Hasso-Plattner-Institut Für Digital Engineering
Enterprise Platform And Integration Concepts

Unsupervised Database Optimization: Efficient Index Selection & Data Dependency-Driven Query Optimization

Dissertation
zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)
in der Wissenschaftsdisziplin *Praktische Informatik*

eingereicht an der
Digital Engineering Fakultät
der Universität Potsdam

von

Jan Michael Koßmann

Betreuer:
Prof. Dr. h.c. mult. Hasso Plattner

Gutachter:
Prof. Dr. Wolfgang Lehner
Prof. Dr. Gunter Saake

Potsdam, 9. April 2022

Abstract

The amount of data stored in databases and the complexity of database workloads are ever-increasing. Database management systems (DBMSs) offer many configuration options, such as index creation or unique constraints, which must be adapted to the specific instance to efficiently process large volumes of data. Currently, such database optimization is complicated, manual work performed by highly skilled database administrators (DBAs). In cloud scenarios, manual database optimization even becomes infeasible: it exceeds the abilities of the best DBAs due to the enormous number of deployed DBMS instances (some providers maintain millions of instances), missing domain knowledge resulting from data privacy requirements, and the complexity of the configuration tasks.

Therefore, we investigate how to automate the configuration of DBMSs efficiently with the help of unsupervised database optimization. While there are numerous configuration options, in this thesis, we focus on automatic index selection and the use of data dependencies, such as functional dependencies, for query optimization. Both aspects have an extensive performance impact and complement each other by approaching unsupervised database optimization from different perspectives.

Our contributions are as follows: (1) we survey automated state-of-the-art index selection algorithms regarding various criteria, e.g., their support for index interaction. We contribute an extensible platform for evaluating the performance of such algorithms with industry-standard datasets and workloads. The platform is well-received by the community and has led to follow-up research. With our platform, we derive the strengths and weaknesses of the investigated algorithms. We conclude that existing solutions often have scalability issues and cannot quickly determine (near-)optimal solutions for large problem instances. (2) To overcome these limitations, we present two new algorithms. *Extend* determines (near-)optimal solutions with an iterative heuristic. It identifies the best index configurations for the evaluated benchmarks. Its selection runtimes are up to 10 times lower compared with other near-optimal approaches. *SWIRL* is based on reinforcement learning and delivers solutions instantly. These solutions perform within 3 % of the optimal ones. *Extend* and *SWIRL* are available as open-source implementations.

(3) Our index selection efforts are complemented by a mechanism that analyzes workloads to determine data dependencies for query optimization in an unsupervised fashion. We describe and classify 58 query optimization techniques based on functional, order, and inclusion dependencies as well as on unique column combinations. The unsupervised mechanism and three optimization techniques are implemented in our open-source research DBMS Hyrise. Our approach reduces the Join Order Benchmark's runtime by 26 % and accelerates some TPC-DS queries by up to 58 times.

Additionally, we have developed a cockpit for unsupervised database optimization that allows interactive experiments to build confidence in such automated techniques. In summary, our contributions improve the performance of DBMSs, support DBAs in their work, and enable them to contribute their time to other, less arduous tasks.

Acknowledgments

Writing a PhD thesis might not seem like a prime example of teamwork at first glance. While this observation is correct for preparing the manuscript, I cannot imagine how the preceding years at the *Enterprise Platform and Integration* Concepts (EPIC) research group could have been a better example of mutual support and teamwork.

The Hasso Plattner Institute provides an exceptional environment for studying and researching computer science. I sincerely thank Prof. Hasso Plattner for creating and funding this institution. Prof. Plattner, his chair representatives, Dr. Matthias Uflacker and Dr. Michael Perscheid, as well as my co-supervisor, Prof. Felix Naumann, supplied helpful, liberal guidance for this thesis over the last years.

Additionally, I thank SAP for funding my research. Specifically, I want to thank our (former) SAP colleagues, in particular, Dr. Alexander Böhm and Dr. Anisoara Nica, for the collaboration and feedback from an industry perspective.

Besides, the invaluable feedback of Martin Boissier, Markus Dreseler, Christopher Hagedorn, Stefan Halfpap, Raphael Jünemann, Lucky Katahanas, Daniel Lindner, Christoph Matthies, Dr. Michael Perscheid, Keven Richly, Dr. Daniel Ritter, Dr. Rainer Schlosser, Dr. Ralf Teusner, and Marcel Weisgut improved this thesis significantly. Thank you!

Also, I wish to thank all students who worked as student assistants on Hyrise, my other research projects, or participated in *Develop your own Database*. Furthermore, I thank the students that wrote their master's thesis or built great demonstrations in the bachelor's projects under my supervision.

Research is only half as productive and rewarding without collaboration. I am grateful for excellent collaborators. Query optimizers would still overlook the potential of data dependencies without Daniel Lindner, Prof. Felix Naumann, and Prof. Thorsten Papenbrock (and their patience). Index selection algorithms and the answer to "*Magic mirror in my hand, which is the best [index selection algorithm] in the land?*", would only be half as interesting without Martin Boissier, Stefan Halfpap, Marcel Jankrift, Alexander Kastius, and especially Dr. Rainer Schlosser. Also, Hyrise would not be such a well-architected, powerful research database system without Markus Dreseler.

I am glad to not only have found excellent collaborators but also friends in our research group. Thank you for making the PhD journey a pleasant time. Moreover, I also thank the *normal office* as well as Marilena Davis and Matthias Herzog for keeping everything together from an administrative and hardware perspective.

Finally, I want to thank my friends and my family, Christian, Gerd, Günter, and Karin, for their enduring and unconditional support as well as for sparking my interest in technology. Speaking of unconditional support, I am deeply grateful for my partner Lou and her repeated encouragement, motivation, and advice throughout my PhD journey.

Zusammenfassung

Sowohl die Menge der in Datenbanken gespeicherten Daten als auch die Komplexität der Datenbank-Workloads steigen stetig an. Datenbankmanagementsysteme bieten viele Konfigurationsmöglichkeiten, zum Beispiel das Anlegen von Indizes oder die Definition von Unique Constraints. Diese Konfigurationsmöglichkeiten müssen für die spezifische Datenbankinstanz angepasst werden, um effizient große Datenmengen verarbeiten zu können. Heutzutage wird die komplizierte Datenbankoptimierung manuell von hochqualifizierten Datenbankadministratoren vollzogen. In Cloud-Szenarien ist die manuelle Datenbankoptimierung undenkbar: Die enorme Anzahl der verwalteten Systeme (einige Anbieter verwalten Millionen von Instanzen), das fehlende Domänenwissen durch Datenschutzanforderungen und die Komplexität der Konfigurationsaufgaben übersteigen die Fähigkeiten der besten Datenbankadministratoren.

Aus diesen Gründen betrachten wir, wie die Konfiguration von Datenbanksystemen mit der Hilfe von Unsupervised Database Optimization effizient automatisiert werden kann. Während viele Konfigurationsmöglichkeiten existieren, konzentrieren wir uns auf die automatische Indexauswahl und die Nutzung von Datenabhängigkeiten, zum Beispiel Functional Dependencies, für die Anfrageoptimierung. Beide Aspekte haben großen Einfluss auf die Performanz und ergänzen sich gegenseitig, indem sie Unsupervised Database Optimization aus verschiedenen Perspektiven betrachten.

Wir leisten folgende Beiträge: (1) Wir untersuchen dem Stand der Technik entsprechende automatisierte Indexauswahlalgorithmen hinsichtlich verschiedener Kriterien, zum Beispiel bezüglich ihrer Berücksichtigung von Indexinteraktionen. Wir stellen eine erweiterbare Plattform zur Leistungsevaluierung solcher Algorithmen mit Industriestandarddatensätzen und -Workloads zur Verfügung. Diese Plattform wird von der Forschungsgemeinschaft aktiv verwendet und hat bereits zu weiteren Forschungsarbeiten geführt. Mit unserer Plattform leiten wir die Stärken und Schwächen der untersuchten Algorithmen ab. Wir kommen zu dem Schluss, dass bestehende Lösung häufig Skalierungsschwierigkeiten haben und nicht in der Lage sind, schnell (nahezu) optimale Lösungen für große Problemfälle zu ermitteln. (2) Um diese Einschränkungen zu bewältigen, stellen wir zwei neue Algorithmen vor. *Extend* ermittelt (nahezu) optimale Lösungen mit einer iterativen Heuristik. Das Verfahren identifiziert die besten Indexkonfigurationen für die evaluierten Benchmarks und seine Laufzeit ist bis zu 10-mal geringer als die Laufzeit anderer nahezu optimaler Ansätze. *SWIRL* basiert auf Reinforcement Learning und ermittelt Lösungen ohne Wartezeit. Diese Lösungen weichen maximal 3 % von den optimalen Lösungen ab. *Extend* und *SWIRL* sind verfügbar als Open-Source-Implementierungen.

(3) Ein Mechanismus, der mittels automatischer Workload-Analyse Datenabhängigkeiten für die Anfrageoptimierung bestimmt, ergänzt die vorigen Beiträge. Wir beschreiben und klassifizieren 58 Techniken, die auf Functional, Order und Inclusion Dependencies sowie Unique Column Combinations basieren. Der Analysemechanismus und drei Optimierungstechniken sind in unserem Open-Source-Forschungsdatenbanksystem Hyrise implementiert. Der Ansatz reduziert die Laufzeit des Join Order Benchmark um 26 % und erreicht eine bis zu 58-fache Beschleunigung einiger TPC-DS-Anfragen.

Darüber hinaus haben wir ein Cockpit für Unsupervised Database Optimization entwickelt. Dieses Cockpit ermöglicht interaktive Experimente, um Vertrauen in automatisierte Techniken zur Datenbankoptimierung zu schaffen. Zusammenfassend lässt sich festhalten, dass unsere Beiträge die Performanz von Datenbanksystemen verbessern, Datenbankadministratoren in ihrer Arbeit unterstützen und ihnen ermöglichen, ihre Zeit anderen, weniger mühsamen, Aufgaben zu widmen.

Contents

1 Introduction

Whenever DBMS vendors are unsure about the right default values, they *empower* the user by introducing new *knobs*.

Goetz Graefe — Principal Scientist at Google

Data is ubiquitous, with diverse applications for business processes and for our personal lives. Forecasts state that an unimaginable volume of data — 15 zettabytes (1.5×10^{22} bytes) — will be generated in 2022 alone [Sta21]. In many cases, data is organized in relational databases that are operated by database management systems (DBMSs). These DBMSs handle the efficient storing and retrieval of the contained data to enable further processing and the extraction of valuable information. Consequently, such databases are the foundation of most of today's business and end-user applications.

At the same time, database systems are complex software systems [Cod90] that offer a large number of configuration options. Well-chosen configurations can significantly affect performance and are essential for efficient workload processing [DTB09; Lu+19; Pav+19]. These configuration options include, inter alia, (i) *physical database design* decisions, e.g., index selection or partitioning criteria; (ii) the configuration of *knob settings*, e.g., the number of concurrently running threads or the buffer pool size; (iii) assigned *hardware resources*, e.g., the available main memory or the number of CPUs; (iv) *schema configurations*, e.g., declaring columns as (non-)nullable or defining keys.

According to Chaudhuri and Weikum [CW18], database administration is a "key factor" of the total cost of ownership (TCO) of database-centric information systems. Typically, highly skilled database administrators (DBAs) perform the tuning and administration of database systems manually. However, these tasks are time-consuming and demanding [HSH07; Pav21] because close-to-optimal configurations depend on multiple factors that influence each other, such as the processed workload, the underlying hardware, and the DBMS version [Ake+17; Che+08]. Furthermore, the number of configuration options is steadily increasing [Ake+17; KS20]. Widely deployed DBMSs have hundreds of knobs [Ake+21] and numerous configuration options for physical database design [Zil+04]. To be more cost-efficient and to identify optimal configurations, we argue that database systems should utilize their internal knowledge about processed workloads and stored data in order to handle their configuration in an unsupervised or autonomous fashion.

Cloud DBMS deployments make autonomous or self-driving database systems [Pav+17] even more desirable [Aba+19; Zha+21] because cloud providers must manage and optimize thousands of database systems. Two considerations shift the responsibility for effectively configuring DBMSs to the providers. First, the cloud vendor's aim of achieving resource efficiency is motivated by the intent to increase scalability and handle more customers. Second, the "cloud's promise of reducing the TCO" [Das+19, p. 667] and improved flexibility for customers is a major driver. The manual tuning of these systems by DBAs is impractical[1] because of the aforementioned complexity of the configuration tasks, the missing domain knowledge resulting from the adherence to data privacy and compliance requirements [Das+19], and the enormous number of systems. In fact, more than 70 % of 312 interviewed DBAs manage only less than 100 (most of them significantly fewer) database instances [Que20]. Hence, managing thousands or millions of instances would require an excessive number of DBAs.

Such considerations regarding cloud DBMS deployments are essential, given that more than 75 % of all databases are projected to be deployed in cloud scenarios in 2022 [Gar20]. Additionally, more than half of the interviewed organizations currently manage enterprise data in the cloud [Que20]. The referenced DBA survey [Que20] even mentions *incorporating cloud technologies* and *automating more tasks associated with database management* as the top two data management infrastructure challenges.

Independent of cloud challenges, workloads of a DBMS may change unpredictably and vary between customers [Che+08; Ma+18; Sch+13], potentially requiring frequent reconfigurations. Events such as a global pandemic, Black Friday, or blizzards might cause workload changes. However, current automated approaches are not sufficient to handle such situations because they are often too slow, e.g., due to their enumerative modus operandi, or ineffective, e.g., due to simplifying heuristic assumptions, in determining beneficial solutions. Even marginal performance deficits can have substantial cost impacts considering the large number of systems in cloud environments [Das+19].

Advancements in many research areas are necessary for building autonomous database systems [KS20]. For example, the capabilities of workload forecasting [Ma+18], efficient [HBR20; KKS22a] and robust [SH20; SH21] tuning algorithms, or accurate and adaptive cost models [MP19] present interesting challenges.

[1]Note that in our and other researchers' opinions [Pav+19], autonomous DBMSs should not replace human DBAs but rather support them in their work and enable them to contribute more time to less demanding tasks.

As it is impractical to solve all challenges simultaneously, we focus on two aspects in this thesis to advance further toward fully autonomous DBMSs. First, we consider the *storage perspective* by investigating existing approaches for index selection, a typical physical database design challenge. Based on our findings, we develop new techniques that identify better index configurations faster. Second, we consider the *query perspective* and examine how information inherent to data — so-called data dependencies — can improve query optimization to obtain query results more rapidly. Moreover, we investigate how database systems can discover and validate beneficial data dependencies in a workload-driven, unsupervised fashion. We chose to focus on these two aspects because of their significant performance impact and their general applicability to relational DBMSs. Despite their differences, both aspects operate in a workload-driven fashion.

In the following, this introductory chapter first gives an overview of recent work in the different areas of autonomous database systems focusing on unsupervised database optimization in Section 1.1. Subsequently, we discuss the motivation for exploring the two aforementioned topics — *efficient index selection* and *data dependency-driven query optimization* — in Sections 1.2 and 1.3 and contextualize them in the research field. Finally, we discuss our research questions and contributions in Section 1.4 before we present the structure of this thesis in Section 1.5.

1.1. Building Blocks of Autonomous Database Systems

Letting DBMSs handle their configuration and optimization autonomously is not a novel idea: the first approaches to automated index selection date back to the 1970s [HC76; LL71]. However, as discussed above, autonomous database systems have attracted research interest recently, and "the field has made several important steps towards this goal in recent years" [Pav+19, p. 41]. The shift to cloud environments and its associated challenges are one reason for the increased interest in autonomous DBMSs [Aba+19]. In addition, advancements in machine learning (ML) support these developments because ML techniques are often employed in autonomous systems to achieve the desired goals. Increased interest in autonomous DBMSs can also be observed in the industry, as vendors of cloud database products provide various automatic tuning capabilities, e.g., *Microsoft Azure's automatic tuning* [Mic22a], *Oracle's autonomous database* [Ora20], *SAP HANA Cockpit's recommendations* [SAP21], and *Snowflake's automatic query optimization* [Sno16]. Furthermore, there are standalone tools for tuning existing database systems, for example, *OtterTune* [Ott; Zha+18], *PGTune* [Vas], or *pganalyze* [Dub].

As well as the advancements in industry, unsupervised database optimization is an active research field. To put the topics that we have researched into context, we provide an overview and divide the broader field of autonomous database systems into research focus areas in Figure 1.1. We also mention recent exemplary publications that appeared after Andrew Pavlo's visionary paper on *Self-Driving Database Management Systems* [Pav+17] for all focus areas. Of course, the references included in the figure cannot cover the field in its entirety, but serve to illustrate the recent, high levels of activity in the field.

The figure presents four focus areas. Three of these areas represent the necessary building blocks for autonomous DBMSs: (i) *System & Integration* (bottom), (ii) *Unsupervised Database Optimization* (middle), (iii) *Impact Forecasting* (right). Research in all of these areas contributes to the aim of achieving autonomous database systems. The last area, (iv) *Learned Components* (left), is another building block that complements such systems, even though it is not necessary for enabling them. In the following subsections, we briefly discuss the scope and covered functionality of the focus areas.

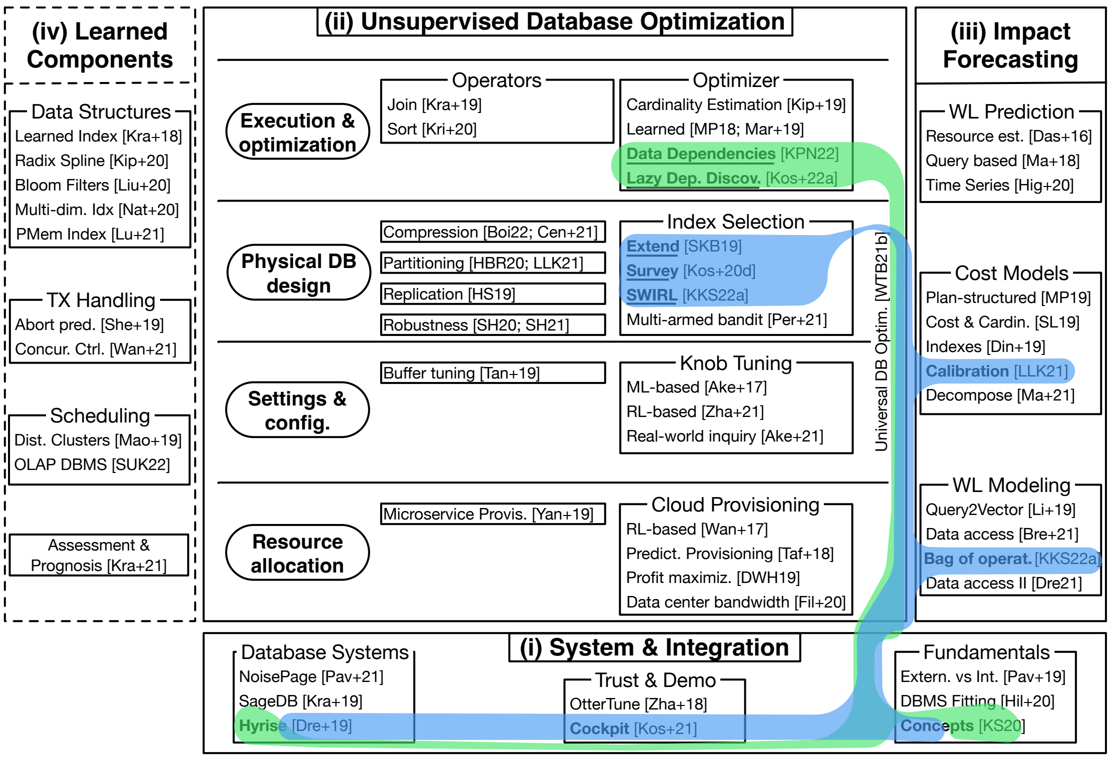


Figure 1.1.: Research focus areas, contributions, and recent publications in the field of autonomous database systems. Our main contributions are highlighted with **bold, underlined type**; supporting contributions are highlighted with **bold type** only. Publications colored in blue are related to Part I and green-colored ones to Part II of this thesis. *WL* is short for workload, *TX* for transaction, and *Demo* for demonstration.

1.1.1. System and Integration

This area investigates the challenges and concepts of autonomous database systems on the system level. Such investigations are important, because the way in which the optimization techniques (focus area (ii) *Unsupervised Database Optimization*) interact with the underlying system can impact the runtime and implementation complexity and also limit the capabilities of these techniques [Dre+19].

Database Systems. For instance, Pavlo et al. [Pav+17] argue that the architecture of new DBMSs must consider autonomous capabilities. *NoisePage* by Pavlo et al. [Pav+21] is a completely rewritten DBMS that takes such autonomous capabilities into account. However, developing an entirely new DBMS is complex and cost-intensive. Therefore, established vendors, e.g., Oracle [Ora20], are more likely to *adapt* their existing systems for autonomous operation. This approach raises questions about the integration of such techniques into existing systems. Integration alternatives are, for example, considered with the Hyrise plugin concept [Dre+19] and in a discussion of the advantages and drawbacks of internal and external DBMS tuning agents [Pav+19].

Fundamentals. Further work discusses fundamental ideas and necessary components for autonomous database systems [KS20]. For instance, Hilprecht et al. [Hil+20] propose *DBMS fitting* to overcome the explicability issues of deep learning approaches, which often appear as a black-box [Rud19]. Such deep learning approaches are often proposed for autonomous DBMSs [Ma+18; Mao+19; Mar+19; MP19; Zha+19; Zha+21] and learn the entire behavior of a database component or functionality from scratch. DBMS fitting presents a fundamentally different approach: In a first step, the *general behavior* of a database component, e.g., the outline of an operator's cost estimation function, is required to be manually implemented. Subsequently, certain parts of this implementation are *fitted* to the concrete scenario with learned parameters during execution. Thereby, existing knowledge can be incorporated and the explicability issues of deep learning approaches can be mitigated. Additionally, the utilized simple white-box models outperformed deep learning approaches in terms of accuracy, generalizability, and data efficiency in the evaluations.

Trust & Demonstration. Moreover, trust has been identified to be a significant issue for autonomous systems [Sto08]. Deploying autonomous DBMSs means transferring the responsibility for efficient workload processing and, in the end, smooth business operations to an autonomous system. Demonstrations [HS20; Kos+21; Zha+18] of well-performing autonomous DBMSs and case studies can increase trust in such systems.

1.1.2. Unsupervised Database Optimization

This area comprises the actual optimization techniques that maximize the DBMS's performance according to a metric, e.g., throughput or resource utilization. These optimization tasks are traditionally performed by DBAs. Like Pavlo et al. [Pav+19], we further divide this focus area into more specific subareas.

Resource Allocation. The work on *resource allocation* examines the question of how to assign hardware resources to systems so that costs are minimized, while performance is guaranteed. Examples include the allocation of network bandwidth and hardware resources, such as memory or processing units. Such techniques are particularly relevant in cloud environments, where resource demands fluctuate [Das+16] and cost efficiency is paramount.

Settings and Configuration. Techniques in this subarea refer to the process of adjusting the DBMS's knobs or settings to increase performance. It is not uncommon for DBMSs to offer hundreds of dependent configuration options [Ake+17]. For example, well-chosen settings for the number of concurrently running threads or the buffer pool size have a large performance impact [Tan+19].

Physical Database Design. Furthermore, we consider *physical database design*, i.e., techniques affecting the database's physical storage layout. Such techniques include, e.g., the selection of beneficial (secondary) indexes, partitioning criteria, data compression, and replication strategies. Again, a reasonable selection of these aspects has a large performance impact. We will substantiate this claim later when we conduct detailed evaluations for index selection in Chapter 4. While automated physical database design has been researched for half a century [LL71], more efficient approaches that consider additional aspects, such as robustness [SH20; SH21], are necessary to fulfill current requirements.

Execution and Optimization. Lastly, there are various autonomous techniques that directly affect the *execution and optimization* of queries. For instance, data dependencies and constraints can be utilized to improve query plans for more efficient execution [KPN22]. In addition, all different stages of query optimization and execution are targeted. Cardinality estimation, typically based on statistics like histograms or sampling, can be replaced with neural network-based cardinality estimation [Kip+20]. Even the heavily engineered optimizer itself can be substituted with learned query optimizers [Mar+19].

1.1.3. Impact Forecasting

The impact of the available configuration options must be evaluated and compared during the optimization processes in order to eventually identify the best option.

Cost Models. Usually, it is impractical to actually implement different options and measure their impact [Kos+20d]. Instead, cost models can estimate the impact on workload processing while avoiding the overhead of actually installing the particular options. Recently, machine learning-based approaches have been developed to overcome the inaccuracies [Lei+15] of traditional, often manually tuned, cardinality and cost estimation approaches. The new techniques focus on, e.g., cost estimation for indexes [Din+19], combined approaches for learned cardinality and cost estimation [SL19], simple operator cost models that are automatically calibrated to cover a broad range of operator instances at runtime [LLK21], or approaches that decompose DBMSs into operating units to predict costs on a more fine-grained level [Ma+21].

Workload Prediction. In addition, workloads vary significantly over time [Ma+18]. Accurate predictions of these workload changes enable the determination of reasonable points in time for database optimizations and more suitable configurations, because the anticipated workload better matches the actual workload. Workload predictions can be implemented on different abstraction levels: such forecasting techniques can predict workloads either in terms of resource consumption, such as CPU time or Disk IO [Das+16; Hig+20] or query arrival rates [Ma+18].

Workload Modeling. Some workload prediction and database optimization techniques rely on workload models and representations that differ from a simple collection of SQL queries. In particular, ML-based techniques typically work with numerical vectors or matrices. For this reason, recent work explores multiple opportunities for workload representation, such as query clustering [Ma+18], vectorization [KKS22a; SL19], decomposition [Ma+21], or data access counters [Bre+21; Dre21].

1.1.4. Learned Components

Data structures and database components were traditionally designed and implemented by human developers. Kraska et al. [Kra+18] showed that such components can be learned entirely by presenting *learned indexes*. Examples include database schedulers [Mao+19; SUK22], transaction handlers [Wan+21], or different index structures [Kip+20; Kra+18; Liu+20; Lu+21; Nat+20]. Work in this area does not necessarily aim to automate tasks that were traditionally manually performed by DBAs. However, since learned database components are somehow *autonomous*, they should not be ignored when investigating autonomous DBMSs. Some topics classified into *optimization and execution* of focus area (ii) *Unsupervised Database Optimization* could also be classified as a *Learned Component* (focus area (iv)), e.g., the learned query optimizer [Mar+19]. However, since execution and, in particular, optimization are of utmost importance for DBMSs [Neu14], we list such topics as part of focus area (ii) *Unsupervised Database Optimization.*

1.2. Part I: Efficient Index Selection

In the following, we motivate and contextualize the two topics, *efficient index selection* and *data dependency-driven query optimization*, that we will explore in this thesis. Well-chosen physical database design can significantly impact the workload processing costs [ANY04]. Therefore, the relevance of physical database design for database performance is undisputed by DBAs and in the literature [FST88; Lig18; LTN07; Zil+04]. For example, secondary indexes are indispensable for the performance of relational database systems [BS18]. Furthermore, modern, cloud-based data warehouse systems apply indexing techniques to process analytical workloads efficiently [Ana; Sno].

For this work, we have selected the index selection problem as a representative for discussing the complexity and challenges of physical database design. This problem describes the process of determining the optimal set of performance-enhancing indexes for a particular workload given certain constraints, such as a storage budget [Pia83].

We chose index selection because it properly represents the challenges common to physical database design. First, the solution space, i.e., the set of possible index combinations, is enormous. It increases with the number of indexable attributes and the number of attributes per index [SKB19]. Second, indexes interact, i.e., the benefit of one index may depend on the existence of other indexes [SPG09]. Therefore, benefits must be recomputed frequently: the benefit associated with an index might change every time another index is created or dropped. Third, it is challenging to accurately quantify the expected benefit of index candidates without physically creating indexes and executing the workload [Din+19]. As a result, index selection approaches must handle these complications to determine performance-enhancing index sets. At the same time, solution runtimes and resource utilization must be kept to a minimum, especially if numerous systems are to be tuned as it is necessary in cloud environments [Das+19].

It is crucial to understand the strengths and weaknesses of the approaches to determine which of the approaches should be applied in a particular scenario. Furthermore, objective analyses are the only way to identify opportunities for improving index selection techniques. Although dozens of index selection solutions have been proposed, such an analysis has not been conducted [Kos+20d]. Hence, this lack of objective analyses is another reason for choosing index selection as a representative for physical database design.

We will discuss our analysis of index selection approaches later in Chapters 3 and 4. The results indicate that, in some problem dimensions, existing approaches have limitations — as depicted in Figure 1.2. There are various state-of-the-art index selection approaches, some of them related to commercial DBMS products. Some existing approaches, such as *DTA*, produce high-quality solutions for complex workloads or provide low index selection runtimes, such as *DB2Advis* and reinforcement learning (RL)-based approaches.

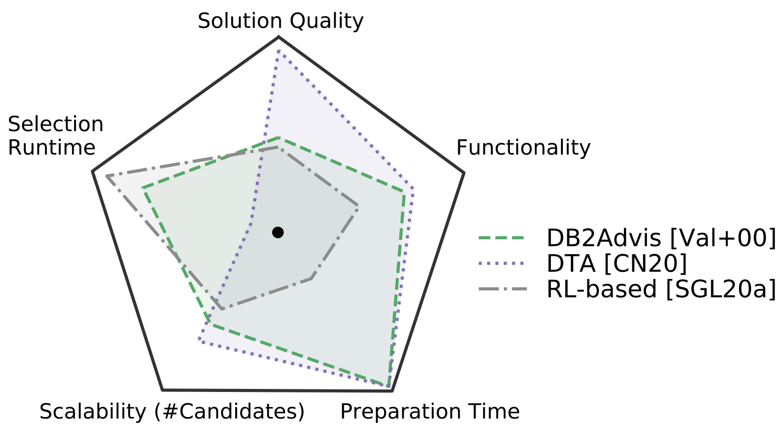


Figure 1.2.: Schematic comparison of index selection approaches. State-of-the-art index selection algorithms show weaknesses in different dimensions. *Functionality* includes, e.g., support for multi-attribute indexes or budget constraints. Larger distance to the center is better.

Nevertheless, they fall short of striking the right balance between both metrics and offering the desired functionality, as displayed in Figure 1.2. We argue that, due to complex and varying workloads, autonomous DBMSs must employ powerful index selection approaches that react quickly to workload changes in order to achieve close-to-optimal performance. In this context, we will survey existing index selection approaches, investigate their performance, and develop new automated index selection techniques that overcome the highlighted shortcomings in Part I of this thesis.

1.3. Part II: Data Dependency-Driven Query Optimization

Optimizations based on physical database design can be complemented by optimizations that utilize properties of the data itself to improve query processing performance. *Data dependencies* are such properties; they express relationships between relational attributes. For example, a functional dependency (FD) expresses that any two records with identical values in the attributes X also have the same values in the attributes Y [Cod71]; in a relation with address data, the combination of the attributes street_address and zip_code could functionally determine the attribute city. Such dependencies can exist due to the very nature of the underlying data, or be artificially introduced to datasets, e.g., with surrogate keys. A more formal introduction of FDs and other data dependencies will be given in Chapter 6 of this thesis.

Data dependencies can be applied during query optimization to achieve more efficient query processing, e.g., aborting scans early, using binary searches instead of linear scans, or removing redundant GROUP BY attributes. For example, the following query can be simplified if we assume the FD mentioned above on the address data relation:

```
SELECT [...] FROM address_data
    GROUP BY street_address, zip_code, city.
```

The grouping attribute city is unnecessary, because all of the elements that fall into the same group for street_address, zip_code necessarily also have the same value for city. Nevertheless, most data dependency-driven techniques are rarely implemented in existing database systems [KPN22]. Data dependencies are not frequently used in modern query processing engines because no comprehensive collection of dependency-driven techniques existed; we present such a collection in Chapter 7. Also, while there are typically many valid data dependencies on a given dataset [SP22], such dependencies are often unknown. Some of them, such as order dependencies (ODs), cannot be manually defined in DBMSs. However, advancements in the field of data profiling [Abe+18] in combination with our workload-driven discovery approach (Chapter 8) increase the feasibility of discovering data dependencies and make query optimization based on these dependencies more viable. We argue that, with ever-increasing data volumes [Que20] and performance demands, autonomous DBMSs must utilize such data-inherent knowledge in the future to achieve efficient query processing. Therefore, we examine how data dependencies can be used for query optimization in Part II of this thesis.

1.4. Research Questions and Contributions

Based on the aforementioned challenges and explanations, we derive three research questions, which guide our work on unsupervised database optimization. The first two questions target the storage perspective, in particular, *efficient index selection*. In contrast, the third question focuses on the query perspective, i.e., the utilization of *data dependencies for query optimization*. We present our contributions in the context of the research questions below. In addition, we also summarize our research in the area of autonomous database systems, which is not part of this thesis' main contributions.

(1) An Experimental Survey of Index Selection Algorithms

Research question: *How can we analyze, compare, and classify unsupervised index selection algorithms and investigate which factors influence their performance regarding the quality of the identified solutions as well as the required runtime?*

Significance: It is crucial to objectively evaluate existing index selection approaches to identify their differences and limitations. Based on these insights, approaches can be chosen for particular application scenarios. Moreover, such analyses are necessary to identify challenges that can be addressed by developing new approaches.

Contribution: We classify seven diverse index selection algorithms by their modus operandi, functionality, strengths, and weaknesses. Furthermore, we develop a platform for evaluating index selection algorithms for different workloads and datasets to investigate which factors influence their effectiveness and efficiency. The platform is flexible and facilitates the extension by other algorithms, workloads, or physical design aspects. Based on the evaluations conducted with this platform, we deduce insights regarding which approaches are beneficial to particular situations. In terms of solution quality, approaches may vary substantially. Regarding the index selection runtime, they may even differ by orders of magnitude. The community actively uses[2] our open-source platform[3] and the generated insights to evaluate and develop new index selection approaches [LZC21; Per+21; WTB21a]. Contribution (1) is based on the following publication:

[Kos+20d]: 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". In: *Proceedings of the VLDB Endowment* 13.11 (2020), pp. 2382–2395.

Note that the detailed contributions of the thesis author to the aforementioned and the following publications are explained at the beginning of the particular chapters below.

(2) Scalable and Effective Index Selection Algorithms

Research question: *How can we scale index selection processes to efficiently determine near-optimal index sets, even for large problem instances and considering complex effects, such as index interaction?*

Significance: Numerous tunable DBMS instances as well as complex, varying workloads pose challenges for existing index selection approaches in cloud environments. Therefore, more scalable, faster, and effective index selections are required to handle such challenges.

Contribution: Based on the previously obtained findings, we propose two new index selection algorithms that overcome existing limitations. Both algorithms complement each other by serving various purposes: (i) *Extend* focuses on identifying (near-)optimal solutions with a heuristic that iteratively *extends* index configurations. While its solutions outperform other approaches for all evaluated benchmarks, it also undercuts its direct competitors in solution time by up to a factor of 10. (ii) *SWIRL* targets cloud scenarios with numerous tunable systems by concentrating on rapid solution times enabled by

[2]Our platform's open-source repository has been forked several times and we are being contacted regularly by fellow researchers regarding index selection evaluation opportunities.

[3]Index selection evaluation platform on GitHub: `https://git.io/index_selection_evaluation`

reinforcement learning, resulting in runtime improvements by orders of magnitude in many scenarios. Simultaneously, the quality of the solutions is, on average, within 3 % of the best solutions.

Both approaches are evaluated extensively and compared with the strongest competitors. We detail the design and implementation decisions of *Extend* and our reinforcement learning-based approach *SWIRL* and survey other index selection approaches based on reinforcement learning. The implementations are provided via open-source repositories [Kos+20c; KKS22b] for *Extend* and *SWIRL*. The material of Contribution (2) is published in the following papers and patents:

[KKS22a]: Jan Kossmann, Alexander Kastius, and Rainer Schlosser. "SWIRL: Selection of Workload-aware Indexes using Reinforcement Learning". In: *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 2022, pp. 155–168.

[Kos+22b]: Jan Kossmann, Rainer Schlosser, Alexander Kastius, Michael Perscheid, and Hasso Plattner. *Training an Agent for Iterative Multi-Attribute Index Selection.* European Patent Application EP22156399.2. February 2022.

[Sch+20]: Rainer Schlosser, Jan Kossmann, Martin Boissier, Matthias Uflacker, and Hasso Plattner. *Iterative Multi-Attribute Index Selection for Large Database Systems.* European Patent EP3719663B1; US Patent Application 16/838,830. October 2020.

[SKB19]: Rainer Schlosser, Jan Kossmann, and Martin Boissier. "Efficient Scalable Multi-attribute Index Selection Using Recursive Strategies". In: *Proceedings of the International Conference on Data Engineering (ICDE)*. 2019, pp. 1238–1249.

(3) Data Dependency-Driven Query Optimization

Research question: *How can a DBMS use data dependencies for improved query optimization, and how can relevant dependencies be identified in an unsupervised fashion?*

Significance: Previous research has identified many data dependencies and applications for query optimization on real-world datasets. However, the potential of data dependencies for query optimization is not utilized by DBMSs in general. Ever-increasing data volumes will not allow such data-inherent optimization potential to be ignored in the future.

Contribution: Initially, we compile a collection of 58 data dependency-based query optimization techniques. We provide brief and intuitive descriptions of all techniques and classify them by their application area, data dependency type, and query optimization phase. In addition, we analyze the impact of selected data dependency-driven optimization techniques on query performance. Furthermore, we present a workload-driven, lazy mechanism that determines valuable dependencies in an unsupervised fashion. The resulting runtime benefits outweigh the mechanism's overhead for the selected techniques, achieving query speed-ups of up to 61 × for some TPC-DS queries and an average runtime reduction for the Join Order Benchmark (JOB) workload of 26 %.

For the impact analysis and evaluation of our approach, both the mechanism and the exemplarily selected techniques are implemented[4] in our research DBMS Hyrise [Dre+19]. This contribution is based on the following previously published material:

[KPN22]: Jan Kossmann, Thorsten Papenbrock, and Felix Naumann. "Data dependencies for query optimization: a survey". In: *VLDB Journal* 31.1 (2022), pp. 1–22.

[Kos+22a]: Jan Kossmann, Felix Naumann, Daniel Lindner, and Thorsten Papenbrock. "Workload-driven, Lazy Discovery of Data Dependencies for Query Optimization". In: *Proceedings of the Conference on Innovative Data Systems Research (CIDR).* 2022.

Complementary Contributions

In addition to the previously mentioned main contributions, we have contributed to the field of autonomous database systems with further research. First, we proposed a *component-based framework* to facilitate the development and system integration of autonomous functionality from an architectural point of view. This framework also includes an integer linear programming (ILP)-based approach to derive optimized tuning orders for mutually dependent aspects of physical database design, e.g., index selection and automated compression configuration. This line of work is not part of this thesis but examined in the following publications:

[KS20]: Jan Kossmann and Rainer Schlosser. "Self-driving database systems: a conceptual approach". In: *Distributed And Parallel Databases (DAPD)* 38.4 (2020), pp. 795–817.

[KS19]: Jan Kossmann and Rainer Schlosser. "A Framework for Self-Managing Database Systems". In: *Proceedings of the International Conference on Data Engineering (ICDE) Workshops.* 2019, pp. 100–106.

[Kos18]: Jan Kossmann. "Self-Driving: From General Purpose to Specialized DBMSs". In: *Proceedings of the VLDB PhD Workshop.* 2018.

Second, for our research DBMS Hyrise, we developed a *plugin concept* that enables the seamless integration of autonomous functionality from a development perspective. Thereby, we avoid the tight coupling of such functionality with database core elements. This plugin concept is demonstrated in the *Hyrise Cockpit for Unsupervised Database Optimization*, which allows the user — such as a DBA — to interactively evaluate the plugins' performance impact, observe their interplay, and comprehend the plugins' decisions to build trust in unsupervised techniques. The cockpit and the plugin concept are discussed in this thesis. This line of work is also part of the following publications:

[4]Source code on GitHub: `https://github.com/Bensk1/phd_thesis/releases/tag/source_code`

[Kos+21]: Jan Kossmann, Martin Boissier, Alexander Dubrawski, Fabian Heseding, Caterina Mandel, Udo Pigorsch, Max Schneider, Til Schniese, Mona Sobhani, Petr Tsayun, Katharina Wille, Michael Perscheid, Matthias Uflacker, and Hasso Plattner. “A Cockpit for the Development and Evaluation of Autonomous Database Systems”. In: *Proceedings of the International Conference on Data Engineering (ICDE).* 2021, pp. 2685–2688

[Dre+19]: Markus Dreseler, Jan Kossmann, Martin Boissier, Stefan Klauck, Matthias Uflacker, and Hasso Plattner. “Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management”. In: *Proceedings of the International Conference on Extending Database Technology (EDBT).* 2019, pp. 313–324

Third, we have investigated how the unsupervised clustering of database tables improves a workload’s execution time. In particular, we examined how to accurately predict the impact of different clustering configurations with *learned cost models.* Based on these predictions, the best clustering configuration can be determined without the need for expensive installation and evaluation. The corresponding findings are not part of this thesis but presented in the following publication:

[LLK21]: Daniel Lindner, Alexander Löser, and Jan Kossmann. “Learned What-If Cost Models for Autonomous Clustering”. In: *Advances in Databases and Information Systems (ADBIS).* 2021, pp. 3–13

1.5. Outline

The remainder of this thesis consists of three parts: Part I examines automatic index selection; it comprises Chapters 2 to 5. Chapter 2 introduces the necessary background information by formalizing the index selection problem; the chapter also highlights the problem’s challenges to demonstrate the complexity of the problem. In Chapter 3, we discuss related work by surveying, explaining, and comparing seven existing index selection algorithms. Afterward, we present an index selection evaluation framework and conduct an experimental evaluation of the previously presented algorithms in Chapter 4. Based on the insights generated by the experimental evaluation, we develop two improved index selection algorithms in Chapter 5 and evaluate their performance.

Part II concerns the usage of data dependencies for query optimization; it comprises Chapters 6 to 8. We start by giving background information on different data dependency types, their properties, and query optimization in Chapter 6. In Chapter 7, we explore related work by surveying 58 query optimization techniques based on data dependencies. Chapter 8 details and evaluates our workload-driven dependency discovery approach.

Part III completes this thesis by presenting our *Cockpit for Unsupervised Database Optimization* (Chapter 9) and our conclusions in Chapter 10.

Part I.

Unsupervised Database Optimization: Efficient Index Selection

2 Background: Index Selection

Indexes are auxiliary data structures that can enhance the performance of certain database operations. Typically, there is a tradeoff between improved performance and elevated storage consumption caused by these data structures. While performance is essential for productive systems, storage is also a scarce resource [Zha+16]. Moreover, index structures do not always accelerate query execution. In fact, additional indexes can even increase the workload processing time, e.g., because inserts or updates require expensive maintenance operations on the index structures [Gra06].

The index selection problem is about finding the set of indexes for a given workload that maximizes the system performance while considering specific constraints, e.g., a storage budget or the number of indexes to create. In this chapter, we first provide the theoretical background by formalizing the problem (Section 2.1) before we discuss the challenges of index selection in Section 2.2.

Parts of this chapter have been published in two research papers [Kos+20d; KKS22a]. The thesis author prepared the majority of the original draft for publication [KKS22a]. The author developed the underlying concept, implemented the approach, and designed and executed all experiments. Kastius and Schlosser supported the reinforcement learning (RL) model's design, improved the material and its presentation, and co-authored the paper. Publication [Kos+20d] was a collaborative effort. The primary authors, Halfpap and the thesis author, contributed the majority of the paper's conceptualization and original draft. The thesis author conducted most of the experiments and their evaluations. Furthermore, the primary authors improved, revised, and extended Jankrift's prototypical version of the evaluation platform. Jankrift and Schlosser contributed to the paper's concept, improved its material and presentation, and co-authored the paper.

2.1. Formalization

In the upcoming chapters of this thesis, we rely on the following notation and formalization of the index selection problem. The notation is summarized in Table 2.1.

Table 2.1.: Notation table for index selection.

Parameters	K	number of attributes
	N	number of queries
	q_n	attributes accessed by query n, $n = 1, ..., N$, subset of $\{1, ..., K\}$, e.g., $q_1 = \{8, 6, 13, 14\}$
	f_n	frequency of query n, $n = 1, ..., N$
	I	set of index candidates
Cost parameters	B	storage budget (constraint)
	L	maximum number of selected indexes, $\|I^*\|$ (constraint)
	$c_n(\emptyset)$	cost of executing query n without an index
	$c_n(I^*)$	cost of executing query n with index configuration I^*
	$C(I^*)$	total workload costs, sum of all query costs under I^*
	m_i	size (storage consumption) of index i, $i \in I$
Variables	W	index width, number of attributes
	W_{max}	largest width considered during index selection
	i	index, ordered set $i = \{i_1, ..., i_W\}$ with W attributes
	i_u	u^{th} attribute of i, $i_u \in \{1, ..., K\}$, $u = 1, ..., W$
	I^*	index selection or configuration, $I^* \subseteq I$
	x_i	index i selected, yes (1) / no (0)
	$M(I^*)$	Occupied storage of index selection I^*. Short for $M(I^*(\vec{x}))$

Workload. A *workload* is a set of N *query templates* or *query classes* defined over a set of tables with K *attributes* or *columns.* A query n is characterized by the subset of attributes $q_n \subseteq \{1, ..., K\}$ that are accessed for its evaluation, $n = 1, ..., N$. Often, algorithms also assign queries a *weight* to differentiate their importance or represent the *frequency* of the query's occurrence. Queries of class n occur with frequency f_n, $f_n \geq 0$.

Index. An index i with W attributes is represented by an ordered set of attributes $i = \{i_1, ..., i_W\}$, where $i_u \in \{1, ..., K\}$, $u = 1, ..., W$. W is also called the *width* of an index and corresponds to the number of contained attributes. W_{max} denotes the largest index width considered during index selection. An index cannot incorporate attributes of multiple tables but is only created on a single table.

Potential Index. A potential index p with W attributes is any index that could be generated from an arbitrary combination of attributes (from the same table) that are part of the workload ($\bigcup_{n=1,...,N} q_n$), i.e., $p = \{i_1, ..., i_W\}$. Note, in the work of Chaudhuri and Narasayya discussing index selection, potential indexes are called *admissible indexes*, which consist of so-called *indexable attributes* [CN97].

Index Candidates. Potential indexes that are considered and evaluated by index selection algorithms are *index candidates I.* Usually (because of their large number and combination possibilities), index selection algorithms cannot consider all potential indexes, e.g., index candidates with many attributes are often discarded. Choosing index candidates from all potential indexes is an essential part of index selection algorithms. Many algorithms focus on *(syntactically) relevant indexes* that contain only attributes that appear together in at least one query. Well-chosen candidates can impact the quality of the final solution significantly [SKB19].

Index Selection. An *index selection* or *index configuration* is a *set* of indexes, $I^* \subseteq I$. Typically, algorithms evaluate different candidate sets and index configurations during the selection process. Note, depending on the context, I^* can refer to any potential index selection as well as the final selection.

Index Size and Selection Status. The *required storage* (or memory for in-memory indexes) for an index $i \in I$ is denoted by m_i, often denoted as *index size*. We use the binary variables x_i, which indicate whether an index $i \in I$ is part of the selection I^* (1 yes, 0 no), i.e., $I^*(\vec{x}) := \bigcup_{i \in I : x_i = 1} \{i\}$, where $\vec{x}$ is a short vector notation for all x_i. Then, the total storage M used by a selection $I^* = I^*(\vec{x})$ amounts to

$$M(I^*(\vec{x})) \text{ or simply } M(I^*) := \sum\nolimits_{i \in I} m_i \cdot x_i. \tag{2.1}$$

Query and Workload Costs. Query and workload *costs* are important input parameters for index selection algorithms. Costs are used to determine the most beneficial indexes. The costs to execute a query of class n depend on the currently active selection of indexes I^* and are denoted by parameters $c_n(I^*)$. Note that c_n must be defined for arbitrary $I^* \subseteq I$. The total workload costs C are defined as the sum of the cost of all queries considering their frequencies and the current index selection I^*:

$$C(I^*) := \sum\nolimits_{n=1,\dots,N} f_n \cdot c_n(I^*). \tag{2.2}$$

Index Selection Problem — Optimization Target. The target of the index selection problem is to solve the following optimization problem given a constraint. Most commonly, as we will discuss in Chapter 3, the constraint is a storage budget, $B \geq 0$, that cannot be exceeded:

$$\underset{x_i \in \{0,1\}, i \in I}{\text{minimize}} \quad C(I^*(\vec{x})) \quad \text{subject to} \quad M(I^*(\vec{x})) \leq B. \tag{2.3}$$

Note, some approaches do not consider storage budgets. Instead, they work with cardinality constraints and restrict the number L of selected indexes, $|I^*| = \sum_{i \in I} x_i$, $|I^*| \leq L$. Other approaches employ multiple constraints, for instance, a budget B and a maximum allowed index selection runtime T. The impact of different constraints on the index selection process is discussed later in Chapter 4.

Hypothetical Index. Hypothetical indexes do not physically exist [CN97]. Their existence is only simulated to trick the optimizer (*what-if* optimization [CN97]) into generating query plans and cost estimations that would also be created if the index was actually physically present. In the work of Valentin et al. [Val+00] and IBM's Db2 [IBM22a], these hypothetical indexes are called *virtual indexes*.

2.2. Challenges

Formally, it has been shown that the index selection problem is NP-complete [Pia83]. While it is similar to the also NP-complete Knapsack Problem [Mat96], the index selection problem is more complex, as we will discuss below. In the following, we further detail the three main challenges of index selection: (i) the large solution space, (ii) index interaction, and (iii) the quantification of index impact. We also provide a first overview on how index selection approaches overcome these challenges and discuss their limitations.

2.2.1. Large Solution Space

For reasonably sized datasets and workloads, numerous — possibly millions of — options for indexation, i.e., index candidates, exist [DPA11; Kos+20d]. The number of *relevant* index candidates depends on the number of attributes accessed by the workload's queries and the maximal number of attributes per index. Typically, multi-attribute candidates are generated by permuting single-attribute candidates. Real-world datasets can even contain tens of thousands of attributes. For instance, single tables of SAP's enterprise resource planning (ERP) system can consist of hundreds of attributes [BSU18]. Furthermore, Faust et al. [Fau+16] have shown that wide indexes (up to 16 attributes) are relevant for relational database management systems (DBMSs) in practice.

Table 2.2 depicts the number of index candidates for increasing index widths and other benchmark metrics for three frequently used database benchmarks: the TPC-H, TPC-DS, Join Order Benchmark (JOB), and one real-world query set based on a workload sample of SAP's ERP system. The benchmarks will be described in more detail in Section 4.1.1. The table demonstrates that the number of index candidates is not simply a function of the number of attributes and queries but instead depends on the dataset's structure and the queries' contents. In addition, it indicates that the number of index candidates is enormous. Given that these candidates are the basis to generate orders of magnitude more combinations of candidates, the size of the solution space appears tremendous. The SAP sample shows that this is particularly true for real-world data: the sample contains a single wide table and some of its queries access many attributes leading to an excessive number of candidates. In practice, evaluating all candidate combinations is, in general, impractical as their number exceeds the number of attributes and index candidates by orders of magnitude [Zil+04]. Hence, enumerating all solutions is infeasible.

Table 2.2.: Solution space metrics for four schemata and workloads. The number of relevant index candidates was determined by generating all permutations of all syntactically relevant indexes. *SAP** refers to a workload sample executed on SAP's S/4 HANA ERP system table ACDOCA (universal journal [SAP17]).

Benchmark	Relations	Attributes	Queries	Relevant W-attribute candidates			
				$W = 1$	$W = 2$	$W = 3$	$W = 4$
JOB	21	108	113	73	218	552	1 080
TPC-H	8	61	22	53	398	3 306	29 088
TPC-DS	24	429	99	248	3 734	68 052	1 339 536
SAP*	1	344	141	344	117 992	40 353 264	$\approx 1.4 * 10^{10}$

Existing index selection algorithms approach the challenge of large solution spaces in multiple ways. Some approaches, e.g., Lan et al.'s solution [LBP20], limit the number of candidates early in the selection process according to specific rules. Others reduce the problem's dimensions by only determining single-attribute indexes, for instance, Whang's *Drop* heuristic [Wha85] (see Section 3.1.2). A different class of approaches, e.g., *CoPhy* [DPA11] (see Section 3.1.7), relies on integer linear programming (ILP) solvers that prune non-optimal options. Schlosser et al. [SKB19] have demonstrated that early candidate reduction can harm performance and that efficient solver-based approaches cannot handle large solution spaces without reducing the candidate set a priori.

2.2.2. Index Interaction

During index selection, the candidates cannot be considered independent because indexes *interact*. According to Schnaitter et al., "[...] an index a interacts with an index b if the benefit of a is affected by the presence of b and vice-versa" [SPG09, p. 1234]. Hence, the existence of one index can affect the potential performance impact of other indexes. Thus, during every step of an index selection process, the currently existing indexes must be considered. This fact requires frequent recomputations of the candidates' benefits because every index creation or removal might drastically impact every other index candidate's potential benefit. Thereby, the complexity of choosing suitable candidates and calculating the best selection is significantly increased. At the same time, interaction is the main differentiator to the classic Knapsack Problem, where the value of items does not depend on previously chosen items.

There are fundamental differences in how the existing approaches deal with index interaction. Some simple heuristics do not consider index interaction, resulting in degraded performance [SKB19]. While many approaches take interaction into account to some degree, some of them expensively recompute every candidate's benefit with every step, e.g., the approaches *Drop* or *Relaxation* [BC05]. We will examine the tradeoff between the solution's performance and computation time later in Section 4.2.

2.2.3. Quantifying Index Impact

For comparing index candidates and choosing the most promising ones, their potential performance impact, $C(I^*)$, must be determined. Thus, index selection algorithms must quantify the benefit of candidates, i.e., the cost savings when the index is utilized for executing the workload at hand. Furthermore, the storage consumption, m_i, is of interest. A possibility to quantify these metrics is to create the index physically, execute and measure each query whose costs are affected by the index, and obtain the size of the index. Usually, this measurement-based approach produces accurate values. At the same time, this approach is prohibitively expensive for large workloads, mainly because quantifying benefits for index interactions would require repeated recreations of the same indexes and executions of the same queries. Later, in the experimental evaluation (Chapter 4), we will see that some algorithms would need to physically create thousands of indexes and determine index benefits via millions of query executions during index selection.

Therefore, most index selection algorithms do not measure but only estimate index benefits to avoid being thwarted too much by determining query costs. It is common to use the database system's optimizer and its cost model for these estimations because the optimizer chooses the query plan and, in particular, which indexes are used. Furthermore, to avoid not only the execution of queries but also the creation of indexes, some database systems support *hypothetical* indexes. Hypothetical indexes are not (entirely) physically created but only inexpensively simulated for cost estimations. The DBMS optimizer considers these hypothetical indexes to generate query plans and cost estimations. This technique is also denoted as *what-if optimization* [CN98].

Although the queries are not executed, what-if optimizer calls take a considerable time because they involve the usual query optimization process. Sometimes, complicated queries are optimized for hundreds of milliseconds and planning time can even surpass execution time [Kos+18; SCJ15]. In particular, for large workloads, where millions of configurations must be evaluated, what-if optimization can become a significant bottleneck. Papadomanolakis et al. [PDA07] show that index selection algorithms spend, on average, 90 % of their runtime in the optimizer. Hence, expensive cost estimations must be kept to a minimum.

While existing work targets the overhead of what-if calls, index selection times are still significant. For example, Chaudhuri and Narasayya discuss techniques to decrease the number of optimizer calls by (i) reducing the set of configurations to evaluate and (ii) deducing costs from simpler configurations [CN97]. Papadomanolakis et al. [PDA07] present a cache-like approach (INUM) to reduce the number of what-if calls. The authors report calculation time improvements of three orders of magnitude without accuracy losses. Besides, Bruno and Nehme present another method to speed up what-if-based cost estimation called C-PQO [BN08]. Instead of issuing many calls for different index configurations, the authors propose a single, more sophisticated optimizer call per query. The single call is more expensive than usual but allows evaluating the impact of multiple configurations afterward without reoptimization.

Cost Estimation Inaccuracies. In general, optimizer-based cost estimations can result in significant estimation errors due to cardinality misestimations [Lei+15] or inaccurate cost models [Wu+13]. As a result, index configurations predicted to be beneficial can, when actually executing queries, result in performance regressions [BAA12; Das+19]. The solution quality of index selection algorithms is, to some extent, bounded by the accuracy of the cost estimations. However, cost estimations are typically the only feasible option for determining costs for large workloads and still allow for comparing the power of different index selection approaches [Kos+20d].

Besides, recent progress in the area of accurate learned cost models could mitigate cost estimation issues. For instance, Ding et al. [Din+19] demonstrate how machine learning classification approaches can be leveraged to predict which plan (with or without index) will be more efficient. The learned cost models based on neural networks of Marcus and Papaemmanouil [MP19] could be used to mitigate the problems arising from inaccurate cost models, too. Furthermore, we combined learned cost models with what-if optimization techniques to improve cost estimations explicitly for physical database design problems [LLK21].

While the index selection process relies on cost estimations, the algorithms are independent of the exact underlying cost model. Hence, improved cost estimation techniques could be utilized without adapting the index selection approaches. In the remainder of this work, we consider the further investigation and development of techniques for more accurate cost estimations out of scope.

3

A Survey of Index Selection Algorithms

In this chapter, we survey seven index selection algorithms. We explain our choice of the investigated algorithms, detail their functioning and parameters, and compare them in Section 3.1. Afterward, Section 3.2 highlights specifics of commercial index selection tools, before Section 3.3 discusses alternative index selection techniques.

Parts of this chapter have been published in the paper [Kos+20d]. The thesis author's detailed contributions to this paper were discussed at the beginning of Chapter 2.

3.1. Investigating Seven Index Selection Algorithms

Index selection algorithms have been published since 1971 and differ in their underlying strategies and complexity. Figure 3.1 gives a time-based overview of the examined publications as well as milestones in the field of index selection algorithms. Before we give detailed explanations of the different algorithms in Sections 3.1.2 to 3.1.8, we provide a high-level comparison and justify the selection of algorithms in Section 3.1.1.

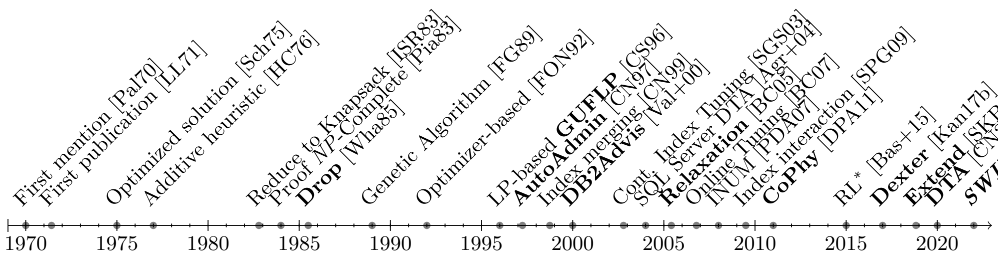


Figure 3.1.: Timeline of milestones in index selection research. Algorithms that are described, implemented, and evaluated as part of this thesis are highlighted with **bold** type. *RL refers to *reinforcement learning*-based approaches.

3.1.1. Algorithm Comparison

We have aimed for a diverse selection of index selection approaches for our survey and the upcoming evaluation. The differences become apparent when we compare the algorithms across several dimensions, as depicted in Table 3.1. In the following, the algorithm names are *italicized* for clarity.

Table 3.1.: Summary of the compared algorithms in chronological order. † Indicates that *DTA* was continuously refined; we consider the latest publication from 2020. Commercial* indicates whether the algorithm is *related* to a commercial index selection tool.

	Drop	AutoAdmin	DB2Advis	Relaxation	CoPhy	Dexter	DTA
Year of publication	1985	1997	2000	2005	2011	2017	2020†
Commercial*	No	Yes	Yes	No	No	No	Yes
Optimization target	Costs	Costs	$\frac{Costs}{Storage}$	$\frac{Costs}{Storage}$	$\frac{Costs}{Storage}$	Costs	Costs
Constraint	# Indexes	# Indexes	Storage	Storage	Storage	Savings (%)	Storage
Multi-attribute	No	Yes	Yes	Yes	(Yes)	(Limit 2)	Yes
Underlying strategy	Reduce	Add	Add	Reduce	Declarative	Add	Add
Index interaction	+++	++	+	+++	+++	+	++
Impl. complexity	+	+++	++	+++	+++	+/++	+++

Year of Publication and Relation to Commercial Tools. We included one of the *first (Drop)*, *intermediate*, and *recent (DTA)* algorithms. Most algorithms were proposed in *research* papers. Some are also related to *commercial systems (AutoAdmin*, *DB2Advis*, *DTA)*. *Dexter* is an *open-source* algorithm.

Optimization Target. When selecting indexes, algorithms either aim to *minimize* the benefit per storage consumption (*CoPhy*, *DB2Advis*, *Relaxation*) or the pure benefit (*AutoAdmin*, *Dexter*, *Drop*, *DTA*). The latter target usually leads to a selection consisting of a few large indexes with a high *absolute* benefit. Minimizing the benefit per storage often selects a larger number of smaller indexes with a better *relative* benefit. Ultimately, the proper optimization target depends on the workload: more index structures might require more individual maintenance operations, which should be considered for transactional workloads.

Constraint. Algorithms also differ in the considered *constraint*, cf. Equation (2.3). Some algorithms allow for specifying an arbitrary upper limit of the total index storage consumption B. Others limit the number of selected indexes L. While additional constraints can usually be implemented easily, *AutoAdmin* is the only algorithm explicitly mentioning the support of both constraints in the original publication [CN97].

Multi-attribute Indexes. All algorithms except *Drop*[5] support the selection of multi-attribute indexes. Dexter is limited to two-attribute indexes. In the case of *CoPhy*, in particular, considering multi-attribute indexes increases the number of necessary cost estimations drastically.

Underlying Strategy. Regarding the underlying strategies, we evaluate *imperative* algorithms (i) starting with an empty index configuration and iteratively *adding* indexes (*AutoAdmin*, *DB2Advis*, *DTA*), and (ii) starting with a large configuration that is *reduced* (*Drop*, *Relaxation*), as well as *declarative* approaches based on (iii) linear programming (*CoPhy*). Machine learning-based approaches will be considered later in Chapter 5.

Index Interaction. All of the algorithms investigated consider *index interaction*, but to a varying extent. The consideration of index interaction in *DB2Advis* and *Dexter* is limited — for *DB2Advis*, it happens mainly late in the final variation phase — because both approaches simultaneously create hypothetical indexes for all candidates. Therefore, interactions cannot be captured on a detailed level.

In contrast, the reductive approaches (*Drop* and *Relaxation*) and *CoPhy* consider index interaction to a high degree. The former capture interactions by only removing single/few candidates per step. Due to its integer linear programming (ILP)-based approach and the multitude of evaluated index configurations, *CoPhy* can theoretically capture even more, respectively *all*, interactions. However, it is impossible to consider all possible index configurations for capturing all interactions in practice and for realistic workload sizes due to the number of options.

AutoAdmin and *DTA* offer an intermediate level of considering interaction. These approaches neither capture single interactions nor add large sets of candidates at once.

Implementation Complexity. Finally, there are significant differences regarding the *complexity* of the algorithms' implementations, e.g., caused by more (*DTA*, *Relaxation*) or less (*Drop*) sophisticated candidate selections and transformations to adapt the current index configuration. Of course, the assessment of this category is, to some extent, subject to our subjective impressions. However, the algorithms' descriptions below and their source code[6] give some indication of their complexity. *Drop*'s implementation is not complex because it is essentially built around two main loops. While *Dexter* and *DB2Advis* are multistep processes, they are not as sophisticated as the remaining approaches. Both essentially create all index candidates and check which of these candidates are used by the what-if optimizer. *DB2Advis*' combination of subsumed indexes and random variation adds complexity. We consider the remaining approaches as the most complex because they

[5]*Drop* theoretically supports multi-attribute indexes. However, the index configuration tests (and, thus, runtime) would significantly increase with the number of candidates. Hence, even for smaller workloads, e.g., TPC-H, the application of *Drop* would become infeasible.

[6]Algorithm implementations on GitHub: `https://git.io/IndexSelectionAlgorithms`

either rely on complex index set transformations (*Relaxation*), multiple recursively called enumeration modes (*AutoAdmin* and *DTA*), or ILP formulations (*CoPhy*). While the latter might not be complex by itself, it requires an external solver, is not as debuggable as the other approaches, and, by being *declarative*, builds on a different programming paradigm.

3.1.2. Drop Heuristic

One of the early index selection algorithms is Whang's *Drop* heuristic [Wha85]. As the name implies, this approach successively drops index candidates. First, the initial candidate index set, $S_{|I|} = I$, comprises every potential single-attribute index. The processing costs for the given workload are determined with all index candidates in place, $C(S_{|I|})$. In each of the following drop phases d, $d = |I|, |I| - 1, ...$, every remaining index candidate $i \in S_d$ is removed from the current candidate set and the workload processing cost is re-evaluated, $C(S_d \setminus \{i\})$. The candidate i^* whose removal leads to the lowest cost is permanently removed for the next phase, $S_{d-1} := S_d \setminus \{i^*\}$.

The original version drops index candidates until no further cost reduction is achieved. Furthermore, Whang's work states that costs are determined by data characteristics, not by the query optimizer. However, because the cost determination is not tightly coupled to the algorithm itself, our implementation can rely on optimizer-based costs for a fair comparison with other approaches.

Parameters. In our implementation of the drop heuristic, the maximum number of (finally) selected indexes $|I| - d$ can be configured.

3.1.3. AutoAdmin

Chaudhuri and Narasayya propose the *AutoAdmin* index selection algorithm for Microsoft SQL Server [CN97]. The iterative algorithm identifies multi-attribute indexes by incrementing the index width W in each iteration. Iterations consist of two steps: first, candidates S_n are determined per query $n = 1, ..., Q$. The union of the candidates of all queries $\bigcup_n S_n$ serves as input for the second step, which considers all queries while determining the best index configuration. The procedure only differs in the considered queries and index candidates for both steps. The algorithm combines a complete enumeration of all subsets of index candidates with r elements and a greedy extension to find $t > r$ indexes. If $r = t$, the enumeration evaluates all index subsets with t elements to guarantee an optimal solution. The number of combinations may be prohibitively large to be evaluated in a reasonable amount of time. If $r = 0$, a pure greedy approach is used to decrease the runtime.

Using the single-attribute (r-attribute) indexes of the first (r^{th}) iteration, two-attribute (($r+1$)-attribute) indexes are created and evaluated in the second (($(r+1)^{th}$) iteration. Chaudhuri and Narasayya propose two strategies to create multi-attribute indexes: selecting more indexes for better results or fewer indexes for faster computation times.

Parameters. While the authors mention that a storage budget or the number of indexes can serve as possible constraints, the latter is used throughout the original paper. Our implementation constraints the number of indexes to be as close to the original as possible. Also, the number of naively enumerated indexes and the index width can be configured.

3.1.4. Anytime DTA

The Anytime algorithm of the Database Engine Tuning Advisor (DTA) for Microsoft SQL Server [CN20] is a continuously refined [Agr+04; CN99; CN07; Mic21a] version of the *AutoAdmin* [CN97] index selection. The core approach to first determine index candidates per query and then identify an index configuration for the entire workload, based on the original greedy enumeration, is the same.

The approach uses the following main extensions:

(i) Iterations with increasing index widths are unnecessary since multi-attribute indexes are considered from the start.

(ii) The candidates and configurations are merged after candidate selection to determine further candidates that are beneficial for multiple queries.

(iii) Index interactions are considered by identifying seed configurations to avoid complete enumerations that may evaluate many, certainly suboptimal, configurations.

(iv) As the name suggests, the tuning times can be limited to guarantee solutions in a reasonable time, even for many candidates and seeds. This property is denoted as *anytime* capability in the course of this thesis.

(v) The algorithm can simultaneously tune indexes, materialized views, and partitioning criteria.

Parameters. The maximum width of index candidates and a limit for the tuning time can be configured.

3.1.5. DB2Advis

Valentin et al. [Val+00] present the algorithm DB2 Advisor (short *DB2Advis*) to identify beneficial indexes for IBM's Db2. *DB2Advis* utilizes the optimizer's what-if capabilities and follows a three-step approach.

In the first step, *DB2Advis* determines index candidates. For each query n, $n = 1, ..., N$, of the workload, hypothetical indexes are created on attributes that, e.g., appear as equality or range predicates or in interesting order [Sel+79] lists. In addition, to not miss any candidate, all potential indexes are added as hypothetical indexes until a certain number of indexes is reached. Afterward, the best plan for query n is retrieved from the optimizer. Previously created hypothetical indexes that are utilized in the resulting plan are added to the set of index candidates (I) used in the next step.

In the second step, all index candidates in I are sorted by their benefit-per-space ratio in decreasing order. The benefit corresponds to the difference in query processing costs with and without indexes. Next, index pairs i_1 and i_2 are combined if i_1 has a higher ratio, and its leading attributes equal i_2's attribute permutation. In this case, the benefit of i_1 is updated, i_2 is removed from the list, and I is potentially resorted corresponding to its updated costs. Then, following the sort order, indexes are added to the final index configuration (I^*) until the storage budget (B) is exceeded.

Lastly, the current index configuration is randomly modified to improve its benefit and account for complex effects like index interaction: sets of the previously calculated solution (I^*) are exchanged with sets of indexes that are not part of the solution (due to the budget constraint). If the variation leads to lower overall costs, it becomes the new solution.

Parameters. There are two main parameters: (i) the time for random variations and (ii) the maximum number of hypothetical indexes that are evaluated in the first step of the algorithm. Our implementation also allows configuring the maximum index width and does not limit the number of hypothetical indexes (ii). Such a preselection could be easily added, also to the other algorithms. We refrained from including it to compare the performance of the core algorithms without *hyperparameter* tuning.

3.1.6. Relaxation

Bruno and Chaudhuri propose to derive indexes from a (presumably too large) optimal index set by repeatedly transforming this set to decrease the storage consumption [BC05]. First, they obtain optimal index configurations for each query. Therefore, they instrument the optimizer and exploit knowledge about its index usage to avoid a brute force approach. Second, the optimal index configuration for the entire workload is defined as the union of all queries' optimal index sets. Afterward, this configuration is repeatedly *relaxed*, i.e., the index configuration is transformed, to lower its storage consumption while keeping the configuration's benefits as high as possible. They use five index transformations: (i) Merging two indexes into one. (ii) Splitting two indexes, i.e., creating an index with shared attributes and up to two (one per input index) indexes with the residual attributes.

(iii) Prefixing an index by removing attributes from the end. (iv) Promoting an index to a clustered index. (v) Removing an index.

Parameters. The maximum width for index candidates can be configured as well as which of the transformations are permitted. Our tool uses a brute force approach to obtain the optimal index configuration per query. As clustered indexes are not considered in this thesis, we did not integrate transformation (iv).

3.1.7. ILP-based Approaches (CoPhy)

Integer linear programming (ILP) is a common approach to solve optimization problems by specifying an optimization target and constraints with linear equations. Then, off-the-shelf solvers are used for calculating optimal solutions. Solvers are optimized to discard invalid and sub-optimal solutions early and, thus, outperform brute force approaches significantly. Commonly, index selection algorithms are formulated as integer linear programming problems, which are not scalable. The problem complexity of ILP formulations for the index selection problem can be reduced by restricting the solution space, e.g., the number of index candidates by decreasing the allowed index width. Of course, limited candidate sets might lead to suboptimal solutions for the unrestricted problem [SKB19].

GUFLP. Caprara and Salazar González derive an ILP formulation [CFM95; CS96] for the index selection problem from a Generalized Uncapacitated Facility Location Problem (GUFLP). They reduce the complexity of the ILP formulation with a restriction of the solution space by allowing the utilization of only *a single index per query*. Thus, opportunities that only arise when multiple indexes exist simultaneously are not taken into account.

CoPhy. Dash et al. [DPA11] propose *CoPhy*, a more sophisticated ILP formulation for the index selection problem. In contrast to GUFLP, their approach considers multiple indexes per query (see also [PA07]) and multiple query plans that are potentially chosen by the optimizer depending on existing indexes. Below, we describe the essence of CoPhy's ILP approach.

The costs of (a fixed query plan and) executing a query using specific indexes have to be at hand, e.g., based on what-if cost estimations. We consider a set S of different index combinations. If a *subset* of index candidates $s \subseteq I$, called *option* in the following, is applied to query n, the costs are $c_n(s)$, $s \in S \cup \{0\}$, where 0 describes the option that no index is used for processing n, $n = 1, ..., N$.

In the ILP, binary variables z_{sn} are used to model whether an index option s is applied to a query n, which depends on the selection of other index options which might be more beneficial, cf. *index interaction*. Binary variables x_i indicate whether an index $i \in I$

with its corresponding storage consumption m_i is selected (as part of at least one chosen option s). The constraints of the ILP guarantee that a unique index option is used for each query n and that the used indexes i do not exceed the storage budget B.

The number of variables and constraints characterizes the complexity of the ILP problem. As ILP formulations require the calculation of all cost coefficients $c_n(s)$, the number of necessary what-if calls can be estimated. In general, ILP approaches do not scale as the problem complexity sharply increases in the number of queries N and the number of options $|S|$. Hence, solver-based approaches are (i) either not applicable for large problems (see [SKB19, Table I]), or (ii) lead to *suboptimal* results as the candidate set sizes, cf. S, need to be reduced a priori (see [SKB19, Figure 3-4]). Heuristic decomposition approaches can be used to mitigate such solver-based scalability issues [SH20].

Parameters. The maximum width of index candidates and the number of applicable indexes per query (1 corresponds to GUFLP) can be specified.

3.1.8. Dexter

Dexter is an open-source index selection tool for PostgreSQL [Kan17a; Kan17b] and was developed by Andrew Kane. The algorithm builds on hypothetical indexes and is divided into two phases. First, the processed queries, together with information about their runtime, are gathered from the plan cache. Queries with the same template but different parameter values are grouped.

The second phase involves multiple sub-steps. (i) The initial costs (of the current index configuration I^* where $I^* = \emptyset$ is possible) of the gathered queries are determined by using the `EXPLAIN` command. (ii) Hypothetical indexes are created for all potential single- and multi-attribute indexes ($W_{max} = 2$) that are not yet indexed. (iii) Again, cost estimations and query plans are retrieved from the query optimizer. The hypothetical indexes created in step (ii) that are part of these query plans become index candidates for the corresponding queries. (iv) For all queries, the index candidate with the most significant cost-savings compared to the costs obtained in step (i) is selected.

Dexter does not consider already existing indexes for deletion. Indexes are created independent of their storage consumption and cannot contain more than two attributes.

Parameters. The tool offers a couple of parameters. The most important is the *minimal cost-savings percentage* (default value 50 %). It defines the minimal cost-savings that must be achieved by an index candidate to be selected.

3.2. Commercial Index Selection Tools

While some of the chosen algorithms are related to tools employed in commercial DBMS products, the reimplemented algorithms do not fully reflect the behavior and performance of the original tools, which may be continuously enhanced and optimized. Such tools need to set further focus points, such as robustness, scalability, time-bound tuning, and integration [Agr+04; CN07]. Microsoft's *DTA* [CN20; Mic21a] and IBM's DB2 Design Advisor [Zil+04] can simultaneously consider multiple physical design aspects, e.g., indexes, partitioning, and materialized views. Moreover, commercial tools must be able to tune dynamic workloads in a production environment and support user interaction [CN07].

3.3. Alternative Approaches

Recently, machine learning-based, and in particular reinforcement learning (RL)-based, approaches for index selection have been proposed [Bas+15; SGL20b; SSD18]. Even though the conceptual idea of applying machine learning to index selection is promising, these approaches make different assumptions and have other limitations, e.g., demanding long preliminary training times [Lic+20], ignoring index interaction [Bas+15], or only considering filter operators [SSD18]. Such approaches will be covered separately in Section 5.2.2.

For techniques like *adaptive indexing* [Idr+11] or *database cracking* [IKM07; SJD13; SJD16], indexing is not a separate task, but happens during *normal* query processing: essentially, physical copies of columns are sorted as a by-product of range queries. Even though adaptive indexing and database cracking are valuable indexing techniques, they are not considered in this thesis. These techniques typically target the automatic indexing of column stores, whereas we investigate generally applicable index selection approaches. Also, Schuhknecht et al.'s [SJD13; SJD16] evaluations demonstrate the dependence of these approaches on particular query access patterns. In addition, it is unclear whether the optimizer's cost models consider adaptive or cracker indexes and if the often highly-optimized code of database operators would have to be adapted to reflect such data structures. Furthermore, these techniques are not available for database systems that offer hypothetical index interfaces, such as PostgreSQL or Microsoft SQL Server, which would hinder a fair comparison.

In 2018, Kraska et al. [Kra+18] presented the idea to replace traditional index data structures entirely with learned models. Afterward, this new approach to the traditional topic of indexing has led to a variety of *learned indexing* approaches [Din+20; FV20; Kip+20; Nat+20] that are supposed to outperform state-of-the-art index structures, e.g., B-trees [Gra11]. However, since learned indexes do not specifically target index selection, but rather aim to replace the underlying index structures, learned indexes are not of increased relevance for this survey. As a side note, Crotty [Cro21] demonstrates that most of the advantage of learned index structures is a result of implicit assumptions, e.g., presorted data or read-only workloads. The advantage degrades if traditional index structures make equivalent assumptions.

3.4. Summary

We surveyed and described seven diverse index selection algorithms. The algorithms were chosen to reflect different underlying strategies and complexities. Based on the detailed survey, we compared the algorithms across different dimensions, for instance, the considered constraints or the degree to which index interaction is supported. We discovered that there are significant differences between the algorithms. For example, most (six out of seven) algorithms support multi-attribute indexes. The support for index interaction and the implementation complexities vary to a large extent. Additionally, we discussed relations to commercial index selection tools and highlighted the differences to alternative approaches, such as *database cracking* and *learned indexes*.

4 An Experimental Evaluation of Index Selection Approaches

This chapter compares the previously surveyed index selection approaches with several experiments. We start by detailing the underlying methodology of our evaluation in Section 4.1, where we discuss the evaluated workloads, the experimental setup, and the implementation of our extensible index selection evaluation platform. The evaluation presents results on different benchmark workloads (Sections 4.2.1 to 4.2.3) before more specific aspects, e.g., algorithm cost breakdowns, are targeted (Sections 4.2.4 and 4.2.5). A list of general and per-algorithm insights as well as a summary of our findings conclude the chapter in Section 4.3.

Parts of this chapter have been published in the paper [Kos+20d]. The thesis author's detailed contributions to this paper were discussed at the beginning of Chapter 2.

4.1. Methodology

An objective comparison of index selection algorithms is challenging because the quality of an algorithm depends on multiple factors: the input (workload and index benefits), the algorithm's configurations or parameters, as well as on the constraints, e.g., the budget, and optimization targets, e.g., pure benefit, ratio of benefit and storage, and the granularity of the solutions. In other words, the best algorithm may depend on the specific evaluation scenario. Thus, we cover a broad range of scenarios for our comparison. This section describes the methodology we applied for evaluating the index selection algorithms presented above as well as the limitations of our evaluation.

4.1.1. Workloads

For our comparison, we use three benchmark workloads of different scales and characteristics to investigate the workload's potential impact on the algorithms' solution quality and runtime. The workloads vary in the number of potential indexes, number of queries, or whether they are based on synthetic or real-world data. In the following, we highlight the differences of the TPC-H [PF00], TPC-DS [NP06], and Join Order Benchmark (JOB) [Lei+15].

Relevant metrics for index selection for the three benchmark workloads and schemata are displayed in Table 2.2 on page 21. The two used Transaction Processing Performance Council (TPC)[7] benchmarks are standardized analytical benchmarks operating on a synthetic dataset that can be scaled with a scale factor. The JOB is based on reasonable queries over the Internet Movie Data Base (IMDB). In contrast to the synthetic datasets, the IMDB comprises real-world data with realistic cardinalities and dependencies. The JOB focuses on the processing of joins.

The table demonstrates the different scales of the benchmarks. The TPC-H benchmark is relatively small and can be utilized for quick evaluations, while the TPC-DS benchmark represents more realistic scenarios and is more abundant in every dimension. Even though the JOB consists of the most queries, the number of potential (multi-attributes) indexes is comparably low. Most JOB queries contain many attributes, but most of them belong to different tables and indexes are only created over attributes of the same table.

4.1.2. Query Cost Evaluation

All compared algorithms require either a large number of query cost determinations (given a fixed index configuration) or cost evaluations of large configurations. Although it is theoretically possible to obtain query costs by physically creating indexes and executing queries (repeatedly), it would take too much time and restrict feasible algorithm settings, especially the number of index candidates, by a too large degree. Therefore, we use hypothetical indexes for cost and index size estimations.

Although these estimates can be inaccurate (see Section 2.2.3), they offer a reasonable combination of *speed, accuracy, and accessibility* and are consistent for all algorithms. Note, in this work, instead of assessing cost estimation approaches, we focus on the evaluation of index selection algorithms. Based on the fact that many of the index selection algorithms proposed in the literature require what-if calls for practical application, it is surprising that database systems do usually not expose (a well-documented) interface to retrieve cost estimates for hypothetical indexes. For instance, the following statement creates a hypothetical index in Microsoft's SQL Server: `CREATE INDEX HypoIndex ON MyTable(attr) WITH STATISTICS_ONLY = -1`. However, the necessary option that differentiates hypothetical indexes, `WITH STATISTICS_ONLY`, is not mentioned in the official, public documentation [Mic22b]. Furthermore, SQL Server

[7]Transaction Processing Performance Council (TPC) homepage: `https://www.tpc.org`

does not offer an interface for predicting the size of hypothetical indexes.

Due to the lack of working and well-documented hypothetical index interfaces, database management system (DBMS) options for the evaluation study are limited. However, the DBMS serves only as a vehicle for the index selection evaluation and all competing algorithms are provided with the same interface for hypothetical indexes. Therefore, the specific DBMS choice is of secondary importance as insights about the index selection algorithms themselves are of general nature and transferable to other systems.

We chose PostgreSQL and the extension HypoPG [Rou15]. HypoPG[8] enables creating, dropping, and size estimation of hypothetical indexes. Using PostgreSQL's `EXPLAIN` command, query plans with arbitrary hypothetical index configurations can be inspected. In doing so, we can determine which indexes are used and the estimated total execution cost for the query plan. By calling `ANALYZE` before an index selection evaluation process is initiated, we ensure the existence of up-to-date statistics that are used for cost estimations and for predicting the storage consumption of hypothetical indexes. Thereby, single-attribute statistics (PostgreSQL holds histograms and the 100 most common values by default) are built for all attributes. `EXPLAIN` reports costs in arbitrary units, which are based on and can be tuned with parameters, e.g., specifying the relative performance of processing a row vs. fetching pages sequentially from disk.

We use PostgreSQL planner's default cost parameters and compared cost estimations for actual and HypoPG's hypothetical indexes for the TPC-H and TPC-DS benchmarks [Kos+20a]. For most of the benchmark queries, the differences are insignificant. TPC-DS query 24 is the only of the 121 (22 TPC-H + 99 TPC-DS) queries where the cost difference exceeds ten percentage points and amounts to 23 percentage points. In addition, we compared the cost estimations of PostgreSQL and the commercial `DBMS-X` for different benchmark queries [Kos+20b]. The relative differences between the query costs estimated by these DBMSs are similar.

Index maintenance and build costs are not part of our analysis. Data-modifying statements are not differently costed if indexes exist, neither by HypoPG nor by PostgreSQL's optimizer, which is not an issue for our evaluation of purely analytical workloads. However, our evaluation platform is not conceptually limited to analytical workloads. A manual cost model could complement PostgreSQL's optimizer to approximate index maintenance costs for transactional workloads.

4.1.3. Constraints and Optimization Targets

In the following, we discuss the optimization targets and constraints, i.e., the index selection limit or allowed runtime, that differ for the compared algorithms.

[8]HypoPG source code and releases on GitHub: `https://github.com/HypoPG/hypopg`

Optimization Target. Index selection algorithms aim to optimize for different metrics. Three of the compared algorithms optimize the ratio of index benefit and storage consumption. In practice, where storage is the limiting factor, it is reasonable to consider the selected index configuration's benefit in relation to its storage consumption. Therefore, we evaluate all algorithms concerning the storage consumption of selected indexes and the *estimated*[9] workload costs. *Drop*, *DTA*, *Dexter*, and *AutoAdmin* minimize workload costs without considering storage consumption. Although it is possible to adapt these algorithms to consider the ratio of benefit and size, we decided to evaluate their original implementation that focuses on minimizing workload costs to keep the differences to the initially published and intended versions as small as possible.

Index Selection Limit. Algorithms differ in the way they constrain the number of the selected indexes. *Drop* and *AutoAdmin*[10] limit the absolute number of indexes. *Dexter* limits the number of selected indexes by the minimal cost-saving percentage. Most commonly, algorithms limit the storage budget. We report workload costs with varying storage consumption for all workloads and algorithms. For *Drop*, *AutoAdmin*, and *Dexter*, the number of selected indexes and the minimal cost-saving percentage allow to implicitly control the storage consumption, which is, in general, increasing for both a higher number of indexes and a lower minimal cost-saving percentage.

We provide increasing storage budgets for the algorithms that terminate when the determined index combination utilizes a certain amount of storage (*DTA*, *DB2Advis*, *Relaxation*, *CoPhy*). For algorithms that constraint the total number of indexes, we increment the number until the resulting index configuration exceeds the maximum budget. For *Dexter*, we vary the minimal cost-saving percentage.

Runtime. For some algorithms, the runtime of the index selection process can amount to dozens of minutes for complex workloads. Therefore, the runtime is essential to consider when evaluating index selection algorithms. Only *DTA* originally supports limiting the runtime of the selection process. In general, runtimes can be indirectly controlled by limiting the investigated index candidates, e.g., the index width or naively enumerated combinations (only for *AutoAdmin* and *CoPhy*). The number of cost requests and, thus, evaluated configurations is the main cost factor for selection algorithms [PDA07].

Moreover, *DB2Advis* allows specifying the time for random substitutions after a preliminary index selection was conducted (see Section 3.1.5). We chose algorithm settings so that runtimes did not become too large. We report runtime details throughout Section 4.2 and a detailed cost breakdown in Section 4.2.4.

[9]Note, we do not report actual runtimes for physically created index selections, because they *could arbitrarily differ* from the estimated costs. Thus, they are not adequate for assessing the algorithms, which *only consider estimates* during selection.

[10]*AutoAdmin* also supports storage budgets, see Section 3.1.3 for further details.

4.1.4. Evaluation Platform

This section presents the evaluation platform we developed to automate the comparison of index selection algorithms. Besides enabling the reader to reproduce all results and retrace the algorithm's selection steps completely, the platform facilitates the future integration of additional algorithms, workloads, or database systems.

Implementation

Our implementation's target is to automate the evaluation of different algorithms, settings, and workloads. The open-source[11] evaluation platform is implemented in Python 3. The automation includes the setup, i.e., data generation and loading, query generation, evaluation of different algorithm parameters, and collection and summary of the results.

The centerpieces of the implementation are the compared selection algorithms that are available as part of the platform's open-source repository. We implemented all algorithms mentioned in Chapter 3 (except *Dexter*), including unit tests, based on the original descriptions and tried to keep them as close to the original as possible. For *Dexter*, we used the publicly available implementation [Kan17a] and embedded it into Python to offer the same interfaces for all algorithms. We implemented the input generation for ILP-based approaches with Python, the model in AMPL [FGK03], and used the Gurobi[12] solver (v8.1.0) for solving the ILPs.

The `CostEvaluation` class implements the determination of query costs for given index configurations. This class can be used transparently by the algorithms, i.e., algorithms do not have to consider how the costs are determined, e.g., whether hypothetical indexes are used or not. The `CostEvaluation` automatically takes care of creating and dropping (hypothetical) indexes based on the current and demanded index configuration. The `CostEvaluation` also handles pruning and caching of cost estimations. Objects of the `Query` and `Table` classes can be accessed to generate index candidates of varying widths or for specific queries.

The `DatabaseConnector` builds an abstraction layer for different database systems. It provides a consistent interface for using what-if capabilities and hides different SQL dialects. Currently, the platform contains connectors for PostgreSQL and SAP HANA, while the latter does not support hypothetical indexes. Therefore, the algorithms must obtain the costs by creating actual indexes and evaluating queries, which is unfeasible for massive workloads and numerous index candidates.

Configuration options, e.g., dataset scale factors, the utilized database system, parameter values for the algorithms, and whether and how often the workload is executed

[11]Index selection evaluation platform on GitHub: `https://git.io/index_selection_evaluation`
[12]Gurobi solver: `https://www.gurobi.com`

with the calculated index selection, can be controlled via a JSON configuration file. The platform's source code repository contains example configuration files as well as the JSON files for the experiments presented in Section 4.2.

Optimizations for Hypothetical Indexes

Obtaining cost estimates for queries given a particular index configuration makes up a large part of the total runtime of index selection algorithms because it requires optimizer invocations as well as inter-process (or network) communication. In the literature, the reduction of optimizer invocations is often a primary focus, e.g., with *atomic configurations* [FST88] as part of the *AutoAdmin* body of work [CN97, p. 3].

Our `CostEvaluation` avoids unnecessary optimizer calls and maintains a (cost estimation) cache to facilitate efficient evaluations. If no index of a given configuration is applicable for a query, the costs without indexes are returned. Additionally, given a query and all possibly applicable indexes of a requested configuration, the cost estimation cache stores the retrieved cost estimates. Note that multiple cache entries (with different sets of possibly applicable indexes) may exist for the same query. We do not limit the cache size and reset the cache at the beginning of each algorithm evaluation to achieve identical start conditions for all algorithms. Besides these optimizations, we did not implement any other techniques to reduce the number of optimizer calls.

4.1.5. Experimental Setup

All index selection experiments are executed on an AMD EPYC 7F72 with 24 cores (base clock: 3.2 GHz, boost: up to 3.7 GHz) running PostgreSQL 12.5 and Python 3 on Ubuntu 20.04 (Kernel 5.4.0-105). For the vital part of what-if optimization and hypothetical indexes, we employ HypoPG [Rou15], commit `238cca5`. For the following experiments, we use PostgreSQL's default index type [Theb], a non-covering B-tree. However, our evaluation platform is not conceptually limited to non-covering B-tree indexes. For this chapter's experiments, f_n is set to 1 for all queries. Hence, all benchmark queries have the same weight. Before the experiments are started, all indexes — including schema-defined primary keys are removed — such that the algorithms have the chance to determine all reasonable indexes.

Algorithm Parameters. For the upcoming experiments, we use the following default parameters. The parameter impact is investigated later in Section 4.2.5. The admissible index width W_{max} was set to 2. Thereby, also the initial index candidate set contains all syntactically relevant candidates of width 2. The time for *DB2Advis*' `TRY_VARIATION` step was set to 0. While this step is part of the original algorithm, random exchanges could be added to all algorithms; we see more value in evaluating the solution quality of the core algorithms instead of solutions affected by chance. For *AutoAdmin*, the number of indexes selected by a naive enumeration was set to 1 to prevent large runtimes. The *DTA* algorithm is designed to be interruptible at *any time* and still deliver acceptable solutions. We granted a maximum runtime of 60 minutes.

Evaluated Metrics. The experiments presented in Section 4.2 focus on and discuss the following metrics:

- The *Relative workload costs* (*RC*) describe to which degree the resulting index configuration I^* decreases the costs for processing the workload. The costs are calculated relative to executing the workload without any indexes or primary keys, $C(I^*)/C(\emptyset)$. The relative workload costs express the *solution quality.*
- *Storage consumption*: the sum of the storage space necessary to accommodate all indexes of I^*, i.e., $M(I^*)$. We always denote the actual storage consumption, which may be equal or lower than the provided budget, in the upcoming evaluation.
- *Granularity* refers to the number of identified solutions in a budget range. For example, while some algorithms are able to identify solutions I^* whose storage consumption almost fully exploits any provided budget, other algorithms might find only a single solution for storage consumptions between 2 and 5 GB.
- The *index selection runtime* to determine I^*. Note, this does not include preparation or learning times.

4.1.6. Limitations

The main limitations of our experimental evaluation are twofold. First, standardized benchmark workloads cannot fully reproduce the challenges posed by real-world workloads [Vog+18]. Our approach to evaluating all index selection algorithms with workloads derived from *three* different benchmarks (cf. Section 4.1.1) on three datasets partially mitigates this limitation. In addition, the queries of the employed JOB are realistic and operate on the real-world IMDB dataset [Lei+15]. Despite not being real-world datasets, the utilized workloads are challenging for the investigated index selection algorithms, as demonstrated in the following section. Thereby, the conducted experiments allow drawing conclusions and generating insights.

Second, since most of the evaluated algorithms are related to commercial tools, no public implementations are available. The original publications only explain the procedures and do not contain (the complete) source code. Thus, we were required to reimplement six of the seven evaluated index selection algorithms. We followed the specifications presented in the original publications as close as possible. Besides, we communicated with the authors of five of the algorithms clarifying implementation details to ensure appropriate reimplementations. We conducted twofold code reviews and made all source code publicly available, allowing other researchers to assess our reimplementations, too.

4.2. Evaluation

Now, we evaluate the seven previously surveyed algorithms for the TPC-H, TPC-DS, and JOB workloads. We present results for each of the workloads (Sections 4.2.1 to 4.2.3) separately regarding the quality of the identified solutions and their corresponding runtime. Both solution quality and runtime are displayed together with the storage consumption of the identified final index configurations. Based on the solution quality experiments, the granularity of the identified solutions is discussed, too.

Section 4.2.4 discusses the impact of hypothetical indexes and cost requests on algorithm runtimes. In the end, Section 4.2.5 sheds light on the influence of different algorithm parameters, i.e., the index width, *DB2Advis*' time for random variations, and *AutoAdmin*'s number of naively enumerated indexes.

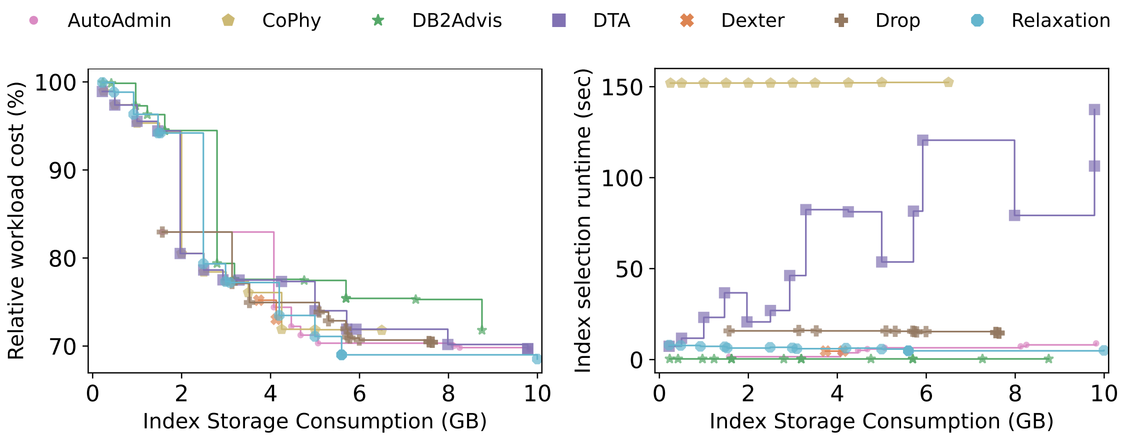
4.2.1. TPC-H

The TPC-H measurements are based on a scale factor of 10. For Figures 4.1(a) and 4.1(b), we excluded the queries 2, 17, and 20 because due to subqueries, their estimated costs are orders of magnitude higher than for TPC-H queries on average in PostgreSQL (see Figure 4.2 that contains all queries). Without the exclusion, these three queries dominate the costs of the entire workload, thereby rendering the index selection problem less complex because an index that decreases the cost of at least one of these queries would consistently outperform indexes for other queries by orders of magnitude.

Figure 4.1(a) depicts the performance of the solutions identified by the investigated index selection algorithms for budgets from 0 to 10 GB. Each marker indicates an index combination identified by an algorithm for a particular storage budget. First, it becomes apparent that most algorithms (all except for *Relaxation*) do not fully utilize the maximum budget of 10GB because they cannot determine indexes that reduce the workload cost any further and fit the budget. For example, *DB2Advis* uses only ≈ 9 GB.

Second, there is a structural difference between the algorithms that aim to purely minimize the costs and use the maximum number of indexes as a constraint and the budget-based algorithms that aim to minimize costs per storage. While the latter identify indexes for low budgets, starting with a few hundred megabytes, *Drop* and *AutoAdmin* need roughly 2 GB to add the first index because they start by adding the index with the largest cost improvement, independent of its size.

Moreover, looking at both the workload cost in Figure 4.1(a) and the runtimes in (b),



(a) Workload processing costs for index configurations with increasing storage consumption.

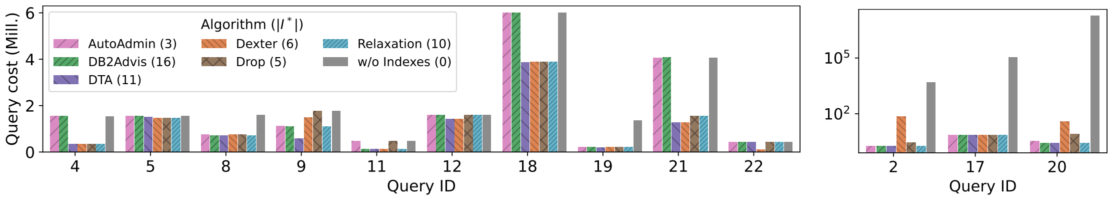
(b) Observed runtime for determining index configurations with increasing storage consumption.

Figure 4.1.: TPC-H benchmark (SF 10) on PostgreSQL, storage budgets from 0 to 10 GB. (a): estimated workload processing costs (relative to estimated costs without indexes), i.e., solution quality. (b): algorithm runtime including cost requests and index simulations. The queries 2, 17, 20 were excluded.

it is not trivial to determine a winner. For multiple budgets, *Relaxation* identifies the best solutions with acceptable runtimes. While the performance for many algorithms in the range of 2 to 6 GB is close, some algorithms achieve significantly better performance than others, e.g., *DTA* vs. *DB2Advis* at 2 GB. Due to the employed optimization target, *AutoAdmin* and *Drop* need large budgets to find their first index, but this index is a substantial improvement over all other algorithms.

Note, *DTA* would still identify usable, possibly equivalent, solutions if its runtime would be limited more drastically. *DTA*'s *anytime* functionality will be investigated later in Section 5.3. *DB2Advis* has a constantly low runtime and still finds acceptable solutions. The low runtime is caused by its modus operandi (see Section 3.1.5). Most other algorithms generate many combinations and check their impact with what-if optimization calls. At the same time, *DB2Advis* only issues a fixed number of $2 \times N$ cost requests, where N refers to the number of queries. *Drop* shows an almost constant runtime since, for each round, it starts with the same extensive set of index candidates, drops them one by one, thereby behaving similarly for every case. *Dexter's* runtime is constant because it does not depend on a budget but the number and complexity of queries. While its solution quality is close to the best, it cannot produce fine-grained solutions and identifies only two solutions for all evaluated budgets.

Finally, Figure 4.2 shows the costs that are achieved for all of TPC-H's queries on a per-query basis with each algorithm's best index combination that does not consume more



(a) Nonexpensive queries. (b) Expensive queries.

Figure 4.2.: Estimated query processing costs for selected queries of the TPC-H (SF 10) benchmark on PostgreSQL. Queries 1, 3, 6, 7, 10, 13, 14, 15, and 16 are omitted as their costs were only marginally affected by indexes for a budget of 5 GB. Nonexpensive queries depicted with linear, expensive queries with log scale; y-axis units are identical. I^* refers to the final index configuration.

than 5 GB[13] of storage to better understand the mechanics of the different algorithms. The figures indicate that most algorithms, except for *Dexter* and *Drop*, manage to identify the most important indexes for the expensive queries shown in Figure 4.2(b). If we also consider Figure 4.2(a), it becomes apparent that no algorithm's configuration is best for all queries, but *DTA* almost achieves this goal. While *Dexter* finds the best index configuration for query 22, its performance differences for the expensive queries (Figure 4.2(b)) are significant.

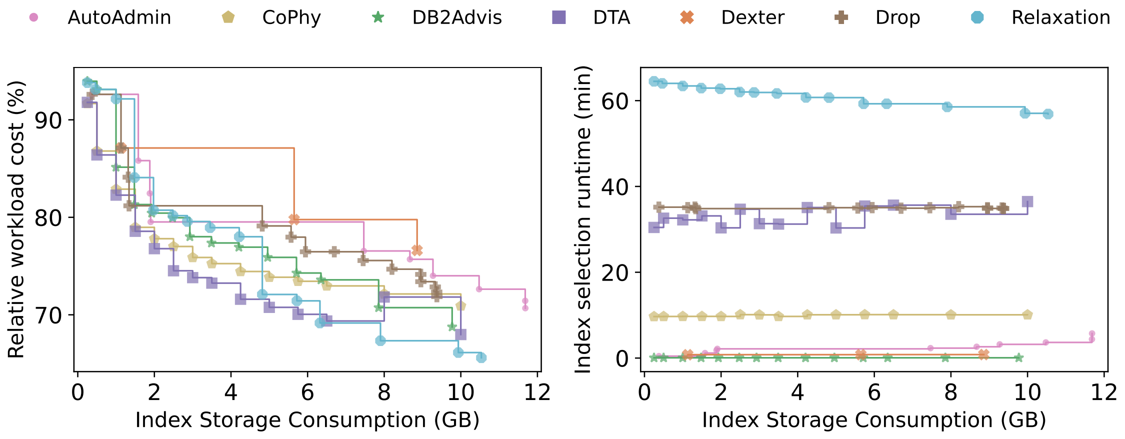
The per-query evaluation offers another interesting observation. Now, *AutoAdmin*, one of the top performers for a budget of 5 GB in Figure 4.1, shows relatively weak results for queries 4, 18, and 21. This effect is probably due to the different workloads, now including queries 2, 17, and 20. This observation is the first indicator that the performance of index selection algorithms strongly depends on the workload at hand.

4.2.2. TPC-DS

The TPC-DS measurements are also based on a scale factor of 10. For all TPC-DS experiments, we excluded the queries 4, 6, 9, 10, 11, 32, 35, 41, and 95 because, again, these queries have the potential to distort a fair assessment of index selection algorithms. The higher complexity of the TPC-DS benchmark compared to TPC-H emphasizes the strengths and weaknesses of the algorithms further.

Figure 4.3(a) shows the solution quality of the investigated index selection algorithms for memory budgets from 0 to 12 GB. In contrast to the TPC-H measurements, the differences for budgets between 2 and 10 GB are more significant. The solutions identified by the algorithms vary to a larger degree because there are more (indexable) attributes and more queries that benefit from indexes. These characteristics open up many more

[13]Our platform allows generating the corresponding charts for other benchmarks and budgets.



(a) Workload processing costs for index configurations with increasing storage consumption.

(b) Observed runtime for determining index configurations with increasing storage consumption.

Figure 4.3.: TPC-DS benchmark (SF 10) on PostgreSQL, storage budgets from 0 to 12 GB. (a): estimated workload processing costs (relative to estimated costs without indexes), i.e., solution quality. (b): algorithm runtime including cost requests and index simulations. The queries 4, 6, 9, 10, 11, 32, 35, 41, and 95 were excluded.

indexing opportunities. Hence, the index selection problem becomes more complex, which can also be observed by the elevated runtimes depicted in Figure 4.3(b).

Drop and *AutoAdmin* identified the first indexes for relatively large budgets, around 2 GB, for the TPC-H experiments. For TPC-DS, they identify beneficial solutions much earlier because there is no single large dominating table, such as the `lineitem` table for TPC-H, in the TPC-DS dataset. Thus, impactful indexes do not have to be that large.

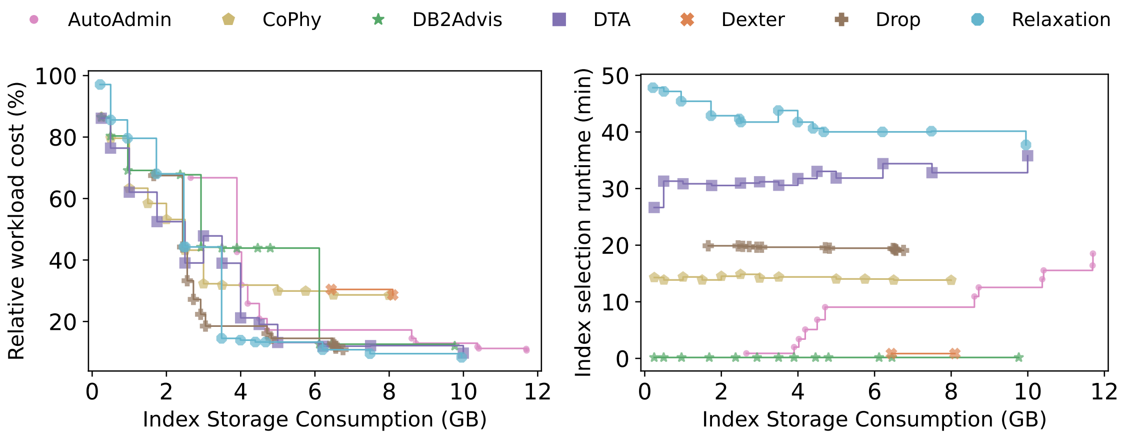
Especially for small budgets, the realized cost improvements are close to each other. For budgets of 2 GB and above, differences start to become pronounced. *DTA* and *CoPhy* take the lead and generate the best solutions with increasing benefits for *DTA*. In contrast to *Drop*, *Dexter*, and *AutoAdmin*, they keep adding relatively small indexes to the solution for a larger budget range. For instance, *CoPhy*'s modus operandi causes this behavior: the index with the largest overall benefit-per-space ratio is added for each step. For budgets larger than approximately 6 GB, *Relaxation* supersedes *DTA* and identifies the best overall solution, too.

In the following, we also consider the runtime dimension for the TPC-DS measurements with Figure 4.3(b). *DB2Advis* achieves again an almost constant runtime caused by its functioning and offers a good tradeoff between runtime and solution quality. While *DB2Advis* performs worse than the other algorithms for medium-sized budgets, its solution quality is comparably good for small and large budgets. The runtime is in the range of seconds compared to minutes for *Drop*, *Relaxation*, *DTA*, *AutoAdmin*, and *CoPhy*. *Relaxation* shows the longest runtime caused by its functioning that requires many transformations to push the storage consumption below the given budget.

The solution quality of *Dexter*'s ready-to-use approach is again not particularly bad *if* a solution is identified. However, its lack of fine-grained solutions becomes more pronounced for the TPC-DS workload for which many, comparably small, indexes can have a substantial impact.

4.2.3. Join Order Benchmark

The Join Order Benchmark measurements are depicted in Figures 4.4(a) and (b). Clearly, the achieved relative cost reduction by more than 5 times is significantly higher compared to the TPC-H and TPC-DS experiments above.



(a) Workload processing costs for index configurations with increasing storage consumption.

(b) Observed runtime for determining index configurations with increasing storage consumption.

Figure 4.4.: Join Order Benchmark on PostgreSQL, storage budgets from 0 to 12 GB. (a): estimated workload processing costs (relative to estimated costs without indexes), i.e., solution quality. (b): algorithm runtime including cost requests and index simulations.

For TPC-H and TPC-DS, *DTA* and *Relaxation* find the solutions with the lowest workload costs for most of the examined budgets. However, medium-sized budgets (from 2 to 3.5 GB) are not handled well by these two algorithms for the JOB experiment because fine-grained solutions are missing. *Drop* identifies many small indexes that decrease the workload cost significantly for medium-sized budgets while it completely lacks solutions for small budgets similar to *AutoAdmin*. Again, this behavior is due to their focus on pure cost reduction and the utilized constraint, i.e., the number of indexes.

CoPhy, *DB2Advis*, and *DTA* find the best solutions for small budgets. Nevertheless, for larger budgets, the lack of multi-attribute indexes and the limit of two indexes per query (both limitations due to complexity) become apparent for *CoPhy.* At $\approx$ 3.5 GB, *Relaxation* starts identifying the best solutions up to the largest evaluated budget. As for TPC-H and TPC-DS, *Dexter*'s solutions are coarse-grained.

Figure 4.4(b) shows similar results compared to the algorithm runtimes for TPC-DS. However, there are a few interesting aspects to note here. While *AutoAdmin* multiplies its runtime, *Drop*'s almost halves, and *Relaxation*'s decreases by almost a third. The overall lower number of attributes (Table 2.2) can explain the last two observations. Thereby, *Drop* and *Relaxation* generate fewer index candidates, resulting in fewer what-if optimizer calls per step. The increase in algorithm runtime for *AutoAdmin* can be explained by a higher number of beneficial single-attribute indexes compared to the TPC-DS.

4.2.4. Cost Breakdown and Cost Requests

Most of the runtime of what-if-based index selection algorithms is usually not spent on the algorithm logic, but on cost estimation calls to the what-if optimizer [DPA11; PDA07]. Usually, algorithms request a cost estimate for a particular query and a given index combination from the (what-if) optimizer. These requests are expensive, but the estimated costs remain the same if neither the query and index combination nor the underlying data change. Therefore, most algorithms cache cost estimation calls. With the developed evaluation platform, all algorithms use the same cost estimation implementation, and hence, the same caching mechanisms. However, the different strategies of the investigated algorithms result in varying cost request patterns and different cache opportunities, which is demonstrated in Table 4.1 for a budget of 5 GB for the TPC-DS benchmark. The platform allows generating this table for further budgets, configurations, and benchmarks.

The runtimes vary significantly, ranging from seconds to more than an hour, caused by different underlying strategies. Factors — besides the modus operandi of the algorithm — that influence the runtime include the number of evaluated configurations, simulated (hypothetical) indexes, cost requests, and cache rates. Table 4.1 indicates that the interplay of these factors is responsible for the resulting runtime.

Generally, for all algorithms, most of the runtime is spent on cost requests (what-if optimization calls). *Drop* is the only exception to this observation. However, most algorithms achieve high cache rates caused by the fact that they repeatedly evaluate similar configurations. Note, retrieving costs from the cache does not come for free since the configuration must be looked up in the cache, and its cost must be obtained.

DB2Advis and *Dexter* evaluate only two configurations. This behavior leads to few cost requests and low runtimes. In contrast to other algorithms, the number of evaluated

Table 4.1.: Algorithm cost breakdown for the TPC-DS (SF 10) benchmark. Storage budget of 5 GB. *Configs* are the number of uniquely evaluated configurations. *Index Simulations* refer to the number of non-unique created hypothetical indexes. *Simulation* and *Costing* refer to the share of runtime that was consumed by index simulations or cost requests. *Naive-2* relates to *AutoAdmin* configured to select two indexes with naive enumeration. *Dexter*'s original implementation does not provide all runtime details.

Algorithm	Configs	Index Simulations	Cost requests			Runtime		
			Total	Non-cached	Cache rate	Total	Simulation	Costing
AutoAdmin	129	10 991	33 851	11 676	65.5%	2.1m	2.0%	95.9%
Naive-2	816	73 504	240 441	73 440	69.4%	15.3m	2.0%	66.5%
CoPhy	3 983	3 982	394 317	52 177	86.8%	10.1m	0.6%	94.9%
DB2Advis	2	7 179	180	180	0.0%	0.1m	24.0%	58.7%
DTA	1 442	25 812	1 650 510	129 811	92.1%	32.2m	0.4%	87.2%
Dexter	2	3 982	180	180	0.0%	0.4m	n/a	n/a
Drop	203	29 144	2 601 450	18 348	99.3%	35.0m	0.6%	19.7%
Relaxation	1 898	51 680	2 982 690	170 863	94.3%	60.7m	0.4%	66.6%

configurations does not depend on the number of candidates. For wider indexes, runtimes can increase dramatically, as discussed below.

Drop and *Relaxation* diverge in the number of evaluated configurations but generate the largest number of cost requests. They are the two slowest approaches. In contrast to all other approaches, they follow the same general concept: they start with an extensive configuration and reduce it until it fits the given budget. For realistically-sized budgets, this behavior leads to long runtimes.

Besides, according to Table 4.1, the impact of index simulation is almost negligible. Naive-2 simulates more than 70 000 indexes, which is only responsible for 2 % of its runtime. Interestingly, for the TPC-DS benchmark, more than 99 % of *Drop*'s cost requests are cached. This fact also explains its constant runtimes in the aforementioned experiments. Most cost requests have already been sent in the previous round due to its modus operandi, i.e., dropping all remaining index candidates in each round.

Expensive Cost Requests. Furthermore, it is essential to note that not all cost requests are equally expensive. Query planning time depends on the query complexity and the number of available (what-if) indexes. Few available indexes lead to planning times in the range of milliseconds for our implementation. *AutoAdmin*, *CoPhy*, *DTA*, and *Drop* usually request costs for small index sets of a few dozen indexes at maximum. However, the modi operandi of *DB2Advis* and *Relaxation* can lead to large index sets for the cost requests because costs are evaluated for all possibly applicable indexes per query. In particular, *DB2Advis* does not issue many but large, more expensive cost requests.

Table 4.2 shows the cost request time for two complex queries of the TPC-DS benchmark. Cost estimations for large index combinations can take prohibitively long, which becomes especially pronounced when wide indexes ($W_{max} > 3$) are searched.

Table 4.2.: Query cost estimation request durations including index simulations for two TPC-DS queries; DNF: the cost request duration exceeded 30 minutes.

Index width	Relevant indexes		Time	
	Query 13	Query 64	Query 13	Query 64
1 attribute	22	49	13ms	12ms
2 attributes	132	287	97ms	44ms
3 attributes	870	1 889	33s	5s
4 attributes	5 910	14 393	DNF	231s

4.2.5. Algorithm Parameter Influence

For the previous experiments, some algorithm parameters were fixed. In this subsection, we investigate the influence of the allowed index width W_{max}, the time for *DB2Advis*' `TRY_VARIATION` step, and *AutoAdmin*'s number of indexes that are selected by naive enumeration. The impact of *DTA*'s runtime limits is evaluated later in Section 5.3.3. The experiments were conducted with TPC-DS (scale factor 10) for budgets between 4 and 8 GB. Other benchmarks did not yield substantially different results.

Index Width. In our evaluations, a large number of attributes per index does not significantly influence the investigated workloads on PostgreSQL. The impact could be higher for real-world workloads, e.g., enterprise systems utilize wide primary keys of up to 16 attributes [Fau+16]. Additionally, more sophisticated query processors might use wide indexes more efficiently. The maximum index width selected by algorithms during our experiments was 6. For example, *AutoAdmin*'s performance improves by 1 to 3 % when the index width is increased from 1 to 2 or 3. Improvements by wider indexes are below 1 %. For *DB2Advis*, we found an anomaly: the performance improved by about 1 % when the index width was increased from 1 to 3 but dropped by $\approx$ 5 % when set to 2.

DB2Advis' runtime is in the range of seconds for $W_{max} \leq 3$ but is not feasible for larger numbers due to expensive cost requests for large index combinations as discussed above. The runtimes of *AutoAdmin* and *DTA* increase with wider indexes, e.g., by a factor of 2 - 3 from single- to two-attribute indexes, because for each admissible index width, the enumeration steps are executed for all queries. The increase in runtime declines over time because the candidate volume decreases per round since candidates are only drawn from previously beneficial indexes (Section 3.1.3).

DB2Advis — Try Variations. The measurable impact of *DB2Advis*' `TRY_VARIATION` step is often small. According to our experiments, variation achieves improvements of about 1 % in workload cost, even when the core algorithm ran in approximately 1 and the variation in 30 seconds. For massive candidate sets, e.g., caused by many permutations when wide indexes are considered, variations can only be effective with a long runtime because chances that beneficial candidates are becoming part of the final combination decrease with a larger population.

AutoAdmin — Naive Enumerations. In our experiments, the number of indexes selected by naive enumeration affects the approach's solution quality only marginally. Sometimes, we even observed better results for smaller numbers. However, the runtime is significantly affected. For the conducted benchmarks, the runtime increased by factors between 3 and 10 when the indexes selected by naive enumeration were increased from 1 to 2. More indexes could not be selected via naive enumeration in an acceptable time.

4.3. Conclusion

Our evaluation of index selection algorithms has shown that there is no general answer to the question of the best algorithm. Instead, it depends on the scenario and multiple factors, such as the provided budgets, the workloads, and the acceptable runtime.

4.3.1. Insights

In the following, we summarize 10 general and 17 algorithm-specific properties to explain the performance differences observed in the experimental evaluation. These insights are generalizable to a certain degree. However, they might not be fully transferable to other systems that rely on different optimizers and execution engines than PostgreSQL.

General Insights. (i) The algorithms' performance is not always consistent. For different workloads or budget restrictions, different algorithms can perform best. (ii) The algorithm should be chosen based on the user's needs. Differences in runtime, solution quality, solution granularity, and multi-attribute index support are significant. (iii) There are two kinds of approaches: query-based (*DB2Advis*, *Dexter*) ones, which evaluate the benefit of all possible indexes at once, and index combination-based (*AutoAdmin*, *Drop*, *CoPhy*, *DTA*, *Relaxation*) approaches, which evaluate the benefit of comparably small index sets. Generally, the latter consider index interaction to a higher degree while query-based ones are orders of magnitude faster as long as the set of evaluated indexes per query is not too large (see Section 4.2.4). *Relaxation* combines both approaches. (iv) The cost of what-if calls or cost requests are not fixed; they depend on the query and evaluated index combination (see Section 4.2.4).

(v) The granularity of the identified solutions is fundamental, particularly if solutions for a specific, fixed budget are required. In such cases, the performance of solutions that lack granularity is suboptimal because such solutions do not adequately utilize the budget (cf. Figure 4.4(a)). (vi) The choice of the algorithm's constraint is essential. Algorithms that halt after a maximum number of indexes usually start with large indexes and do not find small indexes with a significant relative impact (cf. Figure 4.1(a)). (vii) Ordering index candidates by benefit per space instead of purely by benefit was efficient, especially for medium-sized budgets (see Section 4.2.2). This advantage could vanish for transactional workloads. Usually, benefit-per-space approaches favor configurations with many small indexes, which could lead to elevated maintenance costs and lock contention. Therefore, choosing the optimization target accordingly might be beneficial (see Table 3.1). (viii) Reduction-based approaches, i.e., *Drop* and *Relaxation*, become faster with larger budgets. In general, their runtimes are rather long for budgets that are not multiple factors larger than the dataset. (ix) Admitting wide indexes, e.g., $W_{max} > 3$, impacts selection runtimes and is challenging or even infeasible for existing approaches, see Section 4.2.5. (x) There is no dominating algorithm: none of the evaluated algorithms outperforms its competitors in terms of selection runtime and solution quality, cf. Section 4.2.

Drop. (i) The repetitive functioning causes a large number of cost requests, while it leads to the highest cache rates. (ii) An extension to support multi-attribute indexes would result in an infeasible number of index candidates requiring some kind of preselection.

AutoAdmin. The approach finds reasonable solutions, but three properties weakened the algorithm in our evaluation. (i) The number of indexes selected by naive enumeration only had a minor influence on the solution quality. However, the runtime impact was enormous (3-10×, see Section 4.2.5). (ii) *AutoAdmin* only optimizes for the pure index benefit ignoring the index size. Thus, a multi-attribute index with only slightly higher benefit than a single-attribute index but significantly higher memory consumption is selected. This behavior acts in opposition to storage efficiency. (iii) The number of beneficial single-attribute indexes impacts *AutoAdmin*'s runtime considerably.

DB2Advis. (i) *DB2Advis*' solutions are reasonable for high budgets, while the runtime is low. In such cases, the index combination is optimized for every query. (ii) For wider indexes, e.g., $W_{max} > 3$, the runtime increases drastically while the solution quality is not among the best. (iii) Random exchanges by `TRY_VARIATION` do not prove to be very effective (see Section 4.2.5). (iv) Due to its functioning, the approach may miss a locally inferior but globally superior index, e.g., indexes that are beneficial for many queries but never the best for an individual query.

Relaxation. (i) Large candidate sets and *Relaxation*'s transformation rules lead to an enormous number of evaluated configurations and, thereby, to the best consideration of index interaction. (ii) Thus, *Relaxation* performed best in the evaluated benchmarks for large budgets. (iii) The reductive functioning of the approach causes long runtimes for reasonable (budget does not exceed the dataset size by multiple factors) budgets.

CoPhy. (i) The solution quality depends on well-chosen initial candidates and suitable choices of index combinations per query. TPC-H and TPC-DS selections improved with more candidates (wider indexes), whereas JOB improved with a higher number of considered indexes per query. (ii) Cost requests dominate the runtime with more candidates. At the same time, the number of indexes per query impacts the solver runtime.

Dexter. Dexter identifies suitable index combinations and offers low, constant runtimes. However, the granularity of solutions is generally too coarse (cf. Figure 4.3(a)).

DTA. (i) The large number of seed configurations guarantees suitable configurations: *DTA*'s solutions, especially for small to medium-sized budgets, are usually among the best. (ii) Its runtime can be on par with most other approaches when the ability to interrupt the algorithm at *any time* is considered.

4.3.2. Summary

The seven surveyed and evaluated algorithms and three workloads based on the TPC-H, TPC-DS, and JOB are part of our evaluation platform. The platform is also used to evaluate our *Extend* and *SWIRL* index selection algorithms in Chapter 5. The platform's architecture facilitates its extension by other workloads, algorithms, or DBMSs. To enable simple reproducibility and traceability, the platform is open source, offers thoroughly tested algorithm implementations, and handles data and query generation as well as the setup and execution of experiments.

We compared seven index selection algorithms and evaluated the approaches regarding solution quality, runtime, and solution granularity with an extensible evaluation platform that promotes reproducibility. In addition, we evaluated the impact of the algorithms' parameters and presented a detailed cost breakdown to analyze the often enormous runtimes of the algorithms. Based on the experiments, we conclude that no existing index selection algorithm combines good *selection runtime* with outstanding *solution quality.*

5 Two Novel and Efficient Index Selection Approaches

We have shown in the previous chapters that various sophisticated index selection approaches exist. For complex workloads, these approaches produce solutions of high quality or provide low index selection runtimes. However, they fall short of striking the right balance between both metrics or cannot provide the demanded functionality, e.g., support for multi-attribute indexes.

This chapter presents two new efficient index selection approaches to overcome the limitations mentioned above. These approaches target different use cases and are designed to complement each other: *Extend*, presented in Section 5.1, focuses on determining better or equivalent solutions faster (within a few minutes) than other close-to-optimal approaches. On the other hand, *SWIRL* uses reinforcement learning (RL) to identify reasonable index selections instantaneously. *SWIRL*, which is explained in Section 5.2, requires a priori training to achieve rapid solution times. Thus, it is most reasonable to employ *SWIRL* in scenarios where many index selections for similar problems must be obtained. In contrast, *Extend* can be employed universally without any preparations. Both approaches are evaluated and compared with state-of-the-art approaches in Section 5.3.

Parts of this chapter have been published in two papers [KKS22a; SKB19]. The thesis author's detailed contributions to the first publication [KKS22a] were discussed at the beginning of Chapter 2. Regarding the second publication [SKB19], Schlosser is the paper's primary author. The thesis author supported the conceptualization of the presented approach. Furthermore, the author investigated and detailed related work, created a tuned C++ implementation, and co-authored the paper. The paper's evaluation was a collaborative effort of all authors.

5.1. Extend: Index Selections Based on Iterative Index Extensions

This section gives a detailed description of the heuristic multi-attribute index selection algorithm *Extend* (Section 5.1.1). Its iterative approach that constructs indexes stepwise by *extension* does neither limit index candidates nor index width. This procedure is a significant differentiator to existing techniques. Afterward, Section 5.1.2 classifies *Extend* according to the dimensions presented in Section 3.1.1.

5.1.1. Algorithm Description

Before we describe the details of *Extend*'s modus operandi, we first differentiate its candidate generation from other index selection approaches.

Candidate Generation

All of the investigated approaches in Chapter 3 explicitly receive or generate a set of index candidates as a first step. In a second step, these approaches aim to select a beneficial subset of these candidates. The size of the candidate set is essential: if candidates are excluded a priori, the achievable solution quality might be affected [SKB19]. If, on the other hand, the set of candidates is too large, the index selection runtime might grow to an unacceptable amount [SKB19], as shown in Section 4.2. Therefore, for reasonably sized problem instances, it is unrealistic that such approaches consider all potential candidates in the second step.

For the aforementioned approaches, the candidate set is fixed (and often reduced) for the second optimization step. In contrast, *Extend* works with an extension mechanism that enables new index opportunities with every iteration. The candidate set is not restricted a priori; it is, to a certain extent, constructed iteratively during index selection. Thereby, *Extend* is capable of considering all candidates but not all of them simultaneously. Instead, they are considered in a context-dependent fashion. This approach keeps the current candidate set at a reasonable size without potential performance degradations caused by candidate limits.

Extend's Modus Operandi

Next, we describe the details of *Extend*'s modus operandi. Algorithm 5.1 depicts that *Extend* adds an attribute from the set of index extension elements (E) to the index selection (I^*) per iteration. During an iteration, there are two options: (a) the attribute to be added could extend the current index selection as a new single-attribute index.

(b) Alternatively, it could extend an index that is already part of I^*. In this case, the attribute is appended at the end of the index.

Extend selects the option that achieves the highest relative cost reduction, i.e., additional cost reduction per additionally required storage for the (a) new or (b) extended index. In other words, *Extend* tries to mimic the efficient frontier of the index selection problem in terms of cost reduction and size [SKB19]. Thereby, index candidates with the highest benefit-to-storage ratio are chosen first.

Algorithm 5.1 (Extend Algorithm)**.** *To determine multi-attribute index selections for a budget, B, we use the following iterative construction process. The notation for the index selection problem was introduced in Table 2.1 in Section 2.1.*

(1) Let the current index set be $I^* := \emptyset$ and let E be the set of possible index extension elements that holds all attributes accessed by the workload's queries, $E := \bigcup_{n=1}^{N} q_n$.

(2) For each possible index extension $e \in E$ and indexes of the current index configuration $i \in I^*$ consider the following alternative construction steps:

(a) Add e as a new single-attribute index if $I^* \cap \{e\} = \emptyset$ to generate an option $I^*_{\emptyset,e}$.

(b) Append e at the end of index i, effectively increasing the index width of i by one. This generates an option $I^*_{i,e}$.

These construction steps described in (a) and (b) produce a set of index configuration options $\tilde{I}$ with $|\tilde{I}| \leq |E| \times (|I^*| + 1)$. Each of the options in $\tilde{I}$ are duplicates of I^* complemented by a single extension based on the construction steps above.

(3) Only keep options that do not exceed the remaining budget: $\forall \tilde{o} \in \tilde{I} : M(\tilde{o}) \leq B$ to avoid unnecessary evaluations of such options.

(4) Select the option $\tilde{o}^* \in \tilde{I}$ that maximizes the ratio of benefit and additional storage consumption: $\tilde{o}^* := \arg\max_{\tilde{o} \in \tilde{I}} \{I(C(I^*) - C(\tilde{o}))/(M(\tilde{o}) - M(I^*))\}$ by letting $I^* := \tilde{o}^*$.

(5) Repeat Steps (2) to (4) until no further benefit can be realized, i.e., $C(I^*) - C(\tilde{o}) \leq 0$ for every option $\tilde{o} \in \tilde{I}$.

A visual example of the procedure is given in Figure 1 of the original publication [SKB19, p. 1241]. Algorithm 5.1 defines only the functioning of *Extend*. Intentionally, it does not specify implementation details, e.g., the underlying data structures. An open-source Python implementation of *Extend* is provided as part of the index selection evaluation platform [Kos+20c].

Extend's iterative approach focuses explicitly on index interaction by re-evaluating the benefit of index candidates for every pass of step (2). This procedure ensures that interaction is considered to a higher degree compared to other *greedy* approaches, such as *DB2Advis* [Val+00]. At the same time, the similarity of the evaluated configurations enables high cost request cache rates of over 90 % [Kos+20d]. For subsequent iterations, the selections of previous iterations are not discarded. Thereby, the number of index options available in a given iteration is not multiplied but only added up [SKB19].

Step (3)'s purpose is to ensure that the set of index candidates is kept to a minimum: candidates that would exceed the storage budget are not considered for the selection of the best configuration in step (4). In contrast to all surveyed approaches (Chapter 3), there is no general limit for the number of attributes used for multi-attribute indexes. The reason is that the index candidates are constructed flexibly on demand during the index selection process. Based on the insights from the previous experiments, this is a major advantage because the selection runtime of other approaches increases significantly if wider indexes are admitted. Larger widths, e.g., $W_{max} \geq 4$, are prohibitively expensive for competitors, as discussed in Section 4.2.5.

Algorithm Extensions

In the following, we discuss several opportunities to expand the functionality of Algorithm 5.1 and decrease its selection runtime.

Alternative Stop Criteria. First, alternative stop criteria can be implemented for our approach. For instance, *Extend* could stop after a certain number of construction steps or created indexes; see cost parameter L as introduced in Section 2.1. *Extend*'s approach is also suitable for adding time constraints similar to *DTA*. If the time constraint interrupts the procedure, *Extend*'s greedy modus operandi ensures that the best possible, in terms of benefit-to-storage ratio, index candidates have been selected up to the point where the selection process was interrupted. Additionally, *Extend* could be adjusted to stop when a predefined relative cost reduction has been achieved, e.g., if $C(I^*)/C(\emptyset) < 1 - 0.2$ for a cost reduction of 20 %.

Candidate and Option Exclusions. In general, *Extend* does not restrict the candidate set. However, if the selection runtime is required to be decreased, *Extend* offers the flexibility to exclude candidates by reducing the number of extension elements (E). If candidates are removed, fewer options have to be evaluated in the repeated executions of steps (2) to (4) of Algorithm 5.1. Hence, the index selection process is accelerated.

There are multiple options to exclude candidates. Either, only a fixed number of heuristically determined candidates could be allowed. For instance, the corresponding attribute's cardinality could serve as a heuristic to order the candidates accordingly. Alternatively, candidates could be excluded based on the relative benefit they provide in isolation, i.e., $\forall i \in I : (C(\emptyset) - C(\{i\}))/m_i$. Note that such exclusions can affect the achievable solution quality [SKB19].

Furthermore, database management system (DBMS)-dependent rules could prune options ($\tilde{I}$) to further reduce the number of evaluations in step (4). For instance, based on how the particular DBMS utilizes indexes for query processing, certain index options are not feasible. PostgreSQL 12.5 provides a typical example of that. The system only uses an index scan if the query accesses its leading attributes: an index `(A, B)` would not be utilized if only column `B` is filtered.

Minimum Cost Improvement. Toward the end of the selection process, only a small fraction of the initial budget is typically remaining. According to Algorithm 5.1, *Extend* will continue to select indexes until *no* further improvements can be realized or the budget is exceeded. For small remaining budgets, the selected indexes have a relatively small storage consumption and benefit. Thus, the selection process continues while the realized benefits are comparably small. Therefore, *Extend* can be adjusted to support a relative *minimum cost improvement*, min_{impr}, that must be achieved to prevent termination. The algorithm terminates if no index configuration $\tilde{I}$ is able to improve the cost accordingly: $(C(I^*) - C(\tilde{I}))/C(I^*) \geq min_{impr}$ for every possible $\tilde{I}$. This algorithm extension is implemented in our open-source version of *Extend*. The extension prevents the expensive low-return phase at the end of the selection process.

A large value for min_{impr} can drastically accelerate the runtime but harm the solution's quality. In contrast, a small value might not have the desired effect of decreasing the selection runtime at all. It is not trivial to determine the optimal value as it depends on multiple factors, e.g., the workload, the budget, and the admitted index widths.

Parameters

Our open-source implementation [Kos+20c] allows configuring the *minimum cost improvement* and the storage budget for *Extend*. By default, the *minimum cost improvement* is set to 0.3 %, which was experimentally determined to produce a reasonable tradeoff of speed and performance. Furthermore, the index width can be restricted to simplify the comparability with other approaches that require such a restriction.

5.1.2. Classification

According to the classifications presented in Section 3.1.1 and, in particular, Table 3.1, *Extend* is an *imperative* algorithm that *adds* indexes to an empty index configuration. *Extend*'s aims to reduce the relative costs, i.e., the optimization target is $\frac{Costs}{Storage}$. *Storage* is the intended constraint for the algorithm, but as discussed before, other constraints could be easily supported. At the time of writing, *Extend* is not related to any commercial index selection tool. In addition, multi-attribute indexes are supported without limitations. Index interaction is considered to a high degree (+++) because of the iterative re-evaluation of the candidates' benefits. Finally, the implementation complexity of *Extend* is relatively low (+/++); the open-source implementation amounts to roughly 100 lines of code.

5.2. SWIRL: Selection of Workload-aware Indexes using Reinforcement Learning

In this section, we describe our index selection approach *SWIRL* that is based on RL. *SWIRL* incorporates knowledge of vast amounts of workloads during a preparatory training step. Thereby, it is able to match the performance of the best competitors while its runtime outperforms the fastest (but often not as good) algorithms. The focus on low index selection runtimes is justified if index configurations must be optimized for many — in some cases millions [Das+19] — systems, which is valid for cloud environments. Dynamically changing workloads [Ma+18] that demand quick reactions by reconfigurations further strengthen the need for fast approaches.

The idea to rely on *learning* for index selection is based on the insight that — in some scenarios — index selection problems are similar: for instance, in Software-as-a-Service (SaaS) scenarios, thousands of customers run similar workloads on similar schemata because such applications predefine schemata and workloads [Aba+19; Aul+09]. State-of-the-art index selection approaches do not utilize knowledge about such similarities.

If applied carefully, reinforcement learning-based approaches can effectively exploit these characteristics and the fact that massive amounts of training data exist [Din+19]: during training, an agent efficiently learns which indexes are beneficial under what circumstances for the predefined schemas. After training, and in contrast to state-of-the-art approaches, it does not need to enumerate possible solutions expensively. Instead, it *infers suitable indexes almost instantaneously* based on the previously accumulated knowledge. While other approaches must account for complex effects, e.g., index interaction, by costly and iteratively testing multiple configurations, our approach *internalizes* such effects during training. Naturally, to gain this knowledge, extensive training is required, which is justifiable if efficient index configurations can be determined quickly later when the model is frequently applied.

The remainder of this section is structured as follows. Section 5.2.1 provides the necessary background in the area of reinforcement learning. Afterward, Section 5.2.2 surveys existing RL-based index selection approaches and discusses their limitations. Section 5.2.3 describes our approach in detail and explains how its sophisticated workload model enables *SWIRL* to generalize to handle workloads that contain unseen queries and how we reduce training times with *invalid action masking.* In the end, Section 5.2.4 classifies *SWIRL* according to the dimensions presented in Section 3.1.1.

5.2.1. Background: Reinforcement Learning

Reinforcement learning covers a group of algorithms that aim at solving decision problems. Those problems are characterized by processes that repeatedly allow an agent to perform an action a_t from available actions A given a current state $s_t \in S$ [SB98]. The state describes the properties of the environment the agent is currently observing. Depending on the problem and the RL algorithm, A and S can be either discrete or continuous and have arbitrary dimensions. After performing the chosen action, a new state s_{t+1} is reached, and the process repeats. To provide agents with feedback on whether the action was chosen well, they receive a feedback signal, the reward r_t after each decision. The simulation might end at some point, leading to episodes of finite length characterized by the states, the agent's decisions, and the following rewards. The RL problem consists of finding the optimal policy, which maps states to actions, concerning the discounted future long-term reward given a specific starting state at time t:

$$G_t = \sum\nolimits_{k \geq 0} \gamma^k \cdot r_{t+k}. \tag{5.1}$$

The long-term reward is discounted to take into account that further progression in the decision process becomes less predictable. Low values of the discount factor $\gamma \in [0, 1)$ motivate the agent to act more greedily and consider possible long-term rewards less.

To implement an RL system, the agent needs to estimate the best expected value of G_t correctly. The Q-value is the expected value of G_t given a particular state s_t and the chosen action a_t, i.e.,

$$Q(s, a) = \mathbf{E}[G_t | s_t = s, a_t = a]. \tag{5.2}$$

Considering (5.2), the Q-value can be reformulated iteratively, as it incorporates the Q-value of the following state and its long-term reward, G_{t+1}. This fact allows to learn an estimator for the Q-value using the Bellman-update, given an observed state s_t, a performed action a_t, the observed reward r_t, and the follow-up state s_{t+1}:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \eta \cdot (r_t + \max_{a_{t+1} \in A} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)),$$

where $\eta \in (0, 1)$ is the learning rate. Higher η values increase the update size but decrease the stability of the estimation. In this setup, the agent keeps a matrix to store and update the Q-value for each observed combination of s_t and a_t. This representation allows it to derive a policy from the Q-estimation, by greedily choosing the action a that maximizes $Q(s, a)$ in the current state s. Actions are randomly chosen with a specified probability ε to ensure that the agent does not always choose the same actions (and leaves beneficial states unobserved). Moreover, instead of using tabular Q-values, a generic function approximator, such as an artificial neural network (ANN), can represent Q. In this setup, the difference between the network's estimation for the Q-value and the computed target value $r_t + \max_{a_{t+1} \in A} Q(s_{t+1}, a_{t+1})$ is minimized at each learning step.

This concept can be further expanded with policy gradient methods, which do not derive a policy from the learned value estimations but keep a parametric policy $a_t = \pi_\Phi(s_t)$. By adjusting Φ, the mapping from s_t to a_t is changed. In many cases, such a policy can be non-deterministic; instead of deciding for an action, it yields only the probability of performing an action: $\pi_\Phi(a|s)$. Adjusting Φ usually relies on the policy gradient theorem, which allows improving expected rewards via Φ only based on past observations. While *SWIRL* is not conceptually tied to a specific RL algorithm, we rely on Proximal Policy Optimization (PPO) [Sch+17] in this thesis. PPO offers the advantage of adjusting the learning rate automatically. Correctly adjusting the rate stabilizes the learning process by avoiding drastic changes in the agent's behavior and improves overall performance.

Action Masking. The index selection problem can cause states in which not all actions within A are applicable, i.e., not all indexes can be created, for instance, due to budget constraints. Such invalid actions could be modeled by assigning large negative rewards to such actions. *SWIRL* incorporates another feedback mechanism: action masking [HO20]. In this approach, the agent receives the allowed actions as input and is structurally enforced to select an element within this set. This technique shortens the learning process, as this decision element does not have to be represented within the agent's policy anymore. In addition, peak performance can be increased as well [HO20]. Such efficiency considerations are essential if the overall action space consists of many actions, but only a few of these actions are allowed in a given state. If action masking were omitted, the agent would have to explore many invalid actions to recognize which are allowed (depending on the state).

5.2.2. Existing RL-based Index Selection Approaches

Several RL-based index selection approaches have been presented as an alternative to existing rule- and enumeration-based heuristics. In the following, we describe these existing RL-based approaches, discuss their limitations, and highlight differences to our solution before we deduce requirements for an RL-based index selection approach that is competitive to state-of-the-art methods.

Table 5.1 compares the approaches across different dimensions: (i) whether or not multi-attribute indexes are supported, (ii) the constraint for the index selection process, (iii) the availability of an open-source implementation for further experiments and the reproduction of results. Moreover, (iv) we examined how the listed publications incorporate the representation of the workload at hand. We compare whether or not they can (v) generalize to be able to handle unknown workloads that include previously unseen query classes. Lastly, (vi) we mention how the approaches were evaluated.

Table 5.1.: Comparison of reinforcement learning-based index selection approaches. The corresponding publications are referenced in the detailed descriptions of the algorithms. Evaluation: * The queries only filter the `LINEITEM` table. † 12 randomly chosen query templates. ‡ 14 fixed query templates.

		NoDBA	DRLinda	Lan et al.	SMARTIX	DRLISA	SWIRL
(i)	Multi-attribute indexes	No	No	Yes	No	Unspecified	Yes
(ii)	Constraint	# Indexes	# Indexes	# Indexes[14]	# Steps	Improvement	Budget
(iii)	Implementation available	Yes	No	Yes	Yes	No	Yes
(iv)	Workload representation	Yes	Yes	No	No	Unspecified	Yes
(v)	Generalization	+	++	–	–	Unspecified	+++
(vi)	Evaluation	TPC-H*	TPC-H†	TPC-H‡	TPC-H	YCSB	TPC-H/DS, JOB

NoDBA. Sharma et al. were the first to present an RL-based index selection approach, *NoDBA*, capable of creating single-attribute indexes in 2018 [SSD18]. They evaluate their ideas with queries that filter TPC-H's `LINEITEM` table on multiple attributes. The model represents the workload as a matrix that contains the selectivity of every attribute for every query if the query is filtered on this attribute. This model makes the generalization to handle unknown queries theoretically possible even though it is not discussed in the publication. Varying frequencies of the queries are not considered. However, they could be modeled by repeatedly adding the same query to the state matrix, which would increase the size of the state matrix and potentially be unfeasible for larger workload sizes. Their approach does not consider other operators (apart from filter predicates) for index selection. Naturally, this is a significant limitation because other operators, e.g., joins and aggregates, are responsible for a large amount of the overall runtime in typical database workloads [Dre+20; Mül+15]. The authors provide an open-source implementation[15] of their work.

DRLinda. Sadri et al. present *DRLinda* for cluster databases [SGL20a; SGL20b]. While multi-attribute indexes are not supported, considering multiple instances in a database cluster sets a different focus, further complicates the problem, and is a differentiator to all other — including our — approaches. The workload is represented in three ways: (i) an access matrix that encodes for every attribute whether or not it is accessed in a query, (ii) an access vector that counts how often every attribute is accessed in total, and (iii) a selectivity vector that holds selectivity $= \frac{\text{\# unique values}}{\text{\# of rows}}$ for each attribute. The workload representation does not differentiate between accesses caused by different types of

[14]Due to space restrictions, budget results were not reported by the authors. Also, different budgets cannot be handled dynamically but the agent is trained for a fixed budget.

[15]Source Code for NoDBA [SSD18] on GitHub: `https://github.com/shankur/autoindex`

operators. This model could enable the agent to generalize to handle unknown workloads which is not discussed or evaluated in the original publications. There is neither a public implementation nor an evaluation that compares with state-of-the-art index selection approaches available. We have reimplemented *DRLinda* for our evaluations (Section 5.3) and included our implementation in *SWIRL*'s open-source repository [KKS22b].

Lan et al.'s Approach. Lan et al. propose another RL-based solution that allows identifying multi-attribute indexes [LBP20]. With increasing index widths (W), the number of candidates increases drastically; for workloads with hundreds of attributes, thousands of relevant 3-attribute indexes exist, cf. Section 2.2.1. The set of available actions usually comprises one action per index candidate for RL-based approaches. The authors propose five heuristic rules that serve as a preselection to reduce the number of index candidates (and consequently actions) and enable the selection of multi-attribute indexes. Excluding index candidates in advance may limit the potential solution quality [SKB19]. The model does not implement workload representation; in fact, the model is only *trained* for a single, fixed workload. For this reason, it cannot generalize and is only suitable for the exact training workload. The evaluation is conducted on 14 TPC-H queries, and the implementation[16] is publicly available.

SmartIX. Licks et al. present SmartIX: A database indexing agent based on reinforcement learning [Lic+20], provide an open-source implementation[17], and an extensive introduction to RL in the context of index selection. Their implementation cannot create multi-attribute indexes and does not include workload representation. In consequence, generalization is not possible. While the approach is the only one evaluated with the complete TPC-H benchmark and compared to a multitude of other approaches, the best-performing state-of-the-art approaches [Kos+20d] are not included. Their training procedure trades off long trainings (multiple days) against avoiding inaccuracies of cost estimations: SmartIX derives query runtimes from actual query executions instead of what-if-based estimations.

DRLISA. Yan et al. target NoSQL databases with *DRLISA* [Yan+21], which is a differentiator to all other approaches. It is the only approach that stops independently of the number of indexes or a storage budget. Instead, it terminates if no further performance improvement can be realized. According to the paper, the RL model takes a workload representation as input, but we could not find further details regarding the workload representation. Consequently, generalization may or may not be possible. The authors do not mention any publicly available implementation and evaluate their approach with the YCSB (Yahoo! Cloud Serving Benchmark) [Coo+10].

[16]Implementation of Lan et al. [LBP20] on GitHub: `https://github.com/rmitbggroup/IndexAdvisor`
[17]SmartIX's [Lic+20] experimental setup: `https://doi.org/10.5281/zenodo.3254967`

Research Gap: Requirements for Competitive RL-based Index Selection

Based on the motivation regarding fast selection runtimes, the limitations of state-of-the-art (Chapter 4), and RL-based approaches, we derive the following requirements for a competitive RL-based index selection approach. In terms of performance indicators, a potential approach should determine solutions that are competitive *(R-I)* with the solutions of the best state-of-the-art algorithms, e.g., *DTA*, while the computation of such solutions should be significantly faster *(R-II)*, for instance, comparable to the computation times of *DB2Advis*. At the same time, the training duration of the proposed model should not outweigh *(R-III)* the advantage gained during application time.

None of the existing RL-based approaches offers all the functionality commonly expected from index selection approaches, cf. Table 5.1. Multi-attribute indexes are widely deployed in real-world environments [Fau+16]. Thus, we require a powerful new RL-based solution to support multi-attribute indexes *(R-IV)*. Besides, the proposed solution should accept storage budgets *(R-V)* as a constraint. Supporting storage budgets allows more fine-grained solutions than targeting a fixed number of indexes, cf. Section 4.3.1. Moreover, an index selection algorithm based on RL should be able to generalize *(R-VI)* to handle unknown workloads, at least to a reasonable extent. Otherwise, retraining the agent for every workload change would be necessary, which is unrealistic and would make the approach much less attractive given the training durations for RL approaches.

To the best of our knowledge, there is currently no RL-based index selection approach that competes with state-of-the-art algorithms in terms of *R-I* to *R-III* and that unifies the functionality demanded by *R-IV* to *R-VI*. Lastly, the requirements should be evaluated on multiple complex analytical workloads against competitive state-of-the-art approaches. Such an evaluation has not been conducted for existing RL approaches.

5.2.3. Algorithm Description

We detail our approach *SWIRL* that tackles index selection with RL in the following. We start with a schematic overview. Afterward, we explain how we model the index selection problem in an RL-compatible fashion and how we handle the resulting complexity. For example, we describe how the environment's *state*, i.e., the queries, the budget, and the active indexes, is represented. Afterward, we discuss how the workload is represented, how changes of the index configuration are modeled as actions of the agent, and how the agent is rewarded. We detail our implementation and clarify the applied training procedure that enables efficient training and determining solutions of high quality.

Overview

Figure 5.1 depicts an overview of the entities involved in the RL-based index selection process and how they interact. The process is divided into three phases: (i) preprocessing: training and testing workloads are generated, index candidates are determined, and the workload representation model is prepared; (ii) training: the agent learns which indexes are valuable for the provided queries and schema as well as how these indexes interact, and (iii) application: the agent applies the trained model to determine indexes for provided workloads. During (ii) and (iii), the agent iteratively selects indexes for a given workload. This process represents the observed Markov decision process.

Preprocessing. ① The user, e.g., a database administrator (DBA), can specify a set of *representative*[18] *queries*. The potential impact of the specified query set is discussed later under *Workload Modeling and Query Representation*.

② Afterward, *index candidates* are generated based on the input *schema's* attributes and the set of representative queries. Restricting the set of index candidates to relevant ones is crucial [Kos+20d; LBP20] since index candidates correspond to the agent's actions, and too large action spaces complicate the agent's process of determining reasonable solutions and can increase training durations. At the same time, candidates should not be limited too much; otherwise, solutions of high quality cannot be determined [SKB19]. For this reason, not all but most attributes of the schema (and their permutations) should become index candidates. By default, our system generates all *syntactically relevant* index candidates (except for indexes on very small tables, less than 10 000 tuples). More filters can be added flexibly if the number of candidates should be further reduced. *Permutations* are generated according to a user-specified admissible index width (W_{max}). The candidate generation also prepares predictions of the index sizes for every candidate based on the estimates of a *what-if optimizer*.

③ Based on the set of representative queries, random *workloads are generated* as follows. A workload consists of (a subset of) the representative queries and assigns a random frequency to each query. Thereby, we create variability and ensure that the agent has to handle different query-frequency pairs during training. This procedure anticipates a wide variety of workloads later during application.

The created workloads are split into training and test sets. It is guaranteed that the test set contains only workloads that are not part of the training set. Besides, it is possible to specify that a certain number of the representative queries are not part of any training but only of test workloads to guarantee pure out-of-sample predictions. By doing so, we can investigate the agent's capability to generalize to handle unknown workloads.

④ Machine learning models are usually provided with numerical features. The *workload*

[18]Relying on representative queries is in accordance with other learned approaches [HBR20].

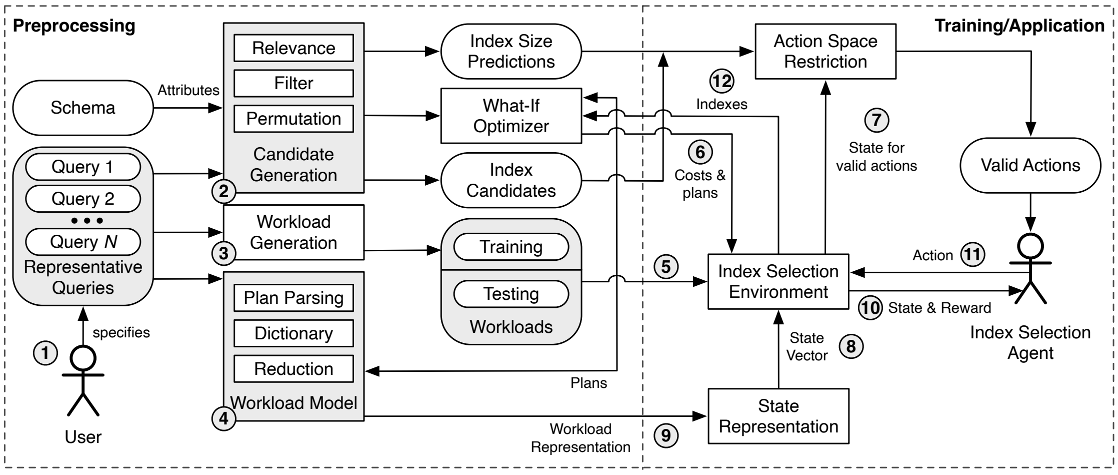


Figure 5.1.: Overview of *SWIRL*, our RL-based index selection approach. The approach is divided into three phases: preprocessing, training, and application.

model is responsible for creating workload representations, i.e., transforming or *featurizing* information about the queries of the current workload to a numerical representation that can be passed to neural networks. This process is crucial because, without a representation, unknown queries cannot be handled. Details are presented later under *Workload Modeling and Query Representation.*

Training. During training, the agent learns which indexes are beneficial in which situations by trying out many index configurations for different workloads in an efficient manner. Thereby, it can also implicitly discover and *internalize* complex effects like the relationship between different index candidates, i.e., index interaction, without these effects being explicitly specified or modeled.

The central components of the RL process are the *index selection environment* and the *index selection agent.* The agent is stateless and provides numerical actions that correspond to the creation of indexes in the environment. The stateful environment encapsulates the corresponding DBMS: it translates and implements the agent's actions in the DBMS, determines their consequences, rewards the agent, informs it about the environment's state, and abstracts other parameters, e.g., the budget.

⑤ The environment retrieves a new workload (③) for every training episode. Within one episode, the workload is constant. ⑥ Subsequently, the costs and plans for every query of the current workload are requested from the what-if optimizer given the current index configuration, which is usually empty for the first step of an episode. ⑦ Then, the agent's action space is restricted to contain only actions that are valid for the environment's current state. These restrictions are a major factor for converging as quickly as possible and allowing thousands of index candidates without limiting the candidate set a priori.

Actions can be limited based on the current workload, the remaining budget, or previous actions; see *Actions* below for details. This procedure is a significant differentiator to existing RL-based index selection approaches.

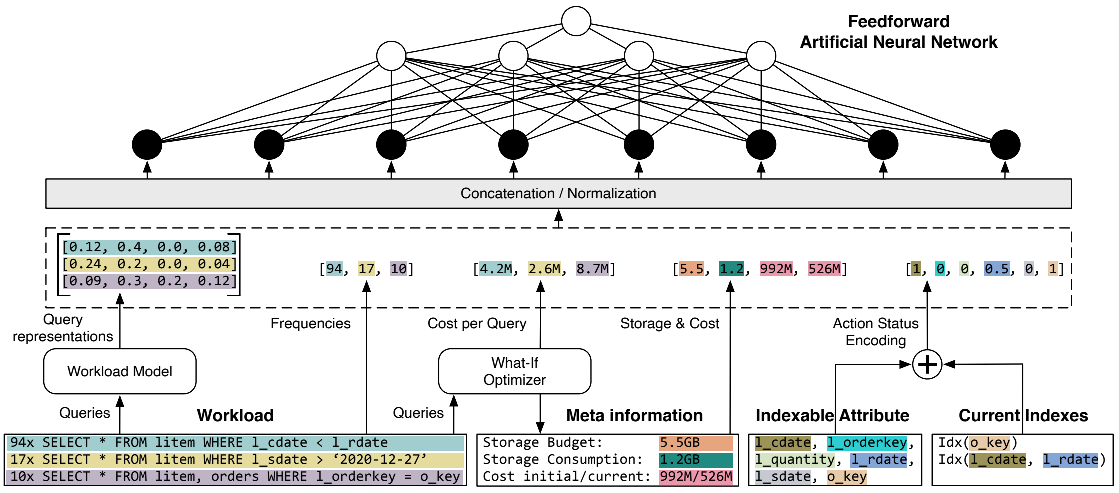
⑧ For the state representation, the current state of the environment, e.g., the remaining budget, current costs, and active indexes, is translated to numerical features so that it can be passed to the agent. This process includes retrieving ⑨ the workload representation from the workload model. ⑩ The environment's state and, if available, a reward are passed to the agent. ⑪ In return, the agent reacts with an action under consideration of the currently valid actions. ⑫ Then, the environment implements the agent's actions by creating indexes via the what-if utilities. The process continues at step ⑥ until there are no valid actions, e.g., if the budget is exceeded or a user-specified maximum number of iterations has passed. After a configurable number of steps, the ANN is updated to reflect the observations collected during the past steps.

Application. After training, our model is applied as follows. Instead of training workloads, at ⑤, the actual workload is received. Starting with an empty index set, the agent repeatedly *evaluates* the fitted ANN to subsequently select the action a_t ⑪ with the highest estimated reward (the best index) for a state s_t ⑩. Choosing a_t leads to a new state s_{t+1} for which a_{t+1} is determined until the budget is exceeded. This procedure can particularly be applied to any *unknown* workload. Even for previously unseen s_t, the ANN can be evaluated, and a_t can be efficiently obtained. Note that the application of our model is fast compared to state-of-the-art non-RL approaches since (i) interactions with the what-if optimizer are not necessary and (ii) due to the trained ANN, only simple evaluations remain to be performed.

State Representation

Figure 5.2 shows how we encode the index selection problem with a simplified sample workload. The sample contains 28 features distributed over 7 vectors enclosed by a dashed box. The number of necessary features to effectively represent a particular instance of the index selection problem largely depends on the number of query classes in the workload, their complexity, and the number of indexable attributes.

The cost and storage information contained in the state representation can be based on actual measurements or estimates obtained from a what-if optimizer. While the latter option produces only estimates, it is much faster. State representations must be updated during training for each of the agent's steps. Typically, there are thousands of steps during training. Therefore, we rely on the what-if-based estimations; actual execution time measurements are impractical. The information contained in the state representation can be divided into three aspects: the workload, meta-information (e.g.,



```
94x SELECT * FROM litem WHERE l_cdate < l_rdate
17x SELECT * FROM litem WHERE l_sdate > ‘2020-12-27’
10x SELECT * FROM litem, orders WHERE l_orderkey = o_key
```

```
Storage Budget:           5.5GB
Storage Consumption:      1.2GB
Cost initial/current:     992M/526M
```

```
l_cdate, l_orderkey,
l_quantity, l_rdate,
l_sdate, o_key
```

```
Idx(o_key)
Idx(l_cdate, l_rdate)
```

Figure 5.2.: State representation for a simplified example workload. The numbers are example values for demonstration purposes.

the budget), and the current index configuration. These aspects influence which indexes of all possible indexes are beneficial: for two different workloads, completely different index configurations might lead to the best performance.

Workload Representation. The workload representation must reflect which query classes are part of the workload as well as the frequency of these queries. For a workload with N query classes and a representation width of R (in Figure 5.2, $N = 3$, $R = 4$), the workload representation consists of (i) N numerical vectors of length R representing the queries' contents, (ii) a vector of length N with a numerical value for each query's frequency, and (iii) a vector that contains the estimated execution cost per query (N values) given the currently active index configuration. Representing the queries' contents is crucial for our approach and a major differentiator to other approaches; without it, the agent cannot learn about the structure of queries, recognize similarities, and, in the end, generalize to handle unseen queries.

Even though the ANN's structure is fixed, a model trained with workload size N can always be utilized to determine index configurations even if the workload size is different, e.g., $\tilde{N}$, during application time. If $\tilde{N} < N$, *padding* can be applied, i.e., query representation, frequency, and cost are set to 0 for $N - \tilde{N}$ queries. Otherwise, if $\tilde{N} > N$, a representative set of the workload with size N must be created. Such a set can always be found, e.g., by focusing on the most relevant queries and summarizing similar queries; *workload compression* [CGN02; Dee+20] has been effectively used for index selection in the past. Also, query clustering approaches that reduce the total query count exist [Ma+18]. Choosing N to be sufficiently large in the beginning can avoid the need for workload compression altogether or, at least, decrease the possible information loss caused by it. Choosing N sufficiently small allows for controlling the model's complexity.

The meta-information contains four scalar features regarding storage and workload cost: (i) a value for the currently specified[19] storage budget (B), (ii) the current storage consumption based on the what-if optimizer's index size predictions, (iii) the initial (without any indexes), and (iv) current cost (C) for executing the entire workload.

The current index configuration encodes for every indexable attribute whether an index is present or not. In the simplest case, with a maximum index width of $W_{max} = 1$, this information can be represented by a binary vector as every index can exist once or not at all. Encoding the index configuration is more challenging if multi-attribute indexes are admitted because there can be millions of index candidates. For example, for TPC-DS with $W_{max} = 4$, there are approximately 1.3 million index candidates according to Section 2.2.1. If we used a binary vector as above, we would increase the number of (very sparsely populated) features by the number of index candidates, which would be infeasible. Wide indexes occur in real-world systems [Fau+16]. At the same time, limiting W_{max} can harm performance [SKB19]. Thus, decreasing the dimensionality by limiting the number of candidates is not an option.

For this reason, we encode the information on the current index configuration for each indexable attribute separately to avoid large feature spaces: the value in the vector is incremented by $1/pos$ for every index that contains the corresponding attribute. *pos* refers to the position in the index. For instance, for `Idx(l_cdate, l_rdate)`, `l_cdate`'s *pos* is 1 and `l_rdate`'s *pos* is $1/2$. If a further index `Idx(c1, c2, c3, l_cdate)` would exist, `l_cdate`'s vector value would be $1.25 = 1 + 1/4$. Modeling the current index configuration like that — in contrast to a binary vector — implicates some loss of information: instead of encoding which exact indexes exist, we encode to which degree attributes are covered by indexes. However, according to our evaluations (Section 5.3), the agent is able to handle this encoding. In addition, the *index selection environment* still maintains the full information. Such information is, e.g., necessary for applying action masking, which will be discussed later.

Concatenation and Normalization. The vectors are concatenated and their contents are normalized before the presented state information is passed to the neural network. We use StableBaseline's `VecNormalize` class for these tasks; it normalizes values X to $\tilde{X}$ using their moving average $\bar{X}$ and the variance $\sigma^2(\cdot)$ as follows ($\varepsilon := 10^{-8}$ prevents possible divisions by zero):

$$\tilde{X} = (X - \bar{X})/(\sigma^2(\bar{X}) + \varepsilon)^{0.5}.$$

[19]Storage budgets are externally specified, e.g., by DBAs or external meta models.

Normalization is applied to improve the ANN's learning behavior. For large inputs, the used activation function `tanh` suffers from vanishing gradients. This effect can be avoided by normalization with zero mean and a variance of one [GBC16].

The number of features passed to the model (F), except for the meta-information (MI), is not fixed. F varies with the problem instance and the configuration at hand. It amounts to:

$$F = N \cdot R + N + N + MI + K. \tag{5.3}$$

The number of query classes in the workload (N) largely affects the size of the frequency, cost per query, and query representation vectors. Also, the representation width (R) is important for large workloads as it is multiplied with N. If the workloads that are passed to the model contain complex, unlike queries, R must be chosen large enough to capture different queries and their similarities properly. Lastly, large database schemas can result in hundreds of thousands of indexable attributes. Hence, we only consider attributes (cf. K) accessed by at least one query. Otherwise, it could result in too many features only to represent the current index configuration. For some Join Order Benchmark (JOB) experiments below, we worked with a workload size of 100 query classes and a representation width of 50, resulting in $100 \times 50 + 100 + 100 + 4 + 73 = 5\,277$ features for 73 indexable attributes.

Workload Modeling and Query Representation

One of the main aims of our approach — and a major differentiator to existing approaches — is to be able to handle query templates that were not part of the training workloads. Of course, these templates should not differ entirely but be reasonably similar to the query templates used during training. Thus, the set of training queries should roughly capture the workload expected at application time.

The desired capability to handle unknown queries requires us to set such queries into context with known ones. These capabilities add complexity to the model: details of the queries must be encoded such that the agent can incorporate them into its decision making, i.e., a detailed representation of the workload respectively of its queries is necessary. This representation must be compact enough to avoid feature explosion and contain enough detail to distinguish queries properly. Besides, computing the representation must not be too complex as it would further increase training durations.

Figure 5.3 depicts how our representation model is built and how representations are inferred. We build representative plans from the set of representative queries by utilizing the what-if optimizer and index candidates. The what-if optimizer is repeatedly invoked for every query to generate various plan alternatives based on different index configurations. Theoretically, the query representation could be built entirely on the queries' SQL strings. However, execution plans contain more details, information about index usage, and might change with the agent's actions, i.e., index decisions. The representative plans are passed to the representation model.

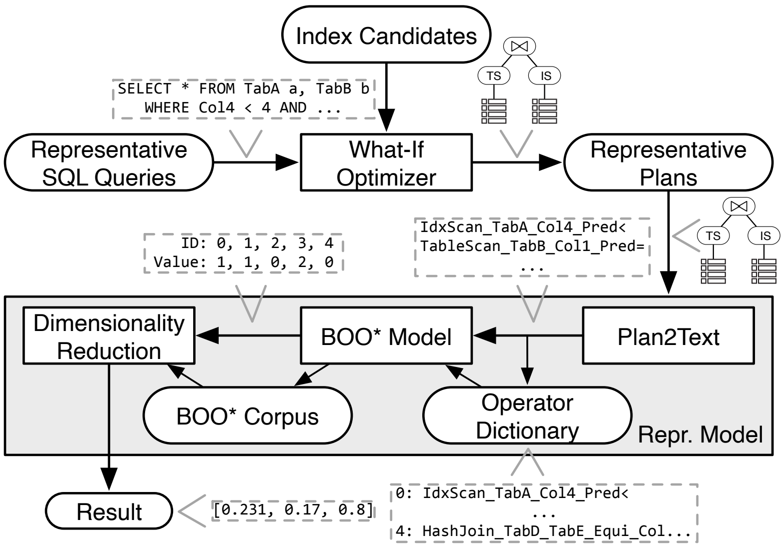


Figure 5.3.: Workload representation example. *BOO refers to Bag Of Operators.

The operators of every plan that are relevant[20] for index selection are transformed into a text representation. For example, under the presence of an index on `TabA.Col4` a text representation `IdxScan_TabA_Col4_Pred<` might be generated. The text representations for all representative plans are stored in the operator dictionary, which assigns an ID to every distinct operator's text representation. These IDs are used in the next step to construct a *Bag Of Operators (BOO)* (cf. bag-of-words model [CP18; Har54]), i.e., a numerical representation of the operators of a query.

Dimensionality Reduction. The BOO could be made part of the state representation and passed to the neural network. Using the BOO without further processing would result in many additional, very sparsely populated features per query. For the TPC-DS benchmark's query templates, we count 839 distinct relevant operators. We would need to incorporate these 839 features for every query of the workload, i.e., N times. Consequently, we apply a dimensionality reduction step. Based on the BOO representations of all representative plans, we build a latent semantic indexing (LSI) [Dee+90] model to reduce the feature count.

Choosing the number of features for representing a query (the representation width R) is a tradeoff decision. Larger values increase the model's size and training times.

[20]We focus on operators that are most affected by indexes and particular attributes of these operators. Source code: `https://github.com/hyrise/rl_index_selection/blob/main/swirl/boo.py`

Smaller values can lead to insufficient workload representations. The *Gensim* library used for the LSI model indicates the amount of information loss for a particular R. We experimented with varying values for R and found that for the examined workloads and $R = 50$, approximately 10 % of information is discarded, see Figure A.1 in Appendix A.1.1. Additionally, since the agent's performance does not improve much when choosing higher R values, $R = 50$ seems reasonable.

We refrained from using other Bag-independent approaches for text vectorization, like Text2Vec [Mik+13], because they require more training data and caused elevated runtimes in our experiments. These higher runtimes might result from Text2Vec relying on more complex neural networks.

Alternative Workload Modeling Approaches. In the literature, alternative workload modeling approaches [Din+19; SL19] exist. The design of such approaches was guided by different use cases than RL-based index selection. For example, the workload modeling of Ding et al. [Din+19] builds the input for a classifier that determines which of two query plan alternatives is cheaper. The modeling approach featurizes query plans based on physical operators, too. However, the featurization is schema-agnostic without explicitly referring to accessed tables or attributes. Instead, it encodes and aggregates different information, e.g., the amount of work done per operator type. Ding et al.'s workload modeling is reasonable for their use case. In contrast, our approach is specifically designed for an RL-based index selection agent. Our agent's actions are directly related to particular operators and attributes. Therefore, the explicit knowledge of how attributes and operators occur in the workload is necessary for good index selections.

Simplifications. If unknown queries are not required to be handled, the workload modeling can be simplified. Some approaches model the workload by only encoding the frequency share of known queries in the current workload, e.g., Hilprecht et al.'s RL-based partitioning approach [HBR20]. We argue that even in cloud scenarios, with a predefined set of standard queries, the assumption that no unknown queries will occur is invalid because customers usually formulate additional queries. The resulting index configuration is likely suboptimal without considering such queries for the agent's decisions.

Actions

The action space determines how the agent can act. In terms of the index selection problem, this means which indexes the agent can create. Our model employs a discrete action space A, where every action is a unique (multi-attribute) index candidate: we set $A := I$. The index candidates are determined during preprocessing, cf. *Overview* above. The existence of thousands of multi-attribute index candidates is not rare for realistic workloads and datasets [SKB19].

Carefully designing and handling the action space is crucial for two reasons. (i) As stated in Section 5.2.1, the training efficiency depends on the number of available actions. More available and dependent actions (index interaction) further complicate the problem for complex combinatorial problems. Simply limiting the index candidates a priori might reduce the size of the action space but can also negatively impact the quality of the determined index configurations, as shown by Schlosser et al. [SKB19]. In addition, (ii) particular actions might be invalid at particular states. Comparable to chess, where the rules restrict the allowed movements for every piece, the index selection process also follows specific rules: we can consider the repeated creation of an existing index or exceeding the storage budget as breaking the rules. In RL, rules are usually enforced in an indirect fashion. Large negative rewards teach the agent the invalidity of certain actions. However, everything that must be learned can potentially increase the training duration and harm performance.

We utilize *invalid action masking* [HO20] to (temporarily) disable actions based on the current state. Thereby, the agent is guided to consider only a subset of all available actions. Valid actions must be updated before every step, and there are four reasons why actions could be marked invalid. These reasons are also demonstrated in Figure 5.4.

(1) **Index candidate irrelevant for workload**. Before the agent's first step, we check whether every index candidate is syntactically relevant for the workload at hand, i.e., whether all of the index's attributes occur in the workload.

(2) **Index would exceed the budget**. Before every step, we calculate for every index candidate whether its creation would exceed the storage budget, given the current storage consumption.

(3) **Index already existing.** After choosing an action a, it is marked invalid such that it cannot be chosen again. Later, action a can be marked valid again, e.g., due to choosing actions associated with multi-attribute indexes.

(4) **Invalid precondition.** Before the first step, all multi-attribute indexes are masked invalid. Only after the agent chose an index `(A)`, all multi-attribute indexes with A as the first attribute, e.g., indexes `(A, B)` and `(A, C)`, are made valid. We follow the intuition of Chaudhuri et al. "that for a two-column index to be desirable, a single-column index on its *leading* column must also be desirable" [CN97, p. 151]. Furthermore, we utilize the *extension* procedure of the *Extend* index selection algorithm presented above. Three- and n-attribute indexes are masked accordingly.

In addition, DBAs might favor preventing the model from handling manually created indexes based on domain knowledge or indexes that guarantee service-level agreements (SLAs). Such indexes can be made entirely inaccessible for the model by invalidating

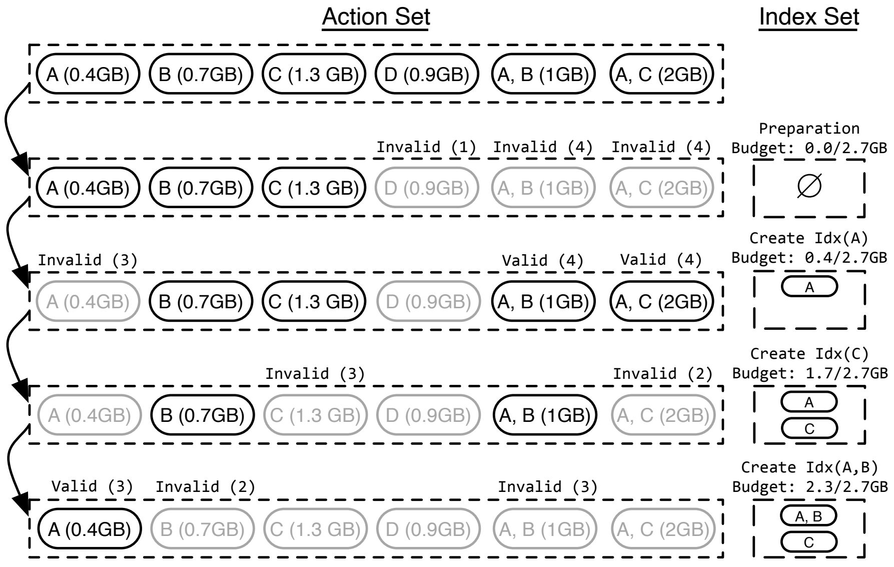


Figure 5.4.: Example for invalid action masking. Numbers in braces on (in)validation actions indicate the reasons for the status change. Creating an index `(A,B)` drops the index `(A)`.

actions affecting them. The effect of action masking on the number of available actions is demonstrated later in Section 5.3.3.

Reward

The agent receives a reward, r_t, for each action. The reward incentivizes beneficial actions and guides the learning process. There are multiple options for building reward functions for index selection. Reward functions could consider relative or absolute cost impacts of indexes, their storage consumption, and their validity. Absolute cost impacts have the disadvantage that these might largely differ for similar actions for different workloads and do not account for the required storage.

We aim to consistently optimize the storage usage for each of the agent's steps. For the above reasons and in line with *Extend*, we chose the additional relative benefit (reduction of workload costs) of an index selection I_t^* per additional utilized storage as the reward:

$$r_t(I_t^*) = \frac{(C(I_{t-1}^*) - C(I_t^*))/C(\emptyset)}{M(I_t^*) - M(I_{t-1}^*)}.$$

In contrast to other approaches, punishing invalid actions with negative rewards is unnecessary due to invalid action masking. The `RewardCalculator` encapsulates the reward handling to facilitate defining and evaluating alternative reward functions.

Implementation

Our open-source implementation [KKS22b] to train, evaluate, and adapt the presented approach is written in Python 3. We rely on the index selection evaluation platform presented in Chapter 4. The platform encapsulates, e.g., the retrieval of query plans and the handling of hypothetical indexes. Cost estimation requests are responsible for the majority of the runtime of index selection algorithms [PDA07]. Therefore, the platform's cost request caching is indispensable for efficient training procedures. Cache rates are investigated later in Section 5.3.3.

Model. For implementing the RL algorithm (PPO), we use Stable Baselines [Hil+18] versions 2 and 3 that rely on Tensorflow, respectively PyTorch. The agent interacts with a database environment that is implemented according to OpenAi's gym [Bro+16] interface. The latent semantic indexing (LSI) model used for workload representation is built with Gensim [ŘS10].

The model's hyperparameters that are displayed in Table 5.2 were experimentally determined. The gamma value appears low compared to other problems that are traditionally solved via RL. A value in the lower range increases the agent's greediness. Simultaneously, it is still chosen high enough to allow long-term considerations in the relatively short episodes of the index selection problem. We have demonstrated in *State Representation* above that the number of features depends mainly on the workload size and the used representation width. Therefore, it might be necessary to adapt the network architecture for larger problem instances before training; the displayed size was sufficient for the scenarios evaluated in Section 5.3.

Flexibility for Adaptations. Our implementation should facilitate further experiments with RL-based approaches for index selection or physical database design in general. For this reason, we strictly modularized *SWIRL*'s implementation. For instance, there are modules for the reward determination, state representation, or the maintenance of the action space. Hence, alternative implementations for these modules can be added easily to evaluate different RL modeling strategies and design decisions experimentally. As a result, the well-defined interfaces of our modular implementation enabled us to reimplement *DRLinda* (cf. Section 5.2.2) in a few hours.

Table 5.2.: Hyperparameters for our PPO reinforcement learning model.

Hyperparam	Value	Hyperparam	Value
Learning rate η	$2.5 \cdot 10^{-4}$	Discount γ	0.5
Batch size	2 048	Clip Range	0.2
ANN Layer Structure for Q and π	256-256	Policy	MLP

Most parameters, e.g., the workload size, maximum index width, or reward function, can be configured via JSON configuration files. Furthermore, the concept of our approach is not tailored to index selection. Instead, it could be extended to other physical database design problems, e.g., automated compression selection [Boi22] or partitioning, as long as the impact of varying configurations can be determined and transformed to a reward.

Miscellaneous

We monitor the model's performance with workloads not part of the training and testing sets at every few thousand steps to prevent overfitting. If the moving average of the performance stops improving, we record the model's current state. Also, we employ extensive caching of cost requests, which largely impacts the training duration, see Section 5.3.3.

5.2.4. Classification

Based on Section 3.1.1, *SWIRL* can be classified as a *machine learning-based* index selection algorithm that *adds* indexes to an empty index configuration. While *SWIRL*'s reward function can be adapted easily, the function used throughout this thesis aims to reduce relative costs. Thus, the optimization target is $\frac{Costs}{Storage}$. Currently, *SWIRL* is not related to a commercial system. *SWIRL* supports multi-attribute indexes without significantly limiting the candidate set a priori. In contrast to *Extend*, the index candidate set must be known in advance to determine the agent's action space. During the extensive preparatory training, a multitude of different index configurations is evaluated. Hence, index interaction is considered to a high degree (+++). The sophisticated state representation and workload modeling and the necessary combination of several frameworks (StableBaselines and OpenAi's gym) result in our open-source implementation's high complexity (+++).

5.3. Evaluation

This section evaluates our two new index selection approaches, *Extend* and *SWIRL*, by comparing their performance to state-of-the-art algorithms. The evaluation is performed with the index selection evaluation platform presented in Section 4.1.4.

We first describe the experimental setup and discuss the choice of the competing algorithms in Section 5.3.1. Thereafter, in Section 5.3.2, we evaluate the solution quality of the identified index configurations and the index selection runtime. Subsequently, we focus on more specific evaluations, e.g., *SWIRL*'s training durations and the effectiveness of the *invalid action masking* technique in Section 5.3.3.

5.3.1. Experimental Setup

The following experiments are executed on the same machine, with identical Python and PostgreSQL versions as mentioned in Section 4.1.5. Again, the evaluation covers the TPC-H, TPC-DS, and JOB. The utilized benchmarks contain complex queries that challenge index selection approaches and differ in dataset size as well as workload complexity (number and intricacy of queries), cf. Section 4.1.1. The upcoming experiments differ from the previous ones in terms of, e.g., the provided budgets, selected query templates, and assigned frequencies. Since *SWIRL* is a learned approach, each experiment is repeated multiple times with different random seeds to ensure stable results.

The experimental evaluation of Section 4.2 aimed at comparing many existing index selection algorithms. This evaluation, in contrast, assesses whether the new approaches meet the target formulated in Chapter 5, closing the gap between index selection approaches that produce either fast or close-to-optimal solutions. Furthermore, we investigate whether *SWIRL* fulfills the requirements for RL-based index selection approaches stated in Section 5.2.2. The different aims are reflected in the design of the following experiments and the choice of competitors.

Experiment Design. For machine learning-based approaches in particular, the experiments need to demonstrate the generalization capabilities of the learned model. For this reason, we randomize workloads for the following experiments: we create workloads of size N by randomly choosing query templates from all of the available templates of a selected benchmark and assigning random frequencies according to a uniform distribution. In addition, for learned approaches, we define a certain number of query templates that are withheld during training and the proportion of these withheld queries in the test workloads used for evaluation. Independent of the unknown templates, we always ensure that test workloads are not used for training. Thus, all evaluated workloads differ from training workloads in three dimensions: (i) the evaluated workloads contain the unknown templates withheld during training. (ii) The exact combination of query templates and (iii) the exact frequency-template-combination have not been seen during training.

Choice of Competing State-of-the-Art Algorithms. Based on the findings of Section 4.2, we compare our *Extend* and *SWIRL* algorithms with the state-of-the-art approaches that were performing best in the two critical dimensions: solution quality and index selection runtime. *DB2Advis* achieved the lowest index selection runtimes and *DTA* is a well-tried [CN20] algorithm with the overall best solution quality.

Choice of Competing RL-Based Algorithms. Additionally, we choose *DRLinda* for RL comparisons as it is the only competitive algorithm that also seeks to generalize to handle unseen workloads [SGL20a; SGL20b]. Even without publicly available source code[21], the descriptions in the publications allow for reimplementation. There are counter-arguments for the comparison with the other RL-based index selection approaches discussed in Section 5.2.2. *NoDBA* [SSD18] has been evaluated for a recent evaluation study [Kos+20d]. In this study, *NoDBA* was not competitive with state-of-the-art approaches. *DRLISA* [Yan+21] focuses on NoSQL and lacks detail on its workload modeling approach, preventing reimplementation. SMARTIX [Lic+20] was not considered because it neither includes workload representation nor supports multi-attribute indexes. Moreover, its long runtimes of multiple days make an extensive evaluation infeasible. Lan et al.'s [LBP20] approach is only trained for a single fixed workload and cannot generalize. Thus, it is not reasonable to evaluate its generalization capabilities.

Note that originally, *DRLinda* does not support storage budgets but a maximum number of indexes to create selections of different sizes. To evaluate selections for a given budget, we subsequently select indexes according to the order associated with *DRLinda*'s solutions for increasing numbers of indexes as long as permitted by the experiment's budget. To achieve better and more fine-grained solutions, we also check whether subsequent (potentially smaller) indexes can be added.

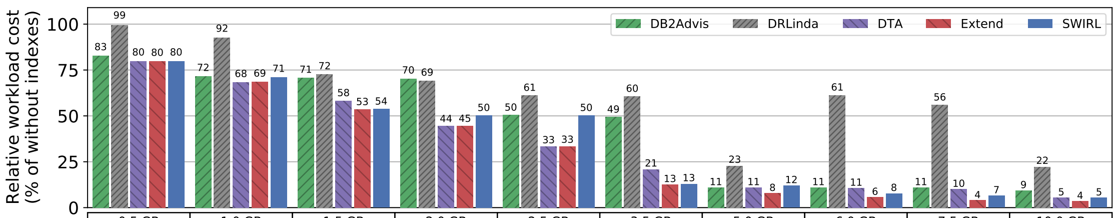
5.3.2. Algorithm Performance

We evaluate the performance in terms of solution quality achieved and observed index selection runtime. For a first, detailed, budget-dependent performance overview, *Extend* and *SWIRL* are compared to the aforementioned competitors, *DB2Advis*, *DTA*, and *DRLinda*, for a single JOB-based workload for budgets from 0.5 to 10GB. We chose the JOB because it operates on real-world data, is the largest of the evaluation platform's benchmarks, and contains the most complex queries. For the following experiment, we use a workload size of N=50. Of the JOB's 113 query templates, 10 are withheld during training, all of these are included in the workload evaluated in Figure 5.5. Hence, 20 % of the workload's query templates are unknown to the learned approaches. The bar chart depicts the estimated workload processing costs for different budgets and the table displays the selection runtime.

We obtain three main insights from Figure 5.5. First, the figure demonstrates that a set of adequately chosen indexes decreases the workload costs significantly. For large budgets, indexes can reduce the processing cost by more than 20 times. Second, *Extend* can determine high-quality solutions: it delivers the best solution in 8 out of 10 cases. It is always faster than its strongest competitor *DTA* by multiple factors.

Third, it becomes apparent that *SWIRL*'s solutions are competitive despite its short

[21]We contacted the authors to obtain the original source code. The authors were not able to provide the code but supported our reimplementation by providing answers to our questions.



	0.5 GB	1.0 GB	1.5 GB	2.0 GB	2.5 GB	3.5 GB	5.0 GB	6.0 GB	7.5 GB	10.0 GB
DB2Advis	6.00s	5.98s	6.26s	6.08s	6.12s	6.03s	6.30s	6.28s	6.28s	5.96s
DRLinda	2.30s	4.45s	3.10s	5.13s	8.29s	8.38s	8.87s	7.95s	11.28s	11.16s
DTA	305.32s	388.74s	392.46s	369.74s	564.69s	762.97s	1159.51s	1258.30s	1748.24s	2434.70s
Extend	91.49s	60.18s	132.74s	70.86s	143.31s	189.94s	203.51s	256.76s	283.12s	310.09s
SWIRL	**0.60s**	**4.33s**	**0.39s**	**2.49s**	**4.10s**	**1.00s**	**3.95s**	**2.33s**	**3.89s**	**3.81s**

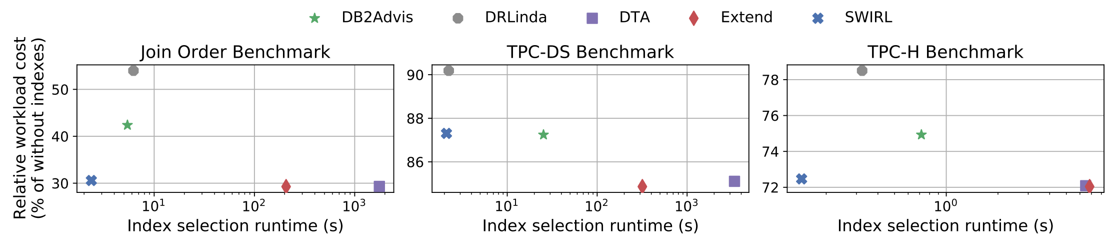
Figure 5.5.: Performance comparison of state-of-the-art approaches vs *SWIRL* and *Extend* for a Join Order Benchmark workload ($N = 50$; 20 % of templates are unknown to *SWIRL*). Chart: workload cost relative to processing without indexes; table: index selection runtime.

index selection runtimes. In terms of cost, its performance is always on par with or more advantageous than its two fast competitors, *DRLinda* and *DB2Advis*: it outperforms *DB2Advis* in 9 out of 10 cases and *DRLinda* in every case. Except for a budget of 2.5 GB, its performance is close or equal to *DTA*'s and *Extend*'s; between 1.5 GB and 3.5 GB *SWIRL*'s solutions appear to lack granularity. For large budgets, *SWIRL* can even outperform *DTA*. In terms of selection runtime, *SWIRL* also excels over all competitors in all 10 cases. *DTA* and *Extend* are often orders of magnitude slower.

Additionally, we can observe that the selection runtimes are not monotonically increasing for most approaches. The reason for this is that, often for increasing budgets, one large index instead of multiple small indexes might be selected and consume the majority of the budget, thereby decreasing the number of index selection decisions and terminating the index selection process earlier. This effect is particularly pronounced for stepwise approaches, such as *Extend* and *SWIRL*. As a side note, the performance of *DRLinda* is not consistent. It is worse for 6 GB and 7.5 GB than for 5 GB.

Average Performance for Numerous Workloads. After comprehensively evaluating one workload, we now evaluate whether our two algorithms show acceptable performance across many different cases. For each of the three benchmarks, we train RL models for *SWIRL* and *DRLinda* and generate 100 random evaluation workloads with different frequencies and 20 % of query templates that are withheld during training. In other words, the upcoming experiments also indicate whether the learned approaches are capable of generalizing to handle unseen workloads. For each of the 100 evaluation workloads, we determine index configurations with all competitors for random budgets between 0.25GB and 12.5GB. Then, we calculate average performances ($\varnothing\, RC$), runtimes ($\varnothing\, t$), as well as the number of evaluation workloads for which an algorithm finds index configurations that achieve the lowest relative workload cost (# Wins) across all competitors[22]. The results are presented in Figure 5.6 and show the relationship between solution quality and index selection runtime.

Figure 5.6 demonstrates that two different kinds of approaches exist. *DB2Advis*, *DRLinda*, and *SWIRL* quickly determine solutions of reasonable quality, while *DTA* and *Extend* require more time but achieve better solutions. Our approaches, *Extend* and *SWIRL*, dominate their direct competitors in both dimensions: selection runtime and solution quality.



	ØRC	Ø t	#Wins	ØRC	Ø t	#Wins	ØRC	Ø t	#Wins
DB2Advis	42.38%	5.4s	1.00	87.24%	25.3s	3.00	74.93%	0.7s	14.17
DRLinda	53.98%	6.2s	0.00	90.19%	2.2s	0.00	78.50%	0.3s	0.00
DTA	29.31%	1761.8s	40.00	85.11%	3365.5s	38.50	72.09%	6.5s	38.67
Extend	**29.27%**	207.0s	**52.00**	**84.86%**	318.1s	**55.50**	**72.04%**	6.8s	**45.17**
SWIRL	30.57%	**2.4s**	7.00	87.31%	**2.1s**	3.00	72.47%	**0.1s**	2.00

Figure 5.6.: Performance comparison across 100 random workloads of each of the TPC-H (SF10), TPC-DS (SF10), and Join Order Benchmark. $RC := C(I^*)/C(\emptyset)$ denotes the relative workload costs, $\varnothing$ the arithmetic mean, t the index selection runtime, *# Wins* the number of winning index configurations, see Footnote 22 for details. The X-axis uses a logarithmic scale.

Extend determines the best solutions across all benchmarks (regarding the # Wins and the relative total workload costs compared to using no indexes: $RC := C(I^*)/C(\emptyset)$. In terms of solution quality, the differences to *DTA* are only marginal. However, with regards to selection runtime, *DTA* is approximately ten times slower for the more complex workloads (JOB and TPC-DS). *SWIRL*'s solutions are, on average, close to the best competitors (*Extend* and *DTA*) and superior over *DRLinda* and *DB2Advis*. For instance, for the JOB, *SWIRL*'s performance is, on average, only 1.3 pp worse than *Extend*'s. The overall large advantages over *DRLinda* can be explained by its relatively simple approach to representing workloads and the fact that only single-attribute indexes are supported.

Regarding the mean selection runtime ($\varnothing\, t$), *SWIRL* outperforms all four competitors after training and undercuts *DTA* and *Extend* by orders of magnitude. For example, for the TPC-DS, *SWIRL* determines index configurations in 2.1 s on average, while

[22]Note, in draw situations, when multiple approaches achieve the same solution quality (RC), the *point* for winning this scenario is split evenly.

DB2Advis takes 12 ×, *Extend* 151 ×, and *DTA* 1 603 × as long. *DRLinda* is only marginally slower. The difference might be due to *SWIRL*'s efficient handling of index candidates via action masking and the native support for budget constraints, see Section 5.3.1.

5.3.3. Specific Evaluations

In contrast to the above experiments, the following evaluations are of less general nature and aim to answer specific questions regarding particular index selection algorithms.

Anytime Capabilities

DTA is conceptually built around the promise of being interruptible, cf. Section 3.1.4. Therefore, *DTA* should return reasonable index configurations at *any time*. However, the results of Figure 5.6 demonstrated a significant runtime difference between *DTA* and its competitors. The following experiment investigates *DTA*'s *anytime* capability to determine whether constrained, faster solutions of *DTA* are comparably good.

Furthermore, we examine how *DTA*'s solutions compare against *Extend*'s under time constraints. For this purpose, we modified *Extend*[23] to offer *anytime* capabilities, too. For reference, we also include *SWIRL* in these experiments.

Figure 5.7 shows the solution quality of identified index configurations for six different JOB workloads with budgets ranging from 0.5 GB to 10 GB. According to the figure, both algorithms can keep the *anytime* promise and identify reasonable solutions under time constraints. *DTA* is able to identify solutions much faster than shown in Figure 5.6. However, the price for faster solutions is, in some cases, a large performance penalty. These performance penalties are most pronounced for budgets starting from 3.25 GB.

Furthermore, while *DTA* determines comparable solutions to *Extend*, the latter identifies equivalent index configurations significantly faster in the vast majority of cases. After all, the observations obtained from additionally considering time constraints do not fundamentally differ from the previous ones discussed above. Therefore, *Extend* dominates *DTA* in terms of selection runtime and solution quality.

Figure 5.7 also emphasizes the differences between learned and imperative approaches: learned approaches can almost instantly determine solutions for any decision problem, i.e., workload-budget-combination. However, learned approaches require extensive preliminary training to achieve the presented runtimes.

[23] Anytime version of *Extend* on GitHub: `https://github.com/hyrise/index_selection_evaluation/blob/master/selection/algorithms/extend_algorithm_anytime.py`

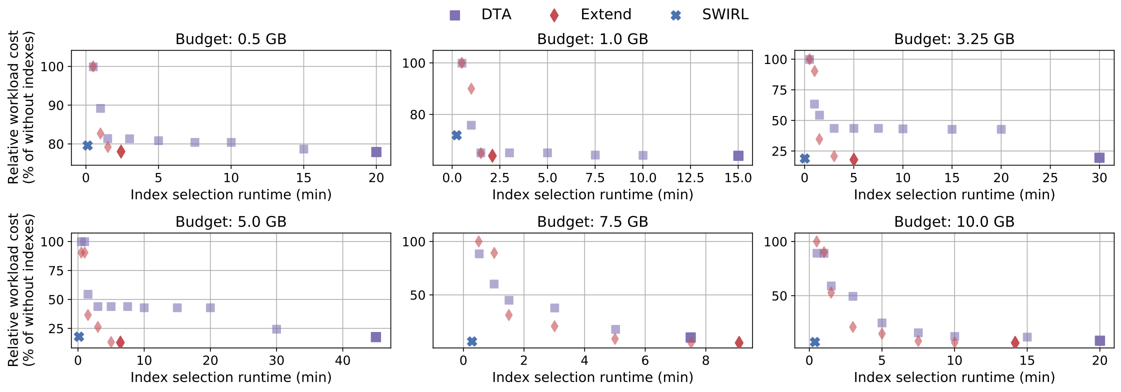


Figure 5.7.: Evaluation of *anytime* capabilities. Comparison of time-constrained solutions for different JOB workloads, $N = 100$, with varying budgets for DTA and $Extend$; $SWIRL$ depicted for reference. Workload cost relative to executing the workload without indexes, $C(I^*)/C(\emptyset)$. Evaluated time constraints: $T \in \{0.5, 1, 1.5, 3, 5, 10, 15, 20, 30, 45, 60\}$ minutes. Solutions with equivalent performance but higher runtime excluded. Best identified solutions depicted with non-transparent markers, others with transparent markers.

SWIRL: Training Duration and Effort

RL-based index selection approaches pay the price for determining efficient configurations upfront: short computation times at application time are exchanged for long, a priori training durations. Table 5.3 shows the training duration and the number of cost requests that occurred during training, along with the number of features and actions for different scenarios. All scenarios were trained with 16 parallel index selection environments.

The training duration refers to the time needed for convergence, i.e., until no further improvements are realized. Obtaining the cost for a query given a particular index configuration is denoted as a cost request. The number of cost requests is crucial for assessing index selection approaches because, even though a single request takes only milli- or microseconds, it is not uncommon that millions of such requests are issued during selection processes, as discussed in Section 4.2.4. In addition, the training duration also contains the time required for computing the state representation, applying action masking, creating and dropping hypothetical indexes, updating the weights of the neural network, and using it for predictions during training.

Several dimensions influence the problem complexity and, thereby, the training durations: (i) the workload size N influences the number of features. Large values increase the time necessary for estimating the workload's execution costs as more queries have to be converted to query plans by the optimizer. (ii) The complexity of the queries: more complex queries cause longer optimization times. (iii) The number of index candidates:

Table 5.3.: Training duration and problem complexity metrics for different benchmark scenarios. W_{max}: admissible index width. t: time. Note, the actions directly correspond to the index candidates, $A = I$.

Benchmark	N	#Features	W_{max}	$\|A\|$	#Episodes	Training duration breakdown			
						Total	Costing	#Cost requests (cached)	∅ Episode t
TPC-H	19	468	1	46	2 272	0.07h	20.2%	1 829 088 (95.9%)	0.1s
TPC-H	19	468	3	3 532	768	0.19h	32.6%	1 802 016 (71.2%)	0.9s
TPC-DS	30	1 750	1	186	751	0.42h	22.0%	3 002 850 (92.5%)	2.0s
TPC-DS	30	1 750	2	3 174	512	0.77h	22.4%	2 995 680 (84.6%)	5.4s
TPC-DS	60	3 310	2	3 174	512	1.31h	23.4%	5 991 360 (86.9%)	9.2s
JOB	100	5 265	1	61	1 616	2.58h	37.4%	10 097 600 (83.3%)	5.7s
JOB	100	5 265	3	819	560	5.52h	47.5%	9 990 400 (63.4%)	35.5s

many candidates lead to large action spaces. The agent requires more time to determine efficient actions, and the existence of more indexes can increase the query optimization time. For these reasons, the evaluated scenarios cover workloads of different sizes from the three benchmarks supported by our platform and different admissible index widths.

Table 5.3 shows that training durations increase with the workload and index candidate complexity and range from multiple minutes to a couple of hours. The training duration is reasonable even for large workload sizes with 100 query templates and 3-attribute indexes. According to Figure 5.7, *Extend* needs ≈ 10 minutes to determine a solution for a scenario for which *SWIRL* trains 331 minutes, or $33\times$ as long. As shown in Figure 5.6 and Figure 5.5, *SWIRL*'s runtimes only amount to a few seconds. Hence, if dozens or hundreds of systems must be tuned (repeatedly), short runtimes compensate for long training durations. The training time of our reimplementation of *DRLinda* is in the same range. While *DRLinda*'s model is simpler, the utilized DQN [Mni+15] is, in Stable Baselines, not as efficient as PPO.

Table 5.3 also demonstrates that a large fraction of the time required for training is caused by cost requests, even though most of these requests can be cached. This effect can be observed [PDA07] for most state-of-the-art index selection approaches, too, cf. Section 4.2.4. Interestingly, training durations can vary significantly for similar numbers of cost requests, which can be observed for the two JOB experiments: the first experiment creates slightly more cost requests but requires only 47 % of the training duration. This effect is probably due to the much higher cost request cache rate, which, in turn, is caused by fewer index candidates and possible actions (due to a smaller W_{max}).

Effectiveness of Action Masking. As explained above, we rely on *invalid action masking* to provide state-dependent action sets, thereby assisting the agent during training by reducing the action space. In the following, we investigate the effectiveness of applying this technique to index selection. Figure 5.8 indicates the effectiveness by depicting the share of valid actions at any given point of a single training episode for a JOB scenario. The figure also shows how many valid actions refer to indexes of widths 1, 2, or 3 and the fraction of these actions that are invalidated because they do not fit the remaining budget.

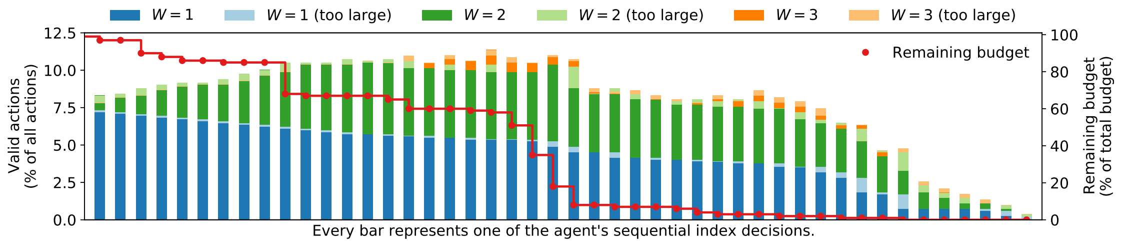


Figure 5.8.: Impact of invalid action masking on the number of available actions for a JOB scenario ($N = 100$, storage budget $B = 10\,\text{GB}$, maximum index width $W_{max} = 3$, and $|A| = 819$ candidates). Lighter colors indicate the fraction of actions that are invalid due to the corresponding index' size given the remaining budget.

Figure 5.8 demonstrates that in the beginning, only $\approx 7.5\,\%$, and at no point more than $12\,\%$ of all actions are valid; with a decreasing remaining budget, more indexes are invalidated because they would exceed the budget. Apparently, many actions are temporarily excluded because their precondition is not met, cf. Figure 5.4 on page 73. In addition, the majority of valid actions refer to indexes of widths 1 and 2. For these reasons, invalid action masking appears to be an effective technique to restrict action spaces and accelerate training procedures.

Consequently, this technique also decreases the required training duration for convergence. According to our experiments, the duration for a non-masking variant increases by eight times for a TPC-H scenario with a maximum index width, $W_{max} = 1$, to achieve comparable performance. This effect is even more pronounced for more realistic problems with larger action spaces: for TPC-H with $W_{max} = 3$, which comes with significantly more index candidates ($|I| = 3\,306$ vs $|I| = 53$), the solution quality of the non-masking version was not close to *SWIRL*'s, even after training more than ten times as long. We observe that the overall performance might be worse even with extended training durations without action masking, which is in line with other work focusing on invalid action masking for policy gradient algorithms [HO20].

SWIRL: Training Data Influence

The following experiments conduct training data influence studies to answer how *SWIRL*'s generalization capabilities depend on the training data. We investigate (i) how the number of unknown query templates and (ii) if the number of unknown templates is fixed, how the selection of unknown templates impacts the agent's performance.

(i) Impact of the Number of Unknown Query Templates. The following experiment is conducted with TPC-H workloads. The TPC-H's small number of query templates (19 in accordance with the 3 templates excluded as described in Section 4.2.1) enables us to easily remove query templates that are essential for generalization. We trained five models for this experiment. For these five models, the number of randomly chosen query templates that are unknown during training is successively increased from 0 % to 63 %. All query templates that are unknown to a model are also unknown to the subsequent models. The first model is trained on all query templates.

After training, the performance of all trained models is evaluated with 50 random workloads and compared with the strongest (*Extend*) and fastest (*DB2Advis*) competitors from the above experiments. Each workload consists of queries generated from 5 different query templates, drawn at random from the set of all of the 19 considered TPC-H query templates. We chose a small workload size so that unknown templates make up a large proportion of the workload. The frequencies are also assigned randomly per template and workload. Budgets are chosen at random between 0.5 GB and 7.5 GB.

Table 5.4 depicts the number of unknown query templates and the resulting performance in relative workload costs compared to using no indexes: $RC := C(I^*)/C(\emptyset)$. Additionally, it sets the results into context (Δ *Extend*, Δ *DB2Advis*), e.g., +1.2 pp means that the competitor could reduce the relative workload costs by 1.2 percentage points more. These results demonstrate that, with an increasing number of unknown queries, the agent's performance continuously degrades because it is less capable of generalizing. These observations are not unexpected and confirm the necessity for detailed workload representations and representative training data. On the other hand, the experiment demonstrates that even with a large number of excluded queries, *SWIRL* is capable of determining reasonable solutions to some extent.

Table 5.4.: Evaluating the impact of the number of unknown query templates during training. Performance in terms of relative workload processing costs, $RC := C(I^*)/C(\emptyset)$, for models with an increasing number of unknown templates. Differences to imperative index selection approaches in percentage points (pp) where positive numbers indicate a disadvantage of the learned models.

Model no.	Unknown templates	Δ *Extend*	Δ *DB2Advis*	⌀ *RC*	*RC* relative to model no. 1
1	0 / 19 (0 %)	+1.2 pp	+0.2 pp	77.4 %	100 %
2	3 / 19 (16 %)	+3.1 pp	+2.1 pp	79.3 %	91.6 %
3	6 / 19 (32 %)	+4.2 pp	+3.2 pp	80.4 %	86.7 %
4	9 / 19 (47 %)	+4.8 pp	+3.8 pp	81.0 %	84.1 %
5	12 / 19 (63 %)	+9.3 pp	+8.3 pp	85.5 %	64.2 %

(ii) Impact of the Selection of Unknown Query Templates. Next, we examine whether *SWIRL*'s performance depends on the particular set of query templates that are unknown during training. The experiment is conducted with workloads based on the TPC-DS benchmark. We chose this benchmark because its size (99 query templates) allows for many different exclusions sets and large workloads simultaneously. In contrast to the JOB, the training and evaluation times are significantly shorter.

For this experiment, we trained ten agents for a workload size of $N = 50$, while 15 % of all of the TPC-DS' query templates were unknown during training. The random seeds for both the selection of the query templates that are unknown during training and the initialization of the weights of the ANN are different for each of the ten agents. Thereby, the set of templates excluded from training is different for each agent, allowing for observing the impact of different training sets in Table 5.5.

Table 5.5.: Evaluating the impact of the exact selection of unknown query templates during training. Performance in terms of relative workload processing costs, $RC := C(I^*)/C(\emptyset)$, for ten models trained with different sets of unknown (during training) templates based on different random seeds. Differences to imperative index selection approaches in percentage points (pp) where positive numbers indicate a disadvantage of the learned models.

Model no.	Δ *Extend*	Δ *DB2Advis*	∅ RC	IDs of unknown templates
1	−0.9 pp	−0.9 pp	84.3 %	26, 28, 36, 37, 39, 40, 42, 48, 52, 53, 57, 60, 82, 87
2	+1.8 pp	−0.7 pp	85.7 %	12, 20, 21, 29, 52, 56, 58, 62, 63, 67, 68, 82, 87, 92
3	−1.2 pp	−1.1 pp	84.7 %	05, 07, 25, 28, 36, 37, 48, 57, 63, 64, 68, 70, 81, 92
4	+1.0 pp	−1.5 pp	86.0 %	05, 22, 29, 34, 36, 43, 52, 60, 65, 68, 82, 86, 87, 97
5	−1.0 pp	−2.0 pp	82.0 %	07, 12, 21, 25, 36, 37, 38, 43, 48, 50, 53, 55, 57, 69
6	+0.2 pp	−0.1 pp	83.5 %	12, 20, 21, 26, 27, 28, 37, 42, 52, 60, 62, 63, 84, 86
7	+2.0 pp	−0.4 pp	85.5 %	07, 21, 36, 48, 51, 62, 63, 70, 73, 82, 84, 87, 96, 97
8	+1.3 pp	−0.8 pp	84.6 %	03, 12, 20, 25, 29, 33, 36, 40, 43, 50, 69, 70, 82, 87
9	+1.9 pp	−0.3 pp	82.5 %	12, 22, 26, 28, 29, 34, 39, 40, 48, 53, 55, 73, 96, 97
10	+2.1 pp	−0.3 pp	84.0 %	01, 05, 12, 21, 25, 28, 30, 37, 52, 53, 57, 67, 96, 99

After training, we evaluate the performance of every differently trained agent with 20 different workloads of size $N = 50$. The queries are drawn at random from the set of all of TPC-DS query templates. Also, it is ensured that 20 % of these query templates are drawn from the set of templates that were unknown during the training of the particular agent. Query frequencies and budgets are also assigned at random.

Table 5.5 depicts the set of query template IDs that are unknown during training and the resulting performance in terms of the relative workload costs compared to using no indexes (RC). The achieved relative costs are compared with *DB2Advis* and *Extend* as above. The results indicate that the performance of the trained agents does not depend on the exact set of unknown query templates to a large degree. Furthermore, the performance is consistent and competitive when compared with *DB2Advis* and *Extend.* The relative costs are in the same range for all ten models. They are always better than *DB2Advis*'s and sometimes better than *Extend*'s. The specific selection of query templates excluded from training appears to be of minor importance if the workload, N, is sufficiently large.

5.4. Discussion and Interpretation

Our two new, efficient index selection algorithms pursue different goals: *Extend* determines close-to-optimal solutions that are at least on par with state-of-the-art approaches. At the same time, *Extend* finds solutions faster than its direct competitors. On the other hand, *SWIRL* fills the gap between the aforementioned algorithms that identify close-to-optimal configurations and those that determine solutions rapidly. We have shown that both approaches achieve their goals by comparing *SWIRL* and *Extend* to three state-of-the-art competitors with complex analytical benchmarks. In the following two subsections, we discuss this result in more detail separately for our two algorithms.

5.4.1. SWIRL

According to the above evaluation, *SWIRL* fulfills the requirements stated in Section 5.2.2. It determines comparably good *(R-I)* (and sometimes better) multi-attribute *(R-IV)* index configurations for different budgets *(R-V)* and for partly unknown workloads *(R-VI)*. *SWIRL*'s index selection runtimes are lower than the runtimes of its competitors *(R-II)* after a preparatory training phase. Due to its stochastic nature, there is no guarantee for close-to-optimal solutions. At times, *SWIRL* performs worse than its competitors. We argue that this drawback is acceptable given the achieved solution time advantage.

SWIRL featurizes the workload's query plans to understand which operations of a query benefit from which index. Naturally, this approach is influenced by the data, i.e., query classes, seen during training. If unknown queries contain too many previously unseen operators, it is more difficult for the agent to associate them with index candidates and optimize decisions. In such cases, the training data should be improved. However, our experiments have shown that the exact selection of training queries is of minor importance if the workload size is sufficiently large.

In addition, *invalid action masking* has proven to be effective in reducing the number of applicable actions, thereby enabling acceptable training times *(R-III)*. Compared to the solution runtimes (for a single problem instance) of state-of-the-art algorithms, the observed training durations are still significant. Consequently, there is a clear tradeoff between long, a priori training durations and low runtimes during application. For this reason, the use of RL-based index selection approaches is not reasonable in all scenarios, but only where the repeated determination of index configurations in similar scenarios is necessary, e.g., in SaaS scenarios where many systems process similar workloads, as motivated in the beginning of this chapter.

5.4.2. Extend

In other scenarios where time-consuming a priori training is not an option or where workloads vary, our evaluations present evidence that *Extend* is a reasonable alternative. *Extend* dominates all competitors in terms of the quality of the identified solutions. Furthermore, *DTA*, which identifies close-to-optimal solutions similar to *Extend*, is clearly outperformed in index selection runtime. Additionally, as discussed in Section 5.1.1, *Extend* allows for simple modifications and extensions due to the simplicity of its core algorithm. For instance, *anytime* capabilities — as proposed by the *DTA* algorithm — can be easily added to *Extend*, as successfully demonstrated above.

In contrast to its competitors, *Extend* does not require the index candidate set to be limited to achieve reasonable runtimes. Its constructive approach, which creates index candidates by *extending* previously selected indexes, is a flexible alternative. Therefore, *Extend* can also determine indexes of arbitrary width without large additional runtime penalties, while the admissible index width, W_{max}, has a significant impact on the index selection runtime for most of its competitors. Increasing W_{max} inflates their candidate sets, which typically increases index selection runtime, as discussed in Section 4.2.5.

5.5. Summary and Future Work

In this chapter, we have presented two novel algorithms for the fifty-year-old index selection problem. Both techniques apply new approaches for handling the two main challenges of index selection: the ample solution space and index interaction, thereby overcoming the shortcomings of their competitors, as schematically depicted in Figure 5.9.

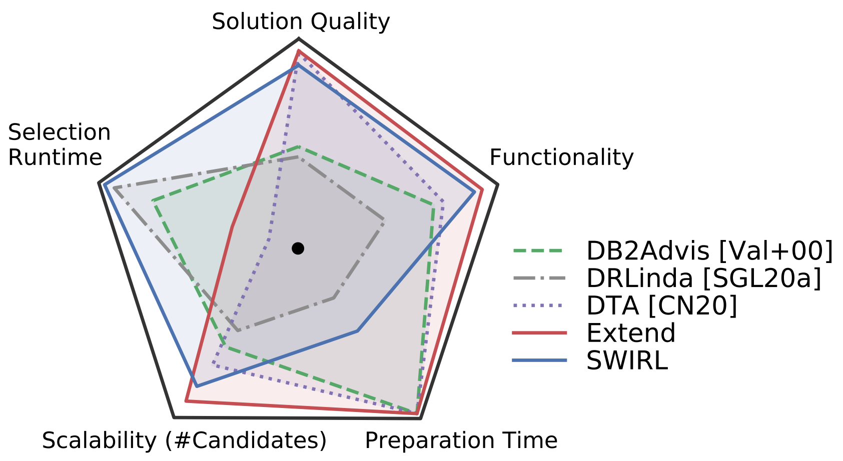


Figure 5.9.: Schematic comparison of index selection approaches. Our approaches complement each other and overcome the weaknesses of state-of-the-art approaches. *Functionality* includes, e.g., support for multi-attribute indexes or budget constraints. Larger distance to the center is better.

Instead of working with a fixed — and potentially large — set of index candidates, *Extend* constructs index candidates on demand during operation. With this approach, *Extend* keeps the solution space small at any given point in time without permanently limiting it. Thereby, the imperative *Extend* algorithm identifies index configurations with better performance than state-of-the-art algorithms.

SWIRL takes an alternative approach to handle the challenge of index interaction. In general, state-of-the-art approaches re-evaluate the benefit of index candidates every time a candidate is chosen to incorporate index interaction effects. In contrast, *SWIRL internalizes* the knowledge about dependencies between index candidates (index interaction) during training. At application time, the underlying ANN enables deriving solutions for arbitrary inputs without costly re-evaluations. Thus, *SWIRL* identifies solutions orders of magnitude faster than state-of-the-art approaches.

In contrast to other RL solutions, *SWIRL* supports multi-attribute indexes and generalizes for handling (partly) unknown workloads. While *invalid action masking* appears to be an effective technique for efficient training, the training procedure can take up to a few hours and be dozens of times higher than solution times of state-of-the-art algorithms. Hence, our two new approaches complement each other: it is most reasonable to apply *SWIRL* in scenarios where many index selection problems must be solved, because it trades off fast solution runtimes against elevated training durations. On the other hand, *Extend* can be applied flexibly without preparatory training procedures to identify close-to-optimal solutions.

Future Work. There are interesting directions for future work for index selection in general and our approaches in particular. Our approaches could be extended to integrate further aspects of physical database design, such as automatic compression [BJ19] or partitioning [HBR20] selection, thereby allowing such decision problems to be considered together, rather than independently [KS20]. This joint optimization [RSB21] could also profit from investigating how the above insights regarding index selection could be transferred to other physical design challenges [ACN00].

We also envision specific ideas for future work for *SWIRL*. Currently, *SWIRL* is trained for a specific schema and the training times are significant. Workloads that are highly different from the training workloads might cause suboptimal results. However, several techniques could overcome these challenges. For example, *transfer learning* [PY10] or its extreme variant *zero-shot learning* [HB22] could speed up the retraining of existing models for different schemas. Another idea is to seed the agent with expert decisions [EKM04] obtained for training workloads to make the training procedure more efficient. Such expert decisions could originate from state-of-the-art algorithms, such as *DTA* or *Extend*. Additionally, different workload representations, e.g., with word2vec [JH18; Mik+13] or other encodings [SL19], could lead to more generalization.

For index selection in general, e.g., robustness and risk aversion [SH20] are interesting aspects. Such aspects are often demanded for the productive use of unsupervised approaches. In addition, even though there are no conceptual limitations, we have not evaluated *SWIRL* for transactional workloads yet. While their queries are usually less complex [AR18], such workloads might pose different challenges, e.g., index maintenance costs [Gra06].

Part II.

Unsupervised Database Optimization: Data Dependency-Driven Query Optimization

6 Background: Data Dependency-Driven Query Optimization

In the first part of this thesis, we investigated unsupervised index selection approaches that aim to improve system performance. In the following second part, we examine how *data dependencies*, which are data-inherent properties, can be used for query optimization.

In this chapter, we provide the common terminology and technical background that are necessary to understand the techniques and classifications in the context of data dependency-based query optimization. Related work is presented thereafter as a survey on data dependency-driven query optimization techniques in Chapter 7. The unsupervised aspect is reflected in the identification, validation, and application of the required data dependencies, which will be discussed in Chapter 8.

The following two sections of this chapter summarize the foundations in the area of query optimization (Section 6.1) and the area of data dependencies (Section 6.2). When we introduce the four data dependency types considered in this thesis in Section 6.2, we also discuss automatic discovery and maintenance algorithms of data dependencies, as these make data dependencies more generally available and, hence, can be seen as an enabler for the dependency-based query optimization techniques.

Substantial parts of this chapter have been published in a journal paper [KPN22]. The thesis author conducted the contained literature survey, developed the classification of optimization techniques, and prepared the majority of the original draft. Papenbrock and Naumann supported the survey's design, contributed to the original draft, and improved the material and its presentation.

6.1. Query Optimization

Relational database management systems (DBMSs) are usually queried with SQL in a declarative way. The query engine of a DBMS, then, transforms these queries into physical query execution plans. In this process, *query optimization* is the task of finding an optimal (or at least very good) physical execution plan with respect to the plan's execution time. It is crucial to find efficient query plans, because the execution times of different physical plans (that yield identical results) for the same query can vary by orders of magnitude [Ioa96] and "the runtime system alone could never get that good performance without an optimizer" [Neu14, p. 1739]. In most systems, the query optimization process is handled by an interplay of three main activities: (i) *cost-independent transformation* (also referred to as *query rewriting*), (ii) *cardinality and cost estimation*, (iii) and *cost-based transformation*. In the following, we briefly outline how these three components operate and how the optimization techniques presented in the following survey relate to them. Later, in Chapter 7, we will use the three optimization activities to classify the various dependency-driven optimization techniques.

Note that not all systems follow this division explicitly. Cascades-style optimizers [Gra95], for example, take a combined approach. The data dependency-based query optimization techniques presented below can still apply to such systems.

Cost-independent transformation describes the process of rewriting a query into a semantically equivalent but presumably more efficient query via static, rule-based transformations [PHH92]. The required rules are characterized by being generally applicable instead of being cost-based, that is, they generally[24] produce superior, more efficient query formulations [Pit18]. Examples for such rewrites are the resolution of views, the removal of unnecessary `DISTINCT` clauses and predicate push-downs. For several rewrites, the presence of certain data dependencies, such as information on keys, is a prerequisite to produce semantically equivalent plans. Hence, many of the dependency-based optimization techniques address query cost-independent transformations.

Cardinality and cost estimation serve to estimate the cost of a query plan a priori. Cost is an indispensable metric to compare different plan alternatives for cost-based transformations. Via cost models, a query optimizer estimates the expected cost for a certain query plan and its individual operators based on cardinality information, logical operator complexities, and hardware-specific costs [Man18]. Cardinality information is usually derived from statistics, e.g., histograms or samples. According to Leis et al. [Lei+15], estimation errors are often responsible for suboptimal plans. Data dependencies can be applied during cardinality and cost estimation to obtain more accurate cardinality estimates or estimates for otherwise missing cardinalities; in this way, data dependencies serve to mitigate estimation errors.

Cost-based transformation is the process of improving physical query plans via cost-driven transformations that depend on the database instance at hand [JK84a]. Based

[24]In theory, downstream *cost-based* transformations *could* produce more efficient plans based on the unmodified version. This behavior is unlikely and depends on the specific system and query.

on transformation rules, the optimizer repeatedly generates different plan alternatives, then requests estimates for the plans' costs from the aforementioned cost models based on concrete cardinality information, and finally chooses the most efficient plan based on these estimated costs. Examples for query plan optimizations are predicate reordering, the ordering of join operators, and the selection of execution strategies for operators with different implementations (e.g., hash-, sort-, or index-based joins). It is apparent that the success of such plan optimizations strongly depends on the underlying data, the specific query, and the query's parameter values.

Data dependency information can be applied also in the cost-based transformation phase to generate more efficient plans. A table scan operation, for instance, usually scans the entire table sequentially. With the information that the data is sorted and in-memory, however, the optimizer can instruct the execution engine to execute it as a binary search; and, with functional dependency information, it can substitute the scan attributes with other attributes that are potentially more efficient to scan. Please note that some of the techniques presented in this thesis might require the extension of existing operators or even the implementation of alternative operators, such as binary scans, joins that terminate early, or the caching of subquery results.

6.2. Data Dependencies

Data dependencies usually provide information about multiple attributes, sometimes even across different relations, and how they relate to each other. According to Chapter 7, the four most popular types of data dependencies for query optimization based on the number of optimizations that use data dependencies are *unique column combinations*, *functional dependencies*, *order dependencies*, and *inclusion dependencies*. In this chapter, we give a brief overview of these four types of data dependencies and provide their formal definitions. Afterward, we discuss particular properties of data dependencies that are relevant for query optimization.

Traditionally, data dependencies stem from data modeling and schema design, e.g., 3NF synthesis, or BCNF decomposition, but data profiling identifies data dependencies from the data themselves, independently of such processes. Because the discovery of data dependencies (of any type) is NP-complete [Abe+18] and sometimes even W[2]- to W[3]-complete [DFR98], mining data dependencies is challenging. For this reason, we also provide pointers to the most recent automatic discovery and maintenance algorithms, which in practice are sufficiently fast to be useful in the context of query optimization on real-world datasets. An introductory overview of data profiling techniques can be found in [Abe+18], while a comprehensive survey is given in [AGN15].

6.2.1. Unique Column Combinations (UCCs)

A *unique column combination* is a set of attributes whose projection on some relational instance has no duplicate entry — all entries are unique [Hei+13]. Thus, UCCs functionally determine all other attributes and, hence, are sometimes denoted as *candidate keys* [SS96].

Examples for UCCs are the combined attributes firstname, lastname, address, date-of-birth in a person table, or an auto-incremented id column that is by its definition a UCC. It is worth noting that most relational database management systems recommend the existence of at least one key per relation, i.e., for such systems, we can expect at least one UCC in every relational instance that can potentially be utilized to optimize queries.

In query optimization, UCCs serve to avoid unnecessary duplicate eliminations, i.e., `DISTINCT` calls, obtain improved cardinality estimations, and optimize joins. Due to the relevance of keys in the relational model, unique column combinations are an old and well-established concept in database theory [LO78].

Definition 1 (Unique column combination)**.** *Given a relational instance r over a relation R, we formally say that a* column combination $X \subseteq R$ *is* unique *(UCC) for R, iff $\forall r_i, r_j \in R, i \neq j : r_i[X] \neq r_j[X]$. A UCC is said to be* minimal*, if no subset of that UCC exists for which the above constraint also holds; hence, $\forall X' \subset X : \exists r_i, r_j \in R, i \neq j : r_i[X] = r_j[X]$.*

Origin of UCCs. In general, there are four ways to introduce uniqueness in relational databases: (i) Database users and applications can explicitly produce uniqueness when processing data by utilizing SQL's `DISTINCT` operation; other SQL clauses, such as `EXCEPT`, `UNION`, `INTERSECT`, or `GROUP BY` may also create uniqueness during query execution. In such cases, query optimizers can infer that the data will be unique at a certain point of the query plan. (ii) Most relational database systems allow specifying unique constraints via SQL DDL, such as `UNIQUE` or `KEY` that automatically enforce uniqueness on certain attributes or attribute sets. (iii) Columns or column combinations can be unique by their very nature, such as UUIDs in computer systems or passport_number for the citizens of a particular country. (iv) UCCs can occur by chance[25], especially for column combinations containing many columns.

Discovery and Maintenance of UCCs. Unique column combinations, especially minimal ones, are neither obvious nor simple to determine, as UCC discovery is a problem in $\mathcal{O}(2^m)$ for datasets with m attributes [Abe+18; Liu+12]. Therefore, efficient, automatic UCC discovery algorithms exist, which can serve these unique column combinations to a query optimizer. Examples of such algorithms are DUCC [Hei+13], and

[25]In this context, *chance* means that dependencies do not originate from data modeling or semantics but occur rather randomly due to a large value codomain.

HPIVALID [Bir+20]. With the incremental profiling algorithm SWAN [AQN14], unique column combinations can be incrementally maintained.

While UCCs are a special case of functional dependencies, which we discuss below, specialized UCC discovery algorithms are more efficient. Thus, we discuss the use in query optimization separately.

6.2.2. Functional Dependencies (FDs)

Real-world data often follow semantics according to which values in certain attributes functionally determine values in other attributes. Hence, *functional dependencies* indicate relationships between database attributes [Cod71]. An FD is a statement $X \to Y$, expressing that any two records in a relational instance r that have the same values in the attributes $X \subseteq R$ also have the same values in the attributes $Y \in R$; the attributes in X *functionally determine* the attributes in Y.

For example, for a relation with address data, the functional dependency zip, street_name, latitude $\to$ city should hold; another relation about planets should feature the functional dependency diameter $\to$ circumference.

Functional dependencies are popular properties, because they serve many use cases, including schema normalization, consistency checking, and data exploration. In query optimization, one can use FDs mainly to remove unnecessary attributes in various SQL operations and to improve cardinality estimates by transferring these estimates via FDs between sets of attributes.

Definition 2 (Functional dependency)**.** *A* functional dependency $X \to A$ *of a relation R holds in a relational instance r over R, iff* $\forall s, t \in r : s[X] = t[X] \Rightarrow s[A] = t[A]$*. The left-hand-side (LHS) attributes X are called* determinant *and the right-hand-side (RHS) attribute A is called* dependent*. Multiple FDs with the same determinant attributes X can be grouped and written as $X \to Y$, with $Y = \bigcup A_i$.*

Some FDs have special properties. An FD $X \to A$ is called *minimal* if no attribute $B \in X$ exists such that $X \setminus B \to A$ is still a valid FD. An FD is called *non-trivial* if $A \notin X$. Although minimality and non-triviality play an important role in data profiling, these two properties have no special meaning for query optimization.

A UCC $X \subseteq R$ induces FDs on all attributes that are not part of the UCC: $\forall A \in R \setminus X : X \to A$. Thus, all optimizations that we introduce for functional dependencies later on in Chapter 7 are also applicable to unique column combinations.

Origin of FDs. In general, functional dependencies exist by the very nature of the underlying data: they represent real-world constraints, semantic relationships, and physical laws that are reflected in the data. Functional dependencies can also be artificially introduced to datasets, e.g., with surrogate keys, which by definition functionally determine all other attributes. In addition, after filtering a relation with an equality predicate on A, the FD $X \to A$ holds for every X [Pau00, p. 68]. Paulley explains how scalar functions can generate FDs and discusses for all relational operators which FDs are valid on their output, given a set of FDs on their input [Pau00, pp. 71–104]. Very many further FDs exist coincidentally on a given dataset. While they carry no semantic meaning, they can nevertheless be exploited by query optimization techniques.

Discovery and Maintenance of FDs. Functional dependencies are usually not provided with the data and determining them manually is hard. More specifically, the FD discovery problem is in $\mathcal{O}(2^m \cdot (\frac{m}{2})^2)$ for datasets with m attributes [Abe+18; Liu+12]. Therefore, a variety of automatic FD discovery algorithms, such as TANE [Huh+99], FDEP [FS99], and HYFD [PN16] have been proposed. The surveys by Liu et al. [Liu+12] and Papenbrock et al. [Pap+15a] present and compare different FD profiling techniques from both a theoretical and a practical perspective, respectively. To discover FDs incrementally or maintain the FDs of a dynamic dataset over time, various incremental profiling algorithms exist [Car+19; Sch+19; Wan+03].

6.2.3. Order Dependencies (ODs)

An *order dependency* (OD) is a statement of the form $\boldsymbol{X} \mapsto \boldsymbol{Y}$[26] specifying that ordering a relational instance r by the attribute list $\boldsymbol{X} \subset R$ also orders r by the attributes $\boldsymbol{Y} \subset R$. Given an OD, we thus know how order decisions on certain attributes propagate to orders of other attributes; this knowledge can be used to optimize order decisions.

To give an intuition, in a date table, the OD month $\mapsto$ quarter holds, but the inverse OD quarter $\mapsto$ month is not valid. Another typical example is salary $\mapsto$ taxrate. One can find many ODs in real-world datasets [SP22]. Order dependencies play an important role in query optimization because many relational operators use sorting or exploit already sorted data for their execution. More specifically, ODs help in selecting suitable operator implementations and support building efficient query plans. They can also be used to, for instance, effectively rewrite, remove, or inject `ORDER BY` clauses.

Ginsburg and Hull were the first to formally introduce the concept of order dependencies [GH83; GH86]. In this thesis, we use the notation of Szlichta et al. [SGG12b].

Definition 3 (Order dependency)**.** *For two lists of attributes $\boldsymbol{X}$ and $\boldsymbol{Y}$ of a relation R, the* order dependency *(OD) $\boldsymbol{X} \mapsto \boldsymbol{Y}$ holds in relation instance r over R, iff $\forall s, t \in r$: $s[\boldsymbol{X}] \preceq t[\boldsymbol{X}] \Rightarrow s[\boldsymbol{Y}] \preceq t[\boldsymbol{Y}]$.*

Note that the comparison operator $\preceq$ compares the $\boldsymbol{X}$ and $\boldsymbol{Y}$ values attribute-wise, i.e., lexicographically via $\leq$ with the first attribute in each list being the most significant

[26]Bold symbols $\boldsymbol{X}$ indicate lists, whereas standard symbols X indicate sets.

one. Following SQL semantics, the comparison is data type specific, which means that it is numerical for numbers and lexicographical for strings. In principle, ODs support different comparators including $\prec$, $\preceq$, $=$, $\succeq$, and $\succ$ as well as combinations of these comparators. An OD with the comparator $=$, for example, is equivalent to a functional dependency [SGG12b] and $\forall s, t \in r : s[\boldsymbol{X}] \preceq t[\boldsymbol{X}] \Rightarrow s[\boldsymbol{Y}] \succeq t[\boldsymbol{Y}]$ means that an ascending $\boldsymbol{X}$ order implies a descending $\boldsymbol{Y}$ order. While the optimizations presented below are often extendable to other comparator types, we assume the comparator to be $\preceq$ ($\overset{\preceq}{\mapsto}$), if not stated differently. Please note that $\preceq$ induces a total ordering.

Origin of ODs. There are three ways to introduce order in relational data: first, users can explicitly create order by specifying an `ORDER BY` clause; second, some database operations produce ordered results as a side-effect of their implementation, e.g., sort-merge joins or sort-based aggregates; third, data can also be naturally ordered by, for instance, timestamp or auto-incremented surrogate key attributes during data ingestion.

Discovery and Maintenance of ODs. The discovery problem for ODs (in set-based notation) is in $\mathcal{O}(2^m)$ with m being the number of attributes in the dataset [Szl+17]. However, automatic discovery algorithms, such as FastOD by Szlichta et al. [Szl+17] or DISTOD by Schmidl et al. [SP22], are efficient in practice, because they use clever search space pruning and most ODs in real-world datasets actually appear relatively early in the discovery process. Similar to the other types of data dependencies, the incremental discovery has also been studied for ODs. For example, a recent incremental discovery algorithm for point-wise ODs is IncPOD [Tan+20].

6.2.4. Inclusion Dependencies (INDs)

If all values in the projection of some attribute combination $\boldsymbol{X}$ also occur in the projection of some attribute combination $\boldsymbol{Y}$ (of the same or a different relation), then an *inclusion dependency* exists between $\boldsymbol{X}$ and $\boldsymbol{Y}$. If, furthermore, the referenced attribute combination is a key, i.e., a UCC, for its relation, the inclusion dependency is a *foreign-key* candidate — in other words, INDs are prerequisites for a foreign-key relationships. While relational database *keys* serve to identify entities within a single table, *foreign-keys* link entities across tables to connect tables and indicate join paths. For example, the three inclusion dependencies

$$\textsf{click.website} \subseteq \textsf{website.url}$$
$$\textsf{sales.item} \subseteq \textsf{items.id}$$
$$\{\textsf{ship.lname, ship.bdate}\} \subseteq \{\textsf{addr.name, addr.dob}\}$$

might represent foreign-key relationships. Such inclusion dependencies can be used in data integration and data linkage scenarios to connect tables across multiple data sources by suggesting join paths. Inclusion dependencies are, however, also useful if they do not describe foreign-key relationships. The inclusion dependency nails.supplier ⊆ screws.supplier, for example, asserts that all nail suppliers also supply screws without supplier being a key in screws. Such general INDs can be used for data exploration and, as we will show in Section 7.5, query optimization.

Our formal definition of inclusion dependencies follows the syntax of Casanova et al. [CFP82] and De Marchi [MLP09]:

Definition 4 (Inclusion dependency)**.** *An* inclusion dependency *(IND) $R_i[\boldsymbol{X}] \subseteq R_j[\boldsymbol{Y}]$ is valid for the two relational instances r_i and r_j of schemata R_i and R_j and the attribute lists $\boldsymbol{X}$ and $\boldsymbol{Y}$ with cardinalities $n = |\boldsymbol{X}| = |\boldsymbol{Y}|$ iff $\forall t_i \in r_i, \exists t_j \in r_j : t_i[\boldsymbol{X}] = t_j[\boldsymbol{Y}]$. We write $\boldsymbol{X} \subseteq \boldsymbol{Y}$ or $R_i.\boldsymbol{X} \subseteq R_j.\boldsymbol{Y}$ if it is clear from the context that an IND is meant; in these cases, the projection is implicit.*

Note that the dependent ($\boldsymbol{X}$) and referenced ($\boldsymbol{Y}$) part denote attribute *lists* for INDs, i.e., their attribute order may differ from the order in R and they may contain repeated attributes. By removing attributes with the same indices from the lists $\boldsymbol{X}$ and $\boldsymbol{Y}$, we can derive generalizations $\boldsymbol{X}' \subseteq \boldsymbol{Y}'$ with $\boldsymbol{X}' \subset \boldsymbol{X}$ and $\boldsymbol{Y}' \subset \boldsymbol{Y}$ from a valid IND $\boldsymbol{X} \subseteq \boldsymbol{Y}$ that are also valid. This is important for query optimization, because a query might not contain all attributes of a known IND but a subset that also forms a valid IND.

Origin of INDs. Relational databases contain INDs, because data models representing real-world data often rely on relationships between tables, and INDs are a prerequisite for foreign-key relationships. Most DBMS implementations allow users to specify foreign keys, e.g., via SQL's `FOREIGN KEY X REFERENCES Y`, which can also specify the behavior if a tuple is `NULL` in one of the key's attributes via `MATCH (FULL|PARTIAL|SIMPLE)`.

Discovery and Maintenance of INDs. Inclusion dependencies are different from the previously mentioned types of dependencies as they can span across multiple relations, the attribute order in the dependency *does* matter, and the position of value combinations within the sets of left- and right-hand-side value combinations *does not* matter. For this reason, their discovery is in $\mathcal{O}(2^m \cdot m!)$ with $m!$ being a simplification [Abe+18; Liu+12]; it is even one of only few real-world $W[3]$-complete problems [BFS17] and, hence, particularly hard. Nevertheless, many data profiling algorithms, such as BINDER [Pap+15b] or SINDY [KPN15], are able to discover INDs in most relational datasets, as experimentally surveyed in [Dür+19]. The maintenance of INDs for dynamic datasets is possible with incremental discovery algorithms, such as S-INDD [SM17].

6.2.5. Properties of Data Dependencies

Dynamic datasets change through inserts, updates, and deletes of records. For this reason, we have already referenced some incremental and dynamic data profiling algorithms that are able to maintain the knowledge about valid data dependencies over time. *Up-to-dateness* is, however, only one of many properties of data dependencies. We briefly discuss other properties and their relevance for query optimization.

Minimality and Completeness. All state-of-the-art data profiling algorithms mine only *minimal* (or *maximal*) data dependencies, because the sets of *all* valid dependencies are usually extremely large. Query optimizers, however, might require non-minimal (or non-maximal) dependencies, which is why these dependencies need to be inferred from discovered dependency sets. In practice, a query optimizer will need to deal with *incomplete* data dependency sets. Fortunately, data dependencies follow certain axiomatizations that enable the simple generation of further dependencies. Functional dependencies, for example, follow Armstrong's axioms (reflexivity, augmentation, and transitivity [Arm74]) that generate additional, also valid FDs from existing FDs. We refer the interested reader to the survey of data dependencies for query optimization [KPN22] for more details on minimality and completeness of data dependencies. Later, in Chapter 8 we present our approach on unsupervised data dependency discovery that also focuses on the challenge of discovering only relevant dependencies.

Approximation. Approximate, partial, and relaxed dependencies are ones that are not valid for the entire dataset [CDP16; Huh+98]. They are produced by approximate (and usually more efficient) discovery algorithms and arise from exact dependencies on dynamic datasets if the exact dependencies are not maintained. When used for query optimization, approximate dependencies can cause incomplete and incorrect results. Hence, they are in general not usable for query optimization unless they are implemented in approximate query processing systems [NK04], other data structures compensate their optimization mistakes [KBS20], or they are used only for cardinality and cost estimation optimizations.

Conditions. Data dependencies are sometimes tied to conditions that limit their scope. A conditional dependency [Boh+07] holds on only a particular subset of tuples for which a specific condition is true. Such dependencies can be used for query optimization in the same way as unconditional dependencies if the query's filter condition (`WHERE` clause) is at least as strict as the condition of the dependencies. In summary, all query optimization techniques surveyed in this thesis require a set of exact[27] data dependencies (or constraints) as input, regardless of whether these constraints are given by the schema or have been discovered from the data.

[27]Optimizations applied during cardinality and cost estimation can also utilize approximate dependencies.

Null Semantics. Relational database systems often use `NULL` values to indicate missing information. The comparison of `NULL` values, i.e., `NULL` = `NULL` evaluates to `unknown` [Cod75], which is sometimes effectively treated as `TRUE` or `FALSE` in SQL. For instance, `DISTINCT`, `GROUP BY`, and `ORDER BY` statements as well as set operations evaluate `NULL` = `NULL` to `TRUE` while filters via `WHERE` and `JOIN` statements evaluate to `unknown` which, in turn, does not satisfy the predicate. So whenever a dataset may contain `NULL` values and we use a data dependency for query optimization, then this dependency needs to be true under the same `NULL` semantics as the SQL operator that is being optimized. For example, a `DISTINCT` can be removed only with a UCC that uses `NULL` = `NULL` semantics and a `JOIN`-removing IND needs to use `NULL` ≠ `NULL`. Because we can configure the `NULL` semantics in most data profiling algorithms, both semantics are technically available, and the optimizer can pick the required ones. Unfortunately, `NULL` semantics are hardly considered in the surveyed literature, which is why we add this information where it is relevant below. To shorten the individual discussions, we define that, if not stated otherwise, all required data dependencies use the `NULL` = `NULL` semantics, which is not only the most commonly required interpretation but also the default configuration for most dependency discovery and maintenance algorithms.

Note that both semantics `NULL` = `NULL` and `NULL` != `NULL` are practical `NULL` interpretations. While this practical interpretation is very useful for our objective of query optimization, a more accurate interpretation of `NULL` values for data dependencies is actually *no information* [AM86], so that the validity of a dependency depends on whether we can find a substitution for all `NULL` values that makes the dependency true (possible world) or we find that any substitution of all `NULL` values makes the dependency true (certain world) [KL16; KLZ15]. Albeit interesting for schema design, semantic reasoning, and many other use cases, possible and certain world interpretations are not relevant for the surveyed query optimization techniques.

7 A Survey of Data Dependency-Driven Query Optimization

Query processing can be accelerated through advancements in different areas, such as utilizing new hardware technologies, improved implementations of database operators, unsupervised physical database design techniques (cf. Part I), or sophisticated query plan modifications as part of the query optimization process. In the following, we consider how *data dependencies* can be utilized for more efficient query processing. More specifically, we provide a comprehensive survey of methods that exploit data dependencies during query optimization and execution to process relational database queries more efficiently.

Most of the data dependency-driven methods discussed in the following have been known for years, but many of them are rarely implemented in existing database systems. One reason for not using data dependencies in modern query processing engines might be that the required data dependencies of a given dataset are often unknown: they have never been specified, are challenging to discover, and can be expensive to maintain. However, the latest developments in data profiling tackle these issues for various types of data dependencies [AGN15; Bir+20; KPN15; Pap+15b; PN16; SP22]. The fact that data profiling algorithms usually discover all valid dependencies in a given dataset is considered later by our workload-driven discovery approach in Chapter 8. Combining modern data profiling algorithms with our workload-driven discovery approach increases the availability of data dependencies for query optimization.

In many use cases, it is crucial to distinguish semantically meaningful dependencies from accidentally valid, i.e., spurious ones. However, for query optimization, the distinction between *genuine* and *spurious* dependencies is irrelevant: if a data dependency is valid, it can be used for optimization — regardless of its genuineness or future correctness.

In the literature, data dependency-driven query optimization techniques have only been partially investigated. An extensive examination of order and uniqueness properties in the context of operators of the relational algebra has been given by Paulley's thesis [Pau00]. This chapter incorporates parts of his findings but also further and more recent research: we include numerous additional optimization techniques and additional types of dependencies. To make the techniques more accessible and facilitate future implementations, we provide concise, intuitive descriptions and links to detailed material.

Substantial parts of this chapter were published in a journal paper [KPN22]. The thesis author's detailed contributions to this paper were discussed at the beginning of Chapter 6.

In this chapter, Section 7.1 provides an overview and classifications for the optimization techniques presented in Table 7.1. The table summarizes the optimizations for different data dependency types in different application areas of the query optimization process. This table also serves as a reference for all of the following descriptions of optimization techniques. Sections 7.2 to 7.5 subsequently describe the proposed optimizations, including examples where appropriate, for all four dependency types. Section 7.6 discusses further optimization opportunities before Section 7.7 provides a short summary of this chapter, concluding remarks, open research questions, and ideas for future work.

7.1. Classification of Dependency-Driven Query Optimization Techniques

This section provides an overview of all optimizations that are explained in detail in the following Sections 7.2 to 7.5. The *optimization technique matrix* in Table 7.1 shows which dependencies enable a particular optimization and which query optimization activity is affected by each optimization. More specifically, the matrix classifies all dependency-driven query optimization techniques with respect to three dimensions:

i. *Dependency type:* Dependency-based query optimization techniques are tied to dependency types because a particular optimization is only enabled if a dependency of the required type exists. Additionally, the discovery and maintenance approaches differ significantly for different dependency types. For these reasons, we chose the dependency type — UCC, FD, OD, or IND — as the main classification dimension.

ii. *Relational operator:* Every optimization targets a particular operator or set of operators. For this reason, we use the operators of the relational algebra as second classification criterion. In practice, most optimizations target only one operator. For those optimizations that affect multiple operators, we show the most relevant operator, which is usually the most expensive one.

iii. *Query optimization activity:* The optimizations are assigned to the query optimization activity that is mainly affected by it. Hence, we use the three activities discussed in Section 6.1 — cost-independent transformation, cost-based transformation, as well as cardinality and cost estimation — as the third classification dimension.

Table 7.1.: Possible optimizations categorized by (i) the examined data dependencies, (ii) the area of application, i.e., operators of the relational algebra, and where possible (iii) the query optimization activity that is affected by the optimization, which is indicated by background color and symbol: cost-based plan transformations (†), cost-independent transformations / rewriting (∗), cardinality and cost estimation (‡). For optimizations that have not been scientifically published, we mention the section where this optimization is explained.

Application area	**U**nique **C**olumn **C**ombinations (Sec. 7.2)	**F**unctional **D**ependencies (Sec. 7.3)	**O**rder **D**ependencies (Sec. 7.4)	**In**clusion **D**ependencies (Sec. 7.5)
Join	• Spurious-free back-joins [YL87] ∗ • Semijoin tranformation [MySa] ∗ • Pipeline with grouping [Day87; Yan95] † • Invisible join [AMH08] †	• Simplification / avoidance [KY83] ∗ • Complexity reduction (Sec. 7.3.2) ∗ • Self-join avoidance [AHV95] ∗ • Plan generation [EFM16] †	• Join avoidance [Szl+11; Szl+14] ∗ • Pipeline with grouping [CS94; Gra93] † • Avoid sort for sort-merge-joins [GHK92; Pau00; SSM96] † • Attribute substitution (Sec. 7.4.2) † • Pipeline index scan with join [Gra93] †	• Join elimination [Che+99; JK84b] ∗ • Substitute relations [DPT06] † • Avoid semijoin reductions (Sec. 7.5.1) † • Accurate cardinalities [IBM22b] ‡
Selection	• Early abort (Sec. 7.2.6) ∗ • Accurate cardinalities (Sec. 7.2.6) ‡	• Early abort (Sec. 7.3.3) ∗ • Substitute attributes [Che+99; Kim+09] † • Estimations w/o independency assumption [CGR01; Ily+04; SSY98; Thed] ‡	• Use binary search [Pau00] †	
Grouping & Aggregate functions	• Grouping is redundant [CS94] ∗ • Accurate cardinalities (Sec. 7.2.6) ‡	• Reduce attributes [BNE13; SGG12b] ∗	• Simplify `MIN`, `MAX`, `MEDIAN` [ONe94; Pau00] ∗ • Sort-based grouping [Pau00; SSM96; Szl+14; YL94; YL95] †	
Projection & Distinctness	• Avoid `DISTINCT` [Pau00; PL94; PHH92] ∗	• Distinctness: see grouping [Wed92] ∗ • Simplification [DD92] ∗ • Estimate projections [GG82] ‡	• Distinctness: see grouping †	
Sorting	• Reduce attributes (Sec. 7.2.6) ∗ • Unstable sorting (Sec. 7.2.6) ∗	• Reduce attributes [Che+99; SSM96; SGG12b] ∗	• Reduce attributes [SGG12a; SGG12b; Szl+14] ∗ • Avoid sort [Gra93; Pau00] ∗ • `ORDER BY` with index [Szl+14] ∗ • Main-memory sorts [Szl+14] † • Substitute attributes [Pau00] † • Accurate estimates (Sec. 7.4.4) ‡	
Set Operations	• `EXCEPT` to `EXCEPT ALL` [Pau00] ∗ • `INTERSECT` to `INTERSECT ALL`[PL94] ∗ • `INTERSECT` to join [Pau00; PL94] † • Accurate cardinalities (Sec. 7.2.6) ‡		• Order optimizations [Pau00] †	• Simplify `UNION` (Sec. 7.5.2) ∗ • Simplify `INTERSECT` (Sec. 7.5.2) ∗ • Eliminate `EXCEPT` (Sec. 7.5.2) ∗ • Accurate cardinalities(Sec. 7.5.2) ‡
Other	• Subquery to join [Pau00; PL94] ∗ • Subquery sort avoidance [Sel+79] †	• Scalar subqueries [DD92]	• Subquery memoization cache [Pau00] ‡ • Sparse over dense indexes [DH82]	• Query folding [DPT06; Gry98; Ile+14] ∗ • Eliminate correlated subqueries [Mic04] (Sec. 7.5.2) ∗

For some systems, techniques might be classified differently w.r.t. their query optimization activity depending on the optimizer's implementation and degree of sophistication. For our classification, we place the techniques into the most likely categories. We also emphasize that some optimizations affect not only the structure of the query plan (e.g., operator reordering) or the choice of operator implementations (e.g., sequential vs. index scan) but also the behavior of operators at runtime. Scan operations, for instance, might abort early in certain cases. We attribute such optimizations to query optimization instead of the execution phase: ultimately, the optimizer prepares for such behavior, decides on the query plan, and instructs the physical operators to act accordingly.

Counting the entries, Table 7.1 shows that more optimizations exist for UCCs (17), FDs (13), and ODs (18) than for INDs (10): INDs are dependencies between unordered sets of values (or value combinations); their known optimizations, therefore, support only join and set operations. Most optimizations influence *cost-independent transformations* (29), followed by *cost-based transformations* (18) and lastly *cardinality and cost estimation* (9).

We also identify a fourth dimension, in which the presented transformation-based optimization techniques could be categorized: the *optimization method* with which the improvement is achieved. This dimension is not visualized in the table, because it does not apply to all depicted techniques.

- *Simplification:* The task that the operator needs to fulfill is simplified, e.g., by removing attributes from a `GROUP BY` list or by omitting the sort phase of a sort-merge join. Avoiding the operator execution altogether is an extreme case of simplification. Examples can be found in [AMH08; AHV95; BNE13; CS94; Che+99; DD92; Gra93; KY83; Mic04; MySa; ONe94; Pau00; PL94; PHH92; Sel+79; SGG12a; SGG12b; Szl+11; Szl+14; Wed92; YL87].
- *Algorithm choice:* Oftentimes, a specific implementation of an operator can be selected from different alternatives. The available data dependencies can guide this decision. For instance, a binary search is usually superior to a sequential scan if a dependency indicates that the data is sorted. Examples are found in [GHK92; Pau00; SSM96; Szl+14; YL94; YL95].
- *Substitution:* Certain data dependencies indicate that an operator can, instead of processing an attribute A on a relation R, process an attribute B on a relation S with the same result. This might be beneficial if B offers superior properties, such as a more compact data type or being indexed. Examples are presented in [Che+99; DPT06; Gry98; Ile+14; Kim+09; Pau00].
- *Pipelining:* Some dependencies provide additional guarantees that enable pipelining between operators in cases where it would usually be infeasible. Examples are found in [CS94; Day87; Gra93; Yan95].

In conclusion, Table 7.1 provides an overview of all optimization techniques presented in this chapter and categorizes them to show which optimizations are enabled by each dependency type.

7.2. Unique Column Combinations

In the following subsections, we present various query optimization techniques that are enabled by the existence of UCCs. (Primary) keys are by definition UCCs, which, vice versa, serve as key candidates [SS96]. For that reason, all presented optimizations can be applied analogously given either key constraints on the schema or UCCs discovered from a relational instance. Apart from query optimization and keys, uniqueness is often used for data integration, indexing, and anomaly detection.

7.2.1. UCCs and Joins

If an SQL query joins two relational instances that both have a UCC, then the resulting relation contains a (not necessarily minimal) UCC, which is the concatenation of both UCCs. Furthermore, if the join attributes are UCCs in both relational instances, they are also unique for the join result. These two properties can be used to track unique column combinations across joins, so that they may be used for the optimization of downstream operators and query plans.

Most UCC-based join optimizations require the existence of uniqueness on at least one of the input relations' join attributes. For example, Dayal shows that in certain cases, aggregations and joins can be pipelined if the grouping attributes contain the primary key, i.e., a UCC of the outer join relation [Day87]. Viable join implementations group the result by tuples of the outer relation's join attributes. Consider the query:

```sql
SELECT R.A, SUM(S.B) FROM R, S
    WHERE R.A = S.A
    GROUP BY R.A.
```

If $R.A$ is a UCC and chosen as outer relation, the join's results can be streamed directly to the aggregate function `SUM(S.B)`, because the absence of duplicate values guarantees that the records are already grouped by $R.A$. Yan specifically states that this applies to candidate keys and UCCs if these are the outer relation's join attributes [Yan95].

An inner join of two relations R and S on the join attributes X can be executed by potentially more efficient semijoin[28] strategies [MySa] if X is a UCC on S, and S does not need to supply columns to the result. Intuitively, the UCC ensures that any tuple $r_k \in R$ from the outer relation can match only one single tuple $s_i \in S$ of the inner relation [CS94],

[28]The semijoin $R \ltimes S$ selects tuples of R that would match a tuple of S if joined, i.e., $R \ltimes S \Leftrightarrow R \bowtie_{\boldsymbol{X}} \pi_{\boldsymbol{X}}(S)$.

i.e., $r_k[X] = s_i[X]$ is unique with respect to $r_k[X]$. This is true, because by the definition of UCCs no second tuple $s_j \in S, i \neq j$ exists for which $s_i[X] = s_j[X]$. If X was not a UCC in S, the inner join could replicate rows in R's instance and, hence, produce a different result than the semijoin. Rewriting inner joins as semijoins is useful, for example, in distributed query optimization, to send less data over the network [Ber+81; Mul90]. For nested loop join strategies, a UCC on the inner join loop's attribute enables aborting the inner loop early and continuing the outer loop as soon as the first match is found.

Yang and Larson illustrated another use case for UCCs when working with derived relations and so-called *back-joins* [YL87]: let us assume a derived relation E_1 from R holds all the required tuples to answer a query Q, but misses some attribute A. With a back-join[29], the attribute A can be obtained from another relation E_2 that was derived from R as well. Such a back-join can produce *spurious* tuples that contain values that originate from different tuples of the base relation R. Performing this back-join on a UCC with attributes X, however, prevents the generation of such spurious tuples, because, by the definition of UCCs, if tuples agree on X, they also agree on all other attributes.

Abadi et al. [AMH08] introduce so-called *invisible joins* for star schemas in column-oriented database engines. This technique improves the performance of foreign- to primary-key joins by, among others, transforming such joins into predicates on fact table columns. Given the required UCC, i.e., a primary key, the optimizer can choose this specialized execution technique to improve the performance of the join. Because the detailed description of this join technique is beyond the scope of this thesis, we refer the interested reader to [AMH08].

7.2.2. UCCs and Grouping

When grouping on a UCC, it is by the definition of UCCs obvious that the maximum group size is one. For this reason, the entire grouping step is superfluous to calculate aggregations on these groups: the data is implicitly grouped already [CS94]. Hence, both sort- and hash-based aggregation implementations can omit the grouping phase, i.e., sorting or hashing, if they are aware of the UCC.

Because UCCs are essentially special forms of functional dependencies, we list further UCC-based optimizations for grouping and aggregation operations with the FD-based optimizations in Section 7.3.1.

[29]A *back-join* is used when a derived relation holds all necessary tuples to answer a query but requires additional attributes from other relations.

7.2.3. UCCs and Distinctness

SQL statements containing the `DISTINCT` keyword are common in practice [Pau00]. Being able to optimize duplicate eliminations is, for this reason, very important.

Paulley and Larson explain that query results in certain combinations with UCCs cannot contain duplicate tuples and, hence, the execution of a `DISTINCT` operation is unnecessary [Pau00; PL94]. Given a UCC X on a relation R, they show: if either (i) all attributes of X are part of the query's projection list, or (ii) a subset $Y \subset X$ is contained in the projection list and the other attributes $X \setminus Y$ are selected via equality predicates, the query result is unique and the `DISTINCT` operation can be removed. Pirahesh et al. mention a similar technique, but not as detailed as Paulley and Larson and without explicitly taking `NULL` values into account [PHH92]. Since distinctness is usually ensured by costly sort- or hash-based approaches, the removal of redundant `DISTINCT` keywords is a substantial optimization.

7.2.4. UCCs and Subqueries

Paulley and Larson describe how correlated subqueries can be transformed into ordinary join queries [Pau00; PL94]. Optimizers can then apply all the rules and optimization techniques that are relevant to joins, e.g., choosing a particular join algorithm with better performance for that particular case or adjusting the join order to find a more efficient query plan [PHH92]. While these techniques have been proposed earlier, e.g., by Kim [Kim82], Ganski et al. [GW87], and Pirahesh et al. [PHH92], Paulley and Larson explicitly consider duplicate entries and `NULL` values.

To illustrate the idea,

```
SELECT R.A, R.B FROM R WHERE EXISTS
    (SELECT * FROM S WHERE S.A = R.A)
```

can safely be rewritten to

```
SELECT R.A, R.B FROM R, S WHERE S.A = R.A
```

iff $S.A$ is a UCC; otherwise, the transformed version of the query might result in more results than the original version. In case a rewrite was possible, also the semijoin strategies might apply to further optimize the query [NK15]. The UCC ensures that the subquery cannot provide more than one matching tuple, which enables the transformation. The rewrite is also possible for a multi-attribute UCC X if the non-join attributes are selected via equality predicates (see Section 7.2.3).

Sometimes, correlated subqueries cannot be unnested. In such cases, the query engine can cache results obtained from subquery evaluations to reuse these results for evaluations with repeated, i.e., the same referenced values. With the cached results, redundant subquery evaluations can be avoided. Selinger et al. develop this idea one step further and propose to first sort the outer relation by the referenced column and then execute the subqueries [Sel+79]; in this way, the query engine needs to cache only one, i.e., the last subquery result. However, if the referenced column is a UCC, both the caching and the sorting are redundant and should not be applied, because repeated values do not exist [Sel+79].

7.2.5. UCCs and Set Operations

While the relational algebra is based on set semantics, SQL generally uses bag semantics. SQL's set operations, such as `INTERSECT`, `EXCEPT`, and `UNION`, however, provide set semantics, unless disabled by the `ALL` keyword [Int92]. By the set definition, sets cannot contain duplicate values, but relations in database systems typically allow them. Therefore, implementations of set operations need to ensure that duplicates are removed before providing the final result.

Because uniqueness plays a central role for set operations, it is apparent that UCCs can be used to optimize them. For the following examples, we assume two tables R and S with a UCC on $R.A$. Paulley and Larson note that "the semantics of `INTERSECT` and `INTERSECT ALL` are equivalent if at least one of the involved tables cannot produce duplicate rows" [Pau00; PL94]. If this precondition is guaranteed by a UCC, the costly duplicate elimination of `INTERSECT` can be avoided by rewriting it as an `INTERSECT ALL`. The rational is as follows: according to the SQL standard [Int92, p. 202], the result of an `INTERSECT ALL` statement on the tables R, S contains $min(m, n)$ instances of a duplicate tuple t, where m and n are the numbers of occurrences of t in R and S. Hence, the UCC guarantees either m or n to be 1 which, in turn, guarantees only a single occurrence of t in the result. Thus, no duplicate elimination is necessary.

Similarly, R `EXCEPT` S can be rewritten to R `EXCEPT ALL` S, simply because a difference operation cannot introduce duplicates if these are not already present in R [Pau00]. Their absence is guaranteed by the UCC on $R.A$.

In addition, some database systems, such as MySQL, do not support `INTERSECT` statements [MySb]. Instead, the documentations of these systems often suggested to express the semantics of `INTERSECT` (manually) with an inner join:

```
SELECT DISTINCT(R.A) FROM R, S
    WHERE R.A = S.A
    OR (R.A IS NULL AND S.A IS NULL).
```

If a UCC exists not only on $R.A$ but also on $S.A$, the `DISTINCT` can be removed and, hence, `INTERSECT` can be formulated as a join. The correct handling of `NULL` values for such rewrites can be achieved as demonstrated above [Pau00; PL94]. The main advantage of these transformations is that they help to avoid costly duplicate removals [Jac05] when these are superfluous.

7.2.6. Further Optimization Opportunities with UCCs

Apart from the main concepts described above, unique column combinations enable some further potential optimizations.

Selection. If a `SELECT` clause defines an equality predicate for all attributes of a UCC, the query can be aborted after the first matching tuple is identified because the UCC guarantees that no other matches can be found. While such an operation on primary-key columns would be handled usually by an index lookup, not all UCCs are necessarily indexed.

Cardinality and Cost Estimation. Uniqueness information can be used for cardinality and cost estimation of various operators. For example, equality predicates on UCC attributes can have at maximum one resulting tuple. Non-equal checks result in either $|R|$ or $|R| - 1$ results. For equality joins, there is at most a single matching tuple, cf. Section 7.2.1. Furthermore, uniqueness information can be utilized to determine the number of groups for `GROUP BY` operations and to estimate lower bounds for set operations.

Sorting. The presence of UCCs allows to remove all attributes following that UCC in the attribute list of an `ORDER BY` clause. For instance, under the UCC X, the attributes Y can be removed from `ORDER BY X, Y`, because sorting by Y would affect only the order of tuples that agree in X. The UCC guarantees that such tuples do not exist.

Stable sort algorithms keep equal elements in their original order. This guarantee is usually exchanged for higher runtime complexity or elevated memory consumption. However, UCCs ensure that no duplicate elements exist so that stable sort algorithms are not needed. So if the execution engine offers multiple physical sort operator implementations, UCCs can be used to find the most suitable implementation during optimization.

Embedded Unique Constraints. Embedded unique constraints allow for expressing the existence of UCCs on fragments of incomplete data, i.e., data containing `NULL` values. Wei et al. demonstrate how such embedded unique constraints can be utilized for query optimization, e.g., to improve the efficiency of joins or scans in queries that handle incomplete data [WLL19].

7.3. Functional Dependencies

The use cases for functional dependencies are manifold. They were initially used to normalize database schemata [Cod70], but meanwhile also aid data cleansing, data integration, and data translation tasks. In the remainder of this section, we focus on their utilization for query optimization.

7.3.1. FDs and Grouping

Grouping operations can be simplified through query rewriting if FDs are present. If the functional dependency $B, C \to A$ holds and a group operation on the combination B, C, A is to be executed, the grouping attribute A is unnecessary, because all elements that fall into the same group for B, C necessarily also have the same value for A. Thus, the grouping attribute A can simply be removed and it is sufficient to group only on B, C [SGG12b]. The same rule applies if the determinant B, C is (partly) filtered in the `WHERE` clause with an equals condition. For example, the `GROUP BY` clause in the query

```
SELECT C, SUM(D) FROM R WHERE B = 17
    GROUP BY C, A
```

can be rewritten to `GROUP BY C`. Since 1999, the SQL standard explicitly allows selecting columns that are not part of the `GROUP BY` clause if they are functionally dependent on grouping columns [Int99]. Date and Darwen [DD92] mentioned this problem earlier in their work. The presented technique can in fact also be applied to several TPC-H queries [BNE13]. For example, TPC-H Query 10 contains a grouping operation on c_custkey, c_name, c_acctbal and further attributes. Because c_custkey is a key, c_custkey $\to$ c_name, c_acctbal, [...] holds. The grouping statement in TPC-H Query 10 can, therefore, be simplified to group on c_custkey only.

As the enforcement of distinctness is a special case of grouping [CS94], the above-described method can also be applied to SQL `DISTINCT` clauses: `DISTINCT X, A` reduces to `DISTINCT X` if $X \to A$. In the context of duplicate elimination, Weddell also explained how duplicate eliminating projections (in the sense of the relational algebra) can be avoided with known functional dependencies [Wed92].

Date and Darwen also partly describe an optimization if a `DISTINCT` is applied after a relation has been filtered with an equals predicate [DD92]: consider $A \to B$ to hold and the query

```
SELECT DISTINCT B FROM R WHERE A = 4.
```

In this case, the `DISTINCT` is necessary, because the selection can possibly yield multiple results. However, above's FD ensures that all resulting rows will have the same value in B, and a costly duplicate elimination can be avoided by just returning the first row.

7.3.2. FDs and Joins

The optimization potential of FDs for joins might appear limited at first glance, because joins test for value correspondences *across* potentially different relations, while FDs test for value dependencies *within* one relation. There are, however, a few interesting applications related to joins in query rewriting and join ordering.

First, Eich et al. examine optimizations for eager aggregation that were initially presented by Yan and Larson [YL95], i.e., group-by operations that are pushed below joins. The authors show how functional dependencies can be used to prune join trees during query plan generation and, hence, speed up the plan generation process by orders of magnitude [EFM16]. For example, consider the question whether a join subtree T_1 that is more expensive than a join subtree T_2 can be pruned during plan generation. In T_1, attribute A of a relation R is (eagerly) grouped before R is joined with S. In T_2, $R \bowtie S$ is executed before attributes A, B (B is an attribute of S) are grouped. Now, T_1 should only be pruned if at least the same set of FDs holds after executing T_2 as after T_1. The intuition is as follows: the execution of a group-by influences which functional dependencies hold in the subsequent intermediate results; without grouping, the FDs would be equivalent for all join trees. If the FDs that are necessary to fulfill the specified query do not hold in the end, a final group-by would have to be added to ensure the correct result. This final group-by operation could introduce additional costs that, when taking the aforementioned pruning criteria into account, renders the plan suboptimal. Note that *early partial aggregation* could even reduce the overall costs [LG09].

Kambayashi and Yoshikawa [KY83] apply FDs to simplify queries that involve joins. Assume a natural join $R \bowtie S$ that joins the relations R and S on the attributes Z with $S[Z] \subseteq R[Z]$ and vice versa. If the FD $X \rightarrow A$ holds and $X \subseteq Z$, then the join attributes Z can be reduced to $Z \setminus A$, because matching values in X guarantee matching values in A. The same reduction also applies to arbitrary self-joins $R \bowtie_Z R$ on same attributes Z, if A, i.e., the removed attribute, does not contain `NULL` values; such `NULL` values would prevent certain result pairs that match on $Z \setminus A$ and would, therefore, arise after the reduction. The described rewriting technique improves the join execution, because fewer join attributes can reduce both computation time and memory consumption.

In combination with selections that are executed prior to the join, FDs can be used for optimizing these joins. Again, consider the FD $X \rightarrow A$ to hold on a relation R and a query that filters R with an equality predicate on X and then joins R with a relation S on A. The FD ensures that all tuples in R that remain after the filter operation have the same value in A. Thus, all tuples from S that match the first filtered tuple in R also match all other remaining tuples in R. This insight can substantially reduce the complexity of the join operation: after joining the first filtered tuple of R to its matching tuples in S, we can reuse the same set of S tuples as matching tuples for all other filtered R tuples by simply duplicating these tuples. In other words, $\sigma_X(R) \bowtie_A S$ with $X \rightarrow A$ can be written as $\sigma_X(R) \times (S \ltimes_A \pi_A(\sigma_X(R)))$, which is, we calculate the cross-product of the filtered R tuples with the S tuples filtered by the right-hand-side value A defined by the X value. For performance reasons, the result of $\sigma_X(R)$ can be cached to execute the filter only once. Also note that, under bag semantics, the *limit* operator must be applied to obtain only the first matching R tuple for the semijoin with S.

Finally, Abiteboul, Hull, and Vianu show that an equality self-join can be avoided in cases, where the join attribute functionally determines all other attributes in the distinct projection of the query [AHV95]. Consider a relational instance r over R with attributes X, Y and the FD $X \rightarrow Y$ to hold. Any self-join of r on X is semantically superfluous and can, therefore, be avoided, because $\forall s, t \in r$: iff $s[X, Y]$ and $t[X, Y']$ the aforementioned FD ensures $Y = Y'$. If R contains the attributes X, Y, Z, but $X \not\rightarrow Z$, the self-join on X is still unnecessary if only X and Y are in the projection of the query. In both cases, the self-join adds a redundant column. More specifically, $\pi_{XY}(R) \bowtie_X \pi_{XY}(R)$ with $X \rightarrow Y$ can be rewritten to $\pi_{XYXY}(R)$. Admittedly, the join also (inefficiently) removes `NULL`-valued records if X contains `NULL` values and it increases the cardinality of the relation if X is not distinct, i.e., it is not a UCC.

7.3.3. FDs and Selection

The use of FDs with selections offers powerful simplifications. Some of the below mentioned optimizations might not be efficiently realizable with standard implementations of physical operators but require alternative implementations.

For an FD $X \rightarrow A$ on a relational instance r over R, consider a query that filters R with two *equality* predicates on X and A. If the filter on X is evaluated first, it then suffices to check only a single element of A, because the FD guarantees that all other tuples (that matched X) have the same value in A: the overall result is empty, if the A value differs from its filter value; otherwise, it is non-empty and no further A value needs to be checked. In an extreme case, a tuple-at-a-time execution model can abort the selection process after checking the first tuple of a potentially large table, if its X value matches but its A value differs.

Furthermore, selections can be shifted to another attribute that is less expensive to process, for instance because it is indexed or of a less complex data type, such as *int* instead of *string*. This technique is also called *predicate introduction*[30] and was originally intended to be used with check constraints by Cheng et al. [Che+99]. Kimura et al. [Kim+09] explicitly mention the application of FDs in such scenarios. Given the FD $A \rightarrow B$ for a relational instance r over schema R and a query with the selection $\sigma_{A=v_A}$, the system could determine the first tuple $s \in r$ with $s[A] = v_A$ (for example, with a partial scan) and, then, find the value v_B as $s[B] = v_B$ that corresponds to the

[30]Predicate introduction can be applied not only if FDs are present but also if correlations [Mic08] or algebraic constraints [BH03] exist, which are not considered in this thesis.

selection value v_A. Now, the system replaces $\sigma_{A=v_A}$ with the cheaper selection $\sigma_{B=v_B}$. This new query serves only as a prefilter, because not all tuples that match on $\sigma_{B=v_B}$ also match on $\sigma_{A=v_A}$. So in the end, the (potentially small) result-set of the adapted query must be re-evaluated on $\sigma_{A=v_A}$. This optimization technique requires highly selective B-predicates and relatively large amounts of data to result in performance advantages. Furthermore, v_B must not be `NULL`, because `NULL` comparisons in selections always resolve to *false*. The final evaluation of $\sigma_{A=v_A}$ can be omitted if the reverse FD $B \rightarrow A$ is also true.

Scalar subqueries are required to return either no or exactly one row. Many systems throw an exception otherwise [Ora; SAP19]. Date and Darwen [DD92] mention a possible optimization regarding such queries in combination with FDs. Consider the query

```
SELECT * FROM R WHERE
    R.B > (SELECT B FROM S WHERE S.A = 4).
```

If the FD $A \rightarrow B$ holds, the equality predicate on A in combination with the FD ensures that B has the same value for all rows. Thus, the result could be computed even though the subquery returns more than a single row. However, to the best of our knowledge, this technique is not implemented in any commercial database system.

7.3.4. FDs and Sorting

FDs can be used to simplify operations that introduce order in the involved relations [Che+99; SGG12b]. We can, in particular, reduce the attribute lists in `ORDER BY` clauses with the use of known FDs, which was first shown by Simmen et al. [SSM96]. For example, the clause `ORDER BY X, A` can be reduced to `ORDER BY X` if the FD $X \rightarrow A$ holds, because for a certain value of X, there is only one value in A.

7.3.5. FDs and Cardinality Estimation

Query plan optimizers or other database components that estimate the cardinality of database operators often assume value independence for the different attributes and uniform value distributions[31] [Lei+15; Lei+18; Sel+79]. As the name implies, functional dependencies indicate the contrary. Known FDs can, therefore, improve cardinality estimations [CGR01; Ily+04; SSY98] and lead to better query plans. For instance, the cardinality of the conjunction of two predicates $\sigma_{A=v_A}$ and $\sigma_{B=v_B}$ is usually estimated as the product of their individual selectivities, which is $\frac{1}{|A|} \cdot \frac{1}{|B|}$. If, however, $A \rightarrow B$ is true, then only $\frac{1}{|A|}$ is the appropriate cardinality estimate, because v_A *always* co-occurs with v_B due to the FD [Thed].

[31]Not all systems assume uniformity for all values. For example, PostgreSQL assumes uniform value distributions only for all values *inside each histogram bucket* [Thee].

Furthermore, given $A \rightarrow B$, we know that $|B| \leq |A|$ and, given both $A \rightarrow B$ and $B \rightarrow A$, we know that $|B| = |A|$. In this way, we can use FDs to let cardinality information (or estimations) propagate from one attribute to another.

Finally, Gelenbe et al. utilize functional dependencies to estimate the size of projections (in the duplicate-removing semantics of the relational algebra) [GG82]. They use the above-described guarantee of $|B| \leq |A|$ in the presence of $A \rightarrow B$. Following from that, $|\Pi_{A,B}| = |\Pi_A|$.

7.4. Order Dependencies

Order information serves a variety of tasks, such as optimizing the physical storage of records (e.g., for run length encoding in columnar data stores [AMF06]) and improving readability of query results by ordering them.

Information about order and so-called interesting orders (first introduced by Selinger et al. [Sel+79]) are a crucial part for query rewriting and query plan optimization; they can, in particular, further be utilized during the actual operator execution and for cost estimation. ODs present an important opportunity to make the most use of order-based optimization techniques, because they help to derive additional order information from knowledge about currently available orders. As a result, knowing that $\boldsymbol{X} \overset{\preceq}{\mapsto} \boldsymbol{Y}$ holds and that $\boldsymbol{X}$ is ordered opens up opportunities to utilize the order information about both attributes $\boldsymbol{X}$ and $\boldsymbol{Y}$ during query optimization. Operations that generate ordered data can be explicitly pushed down (closer to the beginning of the query plan, cf. *sort-ahead* [SSM96]) to enable order-based optimizations for the subsequent operators. The more order information is available, the wider is the range of potential plan optimizations.

We compile information on how exactly order and order dependencies can be utilized for query optimization and to improve the execution of individual operators in the remainder of this section. Some optimization ideas presented in this section are similar to ideas presented for FDs, showing interesting relationships between FDs and ODs.

7.4.1. ODs and Sorting

The purpose of sort operations, explicitly expressed by `ORDER BY` statements in SQL, is to produce order. Hence, it is not surprising that sorting offers several potential optimizations regarding ODs. First, the number of attributes in the order clause can be reduced in the presence of interesting orders [SSM96] and, hence, order dependencies [SGG12a; SGG12b; Szl+14]. A reduced number of sorting attributes leads to fewer sort operations, which can decrease the execution time. Additionally, reducing the number of attributes in the order clause increases the possibility that this operation can be solved with an index.

If $\boldsymbol{X} \overset{\preceq}{\mapsto} \boldsymbol{Y}$ holds on a relation R, the clause `ORDER BY X, Y` can be reduced to `ORDER BY X`, because an ordered $\boldsymbol{X}$ ensures an ordered $\boldsymbol{Y}$.[32] With the aforementioned OD, $\boldsymbol{Y}$ can furthermore be removed from both clauses `ORDER BY W, X, Y` and `ORDER BY W, Y, X` [SGG12b]. The latter might not be intuitively clear, but if we replace $\boldsymbol{Y}$ with $\boldsymbol{X}$, which is possible because $\boldsymbol{X}$ imposes an order on $\boldsymbol{Y}$, the resulting `ORDER BY W, X, X` still guarantees that the result is ordered by $\boldsymbol{Y}$ following from the definition of ODs; obviously, one of the consecutive $\boldsymbol{X}$ could be removed. Sorting $\boldsymbol{Y}$ can be avoided altogether under above's OD and if other previously executed operations, such as sort-merge joins, sort-based aggregates, or index scans internally order $\boldsymbol{X}$ (or $\boldsymbol{Y}$) [Gra93; Pau00]. In the case of $\boldsymbol{X} \overset{\prec}{\mapsto} \boldsymbol{Y}$, `ORDER BY Y, X` can be reduced to `ORDER BY Y`, because $\boldsymbol{X}$ cannot break ties in $\boldsymbol{Y}$.

Furthermore, ODs can be utilized to substitute sorting attributes [Pau00]. Imagine the attributes A (integer), B (string), and C (integer) and the OD $A \overset{\preceq}{\mapsto} B$ to hold. Hence, a statement `ORDER BY B, C` could be replaced by `ORDER BY A, C` because an ordered A implies B to be ordered. Substitutions are beneficial if they decrease the cost of the sorting operation: in our example the costly string-sort was replaced by the cheaper integer-sort. Alternatively, the attribute to be substituted could be replaced by an indexed attribute that allows efficient ordered retrieval.

Szlichta et al. [Szl+14] also mention that ODs and near-sortedness can be combined to execute small, on-the-fly main memory sorts instead of external sorts.

7.4.2. ODs and Joins

Order information can be used to simplify joins in the plan optimization phase, to improve the join operator execution phase, and to better estimate join result-set cardinalities.

Sort-merge join algorithms, as the name implies, are split into two phases: (i) an initial sort phase that provides ordered input for the (ii) merge phase, where the actual join takes place by merging two ordered lists. It is evident that sort-merge joins can benefit from already sorted inputs, by omitting the initial sort phase [GHK92; Pau00; SSM96]. If $\boldsymbol{X} \overset{\preceq}{\mapsto} \boldsymbol{Y}$ holds and $\boldsymbol{X}$ is ordered, sort-merge joins on either $\boldsymbol{X}$ or $\boldsymbol{Y}$ could both omit the sort phase on their corresponding relation, and could be a preferred choice.

Another opportunity is the pipelining of grouping and join operators [CS94; Gra93]. In [CS94], the authors present a cost model-based approach to push grouping operators past joins to find more efficient query plans. Some grouping operator implementations sort the data to create groups. If $\boldsymbol{X} \overset{\preceq}{\mapsto} \boldsymbol{Y}$ holds and $\boldsymbol{X}$ constitutes the grouping attributes, this technique is promising if the later join is either on $\boldsymbol{X}$ or on $\boldsymbol{Y}$.

[32] Note, $\boldsymbol{X} \overset{\preceq}{\mapsto} \boldsymbol{Y}$ is insufficient for this optimization, because the comparator $\prec$ imposes only a partial order. Thus, $\boldsymbol{Y}$ would still be needed to impose a clear order on tuples with the same $\boldsymbol{X}$ values.

Szlichta et al. further describe a data warehouse scenario in which ODs can be used for query rewriting to avoid expensive joins between fact and dimension tables [Szl+11; Szl+14]. Consider the SQL query

```
SELECT [...] FROM sales s, date d WHERE
    s.sold_date_sk = d.date_sk AND
    d.date BETWEEN '20210819' AND '20210823'.
```

The insight that there is usually an order dependency in the date dimension table between a well-constructed surrogate key and natural date values, i.e., $\mathsf{date_{sk}} \overset{\preceq}{\mapsto} \mathsf{date}$, enables the rewriting of joins on dates, because date attributes of the fact and dimension table can be replaced with probably cheaper local predicates. The intuition is as follows: two simple probes find the minimum and maximum[33] surrogate keys for the corresponding dates in the dimension table. These keys are used as local predicates on the fact table:

```
SELECT [...] FROM sales s,
    (SELECT MIN(date_sk) min_d FROM date
        WHERE date >= '20210819') d1,
    (SELECT MAX(date_sk) max_d FROM date
        WHERE date <= '20210823') d2
  WHERE s.sold_date_sk BETWEEN d1.min_d
      AND d2.max_d.
```

The combination of dimension table probing and a local fact table predicate effectively replaces the join. This is possible only because the OD $\mathsf{date_{sk}} \overset{\preceq}{\mapsto} \mathsf{date}$ ensures that the correct surrogate keys are picked for the local predicate via `MIN` and `MAX`. This join replacement optimization, and in particular its knowledge during the query optimization phase, permits another valuable downstream optimization: introducing a local predicate on the fact table enables pruning opportunities which often show a significant performance impact in practice [Dre+20].

Order dependencies can also be used to optimize certain theta joins via query rewriting if combined with UCCs. For this, assume that the OD $\boldsymbol{X} \overset{\preceq}{\mapsto} \boldsymbol{Y}$ holds on a relation R and that X is a UCC. Then, consider the example SQL query

```
SELECT * FROM R Rl, R Rr WHERE
    Rl.X < Rr.X
```

which performs a theta self-join on $\boldsymbol{X}$. Due to the OD, we can replace the term `Rl.X < Rr.X` with the OD's right-hand-side attribute $\boldsymbol{Y}$: `Rl.Y < Rr.Y`. The UCC ensures that R does not contain tuples with the same value in X and potentially unordered values in Y. Scenarios with such a combination of UCC and OD occur naturally if attributes $\boldsymbol{X}$ correlate with, e.g., an incremental surrogate key attribute $\boldsymbol{Y}$.

[33]This optimization benefits from an engine supporting the simultaneous computation of `MIN` and `MAX` with a single scan.

This technique improves the execution of the query if $\boldsymbol{Y}$ has, e.g., an index, a smaller domain, or a more join-friendly data type. The same optimization also applies for the other inequality predicates $\leq$, $>$, and $\geq$ as well as for the OD $\boldsymbol{X} \overset{\succeq}{\mapsto} \boldsymbol{Y}$.

7.4.3. ODs and Grouping

Similar to joins, there are two approaches for grouping data and aggregating data, respectively: hashing and sorting [Mül+15]. Both operators profit from preordered inputs. With the knowledge of ODs, additional inputs are known to be ordered. Hence, optimizers can choose sort-based operators in more cases. Sort-based implementations can benefit from preordered inputs, because the sorting step of the operator can — exactly as for joins — be (partially) omitted [Pau00; SSM96; Szl+14; YL94; YL95]. Hash-based implementations, on the other hand, can exploit ordered inputs to minimize the number of hash calculations, i.e., they simply avoid repeated rehashing of the same value by recycling the previous hash until the next new value in the ordered input is read. The positive effect of ordered data on the performance of hash-based algorithms was investigated by Memarzia et al. [MRB19].

7.4.4. Further Optimization Opportunities with ODs

Apart from the main concepts described above, order dependencies offer various further optimizations.

Selection. Selections can benefit from ODs in the following way: if $A \overset{\preceq}{\mapsto} B$ holds and their relation is ordered by A, then table scans on data that is held in-memory can be replaced with binary searches not only for selections on A but also for selections on B [Pau00], which reduces the complexity from $\mathcal{O}(n)$ to $\mathcal{O}(\log n)$. Range predicates can analogously use a binary search to determine the starting element of the range.

Aggregate Functions. Some aggregation functions, such as `MIN`, `MAX`, and `MEDIAN`, allow obvious optimizations on ordered data [Pau00]: if $A \overset{\preceq}{\mapsto} B$ holds and their relation is ordered by A, it is not necessary to check all elements of the relation to determine the result of `MIN` on either A or B. Instead, it is sufficient to return the first element of the respective attribute [ONe94, p. 566]. This shortcut works analogously for `MAX` and `MEDIAN`, but instead of the first, the last and middle element, respectively, are selected.

Leveraging Indexes. Indexes can be used to retrieve tuples in order, which is useful to pipeline index operations with other operators that rely on ordered data. In this way, we can, for instance, pipeline sort-merge joins with index scans [Gra93]. Also, a clustered index on salary along with the ODs salary $\overset{\preceq}{\mapsto}$ taxes and salary $\overset{\preceq}{\mapsto}$ percent allow a query that contains an `ORDER BY taxes, percent` clause to be evaluated directly by retrieving the data from the index, without an additional sort operation [Szl+14]. Another index optimization based on ODs was proposed by Dong et al., who found that with the knowledge of ODs sparse indexes can be chosen over dense ones to save space [DH82].

Generating Distinctness. According to Chaudhuri et al. [CS94], distinctness is a special case of a grouping operation. Hence, optimizations presented in Section 7.4.3, such as the utilization of already sorted input [Pau00; SSM96], apply here as well.

Set Operations. Set operations, unless explicitly specified, eliminate duplicate rows. As stated above, the removal of duplicate entries often relies on sorting the input data, thereby opening up opportunities for order optimizations for set operations [Pau00]: we consider a `UNION` statement whose select list consists only of $\boldsymbol{Y}$ and we assume $\boldsymbol{X} \overset{\preceq}{\mapsto} \boldsymbol{Y}$ to hold while $\boldsymbol{X}$ has been ordered, for example, by an `ORDER BY` clause that has been pushed down. In such a case, the duplicate elimination within the set operation can avoid an additional sort operation, because the tuples are already sorted with respect to $\boldsymbol{Y}$. Similar optimizations are possible for `INTERSECT` and `EXCEPT`.

Cost Estimation. While executing subqueries, a query engine might decide to cache previously computed results of the inner query to reuse them in case the inner query is executed with the same correlation attribute values again. In the case of ordered correlation attributes (compare the optimization of Selinger et al. [Sel+79] mentioned in Section 7.2.4), e.g., by order dependencies, a cache size of 1 is sufficient [Pau00], because the order guarantees that the subquery will never be executed with the same inputs again once new inputs are read from the ordered correlation attributes. Again, ODs extend the applicability of such order-based optimizations to further attributes.

Also, cost predictions, e.g., for operator costing, can take ODs into account to more accurately estimate execution costs of operators that rely on ordered data.

7.5. Inclusion Dependencies

Inclusion dependencies often serve data linkage and integration scenarios as they may span multiple data sources. This section presents how they can be used for query optimization.

7.5.1. INDs and Joins

By their definition, INDs are most relevant for join operations. An often-used join variant are semijoins $R \ltimes_{\boldsymbol{X}} S$, which are used to filter R by the tuples that have a matching tuple in S. Semijoins can be expressed as joins with a post-projection on the attributes of the R side, which is $\pi_R(R \bowtie_{\boldsymbol{X}} S)$. Given the IND $R.\boldsymbol{X} \subseteq S.\boldsymbol{X}$ with its formal definition $\forall t_r \in R, \exists t_s \in S : t_r[\boldsymbol{X}] = t_s[\boldsymbol{X}]$, all tuples from R match tuples in S so that the semijoin is redundant; hence, it can be removed. Note that this optimization requires all $R.\boldsymbol{X}$

to be `NOT NULL` or the IND to follow the `NULL` $\neq$ `NULL` semantics. This is because the semijoin filters out all tuples with `NULL` values in $R.\boldsymbol{X}$; the IND $R.\boldsymbol{X} \subseteq S.\boldsymbol{X}$ with `NULL` $=$ `NULL` semantics, on the contrary, might find a matching tuple in S with the same `NULL` values as the R tuple.

A popular, yet dependency-independent optimization technique is *semijoin reduction* [Ber+81; Sto+01], which reduces the number of tuples considered by join operators before the actual join is conducted. For this reduction, the matching (non-dangling) tuples are determined a priori by utilizing semijoins. The semijoin reduction $(R \ltimes \pi_{\boldsymbol{X}}(S)) \bowtie_{\boldsymbol{X}} S$ is equivalent to the plain join operation $R \bowtie_{\boldsymbol{X}} S$. The rationale is that identifying the matching tuples in R and joining only these with S is more efficient than joining R and S directly. This semijoin technique is particularly useful for distributed database setups that need to minimize expensive data transfers. Dreseler et al. [Dre+19] have shown this optimization to be beneficial for main memory database systems, too. However, if *all* tuples from R match tuples in S, which is the case under an IND, semijoin reductions are unnecessary but still costly as they effectively perform the join twice. So if the IND $R.\boldsymbol{X} \subseteq S.\boldsymbol{X}$ is known, semijoin reductions on $R.\boldsymbol{X}$ can be avoided. Because the semijoins are used only as prefilters in this optimization, it is possible to use INDs based on `NULL` $=$ `NULL` and `NULL` $\neq$ `NULL` semantics in this optimization: by using `NULL` $=$ `NULL` INDs, we might eliminate semijoins that would filter some records with `NULL`s in the join attributes, but (i) the following join filters them anyway so that the result remains correct, (ii) removing such semijoins might not even impact the performance negatively depending on the number of actually filterable `NULL` records, because the filtering costs might outweigh the data transfer overhead, and (iii) adding a default `NULL` check to all join tuples before sending them would easily solve the issue.

Johnson and Klug [JK84b] as well as Cheng et al. [Che+99] discuss and evaluate techniques to eliminate joins in the presence of INDs and foreign-key constraints, respectively. If a foreign key constraint holds, it allows to eliminate certain joins whose result is known without executing them: given the IND $R.\boldsymbol{X} \subseteq S.\boldsymbol{X}$ and a UCC on $S.\boldsymbol{X}$ on the relations R and S, then the join $R \bowtie_{\boldsymbol{X}} S$ can be avoided if there are no further selections or projections on any attributes of S. This is, because the IND guarantees that every tuple of R joins with a tuple in S, and the UCC ensures the absence of multiple matching tuples. The described optimization, again, requires that either $R.\boldsymbol{X}$ is free of `NULL` values or the IND is valid under `NULL` $\neq$ `NULL` semantics.

The idea of join elimination can be extended to intermediate joins: if there are two joins $R \bowtie_{\boldsymbol{X}} S$ and $S \bowtie_{\boldsymbol{X}} T$ with the INDs $R.\boldsymbol{X} \subseteq S.\boldsymbol{X}$ and $S.\boldsymbol{X} \subseteq T.\boldsymbol{X}$ and a UCC on $S.\boldsymbol{X}$, the two joins can be reduced to a single join $R \bowtie_{\boldsymbol{X}} T$ following from the transitivity of INDs [CFP82]. The UCC on $S.\boldsymbol{X}$ is necessary because duplicate values in $S.\boldsymbol{X}$ would cause record duplication that would be ignored in the optimized case. The removal of S

also requires that the SQL query neither contains attributes from S in its final projection nor that it filters S. The authors state that transitive join queries are suboptimal and could be avoided in the first place. Nevertheless, such transitive joins are often seen in practice, because applications and object-relational mappers automatically generate them. In other cases, database users might only have access to specific views that contain or require such unnecessary joins. For this reason, query optimizers can benefit from these IND-based optimization techniques.

Deutsch et al. show that in certain cases a join $R \bowtie_{\boldsymbol{X}} S$ can be replaced with a join $R \bowtie_{\boldsymbol{X}} T$ [DPT06]. This replacement can be beneficial for peer database systems, in which T might offer faster access or higher availability due to being located at a different site than S. This rewrite requires the INDs $T.\boldsymbol{X} \subseteq S.\boldsymbol{X}$ and $(R \bowtie_{\boldsymbol{X}} S).\boldsymbol{X} \subseteq T.\boldsymbol{X}$ to hold. Because the second IND depends on a query result, a practical implementation would need to check the stricter, but discoverable IND $S.\boldsymbol{X} \subseteq T.\boldsymbol{X}$.

7.5.2. Further Optimization Opportunities with INDs

Although IND-based query optimizations seem to focus primarily on join operations, various further optimization opportunities exist.

Query Folding. Query folding is an optimization technique to answer queries by rewriting them in a way that lets them use certain resources, e.g., cached query results or materialized views instead of base tables. Jarek Gryz demonstrated in [Gry98] how INDs enable further scenarios for the application of query folding with views.

Assume a query accessing the attribute combination $\boldsymbol{X}$ of a relation S. If no materialized view contains $S.\boldsymbol{X}$, this query does not have a query folding and cannot be answered using views. However, assuming that there is an IND $R.\boldsymbol{X} \subseteq S.\boldsymbol{X}$ and a materialized view on $R.\boldsymbol{X}$, the query could be rewritten to access $R.\boldsymbol{X}$ instead of $S.\boldsymbol{X}$ so that the answer to the query can be retrieved via query folding from $R.\boldsymbol{X}$'s materialized view. Clearly, the rewrite may return only a subset of the initial query's actual result, but this might be acceptable for certain use cases where users are not interested in the entire result set or need their answers quickly [Gry98].

Deutsch et al. [DPT06] as well as Ileana et al. [Ile+14] explain how INDs (the original work uses more general tuple-generating dependencies) enable rewritings that utilize materialized views or result caches if set semantics[34] are assumed. For example, a join $S \bowtie_{\boldsymbol{X}} T$ can be replaced by accessing a view V_{ST} if the INDs $S \bowtie_{\boldsymbol{X}} T \subseteq V_{ST}.\boldsymbol{X}$ and $V_{ST}.\boldsymbol{X} \subseteq S \bowtie_{\boldsymbol{X}} T$ hold.

[34]Set semantics occur in SQL-based DBMSs in the presence of keys, set operations, or the `DISTINCT` keyword.

Exists. Correlated subqueries as part of EXISTS statements can also be simplified with INDs. Given the IND $R.A \subseteq S.A$, the query

```
SELECT * FROM R WHERE EXISTS
    (SELECT * FROM S WHERE S.A = R.A)
```

can be computed without accessing S, because the IND ensures that the subquery returns TRUE for every value of $R.A$. For known foreign-key constraints, such EXISTS optimizations have been adopted by productive query optimizers, e.g., by Microsoft SQL Server [Mic04]; whether or not they also use INDs is not known. Note that we require NULL $\neq$ NULL semantics *or* an additional NULL check here because joins and select statements do not match NULL values.

Set Operations. Inclusion dependencies can further be used to simplify the computation of set operations. A query of the form

```
SELECT A FROM R
    UNION
SELECT A FROM S
```

could be rewritten to SELECT DISTINCT A FROM S if the IND $R.A \subseteq S.A$ holds. By the set-based definition of the UNION operation, the result contains all distinct values from $R.A$ and $S.A$ while the IND guarantees that all values from $R.A$ are already included in $S.A$. Here again, the IND must be true under NULL $\neq$ NULL semantics (or $R.A$ must not contain NULL values), because the rewrite would otherwise miss the NULL values from $R.A$ that the UNION would have added. Analogously to the UNION case, this optimization can be applied to INTERSECT operations, but instead of returning the values of $S.A$, the distinct values of $R.A$ need to be returned. For this, the optimization requires an IND with NULL $=$ NULL semantics (or $R.A$ without NULL values), because the INTERSECT removes NULL values from $R.A$ if these are also present in $S.A$. Note that in cases of INTERSECT ALL, the query optimizer can omit even the DISTINCT operation. Besides UNION and INTERSECT also difference operations, such as

```
SELECT A FROM R
    EXCEPT
SELECT A FROM S
```

can be simplified. In the EXCEPT case, the IND guarantees that only an empty result-set can be returned, allowing the query optimizer to skip the execution of the query altogether and return an empty set. Note that EXCEPT removals require INDs with NULL $=$ NULL semantics, because the EXCEPT removes records with NULL values in $R.A$ only if they are also present in $S.A$.

Cardinality Estimation. Similar to some of the dependencies discussed before, INDs also allow for more accurate cardinality estimations. Consider, for example, an IND

$R.\boldsymbol{X} \subseteq S.\boldsymbol{X}$. An equality join on the attribute combination $\boldsymbol{X}$ of these two relations returns a minimum of $|R.\boldsymbol{X}|$ results, because the IND guarantees for every tuple in R at least one matching tuple in S. Again, we need the `NULL` $\neq$ `NULL` semantics for this optimization, because it effectively ensures that the IND is valid only if there are no `NULL` values in $R.\boldsymbol{X}$. In case $S.\boldsymbol{X}$ is a UCC on $S.\boldsymbol{X}$, the IND basically appears as a foreign-key constraint and returns *exactly* $|R.\boldsymbol{X}|$ results. For *theta* joins with predicate $\neq$, i.e., $R \bowtie_{R.X \neq S.X} S$, the number of results is exactly $|R.\boldsymbol{X}| \times (|S.\boldsymbol{X}| - 1)$ records given the IND $R.\boldsymbol{X} \subseteq S.\boldsymbol{X}$ and the UCC $S.\boldsymbol{X}$. Considering implementations of such cardinality estimation strategies in real products, we found only one example (although further examples might exist): IBM's Db2 [IBM22b] database management system uses foreign key constraints to make cardinality estimations more efficient by reducing the number of considered statistical views.

7.6. Additional Optimizations

In this section, we discuss opportunities for further optimizations with semantic integrity constraints and other types of dependencies.

7.6.1. Semantic Query Optimization

The field of *semantic query optimization* [CGM90; HZ80; Kin80; YS89] offers more techniques that utilize further constraints and dependencies for query optimization. For instance, semantic integrity constraints [Sto75] are typically user-defined and encode knowledge about attributes of a relation. For example, a German citizen relation might follow the semantic integrity constraint $\textit{citizen.city} = \textit{Berlin} \rightarrow \textit{citizen.zipcode} \in [10115, 14199]$. While such constraints can also be used for efficient query processing, they differ from data dependencies that do not necessarily carry any semantic meaning.

7.6.2. Further Dependency Types

A substantial body of work discusses how the combination of the *Chase* and *Backchase* procedures can be used to find minimal, equivalent plans for a particular query [DNR08; Mei14]. These procedures can reveal opportunities to use certain auxiliary structures, such as materialized views or indexes, for answering a query [Pop+00]. The dependencies that are often used in the aforementioned work are so-called equality-generating dependencies (egds) and tuple-generating dependencies (tgds) [BV84]. Both egds and tgds can be seen as generalizations of other dependencies [DPT06], including the dependencies covered in this thesis. As such, we already showed their usefulness when presenting optimizations based on INDs above. There are, however, no general automatic discovery algorithms for tgds and egds (yet). What is more, they are usually present in integration scenarios where they span multiple schemata as derived from user-defined schema mappings [FKP05]. Moreover, prior work often assumes set semantics [DPT99; Ile+14], so it is not always suitable for typical relational database scenarios in which bag semantics prevail.

Although the data dependencies examined in this thesis, namely UCCs, FDs, ODs, and INDs, are arguably the most important dependencies for query optimization today, many further types of discoverable data dependencies and data rules exist, such as Multi-Valued Dependencies (MVDs) [Fag77], Neighborhood Dependencies (NDs) [BW01], Sequential Dependencies (SDs) [Gol+09], Denial Constraints (DCs) [Ber11], and others. If an SQL query tests for such a dependency, this test can be avoided if the dependency is already known to be true. Apart from this general rule, we find hardly any research on the use of these dependencies for query optimization and, thus, do not cover them in this collection of dependency-driven query optimization techniques.

7.7. Summary and Open Challenges

In this chapter, we surveyed various core techniques with which database management systems can use genuine and observed data dependencies, namely uniqueness, functional, order, and inclusion dependencies, to improve their query optimization capabilities. The surveyed optimizations are of increasing relevance, because recent advances in the field of automatic data profiling algorithms now enable the efficient discovery of such metadata for large and, hence, practically relevant datasets — database systems today have access not only to user-defined dependencies and constraints but also to much larger corpora of automatically discovered dependencies. Additionally, due to increasing data volumes and constantly high performance demands, database management systems (DBMSs) cannot ignore this data-inherent performance optimization potential.

The compilation of techniques presented in this thesis can serve as a starting point to equip query optimizers with interesting and potentially powerful optimization capabilities that rely on data dependencies. Furthermore, it supports database engineers of established systems in identifying additional techniques to further increase the efficiency of their already advanced execution engines or query optimizers. Finally, we also provide researchers with an exhaustive collection of related work that facilitates the uncovering of open research questions and opportunities for future work. Next, we summarize three major research directions that will also be partially examined in the next chapter.

Efficient Implementation and Integration. While some of the mentioned optimization techniques require relatively simple implementations or are already implemented in some of today's database systems, other advanced techniques will be much more complex to incorporate in practice and might require broad modifications of existing systems. Even though many of the surveyed techniques have been evaluated in the respective research papers, transferring them to practice in commercial database systems is not an easy undertaking and, therefore, a challenge for future work.

Incremental Discovery and Maintenance. The knowledge of available data dependencies is essential for all dependency-driven optimization techniques. Although we did reference incremental discovery and maintenance approaches for all of the four considered dependency types, more efficient approaches that can handle large dynamic datasets under real-world workloads are necessary to enable the surveyed query optimization techniques in practice.

Empirical Impact Evaluation. While this thesis collects many optimization techniques, their general effectiveness, relevance, and overhead are still to be evaluated. A systematic study is needed to measure the effectiveness and quantify the impact of the various optimizations on query performance. Such impacts are neither obvious nor straightforward to determine, as they depend on diverse factors, such as the database system's implementation, the specific query at hand, and the underlying dataset. To judge the relevance of each optimization, an empirical study is needed that evaluates how often and, hence, how likely the presented techniques apply to queries in real-world workloads. This study is a challenging task as it requires a representative collection of query workloads. Finally, another study to quantify the optimization overhead introduced by the proposed optimization techniques would require specific, well-tuned implementations to measure potentially elevated optimizer runtimes and assess code complexity.

8 Integration and Evaluation of Data Dependency-Driven Query Optimization

This chapter presents our approach to how the previously presented data dependency-based query optimization techniques can be applied in a database management system (DBMS). Thereby, we contribute to solving the challenges mentioned in Section 7.7 regarding an efficient integration of such techniques, their performance impact, and the challenging maintenance of data dependencies in the context of query optimization. First, we discuss why most of a dataset's dependencies are typically unknown to DBMSs (Section 8.1). Afterward, we propose an automatic, workload-driven approach that discovers dependencies and enables the unsupervised utilization of data dependencies for query optimization in Section 8.2. The approach is implemented and evaluated with our research DBMS Hyrise, offering speed-ups of up to 61 × for certain TPC-DS queries and reducing the Join Order Benchmark (JOB)'s execution time by 26 % as presented in Section 8.3. Finally, we discuss related work (Section 8.4), before concluding this chapter and presenting ideas for future work (Section 8.5).

Parts of this chapter have been published in a research paper [Kos+22a], which was a collaborative effort. The thesis author developed the underlying concept, prepared the majority of the original draft for publication, created a prototypical implementation, and designed the conducted experiments. Lindner refined and extended the implementation and executed the experiments. Lindner, Naumann, and Papenbrock contributed to the paper's concept and improved the material and its presentation.

8.1. Challenges

Commonly, many data dependencies [SP22] of different types exist for any given dataset, due to natural correlations in the data, schema denormalization, and certain data generation patterns, cf. Section 6.2. In Chapter 7, we have discussed dozens of data dependency-based query optimization techniques. Despite the existence of a variety of such techniques, data dependencies and their query optimization strategies remain largely underutilized because the database systems are not aware of these dependencies. The reasons for this unawareness are three challenges, namely dependency *discovery*, *selection*, and *mutation*, that are still practically unsolved in the context of query optimization for DBMSs. The following subsections discuss these challenges in detail.

8.1.1. Dependency Discovery

The *discovery* challenge describes the fact that database systems usually do not have knowledge about available data dependencies and that their discovery is expensive. Considering the existence of thousands or even millions of dependencies on many datasets [SP22], it becomes apparent that dependencies derived from manually or schema-defined constraints, such as primary or foreign keys, are only responsible for a fraction of all existing dependencies. In some use cases, no manually defined constraints and keys exist. Many scientific datasets, for example, are provided as CSV files that may include column headers but no additional metadata [Abe+18]. Furthermore, some dependencies, such as order dependencies, cannot even be manually defined as constraints in standard SQL and established DBMSs. Automatically determining data dependencies via data profiling algorithms is feasible but expensive and can take hours of processing [Abe+18; SP22]. For example, for a real-world ERP system's table of 45 M rows, one of the most efficient parallelizable functional dependency (FD) profiling algorithms runs 8 hours to completion with 32 parallel threads [PN16]. Even if the dependency mining is run as an asynchronous background task, this runtime is extensive and might outweigh potential dependency-based advantages.

8.1.2. Dependency Selection

The systematically profiled dependencies lead to the second challenge, which is dependency *selection*. Data profiling algorithms discover *all* technically valid dependencies on a given dataset. Therefore, the result sets can become large enough that even storing and efficiently accessing them might be unfeasible during query optimization. For instance, a 1 M rows sample of the NCVoter statewide dataset[35] contains a remarkable amount of 5 M minimal FDs (and many further dependencies of other types) [PN16]. Suppose the overhead for storing all discovered dependencies would be acceptable. In that case, it may still not be possible to find a particular dependency during query optimization fast enough because their retrieval involves not only dependency lookup but also inference: discovered dependencies are usually minimal, but the dependencies required for query

[35]NCVoter dataset: `https://www.ncsbe.gov/results-data/voter-registration-data`

optimization may not be. A strategy to select only dependencies relevant for query optimization from discovered dependency sets does not exist yet.

8.1.3. Dependency Mutation

Even if dependencies could be efficiently discovered and selected for query optimization, a third challenge arises. Dependency *mutation* is still an open issue because every `INSERT`, `UPDATE`, or `DELETE` statement could invalidate expensively mined dependencies. Such an invalidation would make them unsafe for query optimization: numerous optimizations presented in Chapter 7 might produce query plans leading to wrong query results. Also, new and helpful dependencies might appear, for example, caused by the deletion of tuples that violate a dependency's preconditions. In addition, efficiency is another crucial requirement when handling dependency mutation as the efforts for keeping data dependencies up-to-date must not outweigh the realized advantages. Thus, dependency discovery, selection, and mutation need to be focused on improving query processing performance.

8.2. Workload-Driven, Lazy Discovery, Selection, and Mutation of Data Dependencies

We start this section with a short excursus that provides the necessary background of the research DBMS Hyrise. Afterward, we first explain our workload-driven data dependency discovery system. This system determines and validates data dependency candidates for specific query optimization techniques considering only such dependencies relevant to the observed workload. We then explain how changes to the data, which potentially mutate or invalidate dependencies, can be handled by various techniques. While we implemented our data dependency-based query optimization system in Hyrise [Dre+19], the presented concepts are based on generic database concepts and are not technically coupled to this particular DBMS; hence, they are applicable to other DBMSs, too.

8.2.1. Background: Hyrise

To demonstrate our dependency-based query optimization system, we implemented it into the research DBMS Hyrise[36]. In the following, we briefly introduce the architecture and relevant components of Hyrise [Dre+19] since some of its concepts are beneficial to our approach, as we will discuss later in Section 8.2.3. We refer the interested reader to the original paper [Dre+19] or Markus Dreseler's thesis [Dre21] for further details.

[36]Hyrise source code and documentation on GitHub: `https://github.com/hyrise/hyrise`

Architecture. Hyrise is a main memory, column-oriented database system with an implicitly horizontally partitioned storage layout. The partitions are called *chunks* and have a fixed maximum size (default: 65 535 tuples). The fraction of a column that is contained in a chunk is called *segment*. Chunking serves two primary purposes: increased opportunities for data access avoidance by pruning and fine-granular configuration and optimization decisions. For instance, decisions upon which encoding to apply or indexes to create can be taken on segment level and do not have to be made for a complete column. Thereby, more precise decisions can be made, and the overhead of applying new configurations is reduced.

The standard configuration of Hyrise applies dictionary encoding [Lan+16] to all chunks. Furthermore, Hyrise follows an insert-only approach utilizing multiversion concurrency control (MVCC) [BG83; Sch+14]. With MVCC, deletes and updates do not physically modify the original rows. Instead, rows are marked invisible and new versions are appended to the table's last chunk.

SQL Pipeline. SQL queries are handled by Hyrise's SQL pipeline, depicted in Figure 8.1. The SQL pipeline's components transform a declarative SQL query into an imperative query plan to produce a query result table. The pipeline comprises five steps.

(i) The *SQL parser* is the pipeline's first component and transforms a string-based SQL query to an abstract syntax tree (AST) representation consisting of C++ structs. While all of the following components are part of Hyrise's DBMS core, the SQL parser[37] is also available as a standalone component.

(ii) The *SQL translator* creates a *logical query plan (LQP)* based on the *statements*. The LQP is a directed acyclic graph whose nodes are logical operators. These logical operators resemble the operators of the relational algebra.

(iii) Afterward, the *optimizer* applies a set of rules to the LQP to create an *optimized LQP* that is semantically equivalent but potentially more efficient. For instance, the `DependentGroupByReductionRule` can reduce the number of attributes in `GROUP BY` clauses; this optimization was also described in Section 7.3.1.

(iv) Subsequently, the *LQP translator* creates a *physical query plan (PQP)* that consists of *physical operators*, i.e., the actual executable implementations of database operators. For example, a logical `JoinNode` can be translated to a physical `JoinHash`, `JoinSortMerge`, `JoinNestedLoop`, or `JoinIndex` operator. Such choices are guided by the optimizer or hardcoded in the LQP translator.

(v) These physical operators are executed on the data and produce a query result.

[37] Hyrise's standalone SQL Parser on GitHub: `https://github.com/hyrise/sql-parser`

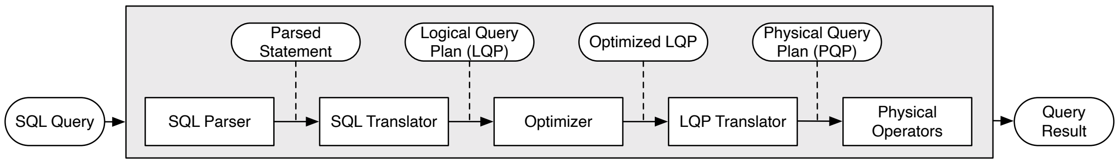


Figure 8.1.: SQL Pipeline of Hyrise: from SQL query to query result.

In addition, Hyrise caches LQPs and PQPs by default. Thereby, the unnecessary repetition of the SQL pipeline's steps is avoided if known queries are executed. The operators' performance data is stored along with the PQPs in the cache. Such data contains, e.g., the operators' execution time and the number of output rows.

Plugins. Hyrise offers a so-called *plugin interface* that enables the implementation of unsupervised components with access to internal resources. Plugins are implemented as dynamic libraries in C++. Thereby, plugins can be (un)loaded at any time and guarantee native execution speed. Plugins can read and write internal data structures without being tightly coupled to the database's core or requiring specific interfaces (cf. [Dre+19, p. 321]), which simplified the implementation of our approach.

8.2.2. Data Dependency Discovery and Selection

Data dependencies are usually unknown. Determining all dependencies that exist on a table or within a dataset without limiting the search space can take minutes or hours for a few million rows [PN16]. The overall target of our approach is to efficiently provide data dependencies that can be used for query optimization. For this reason, our approach focuses only on specifically promising data dependency candidates instead of considering all possible attribute combinations, which is typical for state-of-the-art data profiling algorithms [PN16; SP22].

For this purpose, our approach consists of two phases: first, we determine which data dependency candidates are *relevant*, i.e., which dependencies could be beneficial for processing the system's workload when applied during query optimization. Second, the previously determined candidates are validated. During the second phase, the candidates are ranked according to the potential benefit that their application in query optimization can achieve. In this way, the most promising candidates are promoted to be validated first. This workload-driven, lazy approach explicitly addresses the discovery and selection challenges explained above.

The procedure that determines and validates relevant dependency candidates is depicted in Figure 8.2. The figure contains two main components: (i) A database system (Hyrise in our case) capable of applying dependency-based optimizations and (ii) the *dependency discovery plugin.*

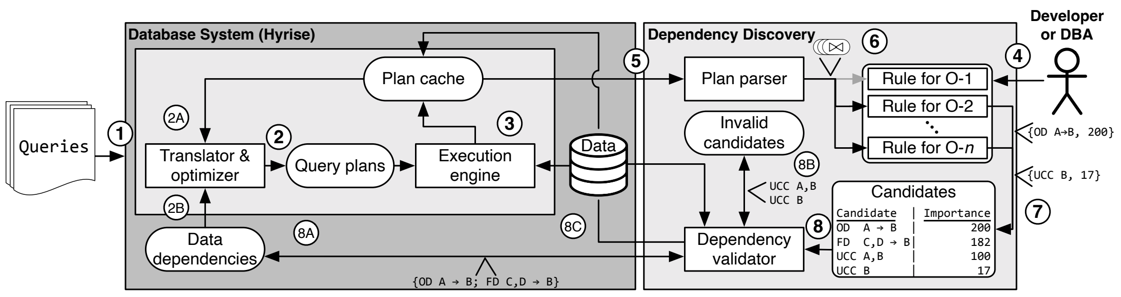


Figure 8.2.: Schematic overview of our dependency discovery and selection procedure and its database system interaction. < indicates that the corresponding example value is passed between the adjacent entities.

Database System. ① Queries are passed as SQL strings to the database system for processing. ② The SQL strings are then translated to logical query plans, optimized, and finally translated to physical query plans. ②A If a query was already processed in the past, its plan might be retrievable from the plan cache to avoid unnecessary retranslation and reoptimization. ②B Also, during optimization, dependencies might be retrieved from the *data dependency store* (simply depicted as *Data dependencies* in Figure 8.2 for brevity) to create more efficient query plans — this extension enables the application of data dependency-based optimizations, such as the optimizations discussed in Chapter 7. ③ Afterward, the query plan gets executed and is stored with runtime information in the query plan cache.

Dependency Discovery. The actual dependency discovery extension is implemented in C++ as a Hyrise plugin[38] to prevent an unnecessary coupling to the database core. The discovery process is executed periodically with a configurable frequency. ④ The basis for this procedure is a set of developer-defined *rules* that provide the logic to derive specific dependency candidates from queries stored in the plan cache, in particular from executed physical and their corresponding logical operators.

We decided on this rule-based approach because it provides a flexible and straightforward mechanism to extend the system's capabilities to generate dependency candidates for new optimizations. The rules must inherit from `AbstractDependencyCandidateRule` and subscribe to certain operator types, e.g., scans or joins. Then, the rules define under what circumstances which dependency candidates are created via an `apply` method. A simplified example rule that subscribes to *joins* (Line 1) is shown in Listing 8.1. An example for the candidate generation logic, as partly contained in Listing 8.1, could be the following: *if two relations are joined, and if the join mode is either* `Semi` *or* `Inner`*, and*

[38]Source code on GitHub: `https://github.com/Bensk1/phd_thesis/releases/tag/source_code`

```
JoinToPredicateRule() : AbstractDependencyCandidateRule(LQPNodeType::Join) {}

vector<DependencyCandidate> apply(
    const shared_ptr<const AbstractLQPNode>& lqp_node) const {
  const auto join_node = static_pointer_cast<const JoinNode>(lqp_node);

  // Consider only semi and inner joins
  if (!(join_node->join_mode == JoinMode::Semi ||
        join_node->join_mode == JoinMode::Inner)) return {};

  // Disregard multi-predicate joins
  const auto& predicates = join_node->join_predicates();
  if (predicates.size() > 1) return {};

  // Check further requirements and calculate importance
  ...
  candidates.emplace_back(TableColumnIDs{filter_column},
                          TableColumnIDs{join_column}, DependencyType::
                          Order, importance_score);
}
// The rule is registered as follows in the Plugin's constructor
add_rule(make_unique<JoinToPredicateRule>());
```

Listing 8.1: Simplified example of a dependency candidate generation rule.

if this relation is filtered with a **`BETWEEN`** *predicate (not shown in Listing 8.1), then create an order dependency (OD) candidate.* In other words, the rules define which dependencies would need to exist so that certain optimizations (O-1, O-2, O-n in Figure 8.2) could become applicable for the corresponding operator.

⑤ The dependency discovery procedure accesses the database system's plan cache; the stored query plans represent the processed workload. The *plan parser* handles the cache's entries, i.e., the physical and logical query plans, in an operator-by-operator fashion. ⑥ The operators are passed to all *rules* that subscribed to the particular operator type (e.g., O-2 and O-n in Figure 8.2). ⑦ Afterward, the rules, which are active components in this architecture, are executed to check the developer-defined requirements and return applicable, operator-specific dependency *candidates* together with an *importance score* — an integer value — for every candidate (e.g., OD and unique column combination (UCC) candidates in Figure 8.2). The importance score expresses how promising a candidate is, i.e., how much workload performance improvement can be expected from utilizing it during query optimization. The most effective dependencies are identified and validated first by ranking the candidates according to their importance scores.

The importance score is determined by the rule that identifies the candidate. The currently implemented rules consider the execution time of the processed operator for the importance score. More sophisticated cost models could be applied in future extensions, e.g., following a *what-if* approach known from physical database design (cf. Section 2.1) where a particular configuration (data dependency in our case) is only simulated or assumed to exist to retrieve cost estimates for a hypothetical scenario from a cost model.

⑧ Next, the dependency candidates are ordered by their score and validated against the underlying data by the *dependency validator*. In our case, where the procedure is implemented into Hyrise, the validation profits from memory-resident data, its columnar storage layout, and dictionary encoding; Section 8.2.3 provides more details. For some dependency types, the dependency validator partly resorts to Hyrise's efficient operator implementations, for instance, the *Sort* operator for validating order dependencies. We also use techniques of existing validation algorithms, such as sampling [PN16], to eliminate candidates quickly. In a more generic setting, traditional SQL- or position list index (PLI)-based validation algorithms could be used [Abe+18]. (8A) After validation, the identified data dependencies are stored in the DBMS' data dependency store to be used during query optimization. (8B) In addition, the plugin also keeps a list of unsuccessfully validated dependency candidates for efficiency reasons. Both stores are accessed during validation to avoid repeated unnecessary validations on unchanged underlying data. (8C) Furthermore, the corresponding query plans are removed from the plan cache to enable reoptimization, and, thereby, the use of the validated dependencies for optimization.

8.2.3. Efficient Data Dependency Mutation

Data changes can invalidate dependencies, e.g., by introducing duplicate values that invalidate UCCs. Then, the use of invalid dependencies for query optimization can lead to poor cost and cardinality estimations, inefficient plans, and, in the worst case, faulty results. While such mistakes might be acceptable in certain cases, such as approximate query processing [NK04], they are usually not tolerable.

In this section, we go over three techniques that tackle the challenge of dependency mutation as explained in Section 8.1.3 and keep dependencies up-to-date: (i) workload-driven discovery, (ii) incremental validation and maintenance, and (iii) the utilization of column store DBMS concepts. In the following, we discuss each technique in more detail.

Genuine Dependency Candidates Based on Workload-Driven Discovery. A general observation about data dependencies is that *spurious* dependencies may change often, while *genuine*, i.e., semantically meaningful dependencies that model a real-world constraint, do not or only rarely change over the lifetime of a dataset [Abe+18]. Because our system is designed to draw the dependencies from executed SQL queries that operate on real-world datasets, a large portion of the discovered dependencies is expected to be genuine. Hence, the workload-driven discovery works as a semantic prefilter, which can significantly reduce the amount of change in the dependency store compared to other mined dependency collections. Adding to this observation, our evaluations (presented below) have also revealed that many beneficial dependencies exist on dimension tables, e.g., date or type tables, which are very rarely updated [KR13, p. 141]. If updates are actually required, the cost of validating dependencies is low due to the comparably small size of dimension tables [KR13], as we will observe later in Section 8.3.4.

Incremental Validation and Maintenance. If our dependency-based query optimization system is applied in data warehouse scenarios, where the data is updated in specific, low-frequent cycles [Son18], keeping the dependencies up-to-date could be integrated into these cycles. Between updates, the dependencies would be guaranteed to be up-to-date.

For more general setups, we focus on more fine-grained updates: given a concrete data change, it is not necessary to re-evaluate *all* previously validated dependencies and to re-evaluate dependencies *entirely*. Instead, we propose the use of efficient incremental validation and maintenance approaches that exist for all four previously surveyed data dependency types (UCCs, FDs, ODs, INDs) as presented in Section 6.2. Note that many DBMSs, e.g., Microsoft SQL Server [Mic21b], SAP HANA [21], and PostgreSQL [Thea], provide index-based techniques originally designed for efficiently enforcing uniqueness constraints that could be adopted for UCC maintenance. Because our system maintains only a tiny subset of all existing dependencies and does not progressively replace invalidated dependencies with new minimal dependencies (as some of the aforementioned incremental discovery approaches do), the dependency maintenance is more efficient.

Utilization of Column Store DBMS Concepts. Apart from the incremental validation techniques, certain column store concepts are beneficial for handling data change in terms of efficient dependency validation and the treatment of potentially invalid dependencies. Modern column stores, such as DuckDB [RM19], HyPer [KN11], or Hyrise [Dre+19], often store values in *chunks* or *data blocks*, which are implicit horizontal partitions of a fixed size that are compressed and made *immutable* when their capacity is reached [Dre+19; Lan+16]. Also, `UPDATE` statements are *append-only* operations that invalidate the original and insert a new tuple containing the updated values.

This architecture can be exploited in three ways: first, some dependency validation algorithms benefit directly from the columnar storage layout, similar to some database operations. Because data dependencies express relationships *between* attributes, i.e., columns, typically, only very few columns need to be read *sequentially* for their validation. In contrast to row-oriented systems, where entire tuples — including unwanted attributes — are read, columnar storage layouts allow accessing the relevant data precisely for sequential accesses [Pla09]. Therefore, column store systems, such as Hyrise, serve to (re)validate the dependencies more efficiently.

The second advantage of such systems is that they often rely on compression techniques, such as dictionary encoding, which can further improve the validation. For example, our implementation uses Hyrise's chunk-wise dictionaries to quickly determine non-unique column combinations during the validation of UCC candidates.

Finally, due to the combination of the chunk-based storage layout and append-only approach, data changes will not invalidate data dependencies on the immutable chunks, which contain the majority of the stored data [Dre+19]. Instead, they *might* only be invalid on the chunk where data is appended. On this chunk, dependencies should be, at least temporarily, considered invalid until this chunk is revalidated. Thereby, even though a dependency might not be maintained upon insert or update, it will still be valid on most chunks, which can be utilized for query optimization. To demonstrate how this observation can be utilized, we consider the UCC-based join-to-semijoin optimization presented in Section 7.2.1 and an insert or update that might potentially invalidate a necessary UCC. An example is depicted in Figure 8.3. During query optimization, two differing branches (originating from *Table 1*) are introduced in the query plan. For the first branch that targets the subset of chunks where the UCC is valid (Chunk #1 and #2 of *Table 2*), dependency-based optimizations, e.g., the more efficient semijoin strategy, can be considered; for the second branch, such optimizations cannot be utilized. Finally, both branches are then combined with a `UNION` operation. Note, for efficiency reasons, our implementation introduces an additional data-induced [OKC19] filter predicate on the second branch to try to reduce *Table 1* before the *InnerJoin*. This predicate filters the join column by the minimum and maximum[39] value of *Table 2's* Chunk #3.

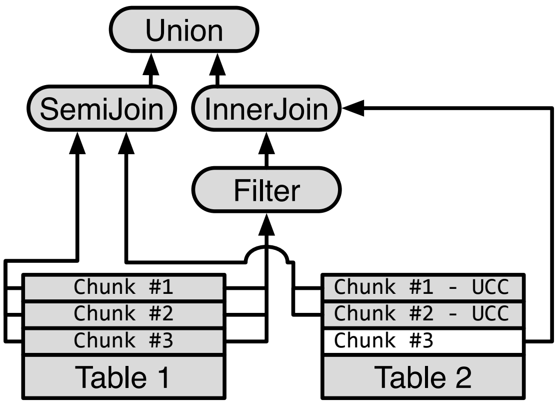


Figure 8.3.: Example query plan considering chunks without data dependencies. The plan consists of two branches: the left one uses a UCC-based optimization and solely accesses the first two chunks of *Table 2* where this UCC is valid. The right one only accesses the third chunk without optimizations.

[39]A chunk's min and max can be retrieved with low overhead if such values are maintained upon insert.

8.3. Evaluation

To evaluate our workload-driven dependency discovery system for query optimization in combination with some of the dependency-based query optimization techniques themselves, we implemented the system as a Hyrise plugin; for the source code, see Footnote 38 above. The plugin analyzes Hyrise's plan cache to determine and validate beneficial dependency candidates, as explained in Section 8.2.2.

8.3.1. Experimental Setup

The following experiments were executed on the same machine as used above, an AMD EPYC 7F72 with 24 cores (base clock: 3.2 GHz, boost: up to 3.7 GHz) with 512 GB of main memory running Ubuntu 20.04 (Kernel 5.4.0-105). We compile Hyrise with g++ version 10. For our evaluation, we use the TPC-H, TPC-DS[40], and Join Order Benchmarks (JOB). These benchmarks have been described in detail in Section 4.1.1. The JOB's real-world data is particularly interesting to evaluate the effects of dependency-driven optimizations in practice. For the TPC-H and TPC-DS, we use a scale factor of 10 and, for the JOB, we use the original paper's dataset[41].

In the subsequent experiments, the system automatically identifies and validates dependency candidates for three optimizations presented in Chapter 7 and implemented in Hyrise. In addition, the performance impact of these optimizations is investigated. The optimizations are labeled *O-1*, *O-2*, and *O-3* and use different data dependencies: *O-1* corresponds to the UCC-based utilization of potentially more efficient semijoin strategies (Section 7.2.1); *O-2* relates to the FD-based reduction of `GROUP BY` attributes (Section 7.3.1); *O-3* refers to the OD-based join avoidance (Section 7.4.2). Note that *O-3* also includes the UCC-based *invisible join* (Section 7.2.1) due to the similarity of these two techniques.

To simulate a scenario where the system has to identify all dependency candidates by itself in an unsupervised fashion, no (foreign) keys or constraints are defined for the benchmarks. Such scenarios occur, e.g., when data resides in CSV files where definitions of keys and other metadata are not provided [Abe+18] or if the schema was not adequately modeled by a database administrator (DBA) or developer. For each benchmark, i.e., the TPC-H, TPC-DS, and JOB, we first execute the entire workload 100 times and calculate the average (mean) of the total workload execution time. Then, we invoke the dependency discovery plugin, which in practice, would run in the background during query processing. Afterward, the workload is executed another 100 times. This time, dependency-based optimizations are applied if the plugin has identified dependencies. In this way, the

[40]Currently, Hyrise supports 47 of the 99 TPC-DS queries. See the source code on GitHub for details.
[41]Dataset for the Join Order Benchmark: `http://homepages.cwi.nl/~boncz/job/imdb.tgz`

experiment measures the execution times of the plugin and the fully optimized workloads separately. All experiments are executed in single-threaded mode because single-threaded experiments are sufficient for demonstrating the impact of dependency-based optimization techniques. We did not obtain different observations from multi-threaded experiments, which is expected, given that our techniques do not introduce any interdependencies interfering with multi-threaded processing.

8.3.2. Limitations

The following evaluation relies on the same workloads as the previous evaluations. Please refer to Sections 4.1.1 and 4.1.6 for a detailed discussion of these workloads and their limitations. Additionally, an evaluation with the trace of a single real-world workload would not allow more robust conclusions for data dependencies. Instead, a large set of real-world workloads operating on different datasets would have to be evaluated to allow more general conclusions. However, “the public availability of real-world database workloads is limited” [Vog+18, p. 1] and the existence of a large number of data dependencies on numerous real-world datasets has been demonstrated before [PN16; SP22].

Moreover, the data dependency-based optimization techniques are not evaluated in complete isolation. Disabling all of Hyrise’s other optimization rules would degrade the performance significantly and distort the measurements. Thus, interactions with Hyrise’s existing optimization rules cannot be avoided. In fact, if dependency-based query optimization were integrated into productive DBMSs, such interactions would also occur. Hence, not evaluating these techniques in pure isolation leads to a more realistic setting.

8.3.3. Optimization Performance

Table 8.1 shows the latency and throughput performance impact of the three data dependency-driven query optimization techniques separately and combined. The impact is compared with a *baseline*. None of the dependency-based optimizations are utilized for this baseline because no dependency is known to the DBMS, as explained in the setup above. Apart from the impact on the execution time, Table 8.1 also shows how many queries were improved or degraded, the average throughput change, and the maximum improvement.

As a first observation, the realized benefits are substantial, which is particularly true for the JOB and TPC-DS, where a combination of all three optimizations reduces the execution times by 26 % (10.7 s of 40.7 s) and 9 % (2.9 s of 31.6 s), respectively. While the observed performance benefits do not represent a formal verification of our approach, they indicate that the proposed selection process promotes useful dependencies. Also,

Table 8.1.: Performance impact (latency and throughput) of optimizations O-1 to O-3. $\sum$ shows the combined execution time of all queries. #↓ indicates the number of improved queries (with lower execution time), #↑ degraded ones; only changes larger/smaller than ±5 % were considered for these two metrics. ∅ indicates the average (geometric mean) throughput change across all of the workload's queries. max↑ shows the maximum observed speed-up.

	JOB (113 Queries)					TPC-DS (47 Queries)					TPC-H (22 Queries)				
	Latency			Throughput		Latency			Throughput		Latency			Throughput	
	$\sum$	#↓	#↑	∅	max↑	$\sum$	#↓	#↑	∅	max↑	$\sum$	#↑	#↓	∅	max↑
Baseline	40.7 s	–	–	–	–	31.6 s	–	–	–	–	38.9 s	–	–	–	–
O-1	-8.5 s	61	20	11 %	2.5 ×	-1.6 s	26	1	9 %	1.5 ×	-0.7 s	2	1	1 %	1.1 ×
O-2	0.0 s	0	0	0 %	0	-0.6 s	2	0	1 %	1.1 ×	-1.5 s	1	0	2 %	1.5 ×
O-3	-7.1 s	65	6	25 %	5.3 ×	-1.1 s	6	0	30 %	61 ×	-0.3 s	0	0	1 %	1 ×
All	-10.7 s	72	13	29 %	5.5 ×	-2.9 s	28	1	40 %	61 ×	-2.1 s	3	1	3 %	1.6 ×

for these two benchmarks, the performance of the vast majority of all queries is affected positively; more than half of JOB's 113 queries are improved. The number of improved queries always outweighs the degraded ones. Also, in the combined TPC-H case, 3 of 22 queries improved while only a single query's performance declined, resulting in a 5 % (2.1 s of 38.9 s) execution time reduction overall.

In addition to the summary statistics, we point out that each of the optimizations causes significant improvements in some queries. Details can be obtained from Figures A.2 to A.4 in Appendix A.1.2. For example, for O-1, we observed a 2.5 × speed-up for JOB's Query 14c; for O-2, we observed a 1.5 × speed-up[42] for TPC-H's Query 10; for O-3, we observed a 61 × speed-up of TPC-DS' Query 32. Furthermore, the table demonstrates that the optimizations compete with each other in some cases. For instance, considering the individual impacts of O-1 and O-3, one could expect to see a larger combined improvement for the JOB, which is not the case because both optimizations target joins. Thereby, the optimizations decrease the other optimization's potential.

Discussion. The magnitude of the query performance impact of dependency-based query optimization techniques depends on the database schema, the queries, the evaluated optimizations, and the underlying data. For example, O-1 and O-3 explicitly target joins, which frequently occur in the JOB and have a more considerable overall impact in the JOB and TPC-DS than aggregates, which are targeted by O-2. Also, the snowflake schema-like data models of the TPC-DS and JOB appear to be more suitable for such optimizations. Finally, it depends on the data itself whether dependencies exist and, hence, whether optimizations can be applied.

[42]For the skewed JCC-H [BAK17] dataset the observed speedup was even higher at ≈ 2.6 ×.

8.3.4. Discovery and Selection Overhead

Using dependencies for query optimization introduces a tradeoff between the realized performance improvements and the overhead for determining dependencies. Therefore, while the overhead of the initial dependency discovery and selection process is visible in some cases, it must be judged considering the achieved performance gains. This overhead is demonstrated in Table 8.2. The *Overhead* columns indicate the time necessary to generate dependency candidates and their validation alongside the number of such dependencies that are depicted in the *Candidates* columns.

Table 8.2.: Candidate information and overhead of unsupervised data dependency discovery. *Candidates* displays the number of candidates divided into the total number of identified candidates for the workload and the successfully validated ones. The *Overhead* time is split into the time taken for candidate generation *(Generate)* as well as for validating these candidates *(Validate)*. *Size* refers to the memory footprint of the validated dependency data structures in bytes.

	JOB (113 Queries)					TPC-DS (47 Queries)					TPC-H (22 Queries)				
	Candidates		Overhead			Candidates		Overhead			Candidates		Overhead		
	Total	Valid	Generate	Validate	Size	Total	Valid	Generate	Validate	Size	Total	Valid	Generate	Validate	Size
O-1	10	10	24 ms	152 ms	760	13	11	8 ms	32 ms	836	5	5	1 ms	1.9 s	380
O-2	0	0	1 ms	<1 ms	0	53	2	1 ms	2 ms	152	19	5	<1 ms	3.4 s	380
O-3	18	15	20 ms	67 ms	1 140	23	7	8 ms	50 ms	580	12	7	1 ms	1.9 s	532
All	20	17	22 ms	223 ms	1 292	72	14	9 ms	56 ms	1 112	26	9	2 ms	3.4 s	684

According to Table 8.2, our approach determines a manageable number of a few dozen dependency candidates for all investigated benchmarks. The share of candidates that are generated from the workload, but that cannot be validated on the datasets, is larger for the TPC workloads. The largest number of dependencies can be validated for the combined JOB scenario, 17. Even in this case, the memory size of the validated dependencies barely exceeds a kilobyte. A list of all dependencies discovered for the three workloads can be obtained from Daniel Lindner's master's thesis [Lin22].

Also, the validation time clearly outweighs the candidate generation time for almost all cases. The different complexities of the underlying procedures can explain this observation: for candidate generation, query plans that consist of a few dozen or a few hundred nodes must be parsed. In contrast, the validation might require analyzing millions of values for the evaluated datasets.

For the scenario where all three optimizations are applied, the system searches for beneficial UCCs, FDs, and ODs. The overhead for O-1, O-2, and O-3 does not necessarily result in the combined overhead as some dependency candidates might be relevant for multiple optimizations. The necessary time and break-even rate (considering the improvements reported in Table 8.1) varies largely for the three benchmarks: for the JOB and TPC-DS, only a small fraction of an entire workload run is necessary to break even. For the JOB, the advantage outweighs the overhead by 44 ×, for the TPC-DS by 45 ×. For the TPC-H, all queries must be executed approximately twice; the realized benefit corresponds to 0.6 × of the observed discovery overhead.

There are two main reasons for the large differences in validation runtime. First, some types of dependency candidates are more challenging to validate than others. For example, determining uniqueness (for UCCs) is generally simpler than sorting large tables with dozens of millions of rows (for ODs). On the other hand, invalidating candidates is often faster than successfully validating candidates because the procedure can be aborted early. In addition, the table's size and the nature of the data impact validation costs. In fact, successfully validating a single UCC on c_address of TPC-H's customer table with ≈1.5 m rows takes 0.8 s and is responsible for 24 % of the discovery and selection runtime. In contrast, the TPC-DS profits from an OD on a small dimension table that can be validated in only 23 ms. The last observation conforms with the argumentation presented in Section 8.2.3: there are valuable data dependencies leading to performance improvements that can be (re)validated quickly.

Discussion. The above experiments have shown that the effort for dependency discovery is usually amortized quickly, particularly for the more realistic datasets, i.e., JOB and TPC-DS. More importantly, the discovery process is not tied to query execution and optimization. Hence, it can be executed as an asynchronous background task whose runtime is significantly less relevant. The storage or memory size of dependencies is in the range of a few kilobytes and hence, negligible.

Furthermore, our experiments have shown that beneficial dependencies exist on different datasets. The ratio of valid and candidate dependencies is high (85 %) for the JOB that is based on real-world data, compared to the TPC-H (35 %) and TPC-DS (19 %). Moreover, our approach is scalable: evaluations on larger TPC-H and TPC-DS datasets show that the performance benefits increase at least as quickly as the validation efforts [Lin22].

8.4. Related Work

In this section, we discuss related work of our workload-driven dependency discovery approach. Some dependency-based query optimization techniques are used by commercial database systems, as explained in Chapter 7. However, to the best of our knowledge, such systems do not determine the necessary dependencies in an unsupervised fashion. Thus, the utilized optimizations are based on user-defined dependencies. Practical examples include UCCs and INDs (referential integrity constraints) to reduce the number of statistical views [IBM22b] in IBM's Db2 or FDs (which are only validated if instructed by the user) for improved selectivity estimates [Thed] in PostgreSQL.

Pena et al. [Pen+18] present a mechanism that automatically incorporates FDs to rewrite SQL queries. Their system needs to discover *all* functional dependencies first. Afterward, they determine which queries might profit from which FDs by comparing (i) matrices representing the FDs and (ii) matrices consisting of attribute occurrences in the workload's queries.

There are a few differences to our approach. The authors do not limit the dependency discovery process to relevant candidates, leading to higher potential discovery, selection, and maintenance costs than our approach. Furthermore, the proposed system does rely on query rewriting-based techniques only. As presented in Section 7.1, numerous techniques cannot be achieved with pure rewriting and optimization opportunities are neglected. Additionally, our system includes other dependency types that show a more considerable optimization potential than FDs. Lastly, the work of Pena et al. does not discuss dependency mutation challenges, as presented in Section 8.1.3.

The partitioning-based technique for FDs presented by Gianella et al. [Gia+02] is similar to our idea of utilizing modern column store concepts to handle potentially invalid dependencies. The authors propose to split a relational table horizontally into two partitions, one in which an FD is true and one that contains all the violations to the FD. Then, queries to the partitioned table also need to be split, i.e., rewritten to read from both partitions. The subquery that reads from the partition, where the FD holds true, can use FD-based optimizations; the other subquery is executed regularly without optimizations. In contrast to this technique, we do not explicitly create partitions but build on chunks that are implicitly created and part of the database system's storage layout. In addition, our approach is not limited to FDs but currently works with FDs, ODs, and UCCs. Also, the motivation for the approaches is different. In general, our approach assumes the validity status of dependencies to be only temporarily unknown due to data-modifying statements and not to be permanently violated. Furthermore, we do not need to rewrite the query itself but can precisely adapt single operators of the query plan.

8.5. Conclusion and Future Work

We presented an approach that efficiently discovers data dependencies based on concrete, given workloads and applies them during query optimization to generate more efficient query plans to improve performance. The three challenges when applying data dependencies for the purpose of query optimization are the *discovery*, *selection*, and *mutation* of relevant dependencies. We proposed an integrated solution that tackles these challenges with our workload-driven, lazy dependency discovery approach, incremental validation and maintenance techniques, and concepts of column store DBMSs.

An evaluation performed with the open-source DBMS Hyrise showed promising results for the analytical benchmarks TPC-H, TPC-DS, and JOB: the observed runtime improvements are substantial with up to 26 % of reduction in workload execution time and up to 61 × query speed-up. The overhead for determining and validating dependency candidates is reasonable: between 2 % and 162 % of the observed execution time reduction. Note that the discovery efforts can run as asynchronous background tasks and are, therefore, not a factor that needs to be considered to impact the system performance immediately. Furthermore, in the conducted experiments, we observed that those dependencies that turned out to be beneficial for query optimization could also be revalidated quickly, which mitigates the impact of dependency mutation.

Future Work. Various ideas for future research directions for workload-driven, lazy data dependency discovery exist. For instance, we have evaluated three dependency-based query optimization techniques in this thesis. Numerous additional techniques could be implemented and evaluated based on the survey in Chapter 7. These techniques might also be based on dependency types that have not been considered, e.g., INDs. In addition, the implemented techniques could be used to improve the optimizer's *cardinality and cost estimation* instead of enabling plan transformations as above.

Another interesting aspect is the transferability of the obtained results to other types of database architectures. For example, the impact of dependency-driven query optimization on the performance of traditional row-oriented, disk-based DBMS, such as PostgreSQL [SRH90], or of cloud-based data warehouses, such as Snowflake [Dag+16] could be evaluated. Moreover, the impact on the discovery overhead caused by non-column-oriented storage layouts could be examined. Lastly, the proposed solutions could be benchmarked with an extended set of real-world datasets and workloads.

Part III.

Application Scenario of Unsupervised Database Optimization and Conclusion

9 A Cockpit for Unsupervised Database Optimization

The first two parts of this thesis have investigated two aspects of unsupervised database optimization that are directly related to improving the performance of database management systems (DBMSs). In this chapter, we present our cockpit for unsupervised database optimization. The cockpit facilitates the adoption process of such unsupervised techniques by offering a platform for interactive experiments and building trust in them.

As discussed in Chapter 1, autonomous DBMSs can support database administrators (DBAs) in complex administration tasks. However, according to interviews with DBAs and DBMS customers, autonomous solutions are often distrusted [Pav+17]: such solutions are believed to only work in artificial scenarios and lack the necessary robustness for operating in real-world environments. An evaluation of the actual benefit on workload performance and the introduced resource and processing overheads by autonomous techniques is necessary to build trust and demonstrate their advantages. Additionally, creating an understanding of the reasoning behind the decisions taken by autonomous systems, i.e., achieving *explainability* [Gun+19], can facilitate the adoption process [Met+19].

Our interactive cockpit aims to create the necessary trust and opportunities for practical experimentation by directly comparing unsupervised with conventional systems regarding their performance during operation. For this purpose, workloads and system configurations can be modified to evaluate database performance in specific situations and scenarios. This direct comparison approach is in line with a similar technique applied to SQL databases in Microsoft Azure [Das+19]. Customer workloads are sent to two systems, system A and B. System A is a conventional system as tuned by the customer. On the other hand, system B is a duplicate of system A and is used for experiments with unsupervised optimization techniques. Thereby, the performance of both systems can be compared in a real-world scenario, while it is guaranteed that the customer system's performance is not affected in any way.

In the following, all examples refer to our research database system Hyrise [Dre+19]. However, the cockpit is not conceptually limited to only supporting Hyrise as a host DBMS. As long as the DBMS provides the necessary metrics and monitoring information via SQL, the cockpit could be connected with other DBMSs. The remainder of this chapter provides an overview of our cockpit along with an application scenario in Section 9.1. Afterward, we discuss the cockpit's architecture in Section 9.2.

Parts of this chapter have been published in a demonstration paper [Kos+21]. The cockpit was developed with nine undergraduate students as part of their final project. The paper's primary authors, Boissier and the thesis author, developed the cockpit's concept and guided the implementation, which was almost entirely handled by the students. Furthermore, the primary authors prepared the original draft for publication. The co-authors improved the paper's material and its presentation.

9.1. Overview

The cockpit should enable database engineers and administrators to experiment with unsupervised database optimization techniques. For this reason, the cockpit incorporates the following key concepts, which are also reflected in the design of the user interface:

- Monitoring: metrics to inspect workload performance (e.g., throughput or latency), system resources (e.g., CPU utilization and memory footprint), and the current workload (e.g., expensive statements and data access counters) are continuously monitored, visualized, and stored in a time-series database for convenient analysis.
- Variable workloads and load intensity: Users can provide their own workloads and datasets or choose from a variety of synthetic ones (e.g., the TPC-H, TPC-DS, TPC-C, and Join Order Benchmark (JOB)). In terms of parallel queries per second, they can regulate the pressure that is put on the system.
- Interactivity: users can activate plugins that conduct unsupervised database optimization and observe their impact on workload performance as well as on the system's resources.
- Explainability: we provide the opportunity to inspect the reasoning behind autonomously taken decisions to facilitate a better comprehension of such decisions.

9.1.1. User Interface

In the following, we explain which information is displayed in the cockpit and how the cockpit enables user interaction. Note that the cockpit's functioning is best demonstrated in an actual usage scenario where its visualization and interaction capabilities can be experienced. A guided demonstration [Kos+21] of the cockpit that includes a tour of the user interface can be obtained online[43].

[43]Screencast of Hyrise cockpit demonstration: `https://vimeo.com/671073749/ffbf158a86`

The web browser-based cockpit contains three monitoring views and three configuration panels. The views provide overviews of the current workload and the evaluated database instances. These views' charts depict general metrics, such as latency, throughput, or CPU and memory utilization. Users can choose to either display each instance separately side by side in the *Comparison* view, as demonstrated in Figure 9.1. Alternatively, the metrics for all instances can be displayed in a single graph in the *Overview* view.

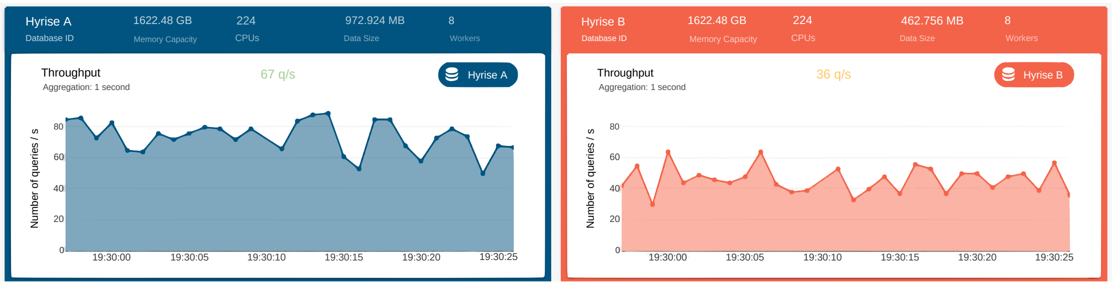


Figure 9.1.: Screenshot of the throughput and system part of the *Comparison* view. Two equivalent Hyrise instances run identical workloads. The left instance was tuned by the unsupervised clustering plugin and exhibited a higher throughput. The compression plugin decreased the right instance's data size.

The *Comparison* view also allows for showing diagrams that cannot be aggregated, e.g., heatmaps (Figure 9.2) displaying access patterns per chunk and attribute or treemaps (Figure 9.3) depicting per-attribute memory consumption. The *Workload Analysis* view displays the most expensive queries per database instance and the runtime share of the different database operators to enable quick analyses of the currently processed workload.

The cockpit user can adjust the workload and the plugins via two configuration panels. The *Workload Configuration* allows (un)loading of table data, starting multiple workloads, and modifying the number of queries per second. Plugins can be (de)activated with the *Plugin* panel. The panel also allows adjusting plugin settings, e.g., the storage budget for an index selection plugin. Moreover, the panel displays log messages that inform about the plugins' actions. These messages are also displayed in the charts above to enable the user to relate performance changes with plugin activity. The following example log messages were obtained when the compression plugin [Boi22] optimized a TPC-H dataset.

```
Compression Plugin [14:54:19]: Compression Plugin initialized.
Compression Plugin [14:55:32]: Encoded 92 of 92 segments of lineitem.l_comment
using LZ4-SIMDBP128: saved 348.85 MB.
Compression Plugin [14:55:49]: Encoded 2 of 3 segments of customer.c_name using
FixedStringDictionary-SIMDBP128: saved 3.48 MB.
Compression Plugin [14:55:52]: Compression optimization finished: memory budget
is feasible (budget: 650 MB, current size of table data: 647.68 MB).
```

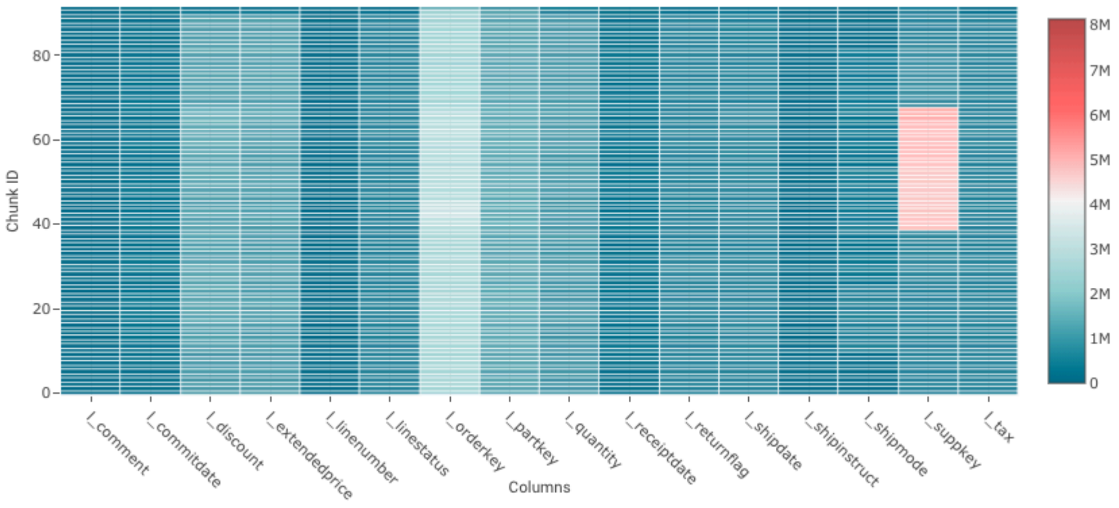


Figure 9.2.: Screenshot of the cockpit's access heatmap for TPC-H's lineitem table. Cells represent segments, i.e., a column's part of a chunk. Warmer colors indicate frequent accesses; such information can be utilized for identifying data access patterns and comprehending unsupervised data clustering decisions.

The flexible interaction with plugins per database instance enables users to experiment with them and observe their performance implications. Thereby, not only the interplay of different plugins [KS20] can be evaluated, but also alternative approaches to the same problem, e.g., various index selection algorithms, can be compared easily.

9.1.2. Application Scenario

In the following, we sketch a possible scenario for using our cockpit in practice. Martine works as a DBA in a large enterprise. The company uses the relational database system Hyrise, which relies on various techniques for unsupervised database optimization. These techniques conduct optimizations, e.g., creating secondary indexes, reducing the main memory footprint via compression, or clustering the data. Martine is interested in the advantages of all these approaches. However, she usually refrains from such approaches as (i) their impacts are hard to predict, (ii) she has experienced devastating performance impacts when applying heavy compression to database tables, (iii) she fears that these approaches fail when unexpected events, e.g., load spikes, occur, and (iv) she is afraid that combining multiple approaches might lead to undesired behavior.

By using our cockpit for unsupervised database optimization, Martine utilizes a recorded query trace of a production system to replay the workload of last year's severe blizzard season to evaluate the impact of the plugins in different highly challenging scenarios. She learns that reinforcement learning (RL)-based index selection provides beneficial configurations instantaneously and that the improved pruning rates caused by automated table clustering quickly set off the initial reorganization costs.

Simultaneously, Martine gains confidence in applying compression as she realizes how much the system can be compressed without affecting the performance of the production system. Her confidence increases when she combines the provided heatmap (cf. Figure 9.2) that displays access counts with the plugin's log messages. As the cockpit also allows her to examine the correlation and interplay of autonomous components, Martine balances the memory reduced via compression with the memory invested into secondary indexes.

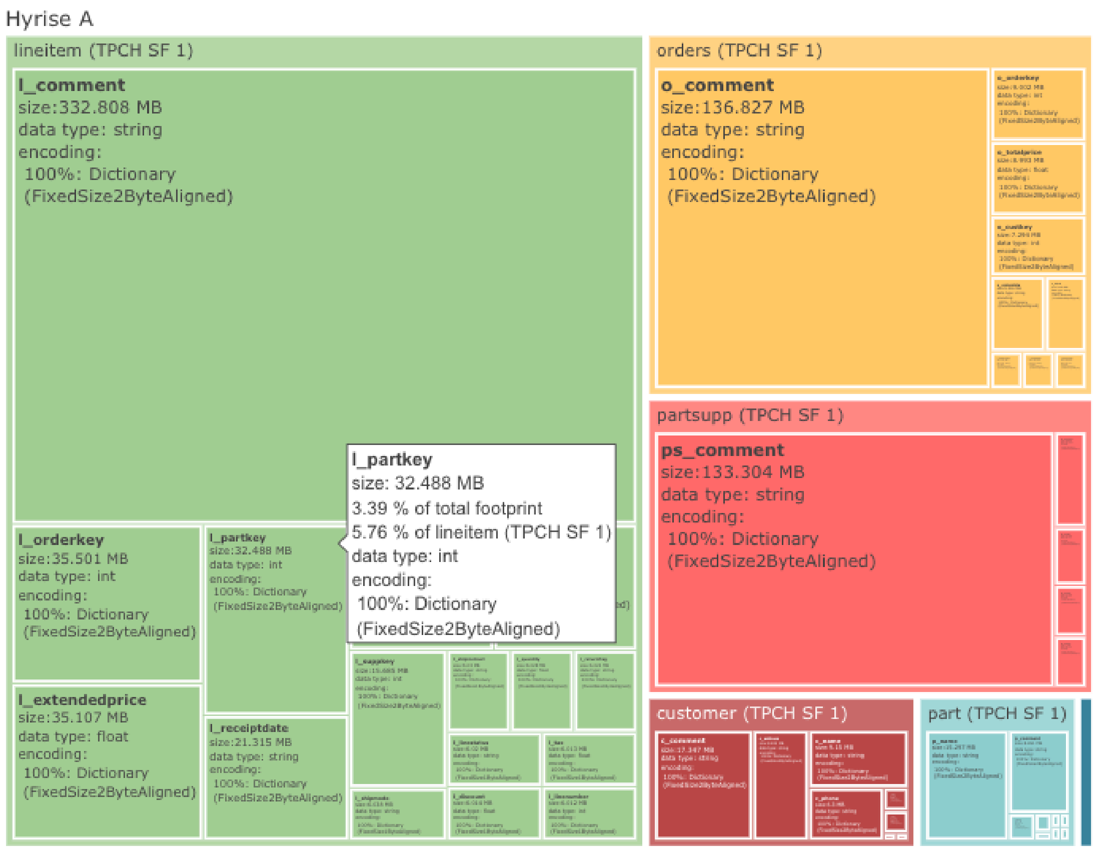


Figure 9.3.: Screenshot of the cockpit's memory footprint treemap for the TPC-H dataset (SF 1). This diagram allows to quickly identify the largest tables and columns. The treemap also offers detailed information about the currently used encoding schemes, data types, and sizes.

9.2. Cockpit Architecture

In the following, we discuss the cockpit's architecture and implementation[44] details. The cockpit's architecture is depicted in Figure 9.4. The components of the cockpit handle three main tasks: (i) the frontend allows the user to interactively examine and modify the processed workload as well as current system configuration, displays performance metrics, and provides access to the plugins that conduct database optimization, (ii) the cockpit backend handles the communication between the frontend and the investigated systems (i.e., Hyrise instances) as well as the periodic monitoring and storing of the displayed metrics. (iii) The workload generator is responsible for creating queries and putting pressure on the evaluated systems.

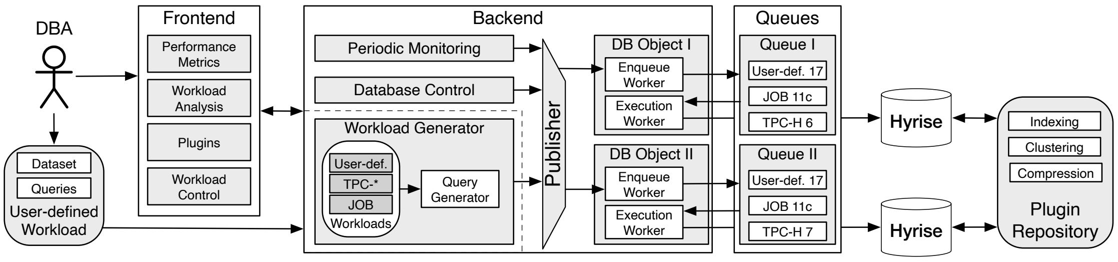


Figure 9.4.: Architecture of the cockpit for unsupervised database optimization. The figure's example workload represents a mixture of a used-provided workload trace *(User-def.)*, TPC-H- and Join Order Benchmark-based queries.

While the frontend is implemented in Vue.js [You], we use Python 3 throughout all other components. Hyrise implements the PostgreSQL wire protocol [Thec] so that queries can be transmitted via widely available clients and libraries. DBMS metrics, for instance, the CPU utilization, access counts, or the memory footprint of a table, can be easily obtained via SQL queries, too.

One of the main challenges during the cockpit's implementation was the harmonization of the, sometimes contrary, design requirements:

- High load generation: for insightful evaluations, the ability to stress the systems under investigation with high system loads is mandatory because trust in unsupervised approaches is only achieved if extreme situations can be handled well.
- Fair load distribution: even though it is technically impossible to generate the same workload — including identical arrival times for queries — for all evaluated systems, the workload generator must ensure that the workload differences are kept as small as possible.

[44]Hyrise cockpit source code on GitHub: `https://github.com/hyrise/Cockpit`

- Simple workload extension: the cockpit should serve as an evaluation platform for DBAs and developers. Thus, providing synthetic queries and data is not sufficient. Instead, users must be able to add their own workloads and data in a simple fashion.

To achieve a fair load distribution, generate a sufficient number of queries, and follow a clear separation of concerns, we decided to decouple the query generation from the actual sending to the database instances. Thereby, these tasks can operate independently in different processes and on different CPU cores to avoid the workload generation becoming a bottleneck. The *query generator* implements the logic for creating a configurable number of queries per second from a predefined query set. By default, all queries of a workload are chosen with the same probability. The cockpit user can modify the query distribution to emphasize specific queries and reproduce realistic scenarios.

Afterward, the *workload generator* passes the queries to the backend database objects, which maintain query queues (implemented with Python's multiprocessing library). The database objects handle the actual query sending with psycopg2 [Var]. We utilize ZeroMQ's [Zer] publisher-subscriber pattern for efficient inter-process communication. Only database control commands that target specific instances, e.g., for loading plugins, are not published but directly sent to the appropriate node. The task of enqueueing new queries is also decoupled from the sending procedure. Otherwise, the delayed response of a poorly performing database instance could affect the fair and even query distribution. Existing real-world workloads can be replayed by simply providing the necessary table data and queries as CSV and SQL files.

Flask [Pal] handles the communication between the backend and frontend. Furthermore, the time series database InfluxDB [Inc19] is used to store the observed metrics permanently. Thereby, we enable the analysis of historical performance data to comprehend particular system and plugin behavior in more detail. We facilitate reproducibility and the cockpit's setup process by providing a Docker setup for all components.

9.3. Summary

We presented our cockpit for unsupervised database optimization. The cockpit allows monitoring performance KPIs, analyses, e.g., of data access patterns, and comprehending the decisions of unsupervised database optimization techniques. The cockpit also serves as a platform for interactive experiments, including side-by-side comparisons of conventional and unsupervised systems. Confidence in unsupervised systems is established by investigating how they compare against conventional, manually administrated systems when both process the same workload. Thereby, the cockpit aims at facilitating the adoption of systems that act in an unsupervised fashion by demonstration.

10 Conclusion

A human *with* a machine beats both human and machine.[45]

Marc-Uwe Kling — Author

Database systems offer a multitude of competing configuration options that must be adequately balanced for good performance and resource utilization. Currently, highly skilled human database administrators (DBAs) take care of the administration of database management systems (DBMSs). Such tasks are time-consuming, expensive, and arduous [Pav+19]. Even further, the complexity of these configuration tasks is increasing due to, e.g., more interdependent configuration options, DBAs lacking domain knowledge in cloud environments, and more complicated as well as flexible workloads. To handle this complexity and keep up with ever-increasing data and query volumes, DBMSs need to become more autonomous and apply *unsupervised database optimization*. For example, heuristics, linear programming (LP) approaches, or machine learning-based methods can be used to determine optimized configurations for database administration tasks.

In this context, we researched how new unsupervised database optimization techniques can efficiently improve the performance of database systems. We focused on two distinct aspects, efficient index selection (Part I) and the use of data dependencies for query optimization (Part II). For both, we first established the necessary foundations by introducing definitions and formalizations. Afterward, we surveyed and analyzed existing approaches by studying related work. Based on these analyses, we developed new, improved approaches.

In Part I, based on the analyses of state-of-the-art index selection algorithms, we presented two novel, improved algorithms that complement each other. In Part II, we developed a workload-driven approach that enables the utilization of data dependencies for query optimization. All of our approaches are available as open-source implementations.

[45]Translated by the author from: Ein Mensch *mit* Maschine schlägt sowohl Mensch als auch Maschine.

The approaches for both aspects were evaluated with industry-standard and real-world data-related DBMS benchmarks. Finally, we presented our cockpit for unsupervised database optimization that offers an interactive experimentation platform for DBAs and developers to build trust in autonomous DBMSs and facilitate their adoption. We believe a combination of the abilities of sophisticated unsupervised approaches and human input to be most powerful. Examples are given in Chapter 5, where incorporating expert decisions can accelerate and improve an agent's training behavior, and in Chapter 8, where the rules to derive dependency candidates are defined manually.

Responses to the Initial Research Questions. The detailed analyses of existing techniques and our newly developed approaches allow us to answer the initially posed (in Section 1.4) research questions:

1. *How can we analyze, compare, and classify unsupervised index selection algorithms and investigate which factors influence their performance regarding the quality of the identified solutions as well as the required runtime?*

   We formalized the index selection problem and highlighted its challenges to facilitate the analysis of existing as well as the development of improved index selection algorithms. Subsequently, significant differences of existing index selection approaches were illustrated with an extensive survey and the reimplementation of seven approaches. The survey describes, analyzes, and compares the techniques across different dimensions, e.g., multi-attribute index support and the consideration of index interaction. Furthermore, we classified the modus operandi of the approaches into additive, reductive, and declarative ones.

   The reimplementations are the basis for an experimental evaluation. For this evaluation, we developed a flexible index selection evaluation platform. The platform allows comparing the quality of the identified solutions and the selection runtime in different scenarios. Moreover, fine-granular cost breakdowns can be created for detailed analyses. The observed differences are significant. In particular, the selection runtimes diverge by orders of magnitude. Besides the seven algorithms and three analytical benchmarks implemented for this work, the platform is easily extensible by other algorithms and workloads.

   Based on an experimental evaluation with the three benchmarks, we derived 10 general and 17 algorithm-specific insights. These insights build the foundation for the development of improved index selection algorithms. In particular, our evaluation indicates that algorithms producing close-to-optimal solutions exist. However, these approaches come with *considerable selection runtimes* for large problem instances. Simultaneously, fast approaches often have *difficulties identifying close-to-optimal solutions* due to less consideration of index interaction effects.

2. *How can we scale index selection processes to efficiently determine near-optimal index sets, even for large problem instances and considering complex effects, such as index interaction?*

   We presented two, new efficient index selection algorithms to overcome the shortcomings of existing solutions. The presented algorithms focus on two different application scenarios: First, *Extend* aims for close-to-optimal solutions and general applicability without preparation. *Extend* is an additive approach that benefits from not limiting index widths and candidates. Instead, it works with an extension mechanism that opens up new index opportunities with every index decision. Previously chosen indexes are *extended* attribute-by-attribute. Thereby, *Extend* determines equivalent or better index configurations faster than other close-to-optimal index selection algorithms.

   Second, *SWIRL* complements *Extend* by focusing on scenarios where many index configurations for similar selection problems are required. *SWIRL* is a reinforcement learning (RL)-based index selection approach that internalizes which index candidates are beneficial in which situations during a preliminary training phase. Afterward, index configurations can be obtained almost instantaneously.

   Based on our evaluations, we conclude that techniques like RL and the iterative extension of indexes allow efficient index selection for large problem instances. Our techniques dominate their direct competitors in both metrics: the solution quality of the identified index configurations and the selection runtime. *Extend*'s solutions have the highest quality on average; it also determines the best solution of all candidates in the majority of evaluated scenarios. Simultaneously, it is an order of magnitude faster for large problem instances than direct competitors. *SWIRL* dominates its direct competitors, too. After training, *SWIRL* determines index configurations faster than all examined approaches; its selection runtime is within 1–3 seconds for large problem instances, while approaches producing high-quality solutions take multiple minutes or even hours in such scenarios. The quality of *SWIRL*'s solutions is close, on average, not more than 3 % worse compared to the quality of the best conventional algorithms, e.g., *Extend*. At the same time, *SWIRL* is capable of outperforming other fast index selection algorithms in terms of solution quality. For instance, for the Join Order Benchmark (JOB), the solutions of *DB2Advis* are, on average, more than 10 percentage points (pp), and for *DRLinda*, more than 20 pp worse. Based on specific experiments, we also conclude that *SWIRL* can generalize to handle unknown workloads. Moreover, *Extend*'s *anytime* capabilities outperform *DTA*'s, which was specifically designed with such capabilities in mind.

3. *How can a DBMS use data dependencies for improved query optimization, and how can relevant dependencies be identified in an unsupervised fashion?*

   Data dependencies, which exist on most real-world datasets, have various applications for query optimization, e.g., for query plan simplifications or improved cardinality estimation. We collected and explained 58 query optimization techniques based on functional, order, and inclusion dependencies as well as on unique column combinations to demonstrate how DBMSs can utilize dependencies for query optimization. These techniques were classified according to the necessary dependency type, the application area, and which query optimization activity is affected.

   Utilizing dependencies in practice is challenging. Typically, dependencies are unknown; their discovery and maintenance are expensive. We presented a workload-driven approach that creates promising dependency candidates by analyzing the processed workload. This analysis is executed as an unsupervised, asynchronous background procedure. Due to the workload analysis, the number of dependency candidates remains reasonable — a few dozens for the analyzed analytical workloads — permitting fast validation. Furthermore, we discussed techniques, e.g., the utilization of column store concepts, to mitigate the effects of invalid dependencies. The conducted evaluations show substantial performance benefits across all evaluated benchmarks. For instance, the JOB's runtime can be decreased by 26 % and some TPC-DS queries can be accelerated by a factor of up to 61. At the same time, the overhead caused by dependency candidate validation is between 2 % (TPC-DS and JOB) and 162 % (TPC-H) of the observed execution time reduction. Hence, the overhead is negligible for the more realistic workloads and datasets (TPC-DS and JOB). For the TPC-H, it pays off after less than two workload executions.

In view of our work on two aspects of unsupervised database optimization, DBMSs can improve their performance under consideration of the currently processed workload. *Extend* and *SWIRL* determine better index configurations faster than other index selection algorithms. Additionally, our workload-driven, lazy data dependency approach enables more efficient query plans by automatically identifying beneficial data dependencies for query optimization. Thereby, we take two crucial steps towards autonomous database systems.

A

Appendix

A.1. Additional Figures

This section presents additional supporting figures for some of the above's experiments.

A.1.1. Additional Figures for Part I

Figure A.1 reports the amount of information loss for different representation widths when reducing the dimensionality of *SWIRL*'s workload representation; see Section 5.2.3.

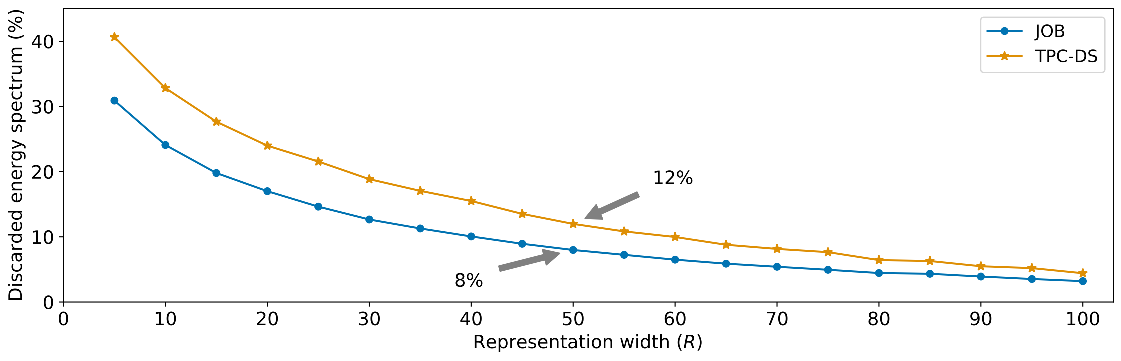
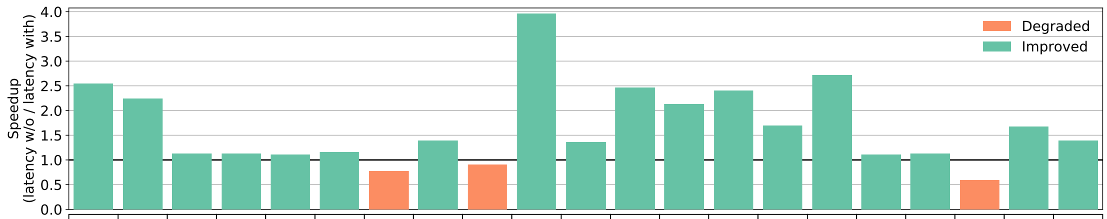


Figure A.1.: Discarded energy spectrum (%), i.e., a measure for the information loss, for increasing representation width (R) values between 5 and 100. Our implementation of SWIRL uses $R = 50$.

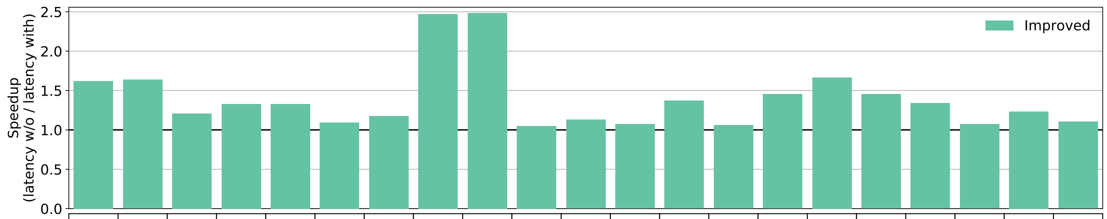
A.1.2. Additional Figures for Part II

Figures A.2 to A.4 report the per-query performance impact of our data dependency-driven query optimization approach presented in Chapter 8. The figures show the impact for the combined case of using all three optimizations, *O-1*, *O-2*, and *O-3*. Only queries whose runtime was increased or decreased by at least 5 % are shown.



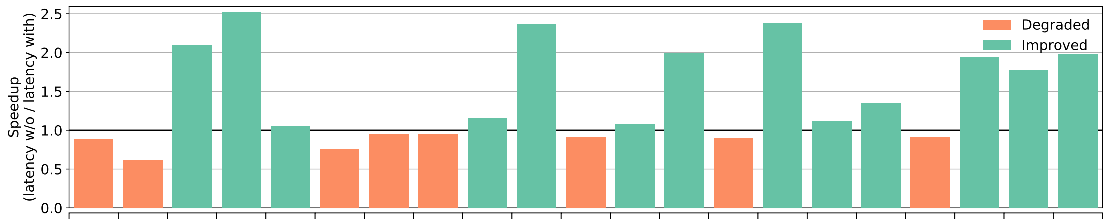
	10a	10b	11a	11b	11c	11d	12a	12b	12c	13a	13b	13c	13d	14a	14b	14c	15a	15b	15d	16a	16b
Speedup	2.54x	2.24x	1.13x	1.13x	1.11x	1.15x	0.77x	1.39x	0.90x	3.96x	1.36x	2.46x	2.13x	2.40x	1.70x	2.72x	1.11x	1.13x	0.59x	1.67x	1.39x
W/o (ms)	166.4	122.2	63.2	62.9	26.5	30.0	54.7	70.4	93.1	321.7	120.3	30.8	722.8	470.0	367.4	553.6	37.1	36.9	83.4	2300.5	3471.7
With (ms)	65.4	54.5	55.8	55.7	23.9	26.0	70.9	50.7	103.3	81.3	88.3	12.5	339.8	195.7	216.5	203.9	33.5	32.7	141.8	1375.1	2496.7

(a) Runtime speedup for the Join Order Benchmark (JOB) queries 10a to 16b.



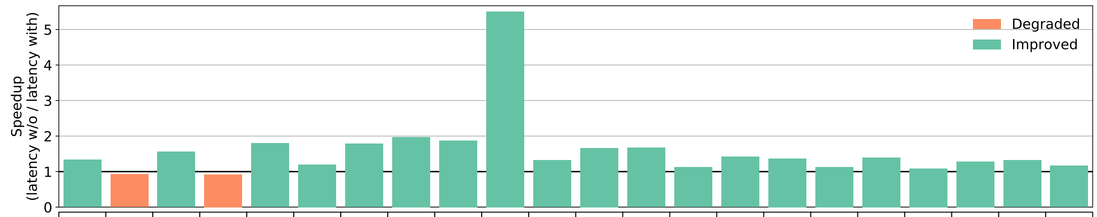
	16c	16d	17a	17b	17c	17e	17f	18a	18c	19a	19b	19c	19d	1a	1b	1c	1d	20a	20b	20c	21a
Speedup	1.62x	1.64x	1.21x	1.33x	1.33x	1.09x	1.18x	2.47x	2.48x	1.05x	1.13x	1.07x	1.37x	1.06x	1.45x	1.67x	1.45x	1.34x	1.07x	1.23x	1.10x
W/o (ms)	2416.7	2389.4	584.0	459.9	434.1	1715.6	989.8	655.1	574.1	422.9	263.2	335.3	993.5	25.2	10.6	12.5	10.6	685.1	578.8	428.6	84.4
With (ms)	1490.9	1459.7	483.7	346.5	326.4	1569.3	840.5	264.9	231.2	402.5	232.9	312.9	725.6	23.8	7.3	7.5	7.3	511.9	539.8	347.4	76.4

(b) Runtime speedup for the JOB queries 16c to 21a.



	22a	22b	22c	22d	23a	23b	23c	24b	25a	25c	26c	27c	28a	28b	28c	29a	29b	29c	2a	2b	2c
Speedup	0.88x	0.62x	2.10x	2.52x	1.06x	0.76x	0.96x	0.94x	1.15x	2.37x	0.91x	1.08x	1.99x	0.90x	2.37x	1.12x	1.35x	0.91x	1.94x	1.77x	1.99x
W/o (ms)	227.7	151.7	670.9	986.2	43.8	37.1	43.1	238.9	243.4	1034.3	452.6	84.4	380.0	48.4	444.5	230.2	232.4	280.4	53.5	44.3	26.8
With (ms)	258.2	246.3	319.6	391.7	41.5	48.8	45.1	252.9	211.6	436.0	498.0	78.5	190.6	54.0	187.3	205.9	171.7	308.2	27.6	25.0	13.5

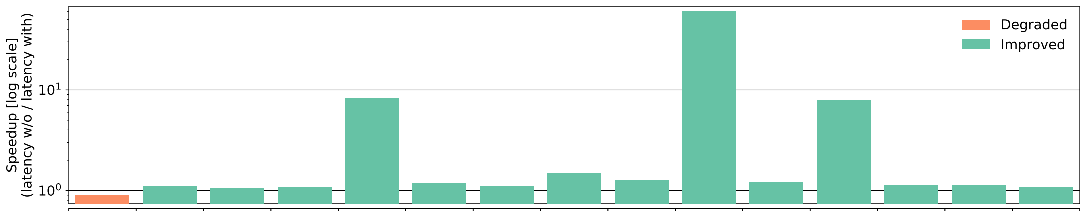
(c) Runtime speedup for the JOB queries 22a to 2c.



	2d	30b	30c	31b	32a	32b	3a	3c	4a	4c	5a	5b	5c	7c	8a	8b	8c	8d	9a	9b	9c	9d
Speedup	1.33x	0.93x	1.56x	0.92x	1.80x	1.20x	1.79x	1.97x	1.88x	5.50x	1.32x	1.66x	1.67x	1.12x	1.42x	1.37x	1.13x	1.40x	1.09x	1.28x	1.32x	1.17x
W/o (ms)	74.8	195.9	420.0	210.4	17.6	42.1	127.2	467.0	131.3	145.3	57.4	127.6	175.3	642.5	51.8	50.4	1988.3	362.1	283.6	129.3	305.8	556.2
With (ms)	56.2	211.4	269.2	228.2	9.8	35.0	71.2	236.8	70.0	26.4	43.4	76.7	104.8	572.2	36.6	36.9	1754.6	259.5	260.3	100.7	231.4	477.0

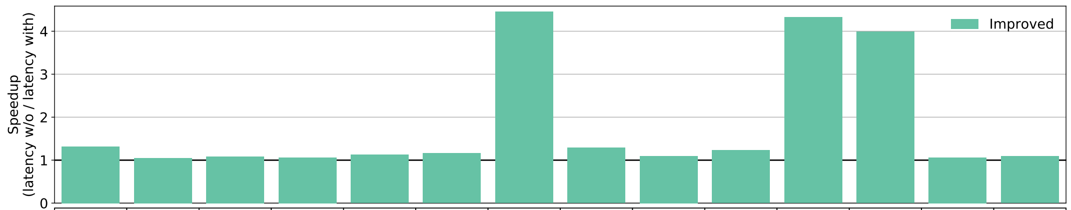
(d) Runtime speedup for the JOB queries 2d to 9d.

Figure A.2.: Speedup (latency *w/o* divided by latency *with*) for JOB queries for optimizations *O-1*, *O-2*, and *O-3*. *With* and *w/o* refer to using data dependency-driven query optimization. Overall 10.7 s (26 %) faster.



	01	06	13	15	16	17	25	26	29	32	34	37	39a	39b	45
Speedup	0.91x	1.10x	1.06x	1.08x	8.24x	1.20x	1.10x	1.50x	1.26x	61.00x	1.21x	7.96x	1.14x	1.14x	1.08x
W/o (ms)	222.0	174.6	459.8	118.0	240.6	391.4	194.1	176.1	603.1	48.8	187.1	301.5	2023.8	2010.1	127.1
With (ms)	243.3	158.2	433.0	108.9	29.2	326.5	176.5	117.2	477.3	0.8	155.2	37.9	1774.7	1761.5	118.1

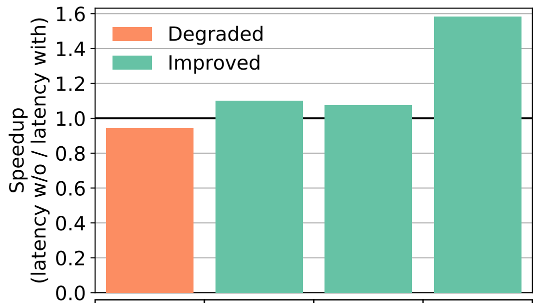
(a) Runtime speedup for the TPC-DS queries 01 to 45.



	48	55	62	65	73	79	82	85	88	91	92	94	97	99
Speedup	1.32x	1.05x	1.09x	1.06x	1.12x	1.16x	4.45x	1.29x	1.10x	1.23x	4.32x	3.99x	1.06x	1.09x
W/o (ms)	1201.1	88.7	619.0	2044.5	96.7	587.1	373.0	326.6	752.1	21.8	65.3	135.9	4131.0	1168.4
With (ms)	909.9	84.5	569.1	1922.4	86.1	505.5	83.8	253.2	684.6	17.7	15.1	34.1	3901.0	1069.4

(b) Runtime speedup for the TPC-DS queries 48 to 99.

Figure A.3.: Speedup (latency *w/o* divided by latency *with*) for TPC-DS queries for optimizations *O-1*, *O-2*, and *O-3*. *With* and *w/o* refer to using data dependency-driven query optimization. Overall 2.9 s (9 %) faster.



	TPC-H 02	TPC-H 08	TPC-H 09	TPC-H 10
Speedup	0.94x	1.10x	1.08x	1.58x
W/o (ms)	43.8	794.0	6269.6	4099.1
With (ms)	46.4	720.8	5829.2	2588.2

Figure A.4.: Speedup (latency *w/o* divided by latency *with*) for TPC-H queries for optimizations *O-1*, *O-2*, and *O-3*. *With* and *w/o* refer to using data dependency-driven query optimization. Overall 2.1 s (5 %) faster.

A.2. List of URLs

Table A.1.: List of URLs mentioned in footnotes of this thesis. Table entries in order of their appearance. Accessed: April 9, 2022.

Description	URL
Index selection evaluation platform on GitHub	`http://git.io/index_selection_evaluation`
Data dependency-driven query optimization on GitHub	`https://github.com/Bensk1/phd_thesis/releases/tag/source_code`
Index selection algorithm implementations on GitHub	`https://git.io/IndexSelectionAlgorithms`
TPC homepage	`https://www.tpc.org`
HypoPG source code and releases on GitHub	`https://github.com/HypoPG/hypopg`
Gurobi solver	`https://www.gurobi.com`
Source code for *NoDBA* [SSD18] on GitHub	`https://github.com/shankur/autoindex`
Source code of Lan et al.'s approach [LBP20] on GitHub	`https://github.com/rmitbggroup/IndexAdvisor`
SMARTIX's [Lic+20] experimental setup	`https://doi.org/10.5281/zenodo.3254967`
SWIRL's Bag Of Operators implementation on GitHub	`https://github.com/hyrise/rl_index_selection/blob/main/swirl/boo.py`
Anytime version of *Extend* on GitHub	`https://github.com/hyrise/index_selection_evaluation/blob/master/selection/algorithms/extend_algorithm_anytime.py`
NCVoter dataset	`https://www.ncsbe.gov/results-data/voter-registration-data`
Hyrise source code and documentation on GitHub	`https://github.com/hyrise/hyrise`
Hyrise's standalone SQL Parser on GitHub	`https://github.com/hyrise/sql-parser`
Dataset for the Join Order Benchmark	`http://homepages.cwi.nl/~boncz/job/imdb.tgz`
Screencast of Hyrise cockpit demonstration	`https://vimeo.com/671073749/ffbf158a86`
Hyrise cockpit source code on GitHub	`https://github.com/hyrise/Cockpit`

A.3. Publications

Our main contributions have been published at international conferences and journals, at VLDB, ICDE, EDBT, CIDR, and in the VLDB Journal. Furthermore, our work resulted in two patent applications:

(1) An Experimental Survey of Index Selection Algorithms

- 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”. In: *Proceedings of the VLDB Endowment* 13.11 (2020), pp. 2382–2395.

(2) Scalable and Effective Index Selection Algorithms

- Jan Kossmann, Alexander Kastius, and Rainer Schlosser. “SWIRL: Selection of Workload-aware Indexes using Reinforcement Learning”. In: *Proceedings of the International Conference on Extending Database Technology (EDBT).* 2022, pp. 155–168.
- Jan Kossmann, Rainer Schlosser, Alexander Kastius, Michael Perscheid, and Hasso Plattner. *Training an Agent for Iterative Multi-Attribute Index Selection.* European Patent Application EP22156399.2. February 2022.
- Rainer Schlosser, Jan Kossmann, Martin Boissier, Matthias Uflacker, and Hasso Plattner. *Iterative Multi-Attribute Index Selection for Large Database Systems.* European Patent EP3719663B1; US Patent Application 16/838,830. October 2020.
- Rainer Schlosser, Jan Kossmann, and Martin Boissier. “Efficient Scalable Multi-attribute Index Selection Using Recursive Strategies”. In: *Proceedings of the International Conference on Data Engineering (ICDE).* 2019, pp. 1238–1249.

(3) Data Dependency-driven Query Optimization

- Jan Kossmann, Thorsten Papenbrock, and Felix Naumann. “Data dependencies for query optimization: a survey”. In: *VLDB Journal* 31.1 (2022), pp. 1–22.
- Jan Kossmann, Felix Naumann, Daniel Lindner, and Thorsten Papenbrock. “Workload-driven, Lazy Discovery of Data Dependencies for Query Optimization”. In: *Proceedings of the Conference on Innovative Data Systems Research (CIDR).* 2022.

Our complementary contributions target further aspects of unsupervised database optimization and have been published at international conferences, journals, and workshops:

Component-based Framework for Autonomous DBMS Functionality

- Jan Kossmann and Rainer Schlosser. "Self-driving database systems: a conceptual approach". In: *Distributed And Parallel Databases (DAPD)* 38.4 (2020), pp. 795–817.
- Jan Kossmann and Rainer Schlosser. "A Framework for Self-Managing Database Systems". In: *Proceedings of the International Conference on Data Engineering (ICDE) Workshops*. 2019, pp. 100–106.
- Jan Kossmann. "Self-Driving: From General Purpose to Specialized DBMSs". In: *Proceedings of the VLDB PhD Workshop*. 2018.

Plugin Concept for DBMS Integration and Hyrise Cockpit

- Jan Kossmann, Martin Boissier, Alexander Dubrawski, Fabian Heseding, Caterina Mandel, Udo Pigorsch, Max Schneider, Til Schniese, Mona Sobhani, Petr Tsayun, Katharina Wille, Michael Perscheid, Matthias Uflacker, and Hasso Plattner. "A Cockpit for the Development and Evaluation of Autonomous Database Systems". In: *Proceedings of the International Conference on Data Engineering (ICDE)*. 2021, pp. 2685–2688.
- Markus Dreseler, Jan Kossmann, Martin Boissier, Stefan Klauck, Matthias Uflacker, and Hasso Plattner. "Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management". In: *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 2019, pp. 313–324.

Unsupervised Table Clustering

- Daniel Lindner, Alexander Löser, and Jan Kossmann. "Learned What-If Cost Models for Autonomous Clustering". In: *Advances in Databases and Information Systems (ADBIS)*. 2021, pp. 3–13.

Besides, we have researched database management system (DBMS) topics that are only of minor relevance to this thesis:

- Markus Dreseler, Jan Kossmann, Johannes Frohnhofen, Matthias Uflacker, and Hasso Plattner. “Fused Table Scans: Combining AVX-512 and JIT to Double the Performance of Multi-Predicate Scans”. In: *Proceedings of the International Conference on Data Engineering (ICDE) Workshops.* 2018, pp. 102–109.
- Jan Kossmann, Markus Dreseler, Timo Gasda, Matthias Uflacker, and Hasso Plattner. “Visual Evaluation of SQL Plan Cache Algorithms”. In: *Proceedings of the Australasian Database Conference (ADC) — Databases Theory and Applications.* 2018, pp. 350–353.
- Markus Dreseler, Timo Gasda, Jan Kossmann, Matthias Uflacker, and Hasso Plattner. “Adaptive Access Path Selection for Hardware-Accelerated DRAM Loads”. In: *Proceedings of the Australasian Database Conference (ADC) — Databases Theory and Applications.* 2018, pp. 3–14.

A.4. Reuse of Material Published by IEEE

List of Figures

List of Tables

Acronyms

ANN	Artificial neural network
DBA	Database administrator
DBMS	Database management system
ERP	Enterprise resource planning
FD	Functional dependency
GUFLP	Generalized Uncapacitated Facility Location Problem
ILP	Integer linear programming
JOB	Join Order Benchmark
LQP	Logical query plan
LP	Linear programming
LSI	Latent semantic indexing
ML	Machine learning
OD	Order dependency
pp	Percentage points
PLI	Position list index
PQP	Physical query plan
RL	Reinforcement learning
TCO	Total cost of ownership
TPC	Transaction Processing Performance Council
SaaS	Software-as-a-Service
UCC	Unique column combination

Bibliography

[Aba+19] Daniel Abadi, Anastasia Ailamaki, David G. Andersen, Peter Bailis, Magdalena Balazinska, Philip A. Bernstein, Peter A. Boncz, Surajit Chaudhuri, Alvin Cheung, AnHai Doan, Luna Dong, Michael J. Franklin, Juliana Freire, Alon Y. Halevy, Joseph M. Hellerstein, Stratos Idreos, Donald Kossmann, Tim Kraska, Sailesh Krishnamurthy, Volker Markl, Sergey Melnik, Tova Milo, C. Mohan, Thomas Neumann, Beng Chin Ooi, Fatma Ozcan, Jignesh Patel, Andrew Pavlo, Raluca A. Popa, Raghu Ramakrishnan, Christopher Ré, Michael Stonebraker, and Dan Suciu. “The Seattle Report on Database Research”. In: *SIGMOD Record* 48.4 (2019), pp. 44–53.

[AMF06] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. “Integrating compression and execution in column-oriented database systems”. In: *Proceedings of the International Conference on Management of Data (SIGMOD).* 2006, pp. 671–682.

[AMH08] Daniel J. Abadi, Samuel Madden, and Nabil Hachem. “Column-stores vs. row-stores: how different are they really?” In: *Proceedings of the International Conference on Management of Data (SIGMOD).* 2008, pp. 967–980.

[AGN15] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. “Profiling relational data: a survey”. In: *VLDB Journal* 24.4 (2015), pp. 557–581.

[Abe+18] Ziawasch Abedjan, Lukasz Golab, Felix Naumann, and Thorsten Papenbrock. *Data Profiling.* Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2018.

[AQN14] Ziawasch Abedjan, Jorge-Arnulfo Quiané-Ruiz, and Felix Naumann. “Detecting Unique Column Combinations on Dynamic Data”. In: *Proceedings of the International Conference on Data Engineering (ICDE).* 2014, pp. 1036–1047.

[AR18] Alberto Abelló and Oscar Romero. “Online Analytical Processing”. In: *Encyclopedia of Database Systems, Second Edition.* Springer, 2018.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[Agr+04] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollár, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. “Database Tuning Advisor for Microsoft SQL Server 2005”. In: *Proceedings of the International Conference on Very Large Databases (VLDB).* 2004, pp. 1110–1121.

[ACN00] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. “Automated Selection of Materialized Views and Indexes in SQL Databases”. In: *Proceedings of the International Conference on Very Large Databases (VLDB).* 2000, pp. 496–505.

[ANY04] Sanjay Agrawal, Vivek R. Narasayya, and Beverly Yang. “Integrating Vertical and Horizontal Partitioning Into Automated Physical Database Design”. In: *Proceedings of the International Conference on Management of Data (SIGMOD).* 2004, pp. 359–370.

[Ake+17] 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 International Conference on Management of Data (SIGMOD).* 2017, pp. 1009–1024.

[Ake+21] Dana Van Aken, Dongsheng Yang, Sebastien Brillard, Ari Fiorino, Bohan Zhang, Christian Billian, and Andrew Pavlo. “An Inquiry into Machine Learning-based Automatic Configuration Tuning Services on Real-World Database Management Systems”. In: *Proceedings of the VLDB Endowment* 14.7 (2021), pp. 1241–1253.

[Ana] Firebolt Analytics. *The Firebolt Cloud Data Warehouse Whitepaper.* n.d. URL: `https://www.firebolt.io/resources/firebolt-cloud-data-warehouse-whitepaper#Indexes` Accessed: April 9, 2022.

[Arm74] William Ward Armstrong. “Dependency Structures of Data Base Relationships”. In: *Information Processing, Proceedings of the IFIP Congress.* 1974, pp. 580–583.

[AM86] Paolo Atzeni and Nicola M. Morfuni. “Functional dependencies and constraints on null values in database relations”. In: *Information and Control* 70.1 (1986), pp. 1–31.

[Aul+09] Stefan Aulbach, Dean Jacobs, Alfons Kemper, and Michael Seibold. “A comparison of flexible schemas for software as a service”. In: *Proceedings of the International Conference on Management of Data (SIGMOD).* 2009, pp. 881–888.

[BW01] Renaud Bassée and Jef Wijsen. “Neighborhood Dependencies for Prediction”. In: *Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*. 2001, pp. 562–567.

[Bas+15] Debabrota Basu, Qian Lin, Weidong Chen, Hoang Tam Vo, Zihong Yuan, Pierre Senellart, and Stéphane Bressan. “Cost-Model Oblivious Database Tuning with Reinforcement Learning”. In: *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA)*. 2015, pp. 253–268.

[BV84] Catriel Beeri and Moshe Y. Vardi. “A Proof Procedure for Data Dependencies”. In: *Journal of the ACM* 31.4 (1984), pp. 718–741.

[BG83] Philip A. Bernstein and Nathan Goodman. “Multiversion Concurrency Control — Theory and Algorithms”. In: *ACM Transactions on Database Systems (TODS)* 8.4 (1983), pp. 465–483.

[Ber+81] Philip A. Bernstein, Nathan Goodman, Eugene Wong, Christopher L. Reeve, and James B. Rothnie Jr. “Query Processing in a System for Distributed Databases (SDD-1)”. In: *ACM Transactions on Database Systems (TODS)* 6.4 (1981), pp. 602–625.

[Ber11] Leopoldo E. Bertossi. *Database Repairing and Consistent Query Answering.* Morgan & Claypool Publishers, 2011.

[Bir+20] Johann Birnick, Thomas Bläsius, Tobias Friedrich, Felix Naumann, Thorsten Papenbrock, and Martin Schirneck. “Hitting Set Enumeration with Partial Information for Unique Column Combination Discovery”. In: *Proceedings of the VLDB Endowment* 13.11 (2020), pp. 2270–2283.

[BFS17] Thomas Bläsius, Tobias Friedrich, and Martin Schirneck. “The Parameterized Complexity of Dependency Detection in Relational Databases”. In: *Proceedings of the International Symposium on Parameterized and Exact Computation (IPEC)*. 2017, 6:1–6:13.

[Boh+07] Philip Bohannon, Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. “Conditional Functional Dependencies for Data Cleaning”. In: *Proceedings of the International Conference on Data Engineering (ICDE)*. 2007, pp. 746–755.

[Boi22] Martin Boissier. “Robust and Budget-Constrained Encoding Configurations for In-Memory Database Systems”. In: *Proceedings of the VLDB Endowment* 15.4 (2022), pp. 780–793.

[BJ19] Martin Boissier and Max Jendruk. “Workload-Driven and Robust Selection of Compression Schemes for Column Stores”. In: *Proceedings of the International Conference on Extending Database Technology (EDBT).* 2019, pp. 674–677.

[BSU18] Martin Boissier, Rainer Schlosser, and Matthias Uflacker. “Hybrid Data Layouts for Tiered HTAP Databases with Pareto-Optimal Data Placements”. In: *Proceedings of the International Conference on Data Engineering (ICDE).* 2018, pp. 209–220.

[BAK17] Peter A. Boncz, Angelos-Christos G. Anadiotis, and Steffen Kläbe. “JCC-H: Adding Join Crossing Correlations with Skew to TPC-H”. In: *Performance Evaluation and Benchmarking for the Analytics Era — TPC Technology Conference (TPCTC), Revised Selected Papers.* 2017, pp. 103–119.

[BNE13] Peter A. Boncz, Thomas Neumann, and Orri Erling. “TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark”. In: *Performance Characterization and Benchmarking — TPC Technology Conference (TPCTC).* 2013, pp. 61–76.

[BS18] Philippe Bonnet and Dennis E. Shasha. “Index Tuning”. In: *Encyclopedia of Database Systems, Second Edition.* Springer, 2018.

[BAA12] Renata Borovica, Ioannis Alagiannis, and Anastasia Ailamaki. “Automated physical designers: what you see is (not) what you get”. In: *Proceedings of the International Workshop on Testing Database Systems (DBTEST).* 2012.

[Bre+21] Michael Brendle, Nick Weber, Mahammad Valiyev, Norman May, Robert Schulze, Alexander Böhm, Guido Moerkotte, and Michael Grossniklaus. “Precise, Compact, and Fast Data Access Counters for Automated Physical Database Design”. In: *Proceedings of the Conference Datenbanksysteme in Business, Technologie und Web (BTW).* 2021, pp. 79–100.

[Bro+16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. “OpenAI Gym”. In: *CoRR* abs/1606.01540 (2016). arXiv: `1606.01540`.

[BH03] Paul Brown and Peter J. Haas. “BHUNT: Automatic Discovery of Fuzzy Algebraic Constraints in Relational Data”. In: *Proceedings of the International Conference on Very Large Databases (VLDB).* 2003, pp. 668–679.

[BC05] Nicolas Bruno and Surajit Chaudhuri. “Automatic Physical Database Tuning: A Relaxation-based Approach”. In: *Proceedings of the International Conference on Management of Data (SIGMOD).* 2005, pp. 227–238.

[BC07] Nicolas Bruno and Surajit Chaudhuri. "An Online Approach to Physical Design Tuning". In: *Proceedings of the International Conference on Data Engineering (ICDE).* 2007, pp. 826–835.

[BN08] Nicolas Bruno and Rimma V. Nehme. "Configuration-parametric query optimization for physical design tuning". In: *Proceedings of the International Conference on Management of Data (SIGMOD).* 2008, pp. 941–952.

[CP18] José Camacho-Collados and Mohammad Taher Pilehvar. "From Word To Sense Embeddings: A Survey on Vector Representations of Meaning". In: *Journal of Artificial Intelligence Research* 63 (2018), pp. 743–788.

[CFM95] Alberto Caprara, Matteo Fischetti, and Dario Maio. "Exact and Approximate Algorithms for the Index Selection Problem in Physical Database Design". In: *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 7.6 (1995), pp. 955–967.

[CS96] Alberto Caprara and Juan José Salazar González. "A Branch-and-Cut Algorithm for a Generalization of the Uncapacitated Facility Location Problem". In: *TOP: An Official Journal of the Spanish Society of Statistics and Operations Research* 4 (1996), pp. 135–163.

[Car+19] Loredana Caruccio, Stefano Cirillo, Vincenzo Deufemia, and Giuseppe Polese. "Incremental Discovery of Functional Dependencies with a Bit-vector Algorithm". In: *Proceedings of the Italian Symposium on Advanced Database Systems.* 2019.

[CDP16] Loredana Caruccio, Vincenzo Deufemia, and Giuseppe Polese. "Relaxed Functional Dependencies — A Survey of Approaches". In: *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 28.1 (2016), pp. 147–165.

[CFP82] Marco A. Casanova, Ronald Fagin, and Christos H. Papadimitriou. "Inclusion Dependencies and Their Interaction with Functional Dependencies". In: *Proceedings of the Symposium on Principles of Database Systems (PODS).* 1982, pp. 171–176.

[Cen+21] Lujing Cen, Andreas Kipf, Ryan Marcus, and Tim Kraska. "LEA: A Learned Encoding Advisor for Column Stores". In: *Proceedings of the Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM).* 2021, pp. 32–35.

[CGM90] Upen S. Chakravarthy, John Grant, and Jack Minker. "Logic-Based Approach to Semantic Query Optimization". In: *ACM Transactions on Database Systems (TODS)* 15.2 (1990), pp. 162–207.

[CGN02] Surajit Chaudhuri, Ashish Kumar Gupta, and Vivek R. Narasayya. “Compressing SQL workloads”. In: *Proceedings of the International Conference on Management of Data (SIGMOD).* 2002, pp. 488–499.

[CN20] Surajit Chaudhuri and Vivek Narasayya. *Anytime Algorithm of Database Tuning Advisor for Microsoft SQL Server.* June 2020. URL: `https://www.microsoft.com/en-us/research/publication/anytime-algorithm-of-database-tuning-advisor-for-microsoft-sql-server` Accessed: April 9, 2022.

[CN97] Surajit Chaudhuri and Vivek R. Narasayya. “An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server”. In: *Proceedings of the International Conference on Very Large Databases (VLDB).* 1997, pp. 146–155.

[CN98] Surajit Chaudhuri and Vivek R. Narasayya. “AutoAdmin ’What-if’ Index Analysis Utility”. In: *Proceedings of the International Conference on Management of Data (SIGMOD).* 1998, pp. 367–378.

[CN99] Surajit Chaudhuri and Vivek R. Narasayya. “Index Merging”. In: *Proceedings of the International Conference on Data Engineering (ICDE).* 1999, pp. 296–303.

[CN07] Surajit Chaudhuri and Vivek R. Narasayya. “Self-Tuning Database Systems: A Decade of Progress”. In: *Proceedings of the International Conference on Very Large Databases (VLDB).* 2007, pp. 3–14.

[CS94] Surajit Chaudhuri and Kyuseok Shim. “Including Group-By in Query Optimization”. In: *Proceedings of the International Conference on Very Large Databases (VLDB).* 1994, pp. 354–366.

[CW18] Surajit Chaudhuri and Gerhard Weikum. “Self-Management Technology in Databases”. In: *Encyclopedia of Database Systems, Second Edition.* Springer, 2018.

[Che+08] Su Chen, Beng Chin Ooi, Kian-Lee Tan, and Mario A. Nascimento. “ST^{2}B-tree: a self-tunable spatio-temporal b$^+$-tree index for moving objects”. In: *Proceedings of the International Conference on Management of Data (SIGMOD).* 2008, pp. 29–42.

[Che+99] Qi Cheng, Jarek Gryz, Fred Koo, T. Y. Cliff Leung, Linqi Liu, Xiaoyan Qian, and K. Bernhard Schiefer. “Implementation of Two Semantic Query Optimization Techniques in DB2 Universal Database”. In: *Proceedings of the International Conference on Very Large Databases (VLDB).* 1999, pp. 687–698.

[CGR01] Paolo Ciaccia, Matteo Golfarelli, and Stefano Rizzi. “On Estimating the Cardinality of Aggregate Views”. In: *Proceedings of the International Workshop on Design and Management of Data Warehouses.* 2001, pp. 12.1–12.10.

[Cod70] Edgar F. Codd. “A Relational Model of Data for Large Shared Data Banks”. In: *Communications of the ACM* 13.6 (1970), pp. 377–387.

[Cod71] Edgar F. Codd. “Further Normalization of the Data Base Relational Model”. In: *IBM Research Report, San Jose, California* RJ909 (1971).

[Cod75] Edgar F. Codd. “Understanding Relations (Installment #7)”. In: *FDT — Bulletin of ACM-SIGMOD* 7.3 (1975), pp. 23–28.

[Cod90] Edgar F. Codd. *The Relational Model for Database Management, Version 2.* Addison-Wesley, 1990.

[Coo+10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. “Benchmarking cloud serving systems with YCSB”. In: *Proceedings of the ACM Symposium on Cloud Computing (SoCC).* 2010, pp. 143–154.

[Cro21] Andrew Crotty. “Hist-Tree: Those Who Ignore It Are Doomed to Learn”. In: *Proceedings of the Conference on Innovative Data Systems Research (CIDR).* 2021.

[Dag+16] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. “The Snowflake Elastic Data Warehouse”. In: *Proceedings of the International Conference on Management of Data (SIGMOD).* 2016, pp. 215–226.

[Das+19] Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic, Vivek R. Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, and Surajit Chaudhuri. “Automatically Indexing Millions of Databases in Microsoft Azure SQL Database”. In: *Proceedings of the International Conference on Management of Data (SIGMOD).* 2019, pp. 666–679.

[Das+16] Sudipto Das, Feng Li, Vivek R. Narasayya, and Arnd Christian König. “Automated Demand-driven Resource Scaling in Relational Database-as-a-Service”. In: *Proceedings of the International Conference on Management of Data (SIGMOD).* 2016, pp. 1923–1934.

[DPA11] Debabrata Dash, Neoklis Polyzotis, and Anastasia Ailamaki. “CoPhy: A Scalable, Portable, and Interactive Index Advisor for Large Workloads”. In: *Proceedings of the VLDB Endowment* 4.6 (2011), pp. 362–372.

[DD92] C. J. Date and Hugh Darwen. In: *Relational Database Writings 1989-1991.* Addison-Wesley, 1992. Chap. The Role of functional Dependence in Query Decomposition, pp. 133–150.

[Day87] Umeshwar Dayal. “Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers”. In: *Proceedings of the International Conference on Very Large Databases (VLDB).* 1987, pp. 197–208.

[Dee+20] Shaleen Deep, Anja Gruenheid, Paraschos Koutris, Jeffrey F. Naughton, and Stratis Viglas. “Comprehensive and Efficient Workload Compression”. In: *Proceedings of the VLDB Endowment* 14.3 (2020), pp. 418–430.

[Dee+90] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. “Indexing by Latent Semantic Analysis”. In: *Journal of the American Society for Information Science* 41.6 (1990), pp. 391–407.

[DNR08] Alin Deutsch, Alan Nash, and Jeffrey B. Remmel. “The chase revisited”. In: *Proceedings of the Symposium on Principles of Database Systems (PODS).* 2008, pp. 149–158.

[DPT99] Alin Deutsch, Lucian Popa, and Val Tannen. “Physical Data Independence, Constraints, and Optimization with Universal Plans”. In: *Proceedings of the International Conference on Very Large Databases (VLDB).* 1999, pp. 459–470.

[DPT06] Alin Deutsch, Lucian Popa, and Val Tannen. “Query reformulation with constraints”. In: *SIGMOD Record* 35.1 (2006), pp. 65–73.

[Din+19] 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 International Conference on Management of Data (SIGMOD).* 2019, pp. 1241–1258.

[Din+20] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. “Tsunami: A Learned Multi-dimensional Index for Correlated Data and Skewed Workloads”. In: *Proceedings of the VLDB Endowment* 14.2 (2020), pp. 74–86.

[DH82] Jirun Dong and Richard Hull. “Applying Approximate Order Dependency to Reduce Indexing Space”. In: *Proceedings of the International Conference on Management of Data (SIGMOD).* 1982, pp. 119–127.

[DFR98] Rodney G. Downey, Michael R. Fellows, and Kenneth W. Regan. “Parameterized Circuit Complexity and the W Hierarchy”. In: *Theoretical Computer Science* 191.1-2 (1998), pp. 97–115.

[Dre21] Markus Dreseler. “Automatic Tiering for In-Memory Database Systems”. PhD thesis. Hasso Plattner Institute, University of Potsdam, Germany, 2021.

[Dre+20] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. “Quantifying TPC-H Choke Points and Their Optimizations”. In: *Proceedings of the VLDB Endowment* 13.8 (2020), pp. 1206–1220.

[Dre+18a] Markus Dreseler, Timo Gasda, Jan Kossmann, Matthias Uflacker, and Hasso Plattner. “Adaptive Access Path Selection for Hardware-Accelerated DRAM Loads”. In: *Proceedings of the Australasian Database Conference (ADC) — Databases Theory and Applications.* 2018, pp. 3–14.

[Dre+19] Markus Dreseler, Jan Kossmann, Martin Boissier, Stefan Klauck, Matthias Uflacker, and Hasso Plattner. “Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management”. In: *Proceedings of the International Conference on Extending Database Technology (EDBT).* 2019, pp. 313–324.

[Dre+18b] Markus Dreseler, Jan Kossmann, Johannes Frohnhofen, Matthias Uflacker, and Hasso Plattner. “Fused Table Scans: Combining AVX-512 and JIT to Double the Performance of Multi-Predicate Scans”. In: *Proceedings of the International Conference on Data Engineering (ICDE) Workshops.* 2018, pp. 102–109.

[DWH19] Bingqian Du, Chuan Wu, and Zhiyi Huang. “Learning Resource Allocation and Pricing for Cloud Profit Maximization”. In: *Proceedings of the Conference on Artificial Intelligence (AAAI).* 2019, pp. 7570–7577.

[DTB09] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. “Tuning Database Configuration Parameters with iTuned”. In: *Proceedings of the VLDB Endowment* 2.1 (2009), pp. 1246–1257.

[Dub] Duboce Labs. *pganalyze — Postgres Performance Optimization.* n.d. URL: `https://pganalyze.com/postgres-performance-optimization` Accessed: April 9, 2022.

[Dür+19] Falco Dürsch, Axel Stebner, Fabian Windheuser, Maxi Fischer, Tim Friedrich, Nils Strelow, Tobias Bleifuß, Hazar Harmouch, Lan Jiang, Thorsten Papenbrock, and Felix Naumann. “Inclusion Dependency Discovery: An Experimental Evaluation of Thirteen Algorithms”. In: *Proceedings of the International Conference on Information and Knowledge Management (CIKM).* 2019, pp. 219–228.

[EFM16] Marius Eich, Pit Fender, and Guido Moerkotte. “Faster Plan Generation through Consideration of Functional Dependencies and Keys”. In: *Proceedings of the VLDB Endowment* 9.10 (2016), pp. 756–767.

[EKM04] Eyal Even-Dar, Sham M. Kakade, and Yishay Mansour. “Experts in a Markov Decision Process”. In: *Advances in Neural Information Processing Systems (NIPS).* 2004, pp. 401–408.

[Fag77] Ronald Fagin. “Multivalued Dependencies and a New Normal Form for Relational Databases”. In: *ACM Transactions on Database Systems (TODS)* 2.3 (1977), pp. 262–278.

[FKP05] Ronald Fagin, Phokion G. Kolaitis, and Lucian Popa. “Data Exchange: Getting to the Core”. In: *ACM Transactions on Database Systems (TODS)* 30.1 (2005), pp. 174–210.

[Fau+16] Martin Faust, Martin Boissier, Marvin Keller, David Schwalb, Holger Bischoff, Katrin Eisenreich, Franz Färber, and Hasso Plattner. “Footprint Reduction and Uniqueness Enforcement with Hash Indices in SAP HANA”. In: *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA).* 2016, pp. 137–151.

[FV20] Paolo Ferragina and Giorgio Vinciguerra. “The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds”. In: *Proceedings of the VLDB Endowment* 13.8 (2020), pp. 1162–1175.

[Fil+20] Reiner H. Santos Filho, Tadeu N. Ferreira, Diogo M. F. Mattos, and Dianne S. V. Medeiros. “A Lightweight Reinforcement-Learning-Based Mechanism for Bandwidth Provisioning on Multitenant Data Center”. In: *Proceedings of the International Conference on Systems, Signals and Image Processing (IWSSIP).* 2020, pp. 331–336.

[FST88] Sheldon J. Finkelstein, Mario Schkolnick, and Paolo Tiberio. “Physical Database Design for Relational Databases”. In: *ACM Transactions on Database Systems (TODS)* 13.1 (1988), pp. 91–128.

[FS99] Peter A Flach and Iztok Savnik. “Database dependency discovery: a machine learning approach”. In: *AI Communications* 12.3 (1999), pp. 139–160.

[FG89] Farshad Fotouhi and Carlos E. Galarce. “Genetic Algorithms and the Search for Optimal Database Index Selection”. In: *Proceedings of the Great Lakes Computer Science Conference*. 1989, pp. 249–255.

[FGK03] R. Fourer, D.M. Gay, and B.W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Thomson/Brooks/Cole, 2003.

[FON92] Martin R. Frank, Edward Omiecinski, and Shamkant B. Navathe. “Adaptive and Automated Index Selection in RDBMS”. In: *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 1992, pp. 277–292.

[GHK92] Sumit Ganguly, Waqar Hasan, and Ravi Krishnamurthy. “Query Optimization for Parallel Execution”. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1992, pp. 9–18.

[GW87] Richard A. Ganski and Harry K. T. Wong. “Optimization of Nested SQL Queries Revisited”. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1987, pp. 23–33.

[Gar20] Gartner. *Magic Quadrant for Cloud Database Management Systems*. By Donald Feinberg, Merv Adrian, Rick Greenwald, Adam Ronthal, Henry Cook. November 2020. URL: `https://www.gartner.com/en/documents/3993398` Accessed: April 9, 2022.

[GG82] Erol Gelenbe and Danièle Gardy. “The Size of Projections of Relations Satisfying a Functional Dependency”. In: *Proceedings of the International Conference on Very Large Databases (VLDB)*. 1982, pp. 325–333.

[Gia+02] Chris Giannella, Mehmet M. Dalkilic, Dennis P. Groth, and Edward L. Robertson. “Improving Query Evaluation with Approximate Functional Dependency Based Decompositions”. In: *Proceedings of British National Conference on Databases (BNCOD)*. 2002, pp. 26–41.

[GH83] Seymour Ginsburg and Richard Hull. “Order Dependency in the Relational Model”. In: *Theoretical Computer Science* 26 (1983), pp. 149–195.

[GH86] Seymour Ginsburg and Richard Hull. “Sort sets in the relational model”. In: *Journal of the ACM* 33.3 (1986), pp. 465–488.

[Gol+09] Lukasz Golab, Howard Karloff, Flip Korn, Avishek Saha, and Divesh Srivastava. “Sequential Dependencies”. In: *Proceedings of the VLDB Endowment* 2.1 (2009), pp. 574–585.

[GBC16] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. *Deep Learning*. Adaptive computation and machine learning. MIT Press, 2016.

[Gra93] Goetz Graefe. “Query Evaluation Techniques for Large Databases”. In: *ACM Computing Surveys* 25.2 (1993), pp. 73–170.

[Gra95] Goetz Graefe. “The Cascades Framework for Query Optimization”. In: *IEEE Data Engineering Bulletin* 18.3 (1995), pp. 19–29.

[Gra06] Goetz Graefe. “B-tree indexes for high update rates”. In: *SIGMOD Record* 35.1 (2006), pp. 39–44.

[Gra11] Goetz Graefe. “Modern B-Tree Techniques”. In: *Foundations and Trends in Databases* 3.4 (2011), pp. 203–402.

[Gry98] Jarek Gryz. “Query Folding with Inclusion Dependencies”. In: *Proceedings of the International Conference on Data Engineering (ICDE).* 1998, pp. 126–133.

[Gun+19] David Gunning, Mark Stefik, Jaesik Choi, Timothy Miller, Simone Stumpf, and Guang-Zhong Yang. “XAI — Explainable artificial intelligence”. In: *Science Robotics* 4.37 (2019).

[HS19] Stefan Halfpap and Rainer Schlosser. “Workload-Driven Fragment Allocation for Partially Replicated Databases Using Linear Programming”. In: *Proceedings of the International Conference on Data Engineering (ICDE).* 2019, pp. 1746–1749.

[HS20] Stefan Halfpap and Rainer Schlosser. “Exploration of Dynamic Query-Based Load Balancing for Partially Replicated Database Systems with Node Failures”. In: *Proceedings of the International Conference on Information and Knowledge Management (CIKM).* 2020, pp. 3409–3412.

[HC76] Michael Hammer and Arvola Chan. “Index Selection in a Self-Adaptive Data Base Management System”. In: *Proceedings of the International Conference on Management of Data (SIGMOD).* 1976, pp. 1–8.

[HZ80] Michael Hammer and Stanley B. Zdonik. “Knowledge-Based Query Processing”. In: *Proceedings of the International Conference on Very Large Databases (VLDB).* 1980, pp. 137–147.

[Har54] Zellig S. Harris. “Distributional Structure”. In: *WORD* 10.2-3 (1954), pp. 146–162.

[Hei+13] Arvid Heise, Jorge-Arnulfo Quiané-Ruiz, Ziawasch Abedjan, Anja Jentzsch, and Felix Naumann. “Scalable Discovery of Unique Column Combinations”. In: *Proceedings of the VLDB Endowment* 7.4 (2013), pp. 301–312.

[HSH07] Joseph M. Hellerstein, Michael Stonebraker, and James R. Hamilton. "Architecture of a Database System". In: *Foundations and Trends in Databases* 1.2 (2007), pp. 141–259.

[Hig+20] Antony S. Higginson, Mihaela Dediu, Octavian Arsene, Norman W. Paton, and Suzanne M. Embury. "Database Workload Capacity Planning using Time Series Analysis and Machine Learning". In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2020, pp. 769–783.

[Hil+18] 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 Wu. *Stable Baselines*. August 2018. URL: `https://github.com/hill-a/stable-baselines` Accessed: April 9, 2022.

[HB22] Benjamin Hilprecht and Carsten Binnig. "One Model to Rule them All: Towards Zero-Shot Learning for Databases". In: *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*. 2022.

[Hil+20] Benjamin Hilprecht, Carsten Binnig, Tiemo Bang, Muhammad El-Hindi, Benjamin Hättasch, Aditya Khanna, Robin Rehrmann, Uwe Röhm, Andreas Schmidt, Lasse Thostrup, and Tobias Ziegler. "DBMS Fitting: Why should we learn what we already know?" In: *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*. 2020.

[HBR20] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. "Learning a Partitioning Advisor for Cloud Databases". In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2020, pp. 143–157.

[HO20] Shengyi Huang and Santiago Ontañón. "A Closer Look at Invalid Action Masking in Policy Gradient Algorithms". In: *CoRR* abs/2006.14171 (2020). arXiv: `2006.14171`.

[Huh+98] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. "Efficient Discovery of Functional and Approximate Dependencies Using Partitions". In: *Proceedings of the International Conference on Data Engineering (ICDE)*. 1998, pp. 392–401.

[Huh+99] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. "TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies". In: *Computer Journal* 42.2 (1999), pp. 100–111.

[IBM22a] IBM. *Db2 for z/OS — DSN_VIRTUAL_INDEXES*. March 2022. URL: `https://www.ibm.com/docs/en/db2-for-zos/12?topic=tables-dsn-virtual-indexes` Accessed: April 9, 2022.

[IBM22b] IBM. *Referential integrity constraints help reduce the number of statistical views*. February 2022. URL: `https://www.ibm.com/support/knowledgecenter/SSEPGG_11.5.0/com.ibm.db2.luw.admin.perf.doc/doc/c0059081.html` Accessed: April 9, 2022.

[IKM07] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. "Database Cracking". In: *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*. 2007, pp. 68–78.

[Idr+11] Stratos Idreos, Stefan Manegold, Harumi A. Kuno, and Goetz Graefe. "Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores". In: *Proceedings of the VLDB Endowment* 4.9 (2011), pp. 585–597.

[Ile+14] Ioana Ileana, Bogdan Cautis, Alin Deutsch, and Yannis Katsis. "Complete yet practical search for minimal query reformulations under constraints". In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2014, pp. 1015–1026.

[Ily+04] Ihab F. Ilyas, Volker Markl, Peter J. Haas, Paul Brown, and Ashraf Aboulnaga. "CORDS: Automatic Discovery of Correlations and Soft Functional Dependencies". In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2004, pp. 647–658.

[Inc19] InfluxData Inc. *Stream Processing with InfluxDB — An Introduction*. September 2019. URL: `https://www.influxdata.com/resources/stream-processing-with-influxdb/` Accessed: April 9, 2022.

[Ioa96] Yannis E. Ioannidis. "Query Optimization". In: *ACM Computing Surveys* 28.1 (1996), pp. 121–123.

[ISR83] Maggie Y. L. Ip, Lawrence V. Saxton, and Vijay V. Raghavan. "On the Selection of an Optimal Set of Indexes". In: *IEEE Transactions on Software Engineering* 9.2 (1983), pp. 135–143.

[Int99] International Organization for Standardization: *ISO/IEC 9075-2:1999 (SQL Standard 1999)*. Standard. December 1999.

[Int92] International Organization for Standardization: *ISO/IEC 9075:1992 (SQL Standard 1992)*. Standard. July 1992.

[Jac05] Jack Klebanoff. *Apache Derby — Intersect & Except Design.* February 2005. URL: `https://db.apache.org/derby/papers/Intersect-design.html` Accessed: April 9, 2022.

[JH18] Shrainik Jain and Bill Howe. “Query2Vec: NLP Meets Databases for Generalized Workload Analytics”. In: *CoRR* abs/1801.05613 (2018). arXiv: `1801.05613`.

[JK84a] Matthias Jarke and Jürgen Koch. “Query Optimization in Database Systems”. In: *ACM Computing Surveys* 16.2 (1984), pp. 111–152.

[JK84b] David S. Johnson and Anthony C. Klug. “Testing Containment of Conjunctive Queries under Functional and Inclusion Dependencies”. In: *Journal of Computer and System Sciences* 28.1 (1984), pp. 167–189.

[KY83] Yahiko Kambayashi and Masatoshi Yoshikawa. “Query Processing Utilizing Dependencies and Horizontal Decomposition”. In: *Proceedings of the International Conference on Management of Data (SIGMOD).* 1983, pp. 55–67.

[Kan17a] Andrew Kane. *Dexter — The automatic indexer for Postgres.* June 2017. URL: `https://github.com/ankane/dexter` Accessed: April 9, 2022.

[Kan17b] Andrew Kane. *Introducing Dexter, the Automatic Indexer for Postgres.* June 2017. URL: `https://medium.com/@ankane/introducing-dexter-the-automatic-indexer-for-postgres-5f8fa8b28f27` Accessed: April 9, 2022.

[KN11] Alfons Kemper and Thomas Neumann. “HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots”. In: *Proceedings of the International Conference on Data Engineering (ICDE).* 2011, pp. 195–206.

[Kim82] Won Kim. “On Optimizing an SQL-like Nested Query”. In: *ACM Transactions on Database Systems (TODS)* 7.3 (1982), pp. 443–469.

[KR13] Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling, 3rd Edition.* Wiley, 2013.

[Kim+09] Hideaki Kimura, George Huo, Alexander Rasin, Samuel Madden, and Stanley B. Zdonik. “Correlation Maps: A Compressed Access Method for Exploiting Soft Functional Dependencies”. In: *Proceedings of the VLDB Endowment* 2.1 (2009), pp. 1222–1233.

[Kin80] Jonathan J. King. “Modelling Concepts for Reasoning About Access to Knowledge”. In: *Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modelling.* 1980, pp. 138–140.

[Kip+19] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. “Learned Cardinalities: Estimating Correlated Joins with Deep Learning”. In: *Proceedings of the Conference on Innovative Data Systems Research (CIDR).* 2019.

[Kip+20] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. “RadixSpline: a single-pass learned index”. In: *Proceedings of the Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM).* 2020, 5:1–5:5.

[KBS20] Steffen Klaebe, Stephan Baumann, and Kai-Uwe Sattler. “PatchIndex — Exploiting approximate constraints in self-managing databases”. In: *Proceedings of the International Conference on Data Engineering (ICDE) Workshops.* 2020, pp. 139–146.

[KL16] Henning Köhler and Sebastian Link. “SQL Schema Design: Foundations, Normal Forms, and Normalization”. In: *Proceedings of the International Conference on Management of Data (SIGMOD).* 2016, pp. 267–279.

[KLZ15] Henning Köhler, Sebastian Link, and Xiaofang Zhou. “Possible and Certain SQL Keys”. In: *Proceedings of the VLDB Endowment* 8.11 (2015), pp. 1118–1129.

[Kos18] Jan Kossmann. “Self-Driving: From General Purpose to Specialized DBMSs”. In: *Proceedings of the VLDB PhD Workshop.* 2018.

[Kos+21] Jan Kossmann, Martin Boissier, Alexander Dubrawski, Fabian Heseding, Caterina Mandel, Udo Pigorsch, Max Schneider, Til Schniese, Mona Sobhani, Petr Tsayun, Katharina Wille, Michael Perscheid, Matthias Uflacker, and Hasso Plattner. “A Cockpit for the Development and Evaluation of Autonomous Database Systems”. In: *Proceedings of the International Conference on Data Engineering (ICDE).* 2021, pp. 2685–2688.

[Kos+18] Jan Kossmann, Markus Dreseler, Timo Gasda, Matthias Uflacker, and Hasso Plattner. “Visual Evaluation of SQL Plan Cache Algorithms”. In: *Proceedings of the Australasian Database Conference (ADC) — Databases Theory and Applications.* 2018, pp. 350–353.

[Kos+20a] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. *Cost estimations of actual and hypothetical indexes on GitHub.* 2020. URL: `https://git.io/CostEstimationAccuracyHypoPGIndexes` Accessed: April 9, 2022.

[Kos+20b] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. *Cost estimations of PostgreSQL and DBMS-X.* 2020. URL: `https://github.com/hyrise/index_selection_evaluation/tree/master/benchmark_results/cost_estimation_PostgreSQL_vs_DBMSX` Accessed: April 9, 2022.

[Kos+20c] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. *Extend's source code.* 2020. URL: `https://github.com/hyrise/index_selection_evaluation/blob/master/selection/algorithms/extend_algorithm.py` Accessed: April 9, 2022.

[Kos+20d] 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". In: *Proceedings of the VLDB Endowment* 13.11 (2020), pp. 2382–2395.

[KKS22a] Jan Kossmann, Alexander Kastius, and Rainer Schlosser. "SWIRL: Selection of Workload-aware Indexes using Reinforcement Learning". In: *Proceedings of the International Conference on Extending Database Technology (EDBT).* 2022, pp. 155–168.

[KKS22b] Jan Kossmann, Alexander Kastius, and Rainer Schlosser. *SWIRL's source code.* 2022. URL: `https://github.com/hyrise/rl_index_selection` Accessed: April 9, 2022.

[Kos+22a] Jan Kossmann, Felix Naumann, Daniel Lindner, and Thorsten Papenbrock. "Workload-driven, Lazy Discovery of Data Dependencies for Query Optimization". In: *Proceedings of the Conference on Innovative Data Systems Research (CIDR).* 2022.

[KPN22] Jan Kossmann, Thorsten Papenbrock, and Felix Naumann. "Data dependencies for query optimization: a survey". In: *VLDB Journal* 31.1 (2022), pp. 1–22.

[KS19] Jan Kossmann and Rainer Schlosser. "A Framework for Self-Managing Database Systems". In: *Proceedings of the International Conference on Data Engineering (ICDE) Workshops.* 2019, pp. 100–106.

[KS20] Jan Kossmann and Rainer Schlosser. “Self-driving database systems: a conceptual approach”. In: *Distributed And Parallel Databases (DAPD)* 38.4 (2020), pp. 795–817.

[Kos+22b] Jan Kossmann, Rainer Schlosser, Alexander Kastius, Michael Perscheid, and Hasso Plattner. *Training an Agent for Iterative Multi-Attribute Index Selection*. European Patent Application EP22156399.2. February 2022.

[Kra+19] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. “SageDB: A Learned Database System”. In: *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*. 2019.

[Kra+18] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. “The Case for Learned Index Structures”. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2018, pp. 489–504.

[Kra+21] Tim Kraska, Umar Farooq Minhas, Thomas Neumann, Olga Papaemmanouil, Jignesh M. Patel, Christopher Ré, and Michael Stonebraker. “ML-In-Databases: Assessment and Prognosis”. In: *IEEE Data Engineering Bulletin* 44.1 (2021), pp. 3–10.

[Kri+20] Ani Kristo, Kapil Vaidya, Ugur Çetintemel, Sanchit Misra, and Tim Kraska. “The Case for a Learned Sorting Algorithm”. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2020, pp. 1001–1016.

[KPN15] Sebastian Kruse, Thorsten Papenbrock, and Felix Naumann. “Scaling Out the Discovery of Inclusion Dependencies”. In: *Proceedings of the Conference Datenbanksysteme in Business, Technologie und Web Technik (BTW)*. 2015, pp. 445–454.

[LBP20] Hai Lan, Zhifeng Bao, and Yuwei Peng. “An Index Advisor Using Deep Reinforcement Learning”. In: *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*. 2020, pp. 2105–2108.

[Lan+16] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. “Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation”. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2016, pp. 311–326.

[LG09] Per-Ake Larson and Cesar A Galindo-Legaria. *Partial pre-aggregation in relational database queries*. US Patent 7,593,926. September 2009.

[Lei+15] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. “How Good Are Query Optimizers, Really?” In: *Proceedings of the VLDB Endowment* 9.3 (2015), pp. 204–215.

[Lei+18] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. “Query optimization through the looking glass, and what we found running the Join Order Benchmark”. In: *VLDB Journal* 27.5 (2018), pp. 643–668.

[LZC21] Guoliang Li, Xuanhe Zhou, and Lei Cao. “Machine Learning for Databases”. In: *Proceedings of the VLDB Endowment* 14.12 (2021), pp. 3190–3193.

[Li+19] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. “QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning”. In: *Proceedings of the VLDB Endowment* 12.12 (2019), pp. 2118–2130.

[Lic+20] Gabriel Paludo Licks, Júlia Mara Colleoni Couto, Priscilla de Fátima Miehe, Renata De Paris, Duncan Dubugras A. Ruiz, and Felipe Meneguzzi. “SmartIX: A database indexing agent based on reinforcement learning”. In: *Applied Intelligence* 50.8 (2020), pp. 2575–2588.

[Lig18] Sam Lightstone. “Physical Database Design for Relational Databases”. In: *Encyclopedia of Database Systems, Second Edition.* Springer, 2018.

[LTN07] Sam Lightstone, Toby J. Teorey, and Thomas P. Nadeau. *Physical Database Design: the database professional’s guide to exploiting indexes, views, storage, and more.* Morgan Kaufmann, 2007.

[Lin22] Daniel Lindner. “Workload-Driven Query Optimization Using Data Dependencies”. Master’s thesis. Hasso Plattner Institute, University of Potsdam, Germany, 2022.

[LLK21] Daniel Lindner, Alexander Löser, and Jan Kossmann. “Learned What-If Cost Models for Autonomous Clustering”. In: *Advances in Databases and Information Systems (ADBIS).* 2021, pp. 3–13.

[Liu+12] Jixue Liu, Jiuyong Li, Chengfei Liu, and Yongfeng Chen. “Discover Dependencies from Data — A Review”. In: *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 24.2 (2012), pp. 251–264.

[Liu+20] Qiyu Liu, Libin Zheng, Yanyan Shen, and Lei Chen. “Stable Learned Bloom Filters for Data Streams”. In: *Proceedings of the VLDB Endowment* 13.11 (2020), pp. 2355–2367.

[Lu+21] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. "APEX: A High-Performance Learned Index on Persistent Memory". In: *Proceedings of the VLDB Endowment* 15.3 (2021), pp. 597–610.

[Lu+19] Jiaheng Lu, Yuxing Chen, Herodotos Herodotou, and Shivnath Babu. "Speedup Your Analytics: Automatic Parameter Tuning for Databases and Big Data Systems". In: *Proceedings of the VLDB Endowment* 12.12 (2019), pp. 1970–1973.

[LO78] Claudio L. Lucchesi and Sylvia L. Osborn. "Candidate keys for relations". In: *Journal of Computer and System Sciences* 17.2 (1978), pp. 270–279.

[LL71] Vincent Y. Lum and Huei Ling. "An Optimization Problem on the Selection of Secondary Keys". In: *Proceedings of the ACM Annual Conference (ACM '71)*. 1971, pp. 349–356.

[Ma+18] 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 International Conference on Management of Data (SIGMOD)*. 2018, pp. 631–645.

[Ma+21] Lin Ma, William Zhang, Jie Jiao, Wuwen Wang, Matthew Butrovich, Wan Shen Lim, Prashanth Menon, and Andrew Pavlo. "MB2: Decomposed Behavior Modeling for Self-Driving Database Management Systems". In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2021, pp. 1248–1261.

[Man18] Stefan Manegold. "Cost Estimation". In: *Encyclopedia of Database Systems, Second Edition*. Springer, 2018.

[Mao+19] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. "Learning scheduling algorithms for data processing clusters". In: *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*. 2019, pp. 270–288.

[MLP09] Fabien De Marchi, Stéphane Lopes, and Jean-Marc Petit. "Unary and n-ary inclusion dependency discovery in relational databases". In: *Journal of Intelligent Information Systems* 32 (2009), pp. 53–73.

[Mar+19] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. "Neo: A Learned Query Optimizer". In: *Proceedings of the VLDB Endowment* 12.11 (2019), pp. 1705–1718.

[MP18] Ryan Marcus and Olga Papaemmanouil. “Deep Reinforcement Learning for Join Order Enumeration”. In: *Proceedings of the Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM)*. 2018, 3:1–3:4.

[MP19] Ryan Marcus and Olga Papaemmanouil. “Plan-Structured Deep Neural Network Models for Query Performance Prediction”. In: *Proceedings of the VLDB Endowment* 12.11 (2019), pp. 1733–1746.

[Mat96] G. B. Mathews. “On the Partition of Numbers”. In: *Proceedings of the London Mathematical Society* s1-28.1 (November 1896), pp. 486–490.

[Mei14] Michael Meier. “The backchase revisited”. In: *VLDB Journal* 23.3 (2014), pp. 495–516.

[MRB19] Puya Memarzia, Suprio Ray, and Virendra C. Bhavsar. “A Six-dimensional Analysis of In-memory Aggregation”. In: *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 2019, pp. 289–300.

[Met+19] Quentin Meteier et al. “Workshop on explainable AI in automated driving: a user-centered interaction approach”. In: *Adjunct Proceedings of AutomotiveUI*. 2019, pp. 32–37.

[Mic04] Microsoft. *Foreign Key Constraints (Without NOCHECK) Boost Performance and Data Integrity*. December 2004. URL: `https://web.archive.org/web/20101219111457/http://www.microsoft.com/technet/abouttn/flash/tips/tips_122104.mspx` Accessed: April 9, 2022.

[Mic08] Microsoft. *Optimizing Queries That Access Correlated datetime Columns*. December 2008. URL: `https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2005/ms177416(v=sql.90)?redirectedfrom=MSDN` Accessed: April 9, 2022.

[Mic21a] Microsoft. *SQL Server 2019 — Database Engine Tuning Advisor*. December 2021. URL: `https://docs.microsoft.com/en-us/sql/relational-databases/performance/database-engine-tuning-advisor?view=sql-server-ver15` Accessed: April 9, 2022.

[Mic21b] Microsoft. *SQL Server 2019 — Unique Constraints and Check Constraints*. May 2021. URL: `https://docs.microsoft.com/en-us/sql/relational-databases/tables/unique-constraints-and-check-constraints?view=sql-server-ver15` Accessed: April 9, 2022.

[Mic22a] Microsoft. *Automatic tuning in Azure SQL Database and Azure SQL Managed Instance.* January 2022. URL: `https://docs.microsoft.com/en-us/azure/azure-sql/database/automatic-tuning-overview` Accessed: April 9, 2022.

[Mic22b] Microsoft. *SQL Server Linux 2019 — CREATE INDEX (Transact-SQL).* January 2022. URL: `https://docs.microsoft.com/en-us/sql/t-sql/statements/create-index-transact-sql?view=sql-server-linux-ver15` Accessed: April 9, 2022.

[Mik+13] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. "Efficient Estimation of Word Representations in Vector Space". In: *CoRR* abs/1301.3781 (2013). arXiv: `1301.3781`.

[Mni+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533.

[Mül+15] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. "Cache-Efficient Aggregation: Hashing Is Sorting". In: *Proceedings of the International Conference on Management of Data (SIGMOD).* 2015, pp. 1123–1136.

[Mul90] James K. Mullin. "Optimal Semijoins for Distributed Database Systems". In: *IEEE Transactions on Software Engineering* 16.5 (1990), pp. 558–560.

[MySa] MySQL. *MySQL 8.0 Reference Manual — Optimizing IN and EXISTS Subquery Predicates with Semijoin Transformations.* n.d. URL: `https://dev.mysql.com/doc/refman/8.0/en/semijoins.html` Accessed: April 9, 2022.

[MySb] MySQL. *MySQL 8.0 Reference Manual — SELECT Statement.* n.d. URL: `https://dev.mysql.com/doc/refman/8.0/en/select.html` Accessed: April 9, 2022.

[NP06] Raghunath Othayoth Nambiar and Meikel Poess. "The Making of TPC-DS". In: *Proceedings of the International Conference on Very Large Databases (VLDB).* 2006, pp. 1049–1058.

[NK04] Ullas Nambiar and Subbarao Kambhampati. "Mining Approximate Functional Dependencies and Concept Similarities to Answer Imprecise Queries". In: *Proceedings of the Workshop on the Web and Databases (WebDB).* 2004, pp. 73–78.

[Nat+20] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. “Learning Multi-Dimensional Indexes”. In: *Proceedings of the International Conference on Management of Data (SIGMOD).* 2020, pp. 985–1000.

[Neu14] Thomas Neumann. “Engineering High-Performance Database Engines”. In: *Proceedings of the VLDB Endowment* 7.13 (2014), pp. 1734–1741.

[NK15] Thomas Neumann and Alfons Kemper. “Unnesting Arbitrary Queries”. In: *Proceedings of the Conference Datenbanksysteme in Business, Technologie und Web Technik (BTW).* 2015, pp. 383–402.

[ONe94] Patrick E. O’Neil. *Database Principles, Programming, Performance.* Morgan Kaufmann, 1994.

[Ora20] Oracle. *Oracle Autonomous Database Technical Overview.* White Paper. July 2020.

[Ora] Oracle. *Scalar Subquery Expressions.* n.d. URL: `https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/Scalar-Subquery-Expressions.html` Accessed: April 9, 2022.

[OKC19] Laurel J. Orr, Srikanth Kandula, and Surajit Chaudhuri. “Pushing Data-Induced Predicates Through Joins in Big-Data Clusters”. In: *Proceedings of the VLDB Endowment* 13.3 (2019), pp. 252–265.

[Ott] OtterTune. *OtterTune — Give your database superpowers.* n.d. URL: `https://ottertune.com/features` Accessed: April 9, 2022.

[Pal70] Frank P. Palermo. “A quantitative approach to the selection of secondary indexes”. In: *IBM Research RJ 730.* 1970.

[Pal] Pallets. *Flask — web development, one drop at a time.* n.d. URL: `https://flask.palletsprojects.com/en/2.0.x/` Accessed: April 9, 2022.

[PY10] Sinno Jialin Pan and Qiang Yang. “A Survey on Transfer Learning”. In: *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 22.10 (2010), pp. 1345–1359.

[PA07] Stratos Papadomanolakis and Anastassia Ailamaki. “An Integer Linear Programming Approach to Database Design”. In: *Proceedings of the International Conference on Data Engineering (ICDE).* 2007, pp. 442–449.

[PDA07] Stratos Papadomanolakis, Debabrata Dash, and Anastassia Ailamaki. “Efficient Use of the Query Optimizer for Automated Database Design”. In: *Proceedings of the International Conference on Very Large Databases (VLDB).* 2007, pp. 1093–1104.

[Pap+15a] Thorsten Papenbrock, Jens Ehrlich, Jannik Marten, Tommy Neubert, Jan-Peer Rudolph, Martin Schönberg, Jakob Zwiener, and Felix Naumann. "Functional Dependency Discovery: An Experimental Evaluation of Seven Algorithms". In: *Proceedings of the VLDB Endowment* 8.10 (2015), pp. 1082–1093.

[Pap+15b] Thorsten Papenbrock, Sebastian Kruse, Jorge-Arnulfo Quiané-Ruiz, and Felix Naumann. "Divide & Conquer-based Inclusion Dependency Discovery". In: *Proceedings of the VLDB Endowment* 8.7 (2015), pp. 774–785.

[PN16] Thorsten Papenbrock and Felix Naumann. "A Hybrid Approach to Functional Dependency Discovery". In: *Proceedings of the International Conference on Management of Data (SIGMOD).* 2016, pp. 821–833.

[Pau00] Glenn Norman Paulley. "Exploiting Functional Dependence in Query Optimization". PhD thesis. Department of Computer Science, University of Waterloo, 2000.

[PL94] Glenn Norman Paulley and Per-Åke Larson. "Exploiting Uniqueness in Query Optimization". In: *Proceedings of the International Conference on Data Engineering (ICDE).* 1994, pp. 68–79.

[Pav21] Andrew Pavlo. *Database Deep Dives with Andy Pavlo.* Interview by Josh Mintz (IBM). April 2021. URL: `https://www.ibm.com/cloud/blog/database-deep-dives-with-andy-pavlo` Accessed: April 9, 2022.

[Pav+17] 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: *Proceedings of the Conference on Innovative Data Systems Research (CIDR).* 2017.

[Pav+19] 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". In: *IEEE Data Engineering Bulletin* 42.2 (2019), pp. 32–46.

[Pav+21] Andrew Pavlo, Matthew Butrovich, Lin Ma, Prashanth Menon, Wan Shen Lim, Dana Van Aken, and William Zhang. "Make Your Database System Dream of Electric Sheep: Towards Self-Driving Operation". In: *Proceedings of the VLDB Endowment* 14.12 (2021), pp. 3211–3221.

[Pen+18] Eduardo H. M. Pena, Erik Falk, Jorge Augusto Meira, and Eduardo Cunha de Almeida. “Mind Your Dependencies for Semantic Query Optimization”. In: *Journal of Information and Data Management* 9.1 (2018), pp. 3–19.

[Per+21] R. Malinga Perera, Bastian Oetomo, Benjamin I. P. Rubinstein, and Renata Borovica-Gajic. “DBA bandits: Self-driving index tuning under ad-hoc, analytical workloads with safety guarantees”. In: *Proceedings of the International Conference on Data Engineering (ICDE)*. 2021, pp. 600–611.

[Pia83] Gregory Piatetsky-Shapiro. “The Optimal Selection of Secondary Indices is NP-Complete”. In: *SIGMOD Record* 13.2 (1983), pp. 72–75.

[PHH92] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. “Extensible/Rule Based Query Rewrite Optimization in Starburst”. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1992, pp. 39–48.

[Pit18] Evaggelia Pitoura. “Query Rewriting”. In: *Encyclopedia of Database Systems, Second Edition*. Springer, 2018.

[Pla09] Hasso Plattner. “A common database approach for OLTP and OLAP using an in-memory column database”. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2009, pp. 1–2.

[Pop+00] Lucian Popa, Alin Deutsch, Arnaud Sahuguet, and Val Tannen. “A Chase Too Far?” In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2000, pp. 273–284.

[PF00] Meikel Pöss and Chris Floyd. “New TPC Benchmarks for Decision Support and Web Commerce”. In: *SIGMOD Record* 29.4 (2000), pp. 64–71.

[Que20] Quest. *Database Professionals Look To The Future: 2020 Trends in Database Administration*. White Paper. January 2020.

[RM19] Mark Raasveldt and Hannes Mühleisen. “DuckDB: an Embeddable Analytical Database”. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2019, pp. 1981–1984.

[ŘS10] Radim Řehůřek and Petr Sojka. “Software Framework for Topic Modelling with Large Corpora”. In: *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. 2010, pp. 45–50.

[RSB21] Keven Richly, Rainer Schlosser, and Martin Boissier. “Joint Index, Sorting, and Compression Optimization for Memory-Efficient Spatio-Temporal Data Management”. In: *Proceedings of the International Conference on Data Engineering (ICDE)*. 2021, pp. 1901–1906.

[Rou15] Julien Rouhaud. *HypoPG — Hypothetical Indexes for PostgreSQL*. 2015. URL: `https://github.com/HypoPG/hypopg` Accessed: April 9, 2022.

[Rud19] Cynthia Rudin. "Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead". In: *Nature Machine Intelligence* 1.5 (May 2019), pp. 206–215.

[SUK22] Ibrahim Sabek, Tenzin Ukyab, and Tim Kraska. "LSched: A Workload-Aware Learned Query Scheduler for Analytical Database Systems". In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2022. Accepted, to appear.

[SGL20a] Zahra Sadri, Le Gruenwald, and Eleazar Leal. "DRLindex: deep reinforcement learning index advisor for a cluster database". In: *Proceedings of the International Database Engineering and Applications Symposium (IDEAS)*. 2020, 11:1–11:8.

[SGL20b] Zahra Sadri, Le Gruenwald, and Eleazar Leal. "Online Index Selection Using Deep Reinforcement Learning for a Cluster Database". In: *Proceedings of the International Conference on Data Engineering (ICDE) Workshops*. 2020, pp. 158–161.

[SS96] Hossein Saiedian and Thomas Spencer. "An Efficient Algorithm to Compute the Candidate Keys of a Relational Database Schema". In: *The Computer Journal* 39.2 (1996), pp. 124–132.

[SAP17] SAP. *Understanding the Universal Journal in SAP S/4 HANA*. 2017. URL: `https://blogs.sap.com/2017/03/01/understanding-the-universal-journal-in-sap-s4hana/` Accessed: March 19, 2023.

[SAP19] SAP. *Expressions — Subqueries in Expressions*. October 2019. URL: `https://help.sap.com/viewer/4fe29514fd584807ac9f2a04f6754767/2.0.04/en-US/20a4389775191014b5a6bf2ccc0df2ed.html` Accessed: April 9, 2022.

[SAP21] SAP. *SAP HANA Administration with SAP HANA Cockpit — Recommendations*. May 2021. URL: `https://help.sap.com/viewer/afa922439b204e9caf22c78b6b69e4f2/2.13.0.0/en-US/ce347b55e371480abc1eb27a9d010f25.html` Accessed: April 9, 2022.

[21] *SAP HANA SQL Reference Guide for SAP HANA Platform 2.0 SPS 05*. 1.1. SAP. 2021.

[SGS03] Kai-Uwe Sattler, Ingolf Geist, and Eike Schallehn. “QUIET: Continuous Query-driven Index Tuning”. In: *Proceedings of the International Conference on Very Large Databases (VLDB).* 2003, pp. 1129–1132.

[Sch+13] Jan Schaffner, Tim Januschowski, Megan Kercher, Tim Kraska, Hasso Plattner, Michael J. Franklin, and Dean Jacobs. “RTP: robust tenant placement for elastic in-memory database clusters”. In: *Proceedings of the International Conference on Management of Data (SIGMOD).* 2013, pp. 773–784.

[SSY98] Bernhard Schiefer, Lori G Strain, and Weipeng P Yan. *Method for estimating cardinalities for query processing in a relational database management system.* US Patent 5,761,653. June 1998.

[Sch+19] Philipp Schirmer, Thorsten Papenbrock, Sebastian Kruse, Felix Naumann, Dennis Hempfing, Torben Mayer, and Daniel Neuschäfer-Rube. “DynFD: Functional Dependency Discovery in Dynamic Datasets”. In: *Proceedings of the International Conference on Extending Database Technology (EDBT).* 2019, pp. 253–264.

[Sch75] Mario Schkolnick. “The Optimal Selection of Secondary Indices for Files”. In: *Information Systems (IS)* 1.4 (1975), pp. 141–146.

[SH20] Rainer Schlosser and Stefan Halfpap. “A Decomposition Approach for Risk-Averse Index Selection”. In: *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM).* 2020, 16:1–16:4.

[SH21] Rainer Schlosser and Stefan Halfpap. “Robust and Memory-Efficient Database Fragment Allocation for Large and Uncertain Database Workloads”. In: *Proceedings of the International Conference on Extending Database Technology (EDBT).* 2021, pp. 367–372.

[SKB19] Rainer Schlosser, Jan Kossmann, and Martin Boissier. “Efficient Scalable Multi-attribute Index Selection Using Recursive Strategies”. In: *Proceedings of the International Conference on Data Engineering (ICDE).* 2019, pp. 1238–1249.

[Sch+20] Rainer Schlosser, Jan Kossmann, Martin Boissier, Matthias Uflacker, and Hasso Plattner. *Iterative Multi-Attribute Index Selection for Large Database Systems.* European Patent EP3719663B1; US Patent Application 16/838,830. October 2020.

[SP22] Sebastian Schmidl and Thorsten Papenbrock. “Efficient distributed discovery of bidirectional order dependencies”. In: *VLDB Journal* 31.1 (2022), pp. 49–74.

[SPG09] Karl Schnaitter, Neoklis Polyzotis, and Lise Getoor. “Index Interactions in Physical Design Tuning: Modeling, Analysis, and Applications”. In: *Proceedings of the VLDB Endowment* 2.1 (2009), pp. 1234–1245.

[SJD13] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. “The Uncracked Pieces in Database Cracking”. In: *Proceedings of the VLDB Endowment* 7.2 (2013), pp. 97–108.

[SJD16] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. “An experimental evaluation and analysis of database cracking”. In: *VLDB Journal* 25.1 (2016), pp. 27–52.

[Sch+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. “Proximal Policy Optimization Algorithms”. In: *CoRR* abs/1707.06347 (2017). arXiv: `1707.06347`.

[Sch+14] David Schwalb, Martin Faust, Johannes Wust, Martin Grund, and Hasso Plattner. “Efficient Transaction Processing for Hyrise in Mixed Workload Environments”. In: *Proceedings of the International Workshop on In-Memory Data Management and Analytics (IMDM) at VLDB*. 2014, pp. 16–29.

[Sel+79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. “Access Path Selection in a Relational Database Management System”. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1979, pp. 23–34.

[SCJ15] Rajkumar Sen, Jack Chen, and Nika Jimsheleishvilli. “Query Optimization Time: The New Bottleneck in Real-time Analytics”. In: *Proceedings of the International Workshop on In-Memory Data Management and Analytics (IMDM) at VLDB*. 2015, 8:1–8:6.

[SM17] Nuhad Shaabani and Christoph Meinel. “Incremental Discovery of Inclusion Dependencies”. In: *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*. 2017, 2:1–2:12.

[SSD18] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. “The Case for Automatic Database Administration using Deep Reinforcement Learning”. In: *CoRR* abs/1801.05643 (2018). arXiv: `1801.05643`.

[She+19] Yangjun Sheng, Anthony Tomasic, Tieying Zhang, and Andrew Pavlo. “Scheduling OLTP transactions via learned abort prediction”. In: *Proceedings of the Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM)*. 2019, 1:1–1:8.

[SSM96] David E. Simmen, Eugene J. Shekita, and Timothy Malkemus. “Fundamental Techniques for Order Optimization”. In: *Proceedings of the International Conference on Management of Data (SIGMOD).* 1996, pp. 57–67.

[Sno16] Snowflake. *Automatic Query Optimization. No Tuning!* May 2016. URL: `https://www.snowflake.com/blog/automatic-query-optimization-no-tuning/` Accessed: April 9, 2022.

[Sno] Snowflake. *Using the Search Optimization Service.* n.d. URL: `https://docs.snowflake.com/en/user-guide/search-optimization-service.html#how-does-the-search-optimization-service-work` Accessed: April 9, 2022.

[Son18] Il-Yeol Song. “Data Warehousing Systems: Foundations and Architectures”. In: *Encyclopedia of Database Systems, Second Edition.* Springer, 2018.

[Sta21] Statista. *Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2025.* June 2021. URL: `https://www.statista.com/statistics/871513/worldwide-data-created/` Accessed: April 9, 2022.

[Sto+01] Konrad Stocker, Donald Kossmann, Reinhard Braumandl, and Alfons Kemper. “Integrating Semi-Join-Reducers into State of the Art Query Processors”. In: *Proceedings of the International Conference on Data Engineering (ICDE).* 2001, pp. 575–584.

[Sto75] Michael Stonebraker. “Implementation of Integrity Constraints and Views by Query Modification”. In: *Proceedings of the International Conference on Management of Data (SIGMOD).* 1975, pp. 65–78.

[SRH90] Michael Stonebraker, Lawrence A. Rowe, and Michael Hirohama. “The Implementation of Postgres”. In: *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 2.1 (1990), pp. 125–142.

[Sto08] Daniel P Stormont. “Analyzing human trust of autonomous systems in hazardous environments”. In: *Proceedings of the Human Implications of Human-Robot Interaction workshop at AAAI.* 2008, pp. 27–32.

[SL19] Ji Sun and Guoliang Li. “An End-to-End Learning-based Cost Estimator”. In: *Proceedings of the VLDB Endowment* 13.3 (2019), pp. 307–319.

[SB98] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning — an introduction.* Adaptive computation and machine learning. MIT Press, 1998.

[Szl+17] Jaroslaw Szlichta, Parke Godfrey, Lukasz Golab, Mehdi Kargar, and Divesh Srivastava. “Effective and Complete Discovery of Order Dependencies via Set-based Axiomatization”. In: *Proceedings of the VLDB Endowment* 10.7 (2017), pp. 721–732.

[SGG12a] Jaroslaw Szlichta, Parke Godfrey, and Jarek Gryz. “Chasing Polarized Order Dependencies”. In: *Proceedings of the Alberto Mendelzon International Workshop on Foundations of Data Management.* 2012, pp. 168–179.

[SGG12b] Jaroslaw Szlichta, Parke Godfrey, and Jarek Gryz. “Fundamentals of Order Dependencies”. In: *Proceedings of the VLDB Endowment* 5.11 (2012), pp. 1220–1231.

[Szl+11] Jaroslaw Szlichta, Parke Godfrey, Jarek Gryz, Wenbin Ma, Przemyslaw Pawluk, and Calisto Zuzarte. “Queries on dates: fast yet not blind”. In: *Proceedings of the International Conference on Extending Database Technology (EDBT).* 2011, pp. 497–502.

[Szl+14] Jaroslaw Szlichta, Parke Godfrey, Jarek Gryz, Wenbin Ma, Weinan Qiu, and Calisto Zuzarte. “Business-Intelligence Queries with Order Dependencies in DB2”. In: *Proceedings of the International Conference on Extending Database Technology (EDBT).* 2014, pp. 750–761.

[Taf+18] Rebecca Taft, Nosayba El-Sayed, Marco Serafini, Yu Lu, Ashraf Aboulnaga, Michael Stonebraker, Ricardo Mayerhofer, and Francisco Jose Andrade. “P-Store: An Elastic Database System with Predictive Provisioning”. In: *Proceedings of the International Conference on Management of Data (SIGMOD).* 2018, pp. 205–219.

[Tan+19] 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”. In: *Proceedings of the VLDB Endowment* 12.10 (2019), pp. 1221–1234.

[Tan+20] Zijing Tan, Ai Ran, Shuai Ma, and Sheng Qin. “Fast Incremental Discovery of Pointwise Order Dependencies”. In: *Proceedings of the VLDB Endowment* 13.10 (2020), pp. 1669–1681.

[Thea] The PostgreSQL Global Development Group. *Constraints — Unique Constraints.* n.d. URL: `https://www.postgresql.org/docs/14/ddl-constraints.html` Accessed: April 9, 2022.

[Theb] The PostgreSQL Global Development Group. *CREATE INDEX.* n.d. URL: `https://www.postgresql.org/docs/12/sql-createindex.html` Accessed: April 9, 2022.

[Thec] The PostgreSQL Global Development Group. *Frontend/Backend Protocol.* n.d. URL: `https://www.postgresql.org/docs/12/protocol.html` Accessed: April 9, 2022.

[Thed] The PostgreSQL Global Development Group. *Statistics Used by the Planner — Functional Dependencies.* n.d. URL: `https://www.postgresql.org/docs/14/planner-stats.html` Accessed: April 9, 2022.

[Thee] The PostgreSQL Global Development Group: *Row Estimation Examples.* n.d. URL: `https://www.postgresql.org/docs/13/row-estimation-examples.html` Accessed: April 9, 2022.

[Val+00] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, and Alan Skelley. "DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes". In: *Proceedings of the International Conference on Data Engineering (ICDE).* 2000, pp. 101–110.

[Var] Daniele Varrazzo. *psycopg — PostgreSQL driver for Python.* n.d. URL: `https://www.psycopg.org` Accessed: April 9, 2022.

[Vas] Alexey Vasiliev. *PGTune — How it works.* n.d. URL: `https://pgtune.leopard.in.ua/#/about` Accessed: April 9, 2022.

[Vog+18] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Mühlbauer, Thomas Neumann, and Manuel Then. "Get Real: How Benchmarks Fail to Represent the Real World". In: *Proceedings of the International Workshop on Testing Database Systems (DBTEST).* 2018, 1:1–1:6.

[Wan+21] Jia-Chen Wang, Ding Ding, Huan Wang, Conrad Christensen, Zhaoguo Wang, Haibo Chen, and Jinyang Li. "Polyjuice: High-Performance Transactions via Learned Concurrency Control". In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI).* 2021, pp. 198–216.

[WTB21a] Junxiong Wang, Immanuel Trummer, and Debabrota Basu. "Demonstrating UDO: A Unified Approach for Optimizing Transaction Code, Physical Design, and System Parameters via Reinforcement Learning". In: *Proceedings of the International Conference on Management of Data (SIGMOD).* 2021, pp. 2794–2797.

[WTB21b] Junxiong Wang, Immanuel Trummer, and Debabrota Basu. "UDO: Universal Database Optimization using Reinforcement Learning". In: *Proceedings of the VLDB Endowment* 14.13 (2021), pp. 3402–3414.

[Wan+03] Shyue-Liang Wang, Wen-Chieh Tsou, Jiann-Horng Lin, and Tzung-Pei Hong. “Maintenance of Discovered Functional Dependencies: Incremental Deletion”. In: *Intelligent Systems Design and Applications*. Springer, 2003, pp. 579–588.

[Wan+17] Zhiguang Wang, Chul Gwon, Tim Oates, and Adam Iezzi. “Automated Cloud Provisioning on AWS using Deep Reinforcement Learning”. In: *CoRR* abs/1709.04305 (2017). arXiv: `1709.04305`.

[Wed92] Grant E. Weddell. “Reasoning about Functional Dependencies Generalized for Semantic Data Models”. In: *ACM Transactions on Database Systems (TODS)* 17.1 (1992), pp. 32–64.

[WLL19] Ziheng Wei, Uwe Leck, and Sebastian Link. “Entity Integrity, Referential Integrity, and Query Optimization with Embedded Uniqueness Constraints”. In: *Proceedings of the International Conference on Data Engineering (ICDE)*. 2019, pp. 1694–1697.

[Wha85] Kyu-Young Whang. “Index Selection in Relational Databases”. In: *Proceedings of the International Conference on Foundations of Data Organization (FoDO)*. 1985, pp. 487–500.

[Wu+13] Wentao Wu, Yun Chi, Shenghuo Zhu, Jun’ichi Tatemura, Hakan Hacigümüs, and Jeffrey F. Naughton. “Predicting query execution time: Are optimizer cost models really unusable?” In: *Proceedings of the International Conference on Data Engineering (ICDE)*. 2013, pp. 1081–1092.

[Yan95] Weipeng P. Yan. “Query Optimization Techniques for Aggregation Queries”. PhD thesis. Department of Computer Science, University of Waterloo, 1995.

[YL94] Weipeng P. Yan and Per-Åke Larson. “Performing Group-By before Join”. In: *Proceedings of the International Conference on Data Engineering (ICDE)*. 1994, pp. 89–100.

[YL95] Weipeng P. Yan and Per-Åke Larson. “Eager Aggregation and Lazy Aggregation”. In: *Proceedings of the International Conference on Very Large Databases (VLDB)*. 1995, pp. 345–357.

[Yan+21] Yu Yan, Shun Yao, Hongzhi Wang, and Meng Gao. “Index selection for NoSQL database with deep reinforcement learning”. In: *Information Sciences* 561 (2021), pp. 20–30.

[YL87] H. Z. Yang and Per-Åke Larson. “Query Transformation for PSJ-Queries”. In: *Proceedings of the International Conference on Very Large Databases (VLDB)*. 1987, pp. 245–254.

[Yan+19] Zhe Yang, Phuong Nguyen, Haiming Jin, and Klara Nahrstedt. "MIRAS: Model-based Reinforcement Learning for Microservice Resource Allocation over Scientific Workflows". In: *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*. 2019, pp. 122–132.

[You] Evan You. *Vue.js — The Progressive JavaScript Framework*. n.d. URL: `https://vuejs.org` Accessed: April 9, 2022.

[YS89] Clement T. Yu and Wei Sun. "Automatic Knowledge Acquisition and Maintenance for Semantic Query Optimization". In: *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 1.3 (1989), pp. 362–375.

[Zer] ZeroMQ authors. *ZeroMQ — An open-source universal messaging library*. n.d. URL: `https://zeromq.org` Accessed: April 9, 2022.

[Zha+18] Bohan Zhang, Dana Van Aken, Justin Wang, Tao Dai, Shuli Jiang, Jacky Lao, Siyuan Sheng, Andrew Pavlo, and Geoffrey J. Gordon. "A Demonstration of the OtterTune Automatic Database Management System Tuning Service". In: *Proceedings of the VLDB Endowment* 11.12 (2018), pp. 1910–1913.

[Zha+16] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. "Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes". In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2016, pp. 1567–1581.

[Zha+19] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. "An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning". In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2019, pp. 415–432.

[Zha+21] 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". In: *VLDB Journal* 30.6 (2021), pp. 959–987.

[Zil+04] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy M. Lohman, Adam J. Storm, Christian Garcia-Arellano, and Scott Fadden. "DB2 Design Advisor: Integrated Automatic Physical Database Design". In: *Proceedings of the International Conference on Very Large Databases (VLDB)*. 2004, pp. 1087–1097.