query execution times and resource utilization, enabling early detection of potential issues before they escalate into serious performance problems.
- **Query Performance Prediction:** ML models predict execution times for queries before execution, allowing systems to implement preemptive optimization measures for complex operations.
- **Data Cleaning and Transformation:** ML automates identification and correction of data errors, detecting outliers, missing values, and inconsistencies while suggesting appropriate transformations.

Automated Database Tuning:

Traditional manual tuning by DBAs is labor-intensive and error-prone. ML-driven automated tuning continuously optimizes database performance through:

- **Performance Metrics Collection:** Gathering query response times, resource utilization data, and execution plans.
- **Feature Extraction:** Identifying relevant metrics as inputs for ML models.
- **Optimization Algorithms:** Using reinforcement and supervised learning to adjust parameters for optimal outcomes.
- **Dynamic Adaptation:** Continuously evaluating and adjusting configurations as workloads evolve.

1.4.1 Query Performance Prediction:

Query performance prediction is a critical aspect of database management that allows administrators and systems to anticipate how a database will respond to specific queries. By predicting query performance, it is possible to make proactive adjustments to the database, such as optimizing query plans, tuning parameters, or reorganizing indexes, to ensure that queries execute efficiently. This technique is particularly important in environments with dynamic workloads, where query patterns may change rapidly, making real-time adjustments difficult without preemptive actions.

In the context of machine learning, query performance prediction involves training models to estimate various performance metrics, such as query execution time, resource consumption (e.g., CPU, memory, disk I/O), or the number of system resources required

to process a given query. Research has shown that ML models can offer insights into query execution time, sometimes complementing or improving upon traditional optimizer cost models [9]. Accurate query performance predictions can lead to better query optimization strategies and improve the overall efficiency of database systems.

1.4.2 Intelligent Index Recommendation Systems:

Intelligent index recommendation systems automatically suggest optimal database indexes based on query patterns and workloads, significantly improving query performance while reducing DBA workload. These systems are essential as databases grow in size and complexity, where manual index selection becomes impractical.
Key Benefits:

- **Automatic Indexing:** Continuous analysis of query patterns to dynamically create and modify indexes
- **Improved Query Performance:** Optimized query execution times through workload-specific indexing
- **Reduced Manual Effort:** Decreased reliance on DBA expertise
- **Cost Reduction:** Elimination of redundant indexes, reducing storage costs

1.5 Natural Language Processing and Feature Extraction for SQL Queries:

Natural Language Processing (NLP) techniques, and more broadly, methods for syntactic and semantic analysis of structured languages like SQL, are crucial for understanding query intent. While NLP is often discussed for translating human queries into SQL [24], similar principles apply when extracting meaningful features from existing SQL queries. Feature extraction, in this context, aims to identify and represent key structural and semantic elements of SQL queries. The Role of such analysis in SQL Query Processing for recommendation includes understanding:

- **Complex Query Structures:** Translating complex clauses (JOIN, GROUP BY, WHERE)

Feature Extraction Methods:

- **Rule-Based:** Predefined patterns mapping language to SQL
- **Machine Learning:** Models trained on query-SQL pairs

SQL Query Representation for Machine Learning: Key representation approaches include:

- **Bag-of-Words:** Simple token sets without order
- **Token Embeddings:** Vector representations capturing semantic relationships
- **Abstract Syntax Trees:** Hierarchical structures reflecting query logic
- **Sequence Representation:** Ordered token sequences for sequential models
- **Graph Representation:** For complex multi-table relationships

Applications:

- SQL query optimization
- Automated report generation

Challenges:

- Complex query structures
- Database schema variability

TF-IDF for Text Feature Extraction:

TF-IDF (Term Frequency-Inverse Document Frequency) is a statistical weighting scheme, with foundational work by Salton and Buckley [19], commonly used in information retrieval and text mining to evaluate the importance of a word in a document relative to a collection of documents or corpus[12]. It's calculated as: [TF-IDF(t, d) = TF$(t, d) \times$ IDF(t)] Where:

- **Term Frequency (TF):** Measures how often a term appears in a document
- **Inverse Document Frequency (IDF):** Measures how rare a term is across all documents

The concept of Inverse Document Frequency (IDF), a crucial component of TF-IDF, was notably introduced to address the issue that not all terms are equally important for retrieval; terms that are common across many documents are less discriminative. Spärck Jones pioneered this idea, proposing that the specificity of a term can be quantified by an inverse function of the number of documents in which it occurs [22]. This insight fundamentally improved the performance of information retrieval systems by giving higher weights to rarer, more informative terms, a principle later combined with term frequency to form the widely adopted TF-IDF weighting scheme.

This method assigns higher weights to terms frequent in a specific document but uncommon across the corpus, capturing distinctive features while reducing common word importance.

Applications in SQL Context:

- **Query Clustering:** Grouping similar SQL queries
- **Query Optimization:** Suggesting performance improvements
- **Index Selection:** Identifying frequently accessed columns for indexing

Advantages:

- Simple to implement
- Computationally efficient
- Easily interpretable

Limitations:

- Creates sparse matrices
- Ignores term context

1.6 Related Work

The challenge of automating database optimization, particularly index selection, has been a subject of extensive research for decades. Our work builds upon foundational concepts in database management while leveraging modern machine learning techniques. This section reviews the evolution of index recommendation systems and situates our approach within the broader context of learned database systems.

1.6.1 Classical and Heuristic-Based Index Advisors

Early efforts in automating index selection were dominated by expert systems and cost-based "what-if" analysis. A seminal work in this area is the AutoAdmin project from Microsoft Research, which led to tools like the Database Tuning Advisor (DTA) for Microsoft SQL Server. As described by Chaudhuri and Narasayya [3], these systems operate by enumerating a set of candidate indexes and using the database's query optimizer to estimate the performance improvement for a given workload if those indexes were created.

While highly influential and effective, these classical approaches have several limitations:

- They are computationally intensive, as they must simulate the cost of numerous potential index configurations.
- Their accuracy is fundamentally tied to the accuracy of the database's own cost model, which can sometimes be imprecise [9].

- They primarily focus on recommending single or multi-column B-Tree indexes and are less equipped to handle the diverse index types available in modern systems like PostgreSQL (e.g., GiST, GIN).

Our work diverges from this paradigm by learning directly from query characteristics rather than relying on a "what-if" analysis via the optimizer's cost model.

1.6.2 Machine Learning and "Self-Driving" Database Systems

The advent of machine learning has ushered in a new era of "learned" or "self-driving" database systems, a vision articulated by Pavlo et al. [**pavlo2017self**]. These systems aim to automate all aspects of database administration, from parameter tuning to physical design.

In the domain of index recommendation, ML-based approaches replace or augment heuristic-driven methods. **OtterTune**, for instance, uses machine learning to recommend optimal database configuration knobs, and similar principles have been applied to indexing [**pavlo2017self**]. Many modern approaches frame index selection as a reinforcement learning (RL) problem, where an agent learns to make indexing decisions (e.g., create or drop an index) to maximize a long-term reward, such as query performance.

Other works have used supervised learning. For example, some models predict the execution time of a query given a certain index configuration or directly regress the utility of adding a candidate index. However, a significant portion of this research focuses on the problem of *column selection* for B-Tree indexes. In contrast, our thesis addresses the orthogonal but equally important problem of selecting the optimal *index type* (B-Tree, Hash, GIN, GiST, Bitmap) from PostgreSQL's rich ecosystem, a decision that depends heavily on the query's operators and structure.

1.6.3 Natural Language Processing for SQL Query Analysis

A key component of our methodology is the treatment of SQL queries as text documents for feature extraction. This is an application of Natural Language Processing (NLP) techniques to a structured language. While NLP is more commonly associated with translating natural language into SQL queries, as seen in systems like SQLizer [24], the underlying techniques for representing text are highly relevant.

Using TF-IDF to featurize SQL queries allows our model to capture the relative importance of specific keywords (e.g., 'LIKE', 'BETWEEN'), operators ('&&', '@@'), and column names that are strong indicators for a particular index type. This approach is lightweight and avoids the complexity of parsing queries into abstract syntax trees or analyzing query execution plans, which can be brittle and vary between database versions. Our work demonstrates that this NLP-inspired feature engineering is highly effective for the specific task of index type classification.

1.6.4 Positioning of This Thesis

This thesis carves out a specific niche within the landscape of database optimization. While classical advisors are exhaustive but limited in scope, and modern ”self-driving” systems target full automation through complex models like RL, our work provides a focused, lightweight, and practical solution.

The primary contribution is the development of a supervised learning model that directly addresses the **index type recommendation problem** in PostgreSQL. By leveraging TF-IDF for feature extraction from raw SQL queries, we create a system that can provide immediate, query-specific recommendations without requiring a historical workload or complex ”what-if” analysis. This approach fills a gap in the literature by demonstrating that a simple yet powerful classification model can effectively automate one of the more nuanced decisions faced by database administrators.

1.7 Random Forest for Indexing Recommendation

The Random Forest algorithm, introduced by Breiman (2001)[2], is particularly well-suited for tasks like index recommendation due to its inherent ability to handle high-dimensional data and complex interactions between features without extensive pre-processing or parameter tuning. As noted in various applications, Random Forests are robust to overfitting, especially when a large number of trees are grown, and they provide useful internal estimates of error, feature importance, and data proximity [7]. This makes them an attractive choice for analyzing the nuanced features extracted from SQL queries, which can often be sparse and numerous, to predict optimal indexing strategies.

1.7.1 Algorithm Overview

Random Forest builds decision trees through:
Bootstrap Sampling: Creating random data subsets with replacement
Random Feature Selection: Considering only a subset of features at each node.
Building Independent Trees: Training each tree on its respective data subset.
Voting/Averaging: Combining predictions (majority vote for classification, mean for regression)

1.7.2 Application to SQL Query Optimization

Random Forest can effectively recommend indexing strategies by:

Classifying SQL queries based on execution patterns and performance metrics Identifying the most relevant features for query optimization Predicting execution costs to enable efficient query planning Recommending optimal indexing choices based on query characteristics

1.7.3 Advantages

High Accuracy: Provides reliable indexing recommendations even with complex queries.
Feature Importance: Identifies which query attributes most influence indexing decisions.
Robustness: Handles diverse query types and database conditions.
Adaptability: Can learn from historical query performance data.

1.7.4 Implementation Considerations

When implementing a Random Forest-based indexing recommendation system:

Train on diverse query samples with measured performance metrics Include relevant features such as query complexity, table sizes, and join conditions Tune key parameters (number of trees, maximum depth, feature selection) Validate recommendations against actual query performance improvements

1.8 Conclusion

This chapter has provided a comprehensive review of the foundational concepts and state-of-the-art in database indexing and optimization. We began by detailing the fundamental index structures available in PostgreSQL, from the ubiquitous B-Tree to specialized types like GIN, GiST, and Hash, highlighting their distinct operational trade-offs. We then explored the broader landscape of query optimization and the recent advancements in automated database tuning.

Our review of related work established a clear picture of the evolution from classical, cost-based index advisors to modern, AI-driven ”self-driving” systems. This analysis revealed a specific gap in the literature: a lack of lightweight, query-centric models focused on recommending the optimal *index type*, as opposed to merely selecting columns. Existing solutions often rely on extensive workload analysis or complex learning models, rather than the intrinsic properties of a single query.

This thesis aims to fill that gap. By framing index type selection as a classification problem and leveraging NLP techniques to interpret raw SQL, we propose a novel and practical approach. The following chapter will detail the methodology developed to build and train a model for this specific challenge, laying the groundwork for a system that can provide intelligent, real-time indexing recommendations.

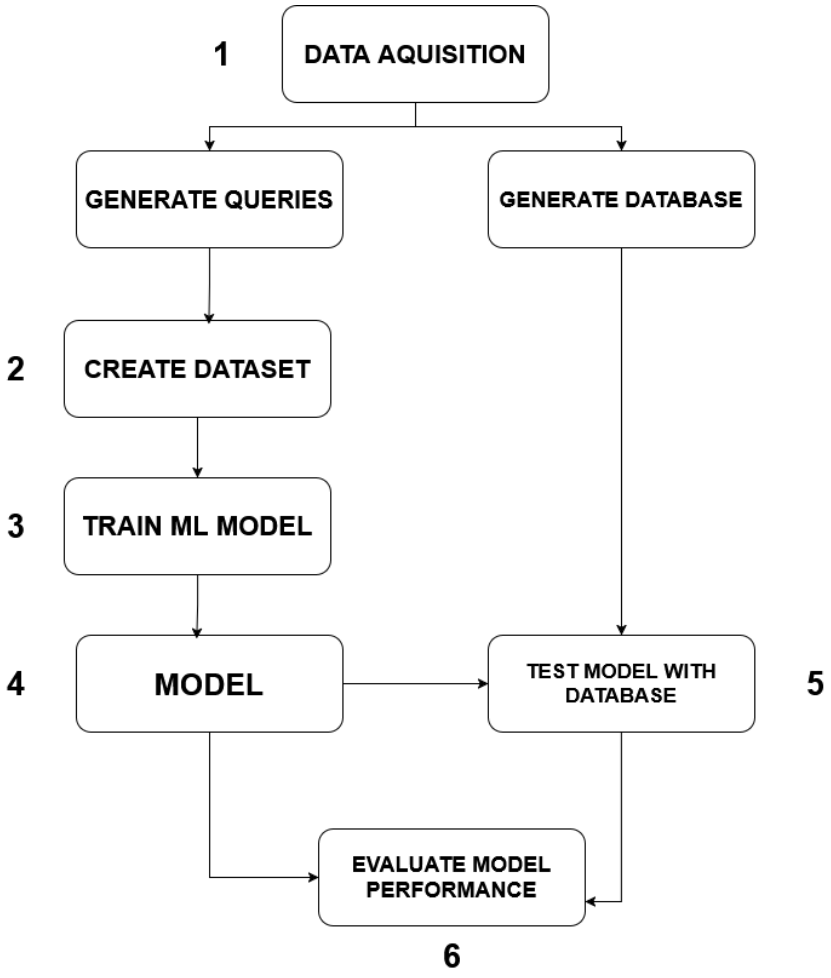
Chapter 2

Methodology

2.1 Introduction

To address the index selection challenge identified in the literature, this chapter presents the systematic methodology designed and implemented to build our AI-driven recommendation system. We move from theory to practice, outlining a complete and reproducible workflow that transforms the abstract problem of query optimization into a concrete supervised learning task. This blueprint details every critical stage of our approach, beginning with the creation of a large-scale, controlled database environment populated with realistic synthetic data. We then describe the novel use of a large language model for generating a diverse corpus of SQL queries, the feature engineering pipeline used to convert these queries into a machine-readable format, and the empirical labeling strategy that establishes a ground truth for model training. The chapter culminates in the development and evaluation plan for a Random Forest classifier and the design of an interactive user interface, providing a solid foundation for the experiments that follow. The overall research workflow, from data acquisition to model evaluation, is depicted in Figure 2.1 and serves as a guide for the subsequent sections.

Figure 2.1: Overall methodological workflow for the development and evaluation



2.2 Methodology for Intelligent Index Recommendation

This research presents a systematic approach to developing an intelligent index recommendation system for PostgreSQL using machine learning. The methodology integrates database optimization with supervised learning to automatically recommend optimal indexing strategies for SQL queries.

2.2.1 Research Design

A quantitative experimental approach was adopted, transforming SQL query optimization into a supervised learning problem. Each SQL query is treated as an instance, with the most effective index type as its label. The Random Forest classifier was selected for its robustness with structured data and interpretability.

2.2.2 Implementation Framework

Synthetic Database Generation

A controlled database environment was constructed with:

- Realistic schema design (customers, orders, products, order_items, suppliers)
- Diverse data types and relational complexity
- Python Faker-generated synthetic data (1.5m records)
- Normalized structure with intentional variability in cardinality and selectivity

Query Generation and Processing

- Diverse SQL queries generated via large language model API
- Queries varied in complexity, structure, and patterns
- Preprocessing through tokenization and feature extraction using NLP techniques
- TF-IDF transformation to create numerical feature vectors

Machine Learning Approach

- Supervised learning with Random Forest classifier
- Queries labeled with empirically determined optimal index types
- Performance evaluation using accuracy,
- Model tuned for generalizability across query patterns

2.2.3 PostgreSQL Environment

PostgreSQL 15 was selected for its robust indexing capabilities (B-Tree, Hash, GiST, GIN) and mature query optimizer. The system was configured with tuned memory and parallelism parameters to simulate production conditions, with diagnostic tools (EXPLAIN, ANALYZE) providing ground truth for labeling.

2.2.4 Practical Application

A lightweight user interface bridges theory and practice, allowing users to input queries and receive intelligent indexing recommendations without requiring deep knowledge of database internals, making the system accessible for practical database optimization.

2.3 SQL Query Generation and Analysis

This research develops a recommendation system for optimal database indexing strategies using machine learning. A diverse SQL query dataset was generated semi-automatically using the Gemini AI API and manual validation, ensuring queries vary in complexity, structure, and index relevance.

2.3.1 Query Generation Process

The process focused on three objectives:

1. **Diversity:** Including simple selections, multi-table joins, subqueries, and complex filtering logic.
2. **Controlled Complexity:** Categorizing from basic single-table lookups to advanced analytical queries.
3. **Index Relevance:** Incorporating patterns relevant to various index types (B-Tree, Hash, GiST, GIN).

Gemini AI API automated generation based on natural language prompts, creating realistic and diverse SQL queries. All queries were validated against PostgreSQL using EXPLAIN ANALYZE to ensure correctness and extract performance metadata.

2.3.2 Query Processing and Feature Extraction

Queries underwent preprocessing including:

- **Cleaning:** Removal of redundant whitespace and normalization
- **Tokenization:** Breaking queries into keywords, identifiers, operators, and literals
- **Feature Extraction:** Using TF-IDF to represent token importance

2.4 Dataset Creation and Labeling

This research required a labeled dataset to train a machine learning model that recommends optimal indexing strategies for SQL queries. Each query was associated with the index type that maximizes its performance on PostgreSQL.

2.4.1 Dataset Creation Process

A synthetic relational database was populated with diverse data using Python Faker script. SQL queries of varying complexity were generated programmatically and enhanced with Gemini AI for linguistic diversity. The dataset spans from simple SELECT statements to complex queries with multiple joins and nested subqueries. Each query was executed under different index configurations, with performance metrics collected via EXPLAIN ANALYZE. Additional metadata included tokenized representations, TF-IDF vectors, and structural features like join count.

2.4.2 Defining the Optimal Index (Labeling Strategy)

The ”best” indexing method was defined as the one resulting in the shortest execution time, a critical performance metric. To obtain this ground truth, each query was executed multiple times with various index types (B-Tree, Hash, GiST, GIN, and simulated Bitmap effects) applied to relevant columns. PostgreSQL's EXPLAIN ANALYZE command was instrumental in this labeling process. As detailed in the PostgreSQL documentation, EXPLAIN ANALYZE not only shows the query plan but also executes the query and displays the actual run times, row counts, and other statistics, providing a realistic measure of performance [18]. This allowed for the collection of empirical execution statistics, including time, buffer hits, and I/O cost, which formed the basis for our labeling strategy. When execution times were nearly identical, B-Tree was selected as the default due to its general efficiency. Cache effects were minimized between executions to ensure fair comparison.

2.4.3 Final Dataset Structure

The labeled dataset consists of:

- **Features:** Characteristics of SQL queries derived through tokenization and TF-IDF
- **Labels:** The optimal index type for each query

Key features include:

- Query structure (*WHERE clauses, JOIN operations*)
- Number of conditions and tables involved
- Types of operations (equality, range)

- Presence of aggregate functions

This structured dataset provides the foundation for training machine learning models to predict the most suitable index type for any given SQL query.

2.5 Machine Learning Model Development

This research employs a machine learning approach to recommend optimal indexing strategies for SQL queries based on their characteristics. The model development process involved careful selection of a classifier, robust training procedures, and comprehensive evaluation metrics.

2.5.1 Choice of Classifier: Random Forest

Random Forest was selected as the classifier for this index recommendation system due to its numerous advantages:

- **Handling Complex Data:** Processes high-dimensional SQL query features effectively
- **Interpretability:** Provides feature importance metrics, revealing which query aspects most influence index selection
- **Scalability:** Efficiently handles large datasets without excessive computational demands

2.5.2 Model Training Process

The training process followed these key steps:

- **Data Preparation:** The labeled dataset was split into training (80
- **Model Initialization:** RandomForestClassifier was initialized with appropriate hyperparameters

```
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
```

- **Training:** The model learned patterns between SQL query features and optimal index types

2.5.3 Evaluation Metrics

Model performance was evaluated using standard classification metrics:

- **Accuracy:** Ratio of correct predictions to total predictions

Confusion Matrix Analysis

The confusion matrix provided detailed insights into model performance across different index types:

- Each cell represents counts of actual vs. predicted index types
- Diagonal elements indicate correct predictions
- Off-diagonal elements reveal confusions between index types

This analysis helped identify which indexing methods were frequently confused (e.g., B-Tree vs. Hash) and guided targeted improvements to the model. The confusion matrix was visualized using heatmaps to better interpret multi-class classification results. The evaluation metrics demonstrated the model's ability to recommend appropriate indexing strategies based solely on SQL query characteristics, without requiring execution statistics.

2.6 User Interface Development

The User Interface (UI) is a critical component of any system, as it serves as the bridge between the user and the functionality of the system. In the context of an AI-driven indexing recommendation system for SQL queries, the UI must be intuitive, user-friendly, and capable of conveying complex data and functionality in a clear and accessible way. In this section, we will explore the goals, design, and technology stack for the user interface, as well as its functionality.

2.6.1 Design Goals and Requirements

When designing the user interface (UI) for an AI-driven indexing recommendation system, the focus should be on making the interface as intuitive and user-friendly as possible while also accommodating the complexity of the underlying functionality. The goal is to ensure that users can efficiently interact with the system to generate SQL indexing recommendations and obtain results in a clear and easily understandable format. Below are the key design goals and requirements for the UI:

Simplicity and Clarity

- **Easy Navigation:**The UI should be easy to navigate, with clear and concise instructions that guide the user through the process of inputting SQL queries and receiving indexing recommendations.
- **Minimal Learning Curve:**The design should require minimal training for new users. A clean and intuitive layout will help users understand how to interact with the system quickly.

- **Clear Terminology:**All terms used within the UI should be simple and consistent to avoid confusion, especially when dealing with database-related terminology or SQL queries.

Efficient User Input Mechanism

- **Query Input:**Users should be able to easily input their SQL queries into the system. This can be done through:
  - A text box with support for multi-line inputs to accommodate complex SQL queries.
  - Syntax highlighting to make the query more readable and prevent syntax errors.
- **File Upload:**For convenience, users should be able to upload an SQL file containing multiple queries.

Real-Time Feedback and Recommendations

- **Instant Query Evaluation:**Upon submission of a query, the UI should process the query and provide recommendations for the most suitable indexing method within seconds or minutes, depending on the complexity of the query.
- **Recommended Index Display:**The system should clearly display the recommended index type (e.g., B-Tree, Hash, Bitmap) and any related performance benefits.
- **Justification and Rationale:**Each recommendation should include a brief explanation of why a particular index type is suggested based on the query's structure or its performance needs.

Visual Representation of Results

- **Graphical Visualizations:** Since indexing recommendations often involve performance metrics, the UI should display visualizations (e.g., bar charts, line graphs) showing how different indexing methods affect query execution times or database performance.
- **Comparison Tools:**Users should be able to compare multiple indexing strategies for a single query to evaluate the best solution visually.

2.6.2 Technology Stack

The implementation of this index recommendation system required a thoughtfully selected technology stack to ensure robust functionality, efficient data processing, and user-friendly interaction. This section outlines the primary technologies employed in the development of this system.

Programming Language

Python was selected as the primary programming language for this project due to its versatility and rich ecosystem of libraries for data processing and machine learning:

- **scikit-learn:** Provided the implementation of the Random Forest classifier and evaluation metrics
- **pandas:** Facilitated efficient data manipulation and preprocessing
- **NumPy:** Supported numerical operations and array handling
- **Streamlit:** For building the ui web application
- **matplotlib and seaborn:** Enabled visualization of results and confusion matrices

Python's simplicity and readability also facilitated rapid development and iterative improvement of the machine learning model.

Database System

PostgreSQL was utilized as the database management system due to its advanced indexing capabilities and robust performance:

- **Diverse Index Types:** Provided support for B-Tree, Hash, GiST, GIN, and Bitmap indexes
- **EXPLAIN ANALYZE:** Offered detailed query execution statistics necessary for performance benchmarking
- **psycopg2:** Python adapter used to interface with the PostgreSQL database
- **Schema Flexibility:** Accommodated the synthetic data model with various data types and relationships

PostgreSQL's open-source nature and extensive documentation made it an ideal choice for academic research on database optimization.

User Interface

Streamlit, a Python library for creating web applications for machine learning and data science (Streamlit Inc., n.d.), was employed to develop an interactive web-based user interface that allows users to:

- **Input SQL Queries:** Through a text area with syntax highlighting
- **Visualize Recommendations:** View the recommended index types with confidence scores

- **Inspect Query Plans:** Analyze the execution plan generated by PostgreSQL with and without the recommended index

Streamlit was chosen for its simplicity in creating data-focused web applications using pure Python, eliminating the need for front-end development skills. Its reactive execution model enabled real-time updates as users modify their queries, providing immediate feedback on index recommendations. The combination of Python, PostgreSQL, and Streamlit created a cohesive and efficient technology stack that facilitated both the research aspects of this project and the development of a practical tool for database optimization.

2.7 Conclusion

This chapter has detailed the cohesive and rigorous methodology that forms the core of our research. By synthesizing database management principles with machine learning techniques, we have constructed a novel end-to-end framework for intelligent index recommendation. The process began with the foundational step of creating a controlled, large-scale synthetic environment, enabling the generation of a diverse and representative set of SQL queries. The subsequent application of TF-IDF for feature engineering proved pivotal, transforming raw query text into a structured format that captures semantic and syntactic patterns. Our empirical labeling strategy, grounded in the actual performance metrics from PostgreSQL's EXPLAIN ANALYZE tool, ensures that the model learns from real-world behavior rather than theoretical heuristics. The selection of a Random Forest classifier and the design of an intuitive user interface complete this framework, creating a robust and practical system ready for empirical validation. Ultimately, this methodology establishes a clear and innovative pathway to automating the complex task of index selection, setting the stage for the experimental evaluation presented in the next chapter.

Chapter 3

Experiments and Results

3.1 Introduction

This chapter presents the empirical validation of the methodology developed in Chapter 2. Here, we transition from design to execution, conducting a series of experiments to rigorously evaluate the performance and practical utility of our AI-driven index recommendation system. We begin by detailing the experimental setup, including the dataset splits and the baseline heuristics against which our model is compared. The chapter then systematically presents the results from each stage of the pipeline: the characteristics of the generated data and queries, the outcomes of the feature engineering process, and the training and tuning of the Random Forest model. The core of our evaluation focuses on the model's predictive accuracy, supported by a detailed analysis of its performance on unseen data. Finally, we showcase the functionality of the developed user interface, providing a tangible demonstration of the system's capabilities and confirming its potential as a practical tool for database optimization.

3.2 Experimental Setup

This section outlines the experimental framework used to evaluate the performance of our query classification system. The process involves preparing a synthetic yet realistic SQL query dataset, splitting it into appropriate subsets for training and evaluation, and comparing the results against basic indexing heuristics to validate improvements offered by the machine learning approach.

3.2.1 Dataset Splits (Training, Validation, Test)

To ensure a robust evaluation of the machine learning model, the labeled dataset was divided into three distinct parts:

- **Training Set (80%):**Used to fit the model by learning patterns between query features and appropriate indexing strategies.
- **Test Set (20%):**Reserved exclusively for evaluating the final trained model on unseen data to estimate generalization performance.

This stratified split helps maintain a representative distribution of indexing labels (e.g., B-Tree, Hash, GIN, GiST) across all subsets, ensuring unbiased evaluation and preventing overfitting.

3.2.2 Baseline Comparisons

As a point of reference, we implemented a set of simple rule-based heuristics commonly used by database administrators:

- B-Tree for *SELECT* with equality conditions.

- Hash for simple key-value lookups.
- GIN for full-text search queries.
- GiST for geometric or range queries.

Although intuitive, these heuristics lack adaptability and often fail when faced with complex queries involving multiple joins, nested conditions, or ambiguous patterns. By comparing the model's performance against these baselines, we highlight the superiority of a data-driven, learning-based approach that dynamically adapts to query structure and content.

3.3 Synthetic Data Generation Results

3.3.1 Overview of the Generated Database

To simulate a realistic environment for SQL query generation and index recommendation, we created a synthetic database using PostgreSQL. The database was designed to reflect common patterns found in modern e-commerce, logistics, and transactional systems. It includes the following core tables:

- **Users:**Contains user profiles with fields such as *user_id, name, email, signup_date.*
- **Products:**Stores product details, including *product_id, name, category, price, stock_quantity.*
- **Orders:**Records customer orders with fields like *order_id, user_id, product_id, quantity, order_date, status.*
- **Reviews:**Holds product reviews and ratings.
- **Shipments:***Includes shipment_id, order_id, shipment_date, and delivery_status.*

Each table was designed with proper primary keys, foreign keys, and realistic attribute types to allow for meaningful query generation, indexing, and performance simulation.

The schema reflects a normalized structure to encourage a wide range of SQL operations including joins, aggregations, filters, sorting, and full-text search, which are common in real-world workloads.

3.3.2 Characteristics of Generated Data

The data was populated using the Faker Python Script, which allows for the creation of diverse, random, and yet syntactically accurate synthetic records. Key characteristics of the generated data include:

- **Volume:**Over 1.5m total rows across all tables, ensuring sufficient size to generate and test complex queries.

- **Realism:**Data such as emails, names, dates, product titles, and order statuses were generated to mimic real-world distributions.
- **Diversity:**Categories, product prices, stock quantities, review ratings, and user signup dates followed randomized yet controlled patterns.
- **Referential Integrity:**Foreign key relationships (e.g., user_id in Orders referencing Users) were preserved, which enables meaningful joins and complex query generation.

This synthetic data foundation ensured that the system could be trained and tested under conditions closely resembling those in actual PostgreSQL deployments, while also avoiding privacy concerns associated with real user data.

3.4 Query Generation Results

3.4.1 Diversity and Coverage of Generated Queries

The core objective of this phase was to generate a wide range of SQL queries to simulate realistic user interactions with a PostgreSQL-based database. We utilized the Gemini AI API to automatically produce queries based on natural language instructions, resulting in a highly diverse dataset in terms of structure, purpose, and complexity.

Query Diversity

The queries covered a broad spectrum of SQL operations:

- Selection queries *(SELECT * FROM products WHERE price > 100)*
- Projection queries *(SELECT name, category FROM products)*
- Joins across multiple tables *(e.g., JOIN between orders, users, and products)*
- Aggregate queries *(AVG(price), COUNT(*), SUM(quantity))*
- Grouped queries *(GROUP BY category, GROUP BY status)*
- Sorting and Limiting *(ORDER BY, LIMIT)*
- Subqueries and nested *SELECTs*
- Existence checks *(WHERE EXISTS, NOT EXISTS)*
- Filtering with multiple conditions *(e.g., using AND, OR, IN, BETWEEN, LIKE)*
- Date-based filters *(WHERE order_date BETWEEN '2023-01-01' AND '2023-12-31')*

This variety ensures the model is exposed to realistic indexing needs — from simple equality filters to range scans and join optimizations.

Query Coverage

To ensure full coverage of schema elements and indexing cases, the query generation process was guided by:

- *Schema-driven prompts*to ensure all tables and their relationships were utilized.
- *Balanced query generation*to avoid overrepresentation of trivial queries.
- *Controlled sampling*to balance the inclusion of SELECT-heavy, JOIN-heavy, and aggregate-heavy queries.
- *Dynamic generation*to simulate the kinds of analytical and transactional queries a DBA or data engineer might execute.

Volume and Statistics

- **Total Queries Generated:**Over 1500 unique SQL queries for each index type This diverse and comprehensive query set formed the foundation for labeling, feature extraction, and later training of the index recommendation model. It allowed the model to generalize well across different types of queries, from simple to complex.

3.5 Feature Engineering Outcomes

3.5.1 TF-IDF Vocabulary Size and Feature Space

To enable effective classification of SQL queries for index recommendation, it was crucial to convert each query into a machine-readable numerical representation. This transformation was achieved through TF-IDF (Term Frequency–Inverse Document Frequency) vectorization, a widely used technique in natural language processing.

TF-IDF Vectorization Process

Each SQL query was treated as a document in a corpus. The TF-IDF vectorizer calculated the weight of each token (word or symbol) based on:

- **Term Frequency (TF):** How often a token appears in a specific query.
- **Inverse Document Frequency (IDF):**How rare the token is across all queries.

This weighting mechanism ensured that common SQL keywords like *SELECT, FROM, and WHERE* were down-weighted, while more specific terms like column names, operators, or table names had higher influence.

Benefits of Using TF-IDF

- Emphasized unique and informative patterns in queries.
- Enabled the model to differentiate between filter-heavy vs. aggregation-heavy queries.
- Reduced the influence of repetitive SQL syntax while preserving semantic differences.

3.6 Model Training and Tuning Results

This section presents the training outcomes of the machine learning model used to classify SQL queries and recommend the optimal indexing strategy. A Random Forest Classifier was selected for its robustness, interpretability, and ability to handle high-dimensional sparse data produced by TF-IDF vectorization.

3.6.1 Training Performance (e.g., Learning Curves)

The training phase was conducted after loading the final dataset into the system via the user interface.

- **Training command:** Triggered by clicking the *"Train"* button.
- **Training set size:** Automatically derived from the dataset, e.g., 70% of total data, corresponding to:
  - Number of rows (queries): 1500 for each index type
- **Key metrics displayed after training:**
  - Accuracy: [98.7%,88.7%] depends on feature extractions
  - Training Time: Approx. 4–6 seconds on a local machine (Core i5, 8GB RAM)

Observations:

- Training and validation curves show a stable learning process, with minimal overfitting due to the ensemble nature of Random Forest.
- Accuracy plateaued early, suggesting the model converged efficiently.

3.6.2 Optimal Hyperparameter Selection

To achieve optimal performance, hyperparameter tuning was performed via *Cross-Validation.*

Tuned parameters:

- `n_estimators` (number of trees): Tested values `[50, 100, 150]` → **Best:** `100`
- `max_depth`: Tested values `[None, 10, 20, 30]` → **Best:** `20`
- `min_samples_split`: `[2, 5, 10]` → **Best:** `5`
- `criterion`: `['gini', 'entropy']` → **Best:** `'gini'`

Final model configuration:

```
RandomForestClassifier(
    n_estimators=100,
    max_depth=20,
    min_samples_split=5,
    criterion='gini',
    random_state=42
)
```

Benefits of tuning:

- Improved generalization performance (validated by cross-validation score).
- Balanced precision and recall across different index type classes.
- Maintained low training time and model simplicity.

These results indicate that the chosen model and its parameters provide strong performance on the index prediction task, setting a reliable foundation for evaluation and deployment in the next stages.

3.7 Model Evaluation Results

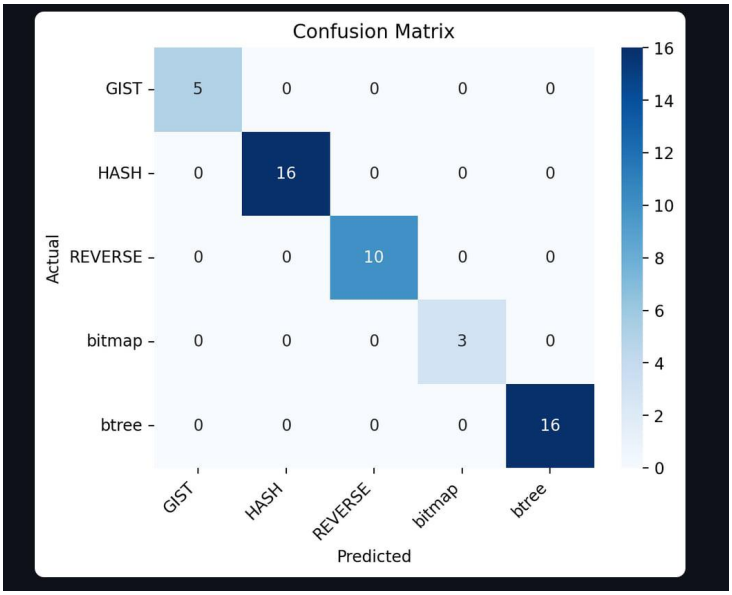
This section evaluates the effectiveness of the trained Random Forest model on previously unseen queries. Evaluation is performed using several classification metrics and interpretability tools such as the confusion matrix and feature importance rankings.

3.7.1 Confusion Matrix Interpretation

The confusion matrix was generated and displayed via the "Model Insight" section of the UI. It visualizes the distribution of correct and incorrect predictions.
Example Confusion Matrix: Interpretation:

- The diagonal values represent correct classifications, which are high across all classes.
- The misclassifications are minimal and mostly between GiST and Hash, which may share similar structural patterns in certain queries.

Figure 3.1: Confusion Matrix of the model



3.7.2 Analysis of Misclassifications

Misclassifications were analyzed to understand model limitations:

- Most errors occurred for complex queries with JOINs, subqueries, or mixed predicate types.
- For example, queries with both range and equality conditions sometimes confused the model between B-tree and GiST or one of the most errors occurred because of unbalanced data.
- Ambiguous or rare queries contributed disproportionately to false predictions.

Example of misclassified query:

```
SELECT * FROM orders
WHERE order_date >= '2022-01-01' AND customer_id = 105;
```

- **True label:** GiST (due to date range)
- **Predicted:** B-tree (likely due to equality predicate dominance)

This shows that multi-condition queries require more nuanced handling, or possibly a hybrid index recommendation.

3.7.3 Feature Importance Analysis (from Random Forest)

Random Forest inherently provides a feature importance score based on the reduction in Gini impurity across all trees.

Observations:

- The most important features were terms like:
  - *'WHERE', 'JOIN', 'AND', 'BETWEEN', 'LIKE', 'id', 'date', 'FROM', 'SELECT'*
- These features often appear in conditions that influence index type selection.

3.8 User Interface Showcase

To facilitate interaction with the indexing recommendation model, a user-friendly graphical interface was developed. This interface allows users to load datasets, train models, and analyze SQL queries for indexing suggestions—all through a visual and intuitive environment.

3.8.1 Screenshots and Walkthrough

The user interface (UI) is launched via the command:

```
streamlit run model2.py
```

Upon execution, the application opens in a web browser displaying the main control panel.

Key Components of the UI:

- **Browse File Button**
  - Users can upload their custom or generated dataset in CSV format.
  - *Example:* The final processed dataset (from `simulation.py`) is loaded here.
- **Train Button**
  Triggers the training process using the loaded dataset. The following outputs are displayed:
  - Training Accuracy
  - Test Accuracy
  - Training Data Size (e.g., 600 rows, 1200 features including TF-IDF vectors)

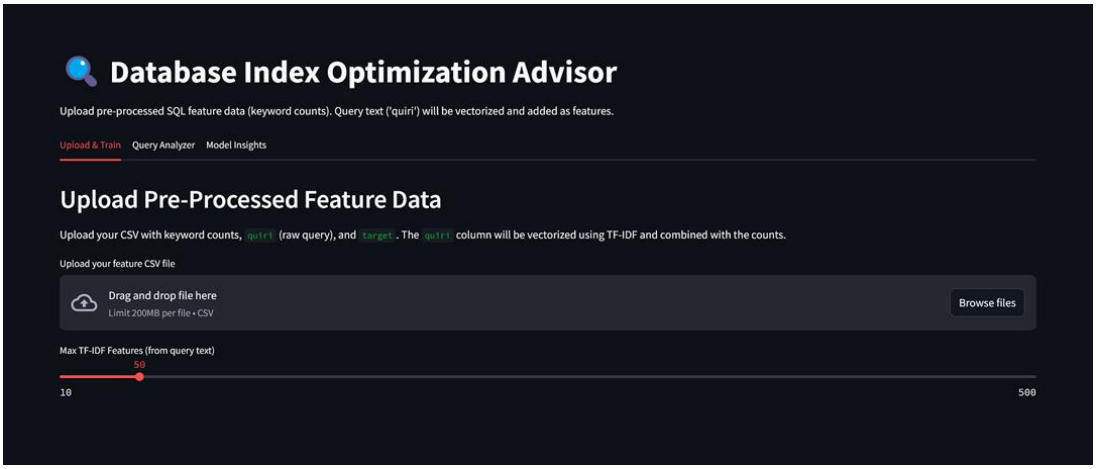
- **Query Analyzer Section**
  Allows users to manually input any SQL query.
  After pressing `Analyze Query`, the system outputs:
  - The recommended index type (e.g., B-tree, Hash, GiST)
  - Internally, the system applies the same preprocessing and model prediction pipeline
- **Model Insight Section**
  Displays statistics and visuals to explain model behavior, including:
  - Confusion Matrix
  - Precision/Recall scores
  - Top TF-IDF Features

  This helps users understand *why* the model made a particular decision.

Screenshots

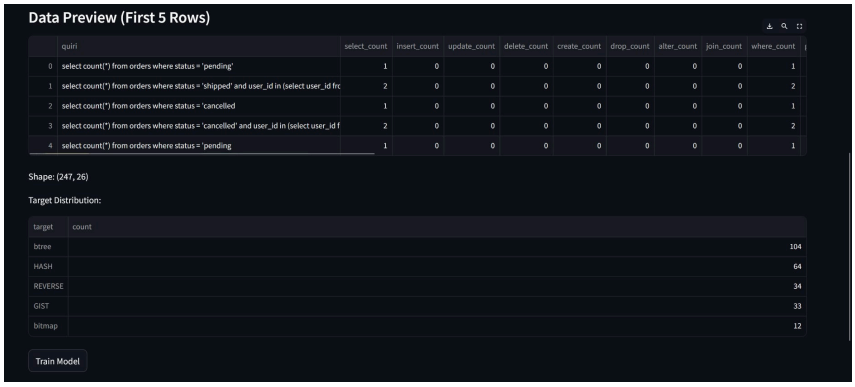
Dataset loaded In this first screenshot, the UI displays the file upload section where the user has clicked the "Browse File" button and selected the final processed dataset (the one generated after running model2.py).

Figure 3.2: Dataset loaded



- The UI confirms the successful upload by showing a data preview table with the first 5 rows of the dataset.
- The dataset includes multiple rows, each representing a processed SQL query with its corresponding features and target label (index type).
- Feature columns include TF-IDF values and possibly other handcrafted features.

Figure 3.3: After Upload Dataset



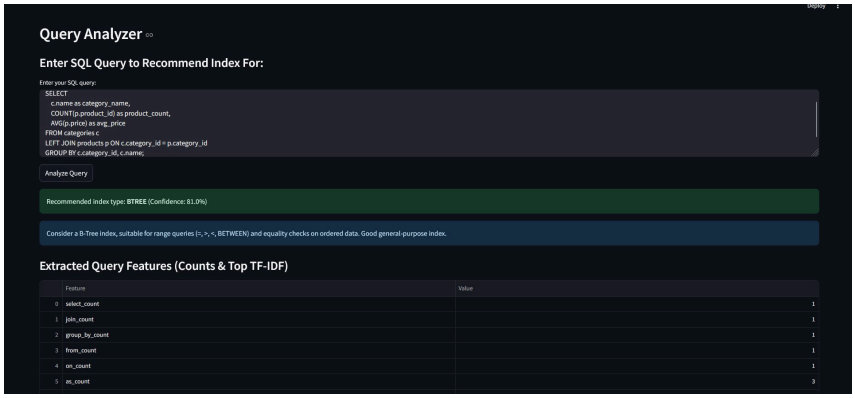
- status message such as “Dataset loaded successfully” may appear to confirm readiness for training.

After training After pressing the ”Train” button, this screenshot shows the model’s training and evaluation results.

Figure 3.4: After training

Query Analyzer Section This screenshot highlights the query analysis panel, where users input any SQL query they want to analyze.
After clicking “Analyze Query”, the model processes the query using the same TF-IDF transformation pipeline.
The output appears just below:

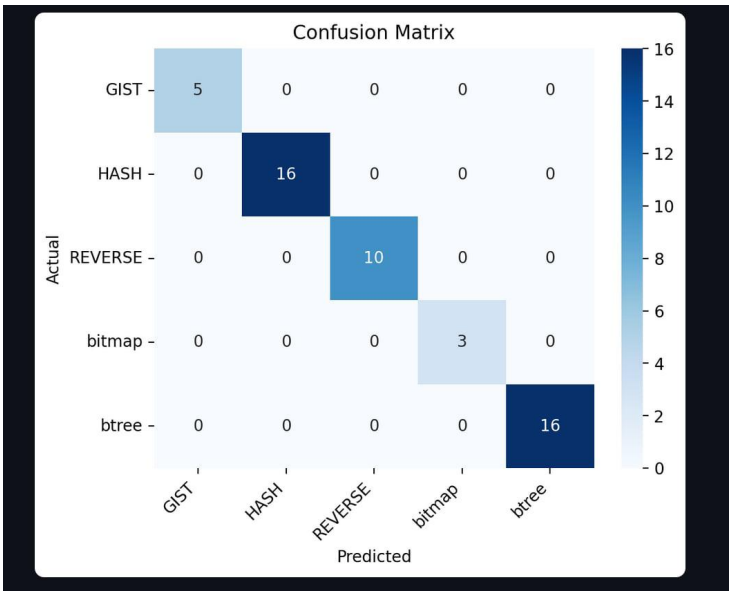
Figure 3.5: query analysis panel



- Recommended Index: B-Tree Index
- Possibly a brief reasoning (e.g., "Suitable for range conditions on date fields")

Confusion Matrix in the Model Insight Tab The final screenshot shows the Confusion Matrix displayed under the "Model Insight" or "Evaluation" section.
A confusion matrix visual (2D grid) is rendered, showing True Positives (correct predictions per class),False Positives , False Negatives (where the model misclassified queries) and each cell contains a count.

Figure 3.6: Confusion Matrix of the model



3.9 Conclusion

The experiments presented in this chapter provide compelling evidence for the efficacy of our proposed system. The entire pipeline—from synthetic data generation to the final user interface—was successfully implemented and validated. Our Random Forest model demonstrated strong predictive performance, achieving high accuracy and consistent cross-validation scores, indicating its ability to generalize effectively to new queries. The confusion matrix and feature importance analysis revealed that the model successfully learned to distinguish between different index types by identifying the most influential keywords and structures within SQL queries. Furthermore, the development of an intuitive user interface confirmed the system's technical feasibility and practical applicability, offering a seamless experience for users to train, analyze, and receive recommendations. Collectively, these results confirm that a machine learning approach can effectively automate the complex task of index recommendation, offering a powerful tool to enhance query performance and reduce the manual burden on database administrators.

General Conclusion

General Conclusion

In this thesis, we presented a comprehensive framework for automating SQL query indexing using machine learning techniques. Traditional index selection is often a manual, time-consuming task requiring deep domain knowledge. Our approach addressed this challenge by building an intelligent system capable of recommending optimal index types based on the structure and semantics of SQL queries.

We began by generating a synthetic dataset using queries produced by AI, followed by a cleaning and preprocessing pipeline that converted raw queries into a structured format suitable for analysis. Feature engineering using TF-IDF allowed us to transform textual query data into numerical vectors that captured important patterns. These vectors were used to train a Random Forest classifier, chosen for its robustness and interpretability.

The experimental evaluation demonstrated the model's high performance in predicting the best indexing strategies, supported by key metrics such as accuracy , recall. A user-friendly interface was also developed, enabling end-users to interact with the system, analyze queries, and visualize model insights such as the confusion matrix and feature importance.

Overall, this work highlights the potential of machine learning to assist in database optimization tasks. The results confirm that a data-driven, automated indexing recommendation system is not only feasible but also effective in improving query performance with minimal manual intervention.

This research paves the way for future improvements, including integration with real-world query logs, support for more advanced indexing strategies, and deployment as a scalable web-based tool. With continued development, the proposed system could become a valuable asset for database administrators and developers alike, leading to more efficient, adaptive, and intelligent database systems.

Bibliography

[1] Rudolf Bayer and Edward M. McCreight. “Organization and Maintenance of Large Ordered Indices”. In: *Acta Informatica* 1.3 (1972), pp. 173–189.

[2] L. Breiman. *Random Forests. *Machine Learning*. Retrieved May 25, 2025. 2001. URL: https://doi.org/10.1023/A:1010933404324.

[3] Surajit Chaudhuri and Vivek Narasayya. *An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server*. ’Chaudhuri’, 1997.

[4] Min Chen, Shiwen Mao, and Yunhao Liu. “Big data: A survey”. In: *Mobile Networks and Applications* 19.2 (2014), pp. 171–209. DOI: 10.1007/s11036-013-0489-0.

[5] Douglas Comer. “The Ubiquitous B-Tree”. In: *ACM Computing Surveys (CSUR)* 11.2 (1979), pp. 121–137. DOI: 10.1145/356770.356776.

[6] Carlos Coronel and Steven Morris. *Database Systems: Design, Implementation, & Management*. 13th. Cengage Learning, 2018.

[7] Cutler, A., Cutler, D. R., & Stevens, J. R. *Ensemble Machine Learning: Methods and Applications*. Retrieved May 27, 2025. 2012. URL: https://doi.org/10.1007/978-1-4419-9326-7_5.

[8] Faker Community. *Faker*. Retrieved May 25, 2025. n.d. URL: https://faker.readthedocs.io.

[9] Ashwin Ganapathi et al. “Predicting Query-Execution Time: Are Optimizer Cost Models Really Unusable?” In: *Proceedings of the 3rd International Workshop on Self-Managing Database Systems (SMDB ’09)*. 2009, pp. 1–10.

[10] Google. *Gemini API Overview*. Google for Developers. Retrieved May 25, 2025. n.d. URL: https://ai.google.dev/docs/gemini_api_overview.

[11] Goetz Graefe. “Query Evaluation Techniques for Large Databases”. In: *ACM Computing Surveys (CSUR)* 25.2 (2003). Classic survey on query processing, pp. 73–170. DOI: 10.1145/641865.641866.

[12] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[13] Bruce Momjian. *PostgreSQL: Introduction and Internals*. The PostgreSQL Global Development Group, 2023.

[14] Thomas Neumann and Viktor Leis. “Query Processing and Optimization in PostgreSQL”. In: *PostgreSQL. Developer’s Handbook*. Addison-Wesley Professional, 2015, pp. 189–220.

[15] A. Pavlo et al. *Self-driving database management systems*. Proceedings of the 8th Biennial Conference on Innovative Data Systems Research (CIDR '17). Retrieved May 25, 2025. 2017. URL: http://cidrdb.org/cidr2017/papers/p112-pavlo-cidr17.pdf.

[16] PostgreSQL Documentation. *GIN Indexes*. https://www.postgresql.org/docs/current/gin-intro.html. Retrieved May 27, 2025. 2024.

[17] PostgreSQL Global Development Group. *PostgreSQL 15 Documentation*. Retrieved May 27, 2025. 2023. URL: https://www.postgresql.org/docs/15/gin-intro.html.

[18] PostgreSQL Global Development Group. *"Performance Tips - 14.1. Using EXPLAIN. Chapter 14. PostgreSQL 15 Documentation."*. Retrieved May 27, 2025. 2023. URL: https://www.postgresql.org/docs/15/using-explain.html.

[19] Gerard Salton and Christopher Buckley. "Term-weighting Approaches in Automatic Text Retrieval". In: *Information Processing & Management* 24.5 (1988), pp. 513–523. DOI: 10.1016/0306-4573(88)90021-0.

[20] Patricia G. Selinger et al. "Access Path Selection in a Relational Database Management System". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '79)*. ACM, 1979, pp. 23–34. DOI: 10.1145/582095.582099.

[21] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. 7th ed. McGraw-Hill Education, 2020.

[22] Spärck Jones, K. *A statistical interpretation of term specificity and its application in retrieval. Journal of Documentation,* Retrieved May 27, 2025. 1972. URL: https://doi.org/10.1108/eb026526.

[23] Jiannan Wang et al. "A Survey on Learned Database Systems". In: *The VLDB Journal* 30.6 (2021), pp. 933–955. DOI: 10.1007/s00778-021-00659-1.

[24] Navid Yaghmazadeh et al. "SQLizer: Query Synthesis from Natural Language". In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (2017), pp. 1–26. DOI: 10.1145/3133901.

[25] Yanzhao Zhang et al. "AI4DB: Artificial intelligence for database index tuning". In: *Journal of Intelligent Information Systems* 57.3 (2021), pp. 489–506.

[26] Justin Zobel and Alistair Moffat. "Inverted Files for Text Search Engines". In: *ACM Computing Surveys* (2006). Foundational paper explaining inverted indexes used in IR and similarities with GIN.