FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Towards Automated Relational Database Partitioning

Vasco David Antunes Pereira Gomes



Mestrado em Engenharia Informática e Computação

Supervisor: Tiago Boldt Pereira de Sousa

External Supervisor: Paul deGrandis

July 11, 2024

© Vasco David Antunes Pereira Gomes, 2024

Towards Automated Relational Database Partitioning

Vasco David Antunes Pereira Gomes

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Carlos Manuel Milheiro de Oliveira Pinto Soares
Referee: Marco António Morais Veloso
Referee: Tiago Boldt Pereira de Sousa

July 11, 2024

Abstract

The growing reliance on the Internet has contributed to a substantial increase in the production and exchange of data within online applications. As the global adoption of these applications grows, so too must their ability to scale. Therefore, architectures must efficiently manage larger volumes of data without compromising overall performance. Optimizing database performance becomes essential to ensure a smooth user experience and efficient resource utilization, eventually leading to higher revenue and lower infrastructure costs.

Currently, Relational Database Management Systems (RDBMSs) stand as the predominant database management system, offering diverse strategies for enhancing performance, ranging from indexes to partitions, coupled with fine-tuning database configurations. Traditionally, the responsibility of optimizing these designs fell upon Database Administrators (DBAs). However, the inherent complexity of large-scale database schemas has driven the emergence of tools and services to automate and enhance these design decisions.

Considerable work has been done on automated strategies for optimizing the use of indexes, partitions, and database configurations. Reinforcement Learning (RL) has been successfully used to automated index and configuration strategies. However, despite its success, a noticeable gap exists in the realm of RL strategies for automated database partitioning. Furthermore, existing literature on RL in database partitioning only focuses on analytical databases, disregarding transactional ones.

With this research, we set ourselves to demonstrate that:

> *Reinforcement Learning-based algorithms can optimize the performance of transactional large-scale database systems through the recommendation of database partitions, reducing queries runtime and systems' tail latency.*

To validate this hypothesis, we first surveyed the state of the art of automated database optimization. Then, we propose an architecture based on RL that can recommend database partitions. Finally, we validate this refined approach using well-established benchmarks, such as the Transaction Processing Performance Council (TPC)-E dataset, alongside an industry scenario. By subjecting our enhanced method to these industry-standard benchmarks, we intend to evaluate its performance and the effectiveness of its recommendations rigorously.

The empirical validation demonstrated a decrease in the runtime in both scenarios. However, specific queries experienced significant performance degradation in the industry scenario, increasing the database's tail latency. These results suggest the potential to use RL approaches in automated database partitioning and that this work can be established as the groundwork for further exploration.

Keywords: Database Tuning, Data Partitioning, Workload, Autonomous Database Management

ACM Classification:

• Information systems → Data management systems → Database administration → Autonomous database administration

Resumo

O crescente uso da Internet tem contribuído para um aumento substancial na produção e troca de dados em aplicações online. À medida que a adoção global dessas aplicações cresce, também deve crescer a sua capacidade de escalar. Portanto, as arquiteturas devem gerir eficientemente volumes maiores de dados sem comprometer o desempenho geral. A otimização do desempenho das bases de dados torna-se essencial para garantir uma experiência de usuário fluida e uma utilização eficiente dos recursos, levando eventualmente a maiores receitas e menores custos de infraestrutura.

Atualmente, os Sistemas de Gestão de Bases de Dados Relacionais permanecem como os sistemas predominantes de gestão de bases de dados, oferecendo diversas estratégias para aprimorar o seu desempenho, desde índices, partições, e a afinação das configurações da base de dados. Tradicionalmente, a responsabilidade de otimizar esses projetos recaía sobre os Administradores de base de dados. No entanto, a complexidade inerente aos esquemas de base de dados em larga escala impulsionou o surgimento de ferramentas e serviços para automatizar e aprimorar essas decisões.

Considerável trabalho já foi realizado em estratégias automatizadas para otimizar o uso de índices, partições e configurações de bases de dados. Aprendizagem por Reforço tem sido usada com sucesso para automatizar estratégias de índices e configurações. No entanto, apesar de seu sucesso, existe uma lacuna perceptível no domínio das estratégias de Aprendizagem por Reforço para particionamento automatizado de bases de dados. Além disso, a literatura existente sobre Aprendizagem por Reforço no particionamento de bases de dados apenas se concentra em bases de dados analíticas, ignorando as transacionais.

Com esta dissertação, nos propomos a demonstrar que:

> *Algoritmos baseados em Aprendizado por Reforço podem otimizar o desempenho de sistemas de bases de dados transacionais em larga escala por meio da recomendação de partições de bases de dados, reduzindo o tempo de execução das consultas às bases de dados e a latência de cauda do sistema.*

Para validar essa hipótese, primeiro realizamos uma revisão do estado da arte da otimização automatizada de bases de dados. Em seguida, propomos uma arquitetura baseada em Aprendizagem por Reforço que pode recomendar partições de bases de dados. Finalmente, validamos essa abordagem refinada usando benchmarks bem estabelecidos, como o conjunto de dados TPC-E, e com um cenário da indústria. Ao submeter o nosso método a esses benchmarks padrão da indústria, pretendemos avaliar rigorosamente seu desempenho e a eficácia de suas recomendações.

A validação empírica demonstrou uma diminuição do tempo de execução em ambos os cenários. No entanto, as consultas específicas sofreram uma degradação significativa de desempenho no cenário da indústria, aumentando a latência de cauda da base de dados. Estes resultados sugerem o potencial para utilizar abordagens de Aprendizagem por Reforço no particionamento automatizado de bases de dados e que este trabalho pode ser estabelecido como base para futuras explorações.

Acknowledgements

Firstly, I would like to thank my supervisor, Professor Tiago Boldt de Sousa, for his guidance and honest advice throughout the dissertation. His availability and high standards were essential for the success of this work. Without him, I might still be staring at a blank page.

I would also like to thank my external supervisor, Paul deGrandis, for sharing his knowledge and experience in the field, which significantly influenced the direction of this work.

I am also thankful to André Cardoso for helping to replicate Kevel's database. His advice was vital during the validation phase of this work.

I am very grateful to my family for their endless support throughout my academic journey. Thank you for trusting my decisions without hesitation and believing in my abilities. Your confidence in me has been both encouraging and slightly terrifying.

I am also extremely grateful to Mariana for always being there since day one. Thank you for pushing me towards social activities and providing unforgettable experiences. You helped me realize there's life beyond the computer screen. Thank you for helping me believe in myself—and for occasionally believing in me more than I did.

Lastly, thank you to my friends who always knew how to find the perfect balance between work and play. Whether it was a spontaneous adventure, a game night, or simply hanging out and doing nothing, you showed me the importance of taking breaks and enjoying the moment. Thank you for being the masters of distraction I never knew I needed.

Vasco Gomes

"E se corre bem?"

Rúben Amorim

Contents

1 Introduction **1**
- 1.1 Context . . . 1
- 1.2 Motivation . . . 2
- 1.3 Problem . . . 2
- 1.4 Research Goals . . . 3
- 1.5 Document Structure . . . 4

2 Background **5**
- 2.1 Databases . . . 5
  - 2.1.1 Relational Database Management System . . . 6
  - 2.1.2 Structured Query Language . . . 7
  - 2.1.3 Performance . . . 9
  - 2.1.4 Query plan . . . 11
  - 2.1.5 Query Optimization . . . 11
  - 2.1.6 Workloads . . . 13
- 2.2 Large-Scale Computing . . . 13
  - 2.2.1 Large-Scale Applications . . . 13
  - 2.2.2 Cloud Computing . . . 14
- 2.3 Reinforcement Learning . . . 15
- 2.4 Summary . . . 16

3 State of the Art **19**
- 3.1 Systematic Literature Review . . . 19
  - 3.1.1 Methodology . . . 19
  - 3.1.2 Literature Research Questions . . . 20
  - 3.1.3 Data Sources . . . 21
  - 3.1.4 Inclusion and Exclusion criteria . . . 21
  - 3.1.5 Search Queries: . . . 21
- 3.2 Database Optimization . . . 22
  - 3.2.1 Indexes . . . 22
  - 3.2.2 Configurations . . . 28
  - 3.2.3 Partitioning . . . 29
- 3.3 Automated Index Management . . . 35
  - 3.3.1 Adaptive indexing . . . 35
  - 3.3.2 Index selection algorithms . . . 36
  - 3.3.3 Summary . . . 41
- 3.4 Automated Configuration Tuning . . . 42
  - 3.4.1 Heuristic . . . 43

3.4.2 Bayesian Optimization . . . 44
3.4.3 Reinforcement Learning . . . 46
3.4.4 Deep Learning . . . 49
3.4.5 Summary . . . 49
3.5 Automated Database Partitioning . . . 51
3.5.1 Deep Reinforcement Learning . . . 51
3.5.2 Schism . . . 52
3.5.3 Database Engine Tuning Advisor . . . 52
3.5.4 SWORD . . . 54
3.5.5 AdaptDB . . . 54
3.5.6 Horticulture . . . 55
3.5.7 DB2 Design Advisor . . . 56
3.5.8 Summary . . . 57
3.6 Performance Evaluation Metrics . . . 57
3.7 Summary . . . 62
3.7.1 LRQ1 . . . 62
3.7.2 LRQ2 . . . 62
3.7.3 LRQ3 . . . 62
3.7.4 LRQ4 . . . 63

4 Problem Statement **65**
4.1 Open Problems . . . 65
4.2 Scope . . . 66
4.3 Hypothesis . . . 66
4.4 Research Questions . . . 68
4.5 Validation and Evaluation . . . 68
4.6 Summary . . . 69

5 Reference Architecture and Implementation **71**
5.1 Overview . . . 71
5.1.1 Preprocessing . . . 73
5.1.2 Training . . . 73
5.1.3 Application . . . 74
5.2 Model . . . 75
5.2.1 State Representation . . . 75
5.2.2 Workload Modeling and Query Representation . . . 76
5.2.3 Actions . . . 77
5.2.4 Rewards . . . 79
5.3 Reference Implementation . . . 80
5.3.1 Database . . . 80
5.3.2 Model . . . 81
5.4 Summary . . . 83

6 Verification and Validation **85**
6.1 Methodology . . . 85
6.2 Empirical Validation . . . 86
6.2.1 TPC-E . . . 86
6.2.2 Kevel . . . 88
6.2.3 Training Duration & Effort . . . 90

6.2.4 Threats to Validity 90
6.3 Comparison to the State of the Art 91
6.4 Summary 93

7 Conclusions **95**
7.1 Summary 95
7.2 Research Questions 96
7.3 Hypothesis Revisited 98
7.4 Contributions 98
7.5 Challenges 99
7.6 Future Work 100

References **103**

A Literature review results **113**

B Execution example **115**

List of Figures

2.1 School relational database UML class diagram 7
2.2 Few records of student represented in the table Student 11

3.1 A clustered index on the column Department. 23
3.2 A non-clustered index on the column Salary. 24
3.3 A dense index on the column ID. 24
3.4 A sparse index on the column ID. 25
3.5 A B-tree index on the column Name. 25
3.6 A Hash index on the column ID. 26
3.7 Bitmap indexes on the column gender and income_level. 26
3.8 A R-tree index with a set of rectangles. 27
3.9 An example of an inverted index. 28
3.10 Horizontal Partitioning on the table Student by the attribute Sex. 29
3.11 Vertical Partitioning on the table Student to take advantage of frequently accessed data. 30
3.12 A Shared-Nothing architecture diagram. 33
3.13 A Shared-Disk architecture diagram. 34
3.14 A Shared-Memory architecture diagram. 34

5.1 Reference architecture UML component diagram. 72
5.2 Reference architecture UML activity diagram. 74
5.3 Construction of the state for a simplified example workload. 75
5.4 Workload modelling process. 77
5.5 Action representation and invalid action masking for a simplified example workload. 79

6.1 TPC-E's transactions execution time distribution. 88
6.2 Kevel's transactions execution time distribution. 89

List of Tables

3.1 Workload benchmarks used to evaluate performance in automated index management tools. . . . 58
3.2 Workload benchmarks used to evaluate performance in automated configuration tools. . . . 59
3.3 Workload benchmarks used to evaluate performance in automated partitioning tools. 59
3.4 Metrics used to evaluate performance in automated index management tools. . . 60
3.5 Metrics used to evaluate performance in automated configuration tools. 61
3.6 Metrics used to evaluate performance in automated partitioning tools. 61

5.1 Hyperparameters used in our model. 82

6.1 TPC-E specification values used in the empirical validation. 87
6.2 TPC-E's empirical validation 87
6.3 Kevel's empirical validation 89
6.4 Training duration and problem complexity metrics for the different databases. . . 90
6.5 Comparison of different automated database partitioning approaches. 93

A.1 Results of the literature review. 114

List of Listings

2.1 Data Definition Language statements on the table Student 8
2.2 Data Modification Language statements on the table Student 8
2.3 Data Query Language statements on the table Student 9
2.4 Data Control Language statements on the table Student 9
2.5 Query to retrieve all male students. 11
3.1 Search query used in the ACM Digital Library. 21
3.2 Search query used in IEEE Xplore. 22
3.3 Example of *postgresql.conf*, the PostgreSQL configuration file. 28
3.4 SQL command to change the global parameter *shared_buffers* to 64MB. 29
3.5 Creating the table Student on PostgreSQL with range partitioning on the attribute AverageGrade. 31
3.6 Creating the table Student on PostgreSQL with hash partitioning on the attribute Name. 32
3.7 Creating the table Student on PostgreSQL with list partitioning on the attribute Sex. 32
5.1 Example of a EXPLAIN command execution in PostgreSQL. 81
6.1 Commands to build the TPC-E Tools for PostgreSQL. 86
6.2 Commands to build the database and measure the transactions. 86
B.1 Configuration file example. 116
B.2 Kevel's queries file sample. 117

Abbreviations

1NF First Normal Form. 7

2NF Second Normal Form. 7

3NF Third Normal Form. 7

4NF Fourth Normal Form. 7

5NF Fifth Normal Form. 7

Amazon EC2 Amazon Elastic Compute Cloud. 85, 86, 91, 93, 99, 100

ANN Artificial Neural Network. 40, 49, 51, 73, 74, 76, 82, 92

API Application Programming Interface. 37, 42, 88

AWS Amazon Web Services. 91

BCNF Boyce-Codd Normal Form. 7

BO Bayesian Optimization. 44, 50, 97

BOO Bag of Operators. 40, 76, 77

CPU Central Processing Unit. 33, 34, 41, 56, 60, 61, 80, 96

DBA Database Administrator. i, 1, 2, 10, 22, 30, 32, 35, 40, 42, 43, 49, 51, 52, 63, 66, 90, 95, 97, 100

DBMS Database Management System. 2, 6, 11, 12, 33, 35, 45, 55, 59, 61, 62, 66

DCL Data Control Language. 8, 16

DDL Data Definition Language. 8, 16

DL Deep Learning. 49, 51, 63, 97

DML Data Manipulation Language. 8, 16

DQL Data Query Language. 8, 16

DRL Deep Reinforcement Learning. 47, 48, 50, 51, 57, 97

DTA Database Engine Tuning Advisor. 36, 37, 42, 52, 57, 59, 60, 63

FK Foreign Key. 6, 88

I/O Input/Output. 10, 36, 41, 56, 61, 80

IDS Integrated Data Store. 5

IT Information Technology. 45

LNS Large-Neighbourhood Search. 55

LP Linear Programming. 39, 42, 97

LSI Latent Semantic Indexing. 77, 82

MCTS Monte Carlo tree search. 41

MESA Memo-Based Search Algorithm. 53

ML Machine Learning. 39, 49, 51, 63, 66, 73, 97

MLP Multi-Layer Perceptron. 82

MPP Massively Parallel Processors. 53

OLAP Online Analytical Processing. 13, 16, 51, 52, 54, 55, 57–59, 61, 62

OLTP Online Transaction Processing. 13, 16, 49, 51, 52, 54, 55, 57–59, 61, 62, 86, 92

PK Primary Key. 6, 22–24, 88

PPO Proximal Policy Optimization. 81–83

RDBMS Relational Database Management System. i, 1, 6, 8–11, 14, 16, 19, 20, 22, 23, 30, 41, 49, 57, 62, 63, 65, 67, 68, 76, 95–97

RL Reinforcement Learning. i, 2, 3, 5, 15, 17, 40–42, 46, 48–51, 63, 66–69, 71, 73, 75–77, 80–83, 90–92, 95–100

SDV Synthetic Data Vault. 88

SQL Structured Query Language. 7, 8, 11, 16, 28, 36, 41, 46, 50, 56, 76, 80, 81

TPC Transaction Processing Performance Council. i, 52, 54, 55, 57, 58, 62, 69, 85, 86, 88, 90, 92, 93, 96, 98, 100

UML Unified Modeling Language. 6

Chapter 1

Introduction

Contents

1.1 Context 1
1.2 Motivation 2
1.3 Problem 2
1.4 Research Goals 3
1.5 Document Structure 4

This chapter outlines the scope and fundamental essence of this dissertation. Firstly, Section 1.1 provides the context of this work. Then, Section 1.2 describes the motivation of the proposed work. Section 1.3 summarizes the central problem under investigation, while Section 1.4 delineates the primary goals of this dissertation. Finally, Section 1.5 describes how this document is structured.

1.1 Context

Large-scale applications are software systems that can manage extensive computational loads, process vast amounts of data, and serve many concurrent users and requests. These applications are built to scale efficiently by adding more machines or increasing the resources within existing machines [107].

Cloud Computing, as an on-demand computing model, empowers companies to deploy their applications at a large scale with minimal effort [80].

A RDBMS is a software system that enables clients to define, create, maintain, and control access to a relational database [22]. Unless stated otherwise, all examples and implementations will be tested against PostgreSQL [56].

One of the DBAs' primary jobs is optimizing the database according to the system requirements. In large-scale transactional environments, ensuring a fast response is critical to improve the user experience. Moreover, the system must maintain the performance level in periods of high demand. Lastly, there are often multiple database instances where cloud vendors offer baseline

configurations in cloud environments. The DBA's responsible for fine-tuning the databases to meet the system's needs.

1.2 Motivation

Currently, 67% of the world's population uses the Internet on a daily basis [112]. The growing reliance on the Internet has contributed to a substantial increase in the generation and exchange of data within online applications and systems. This surge underscores the imperative for these applications to possess scalable architectures capable of efficiently managing vast data without compromising overall performance.

Database Management Systems (DBMSs) are *"a software system that enables users to define, create, maintain, and control access to the database"* [22]. Their relevance is growing for companies [104], as they handle user data, transactions, and real-time analytics within applications. Consequently, these systems often become a single point of failure, and any issues with the DBMS can cause the entire application to suffer from performance degradation or complete interruptions.

In this context, optimizing database performance becomes essential to ensure a smooth user experience and efficient resource utilization, eventually leading to higher revenue and lower infrastructure costs. Transactional databases are pivotal in handling real-time, mission-critical operations, such as financial transactions or online purchases. Sub-optimal database performance can result in latency issues, transaction failures, and a poor user experience.

This optimization process is challenging for DBAs, particularly in cloud environments where thousands of distinct DBMSs instances may coexist. Manually optimizing databases is particularly challenging in such large-scale applications because of the complexity of the database schema, which usually has a lot of tables and relationships. The volume data also makes it harder to identify and address performance bottlenecks manually. Automated strategies for optimizing databases have been a research subject in recent decades, focusing on tuning indexes, partitions, and database configurations.

SWIRL [70] is a RL approach to automated index tuning that exchanges an intensive training effort with rapid tuning runtimes. Empirical comparisons with other state-of-the-art strategies indicate that SWIRL produces competitive index configurations with the best strategies while achieving significantly faster tuning runtimes. The authors propose extending this approach to other physical database designs, such as partitioning or automatic compression.

Motivated by SWIRL and recognizing that the only RL-based approach for automated database partitioning focuses only on analytical databases [60], there is a compelling need to evaluate RL strategies in partitioning tuning specifically tailored for transactional databases.

1.3 Problem

Even though automated database optimization has been a research topic for the last decades, there is a lack of production-ready automated tools to optimize databases through partitioning. Further-

more, despite the recent research in using RL-based approaches to optimize indexes or database configurations, there is a lack of strategies that utilize RL in automated database partitioning.

Hilprecht *et al.* work [60] is the only identified study that uses RL to build a partitioning advisor. However, this approach lacks the adaptability to new workloads or database schemas. Moreover, it is tailored to optimize analytical databases with long-running and complex queries. Analytical databases use partitioning strategies to optimize data distribution to use parallelism for the long and complex analytical queries. On the other hand, transactional databases focus on improving concurrency and ensuring that frequently accessed data has efficient access. Given the different requirements for these types of databases, there is a gap in utilizing RL for transactional databases.

With this in mind, this dissertation aims to evaluate whether the application of RL, which is amply used in automated index and database configuration tuning, can also be effectively used in an automated partitioning tool tailored for transactional large-scale databases.

1.4 Research Goals

This work seeks to research the application of RL in automated partitioning for transactional databases. In pursuit of this objective, it aims to:

Study the requirements of transactional databases in large-scale applications:

Transactional databases of large-scale applications pose specific demands and challenges. We aim to understand the characteristics and performance criteria that transactional databases must meet when operating at scale. Studying these requirements is crucial in building an optimization tool in this context.

Investigate the existing automated strategies to optimize relational databases:

To build an automated database partitioning tool, it is essential to survey the existing automated approaches that optimize relational databases to have a better understanding of their design choices and their respective challenges and trade-offs.

Design a reference architecture for an automated database partitioning tool:

Based on the review of automated optimization approaches, we intend to design a reference architecture for an automated database partitioning tool tailored for transactional workloads. Moreover, we aim to ensure performance in a large-scale environment and the ability to adapt to unknown queries.

Evaluate the implementation of the architecture:

Following the designed architecture, the subsequent step involves implementing the automated database partitioning tool. A comparative analysis will be conducted against other state-of-the-art approaches, utilizing established benchmarks and metrics to evaluate our implementation.

1.5 Document Structure

This document is composed of four chapters, structured as follows:

- Chapter 1, (p. 1), **Introduction**, introduces the goals of this work and the problem under study, as well as its context and motivation.
- Chapter 2, (p. 5), **Background**, explains the fundamental concepts needed to understand this work.
- Chapter 3, (p. 19), **State of the Art**, describes the literature review process and surveys the state of the art of automated database optimization.
- Chapter 4, (p. 65), **Problem Statement**, identifies open challenges, the scope of this work, and formalizes the problem under study and the validation methodology.
- Chapter 5, (p. 71), **Reference Architecture and Implmenetation**, proposes the reference architecture and implementation for recommending database partitioning strategies.
- Chapter 6, (p. 85), **Verification and Validation**, covers the verification and validation process of our hypothesis.
- Chapter 7, (p. 95), **Conclusions**, summarizes the work developed, its main contributions, and outlines possible future work opportunities.

Chapter 2

Background

Contents

2.1 Databases 5
2.2 Large-Scale Computing 13
2.3 Reinforcement Learning 15
2.4 Summary 16

This chapter provides an overview of the essential principles used in the present work. Firstly, Section 2.1 introduces the primary concept of Databases. Then, Section 2.2 describes important aspects of Large-Scale Computing, including Large-Scale Applications and Cloud Computing. Section 2.3 covers the basic concepts of RL, which help understand the proposed solution. Finally, Section 2.4 summarizes the key concepts of this work's background. Readers familiar with relational databases, the usage of cloud computing in large-scale applications, and RL may skip this chapter.

2.1 Databases

"Since the 1950s the computer has replaced traditional methods of accounting and record keeping by a new industry of data processing. As a primary vehicle of communication over both space and time, it has come to form the core of modern information technology" [79]. The persistent storage of this information necessitated the development of databases, made feasible with the introduction of direct access storage media like magnetic disks, which gained widespread availability in the mid-1960s. The evolution of database technology can be categorized into three distinct eras delineated by their data models or structures: the navigational era, the relational era, and the post-relational era.

The pioneering navigational database, Integrated Data Store (IDS), crafted by Charles Bachman for General Electric, stands as a hallmark in database development [9]. Navigational databases primarily retrieve records through references interlinked with other records. The term gained

prominence through the 1973 Turing Award paper "The Programmer as Navigator" by Bachman [8]. This influential paper emphasized the transformative aspect of disk-based databases, enabling programmers to traverse arbitrary navigational links from one record to another.

In 1970, Codd introduced relational databases in his paper "A Relational Model of Data for Large Shared Data Banks" [21]. Codd insisted that applications search by content instead of following links and that data should be stored in a tabular form.

The post-relational era encompasses various types of NoSQL databases that diverge from the traditional relational database model. NoSQL databases employ diverse data structures such as key–value, wide-column, graph, and document, which contrast with the tabular structure of relational databases. By avoiding rigid schemas, NoSQL databases offer fast scalability, enabling efficient management of extensive, often unstructured datasets [106]. The choice of a specific NoSQL database depends on its alignment with the unique challenges it aims to address.

Database Management System (DBMS) was defined by Connolly and Begg as *"a software system that enables users to define, create, maintain, and control access to the database"* [22]. In addition to allowing the creation of databases that store enormous amounts of data, DBMSs allow multiple clients to query data and modify it while enabling the durability of the database [41]. DBMSs' must-have functions can be divided into four main functional groups: Data Definition, Retrieval, Update, and Administration.

2.1.1 Relational Database Management System

RDBMSs are the current most popular management system, accounting for 72% of the used DBMSs in 2023 [27]. RDBMSs support databases based on the relational data model proposed by Codd in [21]. This is the most used database model nowadays and allows for straightforward yet expressive queries with high-level languages [41].

In the relation model, each database has a set of tables, also named relations, composed of rows and columns. Each table row is known as a record or a specific instance of data, while each column represents an attribute or a characteristic of that data. Each table typically has a Primary Key (PK), a unique identifier for each row, ensuring that each record can be uniquely identified. Additionally, Foreign Keys (FKs) establish relationships between tables, linking data across multiple tables to create relationships and enforce referential integrity [21].

Imagine we want to store information about a group of students and the respective cities where they live. A simple relational database schema is to create two tables, Student and City, and use the referential property of the relational model to avoid redundancy and ensure integrity by storing the *CityCode*, City's PK, as a FK in the Student table. The Figure 2.1 exemplifies one possible Unified Modeling Language (UML) schema.

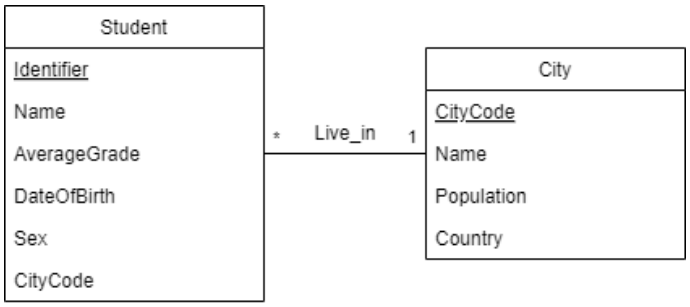


Figure 2.1: A relational database schema representing a school using a UML class diagram. In this simple example, a student living in a single City can be represented with the CityCode Foreign Key and the many-to-one association between the relations.

The relational model emphasizes data integrity, consistency, and the ability to perform complex queries using a standardized query language such as Structured Query Language (SQL), a declarative language originally based upon relational algebra [41].

Normalization is a fundamental principle in relational database design aimed at minimizing redundancy and dependency by organizing fields and table relations. It involves decomposing tables into smaller tables and defining relationships between them to ensure data dependencies and reduce anomalies in data manipulation. The concept of normal forms, introduced by Codd, plays a crucial role in this process, providing guidelines to achieve the desired database structure [21].

The normal forms are a series of progressive stages a database schema can go through to eliminate redundancy and improve data integrity. The First Normal Form (1NF) requires that the table's columns contain atomic, indivisible values, ensuring that each column is unique and each table has a primary key. The Second Normal Form (2NF) builds on this by ensuring that all non-key attributes functionally depend on the primary key, thus eliminating partial dependencies. The Third Normal Form (3NF) goes a step further by removing transitive dependencies, ensuring that non-key attributes are not only dependent on the primary key but also independent of each other. Higher normal forms, such as the Boyce-Codd Normal Form (BCNF), Fourth Normal Form (4NF), and Fifth Normal Form (5NF), address more complex types of redundancy and anomalies. BCNF is a stricter version of 3NF where every determinant must be a candidate key. 4NF deals with multi-valued dependencies, ensuring that a table does not contain more than one independent multi-valued fact about an entity, while 5NF addresses cases where tables are decomposed to eliminate redundancy even in the presence of join dependencies [41].

In this dissertation, unless specified otherwise, the discussion will primarily revolve around databases that adhere to the relational model.

2.1.2 Structured Query Language

Donald Chamberlin and Raymond Boyce started developing SQL at IBM in the early 1970s after their exposure to Codd's relational model [15]. The first version, originally named SEQUEL (Structured English QUEry Language), was crafted to manipulate and retrieve data housed within

IBM's database management system, System R [16]. Commercial implementations of SQL, including Oracle and DB2, emerged in the late 1970s and early 1980s. By 1986, the ANSI and ISO standards groups formally adopted a standardized language definition called "Database Language SQL". [15].

SQL is a language designed to manage data within RDBMSs. SQL can be informally divided into four sublanguages: Data Definition Language (DDL) allows to create and modify database schemas, Data Query Language (DQL) is used to retrieve data, Data Manipulation Language (DML) allows insertions, updates, and deletions of data and Data Control Language (DCL) permits to control access to data [41].

```
CREATE TABLE Student (
    Identifier          bigint,
    Name                varchar(20),
    AverageGrade        int,
    DateOfBirth        date,
    Sex                 char,
    CityCode            int
);

ALTER TABLE Student ADD attribute INT;
ALTER TABLE Student DROP COLUMN attribute;

DROP TABLE Student;
```

Listing 2.1: DDL statements on the table Student. First, a table Student is created with the given attributes. An integer attribute is then added and immediately removed. Lastly, the table Student is deleted.

```
INSERT INTO Student(Identifier, Name, AverageGrade, DateOfBirth, Sex) VALUES
(1,'John',74, '15/02/2001', 'M');

UPDATE Student SET AverageGrade = 80 WHERE Identifier = 1;

DELETE FROM Student WHERE Sex='M';
```

Listing 2.2: DML statements on the table Student. First, a record representing John is inserted into the table Student. Then, its record is updated to set the average grade equal to 80. Finally, all male students are deleted from the database.

```
SELECT * FROM Student;

SELECT Name FROM Student WHERE AverageGrade > 50;
```

Listing 2.3: DQL statements on the table Student. In the first query, all students are retrieved. In the second one, only the students with a positive average grade are retrieved from the table Student.

```
GRANT ALL PRIVILEGES ON Student TO username1;

REVOKE DELETE, UPDATE ON Student FROM username1;
```

Listing 2.4: DCL statements associated with the table Student. The first statement grants "username1" all the privileges on the table Student. The second statement removes the update and deletion privileges from "username1" on that table.

2.1.3 Performance

Database performance is typically assessed by evaluating the query throughput and latency. In specific contexts, the evaluation may include resource consumption as an integral component of performance metrics [63]. In the realm of RDBMSs, performance holds important significance and substantially impacts the success and usability of applications and systems reliant on relational databases.

For applications that heavily depend on RDBMSs for data storage and retrieval, ensuring a smooth user experience is crucial. Performance concerns, such as slow response times, can cause user frustration, potentially threatening the overall usability of the application. Furthermore, the efficiency of RDBMSs performance carries financial implications. Faster query execution results in more effective utilization of hardware resources, decreasing the necessity for costly upgrades.

RDBMSs offer different strategies to improve the performance of the database with additional trade-offs:

- **Indexes:**

  When dealing with vast amounts of data, scanning through every row in a table to locate specific records can be resource-intensive. Just as an index in a book accelerates finding pertinent information, database indexes amplify the speed of retrieving data.

  Indexes are pivotal in enhancing query performance by expediting data retrieval operations, especially when tables contain substantial data volumes. While indexes can significantly boost query speeds, their creation and maintenance involve trade-offs regarding increased storage requirements and potential overhead during data modification operations.

Indexes operate as an additional sorted table, typically containing two crucial components: the key used to sort the records and the specific physical locations of records corresponding to that key within the primary table [57].

- **Configurations:**

  Every RDBMS exposes configuration knobs that control different aspects of database systems, such as user connections, the query optimizer, or the underlying resource management.

  One of the primary benefits of tuning configuration knobs is the ability to tailor the database environment to the specific workload and hardware. For example, adjusting the buffer pool size can improve the speed of data retrieval by keeping frequently accessed data in memory, reducing the need for disk Input/Output (I/O) operations. However, these optimizations come with trade-offs. Over-allocating memory to certain processes might deprive other critical operations of necessary resources, potentially leading to performance bottlenecks. Moreover, changes in workload patterns or hardware configurations might necessitate continuous adjustments to these knobs, making the maintenance of optimal configuration values an ongoing effort [129].

  The different combinations of configuration values impact the system's performance, robustness, and resource usage. The effectiveness of relational database performance relies on optimizing these diverse configurable settings. Correctly setting these values is crucial for generating efficient query plans and enhancing overall system performance [129].

- **Partitioning:**

  When the database experiences an increase in data volume, tables often accumulate a significant number of data rows. Another effective strategy to enhance performance and streamline data usage involves implementing partitioning. Data partitioning involves segmenting a logical database or its components into separate, independent sections.

  One of the primary advantages of partitioning is its impact on query performance. Instead of scanning the entire table during a search query, the process focuses solely on relevant portions based on the defined partitions. This optimized approach significantly reduces query execution time, eliminating the need for exhaustive scanning of the entire table [82].

  On the other hand, managing partitions adds complexity to routine database maintenance tasks such as data loading, indexing, backup, and recovery processes. DBAs need to ensure that these operations are handled correctly across all partitions, which can require additional effort and resources. Moreover, while partitioning can improve query performance for certain types of operations, it can degrade the performance of others. Queries that involve joins across partitioned tables or queries that do not benefit from partition pruning (where the optimizer excludes irrelevant partitions from the query plan) may not see performance improvements or may even suffer due to the overhead of managing partitions [82].

2.1.4 Query plan

A query plan is a set of steps devised by a RDBMS to retrieve data from a relational database using SQL. It is a roadmap detailing how the RDBMS will execute a query, breaking it down into smaller steps such as which indexes or tables to access, the order of data retrieval, and any data operations, like sorting or joining, required to fulfill the query request.

This plan aims to optimize the query's performance by selecting the most efficient approach, considering factors such as available resources and database structure.

In the context of a school database, a necessary component is the table that encapsulates information about the enrolled students. Figure 2.2 exemplifies a potential partial representation of this table.

Identifier	Name	AverageGrade	DateOfBirth	Sex	CityCode
1	John	74	15/02/2001	M	1
2	Joe	54	06/02/2001	M	2
3	Sophia	87	08/10/2001	F	1
4	Bob	63	16/07/2000	M	3
5	Carl	70	23/01/2001	M	3
6	Lily	61	28/07/2000	F	2
7	Diana	92	11/12/2001	F	1

Figure 2.2: Example of a few records of students represented in the table Student.

Imagine that one wants to retrieve from that table the students' identifiers who are Male. A possible SQL query is shown in the Listing 2.5.

```
SELECT Identifier FROM Student WHERE Sex = 'M';
```

Listing 2.5: SQL query to retrieve the identifiers of all male students from the school.

If there are no indexes, a simple query plan is to do a sequential scan on Student while filtering them by Sex. This would require the RDBMS to access every single record in the table.

On the other hand, if the table has an index in the column Sex, the optimizer may use it if it evaluates that it would be faster. In that case, the query plan would just do an index scan.

2.1.5 Query Optimization

When confronted with a query, a DBMS has various access plans to process and generate the query's result. While these plans yield identical final outputs, they differ significantly in their execution time or cost. This cost discrepancy can vary immensely, often spanning several orders of magnitude. Consequently, DBMSs incorporate a vital component known as the query optimizer, which is responsible for evaluating and comparing all available alternatives and ultimately selecting the most efficient plan in terms of execution time.

The query optimization process typically consists of two primary stages: query transformation and optimal plan selection. The transformation stage applies alterations to the query, generating equivalent yet more efficient versions. These alterations involve standardizing query structures, replacing views with their definitions, and simplifying nested queries. Importantly, these changes rely solely on the static attributes of queries, disregarding specific query costs within the targeted DBMS and database. The subsequent stage involves choosing the best plan from the transformed queries. This phase uses a search strategy to explore various access plan possibilities generated in the previous stage. It rigorously evaluates and compares these plans based on cost estimates and heuristics, ultimately identifying the most cost-effective plan to execute and produce the answer for the original query [65].

The estimated cost of a query plan depends on the number of blocks transferred to the main memory. This number is based on the chosen physical organization of the database. In a scenario without any indexes, it can vary between the number of blocks used to store a table or, in the worst case, the number of records in a table (in fact, this cost could even escalate further if the specific blocks containing records are unknown) [50].

Indexes are vital in this optimal plan selection phase since they can significantly reduce access time by avoiding full table scans. Suppose a given table has an index in a column. In that case, the cost of selecting the records with a given condition on that column is approximately the number of blocks containing records that satisfy that condition, which can be significantly lower depending on the type of index and the chosen index structure [65].

Recall the example depicted in Figure 2.2, showcasing a student table with an index on the attribute Sex. Here, the optimizer decides whether employing an index scan outweighs conducting a complete sequential scan on the Students table. Suppose the table contains a relatively small number of records or the index predominantly retrieves a highly prevalent value within the column (consider a scenario where 99% of the students are male). In that case, the query optimizer accurately estimates that utilizing the index does not enhance the query's performance.

Partitions are also crucial in this phase since they allow for the reduction of the overall data scanned and the increase in the efficiency of a query execution plan. The optimizer is able to prune unnecessary partitions based on the query conditions to further accelerate the query optimization process.

Imagine a range partition on the attribute AverageGrade from the example above. This partitioning splits students into Excellent, Good, Ok, and Negative according to average grades. Consider that we want to retrieve all students with a negative average grade. The query optimizer would estimate to use the partition with the range of students with negative grades against sorting in ascending order followed by a sequential scan or simply using a sequential scan. Once again, this decision depends on the table's size and the records' distribution. Still, most of the time, executing the desired query is significantly faster using this partitioning configuration.

The query optimizer gains a more precise understanding by leveraging the selectivity of indexes(i.e., the uniqueness of values in an indexed column). This information enables a more informed selection of the most efficient access path from available indexes. During this process,

the optimizer also leverages information about the partition schemes and their alignment with the query predicates alongside some heuristics to choose the most cost-effective plan to execute.

2.1.6 Workloads

In the context of databases, a workload is a set of representative queries. In large-scale systems, workloads may have different characteristics and, thus, different requirements and needs.

An Online Analytical Processing (OLAP)-style workload is characterized by a set of complex, long-running queries designed to retrieve and process substantial data through aggregations. Predominantly consisting of read queries, this workload is used by organizations to seek comprehensive insights and informed decision-making capabilities. Databases that expect OLAP workloads are optimized for very specific analytical *ad-hoc* queries [17].

On the other hand, Online Transaction Processing (OLTP)-style workload comprises a larger set of short-running queries devised to process transactions. This type of workload has all kinds of queries (read, insert, update, and delete) and is commonly used in transactional-oriented systems. In this case, databases are optimized to handle a vast number of queries concurrently while ensuring the atomicity of the transactions [17].

2.2 Large-Scale Computing

The pervasive integration of the Internet into daily life is an undeniable global phenomenon, with nearly 5 billion individuals actively engaged in its use worldwide [95]. Despite this widespread adoption, the trajectory continues to ascend, evidenced by a staggering 96% of adults in the United States immersing themselves in daily online activities [95].

This growth in Internet usage has generated a need for scalable computing infrastructures, giving rise to large-scale computing paradigms. Large-scale computing encompasses a broad spectrum of applications, technologies, and frameworks designed to address the challenges of extensive networks, vast datasets, and complex computational tasks inherent in the digital landscape [90].

2.2.1 Large-Scale Applications

A large-scale application is a software system that handles many users, data, and interactions on a large scale. These applications are characterized by their ability to manage extensive computational loads, process substantial amounts of data, and serve a high number of concurrent users or requests.

These applications are built to scale efficiently to accommodate increased workload or user demand, both horizontally (or scale out), adding more machines or nodes, and vertically (or scale up), increasing the resources within a single machine [107]. **Scalability** can be defined as *"the ability of a system to accommodate an increasing number of elements or objects, to process growing volumes of work gracefully, and/or to be susceptible to enlargement"* [10].

Large-scale applications frequently encounter unpredictable surges in user traffic or data volume. For that reason, the significance of scalability lies in these applications' ability to manage workloads by allocating resources to meet the increased demand without compromising operational efficiency. Without a scalable architecture, such applications risk encountering constraints that can lead to diminished performance or even service disruptions during peak usage periods [107].

To address the challenges posed by the growing demands on large-scale applications, various strategies and technologies are employed to achieve scalability.

One notable approach is **sharding**, a technique commonly used in RDBMSs to distribute data across multiple nodes or servers [100]. Sharding involves partitioning a large database into smaller, more manageable pieces called shards. Each shard contains a subset of the data and operates independently, reducing the load on a single database server. This distribution allows for parallel processing of queries and transactions, enhancing overall system performance. Sharding can be implemented based on different criteria, such as range-based, hash-based, or even geographic [100].

Another technique to achieve scalability in RDBMSs is **replication**. Replication involves creating copies of the database and distributing them across multiple servers. Each copy, known as a replica, can handle read operations independently, improving the application's read scalability. However, write operations typically need coordination to ensure consistency among replicas [119].

Furthermore, **load balancing** plays a crucial role in distributing incoming requests evenly across multiple servers or nodes. Load balancers ensure that no single server becomes a bottleneck, optimizing resource utilization and preventing any particular node from being overwhelmed [54].

These scalability strategies are not mutually exclusive and are often used in combination to address specific performance bottlenecks in large-scale applications. Implementing a scalable architecture requires careful consideration of the application's requirements, data characteristics, and expected usage patterns.

2.2.2 Cloud Computing

Cloud Computing is an on-demand computing model that gives developers access to large-scale computation resources and the ability to deploy their applications at large scales. It is defined as *"a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction"* [80].

Cloud computing presents an approach to scalability through the flexible provisioning of resources on demand. It facilitates fine-grained, self-service capabilities that allow organizations to dynamically adjust their computing resources without preemptively planning for peak workloads. This flexibility enables scaling up during periods of increased demand and scaling down when resources are not fully utilized [90].

In 2023, 42.5% of European enterprises bought cloud computing services [37]. This is an increasing trend given the reduction in technology infrastructure costs and increase in resource management efficiency [90].

2.3 Reinforcement Learning

This section describes an overview of RL, our chosen approach to automate database partitioning.

In RL, a learning agent interacts with an environment by choosing the actions that lead it towards its goals. Precisely, at each step t, the agent senses an environment state $s_t \in S$ and chooses an action $a_t \in A$. The actions A and states S can be discrete or continuous states, possibly having any number of dimensions, depending on the specific problem and the algorithm being employed [109].

After performing an action a_t, the agent reaches a state s_{t+1} and receives a reward r_{t+1}. This can be modeled mathematically as a Markov decision process [109]. The agent chooses an action in a given state according to a policy π. The goal is to find a policy, mapping states to actions, that maximizes the long-term reward for a starting state at time t:

$$G_t = \sum_{k=0}^{\infty} \gamma^k \cdot R_{t+k+1},$$

where γ is a *discount rate*, $0 \leq \gamma \leq 1$. Rewards are discounted to accommodate the decreasing predictability of future states. Lower values of the *discount rate* drive the agent to act more greedily, prioritizing immediate rewards over potential long-term gains [109].

The agent's accurate estimation of the best-expected value of G_t is crucial. The Q-value represents the expected value of G_t given a certain state s_t and the chosen action a_t:

$$Q(s,a) = E\left[G_t | s_t = s, a_t = a\right]$$

The Q-value formulation can be reformulated by integrating the Q-value of the subsequent state and its associated long-term reward, G_{t+1}. This approach enables the learning of a Q-value estimator through the Bellman-update method given an observed state s_t, a performed action a_t, the reward r_{t+1} of the action, and the next state s_{t+1}:

$$Q(s,a) = E\left[r_{t+1} + \gamma \cdot \max_{a'} Q(s_{t+1}, a') | s_t = s, a_t = a\right]$$

Each Q-value from the observed combinations of s_t and a_t is stored in a matrix accessible to the agent. This allows the learning of a policy from the Q-estimation. A simple approach is to select the action a that greedily maximizes $Q(s,a)$ in the state s. However, during training, the agent selects random actions that are not the best value with a probability ε. This probability decreases over time with a factor called *epsilon decay*. This generates a trade-off between exploiting the best-known option or exploring new options [109].

2.4 Summary

Section 2.1 covers relevant topics in the field of databases, such as RDBMS (Section 2.1.1), SQL (Section 2.1.2), query plans (Section 2.1.4), query optimization (Section 2.1.5), and workloads (Section 2.1.6).

Section 2.1.1 covered the most popular database management system. RDBMSs support databases based on the relational model, where each database has a set of tables composed of rows and columns. This model focuses on data integrity and consistency, and the ability to perform complex queries simply using SQL.

Section 2.1.2 addressed the basics of SQL. It is divided informally into four sublanguages. DDL is used to create and modify database schemas, DQL allows to retrieve data from the database, DML is utilized to insert, update, and delete data, and DCL allows to control the access the database.

Section 2.1.3 highlighted the basics of the primary strategies used in RDBMSs to improve their performance: Indexes, configurations, and partitions. It also identified common performance metrics, such as query throughput, latency, and resource consumption.

Section 2.1.4 described query plans as a set of steps devised by a RDBMS to retrieve data from a relational database. It details how the RDBMS will execute a query, breaking it down into small steps, such as which indexes and tables to access, their retrieval order, and any required data operations. Query plans are necessary to optimize the query runtime according to available resources and the database structure.

Section 2.1.5 explained how queries are optimized at runtime. The optimization process essentially consists of two primary stages: the query transformation and the optimal plan selection. It also highlighted the importance of indexes and partitions in selecting the optimal query plan from the optimizer. Both strategies avoid full table scans and allow the optimizer to choose a different and more cost-effective plan to execute.

Section 2.1.6 highlighted the difference between OLAP and OLTP workloads. OLAP workloads are a set of complex and long-running queries created to retrieve substantial data through table joins. Contrarily, OLTP workloads are a more extensive set of short-running queries commonly used in transactional systems.

Section 2.2 introduces the primary motivation for large-scale computing and the importance of large-scale applications (Section 2.2.1) and cloud computing (Section 2.2.2).

Section 2.2.1 briefly describes large-scale applications. These applications are built to scale horizontally or vertically to accommodate increased demand. Scalability was also defined as *"the ability of a system to accommodate an increasing number of elements or objects, to process growing volumes of work gracefully, and/or to be susceptible to enlargement"* [10]. It also described sharding, replication, and load balancing as common techniques to achieve scalability in these applications.

Section 2.2.2 covered the basics of cloud computing, an on-demand computing model that allows the deployment of large-scale applications in the cloud. It presents an approach to scalability

through the flexible provisions of resources.

Finally, Section 2.3 introduced the main concepts of RL, where an agent learns by interacting with an environment.

Chapter 3

State of the Art

Contents

3.1 Systematic Literature Review 19
3.2 Database Optimization 22
3.3 Automated Index Management 35
3.4 Automated Configuration Tuning 42
3.5 Automated Database Partitioning 51
3.6 Performance Evaluation Metrics 57
3.7 Summary 62

This chapter reviews the state of the art of database optimization. Firstly, Section 3.1 describes the systematic literature review process that guided the research. Then, Section 3.2 describes the different strategies that improve the performance of RDBMSs. Section 3.3 reviews existing automated index management approaches. Moreover, Section 3.4 studies existing automated configuration tuning strategies. Section 3.5 reviews the current automated database partitioning approaches. Then, Section 3.6 identifies the common metrics used to evaluate the performance of RDBMSs. Finally, Section 3.7 summarizes the main findings on the mentioned topics.

3.1 Systematic Literature Review

The dissertation undertook a systematic literature review to investigate the current state of the art within its domain and to tackle several research queries.

3.1.1 Methodology

The systematic review process consisted of the following steps as its methodology:

1. **Literature Research Questions:** Identify the Literature Research Questions that will guide the review.

2. **Data Sources:** Define the data sources to be used.
3. **Search Query:** Elaborate the search query according to the research questions.
4. **Inclusion and Exclusion criteria:** Define the Inclusion and Exclusion criteria.
5. **Screening:** Identify relevant documents according to their similarity to the domain by analyzing the title, abstract, and keywords.
6. **Backward search:** Perform a backward search on potential relevant documents.
7. **Full Text analysis:**: Read the full text of the obtained documents.

The obtained relevant documents can be consulted in the Appendix A.

3.1.2 Literature Research Questions

The following research questions guided the review:

LRQ1. What metrics are used to evaluate the performance of a Relational Database Management System?

Empirical evaluation is crucial in validating and comparing optimizations within a RDBMS. To comprehensively assess performance in the realm of RDBMS, it becomes imperative to identify and understand the metrics used in the literature for this purpose. Section 3.6 will address this research question.

LRQ2. What strategies exist that can improve the performance of a Relational Database Management System?

In the pursuit of optimizing RDBMSs performance, an in-depth exploration and understanding of the diverse array of available strategies is essential. This foundational knowledge will compose the cornerstone of this research, providing the necessary groundwork to delve into. Section 3.2 will address this research question.

LRQ3. What strategies exist to automate the optimization of the performance of a Relational Database Management System?

Understanding what strategies and tools exist to automate the optimization techniques identified in **LRQ2** is another fundamental knowledge of this work. Sections 3.3, 3.4, and 3.5 will cover this research question.

LRQ4. What are the current open problems of automated strategies to improve the performance of a Relational Database Management System?

By pinpointing open problems and outlining future directions arising from the strategies identified in **LRQ3**, we aim to formulate a comprehensive plan that contributes significantly to advancing the current state-of-the-art in automated strategies for database performance optimization. This research question will be addressed throughout Sections 3.3, 3.4 and 3.5.

3.1.3 Data Sources

The data sources selected for the systematic literature review were the digital libraries **ACM Digital Library** and **IEEE Xplore**. ACM Digital Library provides access to vast computing and information technology literature. At the same time, IEEE Xplore is a digital library offering access to engineering, technology, and computer science publications.

3.1.4 Inclusion and Exclusion criteria

To determine which documents would be selected, the literature review obeyed the following inclusion and exclusion criteria:

- **Inclusion criteria:**
  - **IC1:** Must be on the topic of autonomous database management.
  - **IC2:** Research article published on distinguished conferences or journal proceedings.
  - **IC3:** Proposes a novel architecture, tool, or feature within autonomous database management.
  - **IC4:** The contribution is related to index, configuration, or partitioning tuning.
- **Exclusion criteria:**
  - **EC1:** Article unrelated to autonomous database management.
  - **EC2:** This is not a research article published in distinguished conferences or journal proceedings.
  - **EC3:** The novel contribution is unrelated to index, configuration, or partitioning tuning.

3.1.5 Search Queries:

A search query was formulated based on the described research questions. Listing 3.1 illustrates the search query used in the ACM Digital Library, while the Listing 3.2 corresponds to the query used in IEEE Xplore.

```
"query": { Abstract:((optim* OR tun*) AND (index OR knob OR partition*) AND
↪  database AND (automat* OR advisor)) }
"filter": {  }
```

Listing 3.1: Search query used in the ACM Digital Library.

```
"Abstract": database AND (optim* OR tun*) AND (index OR knob OR partition*) AND
↪  (automat* OR advisor)
```

Listing 3.2: Search query used in IEEE Xplore.

3.2 Database Optimization

Optimizing database performance holds significant importance for both users and DBAs. Optimization techniques specifically target increasing throughput or reducing latency within a given workload. Furthermore, specific approaches focus on minimizing resource consumption while imposing limitations on throughput or latency [63]. The following sections will cover the different strategies available to optimize a database, starting with Indexes in Section 3.2.1, followed by Configurations in Section 3.2.2, and finally, Partitions in Section 3.2.3.

3.2.1 Indexes

Numerous indexes have been conceptualized and integrated into database systems for various applications. These indexes can be broadly categorized into one or more of the following categories:

- **Clustered Index:** Clustered indexes change the physical order of records based on the index key. Therefore, only one clustered index can be created for a given table. Almost all RDBMSs create clustered indexes on PKs by default. The essential characteristic of a clustered index lies in its organization of physical data rows, aligning them in correspondence with the index blocks that reference them [57]. Figure 3.1 showcases an example of a clustered index on a column other than the PK.

- **Non-clustered Index:** On the contrary, non-clustered (or unclustered) indexes have a structure separate from the data rows. A non-clustered index contains the key values, and each key value entry has a pointer to the data row containing the key value [57]. Figure 3.2 illustrates a non-clustered index.

- **Single Key Index**: Single key indexes are based on a single column. These indexes are particularly advantageous when a specific attribute is predominantly utilized in isolation without frequent comparisons to others [57]. Figures 3.1. 3.2, 3.3 and 3.4 are all examples of single key indexes.

- **Composite Key Index**: Conversely, composite key indexes are created based on multiple attributes. This index type proves more suitable in scenarios where a set of attributes is consistently utilized together in queries [57].

- **Dense Index:** A dense index consists of pairs of keys and pointers for every record in the table, where each key corresponds to a record, and the pointer points to the physical location of that record [41]. Figure 3.3 illustrates a dense index.

- **Sparse Index:** Instead of having an index record for each data record, a sparse index has only one key-pointer pair per data block. This design choice significantly reduces the space utilization compared to a dense index. However, it results in a longer search time for accessing a record by its key [41]. Figure 3.4 is an example of a sparse index on a PK.

- **Unique Index:** A unique index only allows unique values for each key. It is usually created by default by RDBMSs on unique attributes [57]. Figures 3.3 and 3.4 are examples of unique indexes.

- **Non-unique Index:** In contrast, a non-unique index allows duplicated values for the key [57]. Figure 3.1 illustrates a non-unique index.

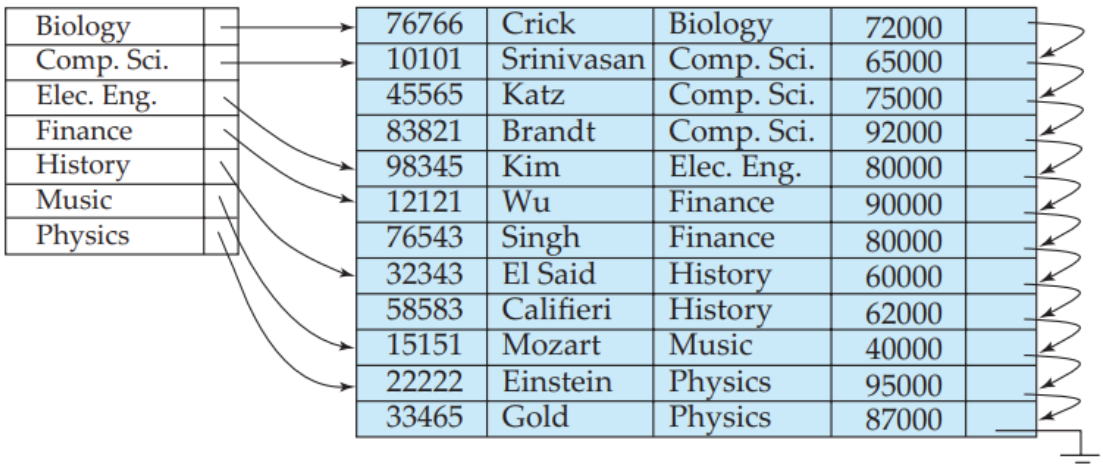


Figure 3.1: A clustered index on the column Department. The table on the right represents an instructor with an ID, Name, Department, and Salary, respectively. Even though ID is a PK, the data needs to be ordered by the Department because of the clustered index on the left. Retrieved from [103].

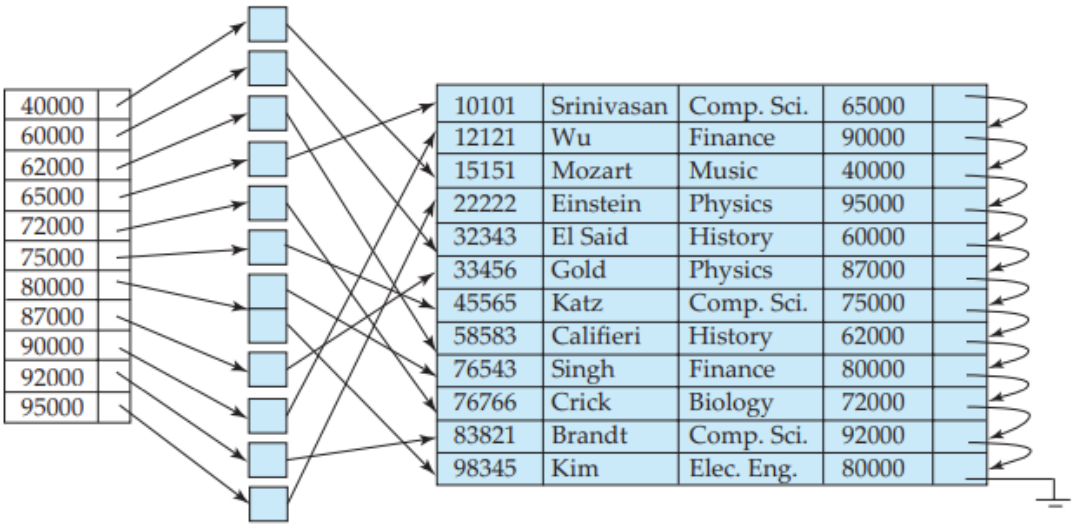


Figure 3.2: A non-clustered index on the column Salary. The table on the right represents an instructor with an ID, Name, Department, and Salary, respectively. The index on the left is non-clustered since the table data is ordered by its PK. Retrieved from [103].

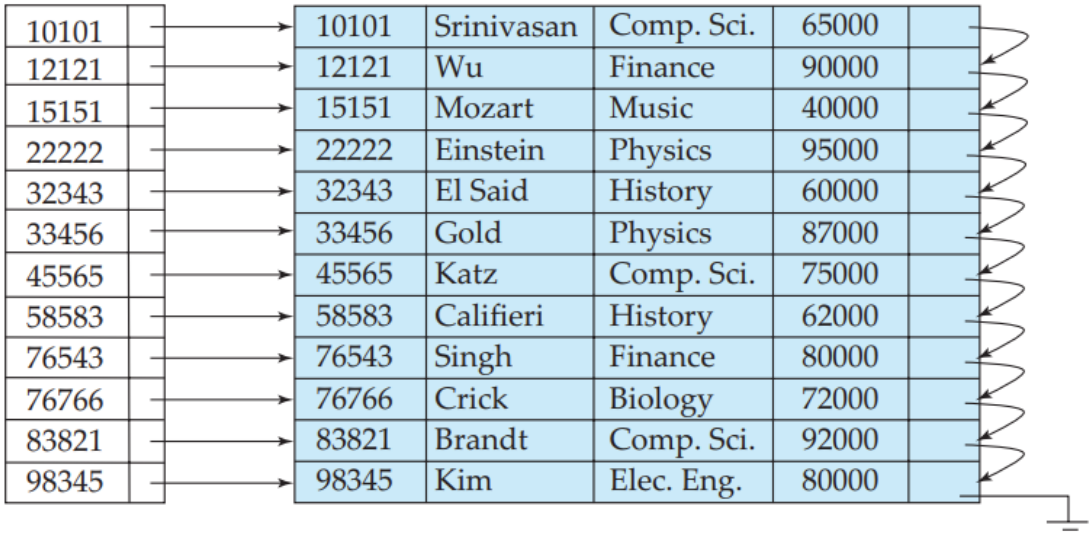


Figure 3.3: A dense index on the column ID. The table on the right represents an instructor with an ID, Name, Department, and Salary, respectively. The index on the left is dense since there is an index entry for each record on the table. Retrieved from [103].

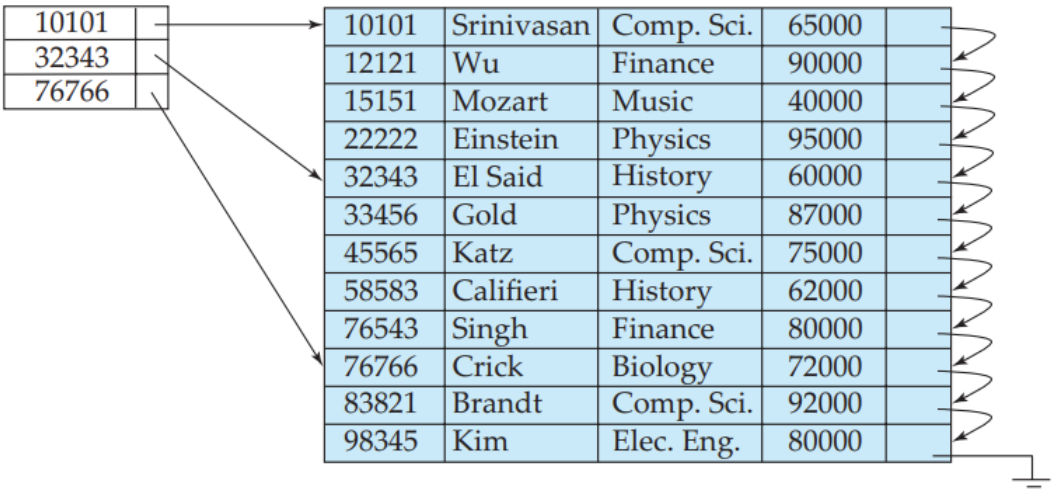


Figure 3.4: A sparse index on the column ID. The table on the right represents an instructor with an ID, Name, Department, and Salary, respectively. The index on the left is sparse since there are records on the table that do not have an index entry. Retrieved from [103].

Indexes can be implemented using a variety of data structures:

- **B-Tree index:** B-trees are the default index type in PostgreSQL. They arrange blocks into a balanced tree structure where all paths from the root to a leaf have identical lengths [41]. B-trees efficiently handle both equality and range queries on sortable data.

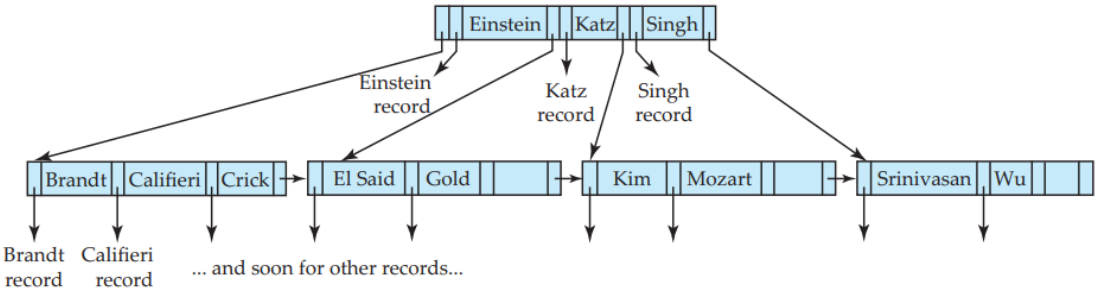


Figure 3.5: A B-tree index on the column Name. Retrieved from [103].

- **Hash index:** Hash indexes employ a hash function h to map a hash key to an integer within the range 0 to $B-1$, where B represents the number of buckets. Each bucket, indexed from 0 to $B-1$, maintains the headers of linked lists, with records stored by linking to the list corresponding to the bucket identified by $h(K)$ [41]. Hash indexes excel in simple equality comparisons with few unique values.

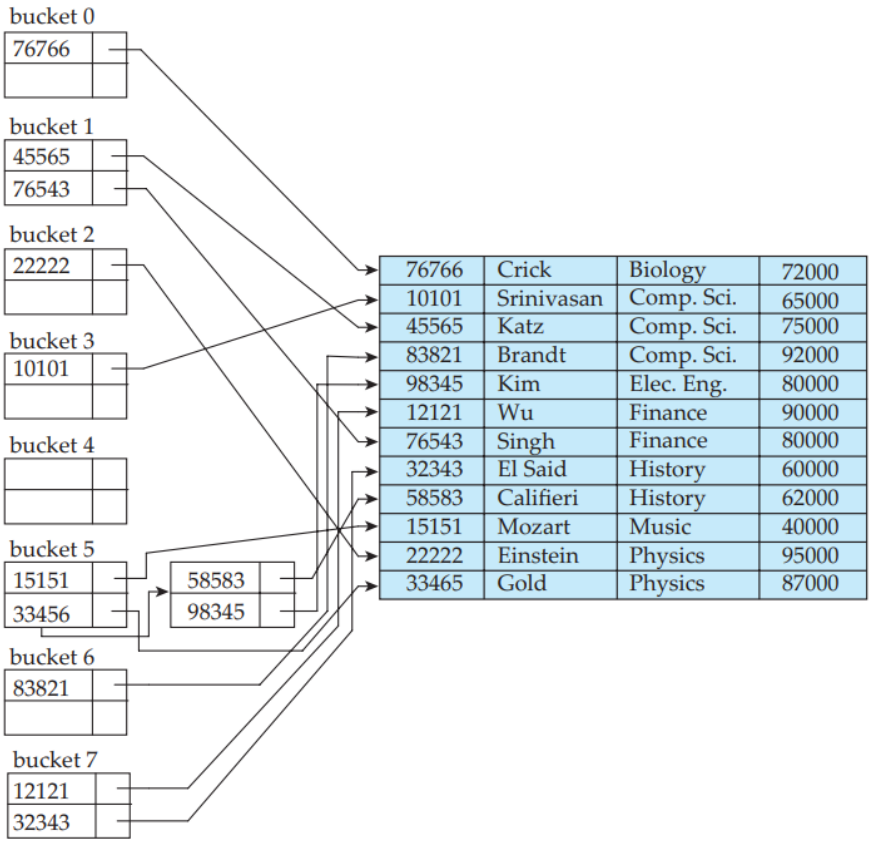


Figure 3.6: A Hash index on the column ID. The hash function used computes the sum of the digits of the ID modulo 8. The table on the right represents an instructor with an ID, Name, Department, and Salary, respectively. Retrieved from [103].

- **Bitmap index:** Bitmap indexes utilize bit arrays, known as bitmaps, and address queries by performing bitwise logical operations on these bitmaps. Traditionally, they perform well with low-cardinality columns, which contain a limited number of distinct values relative to the number of records containing the data [41].

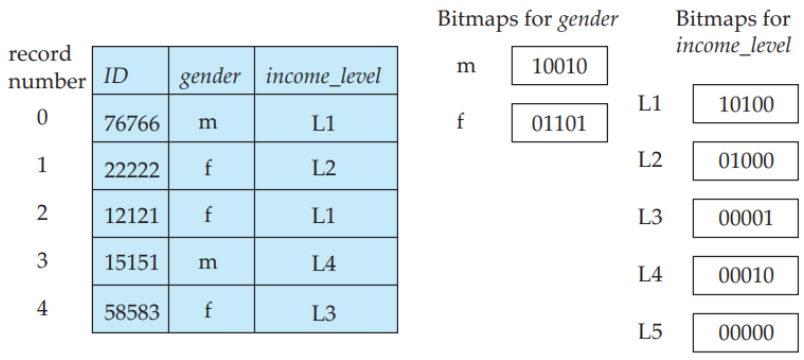


Figure 3.7: Bitmap indexes on the column gender and income_level. The table on the left represents the ID, gender, and income level of an instructor, respectively. Retrieved from [103].

- **R-tree index:** R-trees serve as tree-based data structures designed for spatial access methods, accommodating various spatial data types such as geographical coordinates, rectangles, or polygons. Typically, this tree structure organizes regions by hierarchically grouping them into more significant regions, forming a versatile index for efficient spatial querying and retrieval [41].

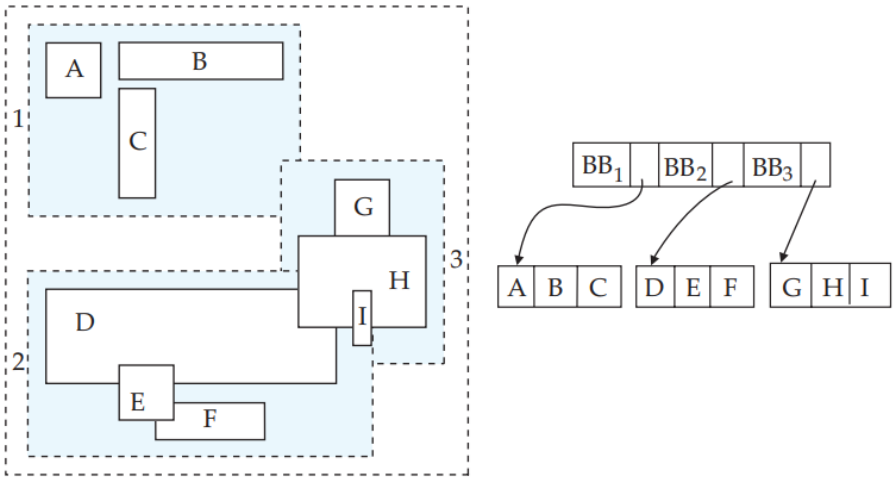


Figure 3.8: An example of a set of rectangles (drawn with a solid line) and the bounding boxes (drawn with a dashed line) of the nodes of an R-tree for the set of rectangles. The R-tree itself is on the right side. The coordinates of the bounding box *i* are referred as *BBi*. Retrieved from [103].

- **Inverted index:** Inverted indexes establish relationships between documents and the words they contain through word-pointer pairs. The pointer, stored in a bucket file, links to instances where a particular word occurs within documents. This structure enables fast full-text searches by quickly identifying documents containing specific terms [41].

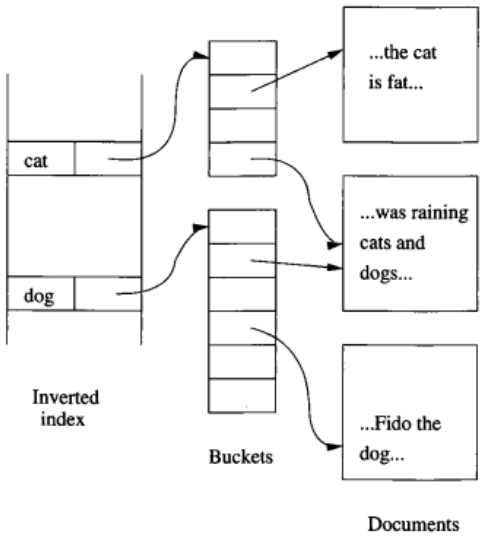


Figure 3.9: An example of an inverted index. The index, on the left, stores the physical location of the documents that contain the given word. Retrieved from [41].

3.2.2 Configurations

PostgreSQL provides access to over 300 distinct configuration options. Typically, these parameters are adjusted by modifying the *postgresql.conf* file, conventionally located within the data directory [51].

```
log_connections = yes
log_destination = 'syslog'
search_path = '"$user", public'
shared_buffers = 128MB
```

Listing 3.3: Example of *postgresql.conf*, the PostgreSQL configuration file. Each parameter is designated on a separate line within the file, adhering to a specific format as detailed in the official documentation [51].

Moreover, PostgreSQL offers alternative methods to interact with these configuration parameters through SQL. Three primary SQL commands facilitate the establishment of configuration defaults. The ALTER SYSTEM command allows for the modification of global default, functionally equivalent to directly editing the *postgresql.conf* file. Furthermore, the ALTER DATABASE command permits the override of global settings on a per-database basis, while the ALTER ROLE command empowers users to override both global and per-database settings with user-specific values [52].

```
ALTER SYSTEM SET shared_buffers = 64MB;
```

Listing 3.4: SQL command to change the global parameter *shared_buffers* to 64MB.

3.2.3 Partitioning

Recall the Student table from the Figure 2.2. One can partition the table by splitting it by its rows or columns. The decision depends on the desired benefits and the system constraints.

Horizontal partitioning, also known as sharding, divides the database horizontally by the rows. This division is achieved by creating partitions, each holding a subset of the table's rows based on the value of the chosen partition attribute. These partitions, or shards, are not necessarily the same size and can contain replicated records. Figure 3.10 demonstrates a possible partitioning configuration by the Sex attribute.

Identifier	Name	AverageGrade	DateOfBirth	Sex	CityCode
1	John	74	15/02/2001	M	1
2	Joe	54	06/02/2001	M	2
4	Bob	63	16/07/2000	M	3
5	Carl	70	23/01/2001	M	3

Identifier	Name	AverageGrade	DateOfBirth	Sex	CityCode
3	Sophia	87	08/10/2001	F	1
6	Lily	61	28/07/2000	F	2
7	Diana	92	11/12/2001	F	1

Figure 3.10: Horizontal Partitioning on the table Student by the attribute Sex. This seems a reasonable partition strategy since it divides the table into two partitions with the same expected size. However, we must ensure that the expected load is distributed evenly while partitioning. This would not be the case if this school had a popular female-specialized program, for example.

Horizontal partitioning enables scalable growth and optimizes resource utilization by spreading data across multiple tables. The choice of a partition key is crucial, as it must ensure a balanced data distribution across partitions to manage the expected workload evenly [4].

The exemplified partitioning meets the size and the distributed workload requirements, but the partitions must have a similar load to avoid *hot* partitions. *Hot* partitions receive more traffic or requests than other partitions. This leads to performance bottlenecks, reduced system efficiency, and data access delays For instance, imagine a school with a popular female-specialized program using this database. Queries related to that program would always use the female partition, which theoretically achieves higher performance since the original table was bigger. However, this could lead to a *hot* partition affecting system availability and increasing the system latency.

Vertical partitioning splits a database vertically based on the columns, so each partition contains a subset of a table's attributes. These partitions are not necessarily disjoint as they can store overlapped columns [82]. The Figure 3.11 exemplifies a possible vertical partitioning.

Identifier	Name	AverageGrade
1	John	74
2	Joe	54
3	Sophia	87
4	Bob	63
5	Carl	70
6	Lily	61
7	Diana	92

Identifier	DateOfBirth	Sex	CityCode
1	15/02/2001	M	1
2	06/02/2001	M	2
3	08/10/2001	F	1
4	16/07/2000	M	3
5	23/01/2001	M	3
6	28/07/2000	F	2
7	11/12/2001	F	1

Figure 3.11: Vertical Partitioning on the table Student to take advantage of frequently accessed data. Consider a school ranking system using students' names and average grades. This partitioning strategy fine-tunes that specific query. However, this configuration adds the overhead of joining the two partitions to get a student's complete profile.

Its primary usage is to reduce the number of reads and the performance costs associated with retrieving frequently accessed data. After analyzing the system's most common queries, one realizes that the students' names and average grades are frequently used together for ranking purposes. In contrast, their birthdate, sex, and city are only used when accessing their complete profile.

In this example, the ranking queries that use the students' names and average grades are really optimized by this partitioning configuration. However, getting all attributes from a student in this configuration has a higher overhead since it is necessary to join the two partitions together. This analysis and trade-off management is the responsibility of the DBA, who must ensure the database is designed and optimized according to the system's needs and requirements.

Current RDBMS provide different strategies to split the database:

- In a **Range partitioning** strategy, each partition is assigned a range of values from some attribute. Usually, each range is defined as inclusive at its lower boundary, exclusive at its upper boundary, and does not overlap with other partitions.

```
CREATE TABLE Student (
    identifier         bigint,
    name               varchar(20),
    averagegrade       int,
    dateofbirth       date,
    sex                char,
    citycode           int
) PARTITION BY RANGE (averagegrade);

CREATE TABLE Student_Excellent PARTITION OF Student
    FOR VALUES FROM (85) TO (100);

CREATE TABLE Student_Good PARTITION OF Student
    FOR VALUES FROM (65) TO (85);

CREATE TABLE Student_Ok PARTITION OF Student
    FOR VALUES FROM (50) TO (60);

CREATE TABLE Student_Negative PARTITION OF Student
    FOR VALUES FROM (0) TO (50);
```

Listing 3.5: Creating the table Student on PostgreSQL with range partitioning on the attribute AverageGrade. Students are divided into Excellent, Good, Ok, and Negative according to their average grades.

- **Hash partitioning** is a method where a hash function based on an attribute determines the partition responsible for storing that record.

```
CREATE TABLE Student (
    identifier         bigint,
    name                varchar(20),
    averagegrade        int,
    dateofbirth       date,
    sex                 char,
    citycode            int
) PARTITION BY HASH (name);

CREATE TABLE Student_1 PARTITION OF Student
    FOR VALUES WITH (MODULUS 3, REMAINDER 0);

CREATE TABLE Student_2 PARTITION OF Student
    FOR VALUES WITH (MODULUS 3, REMAINDER 1);

CREATE TABLE Student_2 PARTITION OF Student
    FOR VALUES WITH (MODULUS 3, REMAINDER 2);
```

Listing 3.6: Creating the table Student on PostgreSQL with hash partitioning on the attribute Name. Hash partitioning tends to perform better than alphabetical range partitioning on strings. In this case, a range partitioning on the name would cause imbalances in the partitions since more names are starting from A to M than from N to Z.

- With a **List partitioning** method, each partition is assigned a list of values to store and update.

```
CREATE TABLE Student (
    identifier         bigint,
    name                varchar(20),
    averagegrade        int,
    dateofbirth       date,
    sex                 char,
    citycode            int
) PARTITION BY LIST (sex);

CREATE TABLE Student_Male PARTITION OF Student
    FOR VALUES IN ('M');

CREATE TABLE Student_Female PARTITION OF Student
    FOR VALUES IN ('F');
```

Listing 3.7: Creating the table Student on PostgreSQL with list partitioning on the attribute Sex. This partitioning strategy can be very powerful or very dangerous. It is the DBA's job to find a proper list of values that evenly distribute the data and the load.

- **Round-robin partitioning** is a strategy to ensure uniform data distribution by choosing the partition based on the insertion order. PostgreSQL does not offer built-in support for this strategy.

- Combining any of the above schemes to design a **Composite partitioning** method is also possible. PostgreSQL allows sub-partitioning any of the "child" partitions.

DBMS employ various architectures that leverage database partitioning for improved efficiency and performance:

- In a **Shared-Nothing** architecture, each node has its own Central Processing Unit (CPU), memory and local disk. Nodes interact solely through the network, with the database physically partitioned across these processing nodes. Accessing multiple database partitions requires distributed transactions, and communication is vital for the commit protocol, ensuring atomicity [92]. This design offers higher performance and availability, but it is challenging to guarantee load balancing and ensure consistency between all the nodes [105].

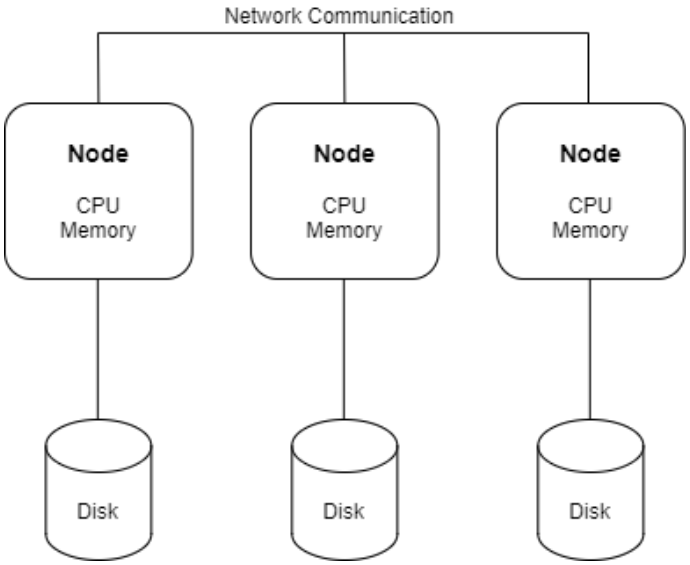


Figure 3.12: A Shared-Nothing architecture diagram. Each node has its own CPU, memory and local disk. Adapted from [88].

- In contrast, a **Shared-Disk** architecture is one in which the nodes share a collection of disk devices, but each has its private memory. This design allows logical database partitioning across nodes, each responsible for accessing and updating specific database portions. This architectural model facilitates dynamic load balancing and guarantees data consistency due to a single data copy [6]. On the other hand, this design scales less well than the Shared-Nothing architecture.

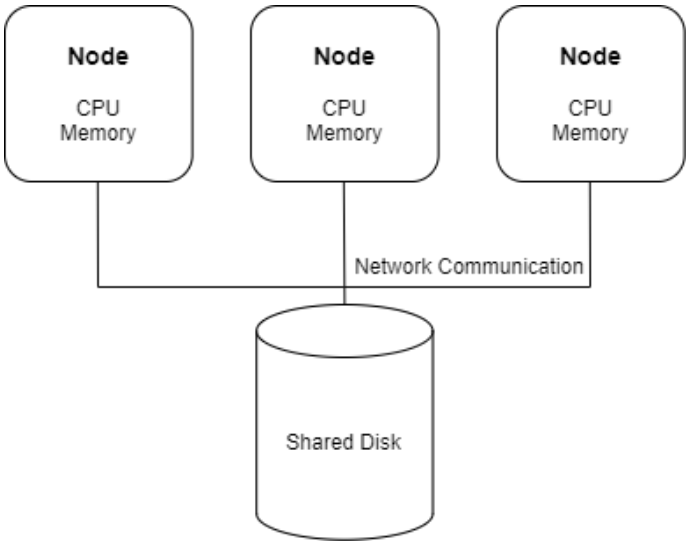


Figure 3.13: A Shared-Disk architecture diagram. Nodes have a shared disk, but each has its own CPU and memory. Adapted from [88]

- The **Shared-Memory** architecture enables multiple nodes to access the entire memory, including main memory and disks, through a fast interconnection network, such as a high-speed bus or cross-bar switch. Under a single operating system's control, this architecture simplifies load balancing and facilitates efficient communication between processors through the main memory. Nonetheless, its real-world implementation faces challenges due to high costs and limited extensibility, limiting its practicality in many scenarios [113].

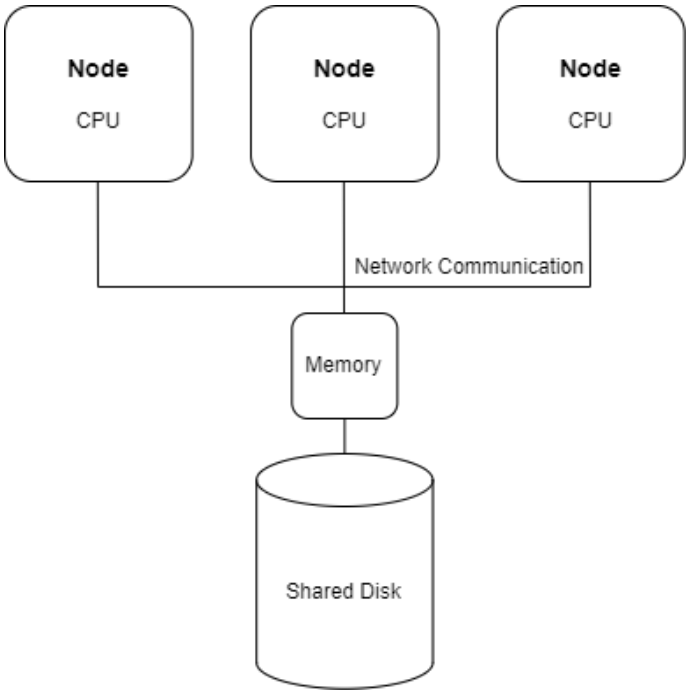


Figure 3.14: A Shared-Memory architecture diagram. Nodes share the entire memory, including main memory and disks. Adapted from [88].

3.3 Automated Index Management

Selecting appropriate indexes poses a challenge even for experienced DBAs because it relies on various factors such as workload, schema, and data characteristics. This complexity is significantly amplified in large-scale deployments encompassing thousands of distinct DBMSs instances. Each of these instances demands tailored tuning, further complicating effective index management [86].

The literature has advocated for automated strategies to address this challenge, and commercial tools are already adopting them. In the following sections, this thesis delves into a contemporary method, Adaptive Indexing, and provides a comprehensive review of index selection algorithms.

3.3.1 Adaptive indexing

Adaptive indexing, the most recent approach, addresses the challenge posed by fully dynamic and unpredictable workloads. This approach constructs indexes partially and incrementally during query processing, specifically in response to query predicates.

However, the primary limitation of adaptive indexing is its need for more reliance on preexisting knowledge or available slack time. It relies on immediate responses to incoming queries without leveraging advanced information or idle periods [48].

However, in contemporary applications like social networks or weblogs, query bursts can be followed by extended periods of inactivity, which presents an opportunity for optimizing this indexing approach [89].

3.3.1.1 Database cracking

Database cracking has pioneered automatic index optimization by adapting the physical database layout to the workload. The core principle of database cracking revolves around creating a cracker index when a column is first used in a query predicate. This specialized index is constructed by duplicating all the data values extracted from the corresponding column in the table's primary data structure. As the same column is repeatedly used in query predicates, the cracker index evolves, continuously refining itself. These refinements occur whenever a finer granularity of key ranges proves beneficial [48].

This optimization process prioritizes key ranges that are queried more frequently, ensuring their representations are continually fine-tuned.

Nonetheless, database cracking exhibits known weaknesses [48]. It demands numerous iterative steps to achieve the optimized representation for a specific key range, resulting in a notable cost for adapting to a new range of queries. Furthermore, the efficiency of transitioning from the initial data representation to a fully optimized state is very dependent on the query pattern, particularly the sequence of boundary keys. It is also worth noting that database cracking performs well in in-memory database systems but may perform less effectively in block-access storage environments.

3.3.1.2 Adaptive merging

Graefe *et al.* [48] propose adaptive merging as an incremental technique to optimize index configurations that adapt more quickly to new data and query patterns than database cracking.

The crucial point of adaptive merging lies in its innovative utilization of partitioned B-trees. Specifically, it concentrates merge operations on key ranges directly relevant to active queries, preserving records within other key ranges in their initial positions. Furthermore, it integrates the merge logic into query execution, distinguishing itself from database cracking. While adaptive merging and database cracking share the attributes of adaptability and incrementality, they fundamentally differ in approach: one relies on merging, while the other relies on partitioning. This distinction results in substantial variations in the speed at which they adapt to new query patterns.

Furthermore, much like traditional B-trees and external merge sort algorithms, the application of adaptive merging to partitioned B-trees aligns well with block-access storage systems. This characteristic distinguishes it from database cracking, which may be less optimal in such storage environments [48].

3.3.2 Index selection algorithms

This section reviews contemporary index configuration selection algorithms that rely on workload analysis. These algorithms find their primary application in online indexing approaches, although their adaptability also extends to offline strategies.

3.3.2.1 Drop

One of the first index selection algorithms is Drop [118]. Whang developed this heuristic considering the problem of selecting a set of indexes that minimize the workload processing cost in a relational database. This cost is measured with the number of I/O accesses.

As the name suggests, this approach successively drops index candidates until no further improvement is achieved. It starts with an initial candidate set, comprising every potential single-column index. At each iteration, every remaining index candidate is temporarily removed, and the workload processing cost is evaluated. The candidate whose removal leads to the highest cost reduction is permanently removed from the candidate set.

This loop is then repeated by dropping two indexes to k indexes simultaneously. Whang states that, in most practical cases, $k = 2$ is sufficient to find near-optimal solutions.

3.3.2.2 Database Engine Tuning Advisor

The Database Engine Tuning Advisor (DTA) [3] is a physical design tool designed specifically for Microsoft SQL Server. It is based on the Anytime algorithm [20] described in this section.

When given a workload consisting of a set of SQL statements and a range of constraints, DTA leverages this information to optimize the database. These constraints include total storage bound, an allotted tuning time, a limit on the number of indexes per table, a maximum number of columns

in an index, and a minimum required improvement for the workload. Optionally, each query within the workload can be assigned a user-defined weight. Otherwise, DTA uses the query's frequency in the workload as its weight.

Given the predefined constraints, DTA's output consists of a physical design configuration that minimizes the total query optimizer estimated workload cost. To determine the optimizer estimated cost of a query for a given configuration, DTA utilizes the "What-If" Application Programming Interface (API). [19]

DTA efficiently divides its tasks into time slices, each representing a complete tuning iteration, invoking all the key modules. If the user stops the process at any time, DTA provides the output of the best configuration achieved at the end of the most recent completed time slice.

Each iteration starts with the module Workload Parsing and Analysis. This module extracts relevant information required for tuning from the workload statements. Next, it compresses the workload by grouping similar queries and chooses a subset of queries to tune from each group. Finally, it analyses the chosen queries to identify the interesting set of tables and columns to focus the tuning process.

Then, within the module Candidate Selection, DTA finds the best configurations for a given query consisting of indexes only, followed by configurations consisting of indexes and materialized views.

Afterwards, in the Merging module, the potential for a sub-optimal recommendation for the input workload arises, primarily due to constraints related to storage or the number of indexes. The Merging process considers the broader context of indexes and materialized views, even if they are not individually optimized for specific queries. Instead, it focuses on configurations that prove beneficial for multiple queries, potentially optimizing the workload as a whole.

The Enumeration module handles the concluding phase of each iteration. This module is responsible for searching all the physical design structures identified by the preceding modules. Its primary objective is to generate a configuration that minimizes the aggregate optimizer's estimated cost for the workload and complies with all the specified constraints.

3.3.2.3 Extend

Extend [97] is an iterative selection algorithm for multi-column indexes that focuses on efficiency without limiting the set of index candidates *a priori*.

Multi-attribute indexes present a favourable trade-off between benefits and costs compared to single-attribute indexes. Consequently, it is highly desirable for index selection systems to incorporate multi-attribute indexes into their considerations. However, determining the optimal set of multi-attribute indexes is computationally challenging, given that the number of potential candidate indexes typically surpasses the number of attributes by several orders of magnitude. In such scenarios, obtaining globally optimal solutions becomes infeasible, as evaluating all conceivable combinations becomes impractical. The complexity of identifying the ideal index selection intensifies due to the intricate interplay among the various index elements, where the presence of one index affects the performance and benefit of others and vice versa. This dynamic relationship is

referred to as index interaction. It is defined as follows: "An index *a* interacts with an index *b* if the benefit of *a* is affected by the presence of *b* and vice-versa" [98].

Extend operates in a greedy stepwise manner, where it continually makes decisions to either add or extend one index per step based on the highest benefit relative to storage usage. The algorithm starts with an empty solution set, selects the single-attribute index that offers the most significant reduction in cost per unit of storage, and adds it to the solution set.

Subsequently, there are typically two options: introducing a new single-attribute index that does not intersect with the current solution set or extending an existing index in the solution set by including a new attribute. The choice between these options is based on the ratio of further cost reduction and additional storage consumption. This method considers the influence of previously chosen indexes in each step, considering index interaction. [69]

The algorithm concludes its execution either when the storage budget is reached or when it becomes impossible to achieve further cost improvements.

3.3.2.4 DB2 Advisor

IBM's DB2 Design Advisor [132] is a tool that automatically recommends physical designs based on the given workload. The advisor has integrated three components for recommending different configurations: indexes and materialized query tables, partitions, and multi-dimensional clusters tables. In this section, we will only cover the component responsible for indexes [114].

The algorithm starts by determining the index candidates. For each query in the workload, columns that appear on equality or range predicates, on ORDER BY or GROUP BY clauses, and on join predicates, are added to the index candidate set.

The set is sorted by their benefit-cost ratio in decreasing order. Any index contained by another index with a superior ratio undergoes merging, subsequently updating their respective ratios. Subsequently, following the descending order, potential index candidates are included in the final index configuration until the storage budget limit is surpassed.

The last step is randomly exchanging some indexes in the final configuration with some that are not part of it. If the resulting cost is lower, it becomes the new final configuration.

3.3.2.5 Relaxation

Relaxation [12] is an algorithm that obtains an index configuration from a very large optimal index set by repeatedly transforming it to satisfy the space constraints.

The algorithm starts by obtaining the optimal index configuration for each query in the workload. The optimal configuration for the workload is the union of all queries' optimal sets.

Then, this vast index configuration is repeatedly relaxed to lower its storage consumption while keeping the benefits as high as possible. Their relaxation is based on five index transformations: Merging two indexes into one, Splitting two indexes, Removing attributes from an index, Promoting an index to a clustered index, and Removing an index.

They use a heuristic to guide their search and choose which transformation to apply to the current configuration. Each transformation can be estimated relative to the expected increase in cost and the expected decrease in space, resulting in a value *penalty*. Therefore, the transformation that minimizes *penalty* is chosen.

The algorithm ends when a given index configuration fits within the space constraints.

3.3.2.6 CoPhy

CoPhy [26] is a Linear Programming (LP) approach to solve the index selection problem. LP is a common choice to solve optimization problems by specifying the optimization goal and constraints with linear equations. Usually, the index selection problem is not scalable to LP due to its complexity. However, one can reduce the solution space by restricting the allowed number of index candidates. Of course, limited candidate sets might lead to sub-optimal solutions.

CoPhy uses binary variables z_{jk} to model whether the index option k is applied to the query j. The index option k is a subset of the index candidates applied to the query k. Its cost is obtained with what-if estimations. The binary variables x_i indicate whether an index from the index candidates is selected, as part of at least one index option.

The constraints of the LP guarantee that a unique index option k is used for each query and that the used indexes i do not exceed the storage budget. Using off-the-shelf solvers, a near-optimal solution can be obtained.

3.3.2.7 Dexter

Dexter [66] is an open-source index selection tool for PostgreSQL developed by Andrew Kane. It is divided into two phases, collecting queries, and generating indexes.

Dexter gathers information about the queries' duration and parse tree in the first phase. Queries with the same parse tree but different values are grouped together as they have the same "fingerprint".

The second phase involves creating hypothetical indexes that show how a query's execution plan would change if the index existed. It starts by getting the cost of the gathered queries with the current index configuration. Then, hypothetical indexes are created on columns that aren't already indexed. The estimated costs and corresponding query plans are once again obtained. All hypothetical indexes that are part of a query plan that reduces the estimated cost compared to the initial configuration are created.

Dexter does not consider deleting existing indexes. Moreover, it only creates hypothetical indexes with one or two columns.

3.3.2.8 Machine Learning-based approaches

With the continuous evolution in the Machine Learning (ML) domain, numerous research studies have been dedicated to optimising index configuration selection for a given workload. This section conducts a review of relevant research efforts in this domain.

SWIRL [70] is a RL methodology designed to accumulate valuable insights about index candidates through an extensive preliminary training phase. This intensive training effort aims to trade off against rapid runtimes, making it particularly relevant for cloud-based business applications dealing with known and unknown queries.

The process of index selection within SWIRL comprises three distinct phases. The initial preprocessing phase involves the DBA selecting a set of representative queries. In conjunction with the database schema, these queries inform the choice of potential index candidates. This step significantly reduces the action space and minimizes training times. Subsequently, random workflows are generated based on these representative queries and divided into training and testing sets.

To address unknown queries, SWIRL employs a workload model that is responsible for converting workload information into a numerical representation suitable for neural networks. It begins by constructing representative query plans from the workload using the what-if optimizer and index candidates. Key operators within the plan are transformed into a textual representation and stored in an operator dictionary, assigning a unique ID to each distinct operator. These IDs collectively form a Bag of Operators (BOO) that numerically represents the operators within a query, akin to the bag-of-words model. Due to the high dimensionality of the BOO, SWIRL employs Latent Semantic Indexing to reduce its number of features.

The subsequent phase involves training the agent to learn which indexes prove beneficial across different workloads by experimenting with various index configurations. The agent formulates a policy utilizing an Artificial Neural Network (ANN) throughout training. Each training episode generates a new workload, acquiring the costs and plans of every query within that workload via the what-if optimizer based on the current index configurations. The agent's action space is then constrained to valid actions, considering previous actions, the remaining budget, or the current workload.

In addition to the workload model, the environment's state representation—comprising the remaining budget, current costs, and active indexes—is numerically transformed for the agent's processing. According to the state, the agent chooses an action from the valid ones, the indexes are created with the what-if utilities, and the next iteration of the training episode starts by getting the new query costs and plans. Each episode concludes upon reaching the budget limit or a user-specified number of iterations.

The final phase involves applying the trained model. Upon receiving a workload, the agent iteratively selects the action with the best-estimated reward until the budget is exceeded. During this application phase, the agent employs the trained ANN to acquire evaluations, offering significantly faster results than the what-if optimizer in existing approaches.

Kossmann *et al.* [70] recommend extending their model to other physical database designs, such as automatic compression or partitioning, as future work.

"Budget-aware Index Tuning with Reinforcement Learning" [121] presents an innovative methodology designed to limit the number of what-if calls in the search for an optimal index configuration. The approach formulates budget-aware index tuning as a Markov decision process

and introduces a solution leveraging Monte Carlo tree search (MCTS), a fundamental technology in RL.

The tuning starts by generating the set of candidate indexes for each query in the workload. These candidates consist of columns that appear in the filter and join predicates, and columns that appear in the GROUP BY, ORDER BY and SELECT clauses. The global set of candidate indexes is obtained by taking the union of each query's set.

In the subsequent phase, the algorithm actively seeks an optimal subset of indexes that effectively minimizes the estimated what-if costs associated with the given workload. Moreover, while pursuing the best index configuration, the algorithm confronts the challenge of determining which configurations and queries deserve what-if evaluations. This dilemma introduces a crucial trade-off between exploration and exploitation. On one hand, the algorithm aims to allocate more what-if evaluations to configurations housing established promising subsets. Conversely, it also explores unknown territories within the search space to find potentially advantageous configurations that have not yet been evaluated.

While this approach allows for limiting the number of what-if calls, it does not allow specifically to establish a user-specified time budget for tuning. Moreover, this approach does not generalize to different workloads or databases.

AutoIndex [130] is an incremental index management system for dynamic workloads. It receives a workload and historical index statistics as input and updates the existing index configuration within resource limits.

This system monitors the workload performance and demands index updates whenever there are performance regressions caused by indexes. AutoIndex utilizes an incremental index management method, matching the workload with templates, extracting promising candidate indexes, and utilizing MCTS to select the high-benefit indexes based on the existing and candidate indexes.

In their method, they first start by reducing the workload size by mapping queries into similar templates. Queries with similar access patterns, such as used columns or SQL format are grouped together, replacing the predicate value with placeholders. Then, candidate indexes are generated based on those templates. Columns on filter and join predicates or columns that appear on GROUP BY and ORDER BY clauses are used to create candidate indexes.

AutoIndex uses MCTS to efficiently identify the most advantageous indexes. The search process concludes upon surpassing a predefined iteration limit or achieving the anticipated performance enhancement threshold. A pivotal component of this approach involves an innovative index benefit estimation model, employing a deep regression framework trained to predict the overall index improvement based on read and write query patterns. This model leverages critical features, including CPU and I/O consumption, alongside data processing costs to accurately measure index performance enhancements.

3.3.3 Summary

This section reviewed several automated index management approaches on RDBMS.

Database Cracking [48] was the pioneering adaptive indexing strategy. It revolves around creating a special index when a column is first used in a query and refining it according to the column ranges.

Adaptive Merging [48] is an adaptive indexing approach that tackles some of the Database Cracking weaknesses. It uses partitioned B-trees to quickly transition the index to a new range of queries.

Drop [118] was one of the first index selection algorithms developed. It consisted of starting with every potential column index and iteratively dropping every candidate until a decent solution was found.

DTA [20] is the Microsoft SQL Server physical design tool. One of its key features is the ability to stop the optimization process at any time.

Extend [97] is an iterative selection algorithm for multi-column indexes that focuses on efficiency without limiting the set of index candidates *a priori* and considering the index interaction.

IBM's DB2 Design Advisor has a component responsible for index optimization [114]. One of the novel features is the enumeration of the index candidate set by the benefit-cost ratio instead of the typical estimated cost.

Relaxation [12] is an index selection algorithm that obtains an index configuration by starting with an extensive optimal index set and repeatedly "relaxing" it to satisfy the space constraints.

CoPhy [26] is a LP approach to solving the index selection problem. It restricts the allowed number of index candidates to reduce the solution space and model the problem as a linear equation.

Dexter [66] is a simple open-source index selection tool that groups identical queries to reduce the space size.

SWIRL [70] is a RL methodology that trades off an extensive preliminary training phase with rapid tuning runtimes. This approach is particularly relevant for cloud-based applications.

Wu *et al.* [121] presented an approach that limits the number of what-if API calls in the search for an optimal index configuration.

AutoIndex [130] is an incremental index management system that uses historical index statistics to update the existing index configuration within resource limits.

Automation of index management techniques in databases faces challenges balancing performance impact, the need for human intervention based on experience, and reliance on triggers or predictions for preemptive optimization, as combining optimization with accurate prediction remains a complex task.

3.4 Automated Configuration Tuning

Originally, DBAs selected a configuration based on their expertise and executed a query workload to gather performance metrics from the database system. If the performance satisfied the desired criteria, they adopted this configuration and concluded the tuning process. However, if the performance was inadequate, DBAs iterated through a diagnostic process, leveraging their experience

to select alternative configuration settings and repeating the evaluation cycle. Unfortunately, these manual tuning methods are time-intensive and heavily reliant on the expertise of the DBA [129].

To overcome these limitations, numerous automated tuning methodologies have been proposed and broadly categorized into four groups: Heuristic tuning methods, Bayesian optimization-based tuning methods, Reinforcement learning-based tuning methods, and Deep learning-based tuning methods. This section provides an in-depth review of these various methodologies and tools following them.

3.4.1 Heuristic

A heuristic approach to problem-solving employs practical methods not bound by guarantees of optimality, perfection, or strict rationality. Instead, these methods prioritize reaching immediate, short-term goals or approximations. In cases where achieving an optimal solution is impossible or impractical, heuristics are effective tools to accelerate the process of finding a satisfactory solution. Heuristics approaches to knob tuning can be divided into rule-based methods or search-based methods [129].

3.4.1.1 Rule-based

A rule-based method follows the rationality of manual tuning where a DBA would follow some rules created by his own experience.

PGTune [116] is an open-source tool that recommends PostgreSQL configuration values according to user-defined information about the database and its hardware configuration. PGTune follows some pre-defined tuning rules to optimize the performance. One rule example from the PostgreSQL documentation is that "a reasonable starting value for *shared_buffers* is 25% of the memory in your system" [53].

Sullivan *et al.* [108] proposed a **probabilistic rule-based methodology** to determine knob settings to maximize performance. Their approach uses pre-defined rules sourced from database experts, aligning them with workload characteristics and system states. Specifically addressing situations involving multiple correlated knobs, the method employs conditional probabilities to make decisions. Based on probability and decision theories, this methodology effectively quantifies uncertainty, facilitating identifying near-optimal decisions within uncertain conditions.

3.4.1.2 Search-based

Search-based tuning methods utilize a hierarchical search approach in the process of selecting optimal knob settings. Initially, these methods segment the knob space into multiple subspaces and execute workloads within each subspace to identify the one demonstrating the highest performance. Following this, they navigate through neighbouring subspaces by altering knob values, aiming to refine the search for optimal settings further. Notably, two prevalent search-based methodologies for knob tuning are employed: an ensemble method that combines multiple search

techniques, and a sampling-optimization method that prioritizes strategic sampling for efficient optimization [129].

OpenTuner [7] is an open-source framework designed for constructing domain-specific, multi-objective program autotuners. This framework facilitates the creation of fully customizable configuration representations, enabling an extensible technique representation tailored to specific domains. Central to the methodology of OpenTuner is the utilization of ensembles containing various search techniques. Multiple pre-built and user-defined search techniques are executed simultaneously, each assessing candidate configurations. Effective techniques that yield superior configurations receive larger test budgets, while underperforming techniques are assigned fewer tests or deactivated entirely. Moreover, multi-objective autotuning represents a versatile approach that enables trade-offs between performance and accuracy and other critical criteria such as energy consumption and memory usage. This methodology allows for developing programs that align with specified performance benchmarks or quality-of-service targets, offering a flexible framework to balance various optimization objectives.

BestConfig [131] is an open-source package that employs a sampling-optimization technique to derive optimal knob configurations. This method begins by gathering a set of configuration samples and subsequently optimizing these selected samples using a search algorithm. Initially, it employs the Divide and Diverge Sampling method to choose and evaluate configuration samples through workload execution or user-defined measurements. Next, it utilizes the Recursive Bound and Search algorithm to optimize these chosen samples by exploring neighbouring configurations. This iterative process operates until the number of allowed tests is surpassed.

3.4.2 Bayesian Optimization

Bayesian Optimization (BO) operates as a sequential, model-based iterative algorithm. It leverages a surrogate model to approximate the objective function, continually refining this model with newly acquired data points gathered through iterative steps. BO commonly initiates the surrogate model with a set of sampled data points. Throughout each iteration, BO employs an acquisition function, employing an exploration-exploitation policy to determine the optimal locations for sampling new points, thus balancing exploring unknown regions with exploiting promising areas within the search space [129].

3.4.2.1 iTuned

iTuned [35] implements an experiment-driven approach to knob tuning based on a Gaussian process. The tuning algorithm starts by collecting samples with the Latin hypercube sampling, a statistical method to generate near-random samples of multidimensional values. It uses those initial samples to conduct experiments to initialize the Gaussian process. These experiments are workload executions with the given parameter configurations that allow measuring metrics such as average transaction response time and throughput. This enables iTuned to balance the trade-off between exploration (understanding the surface) and exploitation (going after known

high-performance regions) that occur in knob tuning. This process is repeated until an expected improvement is obtained or when the user explicitly stops it.

3.4.2.2 Ottertune

Ottertune [115] operates as a tuning service that continually accumulates data from past tuning sessions. Leveraging this data, it constructs models that describe the DBMS's behaviour in response to various knob configurations. The feedback loop generated by each recommendation empowers OtterTune to enhance its models iteratively, refining their accuracy over time. The tool utilises the internal runtime metrics of the DBMS to characterize specific workloads. Ottertune identifies and eliminates redundant or highly correlated metrics. Additionally, it applies Lasso as a feature selection technique on knobs, determining the most influential ones. Possessing these sets of information and leveraging insights from prior tuning sessions, Ottertune uses Gaussian Process regression to suggest knob configurations aimed at optimizing the target metric, typically latency or throughput.

3.4.2.3 ResTune

ResTune [127] is a tool to automatically optimize knobs configuration without violating the service level agreement constraints on latency and throughput. The tool stores meta-features and base models generated from previous tuning tasks on different workloads and hardware environments. In addition to characterising the target workload, this information is used to extract knowledge with weighted ensemble models. This model is then used to predict the performance and resource utilization of new knob configurations. ResTune generates promising configurations of knobs with a Constrained Bayesian Optimization to guarantee the service level agreement constraints while recommending setups that optimize resource utilization.

3.4.2.4 CGPTuner

CGPTuner [14] is an automated configuration tuner leveraging the power of Contextual Gaussian Process Bandit Optimization [71]. This algorithm is an effective tool specifically designed for tasks involving contextual information, much akin to the nature of DBMS workloads. This approach's distinctive advantage lies in its ability to navigate contextual complexness without relying on a predefined knowledge base. CGPTuner optimizes an Information Technology (IT) system's configuration, accounting for multiple layers within the IT stack and dynamically adapting to the prevailing workload.

3.4.2.5 LlamaTune

LlamaTune [67] is a tuner framework that strategically leverages domain-specific knowledge to significantly increase optimisers' sample efficiency. Moreover, it used the power of randomized low-dimensional linear projections, enabling the exploitation of the advantages offered by a

reduced-dimensional space while avoiding the necessity to explicitly identify crucial knobs. One of LlamaTune's distinctive strengths lies in its utilization of biased sampling, a technique carefully designed to promptly detect and account for the impact of certain specialized knob values. This approach plays a pivotal role in enhancing the exploration phase of the optimiser, facilitating early recognition and assessment of the influence of these specialized values on the tuning process. Another unique aspect is that it uses a Sequential Model-based Algorithm Configuration [64] instead of the common Gaussian Process to guide optimization. This surrogate model uses random forest and is shown to perform better in high-dimensional and heterogeneous search spaces [126], as is common in knob tuning.

3.4.2.6 OnlineTune

OnlineTune [128] represents an innovative solution tailored to safely tune online databases within the landscapes of dynamic cloud environments. In response to the inherent dynamism, OnlineTune incorporates environmental factors as contextual features. These include the query arrival rate, query composition within the workload, and underlying data like index usage and data distribution. This tuning system adopts a contextual Bayesian Optimization approach, coupled with context space partitioning, to optimize the database in an adaptive and scalable manner. A notable feature of OnlineTune lies in its initial strategy, commencing the tuning process from configurations known to be inherently safe. OnlineTune intelligently expands the subspace as the optimisation progresses, facilitating further explorations within the configuration space to fine-tune and optimize the database settings iteratively. This proactive and iterative exploration methodology underlines OnlineTune's commitment to enhancing database performance while prioritizing safety within the dynamic environments of cloud environments.

3.4.2.7 LOCAT

LOCAT [122] is a method aimed at the tuning of knobs for Spark SQL applications in real-time. This strategy is designed to balance minimal overhead and performance enhancements. One of its key advancements involves reducing the sample collection size by eliminating configuration-insensitive queries from the workload input. Additionally, it introduces a concept called the Datasize-Aware Gaussian process that effectively models the intricate interplay between performance, workload size, and configuration settings. To further accelerate the tuning process, LOCAT uses the Spearman Correlation Coefficient [123] to identify and focus exclusively on the most influential knobs in the Bayesian Optimization iterations.

3.4.3 Reinforcement Learning

RL revolves around an iterative process where an agent learns a tuning strategy by interacting with the environment. The agent suggests tuning actions based on state features and updates its tuning policy according to received rewards, ultimately optimizing database performance [129].

3.4.3.1 QTune

QTune [75] is a query-aware tuning system integrating Deep Reinforcement Learning (DRL) techniques to optimize database knob configurations. This system operates across various granularities, fine-tuning knobs at query-level, workload-level, or cluster-level. At the workload-level, QTune orchestrates knob optimization aimed at achieving high throughput. Operating at this granularity, QTune generates distinctive feature vectors for each query within the workload, subsequently merging them into a unified vector. These vectors encapsulate crucial information encompassing query specifics such as its type, tables, attributes, operations utilized, and the estimated cost obtained by the query optimizer. One of QTune's significant contributions lies in introducing the Double-State Deep Deterministic Policy Gradient model, leveraging actor-critic networks. This sophisticated model autonomously tackles the tuning challenge by embedding query features and utilizing actor-critic networks to learn relationships among queries, database states, and configurations. By doing so, QTune fine-tunes database knobs, ensuring an optimized performance tailored to the workload and potential future database states.

3.4.3.2 CDBTune

CDBTune [125] is an end-to-end automated tuning system designed for optimizing cloud databases using DRL. Leveraging the Deep Deterministic Policy Gradient method [76], it specializes in fine-tuning knob settings within a high-dimensional configuration space. This process involves utilizing 63 database runtime metrics, such as read/write disk page counts, buffer size, lock time, etc., to build a comprehensive *State* representation of the database's status. Utilizing the Deep Deterministic Policy Gradient as its guiding *Policy* for knob tuning, CDBTune's actor function determines promising knob settings for the next action, while the critic function assesses the potential long-term benefits, thereby guiding the learning process of the actor function.

3.4.3.3 WATuning

WATuning [42] is a workload-aware auto-tuning system built on attention-based DRL. This system pre-trains tuning models for each workload category. WATuning assigns varying weight values to the runtime state of the database system, allowing for precise, fine-grained tuning. Upon encountering a new workload, WATuning calculates the read/write ratio and considers other crucial factors, such as the workload's predicates and estimated costs. These additional workload characteristics augment the decision-making process, enabling WATuning to select a pre-trained tuning model that aligns with a similar read/write ratio and other pertinent workload attributes, ensuring more accurate and refined configuration recommendations. Furthermore, WATuning can dynamically refine its tuning parameters in response to the workload dynamics encountered in practical applications. This adaptability enables WATuning to align more effectively with the dynamic nature of real-world environments, thereby enhancing its ability to provide fine-grained knob recommendations that better suit the current database context.

3.4.3.4 UDO

UDO [117] is a versatile tool designed specifically for offline fine-tuning of database systems to accommodate specific workloads. UDO is designed to mitigate the restarting overhead commonly associated with database adjustments. Its approach involves the classification of database knobs into two distinct categories: heavy knobs, which necessitate a database restart, and light knobs, which do not require such a restart. The innovation within UDO lies in its implementation of a delayed RL algorithm, named delayed Hierarchical Optimistic Optimization, employed to determine optimal settings for heavy knobs. This algorithm strategically collects and rearranges knob settings that share similarities with heavy knobs, thus effectively distributing the reconfiguration cost across multiple knob settings. Different from other tools, this approach involves the selection of knob settings aimed at maximizing the upper confidence bound function. This function contains the average of available tuning rewards, including the delayed rewards.

3.4.3.5 DB-BERT

DB-BERT [111] is a tool designed to tune database knobs by harnessing insights extracted from natural language analysis of manuals and pertinent textual resources. DB-BERT undergoes an initial training phase to refine its model weights by utilising robust, pre-trained language models, specifically the BERT model [31]. This enables the translation of natural language cues into recommended settings. During runtime, DB-BERT dynamically learns to consolidate, adapt, and prioritize these cues, aiming for optimal performance tailored to a specific database system and benchmark. DB-BERT leverages these tuning hints to support the Double Deep Q-Network algorithm in making informed decisions regarding both the selection of knobs and their recommended values. It's important to note the deliberate choice of this simpler RL algorithm owes to the considerably reduced size of the hint space compared to the original configuration space. This selection ensures a more efficient exploration and exploitation of the available hints while optimizing the tuning process.

3.4.3.6 HUNTER

Hunter [13] is an online tuning system designed to improve cloud databases' performance, efficiency, and practicality. Its defining feature lies in a hybrid architecture that combines the use of samples generated by Genetic Algorithms to initialize a more precise exploration by DRL. It leverages Principal Component Analysis, Random Forest techniques, and a Fast Exploration Strategy to accelerate this process. Collectively, these tools reduce the search space and enhance the efficiency of the learning model's updates. Moreover, Hunter introduces an innovative clone and parallelization scheme. This methodology allows for rigorous stress testing of workloads across multiple cloned cloud database instances, thereby enabling faster and more secure knob exploration. This multifaceted approach refines the tuning process and ensures robustness and scalability in optimizing cloud database knobs.

3.4.3.7 BLUTune

BLUTune [58] is an automated knob-tuning system for IBM DB2. This system operates through a multi-stage methodology crafted to manage extensive and complex workloads. Initially, the system's RL agent undergoes training by assimilating optimizer-generated cost estimates of queries, enabling rapid bootstrapping of the model. Subsequently, the process incorporates transfer learning, facilitating precise refinement of the model by leveraging real-time query execution times. Notably, BLUTune employs an advanced advantage actor-critic algorithm [81], an enhancement compared to the original actor-critic algorithms utilized in similar tools. Additionally, it introduces a novel contribution in the form of QEP2Vec, a neural network framework designed for unsupervised learning that creates representations of Query Execution Plans, effectively mapping their high dimensionality into a lower-dimensional embedding space [72].

3.4.4 Deep Learning

Deep Learning (DL) is a ML method based on deep ANN to extract higher-level features or representations from the input data. DL-based methods use multiple layers in the networks and require considerable high-quality samples to train them [73].

3.4.4.1 iBTune

iBTune [110] is a Buffer-tuning tool designed for optimizing Large-scale Cloud OLTP Databases. The tool was conceived to tackle the inherent variability among database instances, necessitating a distinct and highly personalized buffer size while adhering to the confines of total available buffer memory space. When presented with a new database instance, iBTune initiates its process by identifying several potential buffer pool sizes derived from similar instances. These instances exhibit higher cache miss ratios and lower performance. Nevertheless, this strategic selection is grounded in the possibility that these buffer pool sizes are smaller than the current instance's buffer pool size, thereby enhancing resource utilization, and capable of maintaining performance within a defined threshold. Subsequently, employing a DL model, iBTune predicts the performance metrics for the chosen buffer pool sizes. Finally, it selects the minimum size that falls within the predefined performance threshold. However, iBTune is limited in its scope to predicting only the buffer pool size and lacks the versatility to broader tuning knobs beyond this specific parameter.

3.4.5 Summary

This section described automated configuration tuning tools on RDBMS.

Configuration optimization contains heuristic methodologies, categorized into rule-based and search-based approaches. Rule-based methods emulate the intuitive rationale of manual tuning, relying on a DBA's expertise-driven ruleset. In contrast, search-based tuning methods employ hierarchical search algorithms to systematically navigate and identify optimal knob settings within the configuration space.

PGTune [116] is a rule-based, open-source strategy that recommends PostgreSQL configuration values according to pre-defined tuning rules about the database and its hardware information.

Sullivan *et al.* [108] also proposed a rule-based method specifically addressing situations with multiple correlated knobs. The method uses conditional probabilities to recommend configuration values.

OpenTuner [7] is a search-based tool for configuration tuning. Its innovation lies in ensembling diverse search techniques and integrating multi-objective functionalities.

BestConfig [131] is an open-source package that employs a sampling optimization technique to derive optimal configuration values.

Configuration tuning also includes BO-based strategies. BO is an iterative optimization technique that uses probabilistic models to efficiently search and find the optimal solution in scenarios with expensive or time-consuming objective functions, as is the case for configuration tuning.

iTuned [35] is an experiment-driven approach to knob tuning based on a Gaussian process that uses Latin Hypercube Sampling to generate near-random samples.

Ottertune [115] is a tuning service that uses Gaussian Process regression to suggest configuration values. It also applies Lasso to determine the most influential configurations and leverages past tuning sessions to improve its models' accuracy.

ResTune [127] is an automated configuration tuning tool that prioritizes optimizing resource utilization while recommending configuration setups.

CGPTuner [14] is an automated configuration tuner that uses Contextual Gaussian Process Bandit Optimization to leverage contextual workload information.

LlamaTune [67] is a tuner framework that strategically uses domain-specific knowledge to significantly increase optimizers' sample efficiency. It uses a Sequential Model-based Algorithm Configuration instead of the standard Gaussian Process to guide optimization.

OnlineTune [128] is a tuning system focused on safety for online databases in cloud environments, employing a contextual Bayesian Optimization strategy alongside context space partitioning to achieve adaptive and scalable database optimization.

LOCAT [122] is a method aimed at tuning real-time configurations for Spark SQL applications, designed to balance minimal overheads and performance improvements.

Another strategy to optimize the database's configuration is RL. It revolves around an agent learning a tuning strategy by interacting with an environment.

QTune [75] is a DRL-based system that can fine-tune knobs at a query-level, workload-level, or cluster-level. At the workload-level granularity, the optimization is tailored to the workload and also to potential future database states.

CDBTune [125] is an end-to-end automated tuning system designed to optimize cloud database using DRL. It uses 63 database runtime metrics to represent the database's status.

WATuning [42] is a workload-aware tuning system constructed on attention-based DRL. Their system pre-trains different tuning models for each workload category.

UDO [117] is a tool designed for offline fine-tuning databases. One of its main goals is to mitigate the restarting overheads when adjusting database configurations.

DB-BERT [111] uses a pre-trained language model, BERT [31], to gain insights extracted from manuals and textual resources and, consequently, optimize database knobs.

HUNTER [13] is an online tuning system tailored to improve cloud databases' performance, efficiency, and practicality.

BLUTune [58] is an automated configuration tuning system for IBM DB2. They introduce a new way to represent query execution plans utilizing QEP2Vec.

Finally, DL is a ML method based on deep ANN to extract higher-level features or representations from the input data. iBTune [110] leverages DL to optimize the buffer size for large-scale cloud OLTP databases.

Automated configuration tuning encounters challenges in effectively capturing the complex correlations among diverse database configurations. Moreover, although numerous strategies yield significant performance improvements, most of these approaches are finely tuned for specific scenarios, causing them to be impractical and unscalable.

3.5 Automated Database Partitioning

Determining the optimal partitioning strategy for complex database schemas containing multiple tables and join paths is challenging for DBAs. This decision-making process is not exclusively dependent on the structure of the database schema. It is also influenced by table sizes and the workload's dynamics, such as the significance and frequency of joins [60].

The following sections review automated database partitioning strategies that address this challenge.

3.5.1 Deep Reinforcement Learning

In the domain of database partitioning, the utilization of RL is scarcely documented. Hilprecht *et al.* [60] pioneered a Learned Partitioning Advisor for OLAP workloads, leveraging DRL.

Their training methodology involves two distinct phases. Initially, the agent undergoes offline learning, comprehending the trade-offs associated with various partition configurations for diverse workloads. This phase utilizes a simple and fast cost model with limited accuracy. Subsequently, the DRL agent undergoes online refinement, incorporating actual execution times as rewards. This refinement significantly enhances accuracy but substantially escalates computational expenses.

Another notable contribution involves training multiple DRL agents, which collectively constitute a committee. Rather than training a single agent on the entirety of workloads, each expert within this committee focuses on a specific subset of workloads. Their findings demonstrate that employing the inference from the dedicated agent for a particular workload subset yields superior partitioning recommendations when presented with a new mix of workloads. However, this approach encounters limitations when dealing with workloads from a novel database schema, necessitating training a new set of DRL agents.

Another area for improvement lies in the negligence of additional efforts required for reconfiguring the database when suggesting a particular partition configuration. It is the DBA's responsibility to decide whether the potential savings from the recommendations are worth it and to determine an adequate time to deploy them.

3.5.2 Schism

Schism [25] presents a workload-aware methodology designed to improve the scalability of shared-nothing distributed databases handling OLTP workloads. This approach minimizes the number of distributed transactions while ensuring a balanced partitioning scheme.

Given a representative workload and the number of desired partitions, Schism follows five steps to output the partitioning and replication strategy. The first step is to compute the write and read sets for each transaction on the workload.

Then, it creates a graph representing the database and the workload. Each tuple is a node connected with another tuple if the same transaction accesses both. The next step is to use a graph partitioning algorithm to produce a balanced minimum-cut partitioning into the desired number of partitions.

Following the partition, Schism utilizes a decision tree algorithm to extract rules that serve as an "explanation" for the identified partitions. These rules delineate the range of values of frequent attributes within each partition, approximating a compact model that captures the mapping during the partitioning phase.

The final step involves validation, wherein various strategies are compared using the number of distributed transactions as a metric. The solution obtained from the graph partitioning process is compared against the range-predicate partition scheme derived in the previous step. Additionally, comparisons are made with hash-partitioning on the most frequently used attributes and with a full-table replication strategy.

Schism experienced rigorous testing using established benchmark datasets like TPC-C and TPC-E and exploring more complex scenarios derived from social networking websites. These scenarios, recognized for their inherent difficulty in partitioning, provided a robust testing ground for evaluating Schism's scalability and performance.

3.5.3 Database Engine Tuning Advisor

Microsoft's DTA [3] is an automated tool that provides integrated physical design recommendations such as horizontal partitioning, indexes, and materialized views. This section exclusively describes the tool's partitioning module.

Nehme *et al.* [83] presented a partitioning advisor integrated with the underlying parallel query optimizer that thrives in OLAP workloads.

Their work uses the optimizer's cost model and internal data structure, *MEMO*, to find an optimal partition configuration under storage bound efficiently. This advisor differentiates from other state-of-the-art physical design tuners because it does not treat the optimizer as a black box.

Their automated partitioning approach is based on a proposed algorithm called Memo-Based Search Algorithm (MESA), which leverages the *MEMO* structure to perform the what-if optimizations more efficiently and uses a bounding technique on the quality of partial solutions.

They introduce a novel type of partitioning called **-partitioning*, meaning "*that "every" partition or replication option for a base table is simultaneously available. If a table is *-partitioned, the parallel optimizer can pick the concrete partitioning column that is best suited for every given partitioning request (i.e., one that does not requires moving data around) during parallelization post-processing*" [83].

This approach involves the parallel optimizer considering all feasible partitioning alternatives for **-partitioned* tables within a single post-processing step, subsequently presenting execution plans that minimize the overall cost.

However, not all resultant plans remain valid when employing **-partitioned* tables. Specifically, if the final execution plans of queries within the workload propose different concrete partitioning columns for the same table, the configuration becomes invalid. This discrepancy occurs because, in reality, each table can only be physically partitioned in a singular column. Nonetheless, this mechanism empowers the optimizer to consistently opt for the most suitable concrete partition scheme for a **-partitioned* table, thereby establishing lower bounds on the cost of partially specified configurations (those containing some **-partitions*). Remarkably, this process achieves these outcomes without necessitating additional optimization calls.

MEMO is a compact representation of a serial query plan. This data structure is then transformed into the best parallel query plan by a Massively Parallel Processors (MPP) optimizer. Additional details of the *MEMO* structure or its parallelization can be found in [47, 46].

The MESA algorithm starts by creating the workload *MEMO* by making the union of each query representation in the workload. It then uses a brand and bound method to address the ample search space. This method avoids growing the tree as much as possible by choosing only promising nodes to explore with the bounding function described before. Pruning is another important aspect of the algorithm, which discards subtrees that will never be optimal or feasible.

This algorithm avoids multiple calls to the optimizer by gathering common information using the *MEMO* data structure and performing lightweight parallelizations considering the various partition configurations. Leveraging *MEMO* analysis with the **-partitioning* scheme facilitates the computation of lower cost bounds for partial configuration and significantly prunes numerous alternatives without compromising the resultant partitioning quality.

This algorithm demonstrated respectable results during evaluation against alternative methodologies, such as the Rank-Based Algorithm [93] and a Genetic Algorithm. Future enhancements involve extending capabilities to encompass a multi-column partitioning strategy, incorporating range partitioning alongside hash partitioning, and addressing the management of update queries within the workload.

3.5.4 SWORD

SWORD [91] is a workload-aware approach that captures a OLTP workload as a hypergraph and uses it to minimize the number of distributed transactions.

SWORD architecture can be functionally divided into two key components: data partitioning and placement, and incremental repartitioning.

The first module generates a compressed hypergraph representation of the query workload based on the set of transactions and tuples that they access. Each hyperedge represents a transaction, and the nodes spanned by a hyperedge describe the tuples accessed by that transaction. Similar to other approaches, a *k-way balanced min-cut partitioning* of the hypergraph yields a k balanced database partitions and minimizes the number of distributed transactions. This hypergraph partitioning process is facilitated by the hMetis package [68]. Then, hypergraph compression is achieved by clustering nodes into multiple groups and collapsing each group into a *virtual node* via a hash function applied to their primary keys.

The second component, incremental repartitioning, assumes responsibility for monitoring workload changes and maintaining statistics on workload access patterns. Whenever the existing partitioning becomes sub-optimal, this mechanism triggers data migration. Crucially, this migration is incremental, occurring exclusively during periods of low activity.

While SWORD conducted an insightful experiment to elucidate the trade-offs between partitioning time and quality using the TPC-C dataset, it is noteworthy that a more comprehensive comparison with other state-of-the-art approaches is lacking. This comparison would provide a broader perspective on SWORD's efficacy and distinctiveness within workload-aware data partitioning strategies.

3.5.5 AdaptDB

Lu *et al.* presented AdaptDB [77] as an adaptive storage manager for OLAP workloads in distributed settings. AdaptDB's core objective lies in dynamically adapting to shifts in the workload by employing incremental repartitioning. This strategy aims to achieve optimal join performances by avoiding exhaustive table scans or shuffle joins.

AdaptDB creates an initial partitioning configuration using Amoeba's method [101]. This up-front partitioning follows a partitioning tree structure. The tree's root splits the data into two partitions based on a randomly chosen attribute. Subsequently, each partition undergoes further random partitioning based on different attributes until it reaches the desired block size.

AdaptDB's essential contribution is the hyper-join algorithm to join queries over multiple tables. Their work demonstrates that hyper-joins are preferred over shuffle joins if the number of blocks overlapped from the tables is minimal.

The repartitioning phase employs two distinct techniques. The first, *two-phase partitioning*, introduces a new join predicate into the root. In the initial phase, splitting is performed based on the median of the join attribute, followed by splitting on the selection attributes in the subsequent phase.

However, the abovementioned technique optimizes a partitioning tree for a single join attribute, which poses a challenge when tables have multiple foreign keys joining with different tables. AdaptDB introduces a *smooth repartitioning* technique to address this. This approach manages multiple partitioning trees, utilizing the initial tree levels to partition data based on join attributes. It maintains separate trees dedicated to a specific common join attribute within a table. As joins occur, blocks gradually transition from one partitioning tree to another. AdaptDB leverages a cost model and query log to determine the optimal blocks for movement and the most suitable timing for these transitions.

AdaptDB demonstrated promising results compared to systems utilizing scans and shuffle-joins with TPC-H and real-world datasets. However, due to its nature as a self-tuning storage system, direct comparison with partition advisors in traditional systems poses challenges.

3.5.6 Horticulture

Horticulture [87] is an automated tool dedicated to generating optimal database designs for parallel OLTP systems reliant on stored procedures. Its primary objective is to formulate an efficient partitioning strategy that simultaneously minimizes distribution overhead and maintains a balance in temporal access skewness.

Like other state-of-the-art approaches, Horticulture employs search techniques to address the vast search space size. They adapt the Large-Neighbourhood Search (LNS) [102] method to explore the ample solution space without getting entangled in a local minimum solution. The LNS methodology compares potential solutions using a cost model that estimates the performance of the DBMS with a particular design without the necessity of deploying the database. This cost model is estimated to reduce the number of distributed transactions while evenly distributing the execution workload across the cluster.

Horticulture initiates its optimization process by creating a computationally straightforward solution based on the most frequently accessed columns. This configuration serves as an upper limit of the optimal solution.

Subsequently, an iterative approach begins where a new, incomplete solution is generated by intentionally altering a subset of the previous solution. This modification process, termed "relaxation", involves randomly selecting tables and reconfiguring their partitioning attributes.

Afterwards, a local search technique is employed to achieve a new solution using this incomplete solution. If the resulting solution is cheaper than the current best solution, it replaces it as the new optimal choice. This iterative search process stops when no further modifications can be made to neighbouring solutions or after a specific number of iterations where subsequent solutions fail to outperform the current best solution.

Extensive benchmark comparisons against diverse workloads, including assessments against approaches like Schism (section 3.5.2), showcased promising performance results for Horticulture. Future developments for Horticulture aim to support mixed workloads incorporating both OLTP and OLAP. It can be achieved by creating a new cost model that adapts to multiple objectives.

3.5.7 DB2 Design Advisor

As described in Section 3.3.2.4, IBM's DB2 Design Advisor [132] is a tool that automatically recommends physical designs based on the given workload. In this section, we will only cover the component responsible for partitioning [94].

DB2 partition advisor uses the query optimizer to recommend candidate partitions for each table and evaluate the candidates' combinations. It is designed to be used in a shared-nothing parallel database system.

The advisor starts by utilizing the query optimizer in the RECOMMEND mode to get candidate partitions for each table in each statement in the workload. In this mode, for each SQL statement, the optimizer recommends one candidate partition for each table referenced by the query.

Since storing only one candidate per table might miss the overall best partition for the workload, the advisor goes through a partition expansion phase. In this phase, the tool generates additional candidate partitions on keys that are a common subset from two or more existing candidate partitions from the same table, as long as that key has a sufficient predefined number of values.

Subsequently, the DB2 partition advisor performs an enumeration algorithm to combine candidate partitions from different tables and evaluates the workload using the optimizer's EVALUATE mode. In this mode, the optimizer optimizes the query according to the hypothetical partition configuration and returns its query plan and cost estimation.

The DB2 cost model is a linear combination of I/O, CPU, and communication costs. The cost model is based on *per-table* statistics, such as table cardinality, column cardinality, and value distributions, and *per-node* statistics, such as the number of data pages.

To estimate the cost under new partitions, only *per-node* statistics need to be updated according to the new partition configuration and the *per-table* statistics.

Using the EVALUATION mode in an unguided enumeration of all possible configurations would not be feasible. To ease the search process, the advisor first calculates the benefit for each candidate partition by accumulating the differences between the estimated cost in the regular mode and the RECOMMEND mode of queries with that partition. This information will be helpful for the enumeration algorithm.

In their study, Rao *et al.* explored two distinct methods: a rank-based approach and a randomized search algorithm, specifically a genetic algorithm [43]. The subsequent discussion will focus on the rank-based algorithm as it was their chosen method and the one with the best results.

The rank-based algorithm is a tree-based search algorithm where each node corresponds to a partition configuration. It starts with a root node representing an initial configuration where each table is assigned the candidate partition with the highest benefit. The algorithm explores all potential configurations differing in only one partition to expand a node.

To evaluate a configuration, they execute each statement in the workload in the EVALUATE mode. The algorithm caches the query costs to optimize efficiency and prevent redundant evaluations.

Expanded nodes are ranked according to their benefit values and stored in a queue. The configuration with the highest rank is positioned at the forefront of the queue for subsequent exploration. The enumeration process continues until a user-specified time limit is reached.

Upon completion, the advisor provides the best partition selected for each table and the corresponding cost for executing the workload.

3.5.8 Summary

This section described automated partitioning tuning tools on RDBMS.

Hilprecht *et al.* [60] presented a DRL approach to recommend partitions for databases that expect OLAP workloads.

Schism [25] is a workload-aware methodology designed to improve the scalability of shared-nothing distributed databases handling OLTP workloads by minimizing the number of distributed transactions in a partitioning scheme.

Microsoft's DTA partitioning advisor [83] is a tool integrated with the underlying parallel query optimizer that thrives with OLAP workloads.

SWORD [91] is another partitioning approach that minimizes the number of distributed transactions. It captures a OLTP workload as a hypergraph and produces a *k-way balanced min-cut partitioning* to produce k balanced database partitions.

AdaptDB [77] is an adaptive manager for OLAP workloads in distributed settings. One of its core objectives is dynamically adapting to workload shifts by employing incremental repartitioning.

Horticulture [87] is an automated tool designed to generate database designs for parallel OLTP systems. Its primary goal is to build an efficient partitioning strategy that minimizes distribution overhead and maintains a balance in temporal access skewness.

IBM's DB2 Design Advisor [132] has a component responsible for partitioning. DB2 partition advisor [94] uses the query optimizer to recommend partitions in a shared-nothing parallel database system.

Automated partitioning tuning tools are designed for either analytical or transactional workloads and face challenges adapting to mixes or changes in the workload. Moreover, the reviewed strategies often overlook partitioning maintenance and adjustment overheads.

3.6 Performance Evaluation Metrics

It is well-established in the literature to use TPC's workloads as benchmarks to evaluate the system performance. *"The Transaction Processing Performance Council (TPC) is a non-profit corporation focused on developing data-centric benchmark standards and disseminating objective, verifiable data to the industry"* [23]. Tables 3.1, 3.2, and 3.3 summarize the benchmarks used to evaluate the performance of the automated tools reviewed in Sections 3.3, 3.4, and 3.5 respectively.

Most tools have utilized some established benchmark from TPC alongside custom datasets and the respective workload. These datasets usually represent real customer data or are synthetically

created to assess a specific problem. Database Cracking (Section 3.3.1.1), Adaptive Merging (Section 3.3.1.2), the Sullivan *et al.* approach (Section 3.4.1.1) and OpenTuner (Section 3.4.1.2) did not specify the benchmarks used in their evaluation. Dexter (Section 3.3.2.7) and PGTune (Section 3.4.1.1) are open-source tools that did not present any evaluation.

None of the index and configuration tuning tools have restricted their work to a specific type of workload. Hence, in most tools, the evaluation consisted of both OLAP and OLTP workloads, or in some cases, mixed workloads.

On the contrary, automated partitioning tools tackle a specific workload in their problem. For that reason, the transactional and analytical TPC benchmarks are used for tools that expect a OLTP and OLAP workload, respectively.

Tool	Not specified	Custom Datasets	TPC-H	TPC-D	TPC-DS	TPC-C
Database Cracking [48]	•					
Adaptive Merging [48]	•					
Drop [118]		•				
DTA [20]		•	•			
Extend [97]		•				
DB2 [114]				•		
Relaxation [12]		•	•			
CoPhy [26]			•			
Dexter [66]	-	-	-	-	-	-
SWIRL [70]		•	•		•	
Budget-aware [121]		•	•		•	
AutoIndex [130]		•			•	•

Table 3.1: Workload benchmarks used to evaluate performance in automated index management tools.

Tool	Not Specified	Custom Datasets	TPC-H	TPC-W	TPC-C	TPC-DS
PGTune [116]	-	-	-	-	-	-
Sullivan *et al.* [108]	•					
OpenTuner [7]	•					
BestConfig [131]		•				
iTuned [35]		•	•	•		
Ottertune [115]		•	•		•	
ResTune [127]		•			•	
CGPTuner [14]		•				
LlamaTune [67]		•			•	
OnlineTune [128]		•			•	
LOCAT [122]		•	•			•
QTune [75]		•	•			
CDBTune [125]			•		•	
WATuning [42]		•			•	
UDO [117]			•		•	
DB-BERT [111]			•		•	
HUNTER [13]		•			•	
BLUTune [58]		•	•			•
iBTune [110]		•				

Table 3.2: Workload benchmarks used to evaluate performance in automated configuration tools.

Tool	Workload	Custom Datasets	TPC-C	TPC-E	TPC-H	TPC-DS
Schism [25]	OLTP	•	•	•		
Horticulture [87]	OLTP	•	•	•		
SWORD [91]	OLTP		•			
AdaptDB [77]	OLAP	•			•	
Hilprecht *et al.* [60]	OLAP	•			•	•
DTA [83]	OLAP	•			•	•
DB2 [94]	OLAP	•			•	

Table 3.3: Workload benchmarks used to evaluate performance in automated partitioning tools.

Tables 3.4, 3.5, and 3.6 showcase the metrics used by the tools reviewed in Sections 3.3, 3.4, and 3.5, respectively, to evaluate the DBMS performance.

Almost all of the automated index management strategies used the workload runtime, also known as the total latency, as their evaluation metric. Database Cracking (Section 3.3.1.1) and Adaptive Merging (Section 3.3.1.2) used Overhead per query since their approach relies on incrementally adapting the index physical design upon incoming queries. AutoIndex (Section 3.3.2.8) was the only tool using the average throughput in addition to the runtime. In the context of

databases, throughput is the number of queries the database can execute in a given amount of time.

Most automated configuration tools used runtime, throughput, or both in their experimental evaluations. ResTune (Section 3.4.2.3) also uses the ratio of $operations/sec$ and the CPU usage to evaluate its work since its objective is reducing resource consumption while maintaining a good performance level. iBTune (Section 3.4.4.1) also uses the CPU usage as a metric to assess since it is a specific buffer-tuning tool. Given their safety prioritization, OnlineTune (Section 3.4.2.6) also measures the number of unsafe configurations as an evaluation metric.

Tools that automate the partitioning of transactional databases used throughput to evaluate their performance. On the contrary, automated partitioning tools for analytical databases used the total runtime to assess their performance. While AdaptDB (Section 3.5.5) and the Hilprecht *et al.* approach (Section 3.5.1) used the actual workload execution time, DTA (Section 3.5.3), and DB2 partition advisor (Section 3.5.7) used the total workload cost estimated by the optimizer.

Tool	Overhead per query	Runtime	Average Throughput
Database Cracking [48]	•		
Adaptive Merging [48]	•		
Drop [118]		•	
DTA [20]		•	
Extend [97]		•	
DB2 [114]		•	
Relaxation [12]		•	
CoPhy [26]		•	
Dexter [66]	-	-	-
SWIRL [70]		•	
Budget-aware [121]		•	
AutoIndex [130]		•	•

Table 3.4: Metrics used to evaluate performance in automated index management tools.

Tool	Runtime	Throughput	Operations/sec	CPU usage	Num. Unsafe Configs
PGTune [116]	-	-	-	-	-
Sullivan *et al.* [108]		•			
OpenTuner [7]	•				
BestConfig [131]		•			
iTuned [35]	•	•			
Ottertune [115]	•				
ResTune [127]	•	•	•	•	
CGPTuner [14]		•			
LlamaTune [67]		•			
OnlineTune [128]	•	•			•
LOCAT [122]	•				
QTune [75]	•	•			
CDBTune [125]	•	•			
WATuning [42]	•	•			
UDO [117]	•	•			
DB-BERT [111]	•	•			
HUNTER [13]	•	•			
BLUTune [58]	•				
iBTune [110]	•	•		•	

Table 3.5: Metrics used to evaluate performance in automated configuration tools.

Tool	Workload	Throughput	Runtime
Schism [25]	OLTP	•	
Horticulture [87]	OLTP	•	
SWORD [91]	OLTP	•	
AdaptDB [77]	OLAP		•
Hilprecht *et al.* [60]	OLAP		•
DTA [83]	OLAP		•
DB2 [94]	OLAP		•

Table 3.6: Metrics used to evaluate performance in automated partitioning tools.

While traditionally not used for comparing optimization approaches with other works, it is beneficial to incorporate additional metrics for a comprehensive evaluation of the DBMS performance. Examining **resource usage**, which includes the levels of **CPU**, **memory**, and **disk I/O** utilization, becomes important in identifying potential performance bottlenecks within the system [2, 39]. Additionally, assessing the **cache hit ratio**, calculated as the number of cache hits divided by the total number of cache requests, provides insights into the efficiency of the caching mechanism [39]. Moreover, using the **tail latency** with high percentiles, such as 95th and 99th, provides insight into the distribution of response times in a system. The 99th percentile latency is

the response time exceeded by only 1% of the total queries. This metric is often used in real-time systems where a consistent and low-latency experience is critical for user satisfaction [34].

3.7 Summary

Section 3.1 described the systematic literature review performed for this work. It started by explaining the adopted methodology and the corresponding research questions. **LRQ1** aims to identify the metrics used to evaluate database optimization strategies. **LRQ2** focuses on identifying and understand available strategies to optimize a RDBMS. **LRQ3** seeks to comprehend existing automated strategies to optimize a RDBMS. Finally, **LRQ4** aims to identify open problems of these strategies. Then, it defined the data sources, the inclusion and exclusion criteria, and the search query used for the review.

3.7.1 LRQ1. What metrics are used to evaluate the performance of a Relational Database Management System?

Section 3.6 focused on **LRQ1** to identify the evaluation metrics used in the reviewed works. Almost all works use TPC benchmarks to assess their performance. Most index and configuration optimization tools use runtime, throughput, or both as their evaluation metrics. The automated partitioning strategies use throughput if they expect a OLTP workload and the workload runtime if they expect a OLAP workload. Transactional databases emphasize high transaction rates and low response times for individual queries, while analytical databases focus on optimizing the execution time of complex analytical queries over large datasets.

3.7.2 LRQ2. What strategies exist that can improve the performance of a Relational Database Management System?

Section 3.2 addressed the **LRQ2** by identifying and explaining the existing database optimization strategies. Section 3.2.1 described the concept of indexes, how they can be used to improve the RDBMS performance, and the different types and structures of indexes used. Section 3.2.2 explained what are the RDBMS's configuration knobs, their importance, and how to change them in a PostgreSQL environment. Finally, Section 3.2.3 described the motivation to use partitions in a large-scale environment, the different types and strategies to partition a database, and the different DBMS's architectures that leverage database partitioning to improve performance.

3.7.3 LRQ3. What strategies exist to automate the optimization of the performance of a Relational Database Management System?

Section 3.3, which focused on **LRQ3**, introduced the motivation to automate the management of indexes in a DBMS. Section 3.3.1 described the approaches that adapt the physical design of an index based on the queries received, Database Cracking (Section 3.3.1.1) and Adaptive Merging (Section 3.3.1.2). Section 3.3.2 described different index selection algorithms that rely

on workload analysis, including Drop (Section 3.3.2.1), DTA (Section 3.3.2.2), Extend (Section 3.3.2.3), DB2 (Section 3.3.2.4), Relaxation (Section 3.3.2.5), CoPhy (Section 3.3.2.6), Dexter (Section 3.3.2.7), SWIRL (Section 3.3.2.8), Budget-aware Index Tuning (Section 3.3.2.8), and AutoIndex (Section 3.3.2.8).

Section 3.4 also addressed **LRQ3** by describing the automated configuration tuning process. Section 3.4.1 explained the heuristic approach to solve the knob tuning problem. It included the rule-based methods PGTune (Section 3.4.1.1) and Sullivan *et al.* approach (Section 3.4.1.1), and the search-based methods OpenTuner (Section 3.4.1.2) and BestConfig (Section 3.4.1.2). Section 3.4.2 introduced an bayesian-based approach that included iTuned (Section 3.4.2.1), Ottertune (Section 3.4.2.2), ResTune (Section 3.4.2.3), CGPTuner (Section 3.4.2.4), LlamaTune (Section 3.4.2.5), OnlineTune (Section 3.4.2.6), and LOCAT (Section 3.4.2.7). Section 3.4.3 explained RL-based tools, formed by QTune (Section 3.4.3.1), CDBTune (Section 3.4.3.2), WATuning (Section 3.4.3.3), UDO (Section 3.4.3.4), DB-BERT (Section 3.4.3.5), HUNTER (Section 3.4.3.6), and BLUTune (Section 3.4.3.7). Finally, Section 3.4.4 described a DL-based solution, iBTune (Section 3.4.4.1).

Section 3.5 also approached **LRQ3** and described the primary motivation for automating the database partitioning process. It also contained descriptions of existing tools, such as Hilprecht *et al.* approach (Section 3.5.1), Schism (Section 3.5.2), DTA partition advisor (Section 3.5.3), SWORD (Section 3.5.4), AdaptDB (Section 3.5.5), Horticulture (Section 3.5.6), and DB2 partition advisor (Section 3.5.7).

3.7.4 LRQ4. What are the current open problems of automated strategies to improve the performance of a Relational Database Management System?

Throughout Sections 3.3, 3.4, and 3.5 several open challenges of automated strategies to optimize RDBMSs were identified.

Adaptive Indexing (Section 3.3.1) falls short in leveraging user inactivity, presenting an avenue for improvement.

Index selection tools (Section 3.3.2) have inaccurate optimizer estimates, driving the integration of ML techniques. However, the task remains challenging due to dynamic workloads and complex data distribution.

Automated configuration optimization tools (Section 3.4) lack effective handling of complex database settings and suffer from scalability issues.

Automated partitioning tools (Section 3.5) struggle with mixed workloads and overlook associated overhead, leaving the DBA to weigh cost-benefit considerations.

Chapter 4

Problem Statement

Contents

4.1 Open Problems 65

4.2 Scope 66

4.3 Hypothesis 66

4.4 Research Questions 68

4.5 Validation and Evaluation 68

4.6 Summary 69

This chapter formalizes the problem under investigation. First, Section 4.1 identifies the open problems, and Section 4.2 determines the scope of this dissertation. Then, Section 4.3 presents the hypothesis this dissertation aims to validate. Section 4.4 presents the research questions that will drive this dissertation. Section 4.5 describes the methodology used to validate the hypothesis. Finally, Section 4.6 summarizes the topics covered.

4.1 Open Problems

LRQ4 investigated the existing open problems in the field of automated strategies to optimize RDBMSs.

The Adaptive Indexing approach (Section 3.3.1) relies on immediate responses to incoming queries without leveraging advanced information or exploiting idle periods. In contemporary applications, user activity often exhibits burst sequences followed by prolonged periods of inactivity, causing this approach to be suboptimal. The lack of adaptation to irregular user patterns and failure to leverage available information results in reduced efficiency, posing a challenge in achieving optimal system performance [48, 89].

Most index selection tools (Section 3.3.2) use the optimizer estimates of cardinality and cost within their algorithms. Different studies [32, 120, 74] have highlighted the inherent issue of inaccuracies in these estimates. Recent advancements propose integrating ML techniques for estimations in response to this challenge [5, 36, 40]. Despite these efforts, the task remains challenging, primarily due to the dynamic nature of workloads and the distribution of data [32].

In the realm of automated configuration optimization tools (Section 3.4), challenges persist in effectively capturing the complex correlations among various database settings. Furthermore, most of these tools are tailored to address specific issues, causing them to be unscalable and impractical in diverse situations [129].

Existing automated partitioning tools (Section 3.5) face limitations in handling mixed and dynamic workloads, often specializing in either analytical or transactional scenarios. Additionally, the reviewed approaches overlook the associated partitioning overhead and maintenance considerations. Consequently, the responsibility falls on the DBA to assess whether the benefits derived from partitioning justify the associated costs and determine an opportune deployment time.

Currently, the optimization components of database systems work separately. There is a research effort on self-driving DBMSs that work as a complete system without human effort to optimize its performance [85].

4.2 Scope

Considering the limitations detailed in Section 4.1, this dissertation investigates how to optimize the performance of transactional large-scale database systems. Based on the literature findings and open problems described before, RL is successfully used in automated index and configuration strategies. Despite the shown success in studies such as the SWIRL [70], RL is scarcely being studied in automated partitioning strategies. We believe that RL could also apply to partitioning due to their similarities. Additionally, there is a gap in utilizing RL in transactional database systems since the only work that leverages RL for database partitioning only addresses analytical databases [60].

Thus, the scope of this work includes the design and implementation of a RL-based architecture to recommend partitions for transactional large-scale database systems, leveraging the literature's findings in this field and comparing it with current state-of-the-art strategies. It's important to note that the actual implementation of the database partitioning is not within the scope of this work.

4.3 Hypothesis

Based on the problems identified in Section 4.1 and to achieve the goals defined in Section 1.4, this work pretends to validate the following hypothesis:

> *Reinforcement Learning-based algorithms can optimize the performance of transactional large-scale database systems through the recommendation of database partitions, reducing queries runtime and systems' tail latency.*

We can deconstruct our hypothesis as follows:

Reinforcement Learning-based algorithms

Given the complexity of manually optimizing a relational database, automated database optimization has been a research topic over the last decades. Several strategies exist to fully or partly automate the optimization process of a RDBMS. RL has been a frequently adopted strategy, given its ability to adapt to dynamic and complex workloads. Despite success in automated index and database configurations optimization strategies, RL is scarcely used in automated partitioning. Therefore, leveraging the knowledge from existing RL-based solutions for index and configuration tuning, we intend to design a RL-based approach to optimize databases by recommending database partitions.

Optimize the performance of transactional large-scale database system

Optimizing a database's performance is essential for the functioning of any online system. This holds especially true for large-scale applications, where the magnitude of data storage and exchange in their databases requires careful optimization.

Furthermore, transactional systems handle critical real-time operations, such as financial transactions or online purchases. Sub-optimal database performance can result in latency issues, transaction failures, and, ultimately, a poor user experience.

The existing literature has been observed to focus solely on optimizing analytical databases when using RL methodologies [60]. Therefore, a gap emerges when considering the need for more relevant initiatives to optimize transactional databases through RL techniques.

Recommend database partitions strategies

The optimization of RDBMSs is conventionally achieved by correctly utilizing indexes and partitions and properly configuring database settings. Prior studies have demonstrated the efficacy of RL techniques in fine-tuning database indexes, showcasing their potential to boost database performance significantly.

In a noteworthy example, SWIRL [70] has successfully utilized RL to optimize database indexes. It is worth highlighting that SWIRL recognizes the prospect of extending their RL approach to address the optimization of database partitioning. However, despite the advances made in optimizing indexes, a gap exists in applying RL strategies for database partitioning. Therefore, our work aspires to bridge this gap by proposing and exploring RL-based strategies for database partitioning.

Reducing queries runtime and systems' tail latency.

The primary goal of our implementation is to reduce queries runtime since it is critical that clients get fast responses in transactional systems. Additionally, reducing the tail latency of the

system is also a vital performance metric to ensure a consistent and low-latency experience for the majority of the users and will help us identify possible edge cases.

4.4 Research Questions

The following research questions were formulated to guide this dissertation:

RQ1. What metrics are used to evaluate the performance of a transactional large-scale database system?

Identifying the right metrics used to evaluate the performance of transactional large-scale database systems is crucial when assessing the improvement of an optimization approach. These metrics will be a subset of the ones identified in Section 3.6 from **LRQ1**.

RQ2. What strategies exist that can improve the performance of a Relational Database Management System?

In the pursuit of optimizing RDBMSs performance, an in-depth exploration and understanding of the diverse array of available strategies is essential. This foundational knowledge will compose the cornerstone of this research, providing the necessary groundwork to delve into. Section 3.2 addressed this research question.

RQ3. What strategies exist to automate the optimization of the performance of a Relational Database Management System?

Understanding what strategies and tools exist to automate the optimization techniques identified in **RQ2** is another fundamental knowledge of this work. Sections 3.3, 3.4, and 3.5 covered this research question.

RQ4. Can we design a RL-based architecture that recommends database partitioning strategies that improve the performance of transactional large-scale database systems?

Leveraging the knowledge obtained from **RQ2** and **RQ3**, we intend to design a RL-based architecture that recommends database partitioning strategies that improve the overall performance of transactional large-scale database systems.

RQ5. How does such architecture compare with other state-of-the-art approaches? What are its trade-offs?

Comparing the architecture from **RQ4** with current state-of-the-art approaches is essential to evaluate it. We also intend to understand the trade-offs to assess its generalizability and applicability across different scenarios.

4.5 Validation and Evaluation

Zelkowitz and Wallace [124] defined the following four categories for experimental models for technology validation:

Scientific method: Scientists develop a theory to explain a phenomenon; they propose a hypothesis and then test alternative variations of the hypothesis. As they do so, they collect data to verify or refute the claims of the hypothesis [124].

Engineering method: Engineers develop and test a solution to a hypothesis. Based upon the results of the test, they improve the solution until it requires no further improvement [124].

Empirical method: A statistical method is proposed as a means to validate a given hypothesis. Unlike the scientific method, there may not be a formal model or theory describing the hypothesis. Data is collected to verify the hypothesis [124].

Analytical method: A formal theory is developed, and results derived from that theory can be compared with empirical observations [124].

The validation process of this dissertation uses both Engineering and Empirical methods.

In the initial phase, we will leverage knowledge from the literature review to develop a tool based on RL to recommend partitions. Building upon the success and well-documented nature of SWIRL [70], which utilizes RL for index recommendations, our development process will follow an engineering approach. The solution will be developed and refined iteratively based on the outcomes of evaluations.

In the second phase, we aim to empirically validate our hypothesis. Following a similar validation approach to SWIRL [70], we will employ well-established TPC [23] benchmarks, specifically TPC-E. This workload is widely utilized in transactional systems. Evaluation metrics will include the total runtime of queries (total latency) and tail latency. The former provides a comprehensive overview of solution quality, while the latter ensures that the majority of requests receive quick responses, a critical aspect in transactional systems. To assess the impact of implementing recommended partitions, we will evaluate the performance of a PostgreSQL database both before and after the implementation of partitions. Additional metrics, such as additional space requirements and the time taken for partitioning, will be gathered to comprehensively understand the trade-offs associated with the proposed solution.

4.6 Summary

Section 4.1 highlighted the open problems identified in the literature review, particularly the lack of RL strategies to automate database partitioning despite its success on other optimization strategies, such as indexing or database configurations.

Then, Section 4.2 identified this dissertation's scope as optimizing the performance of transactional large-scale database systems.

Section 4.3 presented the dissertation's hypothesis that we intend to validate:

> *(1) Reinforcement Learning-based algorithms can (2) optimize the performance of transactional large-scale database systems (3) through the recommendation of database partitions, (4) reducing queries runtime and systems' tail latency.*

Section 4.4 outlines the research questions that guided this work.

Section 4.5 described the validation and evaluation process. In the first phase, the engineering method will be followed to implement the solution. Then, the solution will be empirically evaluated in the second phase to validate our hypothesis.

Chapter 5

Reference Architecture and Implementation

Contents

5.1 Overview 71

5.2 Model 75

5.3 Reference Implementation 80

5.4 Summary 83

This chapter explains how we use RL for partition recommendation. Section 5.1 starts with an overview of our architecture. Then, Section 5.2 discusses how we model the partition selection problem to work with RL, addressing its complexity. Section 5.3 details our implementation of the reference architecture. Finally, Section 5.4 summarizes the reference architecture and its implementation.

5.1 Overview

The proposed reference architecture, illustrated in Figure 5.1, is divided into three phases: preprocessing (Section 5.1.1), where training and testing workloads are generated, partition candidates are identified, and the workload representation model is prepared; training, (Section 5.1.2), where the agent learns the value of different partitions for the given queries; and application, (Section 5.1.3), where the agent utilizes the trained model to select partitions for specific workloads. During the training and application phases, the agent iteratively selects partitions for the given workloads, forming the Markov decision process that the RL algorithm observes (Section 2.3).

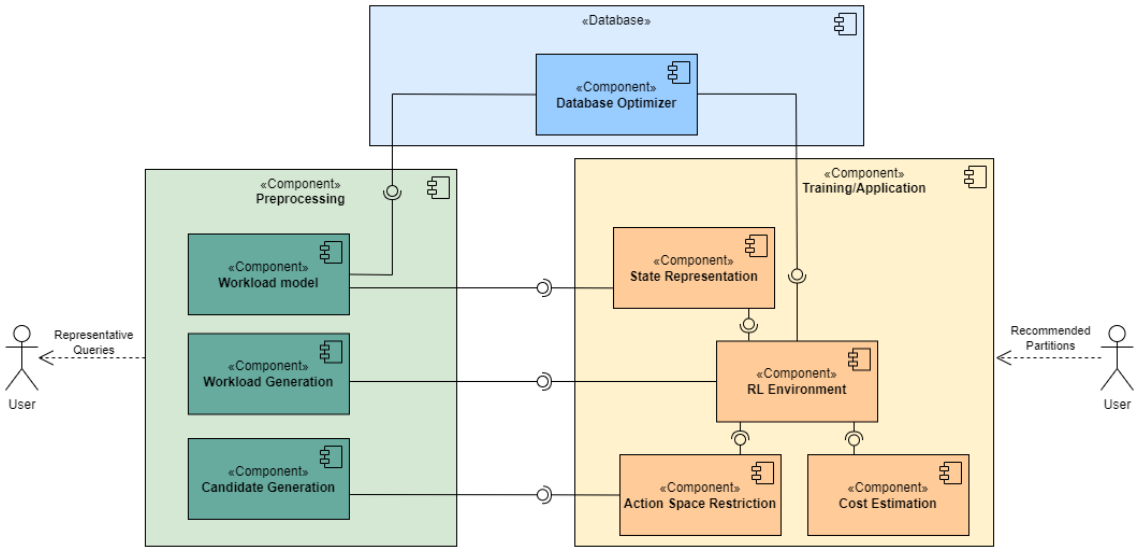


Figure 5.1: Reference architecture UML component diagram. It can be logically divided into two different components, the Preprocessing, and the Training/Application. Initially, the Preprocessing component depends on the representative queries chosen by the user. In the end, the user depends on the partitions the Training/Application recommends.

This architecture can be logically divided into the Preprocessing and the Training/Application components, depending on the use case. The agent's training and the model's application are grouped together since they will use the same sub-components. The user starts by selecting the representative query set to use in the Preprocessing module. The Workload Generation will create training and test sets of workloads based on the representative queries. The Workload Model will also use those queries to create a workload representation with the help of the Database Optimizer. The partitioning candidate keys will be generated with the Candidate Generation. The primary component of the Training/Application, the RL Environment, receives a workload and uses the Database Optimizer to get the initial query plans and costs. It also requires the Cost Estimation to estimate the costs of the current partitioning configuration for that workload. Then, using the available actions coming from the Action Space Restriction, and the state obtained from the State Representation, it selects an action and moves into the next iteration by getting the cost estimated by the Cost Estimation. When applying the model, the agent just greedily chooses the best action, while in training, it balances between exploring new states or exploiting already seen ones. The user will receive a set of recommended partitioning strategies when no valid actions are left.

This architecture was initially inspired by the SWIRL (Section 3.3.2.8) architecture, given its success in automated indexing and its prospects of extending it to other physical design problems. We adjusted and extended SWIRL's concepts to address the specific challenge of partitioning.

5.1.1 Preprocessing

The user starts by specifying a set of representative queries. Afterward, the column candidates for partitioning are generated according to the database schema and the set of representative queries. The fundamental impact of the set and the candidate generation process is detailed in Section 5.2.2.

Furthermore, random workloads are generated based on the representative queries and divided into training and test sets. Additionally, a subset of the representative queries can be exclusively reserved for testing, ensuring they are not included in training workloads. Both the size of the training and testing sets and the subset of queries to reserve for testing can be configured in our tool. This approach allows the rigorous evaluation of the agent's ability to generalize and handle unknown queries, thereby ensuring the validity of out-of-sample predictions.

Finally, the set of representative queries is also used to create the workload representation. Given that ML models typically require numerical features, the workload model generates these representations by converting information about the current workload into a numerical format suitable for ANN. This transformation is essential, enabling the model to manage previously unseen queries. Section 5.2.2 provides a detailed explanation of this process.

5.1.2 Training

During training, the agent learns to identify advantageous partitioning strategies for various situations by experimenting with multiple partition configurations across different workloads. This process enables the agent to implicitly learn complex relationships, such as the interactions between different partition candidates, the optimal number of partitions for a column, or the specific values to partition, without the need for explicit specification or modeling of these effects. Figure 5.2 illustrates the training process.

In each training episode, the environment selects a new workload. Then, for every query in the training workload, each query's costs and plans are obtained from the database optimizer based on the initial physical design. Subsequently, the current cost is estimated based on the initial cost, plans, and the current partition configuration, which starts as empty at the beginning of each training episode. Section 5.2.4 details comprehensively the cost evaluation methodology.

Following that, the agent's action space is constrained to include only valid actions for the current state of the environment. These restrictions are crucial for achieving rapid convergence and managing many possible partitioning actions without preemptively limiting the candidate set. Actions can be forbidden based on the current workload or previous actions, as Section 5.2.3 clarifies.

The RL state translates into numerical features the RL environment, including the workload representation, the current and initial costs, and the current active partitions. Section 5.2.1 provides a detailed demonstration of this procedure.

Afterward, the RL state and current reward are provided to the RL agent, selecting an action from the current set of valid actions. The different actions available to the agent are detailed in Section 5.2.3. After choosing an action, this iterative episode continues until no valid actions

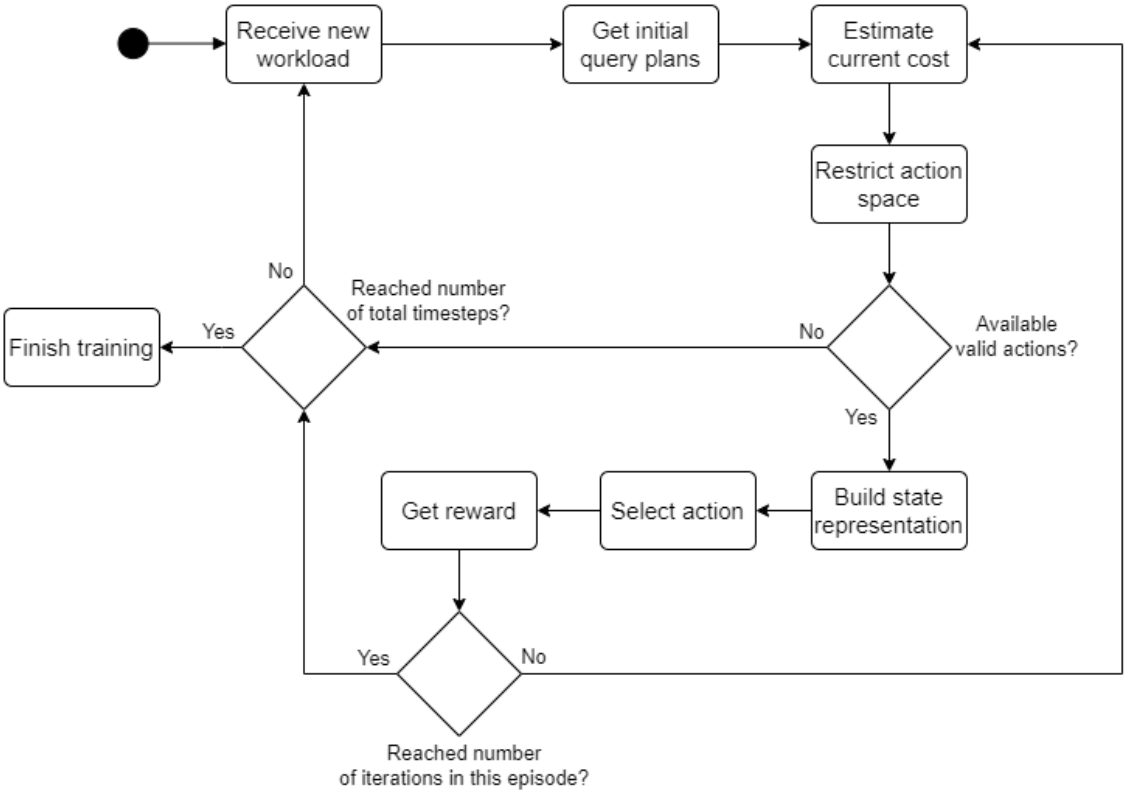


Figure 5.2: Reference architecture UML activity diagram. Each training episode has an iteration loop until no valid actions are left or when the maximum number of iterations is reached. Each training session starts an episode with a new workload until a maximum number of total timesteps is reached.

remain or a user-specified maximum number of iterations is reached. Upon completing a training episode, a new episode begins until a maximum number of timesteps is achieved or no significant improvements are observed. We recommended selecting at least a maximum of 20000 timesteps. Note that more extensive and complex workloads require a more significant number of timesteps.

5.1.3 Application

The actual workload is received to apply the learned model. The user can choose the same workload or one different from the one selected in training. Starting with an empty partitioning set, the agent iteratively evaluates the trained ANN to select the action with the highest reward until no valid actions remain. This procedure can be applied to an unknown workload, as the ANN can evaluate unseen states. When it reaches a state without valid actions, a set with partitioning strategies is recommended to the user. Notably, our model's application is almost immediate since only simple evaluations are required due to the trained ANN.

5.2 Model

This section outlines our modeling approach for RL. We begin by describing the representation of the environment in Section 5.2.1. Next, we explain the transformation process of the workload into a numeric representation in Section 5.2.2. Following that, we detail the available actions and the methods used to restrict the action space in Section 5.2.3. Finally, we present our query cost estimation methodology, part of the reward function, in Section 5.2.4.

5.2.1 State Representation

In RL, the representation of the state is crucial because it encapsulates all the relevant information about the environment required for the agent to make informed decisions. A well-designed state representation can significantly enhance the RL algorithm's learning efficiency and performance by providing a concise yet comprehensive portrayal of the problem space. In our work, we need to represent the state to effectively capture the characteristics of the partition selection problem, which involves a complex interplay between various workload features, different strategies for partitioning, and their respective cost. Figure 5.3 showcases the state representation for a simplified example workload.

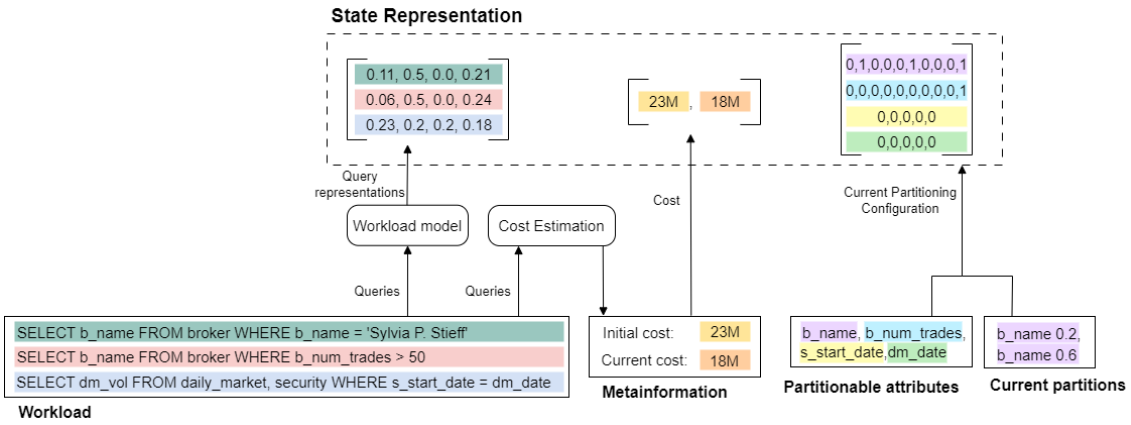


Figure 5.3: Construction of the state for a simplified example workload. The state is represented by three different aspects: the numerical query representations obtained with Workload Model, the initial and current cost estimated with Cost Estimation, and the current partitioning configuration calculated with the Partitionable attributes and the Current partitions.

The number of necessary features to accurately represent the environment depends on the number of queries in the workload, their complexity, and the number of partitionable columns. The state representation can be divided into three aspects: the workload representation, the cost information, and the current partitioning configuration.

The workload is represented as numerical vectors. For a workload with N queries and a specified representation width of R, it is represented N numerical vectors of length R. In the example of Figure 5.2.1, $N = 3$ and $R = 4$. Representing the queries' contents is crucial and a key differentiator from other approaches. Without it, the agent cannot learn about the structure of queries,

recognize similarities, and ultimately generalize to unseen queries. Moreover, although the structure of the ANN remains unchanged, a model trained with a workload of size N can still be used to recommend partitions for a workload of size M during the application phase. If M is smaller than N, padding can be applied to $M-N$ queries. Conversely, if M is larger than N, a representative subset of size N must be created. Methods like workload compression or query clustering are commonly and effectively used to reduce the total number of queries [18, 29, 78]. Choosing N is a critical task that depends on the complexity of the database and the problem. The user must balance the trade-off between minimizing the potential information loss and not excessively increasing the model's complexity. Section 5.2.2 explains the workload modeling and the transformation of queries into numerical vectors in more detail.

The cost information contains two scalar features representing the initial cost of executing the entire workload and the current one. The initial cost is estimated from the RDBMS optimizer. Every RDBMS can devise a query plan for a given query and estimate its execution cost. We use our own cost estimation model for the current workload cost based on the initial cost, the number of partitions accessed, and the ratio of data accessed in each partition. Section 5.2.4 explores in detail this cost estimation model.

The current partitioning configuration encodes the current partitioning strategy for every partitionable attribute. A binary vector represents this information as every possible action is either taken or not. Section 5.2.3 describes the possible partitioning strategies and their corresponding actions.

Before the state information is passed to the ANN, the vectors are concatenated, and their contents normalized. These techniques are used to improve the learning behavior of the network [61].

5.2.2 Workload Modeling and Query Representation

Unlike the only existing RL approach for database partitioning (Section 3.5.1), our primary objective is to represent queries in a manner that handles queries not included in the training sets. To achieve this capability, it is essential to contextualize unknown queries with known ones, thereby increasing the model's complexity. Thus, a detailed yet compact representation of the workload and its queries is required. This representation must balance being sufficiently detailed to distinguish between queries and compact enough to prevent feature explosion. Creating this representation must also be efficient to avoid significantly extending training durations.

Figure 5.4 illustrates the process of building the workload representation. Initially, representative execution plans are constructed from the initial queries using the RDBMS optimizer. We utilize execution plans rather than SQL strings to represent the queries, as execution plans contain more detailed information and can be more easily generalized, even for queries with different SQL strings.

Each operator in the execution plans is transformed into a text representation and stored in an operator dictionary, which assigns a unique ID to each distinct operator representation. These IDs are then used to construct a BOO model, providing a numerical representation of the query operators. Instead of directly using the BOO in the state representation, which would cause numerous

and very sparse features per query, we apply a dimensionality reduction. Based on the BOO representations of all representative plans, we create a Latent Semantic Indexing (LSI) [30] model to reduce the feature count. Selecting the number of features, or representation width R, involves a tradeoff: larger values for R increase the model's size and training times, while smaller values can lead to inadequate workload representations. Based on SWIRL experiments, $R = 50$ appears to be a reasonable choice, as it discards only 10% of the information and higher values for R do not significantly enhance the agent's performance [70]. We decided to use the same value since the implementation of this query representation module does not differ from SWIRL's. The difference is the way each implementation uses the information from that representation.

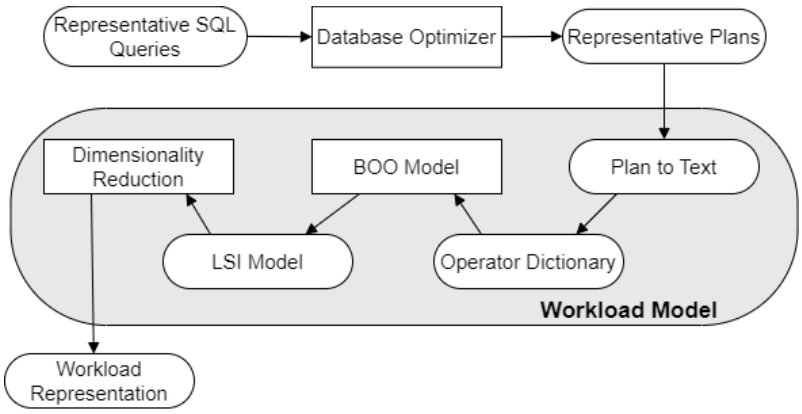


Figure 5.4: Workload modelling process. Rectangles refer to external tools, while ellipsis alludes to our implementation. The database optimizer retrieves their respective plans based on the representative SQL queries. Those plans will be converted into text and stored in an Operator Dictionary. That dictionary will be useful to create a numerical representation of the queries, using a BOO model. To reduce the number of features to an acceptable size, we use a LSI model. The final numerical workload representation is then obtained.

5.2.3 Actions

The action space determines how the agent can interact with the RL environment. Our model utilizes a discrete action space and allows different actions for every partitionable column depending on its type:

- If a column type is either *date* or *timestamp*, the agent has four different disjunctive actions on that column: **daily, weekly, monthly, or yearly partitioning**.
- If a column has other acceptable types (*numeric*, *integer*, *decimal*, *bigint*, *real*, *text*, *varchar*, *char*), the agent has nine different actions to choose from. Each action represents a split in the table that would create two partitions. These partitions are always **range partitions**, and each action has different split values from the 10th percentile to the 90th percentile. For example, if the agents choose the first and fourth action in a column, we would have a range partition for the first 10th percentile of data, another for the data between the 10th and the 40th percentile, and another for the remaining data.

- Independently of its type, every partitionable column has an action to **stop further partitioning on that column**. If the agent chooses this action first for a given column, there will be no partitioning.

We utilize invalid action masking [62] to significantly reduce the training duration by forbidding certain actions given the current partitioning configuration. The current set of valid actions is updated before every step. There are different reasons why an action is marked invalid:

- **Column is irrelevant for the workload:** Before the agent starts learning, we check if the column corresponding to that action occurs in the workload.

- **Action already taken:** After choosing an action, it is marked invalid, so it cannot be chosen again.

- **Table is already partitioned by another column:** If the agent already chose an action that partitions the table using a given column, all other actions corresponding to different columns in that table are marked invalid since a table can only be partitioned by a single column.

- **There is already a time-related partition in this column:** If the agent already chose to implement daily, weekly, monthly, or yearly partitioning, all remaining time-related partitioning strategies are also marked invalid.

Figure 5.5 illustrates, with the simplified example workload from Figure 5.3, the representation of the current partitioning configuration, the available actions, and the usage of invalid action masking. Initially, without any partitions, every non-date attribute has 10 possible actions: the 9 range partitions and an action that stops partitioning. On the other hand, date attributes have only 5 possible actions: daily, weekly, monthly, yearly partitioning, or none. After the agent chooses to partition the table broker with the attribute b_name on the 20th percentile, its own row of partitioning configuration (adding that partition) and available actions (removing that partition as available) are updated. It also updates any other attribute in the broker table to prevent another partitioning on that table. As a result, all actions for the b_num_trades become unavailable. Then, the agent chooses to partition again on the same column in the 60th percentile, and the corresponding row is updated. Finally, the agent chooses the last action for b_name, meaning it wants to stop further partitioning on that column. Consequently, all other available actions with that attribute become unavailable.

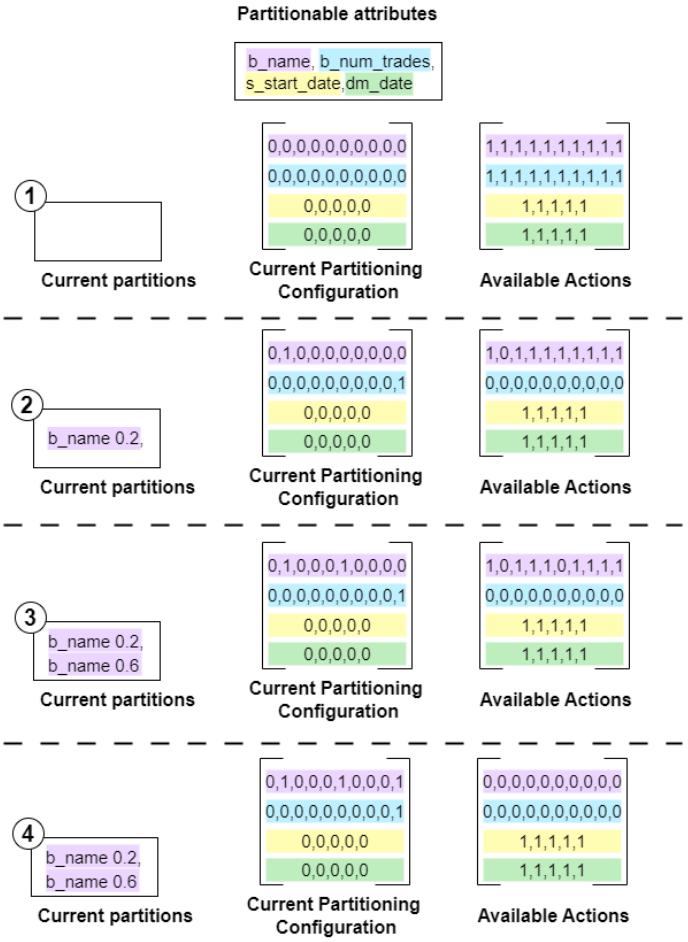


Figure 5.5: Action representation and invalid action masking for a simplified example workload. The current partitioning configuration is represented by vectors corresponding to the actions taken. Adding partitions makes the actions of other columns in that table unavailable with invalid action masking.

5.2.4 Rewards

The agent is rewarded for each chosen action that guides the learning process. Our reward function, r_t, considers the additional relative benefit of a partitioning configuration P in each step t:

$$r_t(P_t) = \frac{C(P_{t-1}) - C(P_t)}{C(\emptyset)}$$

Our cost model estimates, for a given query Q and partitioning configuration P_t, the query execution time:

$$C(P_t, Q) = \frac{C(\emptyset, Q) * N(P_t, Q)}{R(P_t, Q)},$$

where $C(\emptyset, Q)$ represents the initial query cost estimated by the optimizer without any partitioning, $N(P_t, Q)$ symbolizes the total number of accessed partitions with that query, and $R(P_t, Q)$

represents the percentage of data accessed in the partitions.

We decided to use the initial estimate by the optimizer in the reward function since it rewards the agent to look for opportunities to optimize the most time-consuming queries. Moreover, using the number of accessed partitions is also crucial, given that it is a common performance bottleneck in transactional systems [25, 87, 91], and it also prevents the creation of partitions that, in theory, just slightly improve the performance. Finally, it is essential to calculate and use the percentage of the data accessed by the partitions. Without it, the agent would be naturally biased towards larger partitions (e.g., accessing the last five days should have much worse performance than accessing the whole last year on a yearly partition).

In contrast to other approaches, punishing invalid actions with negative rewards is unnecessary due to the implementation of action masking.

5.3 Reference Implementation

The reference architecture proposed above can be instantiated in multiple contexts. For our reference implementation, we opted to do it against PostgreSQL due to its widespread availability and adoption [28], as well as its comprehensive documentation and robust support for database partitioning. This section details our open-source implementation of the reference architecture [44]. It can also be explored in the study's replication package [45]. Section 5.3.1 details the relational database system used and the retrieval of the query plans. Section 5.3.2 explores in detail the tools and libraries used to implement the RL model. Additional details with an execution example can be consulted in the Appendix B.

5.3.1 Database

This implementation was tested using the relational database system PostgreSQL 11. However, our approach is not tightly coupled to PostgreSQL, as it can be adapted to any other relational database system that supports retrieving query plans and query costs.

In PostgreSQL, the query planner, also known as the optimizer, analyzes the SQL query and determines the most efficient way to retrieve the requested data. This involves evaluating various execution strategies and selecting the one with the lowest estimated cost. The query plan includes details about the order of operations, the use of indexes, data retrieval methods (such as sequential scans or index scans), and join algorithms. Query costs in PostgreSQL are numerical estimates used by the query planner to determine the efficiency of different execution strategies. These costs are not actual time measurements but are relative values that help the planner compare various plans. The cost is divided into two main components: the startup and total costs. The startup cost represents the time to begin producing the first row of the result, while the total cost estimates the time to produce all rows of the result set. These costs consider factors such as disk I/O, CPU usage, and network latency.

Retrieving query plans and costs in PostgreSQL is straightforward using the EXPLAIN or EXPLAIN ANALYZE commands. The EXPLAIN command generates the query plan for a

given SQL query, displaying the estimated costs and the execution strategy chosen by the planner. To obtain more detailed information, including actual execution times and row counts, the EXPLAIN ANALYZE command can be used. This command runs the query and provides both the estimated and actual execution metrics [50].

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 7000;

                        QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=7001 width=244)
   Filter: (unique1 < 7000)
```

Listing 5.1: Example of a EXPLAIN command execution in PostgreSQL. The planner chooses to perform a sequential scan on the column tenk1, while applying a filter to each row it scans. The numbers quoted in the parentheses are, from left to right: the estimated start-up cost (time expended before the output phase can begin, e.g., time to do the sorting in a sort node), estimated total cost, estimated number of rows retrieved, and estimated width, in bytes, of rows outputted. Retrieved from the official documentation [50].

To execute such commands or any other SQL queries within our implementation code, we use Psycopg2, a Python library that allows for efficient and lightweight connections between Python applications and PostgreSQL databases [49].

5.3.2 Model

In our study, we utilized Python 3.7 and Stable Baselines 2 [59], which is built on TensorFlow [1] and PyTorch [84], to implement our chosen RL algorithm, Proximal Policy Optimization (PPO) [99]. Although Stable Baselines supports various RL algorithms, we specifically employed PPO in alignment with the experiments conducted by SWIRL. This choice was driven by our primary objective to evaluate the overall performance of RL in database partitioning rather than to identify the most effective RL algorithm for this task. Nevertheless, PPO offers the advantage of automatic learning rate adjustment, stabilizing the learning process by preventing abrupt changes in the agent's behavior and enhancing overall performance.

The model's hyperparameters are detailed in Table 5.1. Given our focus on applying a successful approach in automatic indexing to partitioning, the hyperparameters' values were initially based on those used by SWIRL and were subsequently adjusted through experimentation to suit our specific problem better:

Learning Rate. The learning rate controls how much the model's weights are adjusted with respect to the gradient of the loss function during training. In PPO, it determines the step size for updating the policy and value networks. A higher learning rate can lead to faster learning but can also cause instability, while a lower learning rate can lead to more stable but slower

learning. Given the complexity of our problem, we decided to have a slower learning rate than the default, $3 * 10^{-4}$.

Batch size. The number of training samples used to update the model's weights in one iteration. Larger batch sizes can provide more stable gradient estimates but require more memory and computation. We increased the batch size from 64 to 2048 to ensure the agent does not overfit to specific workloads or partitioning configurations and learns how to deal with various workloads with different partitioning configurations.

ANN layer structure. Refers to the architecture of the ANN used in PPO, particularly the policy and value networks. In this case, we decided to follow SWIRL's suggestions, and both networks have two layers of size 256.

Discount factor. The discount factor determines how much future rewards are considered when estimating a state's value. A discount factor of 0 means the agent only considers immediate rewards, while a discount factor close to 1 considers long-term rewards more heavily. We chose a discount factor of 0.2, which is lower than the usual RL problems since we do not mind that our agent behaves more greedily than expected to the better partitioning strategies, given that each table can only have 1 column partitioned.

Clip range. PPO introduces a clipped objective function to prevent large updates to the policy, which can destabilize training. The clip range sets the threshold for clipping the probability ratio between the new and old policies. If the ratio is outside the range, it gets clipped to the boundary values. This prevents the policy from changing too drastically in a single update. We kept the default value of 0.2 for the clip range.

Policy. In the context of PPO, the policy refers to the model that maps states to actions. An Multi-Layer Perceptron (MLP) policy is a neural network-based policy using a multi-layer perceptron architecture to map states to actions. We chose the MLP policy since it's common in PPO due to its flexibility and ability to learn complex, non-linear policies.

Hyperparameter	Value
Learning rate	$2.5 * 10^{-4}$
Batch size	2048
ANN layer structure	256 x 256
Discount factor	0.2
Clip range	0.2
Policy	MLP

Table 5.1: Hyperparameters used in our model. These values were determined initially by SWIRL configuration values and dynamically adjusted through our experiments.

The interaction between the agent and the environment is implemented using OpenAI's Gym interface [11], and the LSI model employed to represent the workload is created with Gensim [133].

We regularly check the model's performance on new workloads not included in the training and testing sets every few thousand steps to avoid overfitting. If the performance's moving average stops improving, we save the model's current state. Additionally, we use extensive caching of cost requests to shorten the training time.

5.4 Summary

This chapter described our reference architecture based on RL to recommend database partitioning strategies and detailed our proposed implementation.

Section 5.1 outlined an overview of the proposed architecture. It is divided into three phases: preprocessing, where workload for training and testing are generated, the partition keys are identified, and the workload representation model is constructed; training, where the agent actively tries different partition configurations for different workloads and learns with it; and application, where the agent recommended partitions for a given workload.

Then, Section 5.2 described in detail ourRL model approach. First, Section 5.2.1 presented our state representation, composed of a numerical query representation, the initial and current workload cost estimations, and the current partitioning configuration. Section 5.3.2 explained our workload modeling process, which converts the representative query plans into numerical representations. Then, Section 5.2.3 identified the different possible actions in our RL environment and our invalid action masking, used to reduce the training times. Finally, Section 5.2.4 presented our reward function and cost estimation model for a given query and partitioning configuration.

Section 5.3 presented the open-source implementation details of the proposed architecture. Section 5.3.1 details the chosen relational database system, PostgreSQL, and the retrieval of the query plan in that system. Section 5.3.2 identified the chosen RL algorithm, PPO, and the associated hyperparameters. All the libraries and packages used for the implementation were also specified.

Chapter 6

Verification and Validation

Contents

6.1 Methodology 85

6.2 Empirical Validation 86

6.3 Comparison to the State of the Art 91

6.4 Summary 93

This chapter verifies and validates our hypothesis, addressing **RQ5**. First, Section 6.1 describes the methodology utilized in the verification and validation process. Then, Section 6.2 presents the empirical validation of the proposed solution, assessing its quality with two different databases, examining its training duration and effort, and debates potential threats to the validity of our results. Section 6.3 performs a theoretical comparison against other state-of-the-art approaches. Finally, Section 6.4 summarizes the validation process.

6.1 Methodology

To verify our hypothesis, we perform an empirical validation and a theoretical comparison with other state-of-the-art approaches, addressing **RQ5**.

Section 6.2 empirically evaluates our solution's quality by measuring the query execution time with the recommended partitioning strategy. We use TPC-E to benchmark our solution with standard characteristics in Section 6.2.1, and Section 6.2.2 evaluates its practicality in an industry scenario utilizing part of Kevel's schema and data. Both use cases will be measured in an Amazon Elastic Compute Cloud (Amazon EC2) instance. Finally, Section 6.2.3 examines the duration and effort of our approach.

Then, Section 6.3 compares our work with other state-of-the-art approaches to automated partitioning, assessing its trade-offs.

6.2 Empirical Validation

In this section, we empirically evaluate the proposed implementation against the set of metrics identified in Section 3.6, such as total runtime, throughput, and tail latency (99th percentile).

All experiments were conducted in a m7g.large Amazon EC2 instance (2vCPU and 8GiB of RAM) with Python 3.7 and PostgreSQL 11.22. Each experiment was repeated ten times with random seeds to ensure stable results. All recommended partitions were implemented as range partitions in PostgreSQL, with the bounds either being chosen by the agent or predefined in the case of the time-related partitions.

6.2.1 TPC-E

One of the TPC's standardized benchmarks is TPC-E, specially designed to evaluate the performance of OLTP systems [24]. It simulates the complex activities of a brokerage firm, including trade transactions, account queries, and market research, with a diverse mix of read-and-write operations. TPC-E ensures objective, consistent, and scalable performance evaluations, assisting in assessing and selecting hardware and software configurations for optimal transaction processing capabilities.

In our benchmark evaluation, we utilized DBT-5 [33], a tool that facilitates the creation of standard databases with customizable parameters and subsequent performance evaluation through a series of transactions.

To build a TPC-E database, we downloaded the TPC-E Tools directly from the TPC [24]. The kit requires the TPC-E Tools to be built for the specific database management system to be tested. DBT-5 provides a script to apply patches and code to compile the TPC-E Tools. To build the TPC-E Tools for PostgreSQL, we unzip the TPC-E Tools zip file into an empty directory and run *dbt5-build-egen* against the resulting directory:

```
mkdir /tmp/egen
cd /tmp/egen
unzip /path/to/*-tpc-e-tool.zip
dbt5-build-egen /tmp/egen
```

Listing 6.1: Commands to build the TPC-E Tools for PostgreSQL.

After that, we build the database with our specification values and DBT-5 to measure the transactions:

```
dbt5-build --tpcetools=/tmp/egen pgsql -c5000 -f25 -w300
dbt5 run pgsql /tmp/results
```

Listing 6.2: Commands to build the database and measure the transactions.

Due to hardware constraints, we employed a smaller scale factor than the original benchmark. Consequently, our database size was 102 GiB, compared to the standard benchmark size of approximately 1.7 TiB. Table 6.1 presents the specification values used in our evaluation alongside those of the standard benchmark, including the number of customers present in the database, the scale factor corresponding to the number of clients per transaction, and the initial trade days, which represents the number of business days used to populate the trades.

Specification	**Original**	**Used**
Number of costumers	5000	5000
Scale Factor	500	25
Initial Trade Days	300	300

Table 6.1: TPC-E specification values used in the empirical validation compared to the original TPC-E specification. The number of customers is kept the same. Scale Factor, the number of customers per transaction, is reduced to 25. Initial Trade Days, the number of business days used to populate the trades, is also maintained.

Table 6.2 shows the performance results with the benchmark database before and after partitioning. The total workload runtime and throughput decreased by 11%, while the 99th percentile latency reduced by 7%. Figure 6.1 illustrates the seemingly uniform decrease in the query execution time distribution on both physical designs.

Performance metrics	**Base**	**Partitioned**	**Improvement**
Total runtime (ms)	218.12	193.89	+11.11%
Throughput (t/ms)	0.133	0.150	+11.11%
Tail latency (ms)	21.30	19.81	+7.00%

Table 6.2: TPC-E's empirical validation. The recommended partitioning configuration improved the performance across the different metrics.

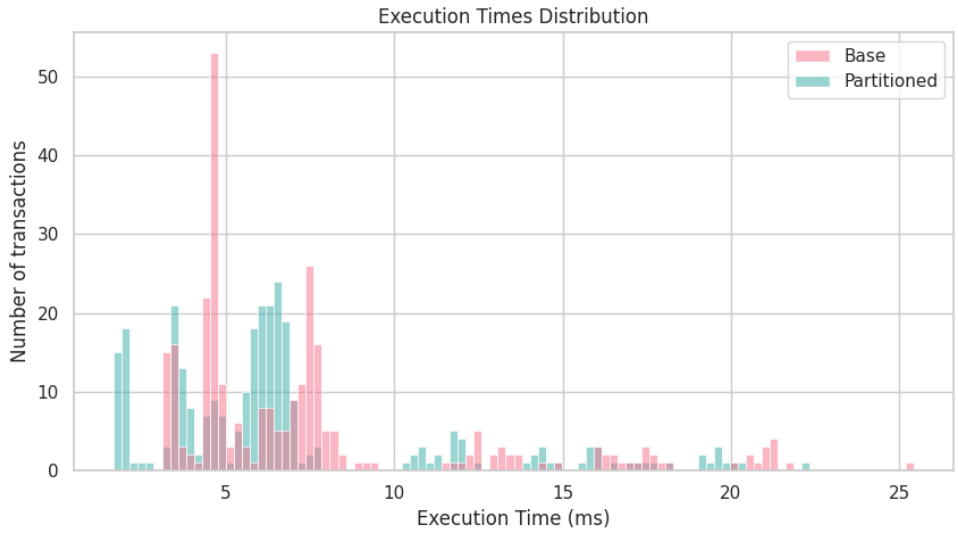


Figure 6.1: TPC-E's transactions execution time distribution. It suggests an improvement across the different transactions, including the most time-consuming transactions.

Even though the results were positive, they did not fully meet our initial expectations. One of the main challenges was the limitation of PostgreSQL regarding the partitioning key. PostgreSQL does not allow constraints (such as UNIQUE, PK, FK) or indexes on attributes other than the partitioning key. For instance, if a table is partitioned by attribute A, attribute B cannot be used as a primary key. The original TPC-E database already had a predefined set of indexes and constraints. Consequently, these constraints and indexes had to be adjusted to include the partitioning key. The modifications to accommodate PostgreSQL's constraints may not have been fully optimal for the TPC-E workload, leading to less-than-expected performance improvements.

6.2.2 Kevel

To further evaluate the practicality of our solution, we utilized part of Kevel's schema as a real-world scenario. Kevel is a company that provides flexible ad tech infrastructure APIs for retail media.

We collaborated with Kevel's employees to replicate their scenarios and use cases as accurately as possible. We were granted access to their schema and relevant table information, including table size, number of rows, and a data sample. All sensitive information in the sample was anonymized. Notably, since critical information was stored in JSON columns, which cannot be partitioned, anonymization did not result in any loss of data value. Using this data and Synthetic Data Vault (SDV), a Python library that creates synthetic data while preserving data distribution and relationships, we constructed and populated a database scaled to 1/40 of Kevel's original database.

Additionally, we obtained a set of query templates executed on these tables during a week using PostgreSQL's *pg_stat_statements* module [55]. We grouped similar queries and identified

the templates that consumed the most time. These query templates have, by definition, placeholders for any constants used in the query conditions. Consequently, we generated random uniform values for these placeholders to produce a representative set of queries.

Table 6.3 shows the obtained performance for the base database and the database with the recommended partitions. After applying the recommended partitioning strategies, the total runtime and throughput were improved by just 5%, while the tail latency increased by 34%. This is demonstrated in Figure 6.2, where most of the query execution times decreased, but some clear outliers got their performance worse with the partitioning strategies.

Performance metrics	**Base**	**Partitioned**	**Improvement**
Total runtime (ms)	75.84	72.24	+4.75%
Throughput (t/ms)	0.396	0.415	+4.75%
Tail latency (ms)	3.51	4.70	−33,90%

Table 6.3: Kevel's empirical validation. The recommended partitioning configuration improved the total runtime of the queries, but it deteriorated the database's tail latency. This indicates that even with the overall improvement in performance, the new physical design slowed some queries.

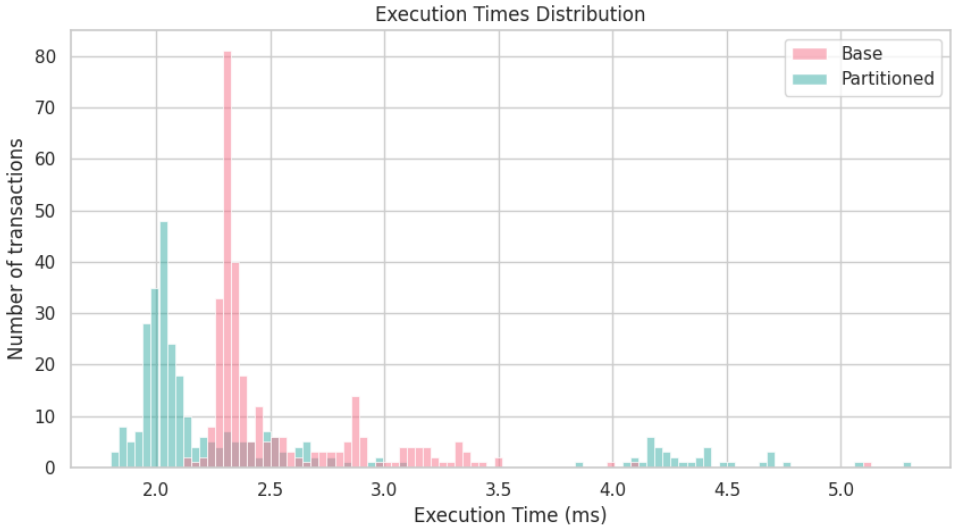


Figure 6.2: Kevel's transactions execution time distribution. It is clear that the partitioning configuration led to a decent improvement in several queries, but a group of queries got their performance decreased.

The results show that while the total execution time improved, alongside most of the queries' performance, the recommended partitions worsened the performance of some particular queries. Further analyzing those particular queries, we noticed that almost all were part of the same template and the template with the least time consumed. Given that our implementation is tailored to improve the most time-consuming queries, this can suggest that the agent preferred to decrease

the performance of this particular group to increase the performance of the most time-consuming groups.

This shows the importance of the representative workload, which can be harder in real-world scenarios due to the dynamic and diverse nature of the queries executed against the database. Workloads can vary significantly over time due to changing user behaviors, evolving application requirements, and fluctuating data volumes. Accurately capturing this variability requires continuous monitoring and analysis, which can be complex and resource-intensive. Without a representative workload, optimization strategies might be based on incomplete or biased data, leading to improvements in some areas at the expense of others. This highlights the necessity of detailed workload analysis to develop effective and balanced optimization solutions.

Moreover, it shows the importance of human intervention during the database optimization process, particularly in partitioning. Having a DBA involved is crucial since it brings a deep understanding of the database schema, the nature of the data, and the specific requirements of the database applications. DBAs can provide insights into the business context, ensuring that the optimization process aligns with the overall goals and priorities of the organization. They can also validate and fine-tune recommended partitioning strategies, ensuring that changes do not negatively impact other important aspects of the database performance.

6.2.3 Training Duration & Effort

Table 6.4 describes the training duration and effort in both scenarios. TPC-E's training process had many more available actions due to the larger amount of tables and columns compared to Kevel's database. As expected, larger workloads will require more training episodes. However, it is worth pointing out that it is not proportional to the number of possible actions due to the implementation of invalid action masking (Section 5.2.3). Moreover, the training duration is also heavily related to the number of queries in the workload (N) and the database size. Larger workloads will create more complex state representation (number of features) and use more time to estimate and retrieve the query plans. A bigger database will slow down the retrieval of the initial query plan.

Database	Database size	N	#Features	#Actions	#Episodes	Training Duration
TPC-E	106 Gib	55	882	550	34991	1.04h
Kevel	21 Gib	30	212	30	19961	0.27h

Table 6.4: Training duration and problem complexity metrics for the different databases. The size of the database and the workload influence the number of features and actions in the RL environment, the number of training episodes, and the training duration.

6.2.4 Threats to Validity

Given the empirical basis of the validation process, it is crucial to consider potential threats to the validity of its findings. Therefore, this section evaluates both the internal validity (i.e., the

likelihood that other factors influenced the conclusions) and external validity (i.e., the degree to which the results can be generalized to different contexts) of this study [38].

6.2.4.1 Internal Threats

The executed queries have very small response times.

Throughout our validation process, all queries are executed in under 25 milliseconds. Although each experiment is repeated 10 times, slight variations in the hardware can cause noticeable differences in query execution times.

The measurement of query execution times and the database were on the same Amazon EC2 instance.

The script responsible for executing and measuring the queries was run on the same Amazon EC2 instance as the PostgreSQL database. This setup may have led to competition for resources, potentially influencing the results.

6.2.4.2 External Threats

The Amazon EC2 instances use hardware virtualization.

Although Amazon Web Services (AWS) aims to reduce performance variability, this virtualization can cause minor performance differences between experiments, which might affect the ability to generalize the results to other execution environments.

The databases were scaled down.

Due to space constraints, the databases used during the validation were only a subset of the original dataset was used. While we expect our results to be even greater on higher scales, it is impossible to have that guarantee.

The hardware and PostgreSQL configuration were constant.

Given that our experiments were conducted on the same Amazon EC2 instance and the PostgreSQL configurations were kept constant, our study's findings are specific to the particular hardware and configuration used. Without testing on a range of hardware setups and different PostgreSQL configurations, we cannot be sure that the results will hold true in other environments.

6.3 Comparison to the State of the Art

Our literature review (Section 3.1) revealed a gap in the application of automated database partitioning for transactional databases using RL. Consequently, this section compares our study with other state-of-the-art approaches in automated database partitioning for transactional databases.

Table 6.5 provides a theoretical comparison of the different approaches. We define preprocessing time as the duration required to prepare the solution, including the agent training time in our study. On the other hand, application time refers to the time needed to use the solution once it is ready. Differentiating these two concepts is crucial as different use cases may have varying

requirements. An online application may prefer solutions with low application times to avoid delays, while an offline application might focus on the overall time consumed. Additionally, we assess the ability of each approach to generalize to unknown queries, an essential factor in our methodology. The evaluation methodologies are also compared, specifically whether the standard benchmark TPC-E is used. Since TPC has declared TPC-C obsolete, TPC-E remains the sole standard benchmark for transactional workloads, making it a fundamental part of the evaluation process.

This study has a higher preprocessing time than other approaches, given the nature of RL. The RL needs a good amount of training time to be able to recommend valuable partitioning strategies and to be able to generalize to unseen queries. On the other hand, compared to the other works, it has a really low application time due to the trained ANN, meaning that only simple evaluations are performed. We used TPC-E to evaluate our solution alongside an industry use case.

Schism [25] minimizes the number of distributed transactions while ensuring a balanced partitioning scheme by representing the database and the workload as a graph. Schism has a preprocessing phase, in which it computes the write and read sets of tuples accessed by each transactions. Once ready, getting the partitions has a duration in the order of the minutes. Moreover, for a new of set of unseen queries, the whole process needs to be repeated given. Schism extensively evaluated its solution with a couple of real-world scenarios and standard benchmarks, including TPC-E.

Horticulture [87] initiates its optimization process by creating a straightforward solution based on the most frequently accessed columns. Subsequently, an iterative approach begins where a new, incomplete solution is generated by intentionally relaxing the previous solution based on the number of distributed transactions. It has a quick preprocessing stage where it transforms the workload into graphs, but it can take several hours to reach a decent solution. It does not have the ability to handle a different representative workload. Horticulture also validates its approach with different benchmarks, including the standard TPC-E.

SWORD [91] captures a OLTP workload as a hypergraph and uses it to minimize the number of distributed transactions. It can be functionally divided into two key components: data partitioning and placement, and incremental repartitioning. Similarly to Schism, SWORD preprocesses the tuples accessed by each transaction to build the hypergraph. Their application time is also in the order of minutes if we consider the first component. The incremental repartitioning component allows the handling of unseen queries. However, it needs to monitor any workload changes online and maintain statistics on workload access patterns. SWORD only used the obsolete TPC-C to evaluate its solution.

	Our work	Schism [25]	Horticulture [87]	SWORD [91]
Preprocessing Time	High	Medium	Low	Medium
Application Time	Low	Medium	High	Medium
Handle unknown queries	Yes	No	No	Yes
Uses TPC-E to evaluate	Yes	Yes	Yes	No

Table 6.5: Comparison of different automated database partitioning approaches. The preprocessing time considers the preparation stages of the solutions, while the application time refers to the actual usage of the solution. Both times are described relatively to the other works. In all compared studies, there seems to be a trade-off between those times. Contrary to Schism and Horticulture, our implementation can completely generalize to unseen queries. SWORD handles unknown queries with the constraint of keeping it online to monitor the workload changes. All studies except SWORD use the most recent standard benchmark for transactional databases.

6.4 Summary

Section 6.1 summarizes the methodology for the verification and validation process. We performed an empirical validation using two different databases. TPC-E is used to benchmark the solution with standard characteristics of transactional large-scale database systems. We have also evaluated the practicality of the solution in an industry scenario with Kevel.

Section 6.2 describes the empirical validation results. Our implementation got a uniform improvement in performance across the different metrics with TPC-E (Section 6.2.1). However, Section 6.2.2 identified concerns with part of Kevel's data in the second scenario. The tail latency increased, meaning that part of the queries' performance decreased considerably since the overall query throughput also decreased. The results from the empirical validation show a decent potential, but they also show the importance of human intervention during this process, which can bring a deeper understanding of the database's schema and nature and the specific requirements of database applications. Then, Section 6.2.3 also demonstrates the training duration and effort for both cases. It exhibited the positive impact of the invalid action masking and the influence of the database's and workload's size in the training duration. Finally, Section 6.2.4 specifies potential threats to the validity of the findings. These include internal threats, such as the short query response time, and external threats, like using the Amazon EC2 hardware virtualization, the small scale of the databases, and the fact that the hardware and PostgreSQL configurations were not changed throughout the process.

Finally, Section 6.3 outlines the trade-offs between our proposed solution and other automated partitioning strategies for transactional databases. There is a trade-off between the preprocessing and application times across the different studies. However, it is noteworthy that our solution may offer more practicality since it does not require additional training or processing to handle unseen queries.

Chapter 7

Conclusions

Contents

7.1 **Summary** 95
7.2 **Research Questions** 96
7.3 **Hypothesis Revisited** 98
7.4 **Contributions** 98
7.5 **Challenges** 99
7.6 **Future Work** 100

This chapter presents the key conclusions and final remarks of this dissertation (*cf.* Section 7.1. Section 7.2 identifies the primary findings of each research question. Then, Section 7.3 revisits the hypothesis. Section 7.4 summarizes the main contributions of this work. Furthermore, Section 7.5 specifies the main difficulties that arose during this dissertation. Finally, Section 7.6 identifies opportunities for future work.

7.1 Summary

RDBMSs remain the dominant form of database management, employing various strategies such as indexes, partitions, and fine-tuned configurations to enhance performance. Historically, the task of optimizing these elements has been the sole responsibility of DBAs. However, the increasing complexity of large-scale database schemas has incentivized the development of tools and services to automate these design decisions.

Our literature review has identified several automated strategies for optimizing indexes, partitions, and configurations (*cf.* Chapter 3). Among these, RL has proven effective in automating index and configuration strategies. Despite these advancements, a significant gap persists in applying RL strategies to automated database partitioning, particularly for transactional databases, as current research primarily focuses on analytical databases.

In response to these findings, we have developed a reference architecture that leverages RL to recommend database partitioning strategies for large-scale transactional database systems (*cf.* Chapter 5). This architecture aims to reduce query runtime and minimize database tail latency.

We validated our approach using established benchmarks such as the TPC-E datasets, alongside an industry scenario using Kevel's data (*cf.* Section 6.2). Our work demonstrated positive results in the standard benchmark scenario, increasing overall performance. However, in the industry scenario, our solution exhibited some fragility. While overall runtime was reduced, certain queries experienced significant performance degradation, increasing the database's tail latency. These results emphasize the importance of human intervention and highlight the inherent complexities of this problem in real-world scenarios.

Finally, we compared our implementation with other state-of-the-art approaches to automated partitioning for transactional databases (*cf.* Section 6.3. Our study reveals clear trade-offs in preprocessing and application times across different solutions. Notably, our implementation balances a high training time with the capability to handle unseen queries and provide recommendations, offering a more practical solution in certain online scenarios.

7.2 Research Questions

This section reviews the research questions that directed this dissertation, demonstrating how each question was addressed and summarizing the main findings.

RQ1. What metrics are used to evaluate the performance of a transactional large-scale database system?

Based on our literature review (Section 3.6), the performance of transactional large-scale database systems can be assessed using metrics such as throughput, total runtime, tail latency, and resource usage. Both throughput and the total runtime are commonly used to compare with other optimization strategies. Additionally, the tail latency with high percentiles, such as 95th and 99th, provides insights into the distribution of query response times. It is often used in systems where a consistent and low-latency experience is critical for user satisfaction. Moreover, analyzing resource usage, including the levels of CPU, memory, or disk utilization, helps identify potential performance bottlenecks within the database.

RQ2. What strategies exist that can improve the performance of a Relational Database Management System?

As described in Setion 3.2, the performance of RDBMSs can be improved with the usage of indexes, configuration knobs, or partitions.

Indexes are special data structures that store a small portion of the data from the database tables in a way that makes search and retrieval operations more efficient. They can be used when frequent read operations need to retrieve data quickly.

Configuration knobs refer to the various settings and parameters that can be adjusted within the database management system to optimize its performance. These settings can include memory

allocation, cache sizes, and query optimization parameters. By fine-tuning these knobs, DBAs can better match the system's performance to the specific workload it is handling, leading to improved efficiency and faster query processing times.

Partitioning involves dividing a database into smaller, more manageable pieces. Each partition can be managed and accessed independently, leading to significant performance improvements, especially for large databases. Partitioning is useful when the database must handle large volumes of data with diverse access patterns, allowing for more efficient data retrieval and maintenance.

RQ3. What strategies exist to automate the optimization of the performance of a Relational Database Management System?

Considering the thirty-eight works surveyed in our literature review (Sections 3.3, 3.4, and 3.5), we have identified several different strategies to automate the performance optimization of RDBMSs.

Database Cracking [48] and Adaptive Merging [48] are two approaches to adaptive indexing, a strategy that builds indexes partially and incrementally during query processing. Another strategy for automated index management is to find the most suitable index configuration for a given workload using LP, heuristics, and, more recently, ML.

Automated configuration tuning approaches can be broadly divided into four groups. Heuristic tuning includes rule-based methods that mimic the rationale of manual tuning performed by a DBA, and search-based methods that employ hierarchical search strategies to identify optimal configurations. BO operates as a model-based iterative model, leveraging a surrogate model to approximate the objective function and continually refine it. RL revolves around an iterative process where an agent learns a policy by interacting with the environment. The agent suggests configuration values based on the current state and updates its policy according to received rewards. Finally, some more advanced approaches try to use DL to extract higher-level features from the given data.

We also identified works that automate the database partitioning process. Hilprecht *et al.* [60] presented a DRL approach for partitioning analytical databases. AdaptDB [77] proposed a novel self-tuning storage system that dynamically adapts to changes in the workload by employing incremental repartitioning. The remaining works use heuristic search-based tuning methods to find near-optimal partitioning configurations.

RQ4. Can we design a RL-based architecture that recommends database partitioning strategies that improve the performance of transactional large-scale database systems?

Based on the results in **RQ2** and **RQ3**, it became clear to us that, despite the shown success in automated index and configuration tuning, RL was not being used in automated database partitioning approaches, especially for transactional systems.

We described in detail our proposed RL-based architecture to recommend database partitioning strategies to transactional large-scale databases (Section 5.1 and 5.2), adapted from SWIRL, a workload-aware index tuning tool [70]. Moreover, implementation details were also described and explained throughout Section 5.3.

RQ5. How does such architecture compare with other state-of-the-art approaches? What are its trade-offs?

We empirically validate our proposed implementation (Section 6.2 using two different scenarios. The first uses TPC-E as a standard benchmark for the performance of transactional database systems. The second use case evaluates the practicality of our solution, validating it with data from a real-world company, Kevel. While the results were overall positive, there are concerns about the practicality of the solution given that a minority of the evaluated queries decreased considerably in performance in the second scenario despite the performance improvement of the remaining queries.

Furthermore, Section 6.3 compares our work and other automated database partitioning work for transactional databases. There is a clear trade-off between the preprocessing and application times between our solution and the remaining compared works. However, given that our high training time allows it to handle unknown queries, our approach thrives in scenarios where workloads are expected to change and quick partitioning recommendations are needed. Tools such as Schism [25] or Horticulture [87] cannot manage unseen queries without reapplying their solution, and SWORD [91] only has that ability if it monitors the workload changes online.

7.3 Hypothesis Revisited

This work aimed to validate the following hypothesis:

> *Reinforcement Learning-based algorithms can optimize the performance of transactional large-scale database systems through the recommendation of database partitions, reducing queries runtime and systems' tail latency.*

The proposed architecture and implementation (Chapter 5) demonstrated that RL can be used to recommend database partitions. Our empirical validation (Section 6.2) indicated that the recommended partitions increase the overall performance in both evaluated scenarios. However, it is important to note an increase in the tail latency in the industry scenario. Additionally, our comparison with other state-of-the-art approaches to automated partitioning in transactional databases (Setion 6.3) showed that RL strategies involve clear trade-offs and may be particularly advantageous in situations requiring a fast application time.

We conclude that our hypothesis is supported and that RL holds significant, yet overlooked, potential for automated database partitioning in both the literature and the industry.

7.4 Contributions

The work developed in this dissertation resulted in the following contributions to the state of the art of automated database optimization:

Literature Review on Automated Database Optimization strategies. A systematic literature review was performed to survey the state of the art of automated database optimization (Chapter 3).

Reference Architecture for Automated Database Partitioning. We designed an RL-based architecture to recommend database partitioning strategies for transactional large-scale database systems (Sections 5.1, and 5.2).

Open-source implementation. We implemented an open-source [44] prototype system that recommends database partitioning strategies in PostgreSQL (Section 5.3). To support the replication and extension of this work, we have created a replication package [45], containing the source code for our implementation.

Validation. To verify the quality and practicality of the proposed solution, we empirically validated the implementation using established benchmarks and metrics (Chapter 6). The replication package contains detailed instructions for reproducing the validation experiments [45].

7.5 Challenges

Throughout this dissertation, we encountered obstacles that delayed our progress. While we overcame some challenges, others remained unresolved and referenced as future work.

The first challenge occurred when adapting SWIRL's architecture to our problem. SWIRL uses HypoPG [96], a PostgreSQL extension that enables the creation of hypothetical indexes. They are useful for determining whether specific indexes can improve query performance, as they allow us to see if PostgreSQL will use these indexes without creating them. SWIRL uses this extension to obtain query plans and cost estimates, thereby assessing the potential benefit of the indexes for a given workload. HypoPG offers a beta version that supports hypothetical partitions. However, this version was not fully operational and lacked adequate documentation and testing. After contacting the author, we learned there were no immediate plans to release a stable version. As a result, we had to modify our initial adaptation plan and create our cost estimations using information about the partitions being used and the initial query plans and costs.

Another recurring challenge throughout the development and validation phases was the significant amount of time required to test small modifications to the implementation locally. Each time we made slight improvements to our tool or tested a new feature, it was necessary to fully train the RL agent, obtain recommendations, apply the partitions to a new database, and load it with the same data. Even with small-scale databases, implementing all recommended partitions and populating the database was time-consuming. Consequently, many minor changes or features were tested together due to these time constraints, which may have led to some being inadequately assessed.

Finally, memory constraints presented another obstacle during implementation. Since this work was developed on a personal laptop, testing the solution locally with large databases proved challenging. As a result, most of our tests during the development phase were conducted using small-scale databases. Additionally, in the validation phase, even though we used Amazon EC2 for validation, we still could not afford to create databases of the same size as the original ones

(both TPC-E and Kevel had databases on the order of terabytes). This limitation affected our ability to fully replicate the conditions and challenges of working with large-scale data, potentially impacting the accuracy and robustness of our findings.

7.6 Future Work

This work represents a noteworthy step towards establishing a RL-based solution for automated database partitioning. Nonetheless, further improvements and extensions can be obtained in the following areas:

Expand HypoPG to support hypothetical partitions. As described in the previous section, HypoPG currently does not offer a stable version for hypothetical partitions. Expanding it or making it stable could greatly impact the quality of the recommendations since the estimations based on the query plans with those hypothetical partitions might be more accurate than the existing cost estimations.

Automatic implementation of partitions. Currently, our solution gives the responsibility of implementing the recommendations to the DBA. It could be interesting to expand this solution to automatically or semi-automatically, with the approval of the DBA, create the partitions in the database and repopulate it accordingly.

Allow to revert partitions. Another interesting future work is to expand this solution to allow the agent to change or fully revert existing partitions in the database. This is essential in real-world scenarios, given that current partitions may become obsolete or no longer optimal. However, it is very challenging to derive the query execution plan without a given partition and understand its possible benefit.

Consider partitioning and repopulating costs. Our solution does not consider the costs of partitioning and repopulating the database with the new physical design when suggesting partitions. Adding such costs to our solution would avoid situations where a huge transformation in the database is suggested, but the tiny improvements are not worth it.

Expand to other partitioning strategies. Our architecture can only recommend time-related or range partitions with fixed intervals. Adding different partitioning strategies, such as hash or list partitioning, or improving the current implementation of range partitioning can be interesting future work.

Finally, it would also be valuable to expand the verification and validation through the following practices:

Expand empirical validation for different configurations. Our empirical validation was conducted in the same Amazon EC2 instance and with the same PostgreSQL configurations. Additional experiments on different hardware and with different PostgreSQL configurations would make the validation process more robust.

Validate on larger databases. Due to the challenges identified in the previous sections, our databases used in the validation were scaled down. Using resources to test this work on larger databases would help us better understand the impact of our solution in such large-scale scenarios.

Empirical comparison with other state-of-the-art approaches. While we conducted a theoretical comparison with other state-of-the-art approaches of automated partitioning for large-scale transactional databases, an empirical comparison with these methods would be highly beneficial. This represents an ambitious future goal, as it would require implementing these approaches due to the lack of available open-source implementations.

References

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[2] Evans J. Adams. Workload models for dbms performance evaluation. In *Proceedings of the 1985 ACM Thirteenth Annual Conference on Computer Science*, CSC '85, page 185–195, New York, NY, USA, 1985. Association for Computing Machinery.

[3] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollar, Arun Marathe, Vivek Narasayya, and Manoj Syamala. Database tuning advisor for microsoft sql server 2005. In *VLDB*. Very Large Data Bases Endowment Inc., August 2004.

[4] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. SIGMOD '04, page 359–370, New York, NY, USA, 2004. Association for Computing Machinery.

[5] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B. Zdonik. Learning-based query performance modeling and prediction. In *2012 IEEE 28th International Conference on Data Engineering*, pages 390–401, 2012.

[6] Mansaf Alam and Kashish Shakil. Cloud database management system architecture. *UACEE International Journal of Computer Science and its Applications([ISSN 2250 - 3765)*, 3:27–31, 01 2013.

[7] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, page 303–316, New York, NY, USA, 2014. Association for Computing Machinery.

[8] Charles W. Bachman. The programmer as navigator. *Commun. ACM*, 16(11):653–658, nov 1973.

[9] Charles W. Bachman. The origin of the integrated data store (ids): The first direct-access dbms. *IEEE Annals of the History of Computing*, 31(4):42–54, 2009.

[10] André B. Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the 2nd International Workshop on Software and Performance*, WOSP '00, page 195–203, New York, NY, USA, 2000. Association for Computing Machinery.

[11] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[12] Nicolas Bruno and Surajit Chaudhuri. Automatic physical database tuning: A relaxation-based approach. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, page 227–238, New York, NY, USA, 2005. Association for Computing Machinery.

[13] Baoqing Cai, Yu Liu, Ce Zhang, Guangyu Zhang, Ke Zhou, Li Liu, Chunhua Li, Bin Cheng, Jie Yang, and Jiashu Xing. Hunter: An online cloud database hybrid tuning system for personalized requirements. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 646–659, New York, NY, USA, 2022. Association for Computing Machinery.

[14] Stefano Cereda, Stefano Valladares, Paolo Cremonesi, and Stefano Doni. Cgptuner: A contextual gaussian process bandit approach for the automatic tuning of it configurations under varying workload conditions. *Proc. VLDB Endow.*, 14(8):1401–1413, apr 2021.

[15] Donald D. Chamberlin. Early history of sql. *IEEE Annals of the History of Computing*, 34(4):78–82, 2012.

[16] Donald D. Chamberlin and Raymond F. Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '74, page 249–264, New York, NY, USA, 1974. Association for Computing Machinery.

[17] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and olap technology. *SIGMOD Rec.*, 26(1):65–74, mar 1997.

[18] Surajit Chaudhuri, Ashish Kumar Gupta, and Vivek Narasayya. Compressing sql workloads. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, page 488–499, New York, NY, USA, 2002. Association for Computing Machinery.

[19] Surajit Chaudhuri and Vivek Narasayya. Autoadmin "what-if" index analysis utility. *SIGMOD Rec.*, 27(2):367–378, jun 1998.

[20] Surajit Chaudhuri and Vivek Narasayya. Anytime algorithm of database tuning advisor for microsoft sql server. June 2020.

[21] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, jun 1970.

[22] Thomas Connolly and Carolyn Begg. *Database systems. A practical approach to design, implementation and management*. 01 2010.

[23] Transaction Processing Performance Council. Tpc. https://www.tpc.org/, 2024.

[24] Transaction Processing Performance Council. Tpc-e. https://www.tpc.org/tpce/, 2024.

[25] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: A workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.*, 3(1–2):48–57, sep 2010.

[26] Debabrata Dash, Neoklis Polyzotis, and Anastasia Ailamaki. Cophy: A scalable, portable, and interactive index advisor for large workloads. *Proc. VLDB Endow.*, 4(6):362–372, mar 2011.

[27] DB-engines. Dbms popularity broken down by database model. https://db-engines.com/en/ranking_categories, 2023.

[28] DB-engines. Ranking of relational dbms. https://db-engines.com/en/ranking/relational+dbms, 2023.

[29] Shaleen Deep, Anja Gruenheid, Paraschos Koutris, Jeffrey Naughton, and Stratis Viglas. Comprehensive and efficient workload compression. *Proc. VLDB Endow.*, 14(3):418–430, nov 2020.

[30] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. *J. Am. Soc. Inf. Sci.*, 41:391–407, 1990.

[31] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[32] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. AI Meets AI: Leveraging Query Executions to Improve Index Recommendations. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1241–1258, Amsterdam Netherlands, June 2019. ACM.

[33] Rilson Oscar do Nascimento. Dbt-5: uma implementação de código aberto do tpc-e para avaliação de desempenho de sistemas de processamento transacional, 2008.

[34] Ruogu Du, Bohan Zhang, and Andy Pavlo. Query latency vs. throughput as a database tuning objective. https://ottertune.com/blog/query-latency-vs-thoughput, 2021.

[35] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning database configuration parameters with ituned. *Proc. VLDB Endow.*, 2:1246–1257, 2009.

[36] Jennie Duggan, Ugur Cetintemel, Olga Papaemmanouil, and Eli Upfal. Performance prediction for concurrent database workloads. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, page 337–348, New York, NY, USA, 2011. Association for Computing Machinery.

[37] Eurostat. Cloud computing - statistics on the use by enterprises. https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Cloud_computing_-_statistics_on_the_use_by_enterprises#Cloud_computing_as_a_service_model_for_meeting_enterprises.E2.80.99_ICT_needs, 2023.

[38] Robert Feldt and Ana Magazinius. Validity threats in empirical software engineering research - an initial survey. pages 374–379, 01 2010.

[39] Tim Fritz. 10 database monitoring metrics to track for optimal performance. https://blog.quest.com/10-database-monitoring-metrics-to-track-for-optimal-performance/, 2020.

[40] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L. Wiener, Armando Fox, Michael Jordan, and David Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *2009 IEEE 25th International Conference on Data Engineering*, pages 592–603, 2009.

[41] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems: the complete book*. Pearson Prentice Hall, Upper Saddle River, N.J, 2nd ed edition, 2009. OCLC: ocn191926993.

[42] Jia-Ke Ge, Yan-Feng Chai, and Yun-Peng Chai. Watuning: A workload-aware tuning system with attention-based deep reinforcement learning. *J. Comput. Sci. Technol.*, 36(4):741–761, jul 2021.

[43] D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison Wesley series in artificial intelligence. Addison-Wesley, 1989.

[44] Vasco Gomes. Towards automated relational database partitioning - open source implementation. https://github.com/vascoG/rl_partition_selection, 2024.

[45] Vasco Gomes. Towards automated relational database partitioning - replication package. https://doi.org/10.5281/zenodo.12607620, 2024.

[46] G. Graefe and W.J. McKenna. The volcano optimizer generator: extensibility and efficient search. In *Proceedings of IEEE 9th International Conference on Data Engineering*, pages 209–218, 1993.

[47] Goetz Graefe. The cascades framework for query optimization. *IEEE Data(base) Engineering Bulletin*, 18:19–29, 1995.

[48] Goetz Graefe and Harumi Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 371–381, Lausanne Switzerland, March 2010. ACM.

[49] Federico Di Gregorio. Psycopg2. https://pypi.org/project/psycopg2/, 2023.

[50] PostgreSQL Global Development Group. Chapter 14.1. using explain. https://www.postgresql.org/docs/16/using-explain.html, 2023.

[51] PostgreSQL Global Development Group. Chapter 20.1.2. parameter interaction via the configuration file. https://www.postgresql.org/docs/16/config-setting.html#CONFIG-SETTING-CONFIGURATION-FILE, 2023.

[52] PostgreSQL Global Development Group. Chapter 20.1.3. parameter interaction via sql. https://www.postgresql.org/docs/16/config-setting.html#CONFIG-SETTING-SQL, 2023.

[53] PostgreSQL Global Development Group. Chapter 20.4.1. memory. https://www.postgresql.org/docs/16/runtime-config-resource.html#RUNTIME-CONFIG-RESOURCE-MEMORY, 2023.

[54] PostgreSQL Global Development Group. Chapter 27. high availability, load balancing, and replication. https://www.postgresql.org/docs/16/high-availability.html, 2023.

[55] PostgreSQL Global Development Group. pg_stat_statements — track statistics of sql planning and execution. https://www.postgresql.org/docs/current/pgstatstatements.html, 2024.

[56] PostgreSQL Global Development Group. Postgresql: The world's most advanced open source relational database. https://www.postgresql.org/, 2024.

[57] Manoj Gupta and Dharmendra Badal. A study on indexes and index structures. 2:212–222, 02 2013.

[58] Connor Henderson, Spencer Bryson, Vincent Corvinelli, Parke Godfrey, Piotr Mierzejewski, Jaroslaw Szlichta, and Calisto Zuzarte. Blutune: Query-informed multi-stage ibm db2 tuning via ml. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, CIKM '22, page 3162–3171, New York, NY, USA, 2022. Association for Computing Machinery.

[59] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, , and Yuhuai W. Stable baselines. https://github.com/hill-a/stable-baselines/, 2022.

[60] Benjamin Hilprecht, Carsten Binnig, and Uwe Roehm. Learning a partitioning advisor with deep reinforcement learning, 2019.

[61] Lei Huang, Jie Qin, Yi Zhou, Fan Zhu, Li Liu, and Ling Shao. Normalization techniques in training dnns: Methodology, analysis and application. *CoRR*, abs/2009.12836, 2020.

[62] Shengyi Huang and Santiago Ontañón. A closer look at invalid action masking in policy gradient algorithms. *The International FLAIRS Conference Proceedings*, 35, May 2022.

[63] Shiyue Huang, Yanzhao Qin, Xinyi Zhang, Yaofeng Tu, Zhongliang Li, and Bin Cui. Survey on performance optimization for database systems. *Science China Information Sciences*, 66(2):121102, February 2023.

[64] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In Carlos A. Coello Coello, editor, *Learning and Intelligent Optimization*, pages 507–523, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[65] Matthias Jarke and Jurgen Koch. Query optimization in database systems. *ACM Comput. Surv.*, 16(2):111–152, jun 1984.

[66] Andrew Kane. Dexter: The automatic indexer for postgres. https://github.com/ankane/dexter, 2023.

[67] Konstantinos Kanellis, Cong Ding, Brian Kroth, Andreas Müller, Carlo Curino, and Shivaram Venkataraman. Llamatune: Sample-efficient dbms configuration tuning. *Proc. VLDB Endow.*, 15(11):2953–2965, jul 2022.

[68] George Karypis and Vipin Kumar. hMetis: a hypergraph partitioning package, 1998.

[69] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. Magic mirror in my hand, which is the best in the land?: an experimental evaluation of index selection algorithms. *Proceedings of the VLDB Endowment*, 13(12):2382–2395, August 2020.

[70] Jan Kossmann, Kastius, and Rainer Schlosser. SWIRL: Selection of Workload-aware Indexes using Reinforcement Learning. February 2022.

[71] Andreas Krause and Cheng Ong. Contextual gaussian process bandit optimization. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 24. Curran Associates, Inc., 2011.

[72] Quoc V. Le and Tomas Mikolov. Distributed representations of sentences and documents, 2014.

[73] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015.

[74] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, nov 2015.

[75] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proc. VLDB Endow.*, 12(12):2118–2130, aug 2019.

[76] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2019.

[77] Yi Lu, Anil Shanbhag, Alekh Jindal, and Samuel Madden. Adaptdb: Adaptive partitioning for distributed joins. *Proc. VLDB Endow.*, 10(5):589–600, jan 2017.

[78] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 631–645, New York, NY, USA, 2018. Association for Computing Machinery.

[79] Michael S. Mahoney. The history of computing in the history of technology. *Annals of the History of Computing*, 10(2):113–125, 1988.

[80] P M Mell and T Grance. The NIST definition of cloud computing. Technical Report NIST SP 800-145, National Institute of Standards and Technology, Gaithersburg, MD, 2011. Edition: 0.

[81] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning, 2016.

[82] Shamkant Navathe, Stefano Ceri, Gio Wiederhold, and Jinglie Dou. Vertical partitioning algorithms for database design. *ACM Trans. Database Syst.*, 9(4):680–710, dec 1984.

[83] Rimma Nehme and Nicolas Bruno. Automated partitioning design in parallel database systems. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, page 1137–1148, New York, NY, USA, 2011. Association for Computing Machinery.

[84] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[85] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C. Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. Self-driving database management systems. In *Conference on Innovative Data Systems Research*, 2017.

[86] Andrew Pavlo, Matthew Butrovich, Ananya Joshi, Lin Ma, Prashanth Menon, Dana Van Aken, Lisa Lee, and Ruslan Salakhutdinov. External vs. Internal: An Essay on Machine Learning Agents for Autonomous Database Management Systems. *IEEE Data Eng. Bull.*, 42:32–46, 2019.

[87] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, page 61–72, New York, NY, USA, 2012. Association for Computing Machinery.

[88] Andy Pavlo and Jignesh Patel. CMU 15-445/645 Database Systems (Fall 2023) :: Introduction to Distributed Databases :: Lecture 21. https://15445.courses.cs.cmu.edu/fall2023/slides/21-distributed.pdf, 2023. Lecture at Carnegie Mellon University.

[89] Eleni Petraki. Holistic indexing: offline, online and adaptive indexing in the same kernel. In *Proceedings of the on SIGMOD/PODS 2012 PhD Symposium*, pages 15–20, Scottsdale Arizona USA, May 2012. ACM.

[90] Ling Qian, Zhiguo Luo, Yujian Du, and Leitao Guo. Cloud computing: An overview. In *Cloud Computing: First International Conference, CloudCom 2009, Beijing, China, December 1-4, 2009. Proceedings 1*, pages 626–631. Springer, 2009.

[91] Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. Sword: Scalable workload-aware data placement for transactional workloads. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, page 430–441, New York, NY, USA, 2013. Association for Computing Machinery.

[92] Erhard Rahm. Parallel query processing in shared disk database systems. *SIGMOD Rec.*, 22(4):32–37, dec 1993.

[93] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy Lohman. Automating physical database design in a parallel database. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, page 558–569, New York, NY, USA, 2002. Association for Computing Machinery.

[94] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy Lohman. Automating physical database design in a parallel database. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, page 558–569, New York, NY, USA, 2002. Association for Computing Machinery.

[95] Hannah Ritchie, Edouard Mathieu, Max Roser, and Esteban Ortiz-Ospina. Internet. *Our World in Data*, 2023. https://ourworldindata.org/internet.

[96] Julien Rouhaud. Hypopg. https://hypopg.readthedocs.io/en/rel1_stable/, 2023.

[97] Rainer Schlosser, Jan Kossmann, and Martin Boissier. Efficient scalable multi-attribute index selection using recursive strategies. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1238–1249, 2019.

[98] Karl Schnaitter, Neoklis Polyzotis, and Lise Getoor. Index interactions in physical design tuning: Modeling, analysis, and applications. *Proc. VLDB Endow.*, 2(1):1234–1245, aug 2009.

[99] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[100] Amazon Web Services. What is database sharding? https://aws.amazon.com/what-is/database-sharding/, 2024.

[101] Anil Shanbhag, Alekh Jindal, Yi Lu, and Samuel Madden. Amoeba: A shape changing storage system for big data. *Proc. VLDB Endow.*, 9(13):1569–1572, sep 2016.

[102] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In Michael Maher and Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming — CP98*, pages 417–431, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[103] Abraham Silberschatz, Henry F Korth, and Shashank Sudarshan. Database system concepts. 2011.

[104] Statista. Size of the database management system (dbms) market worldwide from 2017 to 2021. https://www.statista.com/statistics/724611/worldwide-database-market/, 2022.

[105] Michael Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[106] Christof Strauch, Ultra-Large Scale Sites, and Walter Kriha. Nosql databases. *Lecture Notes, Stuttgart Media University*, 20(24):79, 2011.

[107] Vadim Struk. Decoding the complexities of large-scale web application development. https://relevant.software/blog/guide-large-scale-web-application-development/, 2023.

[108] David G. Sullivan, Margo I. Seltzer, and Avi Pfeffer. Using probabilistic reasoning to automate software tuning. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, page 404–405, New York, NY, USA, 2004. Association for Computing Machinery.

[109] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018.

[110] Jian Tan, Tieying Zhang, Feifei Li, Jie Chen, Qixing Zheng, Ping Zhang, Honglin Qiao, Yue Shi, Wei Cao, and Rui Zhang. Ibtune: Individualized buffer tuning for large-scale cloud databases. *Proc. VLDB Endow.*, 12(10):1221–1234, jun 2019.

[111] Immanuel Trummer. Db-bert: A database tuning tool that "reads the manual". In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 190–203, New York, NY, USA, 2022. Association for Computing Machinery.

[112] International Telecommunication Union. Measuring digital development - facts and figures 2023. https://www.itu.int/hub/publication/d-ind-ict_mdd-2023-1, 2024.

[113] Patrick Valduriez. *Shared-Memory Architecture*, pages 2638–2638. Springer US, Boston, MA, 2009.

[114] G. Valentin, M. Zuliani, D.C. Zilio, G. Lohman, and A. Skelley. Db2 advisor: an optimizer smart enough to recommend its own indexes. In *Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073)*, pages 101–110, 2000.

[115] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1009–1024, New York, NY, USA, 2017. Association for Computing Machinery.

[116] Oleksii Vasyliev. Pgtune. https://pgtune.leopard.in.ua/, 2023.

[117] Junxiong Wang, Immanuel Trummer, and Debabrota Basu. Udo: Universal database optimization using reinforcement learning. *Proc. VLDB Endow.*, 14(13):3402–3414, sep 2021.

[118] Kyu-Young Whang. Index selection in relational databases. In *International Conference on Foundations of Data Organization and Algorithms*, page 487–500, 1985.

[119] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings 20th IEEE International Conference on Distributed Computing Systems*, pages 464–474, 2000.

[120] Wentao Wu, Yun Chi, Shenghuo Zhu, Junichi Tatemura, Hakan Hacigümüs, and Jeffrey F. Naughton. Predicting query execution time: Are optimizer cost models really unusable? In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 1081–1092, 2013.

[121] Wentao Wu, Chi Wang, Tarique Siddiqui, Junxiong Wang, Vivek Narasayya, Surajit Chaudhuri, and Philip A. Bernstein. Budget-aware index tuning with reinforcement learning. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1528–1541, New York, NY, USA, 2022. Association for Computing Machinery.

[122] Jinhan Xin, Kai Hwang, and Zhibin Yu. Locat: Low-overhead online configuration auto-tuning of spark sql applications. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 674–684, New York, NY, USA, 2022. Association for Computing Machinery.

[123] Jerrold H. Zar. *Spearman Rank Correlation*. John Wiley Sons, Ltd, 2005.

[124] M.V. Zelkowitz and D.R. Wallace. Experimental models for validating technology. *Computer*, 31(5):23–31, 1998.

[125] Ji Zhang, Ke Zhou, Guoliang Li, Yu Liu, Ming Xie, Bin Cheng, and Jiashu Xing. Cdbtune+: An efficient deep reinforcement learning-based automatic cloud database tuning system. *The VLDB Journal*, 30(6):959–987, jun 2021.

[126] Xinyi Zhang, Zhuo Chang, Yang Li, Hong Wu, Jian Tan, Feifei Li, and Bin Cui. Facilitating database tuning with hyper-parameter optimization: A comprehensive experimental evaluation. *Proc. VLDB Endow.*, 15(9):1808–1821, may 2022.

[127] Xinyi Zhang, Hong Wu, Zhuo Chang, Shuowei Jin, Jian Tan, Feifei Li, Tieying Zhang, and Bin Cui. Restune: Resource oriented tuning boosted by meta-learning for cloud databases. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2102–2114, New York, NY, USA, 2021. Association for Computing Machinery.

[128] Xinyi Zhang, Hong Wu, Yang Li, Jian Tan, Feifei Li, and Bin Cui. Towards dynamic and safe configuration tuning for cloud databases. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 631–645, New York, NY, USA, 2022. Association for Computing Machinery.

[129] Xinyang Zhao, Xuanhe Zhou, and Guoliang Li. Automatic database knob tuning: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 35(12):12470–12490, 2023.

[130] Xuanhe Zhou, Luyang Liu, Wenbo Li, Lianyuan Jin, Shifu Li, Tianqing Wang, and Jianhua Feng. Autoindex: An incremental index management system for dynamic workloads. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 2196–2208, 2022.

[131] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. Bestconfig: Tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 338–350, New York, NY, USA, 2017. Association for Computing Machinery.

[132] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden. Db2 design advisor: Integrated automatic physical database design. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, page 1087–1097. VLDB Endowment, 2004.

[133] Radim Řehůřek and Petr Sojka. Software framework for topic modelling with large corpora. pages 45–50, 05 2010.

Appendix A

Literature review results

Tool	Optimization Strategy	Published	Data Source	Section
Schism [25]	Partition	2010	ACM Digital Library	3.5.2
Horticulture [87]	Partition	2012	ACM Digital Library	3.5.6
SWORD [91]	Partition	2013	ACM Digital Library	3.5.4
AdaptDB [77]	Partition	2017	ACM Digital Library	3.5.5
Hilprecht et al. [60]	Partition	2020	ACM Digital Library	3.5.1
DTA [83]	Partition	2011	ACM Digital Library	3.5.3
DB2 [94]	Partition	2002	ACM Digital Library	3.5.7
Database Cracking [48]	Index	2010	ACM Digital Library	3.3.1.1
Adaptive Merging [48]	Index	2010	ACM Digital Library	3.3.1.2
Drop [118]	Index	1993	IEEE Xplore	3.3.2.1
DTA [3]	Index	2005	ACM Digital Library	3.3.2.2
Extend [97]	Index	2019	IEEE Xplore	3.3.2.3
DB2 [114]	Index	2000	IEEE Xplore	3.3.2.4
Relaxation [12]	Index	2005	ACM Digital Library	3.3.2.5
CoPhy [26]	Index	2011	ACM Digital Library	3.3.2.6
Dexter [66]	Index	-	Additional References	3.3.2.7
SWIRL [70]	Index	2022	Additional references	3.3.2.8
Budget-aware [121]	Index	2022	ACM Digital Library	3.3.2.8
AutoIndex [130]	Index	2022	IEEE Xplore	3.3.2.8
PGTune [116]	Configurations	-	Additional References	3.4.1.1
Sullivan *et al.* [108]	Configurations	2004	ACM Digital Library	3.4.1.1
OpenTuner [7]	Configurations	2014	ACM Digital Library	3.4.1.2

Tool	Optimization Strategy	Published	Data Source	Section
BestConfig [131]	Configurations	2017	ACM Digital Library	3.4.1.2
iTuned [35]	Configurations	2009	ACM Digital Library	3.4.2.1
Ottertune [115]	Configurations	2017	ACM Digital Library	3.4.2.2
ResTune [127]	Configurations	2021	ACM Digital Library	3.4.2.3
CGPTuner [14]	Configurations	2021	ACM Digital Library	3.4.2.4
LlamaTune [67]	Configurations	2022	ACM Digital Library	3.4.2.5
OnlineTune [128]	Configurations	2022	ACM Digital Library	3.4.2.6
LOCAT [122]	Configurations	2022	ACM Digital Library	3.4.2.7
QTune [75]	Configurations	2019	ACM Digital Library	3.4.3.1
CDBTune [125]	Configurations	2021	ACM Digital Library	3.4.3.2
WATuning [42]	Configurations	2021	ACM Digital Library	3.4.3.3
UDO [117]	Configurations	2021	ACM Digital Library	3.4.3.4
DB-BERT [111]	Configurations	2022	ACM Digital Library	3.4.3.5
HUNTER [13]	Configurations	2022	ACM Digital Library	3.4.3.6
BLUTune [58]	Configurations	2022	ACM Digital Library	3.4.3.7
iBTune [110]	Configurations	2019	ACM Digital Library	3.4.4.1

Table A.1: Results of the literature review.

Appendix B

Execution example

This appendix exemplifies the training and application of our proposed implementation.

Listing B.1 shows an example of a JSON file used to configure our implementation. A description of the configuration options can be found in the replication package [45].

Additionally to the configuration file, your database needs to have the same name as the `workload[benchmark]` in the configuration file, and you need your queries in a file inside the `query_files` folder, as exemplified in the Listing B.2.

To replicate the experiments done in the validation phase of the dissertation, follow these steps:

1. Execute `python -m SWPRL <configuration_file.json>` to train the agent.
2. Execute `python -m SWPRL <configuration_file.json> load` to get the recommended partitions.
3. You can find the recommended partitions in the `stdout` or in the `final_partitions.json` file in the corresponding experiment folder.
4. Apply those partitions manually in your database.
5. You can analyze the performance of the queries with the new physical design with `SWPRL/scripts/evaluate_transactions.py` and build comparison charts with `SWPRL/scripts/build_charts.py`.

```
{
  "id": "Kevel example test",
  "description": "Example test",
  "gym_version": 2,
  "timesteps": 20000,
  "random_seed": 23,
  "parallel_environments": 1,
  "pickle_cost_estimation_caches": false,
  "action_manager": "PartitionActionManager",
  "observation_manager": "PartitionPlanEmbeddingObservationManager",
  "reward_calculator": "RelativeDifferenceToPreviousReward",
  "max_steps_per_episode": 200,
  "validation_frequency": 1000,
  "result_path": "final_experiment_results",
  "workload_embedder": {
    "type": "PlanEmbedderLSIBOW",
    "representation_size": 50
  },
  "rl_algorithm": {
    "algorithm": "PPO2",
    "stable_baselines_version": 2,
    "gamma": 0.2,
    "policy": "MlpPolicy",
    "model_architecture": {
      "net_arch": [
        {
          "vf": [256,256],
          "pi": [256,256]
        }
      ]
    },
    "args": {
      "n_steps": 64
    }
  },
  "workload": {
    "benchmark": "Kevel",
    "scale_factor": 1,
    "varying_frequencies": false,
    "training_instances": 5000,
    "validation_testing": {
      "number_of_workloads": 5,
      "unknown_query_probabilities": [0.1]
    },
    "similar_workloads": false,
    "excluded_query_classes": [],
    "unknown_queries": 0
  }
}
```

Listing B.1: Configuration file example.

```
insert into "order_storage"."latest_orders" ("order_id","timestamp","timeline_id")
 ↪  values ('000456789',1714517999946,'10170000') on conflict ("order_id") do
 ↪  update set
 ↪  "timestamp"=EXCLUDED."timestamp","timeline_id"=EXCLUDED."timeline_id"
insert into "order_storage"."latest_orders" ("order_id","timestamp","timeline_id")
 ↪  values ('004789012',1714517999946,'10170001') on conflict ("order_id") do
 ↪  update set
 ↪  "timestamp"=EXCLUDED."timestamp","timeline_id"=EXCLUDED."timeline_id"
insert into "order_storage"."latest_orders" ("order_id","timestamp","timeline_id")
 ↪  values ('014567890',1714517999946,'10170002') on conflict ("order_id") do
 ↪  update set
 ↪  "timestamp"=EXCLUDED."timestamp","timeline_id"=EXCLUDED."timeline_id"
SELECT x2."order" FROM (SELECT "order_id" AS x3, "timestamp" AS x4, "timeline_id"
 ↪  AS x5 FROM "order_storage"."latest_orders" WHERE "order_id" IN ('002456789',
 ↪  '004123456', '006789012', '008456789', '010123456', '012789012', '014456789',
 ↪  '016123456', '018789012', '020456789', '022123456', '024789012', '026456789'))
 ↪  x6, "order_storage"."orders_timeline" x2 WHERE x6.x5 = x2."id" ORDER BY x6.x4
 ↪  DESC, x6.x3 DESC;
SELECT x2."order" FROM (SELECT "order_id" AS x3, "timestamp" AS x4, "timeline_id"
 ↪  AS x5 FROM "order_storage"."latest_orders" WHERE "order_id" IN ('003123456',
 ↪  '005456789', '007123456', '009789012', '011456789', '013123456', '015789012',
 ↪  '017456789', '019123456', '021789012', '023456789', '025123456')) x6,
 ↪  "order_storage"."orders_timeline" x2 WHERE x6.x5 = x2."id" ORDER BY x6.x4
 ↪  DESC, x6.x3 DESC;

...
```

Listing B.2: Kevel's queries file sample.