BENJAMIN HILPRECHT

DATA-EFFICIENT LEARNED DATABASE COMPONENTS

DATA-EFFICIENT LEARNED DATABASE COMPONENTS

Doctoral thesis by
Benjamin Hilprecht, M.sc.

submitted in fulfillment of the requirements for the
degree of *Doctor rerum naturalium (Dr. rer. nat.)*

Reviewers
Prof. Dr. rer. nat. Carsten Binnig
Prof. Dr ès sc. Immanuel Trummer

Computer Science Department
Technical University Darmstadt

August 18, 2022

Benjamin Hilprecht
Data-Efficient Learned Database Components
Darmstadt, Technical University of Darmstadt, 2022
Viva voce: 18.10.2022

Please cite this work as
URN: urn:nbn:de:tuda-tuprints-225311
URI: https://tuprints.ulb.tu-darmstadt.de/id/eprint/22531

This document is provided by TUprints, the publication service of the Technical University of Darmstadt
https://tuprints.ulb.tu-darmstadt.de

ERKLÄRUNG LAUT PROMOTIONSORDNUNG

§8 Abs. 1 lit. c PromO

Ich versichere hiermit, dass die elektronische Version meiner Dissertation mit der schriftlichen Version übereinstimmt.

§8 Abs. 1 lit. d PromO

Ich versichere hiermit, dass zu einem vorherigen Zeitpunkt noch keine Promotion versucht wurde. In diesem Fall sind nähere Angaben über Zeitpunkt, Hochschule, Dissertationsthema und Ergebnis dieses Versuchs mitzuteilen.

§9 Abs. 1 PromO

Ich versichere hiermit, dass die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Quellen verfasst wurde.

§9 Abs. 2 PromO

Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, 18.08.2022

Benjamin Hilprecht

Intelligence is based on how efficient a species became at doing the things they need to survive.

— Charles Darwin

ABSTRACT

While databases are the backbone of many software systems, database components such as query optimizers often have to be redesigned to meet the increasing variety in workloads, data and hardware designs, which incurs significant engineering efforts to adapt their design. Recently, it was thus proposed to replace Database Management System (DBMS) components such as optimizers, cardinality estimators, etc. by Machine Learning (ML) models, which not only eliminates the engineering efforts but also provides superior performance for many components.

The predominant approach to derive such learned components is workload-driven learning where ten thousands of queries have to be executed first to derive the necessary training data. Unfortunately, the training data collection, which can take days even for medium-sized datasets, has to be repeated for every new database (i.e., the combination of dataset, schema and workload) a component should be deployed for. This is especially problematic for cloud databases such as Snowflake or Redshift since this effort has to be incurred for every customer.

This dissertation thus proposes data-efficient learned database components, which either reduce or fully eliminate the high costs of training data collection for learned database components. In particular, three directions are proposed in this dissertation, namely *(i)* we first aim to reduce the number of training queries needed for workload-driven components before we *(ii)* propose data-driven learning, which uses the data stored in the database as training data instead of queries, and *(iii)* introduce zero-shot learned components, which can generalize to new databases out-of-the-box, s.t. no training data collection is required.

First, we strive to reduce the number of training queries required for workload-driven components by using simulation models to convey the basic tradeoffs of the underlying problem, e.g., that in database partitioning the network costs of shuffling tuples over the network for joins is the dominating factor. This substantially reduces the number of training queries since the basic principles are already covered by the simulation model and thus only subtleties not covered in the simulation model have to be learned by observing query executions, which we will demonstrate for the problem of database partitioning. An alternative direction is to incorporate domain knowledge (e.g., in a cost model we could encode that scan costs increase linearly with the number of tuples) into components by designing them using differentiable programming. This significantly reduces the number of

learnable parameters and thus also the number of required training queries. We demonstrate the feasibility of the approach for the problem of cost estimation in databases. While both approaches reduce the number of training queries, there is still a significant number of training queries required for unseen databases.

This motivates our second approach of data-driven learning. In particular, we propose to train the database component by learning the data distribution present in a database instead of observing query executions. This not only completely eliminates the need to collect training data queries but can even improve the state-of-the-art in problems such as cardinality estimation or Approximate Query Processing (AQP). While we demonstrate the applicability to a wide range of additional database tasks such as the completion of incomplete relational datasets, data-driven learning is only useful for problems where the data distribution provides sufficient information for the underlying database task. However, for tasks where observations of query executions are indispensable such as cost estimation, data-driven learning cannot be leveraged.

In a third direction, we thus propose zero-shot learned database components, which are applicable to a broader set of tasks including those that require observations of queries. In particular, motivated by recent advances in transfer learning, we propose to pretrain a model once on a variety of databases and workloads and thus allow the component to generalize to unseen databases out-of-the-box. Hence, similar to data-driven learning no training queries have to be collected. In this dissertation, we demonstrate that zero-shot learning can indeed yield learned cost models which can predict query latencies on entirely unseen databases more accurately than state-of-the-art workload-driven approaches, which require ten thousands of query executions on every unseen database.

Overall, the proposed techniques yield state-of-the-art performance for many database tasks while significantly reducing or completely eliminating the expensive training data collection for unseen databases. However, while the proposed directions address the prevalent data-inefficiency of learned database components, there are still many opportunities to improve learned components in the future. First, the robustness and debuggability of learned components should be improved since as of today they do not offer the same transparency as standard code in databases, which can render the components less attractive to be deployed in production systems. Moreover, to increase the applicability of data-driven models it is desirable to increase the coverage of supported queries, e.g., queries involving wildcard predicates on string columns, which are currently not supported by data-driven learning. Finally, we envision that a broader set of tasks should be supported in the future by zero-shot models (e.g.,

query optimization) potentially converging towards complete zero-shot learned systems.

ZUSAMMENFASSUNG

Während Datenbanken das Fundament vieler Softwaresysteme bilden, müssen Datenbankkomponenten wie Anfrageoptimierer häufig neu entworfen werden, um der zunehmenden Vielfalt an Arbeitslasten, Daten und Hardwaredesigns gerecht zu werden, was einen erheblichen technischen Aufwand für die Anpassung ihres Designs bedeutet. Vor kurzem wurde daher vorgeschlagen, DBMS-Komponenten wie Optimierer, Kardinalitätsschätzer etc. durch ML-Modelle zu ersetzen, wodurch nicht nur der Entwicklungsaufwand entfällt, sondern auch eine bessere Leistung für viele Komponenten erzielt wird.

Der vorherrschende Ansatz zur Erstellung solcher gelernten Komponenten ist das Workload-getriebene Lernen, bei dem zunächst zehntausende Abfragen ausgeführt werden müssen, um die erforderlichen Trainingsdaten zu erhalten. Leider muss die Erhebung von Trainingsdaten, die selbst für mittelgroße Datensätze Tage dauern kann, für jede neue Datenbank (d. h. Kombination aus Datensatz, Schema und Arbeitslast) wiederholt werden. Dies ist vor allem bei Cloud-Datenbanken wie Snowflake oder Redshift problematisch, da dieser Aufwand für jeden Kunden anfällt.

In dieser Dissertation werden daher dateneffiziente gelernte Datenbankkomponenten vorgeschlagen, die die hohen Kosten der Trainingsdatenerhebung für gelernte Datenbankkomponenten entweder reduzieren oder ganz eliminieren. Insbesondere werden in dieser Dissertation drei Richtungen vorgeschlagen, nämlich zielen wir *(i)* zunächst darauf ab, die Anzahl der benötigten Trainingsabfragen für Workload-getrieben Komponenten zu reduzieren, bevor wir *(ii)* datengetriebenes Lernen vorschlagen, das die in der Datenbank gespeicherten Daten als Trainingsdaten anstelle von Abfragen verwendet, und *(iii)* gelernte Zero-Shot-Komponenten einführen, die out-of-the-Box auf neue Datenbanken verallgemeinert werden können, d.h. bei denen keine Trainingsdatenerfassung erforderlich ist.

Zunächst haben wir als Ziel, die Anzahl der für Workload-getriebene Komponenten erforderlichen Trainingsabfragen zu reduzieren, indem wir Simulationsmodelle verwenden, um die grundlegenden Effekte des zugrunde liegenden Problems zu vermitteln, z.B. dass bei der Datenbankpartitionierung die Netzwerkkosten für das Verteilen von Tupeln über das Netzwerk für Joins der dominierende Faktor sind. Dies reduziert die Anzahl der Trainingsabfragen erheblich, da die grundlegenden Prinzipien bereits durch das Simulationsmodell abgedeckt sind und somit nur noch die Feinheiten, die nicht durch das Simulationsmodell erfasst sind, durch Beobachtung von Abfrageausführungen erlernt werden müssen, was wir für das Problem der Datenbankpartitionierung demonstrieren werden. Eine alternative

Richtung ist, Domänenwissen (z.B. könnten wir in einem Kostenmodell kodieren, dass die Scankosten linear mit der Anzahl der Tupel ansteigen) in die Komponenten einzubauen, indem wir sie mit differenzierbarer Programmierung entwerfen. Dies reduziert die Anzahl der lernbaren Parameter und damit auch die Anzahl der erforderlichen Trainingsabfragen erheblich. Wir demonstrieren die Machbarkeit des Ansatzes für das Problem der Kostenabschätzung in Datenbanken. Obwohl beide Ansätze die Anzahl der Trainingsabfragen reduzieren, ist immer noch eine erhebliche Anzahl von Trainingsabfragen für ungesehene Datenbanken erforderlich.

Dies motiviert unseren zweiten Ansatz des datengetriebenen Lernens. Wir schlagen insbesondere vor, die Datenbankkomponente zu trainieren, indem wir die Datenverteilung in einer Datenbank lernen, anstatt die Ausführung von Abfragen zu beobachten. Dies macht nicht nur das Sammeln von Trainingsdaten überflüssig, sondern kann sogar den Stand der Technik bei Problemen wie der Kardinalitätsschätzung oder AQP verbessern. Während wir die Anwendbarkeit auf eine breite Auswahl zusätzlicher Datenbankaufgaben wie die Vervollständigung unvollständiger relationaler Datensätze demonstrieren, ist datengetriebenes Lernen nur für Probleme nützlich, bei denen die Datenverteilung ausreichende Informationen für die zugrunde liegende Datenbankaufgabe liefert. Für Aufgaben, bei denen Beobachtungen von Abfrageausführungen unverzichtbar sind, wie z.B. bei der Kostenschätzung, kann datengetriebenes Lernen jedoch nicht genutzt werden.

In einer dritten Richtung schlagen wir daher Zero-Shot-gelernte Datenbankkomponenten vor, die auf eine breitere Palette von Aufgaben anwendbar sind, einschließlich solcher, die Beobachtungen von Abfragen erfordern. Motiviert durch die jüngsten Fortschritte im Transfer-Lernen schlagen wir vor, ein Modell einmalig auf einer Vielzahl von Datenbanken und Arbeitslasten vorzutrainieren und so der Komponente zu ermöglichen, sofort auf unbekannte Datenbanken zu generalisieren. Ähnlich wie beim datengesteuerten Lernen müssen also keine Trainingsabfragen gesammelt werden. In dieser Dissertation zeigen wir, dass Zero-Shot-Learning tatsächlich gelernte Kostenmodelle hervorbringen kann, die Abfragelatenzen auf gänzlich unbekannten Datenbanken genauer vorhersagen können als moderne Workloadgetriebene Ansätze, die zehntausende von Abfrageausführungen auf jeder unbekannten Datenbank erfordern.

Insgesamt liefern die vorgeschlagenen Verfahren für viele Datenbankaufgaben eine Leistung auf dem neuesten Stand der Technik, während die teure Erhebung von Trainingsdaten für unbekannte Datenbanken erheblich reduziert oder ganz vermieden wird. Während die vorgeschlagenen Richtungen die vorherrschende Dateneffizienz von gelernten Datenbankkomponenten adressieren, gibt es jedoch noch viele Möglichkeiten, gelernte Komponenten in Zukunft zu ver-

bessern. Erstens sollten die Robustheit und die Debugging-Fähigkeit der gelernten Komponenten verbessert werden, da sie derzeit nicht die gleiche Transparenz wie Standardcode in Datenbanken bieten, was die Komponenten für den Einsatz in Produktionssystemen weniger attraktiv machen kann. Um die Anwendbarkeit von datengetriebenen Modellen zu erhöhen, ist es außerdem wünschenswert, die Abdeckung der unterstützten Abfragen zu erhöhen, z.B. Abfragen mit Wildcard-Prädikaten auf String-Spalten, die derzeit nicht von datengetriebenem Lernen unterstützt werden. Schließlich könnten in Zukunft viele weitere Aufgaben durch Zero-Shot-Modelle unterstützt werden (z.B. Abfrageoptimierung), was vollständige Zero-Shot-gelernte Systemen ermöglichen könnte.

PUBLICATIONS

The following peer-reviewed publications are part of this cumulative dissertation. Their content is printed in Part ii, Chapters 7 to 12.

[1] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. "Learning a Partitioning Advisor for Cloud Databases." In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. Ed. by David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo. ACM, 2020, pp. 143–157. DOI: 10.1145/3318464.3389704. URL: https://doi.org/10.1145/3318464.3389704.

[2] 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: *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020. URL: http://cidrdb.org/cidr2020/papers/p34-hilprecht-cidr20.pdf.

[3] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. "DeepDB: Learn from Data, not from Queries!" In: *Proc. VLDB Endow.* 13.7 (2020), pp. 992–1005. DOI: 10.14778/3384345.3384349. URL: http://www.vldb.org/pvldb/vol13/p992-hilprecht.pdf.

[4] Benjamin Hilprecht and Carsten Binnig. "ReStore - Neural Data Completion for Relational Databases." In: *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. Ed. by Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava. ACM, 2021, pp. 710–722. DOI: 10.1145/3448016.3457264. URL: https://doi.org/10.1145/3448016.3457264.

[5] Benjamin Hilprecht and Carsten Binnig. "One Model to Rule them All: Towards Zero-Shot Learning for Databases." In: *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org, 2022. URL: https://www.cidrdb.org/cidr2022/papers/p16-hilprecht.pdf.

[6] Benjamin Hilprecht and Carsten Binnig. "Zero-Shot Cost Models for Out-of-the-box Learned Cost Prediction." In: *Proc. VLDB Endow.* 15.11 (2022), pp. 2361–2374. DOI: 10.14778/3551793.

3551799. URL: http://www.vldb.org/pvldb/vol15/p2483-hilprecht.pdf.

Further co-authored peer-reviewed publications are:

[1] Moritz Kulessa, Benjamin Hilprecht, Alejandro Molina, Kristian Kersting, and Carsten Binnig. "Towards Model-based Approximate Query Processing." In: *AIDB@VLDB 2019, 1st International Workshop on Applied AI for Database Systems and Applications, Held with VLDB 2019*. Ed. by Berthold Reinwald and Bingsheng He. 2019.

[2] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. "Towards learning a partitioning advisor with deep reinforcement learning." In: *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2019, Amsterdam, The Netherlands, July 5, 2019*. Ed. by Rajesh Bordawekar and Oded Shmueli. ACM, 2019, 6:1–6:4. DOI: 10.1145/3329859.3329876. URL: https://doi.org/10.1145/3329859.3329876.

[3] Marius Gassen, Benjamin Hättasch, Benjamin Hilprecht, Nadja Geisler, Alexander Fraser, and Carsten Binnig. "Demonstrating CAT: Synthesizing Data-Aware Conversational Agents for Transactional Databases." In: *Proc. VLDB Endow.* 15.12 (2022), pp. 3586–3589. DOI: 10.14778/3554821.3554850. URL: http://www.vldb.org/pvldb/vol15/p2335-gassen.pdf.

[4] Johannes Wehrstein, Benjamin Hilprecht, Benjamin Olt, Manisha Luthra, and Carsten Binnig. "The Case for Multi-Task Zero-Shot Learning for Databases." In: *AIDB@VLDB 2022, 4th International Workshop on Applied AI for Database Systems and Applications, Held with VLDB 2022, Monday, September 5, 2022, Sydney, Australia*. Ed. by Umar Farooq Minhas and Yingjun Wu. 2022.

Due to the nature of the synopsis and for better readability, selected paragraphs from these publications were transferred verbatim throughout the synopsis without explicit labeling as suggested in the department regulations "Kumulative Dissertation und Eigenzitate in Dissertationen" (21.09.2021) §1.

ACKNOWLEDGMENTS

First and foremost, I would like to express my gratitude to Prof. Dr. Carsten Binnig for being an incredible advisor who has always inspired me to grow both personally and academically. He has been a truly ambitious, ingenious, motivating, and supportive mentor over the years.

I would also like to thank Prof. Dr. Immanuel Trummer for readily taking the time to review this dissertation.

My colleagues at the Data Management Lab at TU Darmstadt have made the last years very enjoyable for me. I am sincerely grateful for the supportive atmosphere, the feedback on ideas and work, and the excellent discussions we had, as well as the numerous after-work events. My thanks also go to Mona, who has always helped me with hurdles big and small during my time in the lab.

Fortunately, in the past years, I had the chance to work with great people on many projects, including Prof. Dr. Andreas Schmidt, Prof. Dr. Kristian Kersting, Prof. Dr. Uwe Röhm, Patrick Schramowski, Dr. Alejandro Molina and Dr. Moritz Kulessa, from whom I could learn a lot. I would also like to thank my colleagues, including Lasse, Nils, and Matthias for the many nice evenings after work.

I am also very grateful to my parents, brother, and friends for their constant support. Finally, I would like to thank Laura for not only helping me through the downs of this time but also celebrating the ups with me.

CONTENTS

ACRONYMS

ML	Machine Learning
DBMS	Database Management System
RL	Reinforcement Learning
UDF	User-defined Function
AQP	Approximate Query Processing
DNN	Deep Neural Network
SPN	Sum-product Network
RSPN	Relational Sum-product Network

Part I

SYNOPSIS

1 INTRODUCTION

Databases are the backbone for managing data in many software systems deployed today. However, due to the increasing variety in workloads, data and hardware designs, significant manual and engineering efforts are required to adapt their design. Hence, it was recently proposed to instead *learn* the design of database components, which reduces the efforts to adapt databases and can even improve the end-to-end system performance. However, state-of-the-art approaches require a costly training data collection for every new database[1] they should be trained for.

In the following, we will first discuss how the diversity in workloads, data and hardware deployments can be addressed in the database design and afterwards discuss how learned database components can be leveraged to address this challenge. We will then describe the fundamental limitations of such approaches today and introduce the concrete contributions of this dissertation.

1.1 THE NEED FOR DATABASE ADAPTION

In particular due to the widespread adoption of cloud databases [8, 30, 122, 189], there is an increasing diversity both in the observed data and workloads [155, 174] but also in the different hardware configurations. There are two main directions to address this increasing complexity in the design of DBMSs: *(i) configurable databases*, which allow the user to adapt and tune the system and *(ii) specialization*, where database components are tailored to a particular deployment. However, both approaches incur significant manual or engineering efforts and yet do not always enable an optimal system performance.

First, many databases today offer configurability by introducing tuning knobs [3, 18, 37] such as the buffer pool size and allowing users to decide on the physical design, e.g., by creating indexes [171] or materialized views [4]. While this enables the specialization of the database to some extent, it comes with two major downsides: the tuning itself is non-trivial due to the complex interactions of tuning knobs, workloads and the physical design and thus a significant manual effort is often required to tune the knobs. While there have been approaches to alleviate this problem using automated configuration [4, 6, 134], even with automation the end-to-end system performance

1 Throughout this dissertation, we will refer to a new schema along with the data and workload as database. Hence, we explicitly do not refer to different database systems using this term.

might still be suboptimal. The reason is that the underlying database components but also automation approaches heavily rely on heuristics inherently in their design. For instance, automated physical design advisors often propose designs, which are far from optimal and could be improved significantly [4, 81, 99].

A second approach to cope with the increasing complexity is the specialization of database components to specific deployment conditions such as new types of hardware. This however incurs a significant engineering overhead. For instance, query optimizers frequently have to be adapted by engineers to reflect novel hardware setups with different network or storage costs [93], new physical layouts such as column stores [159] or optimizations such as materialized views [55], all of which have a significant impact on which physical plans are suitable for a particular query. Over the years, this resulted in highly complex code that could only be maintained by a few experts [112]. However, despite all the engineering efforts worth thousands of person-engineering-hours [112], query optimizers still frequently do not come up with efficient physical execution plans [91, 93].

Hence, both specialization and configuration incur significant manual or engineering overheads but still cannot guarantee an optimal system performance due to the reliance on heuristics.

1.2 TOWARDS LEARNED DATABASE COMPONENTS

In contrast to database configuration or manual specialization, the idea of *learned database components* is to leverage ML to tailor a component to particular hardware, dataset and workload [83]. The core idea is to replace previously manually designed parts of a database by a ML model and thus *learn* a suitable design automatically. This not only allows for automatic adaption of database components but also often yields superior components. This is due to the fact that ML can typically cope with the underlying complexity of the problem (e.g., the before mentioned interactions of physical design and workloads) while not relying on heuristics. Hence, the search space of possible designs is captured more holistically and thus the performance can often be further improved.

In fact, many DBMS components have successfully been replaced by learned counterparts including query optimizers [109, 112, 184], cardinality [79, 185, 186] and cost estimators [113, 161] and even indexes [34, 35, 48, 80, 84] or schedulers [107, 153]. In addition, ML-based approaches have been devised to optimize the physical design of databases, e.g., by selecting materialized views [59, 99, 188] or indexes [33, 82, 140, 193]. We will now first describe the advantages of learned DBMS components before we introduce the main paradigm today, namely workload-driven learning.

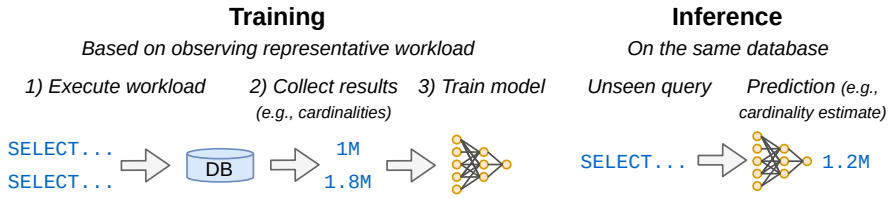


Figure 1.1: Training and Inference of Workload-driven Models. For the training, we first have to observe a representative workload (e.g., queries along with their cardinalities). At inference time, the model can be used on the same database to obtain predictions (e.g., cardinalities for unseen queries).

The resulting learned DBMS components have several advantages compared with manually designed ones. First, they reduce the engineering effort to adapt the respective component. In particular, in many cases, novel workloads, data distributions or hardware characteristics, which have previously required to manually redesign the component, can now be handled by simply retraining the respective ML model. This significantly reduces the engineering efforts for such adaptions.

Second, in some cases the learned counterparts can even outperform existing DBMS components. The reason is that the latter are usually based on general-purpose heuristics and hence not optimized for a specific database instance that comes with a particular combination of data, workload and hardware. In contrast, learned components can be tailored to the peculiarities of a particular instance and are thus more specialized. For instance, a learned query optimizer could observe typical User-defined Functions (UDFs) that are used and observe their selectivities which are useful for query optimization [155]. While it is hard to reason about general UDFs, in the case of a single DBMS instance, there are typical patterns that can be exploited, which a specialized learned component can leverage. In general, it was shown that in many cases such *instance-optimized* components outperform classical designs [79, 83, 109].

The predominant approach of designing such learned DBMS components today is *workload-driven* learning, where the main idea is to first observe a representative set of queries executed on a database and afterwards use these as training data for an ML model (cf. Figure 1.1). For instance, for a learned cardinality estimator [79], we would run thousands of queries with different join paths and predicates and observe the result size (i.e., the cardinality). These pairs of queries and cardinalities would then serve as training data for an ML model. If we now want to estimate the cardinality of an unseen query at runtime, we would feed it into the trained model, which would in turn output a cardinality estimate. Since the model has internalized potential correlations in the data by observing cardinalities of previously executed

training queries, the result will likely be more accurate than those of classical histogram-based approaches, which assume independence.

While a large class of the learned database components are based on Reinforcement Learning (RL) [59, 82, 89, 99, 112, 184, 192], the approaches generally still follow the workload-driven paradigm. In particular, in RL, the idea is to solve a task by allowing an agent to repeatedly interact with an environment by taking actions [148]. During training, the agent observes rewards and consequently learns which actions likely result in high rewards. Over time, the agent will thus be able to effectively solve the task at hand. To demonstrate how RL can be leveraged for learning DBMS components, let us consider the task of query optimization [112]. In this example, the environment is a particular database, the state could be a partial physical query plan and actions are to add a particular operator to the current plan. Hence, the agent can incrementally add physical operations to the current plan to derive the final execution plan. Our goal is to find efficient execution plans and thus the reward is the negated execution latency of the final resulting plan. During the training, the agent learns to suggest efficient physical plans for the given queries by trial and error. Importantly, the training data in such approaches is thus still comprised of queries, which are executed on the database at hand, and thus the resulting approaches are similarly workload-driven.

1.3 LIMITATIONS OF WORKLOAD-DRIVEN LEARNING

While workload-driven learned DBMS components have successfully been applied to a large set of database tasks, they cause repeated high costs for training data collection. We will next provide more details on this fundamental limitation.

First, workload-driven learned DBMS components typically require ten thousands of query executions as training data on the database they should be used for. For instance, learned cardinality estimators [79, 161] use approximately 100,000 training queries, i.e., pairs of queries associated with their observed output cardinalities. Training workloads of this size already take approximately 34 hours to be executed on a medium-sized database of just several GBs using Postgres [72]. These costs, of course, increase significantly for large databases and can easily be a barrier to the use of learned database components. For instance, it was reported that the training data collection of just ten thousand queries for a database size of 1 TB would require more than six months [173]. Such immense costs for training data collection are unacceptable in many cases and render workload-driven learning impractical especially for large-scale databases.

Second, these high costs of training data collection are repeated every time the underlying database is updated [72]. For instance, due to inserts the cardinalities of the training queries will no longer

be valid and thus the output cardinalities of the trained model will become inaccurate since they are based on stale training data. The only way to refresh the underlying model is to retrain it using queries that reflect the new data in the database and thus we again have to run many queries on the updated database and use these to train the model.

Finally, the resulting models are tied to a single database [62], i.e., a single schema, dataset and workload. As soon as we encounter a new database, we will not be able to reuse the workload-driven model and thus have to again incur the high costs of training data collection. The reason is that the architectures of workload-driven models come with the inherent assumption that the underlying database is fixed and thus they will either not be usable at all for a new database or the predictions will be very inaccurate since the model quality will degrade heavily.

Overall, workload-driven models are thus not *data-efficient* since they *(i)* require thousands of training data samples in the form of query executions, which is costly for larger databases, and *(ii)* since the training data becomes unusable in case of updates of the same database or a new database. This renders workload-driven models especially unusable for cloud database providers (e.g., Snowflake and Redshift) since the high costs of training data collection are incurred for every new customer that typically comes with a new schema, dataset and workload.

1.4 DATA-EFFICIENT LEARNED DATABASE COMPONENTS

The dissertation proposes *data-efficient* learned database components that reduce or completely eliminate the previously described high costs of training data collection. We believe this has the potential to increase the traction of using learned components in practical systems - especially for cloud databases, where costs can be significantly reduced for new customers.

In this dissertation, instead of showing how the data-efficiency of individual components can be improved, we will introduce three conceptual approaches, that are applicable to a broad set of learned components. To validate the feasibility, we will also demonstrate how these can be used to instantiate learned approaches for concrete DBMS tasks that outperform the state-of-the-art with significantly less or no training queries at all for an unseen database. In particular, we have intentionally chosen tasks that have are central in a DBMS system but difficult to solve today such as cardinality estimation, cost estimation or physical design tuning.

CONTRIBUTIONS We propose three different directions to enable data-efficient learned DBMS components: *(i)* improving the data-

efficiency of workload-driven approaches, *(ii)* data-driven learning and *(iii)* zero-shot learned components. For each direction, the underlying conceptual ideas are validated by solving particular database tasks as additional individual contributions.

1. *The first direction* of this dissertation is to improve the data-efficiency of workload-driven models. Specifically, we propose to bootstrap learned components using simulation [69], which allows the models to learn the basic principles of the underlying database problem by interacting with a simulation instead of a real system. This can already significantly reduce the necessary number of training queries. Moreover, we suggest models that explicitly encode domain knowledge about databases [67]. Since this reduces the number of parameters, we can train such models with orders of magnitude fewer training queries.

2. *The second direction* of this dissertation is to introduce a paradigm called data-driven learning, where the idea is to completely avoid the use of queries as training data. Instead, we aim to learn the data distribution present in a concrete database, which is used at runtime to solve the task at hand. Specifically, we show how the tasks of cardinality estimation and AQP can be solved with data-driven models, which outperform the state-of-the-art without requiring any training queries [72]. Moreover, we demonstrate that the underlying ability to learn the data distribution of relational schemas also helps to correct the error induced by incomplete datasets in analytics [63]. While the need for training queries is eliminated, data-driven learning is only applicable to a subset of tasks where it is sufficient to know the data distribution.

3. *The third direction* of this dissertation is thus to suggest zero-shot learned database components [62] that can be used also for tasks where the data distribution alone is insufficient. The main idea is to pretrain a learned component on a diverse set of databases and workloads, which allows the model to generalize to unseen databases and workloads out-of-the-box, i.e., without observing additional training queries.

OUTLINE The synopsis continues in Chapter 2, where the three directions along with the corresponding key ideas are set into perspective. Afterwards, we provide more details on each individual direction in Chapters 3 to 5, which correspond to the publications in Part ii. In particular, Chapter 3 introduces approaches to improve the data-efficiency of workload-driven models using simulation and incorporating domain knowledge. Chapter 4 introduces the idea of data-driven learning along with the applications for cardinality estimation, AQP and analytics over incomplete data. In Chapter 5, the

contributions towards zero-shot learned database components are summarized. Finally, Chapter 6 concludes the synopsis with a perspective about the contributions towards data-efficient learned DBMS components and proposes future research directions and open challenges.

2

DATA-EFFICIENT LEARNED DATABASE COMPONENTS

In the following chapter, we will provide an overview of the directions proposed in this dissertation to enable data-efficient learned database components. The general structure follows the three directions previously introduced in Section 1.4, namely *(i)* improving the data-efficiency of workload-driven learning, *(ii)* data-driven learning and *(iii)* zero-shot learned components.

As depicted in Figure 2.1, in the first direction, the main criticism of workload-driven learning of expensive training data collection is already alleviated since the number of required training queries is reduced significantly. In contrast, the latter two directions completely eliminate the need to collect training data for unseen databases by refraining from using queries as training data (data-driven learning) or enabling models that generalize to unseen databases (zero-shot learned components). While data-driven learning and zero-shot learning could be seen as alternative paradigms for efficient data-driven learning, we will demonstrate that depending on the concrete task, one or the other approach should be applied and they can even be used in combination.

To validate each of the three directions, we implemented the ideas for a concrete set of database tasks. This yielded data-efficient learned database components that outperform the state-of-the-art approaches while requiring fewer or no training queries at all. In this chapter, we will similarly introduce the concrete database tasks that we tackled.

2.1 DATA-EFFICIENT WORKLOAD-DRIVEN LEARNING

In the first direction, we propose to alleviate the repeated high costs of workload-driven learning by reducing the number of training queries that have to be observed in order to train a model. In general, we thus still rely on the paradigm of workload-driven models that are specialized for a single database but reduce the costs of training data collection.

In particular, we follow two approaches: *simulation* to bootstrap the learned components for a new database, which reduces the number of required training queries and *differentiable programming* where domain knowledge about the component is incorporated. The two approaches are validated on the important tasks of partitioning design as part of the physical design tuning and the cost estimation task, respectively.

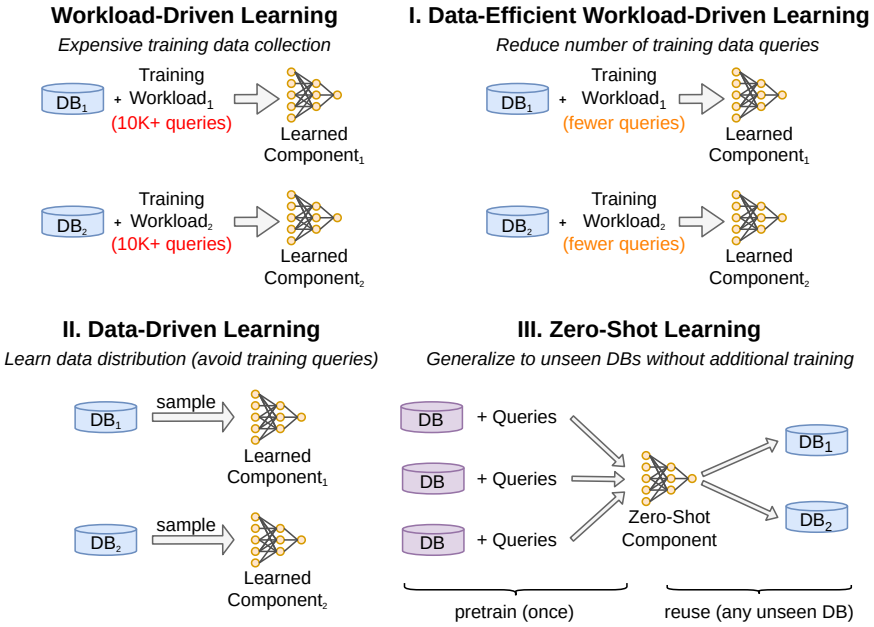


Figure 2.1: Proposed Directions towards Data-Efficient Learned Database Components. *(i)* We first aim at improving the data-efficiency of workload-driven learning, which however merely alleviates the need for training queries for every new database. We thus propose the paradigms of *(ii)* data-driven learning where we learn from the data distribution instead and *(iii)* zero-shot learned components, which generalize to unseen databases out-of-the-box.

Simulation for Data-efficient Learned Partitioning

In workload-driven learning, a new model has to be trained for every single database [79, 161]. Consequently, the general principles of the database task have to be learned for every new database from scratch. For instance, for database partitioning we have to observe for every new database that network shuffles for joining large tables are expensive [134]. This makes workload-driven learning particularly costly since even basic principles of the underlying problem have to be learned by observing many training queries for every new database.

Hence, the idea of simulation is to bootstrap the learned component using a simplified model that reflects the basic principles of the underlying database task, e.g., that a partitioning that requires many large tables to be shuffled over the network is not optimal. In particular, the simulation model is used to generate training examples for an unseen database without actually executing queries. While these training samples cannot possibly convey all subtle trade-offs of the underlying problem, they can reflect the basic principles. The learned component will thus first be trained on this artificial training data to learn the basic principles and afterwards on queries executed in the real database to further learn the trade-offs, which are not reflected in the simulation model. This can help to reduce the number of training queries, which are required in total.

In the context of this dissertation, we aimed at designing a learned model to decide on the database partitioning. In this task, we are interested in how tuples of tables should be distributed among nodes in a distributed cluster (e.g., by replicating all tuples or sharding based on some key column). This task is typically decided manually by users in the cloud but is non-trivial and has a significant impact on the overall performance of the system. The reason is that the partitioning determines how many network shuffles of tuples are required in order to execute distributed joins, which is a costly operation. Previous work on automated partitioning [3, 127, 145] formalizes the problem as an optimization problem and thus relies on cost models to estimate the runtime of queries for different partitionings. Unfortunately, cost estimators are often largely under- or overestimating the true query runtime [91] and thus such approaches are prone to select suboptimal partitionings.

In contrast, we propose to train a RL agent for partitioning, where the agent explores different partitionings and their impact on the workload runtime by trial-and-error. Hence, our approach is not affected by inaccurate cost estimates. Over time, it will learn, which partitionings are suitable for a particular workload. However, this requires that many training queries are executed for various partitionings, which is costly. Hence, we designed a simplified cost model that approximates the query runtime by estimating the network cost for a particular workload, which we then use to bootstrap the agent. As we have shown, this can reduce the number of required training queries. Overall, the agent was able to suggest partitionings, which improve the system performance while outperforming state-of-the-art approaches and heuristics. More details are presented in Section 3.1 and the corresponding publication in Chapter 7.

Incorporating Domain Knowledge using Differentiable Programming

In addition, we suggest including domain knowledge about the specific database task to be solved. This can significantly reduce the number of parameters of the ML model and in turn the number of required training queries. We will first describe the basic idea of differentiable programs in databases and afterwards provide more details on the concrete use case of cost estimation.

The approach is inspired by a recent shift in ML called differentiable programming towards simpler white-box models [176] as opposed to Deep Neural Networks (DNNs) with increasingly many parameters. For instance, differentiable programming has successfully been used in computer vision [98] to include domain knowledge about image features (e.g., edge detectors) or to learn a physics engine where the laws of physics are encoded [9].

The idea for deriving data-efficient learned database components is that database developers sketch the design of the component by implementing a differentiable program, i.e., a program that already encodes the basic behavior of a component but with learnable parameters to specialize the code to a particular database, hardware and workload. As such, the domain knowledge about the component is already encoded. The parameters of the differentiable program are then learned via backpropagation using training queries just like in workload-driven models. However, since there are fewer parameters, we only need to observe a fraction of queries required for workload-driven learning.

As a concrete use case, we consider cost estimation, i.e., predicting the latency of a query. In particular, we modeled the general runtime complexity of different operators (e.g., by modeling the runtime of a scan operation as a linear function) by defining differentiable programs. We then used query latencies as training data to fit the parameters and could afterwards predict the costs for other queries. We could demonstrate that for simple queries we could predict the latencies as accurately as workload-driven models [113, 161] but using orders of magnitude fewer training queries. We provide more details in Section 3.2 and the corresponding publication in Chapter 8.

Discussion

Both directions can significantly improve the data-efficiency of workload-driven models for new databases. However, they still do not fully eliminate the need for training queries on every new database. In addition, simulation and in particular differentiable programming can introduce a major engineering overhead.

In particular, the simulation approach requires a simplified model that captures the basic trade-offs of the underlying database task, which can be challenging to design. The reason is that ML is often applied for database tasks with complex interactions, which are hard to model explicitly even in a simplified model. However, even when the simulation model is overly simplistic, the ML model can still learn the more involved effects by observing training data. This is different in the differentiable programming approach. Here, the general logic has to be completely defined in the program which is challenging for more complex problems. For instance in cost estimation, operators are known to have complex interactions such as caching effects. Modeling all such effects explicitly (which is necessary for differentiable programming) is challenging.

Hence, the costly training data collection of workload-driven learning is not completely eliminated and additional complexity in the development process of learned database components is introduced.

2.2 DATA-DRIVEN LEARNING

The main drawback of workload-driven training is the expensive training data collection. While the previously introduced approaches reduce the number of required query executions and thus improve the data-efficiency, this effort is still a burden if models should be trained for new databases. We thus propose a radically different approach, where we only learn from the data in the database itself instead of query executions, which completely eliminates the high costs of training data collection.

In particular, in our approach of data-driven learning, which is the second direction proposed in this dissertation, the idea is to instead learn the data distribution to solve a database task, which completely eliminates the need for training queries. We will now introduce two concrete learned components based on this paradigm: DeepDB, which enables cardinality estimation and AQP, as well as ReStore, which tackles analytics over incomplete relational data.

DeepDB: Data-Driven Learning for Cardinality Estimation and AQP

In DeepDB, the idea is to learn generative ML models over the relational schema of a database to capture the data distribution. Hence, the models only require samples of the data to be trained and no training queries at all. We can then exploit this knowledge about the data distribution at inference time to solve the tasks such as cardinality estimation or AQP. We will now first introduce both tasks that are tackled using DeepDB before we highlight the main contributions.

In cardinality estimation, the goal is to estimate the result size of intermediate joins with predicates and it is crucial for join ordering and query optimization [91]. Traditionally, cardinalities are estimated using simple histogram-based methods that assume independence. However, due to correlations in the data, which are not captured in histograms, the cardinalities are orders of magnitude off in practice [93]. Hence, workload-driven models [79, 126, 154, 161] were suggested that observe ten thousands of different queries along with their cardinalities to predict the cardinalities for unseen queries at runtime, which can provide more accurate estimates but are costly due to the training data gathering.

In contrast, in DeepDB, we only require samples of the database to train the ML models. For instance, if we want to estimate how many customers in our database are from Europe and younger than thirty, we can approximate this query by estimating the probability of the joint event (younger than thirty and from Europe) using the generative model and multiplying it with the number of tuples in the database to obtain an estimate. While this basic technique of reducing the problem to probability estimation was previously suggested, prior

approaches [60, 186] do not allow cardinality estimates for arbitrary ad-hoc joins in the schema. In particular, the approaches do not scale to larger schemas since all potential join paths of queries require a different model and thus for larger schemas the number of models to support different join paths grows combinatorically. In DeepDB, we instead propose a method to support arbitrary equi-joins with a linear number of models.

In AQP [22], the goal is to approximate the result of a potentially long-running query quickly, which is in particular interesting for interactive applications [2]. Analogously to cardinality estimation, capturing the data distribution using generative models is sufficient to approximate many queries. For instance, if we are now interested in the average age of customers in Europe, we could query our generative model for the conditional expectation of the age given that a customer is from Europe. In contrast to DeepDB, prior approaches suffer heavily from the curse of dimensionality and thus do not scale to larger schemas or wider tables [20, 105].

DeepDB introduces two main contributions: Relational Sum-product Networks (RSPNs) and probabilistic query compilation. RSPNs are Sum-product Networks (SPNs) [118, 141] optimized for relational data, i.e., generative models that model the probability distribution of joins. Importantly, RSPNs can also be updated efficiently, which makes it cheaper to adapt them to inserts, updates and deletes of the underlying database. However, if used naively, we would require a single RSPN for every possible join path in the schema (e.g., `customer`$\bowtie$`order`, `order`$\bowtie$`orderline`, `customer`$\bowtie$`order`$\bowtie$`orderline`,...) similar to prior work [60, 186], which does not scale to larger schemas. Instead, we introduce probabilistic query compilation, which enables arbitrary equi-joins since we are able to combine RSPNs to approximate a larger join and are also able to approximate a subjoin of an RSPN. In the experimental evaluation, we demonstrate that DeepDB provides cardinality estimates, which are orders of magnitude more accurate than those of both traditional approaches based on histograms as well as workload-driven models while requiring no training queries at all. In addition, DeepDB also supports accurate AQP results for a large class of analytical queries outperforming the state-of-the-art. DeepDB is discussed in more detail in Section 4.1 and the corresponding publication in Chapter 9.

ReStore: Data-Driven Completion of Incomplete Relational Datasets

In fact, learning the data distribution of a relational database is also useful for other tasks beyond cardinality estimation and AQP. In particular, we also considered analytical queries over incomplete datasets, where capturing the data distribution can help to compensate for

errors due to incompleteness. We will now first provide an intuition on the problem and afterwards discuss the concrete contributions.

Classically, databases for analytical query processing come with the implicit assumption that the data in all tables is complete, i.e., that no tuples are missing. Traditionally, this assumption often held since the data mostly came from well-curated internal sources. However, for analytics it is common to augment the datasets with open data, which can quickly result in incomplete datasets. Consequently, the query results can deviate significantly in incomplete datasets and result in erroneous conclusions. For instance, let us assume we want to derive a housing database in the US and have all neighborhoods listed in the corresponding table but in the apartments table we only have all entries for some states. The missing apartments could mainly be located in neighborhoods with high average rents and thus if we computed the average rent in a SQL query the result will be biased.

The only way to deal with missing tuples in a relational database today is to manually complete the data, which is very time-consuming and often not even possible. While there has been already significant work to impute missing values (e.g., replace a missing attribute) including learned approaches [146, 181, 187] or approaches to compensate for missing tuples in a single table [131], there is no system that can handle incomplete tables in a relational schema where tuples are missing and might introduce a bias.

This task can be tackled using ReStore, which follows a data-driven approach. The main idea is to use the complete tables in a database as evidence to synthesize the missing tuples even if the missing data introduces a bias in the incomplete table. For instance, in the housing database example, we would synthesize apartment tuples for the neighborhoods with potentially higher rents. Importantly, we exploit the data distribution in this synthesis process. For instance, if we know that in neighborhoods with a higher population density the rents tend to be higher, we will also synthesize apartment tuples with higher rents for such neighborhoods and thus compensate for the bias. Hence, we can again leverage the data distribution in the database to solve the database task. Specifically, we introduce neural completion models which are tailored to the completion of relational datasets. In the evaluation, we demonstrate that this helps to reduce the error of aggregate queries by up to 390% on real-world data. We detail the presentation of ReStore in Section 4.2 and the corresponding publication in Chapter 10.

Discussion

Overall, data-driven learning realizes the vision of solving database tasks without the expensive training data collection of workload-driven learning. Interestingly, the results are often even superior to

workload-driven alternatives, e.g., in cardinality estimation. To achieve this, in fact we only have to train models on samples of the database to capture the data distribution.

However, data-driven learning can only be applied to tasks where knowledge about the data distribution is sufficient. Most importantly, tasks that require information about the workload are not supported since it is crucial to observe query executions for such tasks (e.g., to solve cost estimation we have to observe queries to capture runtime characteristics of operators). Note that while data-driven learning cannot solve such tasks in isolation, it might still provide informative features. For instance, the intermediate cardinalities of a query are an important signal for cost predictions since they determine the size of intermediate joins, which often dominate the runtime. We will provide more details about this approach in the following chapter.

2.3 ZERO-SHOT LEARNED COMPONENTS

As a third contribution, we propose so-called zero-shot learned database components. As mentioned before, data-driven models eliminate the costly training data collection in the form of query executions but are not applicable to tasks that require information about the workload. In contrast, workload-driven models support a broader range of tasks (e.g., cost estimation) but are tied to a single database and thus the costly training data collection is repeated frequently. As such, with zero-shot learned components we strive to combine the benefits of data-driven learning and the broad applicability of workload-driven learning. We will first describe the general idea and discuss why workload-driven models cannot simply be adapted to obtain zero-shot models before we finally introduce our application of zero-shot cost estimation.

The general idea behind zero-shot learning for databases is motivated by recent advances in transfer learning such as GPT-3 [17]. In particular, zero-shot database components are pretrained *once* on a variety of different databases and thus allow the models to generalize to unseen databases out-of-the-box, i.e., without having to run additional training queries.

Unfortunately, it is not straightforward to derive zero-shot learned components from workload-driven counterparts. The reason is that the underlying assumption that the database is fixed is deeply embedded in such model architectures and it is thus not sufficient to pretrain such a model on several databases to obtain a zero-shot model. For instance, to featurize that a certain column is queried, workload-driven models typically one-hot encode the involved columns (by assigning a fixed position in a binary vector to each column) [79, 112, 161]. However, this featurization is inconsistent if applied to a different database since the columns, which are assigned the same position in the vector, could

have completely different data distributions or even data types. Hence, even though the featurization seems to be identical to the model, the actual encoded database is vastly different and thus the performance of the model will deteriorate. Hence, to design zero-shot models, we have to derive a representation of the database that is both informative enough to solve the problem at hand but still has similar semantics for different databases.

In our initial approach, we considered the task of cost estimation, i.e., predicting the query latency, which is important for query optimization, physical design tuning, etc. Traditionally, this task is approached using simplified cost models in the query optimizer [93]. However, these are known to be a bad proxy for actual execution time. Moreover, this task is non-trivial to solve due to complex interactions of operators in the physical execution plan such as caching effects [113]. While there has previously been research on learned cost estimation, existing approaches [112, 161] are workload-driven and thus the resulting models cannot be generalized across databases.

As a core contribution to enable zero-shot cost estimation, we thus introduce a model architecture that uses a representation of queries, which generalizes across databases. Specifically, we suggest modeling the query, for which we want to predict the costs along with the involved tables, columns and predicates as a graph, where the graph nodes are annotated with *transferable* features that can be derived from any database. This enables a consistent representation across databases and is the main prerequisite to enable a pretraining and generalization across databases. In the experimental evaluation, we demonstrate that zero-shot cost estimation enables accurate cost predictions for unseen databases without any training queries on the test database. In particular, workload-driven models require thousands of query executions as training data to provide a similar performance.

As mentioned before, data-driven models provide the intermediate cardinalities as additional features for the graph representation, which is an important signal to estimate the costs of potentially large joins. Hence, rather than providing an alternative, we believe that in many cases it can be beneficial to combine both zero-shot and data-driven learning. We provide more details on our zero-shot learning in Chapter 5 and the corresponding publication in Chapter 11.

3 DATA-EFFICIENT WORKLOAD-DRIVEN LEARNING

In this chapter, we summarize the first steps of this dissertation to alleviate the high costs of training queries for workload-driven models. Specifically, we suggest reducing the number of required training queries by *(i)* bootstrapping the components using simulation-based approaches and *(ii)* incorporating domain knowledge.

As the first contribution in Section 3.1, we introduce a simulation-based approach where in the first step a component learns the basic concepts by interacting with a simplified simulation model and is only afterwards further trained on actual query executions to learn effects not captured in the simulation. This requires fewer actually executed training queries since the model can already learn the basic tradeoffs from the simulation model. As a concrete task, we consider the database partitioning problem, which we will tackle using the derived data-efficient models.

As a second contribution, we propose to incorporate domain knowledge into the design of workload-driven components in Section 3.2. Specifically, we suggest implementing database components using differentiable programs with parameters that are learned by observing queries. Since differentiable programs come with significantly fewer learnable parameters compared to workload-driven models, less training data is required and thus fewer queries need to be observed.

Finally, we discuss the key findings in Section 3.3. The full details of the corresponding publications can be found in Chapters 7 and 8.

3.1 SIMULATION FOR DATA-EFFICIENT LEARNED PARTITIONING

As the first contribution of this dissertation, we suggest bootstrapping learned components using simulations to alleviate the high costs of training data collection. In particular, we will focus on the task of database partitioning, which is an important design problem for cloud data warehouses. We will first provide details on the problem of database partitioning in Section 3.1.1, which we will tackle using a data-efficient workload-driven approach (cf. Section 3.1.2), before we discuss the key findings in Section 3.1.4.

PUBLICATION The work on data-efficient learned partitioning is published in the peer-reviewed publication "Learning a Partitioning Advisor for Cloud Databases" in the *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020* [69], cf. Chapter 7.

CONTRIBUTIONS OF THE AUTHOR The contributions to the above publication by Benjamin Hilprecht, the author of this dissertation, are as follows. Benjamin Hilprecht is the leading author and was thus responsible for the proposed approach for the learned partitioning advisor, the experimental evaluation, and the manuscript. The co-authors Uwe Röhm and Carsten Binnig contributed invaluable feedback. All authors agree with the use of the publication for this dissertation.

3.1.1 *Partitioning as a Physical Design Problem*

We now first provide details on the problem of cloud database partitioning. As previously mentioned, software-as-a-service offerings in the cloud for data warehousing are becoming more and more popular. Using these services, customers can easily deploy a database, define their database schema, upload their data and then query the database using a cluster of machines. While most steps of the provisioning are automated, the partitioning, i.e., how tuples are physically distributed to nodes in a distributed database, often has to be chosen manually by customers in the cloud. For instance, in Azure's Data Warehouse but also in Amazon Redshift customers have to choose a partitioning attribute of a table to split large tables horizontally across multiple machines.

While optimally partitioning a database is a non-trivial task it has a significant impact on the overall performance. For example, analytical queries typically involve multiple joins over potentially large tables. If two tables are co-partitioned on the join attributes they can be joined locally on each node avoiding costly network transfers. Deciding for complex schemata with many tables and possible join paths which tables should be co-partitioned is a non-trivial task since this not only depends on the schema but also on other factors such as table sizes, the query workload (i.e., which joins are actually important and how often tables are joined), or hardware characteristics such as network speed and of course the database implementation itself.

There already exists a larger body of work to automate the physical design of distributed DBMSs including the data partitioning [3, 127, 145]. These advisors formalize the problem as an optimization problem and thus rely on cost models to estimate the runtime of queries for different partitionings. However, this approach is unsuitable for cloud providers: First, cloud providers typically allow customers to deploy their DBMS solutions on various hardware platforms which renders the problem of acquiring exact cost models a challenge on its own. Second, even if the cost model is tuned for a given hardware platform, optimizer cost estimates are still notoriously inaccurate [91] resulting in non-optimal partitioning designs if existing automated design approaches are used as we show in our experiments.

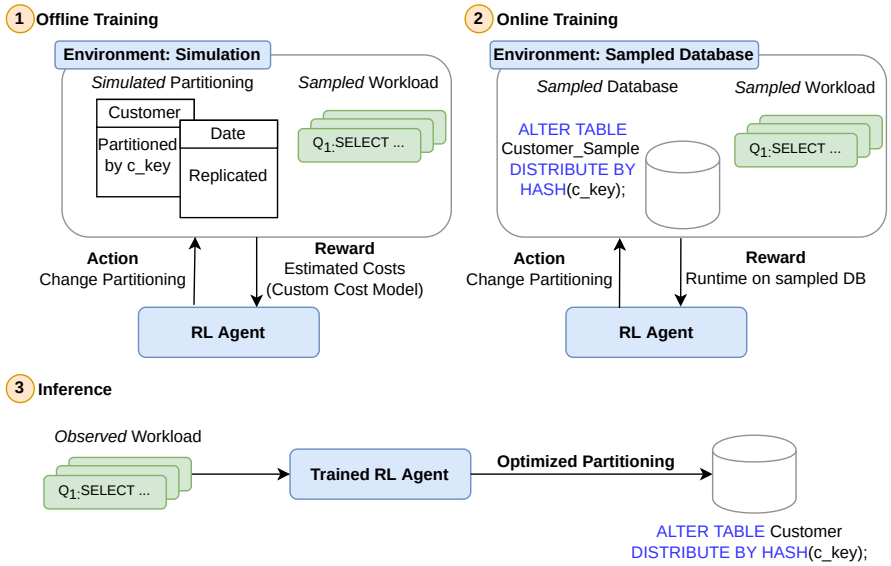


Figure 3.1: Overview of RL-based approach to Learn a Cloud Partitioning Advisor. First, the advisor is bootstrapped using a simulation model (1). In an optional online training phase (2), the RL-agent is fine-tuned using actual query executions on different partitioning. Finally, at inference time (3), the advisor can be used to suggest partitionings for unseen workload mixes.

3.1.2 *Towards a Learned Partitioning Advisor*

The basic idea of our approach is thus to train an RL agent for each cloud customer that learns the tradeoffs of different partitioning designs for a given database schema and different workloads by trial and error. Learning these tradeoffs is appealing since cost models are known to be notoriously inaccurate [91] and would thus over- or underestimate the benefits of certain partitionings. In contrast, RL agents learn by choosing actions and observing rewards, which they seek to maximize. In our setup, the environment is the DBMS which the agent manipulates with actions that change the partitioning of individual tables. During the training phase, the agent learns to minimize the runtime of a given workload consisting of a mix of representative queries. The agents thus learns the effects of different partitionings on individual query latencies.

Naïvely, we could train the agent on the customer database directly but this would require a high effort to collect the training data. For instance, repartitioning a large database table can take several minutes to complete. Unfortunately, during the training phase, the agent requires several of these actions to learn the effects. We therefore separate the training process into two phases: (1) *offline* and (2) *online* training. An overview of our approach is depicted in Figure 3.1. In the offline training phase, the agent solely interacts with a "simulation" of the customer database. Since the network is typically the bottleneck of distributed joins, we developed a simple yet generic cost model

focused on the network overhead required to answer a query given a certain partitioning. In combination with the metadata (schema and table sizes) about the customer database, we can estimate the query costs given a partitioning in our simulation. These estimates are used as rewards for the agent. Though not precise, this bootstraps the agent and enables it to already find a reasonable partitioning given a production workload (i.e., a mix of SQL queries). In our experiments, we show that an RL agent using this approach is already able to find partitionings that are on par with traditional optimization-based partitioning advisors that rely on DBMS internal cost models.

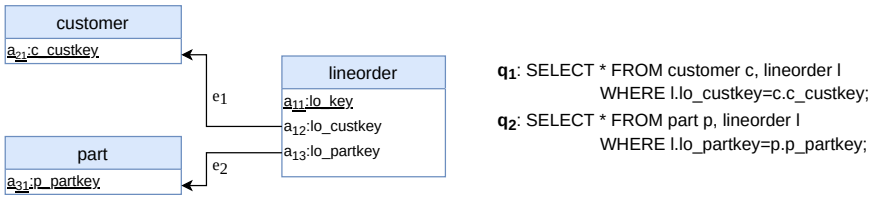
In an optional online training phase, the agent then does not just interact with a simulation but with a real database. However, instead of using the complete database we only use a sample of the data to speed-up this step of the training phase. The benefit of this phase is that it does not depend on the accuracy of our simple network-centric cost model anymore. Instead, we can simply measure the runtimes of queries on the sampled database to compute the rewards of the agent. Consequently, the agent learns the effects of partitionings more accurately.

Once the training is completed, we finally use the agent to make actual partitioning decisions. As input, it requires a workload, i.e., which queries were submitted in a certain time window. Based on this workload, the agent suggests partitionings, which we deploy on the actual customer database.

3.1.3 *Partitioning as an RL Problem*

We now show how the partitioning problem can be formulated as an RL problem including how we featurize the DBMS schema and a SQL workload. In order to formulate the partitioning problem as an RL problem, we model the database and the workload as state and possible changes in the partitioning as actions. Rewards correspond to the gain in performance for a given workload. During training, the agent thus learns the impact of different partitionings on the workload. Figure 3.2 shows an example of our encoding for a simple database with three tables and a workload with two queries. Before we introduce the details of our representation for the partitioning problem, we provide a short overview of Q-learning, which we use to realize the RL agent.

Q-LEARNING. A popular approach for RL is Deep Q-learning [115], where we strive to learn a function $Q(s, a)$ that approximates the future rewards if we choose the action a in state s. Intuitively, if we approximate the function correctly, we can in each state choose an optimal action by selecting $\text{argmax}_{a \in A} Q(s, a)$. However, during training, we also have to select random actions such that there is a tradeoff between exploration and exploitation of what we have learned



(a) Database and Workload

Foreign-Key Edges:
Edge e_1 for `lo_custkey`→ `c_custkey`: active
Edge e_2 for `lo_partkey`→ `c_partkey`: inactive
$s(E) = \left(e_1, e_2\right) = \left(1, 0\right)$

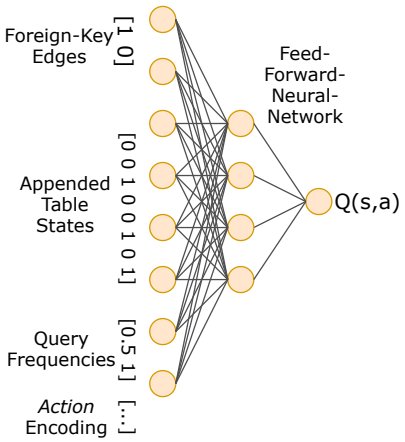
Table States:
`lineorder` partitioned by `lo_custkey`
$s(lineorder) = \left(r_1, a_{11}, a_{12}, a_{13}\right) = \left(0, 0, 1, 0\right)$

`customer` partitioned by `c_custkey`
$s(customer) = \left(r_2, a_{21}\right) = \left(0, 1\right)$

`part` replicated
$s(part) = \left(r_3, a_{31}\right) = \left(1, 0\right)$

Query Frequencies:
q_2 occurs twice as frequently as q_1
$s(Q) = \left(f_1, f_2\right) = \left(0.5, 1\right)$

(b) State Representation



(c) Q-Network with Encoded State

Figure 3.2: State Representation of Simplified SSB Schema and Workload. The current partitioning (a) is encoded by representing the current partitioning state for each table and binary edges denoting whether two tables are copartitioned for a particular join (b), while the workload is represented as the frequencies of each query in the workload mix. This representation is fed along with a potential action into a Q-network that predicts the Q-value, which can be used to decide on partitioning actions at inference time.

so far. Usually, exploration is realized by picking a random action with probability ε, which is decreased over time [164].

PARTITIONING STATE. The most important part of the state is to model a partitioning for a given database. For simplicity, we assume that only one partitioning scheme is used (e.g., hash-partitioning) that horizontally splits a table into a fixed number of shards (which is equal to the number of nodes in the database cluster). Moreover, replicated tables are also copied to all nodes in the cluster. In fact, these are the partitioning/replication options supported by the two DBMSs we used in our evaluation. However, in general, our approach can easily be extended to more complex partitioning schemes as well. Following the assumptions that a table T_i can either be replicated or alternatively partitioned by one of its attributes $a_{i1}, a_{i2}, \ldots, a_{in}$, we can encode the state as a binary vector using a one-hot encoding $s(T_i) = \left(r_i, a_{i1}, a_{i2}, \ldots, a_{in}\right)$, where r_i encodes whether a table is replicated and the remaining bits indicate whether an attribute is used for

partitioning. For instance, if the `part` table in Figure 3.2a is replicated, its state vector is $(r_3, a_{31}) = (1, 0)$ whereas the `customer` table is partitioned by the attribute a_{21} and the resulting vector is $(r_2, a_{21}) = (0, 1)$ (as shown in in Figure 3.2b).

To reduce the exploration of sub-optimal partitionings, we further extend the state representation making it explicit which tables are co-partitioned, i.e., the partitioning attributes of the tables match their join attributes. For instance, if the `customer` and `lineorder` table in Figure 3.2 are partitioned by the attributes `lo_custkey` and `c_custkey` respectively, we can join them locally on each node without shuffling. To explicitly encode co-partitioning we introduce the concept of edges; i.e., if an edge between a pair of join attributes a_{ir} and a_{js} of the corresponding tables T_i and T_j is activated, it guarantees co-partitioning. For instance, since the edge e_1 in Figure 3.2b is active the `customer` and `lineorder` tables are co-partitioned. The fixed set of possible edges E can easily be extracted from the given schema and workload (i.e., all possible join paths). Since every edge can either be active or inactive, the edge states can be represented as a fixed-size binary vector. To represent the features for the partitioning of a database with multiple tables as input for our Q-Network, we append the state vectors of all tables. For instance, the edge vectors and individual table vectors of Figure 3.2b are appended in Figure 3.2c and fed into the Q-network. Since this input is of fixed length, we are able to use a feed-forward neural network to predict the Q-value.

WORKLOAD STATE. Moreover, we need to model the workload as part of the state since for the same database schema, different workloads result in different partitioning strategies that should be selected. Formally, a workload is a set of SQL queries $Q_1, Q_2, \ldots, Q_n$. One way to model the workload is to encode each query using different one-hot encoded vectors, i.e., one vector for the set of tables, join predicates, where conditions, etc., similar to [79, 161]. However, this modeling approach assumes that only queries of a typical pattern occur (e.g., queries without nesting) and thus this approach is not suited for our approach since a partitioning advisor should be trained on arbitrary workloads where the query patterns are not known in advance and complex queries involving nested queries and complex predicate conditions appear.

Encoding nested queries with the featurization as proposed in [79, 161] would be in general possible but result in an overly complex encoding with many more input vectors and a neural network structure, which requires extensive training. However, a more complex encoding is still only able to represent a fixed class of queries. Moreover, more complex encodings typically require orders of magnitude more training data.

We thus take a different route to featurize the workload based on the observation that OLAP workloads are typically composed of complex but *recurring* queries. We assume that a representative set of possible queries q_i in a workload of queries Q is known in advance which is not uncommon in OLAP workloads. To encode a specific workload, we use a vector where an entry encodes the current normalized frequency f_i of a query q_i: $s(Q) = \left(f_1, \ldots, f_m\right)$. That way, the input state can represent different query mixes. For example, since the query q_2 occurs twice as often as the query q_1 the frequency vector becomes $(0.5, 1)$ in Figure 3.2b.

Moreover, completely new queries can be supported in our state encoding without the need to train a new RL agent from scratch. One case that we typically see in analytical workloads is that the same query is used with different parameter values resulting in different selectivities. In order to support this case, we bucketize queries into classes with different selectivity ranges and use different entries in $s(Q)$; i.e., one for each bucket. That way, if a query is used with a new set of parameter values, it is supported by finding the corresponding entry in $s(Q)$ and increasing the query frequency f_i. For supporting completely new SQL queries and not just new parameter values of existing queries in the workload, we provide entries in $s(Q)$ that are initially set to 0 (i.e., no query of this type occurs in the workload) and use those entries for new queries if they occur.

ACTIONS. A small state space is essential to apply Q-learning because we have to compute the Q-values for all possible actions to decide which action to execute in a state. We designed the actions to affect at most the partitioning of a single table. More precisely, we support two types of actions: (1) partitioning a table by an attribute or (2) replicating a table. During training, the RL agent can only select one of these actions at each step. This reduces the repartitioning costs during training since similar partitionings are observed successively.

In addition, we provide an action for (de-)activating edges as a shortcut to change the partitioning. Intuitively, activating an edge co-partitions two tables while the de-activation of edges allows follow-up actions to choose a new strategy (e.g., replication discussed above). It is important that the set of edges to be activated is conflict-free. For this, we solely allow activating an edge if there are no two edges, which requires a table T_i to be partitioned by different attributes a_{ir} and $a_{ir'}$. For example, edge e_2 cannot be activated in Figure 3.2 because e_1 is already active. First, the conflicting edge e_1 would have to be deactivated.

An action a is encoded similarly to the partitioning and workload state: we use appended one-hot encoded vectors to capture the information required for an action, i.e., the kind of action (replicate, partition, (de-) activate an edge, etc.), the affected table and attribute

as well as the (de-)activated edge. Both the state s and an action a are then used as input for the neural network to predict the Q-value $Q(s, a)$.

REWARDS. The overall goal of the learned advisor is to find a partitioning that minimizes the runtime for the workload mix (queries and their frequencies) modeled as part of the input state. This objective has to be minimized by the RL agent and can be used as a reward. Estimates of the simple network-centric cost model $c_m(P, q_i)$ for the queries q_i given a partitioning P are used for the offline training and actual runtimes $c_r(P, q_i)$ for the online training. Since the RL agents seeks to maximize the reward, we use negative costs in the reward definition resulting in $r = -\sum_{j=1}^{m} f_j c(P, q_j)$.

We decided to exclude the costs of repartitioning the database as rewards in our learning procedure since we aim for setups where we expect that repartitioning does not happen that often and can be executed in the background, especially for OLAP workloads and thus does not have a negative effect on the actual workload execution. In case repartitionings should be used more frequently, these costs should be included in the rewards to prefer repartitionings that can be applied with less cost.

3.1.4 *Key Findings*

We now present the key findings of the experimental evaluation. In particular, we will study whether *(i)* the RL-based approach suggests competitive partitionings and *(ii)* whether the offline phase using the simulation model indeed reduces the number of required training queries. Note that more details on the proposed RL approach as well as a more extensive empirical evaluation including additional benchmarks (SSB and TPC-DS) can be found in Chapter 7.

For the evaluation, we first trained the RL agent in the offline training phase using our simplified cost model before we further fine-tuned it using actual workload executions on actual partitionings in the online training phase. We then use both the trained RL agent as well as the baselines to suggest a partitioning and compare the total workload runtime. As dataset and workload, we utilized the challenging analytical queries of the TPC-CH benchmark [47]. As baselines, we first evaluate heuristics, which are commonly used in practice to select a suitable partitioning. In addition, we also select the partitioning that minimizes the optimizer cost estimates, which would be chosen by state-of-the-art automated approaches [3, 127, 145]. To also quantify the benefit of the online phase, we in addition evaluate the partitioning that is suggested by an RL agent that is only trained offline, i.e., with the simulation model only.

As we can see in Figure 3.3, the partitioning suggested by the online-trained agent is 20% superior to the partitioning of the offline-trained

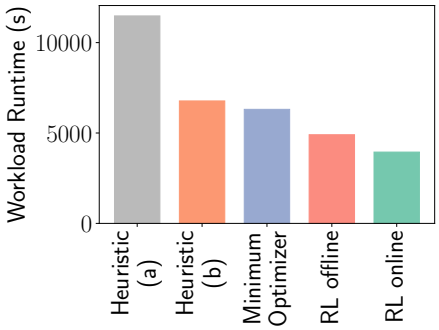


Figure 3.3: TPC-CH Runtimes for Partitionings found by both our RL Approach and Baselines. The RL-approaches outperform the baselines as well as the optimal partitioning suggested by the optimizer cost model. The optional online phase further improves the RL-agent, which suggests an even more suitable partitioning afterwards.

agent and also significantly outperforms both the heuristics and the partitioning identified by minimizing the optimizer costs. The reason is that the baselines either rely on heuristics or optimizer costs, which cannot tradeoff the impact of different partitionings accurately enough. Analogously, the offline trained RL agent is affected by inaccuracies of our simple network-centric cost model and thus does not select the optimal partitioning. In contrast, the online phase is not affected by the inaccuracy of our simulation model.

Finally, we want to quantify the data-efficiency of the proposed RL approach for partitioning. The online-phase with all optimizations and for a model that was bootstrapped offline took 13.3 hours of executed workloads. In contrast, training an agent from scratch without the offline phase (but all remaining optimizations suggested in Chapter 7) would require 33.4 hours. Hence, the simulation-based approach indeed improves the data-efficiency of workload-driven models. We believe that a training time of several hours is acceptable since the model has to be trained only once for different workload mixes and can afterwards be used as a partitioning advisor if the workload changes. Moreover, especially in cloud setups, we can easily clone the instances. Hence, setting up a similar cluster to retrain the agent for several hours to obtain a refined model should be feasible considering that customers usually have one cluster provisioned all the time to do analytics.

We can thus conclude that simulation can indeed improve the data-efficiency of workload-driven models while still solving the database task of the component effectively.

3.2 DIFFERENTIABLE DATABASES

As a second contribution of this dissertation, we suggest incorporating domain knowledge into workload-driven components to reduce the

high costs of training queries. Specifically, we propose to implement database components as differentiable programs with free parameters that can be optimized for a specific database instance. This yields so-called FITable DBMSs, which we will introduce in Section 3.2.1. We will then present the key findings in Section 3.2.2.

PUBLICATION The work on FITable DBMSs is published in the peer-reviewed publication "DBMS Fitting: Why should we learn what we already know?" in the *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings* [67], cf. Chapter 8. The source code and data [68] are publicly accessible for reproducibility and future research.

CONTRIBUTIONS OF THE AUTHOR The contributions to the above publication by Benjamin Hilprecht, the author of this dissertation, are as follows. Benjamin Hilprecht is the leading author and was thus responsible for the idea of the proposed approach as well as the implementation and evaluation of the fittable cost models, and the respective parts of the manuscript. The co-authors Carsten Binnig, Tiemo Bang, Muhammad El-Hindi, Benjamin Hättasch, Aditya Khanna, Robin Rehrmann, Uwe Röhm, Andreas Schmidt, Lasse Thostrup, and Tobias Ziegler contributed the implementation of the actual pipelines, for which the costs were modeled in the experiments, the remaining parts of the manuscript, as well as invaluable feedback and discussions. All authors agree with the use of the publication for this dissertation.

Listing 3.1: Fittable Function for Simple Cost Model

```
# table-size = size in Byte / no-tuples = number of tuple in table
def cost_scan_op(params, table_size, no_tuples):
        # piecewise linear model
        if table_size< params['cache-size']:
                slope = params['a_in']
                intercept = params['b_in']
                cost_per_tuple = slope * table_size + intercept
        else:
                slope = params['a_out']
                intercept = params['b_out']
                cost_per_tuple = slope * table_size + intercept

        return no_tuples * cost_per_tuple
```

3.2.1 *FITable DBMSs*

The vision of a FITable DBMSs is that *DBMS components (or parts of them) are implemented as differentiable functions* that allow us to adapt the behavior of the component to optimally support a concrete workload and hardware. For instance, a simplified cost model to estimate the execution time of a scan operator in a main-memory DBMS can be modeled as a differentiable function `cost_scan_op` as shown in

Listing 3.1. The main idea of this function is that the costs for reading a tuple depend on the table size which can be represented by a piece-wise linear function using two segments for tables that fit into the cache and for those, which spill out of the cache.

The main benefit of fittable code is that it not only leverages the domain knowledge of the developer (e.g., that the tuple-access cost can be modeled as a piece-wise linear function in our example) but more importantly that the concrete behavior can be fitted automatically to the actual behavior.

The fittable part of the code is captured by parameters that can be learned from concrete behavior. In our example, the learnable parameters are the slope (i.e., `params['a_in']` and `params['a_out']`) and intercept (i.e., `params['b_in']` and `params['b_out']`) of both segments. For fitting the cost model, the actual costs of running the scan operator on different table sizes need to be collected. Since functions are differentiable, normal gradient-based optimization can be used to fit the parameters (i.e., minimizing the error of the cost function) as shown in Figure 8.2. Once the parameters are fitted they can be used at runtime of a DBMS, just like fully specified source code.

The power of differentiable programming stems from the fact that the database developer does not have to come up with the gradients herself. Instead, frameworks such as Autograd[1] support automatic differentiation [176] of ordinary code, which may contain all the usual control structures, including loops, if statements, recursion, and closures. In our example, the code for the cost function in Listing 3.1 is implemented using a normal if-else control flow that can be differentiated automatically.

Overall, fittable code in contrast to black-box DNN models thus provides many advantages: First, fittable code is more *data-efficient*, i.e. we require much less training data since the differentiable function already defines the basic shape of a function that needs to be learned. Furthermore, fitting a differentiable function does not always need to rely on gradient-based methods that typically require multiple passes over the training data. Instead, it can often be implemented by computationally much simpler approaches that only require a single pass [48]. Second, fittable functions typically *generalize* better and are less susceptible to small changes in the input, since they already define a reasonable behavior based on their shape. Finally, fitted code is *explainable and debuggable*. If the behavior is unexpected, the developer can debug the DBMS code (as usual) since the general code structure reflecting the domain knowledge is still interpretable and remains unchanged.

1 https://github.com/HIPS/autograd

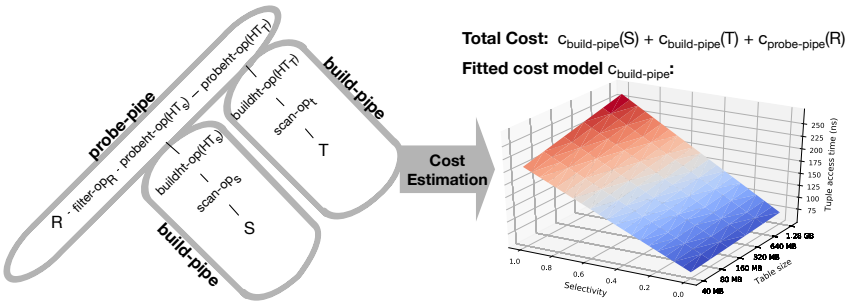


Figure 3.4: Basic idea of our fittable cost model - The total cost of a query plan is estimated based on fitted cost models for each pipeline type. In this example, the build-pipeline type is used in two instantiations over tables S and T and the probe-pipeline type is used in one instantiation over table R, which probes into the hash tables HT_S and HT_T, created by the other two pipelines. The cost models for each pipeline type are based on general features of a pipeline, such as the size of the input table, tuple-width, selectivity of operators, etc.

3.2.2 *Key Findings for Fittable Cost Models*

As a concrete use case to validate the vision of FITable DBMSs, we consider the task of cost estimation, i.e., predicting the query latency. The main idea of fittable cost models is to encode knowledge about the general shape of cost functions for individual operators. We will now provide more details on the evaluation with a particular emphasis on the data-efficiency of the approach. A more extensive description of the approach, as well as additional experiments, can be found in Chapter 8.

The model is targeted towards DBMSs that execute SQL queries in a pipelined manner, which is the case for most commercial DBMSs that either implement a classical iterator model (for individual tuples or blocks of tuples) or DBMSs that rely on pipeline-based code generation for query execution, such as Hyper. In order to estimate the execution time of complete query plans, the model estimates the costs of each pipeline and then aggregates the cost to compute the total cost of that query plan. The core components of our model are thus fitted cost models that we use to estimate the costs of individual pipelines (cf. Figure 3.4). For the actual fitting of the cost models of the different pipeline types, we collect the actual runtime for a variety of pipeline instances for a given hardware platform. We use this collected training data for gradient-based optimization to fit the cost model and learn the parameters of pipelines end-to-end.

In an initial experiment, we now demonstrate the data-efficiency of fittable cost models. In particular, we train both our fittable cost model as well as a state-of-the-art workload-driven approach for cost

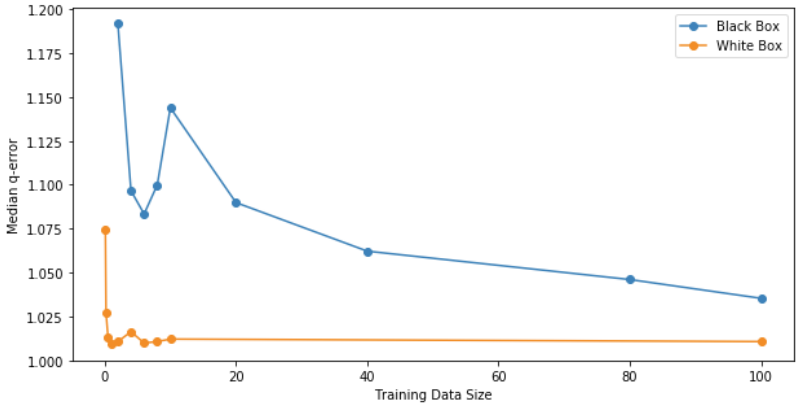


Figure 3.5: Exp. 2 - Data-efficiency of our fittable cost model. This plot shows the result for the scan-pipeline comparing the median q-error of our model-based (white-box) to a DNN-based model (black-box) based on [161], when using only x% of the original training data.

estimation [161] with a varying number of training queries for a scan-pipeline. Note that we conducted more extensive experiments on additional pipelines in Chapter 8. We then compare how accurately the models can predict the actual latencies for the pipeline using unseen configurations, i.e., on different datasets. In particular, we report the commonly used median q-error [79], which denotes the factor the estimation differs from the actual runtime in the median, i.e., a q-error of one would be a perfect estimation and more inaccurate estimates yield higher q-errors.

The results for learning the cost model for simple query plans on a single table are shown in Figure 3.5. We can see that our white-box model can already achieve a low q-error with only 5% of the training data. In contrast, the black-box model requires much more training data to achieve a low q-error even for these simple queries. More interestingly, if we provide the full training data to the black-box model, it is not able to reach the same accuracy that our white-box model achieves with only 5% of the training data.

Finally, we can thus conclude that differentiable programming can indeed significantly reduce the number of required training data queries of workload-driven models and still provide a competitive accuracy.

3.3 DISCUSSION

As demonstrated in the previous sections, the proposed contributions can in fact reduce the required number of training queries for workload-driven learning significantly. In particular, for the simulation-based approach, we required 2.5x fewer training query executions and orders of magnitude less training data for the differentiable programming-based approach for cost estimation.

However, these savings come at the cost of an increased engineering effort since the basic effects of the tasks have to be explicitly encoded by developers either in the simulation model (for the simulation-based approach) or in the differentiable program (in FITable DBMSs). While in the first case, an imprecise simulation can still be compensated by more query executions at the cost of a more expensive online training phase, for FITable DBMSs the complete behavior of the component has to be explicitly encoded. For instance, for more broadly applicable fittable cost models we would also have to manually encode complex caching effects or interactions among queries for inter-query parallelism which is hard to model explicitly. Hence, overall the savings for a new unseen database depend on the nature of the underlying task, i.e., whether the basic tradeoffs are known and can be encoded. For some problems, it might even be impossible to explicitly model all underlying effects and thus differentiable programming would not even be applicable.

Finally, while the training queries can be reduced, we still require query executions to gather training data for an unseen database. For instance, for the learned partitioning advisor, we still required 13.3 hours of observed workload to train a model for the TPC-CH benchmark. In the case of cloud vendors, these might still be unacceptably high costs since the query executions would be incurred not only for every new customer but even in case of significant updates for a single customer. This motivates the further directions proposed in this dissertation.

4 DATA-DRIVEN LEARNING

Data-efficient workload-driven learning does not fully eliminate the need for training queries for unseen databases. This motivates the second direction of this dissertation of data-driven learning, which is a new paradigm for learned database components. In particular, the idea is to only learn the data distribution by training on samples of the original database instead of running a representative training workload. The data distribution is then used at runtime for the task at hand.

This chapter summarizes the contributions of this dissertation towards data-driven learning. First, in Section 4.1, we will introduce our system DeepDB, which outperforms the state-of-the-art for cardinality estimation and AQP without requiring any training queries. Second, in Section 4.2, we will demonstrate that data-driven learning is more broadly applicable. In particular, we will consider analytical queries over incomplete relational datasets, which can be biased due to incompleteness. We will introduce our system ReStore that leverages the data distribution to compensate for this bias.

Finally, we discuss our results in Section 4.3. The full details on the corresponding publications can be found in Chapters 9 and 10.

4.1 DEEPDB: DATA-DRIVEN LEARNING FOR CARDINALITY ESTIMATION AND AQP

As a first contribution towards data-driven learning, we introduce our system DeepDB, which supports both cardinality estimation and AQP solely by learning from the data instead of a representative workload. We will first provide an overview of the system and discuss applications in Section 4.1.1 before we discuss details on the model architecture in Section 4.1.2 and present the key findings in Section 4.1.3. We provide more details as well as more experiments on updates and AQP in Chapter 9.

PUBLICATION The work on data-efficient learned partitioning is published in the peer-reviewed publication "DeepDB: Learn from Data, not from Queries!" in the *Proc. VLDB Endow.* [72], cf. Chapter 9. The source code and data [71] are publicly accessible for reproducibility and future research.

CONTRIBUTIONS OF THE AUTHOR The contributions to the above publication by Benjamin Hilprecht, the author of this dissertation, are

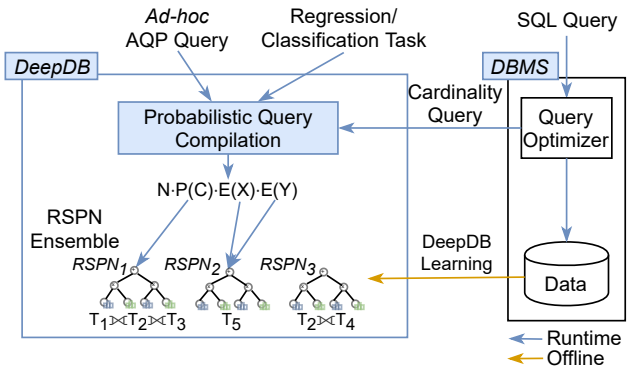


Figure 4.1: Overview of DeepDB. RSPNs capture the data distribution present in the database at hand. At runtime, these can be used to solve tasks such as cardinality estimation or AQP. Probabilistic query compilation allows to flexibly combine several models at runtime to handle such queries for ad-hoc join paths.

as follows. Benjamin Hilprecht is the leading author and was thus responsible for the proposed approach for data-driven learning for cardinality estimation and AQP, the experimental evaluation, and the manuscript. Andreas Schmidt contributed the implementation of updates of RSPNs. Note that preliminary version work covering only AQP on a single table was published as a workshop paper [86]. Hence, the author of this dissertation had to introduce significant new contributions to support arbitrary joins over relational datasets, cardinality estimation and also extensions of SPNs yielding RSPNs, which are the main contributions of the full paper. The co-authors Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig contributed invaluable feedback. All authors agree with the use of the publication for this dissertation.

4.1.1 *Overview and Applications*

OVERVIEW As shown in Figure 4.1, the main idea of DeepDB is to learn a representation of the data offline. An important aspect of DeepDB is that we do not aim to replace the original data with a model. Instead, a model in DeepDB augments a database similar to indexes to speed-up queries and to provide additional query capabilities while we can still run standard SQL queries over the original database.

To optimally capture relevant characteristics of relational data in DeepDB, we developed a new class of models called *Relational Sum Product Networks* (RSPNs). In a nutshell, RSPNs are a class of deep probabilistic models that capture the joint probability distribution over all attributes in a database that can then be used at runtime to provide the answer for different user tasks.

While RSPNs are based on Sum Product Networks (SPNs) [117, 141], there are significant differences: (1) While SPNs support only single tables and simple queries (i.e., no joins and no aggregation functions),

RSPNs can be built on arbitrary schemata and support complex queries with multi-way joins and different aggregations (COUNT, SUM, AVG). Moreover, RSPNs also go beyond the idea of other recent learned data models that need to know join paths a priori such as [105, 186] since RSPNs allow true ad-hoc joins by combining RSPN models. (2) Another major difference is that RSPNs support direct updates, i.e., if the underlying database changes the RSPN can directly ingest the updates without the need to retrain the model. (3) RSPNs also include a set of database-specific extensions such as NULL-value handling and support for functional dependencies.

Once the RSPNs are created offline, they can be leveraged at runtime for a wide variety of different applications, ranging from user-facing tasks (e.g., to provide fast approximate answers for SQL queries) to system-internal tasks (e.g., to provide estimates for cardinalities).

In order to support these tasks, DeepDB provides a new so-called *probabilistic query compilation* procedure that translates a given task into evaluations of expectations and probabilities on RSPNs. A key difference of DeepDB in contrast to previous approaches, which model the data distribution [105, 186], is that it supports ad-hoc joins in the schema, i.e., we do not require a single model per join path but can combine models flexibly at runtime. Our probabilistic query compilation is thus crucial to support larger schemas and was used by many follow-up publications [185, 195]. We now give a brief overview of the applications currently supported by the query compilation engine of DeepDB.

CARDINALITY ESTIMATION The first task DeepDB supports is cardinality estimation for a query optimizer. Cardinality estimation is needed to provide cost estimates but also to find the correct join order during query optimization. A particular advantage of DeepDB over existing learned approaches for cardinality estimation [79, 161] is that we do not have to create dedicated training data, i.e. pairs of queries and cardinalities. Instead, since RSPNs capture the characteristics of the data independent of a workload, we can support arbitrary join queries without the need to train a model for a particular workload. Moreover, RSPNs can be kept up to date at low costs similar to traditional histogram-based approaches, which is different from other workload-driven learned approaches for cardinality estimation such as [79, 161] which require retraining.

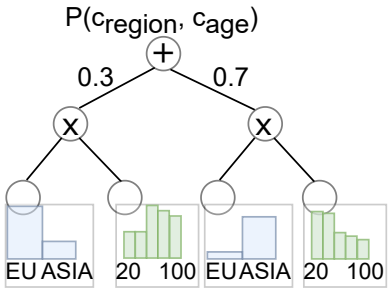
APPROXIMATE QUERY PROCESSING (AQP) The second task we currently support in DeepDB is AQP. AQP aims to provide approximate answers to support faster query response times on large datasets. The basic idea of how a query on a single table is executed inside DeepDB is simple: for example, an aggregate query $\texttt{AVG}(X)$ with a where condition C is equal to the conditional expectation $\mathbb{E}(X \mid C)$ which can be approximated with RSPNs. In DeepDB, we implement a

c_id	c_age	c_region
1	80	EU
2	70	EU
3	60	ASIA
4	20	EU
...	...	...
998	20	ASIA
998	25	EU
999	30	ASIA
1000	70	ASIA

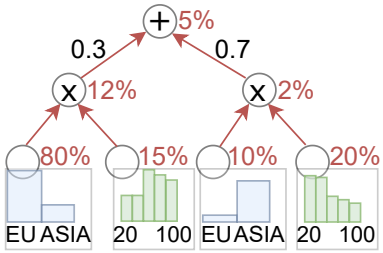
(a) Example Table

c_age	c_region
80	EU
70	EU
60	ASIA
20	EU
...	...
...	...
20	ASIA
25	EU
30	ASIA
70	ASIA

(b) Learning with Row/ Column Clustering



(c) Resulting SPN



(d) Probability of European Customers younger than 30

Figure 4.2: Customer Table and corresponding SPN. Recursive row and column clusterings (b) yield sum and product nodes in the resulting SPN (c), which can afterwards be used to compute probabilities and expectations over predicates on arbitrary columns.

more general AQP procedure that leverages the fact that RSPNs can support joins of multiple tables. A major difference to other learned approaches for AQP such as [105, 167] is again that DeepDB supports ad-hoc queries and is thus not limited to the query types covered by the training set.

4.1.2 *Learning a Deep Data Model*

In this section, we introduce Relational Sum Product Networks (RSPNs), which we use to learn a representation of a database and, in turn, to answer queries using our query engine explained in the next section. We first review Sum Product Networks (SPNs) and then introduce RSPNs. Afterwards, we describe how an ensemble of RSPNs can be created to encode a given database multiple tables.

4.1.2.1 *Sum Product Networks*

Sum-Product Networks (SPNs) [141] learn the joint probability distribution $P(X_1, X_2, \ldots, X_n)$ of the variables X_1, X_2, ..., X_n in the dataset.

They are an appealing choice because probabilities for arbitrary conditions can be computed very *efficiently*. We will later make use of these probabilities for our applications like AQP and cardinality estimation.

For the sake of simplicity, we restrict our attention to Tree-SPNs, i.e., trees with sum and product nodes as internal nodes and leaves. Intuitively, sum nodes split the population (i.e., the rows of dataset) into clusters and product nodes split independent variables of a population (i.e., the columns of a dataset). Leaf nodes represent a single attribute and approximate in the present paper the distribution of that attribute either using histograms for discrete domains or piecewise linear functions for continuous domains [118]. For instance, in Figure 4.2c, an SPN was learned over the variables *region* and *age* of the corresponding *customer* table in Figure 4.2a. The top sum node splits the data into two groups: The left group contains 30% of the population, which is dominated by older European customers (corresponding to the first rows of the table), and the right group contains 70% of the population with younger Asian customers (corresponding to the last rows of the table). In both groups, region and age are independent and thus split by a product node each. The leaf nodes determine the probability distributions of the variables *region* and *age* for every group.

Learning SPNs [50, 118] works by recursively splitting the data into different clusters of rows (introducing a sum node) or clusters of independent columns (introducing a product node). For the clustering of rows, a standard algorithm such as *KMeans* can be used or the data can be split according to a random hyperplane. To make no strong assumptions about the underlying distribution, Randomized Dependency Coefficients (RDCs) are used for testing the independence of different columns [101]. Moreover, independence between all columns is assumed as soon as the number of rows in a cluster falls below a threshold n_{min}. As stated in [117, 141], SPNs in general have a polynomial size and allow inference in linear time w.r.t. the number of nodes. However, for the configurations we use in our experiments, we can even bound the size of the SPNs to linear complexity w.r.t. the number of columns in a dataset since we set $n_{min} = n_s/100$ (i.e. relative to the sample size), which turned out to be a robust configuration.

With an SPN at hand, one can compute probabilities for conditions on arbitrary columns. Intuitively, the conditions are first evaluated on every relevant leaf. Afterwards, the SPN is evaluated bottom up. For instance in Figure 4.2d, to estimate how many customers are from Europe and younger than 30, we compute the probability of European customers in the corresponding blue *region* leaf nodes (80% and 10%) and the probability of a customer being younger than 30 (15% and 20%) in the green *age* leaf nodes. These probabilities are then multiplied at the product node level above, resulting in probabilities of 12% and 2%, respectively. Finally, at the root level (sum node), we have to consider the weights of the clusters, which leads to $12\% \cdot 0.3 +$

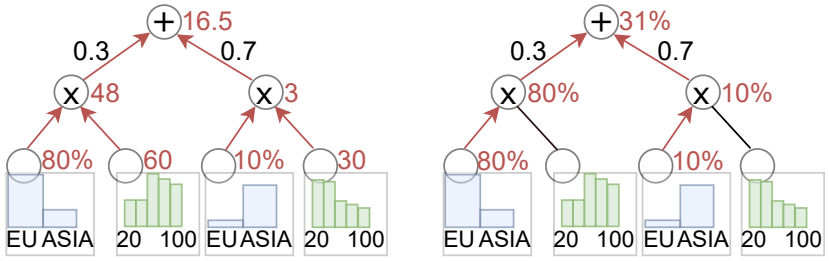
$2\% \cdot 0.7 = 5\%$. Multiplied by the number of rows in the table, we get an approximation of 50 European customers who are younger than 30.

4.1.2.2 *Relational Sum-Product Networks*

One important issue with SPNs is that they can only capture the data of single tables but they also lack other important features needed for DeepDB. To alleviate these problems, we now introduce RSPNs.

EXTENDED INFERENCE ALGORITHMS The first and most important extension is that for many queries such as AVG and SUM expectations are required (e.g., to answer a SQL aggregate query, which computes an average over a column). In order to answer these queries, RSPNs allow computing expectations over the variables on the leaves to answer those aggregates. To additionally apply a filter predicate, we still compute probabilities on the leaves for the filter attribute and propagate both values up in the tree. At product nodes, we multiply the expectations and probabilities coming from child nodes whereas on sum nodes the weighted average is computed. In Figure 4.3, we show an example of how the average age of European customers is computed. The ratio of both terms yields the correct conditional expectation. A related problem is that SPNs do not provide confidence intervals. We also developed corresponding extensions on SPNs.

DATABASE-SPECIFICS Finally, SPNs lack support for important database specifics: (1) First, SPNs do not provide mechanisms for handling NULL values. Hence, we developed an extension where NULL values are represented as a dedicated value for both discrete and continuous columns at the leaves during learning. Furthermore, when computing conditional probabilities and expectations, NULL values must be handled according to the three-valued logic of SQL. (2) Second, SPNs aim to generalize the data distribution and thus approximate the leaf distribution, abstracting away specifics of the dataset to generalize. For instance, in the leaf nodes for the age in Figure 4.2c, a piecewise linear function would be used to approximate the distribution [118]. Instead, we want to represent the data as accurately as possible. Hence, for continuous values, we store each individual value and its frequency. If the number of distinct values exceeds a given limit, we also use binning for continuous domains. (3) Third, functional dependencies between non-key attributes $A \rightarrow B$ are not well captured by SPNs. We could simply ignore these and learn the RSPN with both attributes A and B, but this often leads to large SPNs since the data would be split into many small clusters (to achieve independence of A and B). Hence, we allow users to define functional dependencies along with a table schema. If a functional dependency $A \rightarrow B$ is defined, we store the mapping from values of A to values of B in a separate dictionary of the RSPN and omit the column B when



(a) $\mathbb{E}(\texttt{c\_age} \cdot 1_{\texttt{c\_region='EU'}})$ (b) $P(\texttt{c\_region='EU'})$

Figure 4.3: Process of computing $\mathbb{E}(\texttt{c\_age} \mid \texttt{c\_region='EU'})$. The predicates are pushed down to the leaf nodes and in a bottom-up pass probabilities are either combined by multiplication (product nodes) or weighted summation (sum nodes).

learning the RSPN. At runtime, queries with filter predicates for B are translated to queries with filter predicates for A.

UPDATABILITY Finally, a last important extension of RSPNs over SPNs is the direct updatability of the model. If the underlying database tables are updated, the model might become inaccurate. For instance, if we insert more young European customers in the table in Figure 4.2a, the probability computed in Figure 4.2d is too low and thus the RSPN needs to be updated. As described before, an RSPN consists of product and sum nodes, as well as leaf nodes, which represent probability distributions for individual variables. The key idea to support direct updates of an existing RSPN is to traverse the RSPN tree top-down and update the value distribution of the weights of the sum-nodes during this traversal. For instance, the weight of a sum node for a subtree of younger European customers could be increased to account for updates. Finally, the distributions in the leaf-nodes are adjusted.

4.1.3 *Key Findings*

We now present the key findings of the experimental evaluation. In particular, we will demonstrate that DeepDB outperforms the state-of-the-art cardinality estimation approaches while at the same time not requiring any training queries. Note that an extensive evaluation including update, generalization, and AQP experiments can be found along with more details on our approach in Chapter 9.

To evaluate the cardinality estimation performance, we chose the JOB-light benchmark [79, 91], which is more challenging than traditional benchmarks such as TPC-H since the underlying dataset comes with complex correlations, which also make cardinality estimation hard in practice. As baselines, we used the following learned and traditional approaches: First, we trained a Multi-Set Convolutional Network (MSCN) [79] as a learned baseline, which is a state-of-the-art workload-driven approach. As a representative of a synopsis-based technique, we implemented an approach based on wavelets [21]. Fi-

Table 4.1: Estimation Errors for the JOB-light Benchmark. DeepDB offers orders of magnitude more accurate cardinality estimates both in the median but also in the tail performance.

	median	90th	95th	max
DeepDB	1.34	2.50	3.16	39.63
MCSN	3.22	65	143	717
Wavelets	7.64	9839	15332	564549
Postgres	6.84	162	817	3477
IBJS	1.67	72	333	6949
Random Sampling	5.05	73	10371	49187

nally, we use the standard cardinality estimation of Postgres 11.5 as well as online random sampling and Index-Based Join Sampling (IBJS) [92] as non-learned baselines.

ESTIMATION QUALITY The resulting cardinality estimation accuracies are depicted in Table 4.1. The prediction quality of cardinality estimators is usually evaluated using the q-error [79, 116], which is the factor by which an estimate differs from the real execution join size. For example, if the real result size of a join is 100, the estimates of 10 or 1k tuples both have a q-error of 10. In Table 4.1, we depict the median, 90-th and 95-th percentile and max q-errors for the JOB-light benchmark of our approach compared to the other baselines. As we can see DeepDB outperforms the best competitors often by orders of magnitude. While IBJS provides a low q-error in the median, the advantage of learned MCSNs is that they outperform traditional approaches by orders of magnitude for the higher percentiles and are thus more robust. DeepDB not only outperforms IBJS in the median, but provides additional robustness having a 95-th percentile for the q-errors of 3.16 vs. 143 (MCSN). The q-errors of both Postgres and random sampling are significantly larger both for the medians and the higher percentiles. Finally, wavelets have the highest error since they suffer from the curse of dimensionality.

TRAINING OVERHEAD In contrast to other learned approaches for cardinality estimation [79, 161], no dedicated training data is required for DeepDB. Instead, we just learn a representation of the data. The training of the base ensemble including all data preparation steps takes 48 minutes. In contrast, for the MCSN [79] approach, 100k queries need to be executed to collect cardinalities resulting in 34 hours of training data preparation time (when using Postgres). Moreover, the training of the neural network takes only about 15 minutes on a Nvidia V100 GPU. As we can see, our training time is much lower since we do not need to collect any training data for the workload. Another advantage is that we do not have to re-run the queries once the database is modified.

Finally, we can conclude that DeepDB indeed eliminates the high costs for training data collection of workload-driven approaches and still provides more accurate cardinality estimates. In Chapter 9, we will demonstrate that a similar observation holds for AQP and that DeepDB supports efficient updates in addition.

4.2 RESTORE: DATA-DRIVEN COMPLETION OF INCOMPLETE RELATIONAL DATASETS

As a second application of data-driven learning, we tackle the problem of data completion for incomplete relational datasets. Our goal is to alleviate potential biases due to the incompleteness of the data to enable users to still draw conclusions if only incomplete data is available. The idea is to again only exploit the data distribution to solve this task yielding a data-driven learned component. We will first provide a detailed problem statement for the underlying task and afterwards give an overview of our system called ReStore in Section 4.2.1.1. We will finally discuss the key findings in Section 4.2.3.

PUBLICATION The work on data-driven completion of incomplete relational datasets is published in the peer-reviewed publication "ReStore - Neural Data Completion for Relational Databases" in *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021* [63], cf. Chapter 10. The source code and data [64] are publicly accessible for reproducibility and future research.

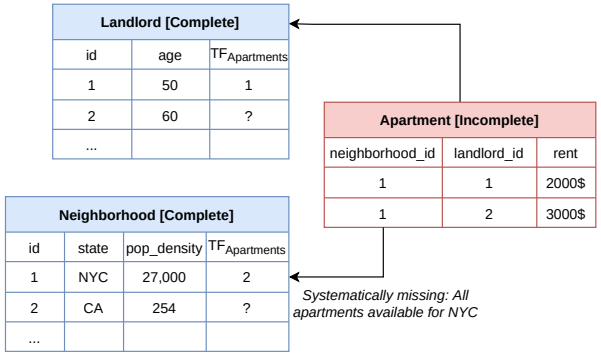
CONTRIBUTIONS OF THE AUTHOR The contributions to the above publication by Benjamin Hilprecht, the author of this dissertation, are as follows. Benjamin Hilprecht is the leading author and was thus responsible for the proposed approach for relational data completion, the experimental evaluation, and the manuscript. The co-author Carsten Binnig contributed invaluable feedback and agrees with the use of the publication for this dissertation.

4.2.1 *Overview*

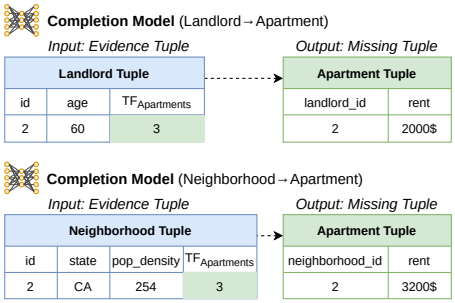
In this section, we introduce the problem statement before we give an overview of our approach and discuss the general assumptions.

4.2.1.1 *Problem Statement*

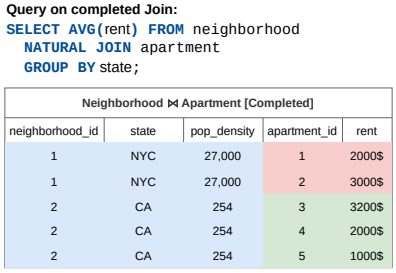
We are given an incomplete database D^i that consists of complete tables $T_1, T_2, \ldots$ and incomplete tables $T_j, T_{j+1}, \ldots$. The goal is to generate data for the incomplete tables $T_j, T_{j+1}, \ldots$ based on the available data that allows us to answer a query workload $Q_1, Q_2, \ldots, Q_n$ of aggregate queries such that query results on the completed database $Q_i(D^c)$ are close to the query results on the true (complete) database



(a) Annotated Example Schema.



(b) Models Synthesize Missing Tuples.



(c) Incompleteness Join.

Figure 4.4: Overview of ReStore to synthesize missing data (green) from existing data (blue and red). (a) Based on the annotated schema and the available data, the completion models are learned. (b) The learned schema-structured model can be used to synthesize a missing `apartment` tuple using a complete `neighborhood` tuple as input. (c) The model generates missing data for a given user query at runtime to answer queries over incomplete tables. The generated tuple factors (TFs) allow us to estimate the number of missing tuples.

$Q_i(D)$. Note that this formulation allows us to generate missing data individually for each query to answer the given query as accurately as possible. However, we can still cache generated data such that we do not need to generate new data for every query individually as we discuss later.

An important question for this problem is how to measure success. Based on our problem definition, a natural metric is how much the relative error of a query result on the incomplete database can be reduced by completing the data; i.e., how much more accurate the query results are after the completion. The relative error reduction for a given query Q_i can thus be defined as follows:

$$\textit{Rel. Error Reduction} = E_r(Q_i(D^i), Q_i(D)) - E_r(Q_i(D^c), Q_i(D)) \quad (4.1)$$

where the relative error E_r is the difference between the two query results normalized by the true query result. While for aggregate queries without a group-by, the relative error is trivial, for group-by queries we use the average relative error over all result tuples [72].

A limitation of the relative error reduction metric is that it does not show how well the bias of the incomplete database can be reduced independent of a given workload. We thus use a second metric called *bias reduction* to measure the success of data completion. This metric shows how well the true data distribution of a given attribute could be restored. For continuous attributes X, the bias reduction is defined as follows:

$$\textit{Bias Reduction} = 1 - \frac{|AVG^c(X) - AVG(X)|}{|AVG(X) - AVG^i(X)|} \quad (4.2)$$

where $AVG^c(X), AVG^i(X), AVG(X)$ are the averages of attribute X on D^c, D^i and D, respectively. Hence, the bias reduction is normalized in the interval $[0, 1]$ where larger values are preferable. For categorical attributes, we use the fraction of the biased attribute value since an average cannot be computed.

4.2.2 *Our Approach*

Our approach called ReStore to tackle this problem consists of the two steps depicted in Figure 4.4: First, a user has to (once) annotate a database schema before we train neural completion models that can be used to generate the missing data required to execute aggregate queries over the completed database.

SCHEMA ANNOTATION. In the annotation step, a user must indicate for a given incomplete database which tables are complete and which ones are incomplete. An example for an annotated schema is depicted in Figure 4.4a which consists of three tables of a housing database where two tables are marked as complete (`landlord` and `neighborhood`) and one table (`apartment`) is marked as incomplete.

In addition, information about the relationships between tables needs to be annotated. Here, the user has to provide information on whether there are any *complete* foreign-key relationships between tuples from a complete table and an incomplete table. For example, in Figure 4.4a all apartments of neighborhoods in NYC are available but not those for CA. In many of the application scenarios, the information, which relationships are complete, is known a priori and thus does not cause additional manual annotation overhead. For example, often a complete subset of data (e.g., apartments of a certain state) is available.

Based on the annotation, so-called tuple factors (TF) [72] can now be automatically computed to capture information about the relationships across complete and incomplete tables as shown in Figure 4.4a (e.g., how many apartments a complete neighborhood has). Based on the available data and the computed tuple factors, we then learn our completion models.

The user might also have other additional information, which can help to further enhance the quality of the synthesized data. Among these are table sizes for incomplete tables or aggregate statistics (e.g., average rental prices in certain states). Using techniques like iterative proportional fitting [131], this information can be used to improve our generated data. These techniques are orthogonal to our approach and we thus exclude those in the remainder.

MODEL TRAINING AND DATA COMPLETION. Given an annotated schema, we can now learn the completion models. As depicted in Figure 4.4b, two completion models have been learned that can either take data from the complete `neighborhood` table or the complete `landlord` table to synthesize missing `apartment` tuples. By taking complete tables as evidence our models synthesize missing tuples even if there is a bias in the missing data since we capture correlations across tables (e.g., which types of apartments are expected based on the characteristics of the neighborhoods).

These completion models can now be used at runtime to complete the missing data for a given user query. For instance, if a user wants to know the average rent per state, we first compute the completed join `neighborhood` $\bowtie$ `apartment`. More precisely, we introduce a new operator called *incompleteness join* to join complete and incomplete tables that generate the missing tuples needed to make the join complete. In our example, the incompleteness join would generate apartments for neighborhoods where the data is missing using the appropriate completion model of Figure 4.4b. Once the missing tuples for the join are generated (i.e., the incompleteness join produced its output), we can compute the aggregated result using a normal aggregation operator.

We decided to complete data on a per-query basis at runtime since completing the full database might be too expensive (and actually not

needed) for large datasets. However, it is important to note that the models are not query-dependent and only have to be learned once for an incomplete schema and can be reused across queries. Moreover, the generated data can still be materialized or even generated a priori.

SUPPORTED SCHEMA AND QUERIES. In general, our approach supports any relational schema where tables are connected via foreign-key relationships. For the workload, we currently limit ourselves to acyclic Select-Project-Aggregate-Join (SPJA) queries where joins are equi-joins along foreign-key relationships which are typical queries for decision making. An important aspect is that we can support arbitrary filter predicates or aggregate functions as well as any number of group-by attributes. The reason is that once data is completed for a join, we use normal query operators (e.g., filter or aggregate operators) to compute the query results. Supporting other types of queries, however, is indeed possible. For example, other join types (e.g., non-equi joins) could be added by deriving tuple factors that represent these join conditions.

4.2.2.1 *Discussion*

The central assumption of our approach is that both missing and available tuples have consistent correlations; i.e., while there can be a bias in the available tuples, it is required that the missing tuples have the same correlations between attributes as the remaining tuples. This is not a requirement specifically for ReStore but for any system that uses ML to complete a dataset since otherwise the available tuples cannot be used as evidence to predict the missing tuples. More technically, the conditional distributions of missing tuples t_m given an evidence tuple t_e should be equivalent for remaining and missing tuple distributions, i.e., $P_m(t_m \mid t_e) \approx P_r(t_m \mid t_e)$. If this assumption holds the main factor determining how accurately the original query result can be restored is the *predictability* of the query attributes as we will later show in our experimental evaluation. If the attributes are not predictable given the evidence tuples, our models will complete the data with lower confidence.

4.2.3 *Key Findings*

We now present the key findings, where we will demonstrate that ReStore can significantly reduce the bias in incomplete real-world datasets. Additional experiments on synthetic data that study which characteristics determine the extent to which the data can be debiased as well as accuracy and performance aspects and experiments on full analytical queries can be found in Chapter 10.

We now analyze how well ReStore can debias two real-world datasets, namely a housing and a movies dataset with different in-

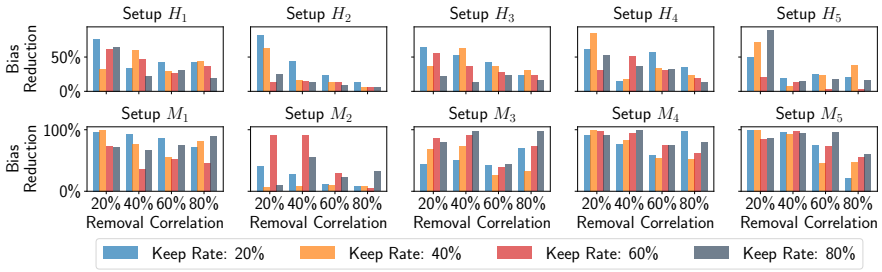


Figure 4.5: Bias Reductions for Real-World Data using the Setups of the Housing (H_i) and Movies (M_i) Datasets.

complete setups. In particular, we vary which tables are incomplete and which attribute is biased due to the incompleteness: we use five different setups denoted as H_i and M_i for the housing and movie data, respectively (cf. Chapter 10). In addition, we vary how much data is removed (i.e., varying the `keep_rate` parameter) as well as how much bias is introduced by the removed tuples (i.e., varying the `removal_correlation`). We then report the *bias reduction* as defined in Eq. 4.2 that expresses how well the original dataset could be restored and how much bias could be removed.

The results are shown in Figure 4.5 for all five setups given a variety of keep rates (between 20% and 80%) and removal correlations. As we see, the bias can significantly be reduced for all setups indicating the high quality of our completion models. This especially holds for the setups of the movies dataset where up to 100% of the bias can be removed. In general, a lower removal correlation is beneficial for our approach. The reason is that the lower the correlation, the more examples of high attribute values (for continuous attributes) remain in the training set and thus the model can learn more precisely what leads to those higher values. During the completion, it can then predict more accurately whether larger values are likely to occur. The keep rates do not seem to have a significant impact. The reason is that there are two opposing effects. On the one hand, a larger keep rate leads to a larger training dataset and the model can thus learn the distribution more accurately. On the other hand, the absolute error $|AVG_{complete}(X) - AVG_{incomplete}(X)|$ becomes smaller and the model has to predict more extreme values to correct the bias. Consequently, we do not see more accurate completions for larger keep rates.

Finally, we can conclude that ReStore can in fact reduce the bias in incomplete relational datasets by exploiting the data distribution. As we will show in Chapter 10, ReStore can also compensate errors due to incompleteness in analytical query results while providing confidence estimates on how accurate the completion likely is. Since ReStore is also a data-driven approach, we again do not require any query executions to train the models.

4.3 DISCUSSION

As demonstrated in this chapter, data-driven learning indeed fulfills the promise of state-of-the-art performance for many database tasks (cardinality estimation, AQP and relational dataset completion) while fully eliminating the high costs of training data collection for an unseen database. We have shown that for solving these tasks a learned representation of the data distribution is in fact sufficient. In the subsequent paragraphs, we will highlight the challenges one has to solve to leverage data-driven learning for a particular database task. Finally, we will discuss why data-driven learning is no silver bullet for all database tasks.

It is in general not straightforward to use a model that has learned the data distribution to solve a concrete task. In fact, both DeepDB and ReStore require extensive algorithms to determine how the models should be queried to support a particular task. For instance, DeepDB requires probabilistic query compilation to support arbitrary ad-hoc cardinality and AQP queries in the schema, i.e., without knowing all join paths beforehand. In contrast, ReStore requires specialized completion algorithms that determine what combination of models are used to complete the data for a specific query.

Another aspect is that for data-driven learning we typically cannot use off-the-shelf ML models but have to extend the models significantly to be a good fit for the learned component. For instance, for DeepDB it is crucial to be able to update the underlying models efficiently. Analogously, for ReStore we had to modify autoregressive models to also incorporate the entire relational schema as evidence. Hence, while it is appealing to realize a learned database component using data-driven learning, it is often not straightforward, which ML techniques should be used and how they have to be extended to support the particular task.

Finally, data-driven learning is not applicable to every database task, most importantly tasks that require knowledge about the executed workloads. For instance, for cost estimation, we have to observe query executions to learn about the operator characteristics. Merely knowing the data distribution is not sufficient for such tasks since it does not provide sufficient information. This motivates the third direction proposed in this dissertation. As mentioned before, while data-driven learning cannot solve such tasks in isolation, it might still provide useful features as we will discuss in the next chapter.

5 ZERO-SHOT LEARNED COMPONENTS

While data-driven learning eliminates the need to collect training queries for unseen databases, it is not applicable to database tasks that require knowledge about query executions. This motivates the third proposed direction of this dissertation - zero-shot learned database components. In contrast to data-driven learning, zero-shot learned components are applicable to a broader set of database tasks including those that require knowledge about workload executions such as cost estimation. The general idea is to first pretrain a model on a diverse set of databases and workloads to then allow the model to generalize to unseen databases out-of-the-box. Hence, zero-shot learned components avoid the high costs of training data collection similarly to data-driven learning.

In the following, we will first reference the two publications introducing zero-shot learned components and describe the contributions of the author of this dissertation. Thereafter, in Section 5.1, we will provide an overview of zero-shot learning in general and the key challenges before we provide an overview of our proposed zero-shot model for cost estimation (cf. Section 5.2). We will then present the key findings in Section 5.3 and finally discuss the contributions in Section 5.4. For full details on the publications, we refer to the Chapters 11 and 12.

PUBLICATIONS The work on zero-shot learned database components is published in two peer-reviewed publications. A broader vision paper outlining potential use cases and key challenges is published as "One Model to Rule them All: Towards Zero-Shot Learning for Databases" in the *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022* [62], cf. Chapter 11. A significantly extended full paper that focuses on zero-shot cost estimation exclusively is published as "Zero-Shot Cost Models for Out-of-the-box Learned Cost Prediction" in the *Proc. VLDB Endow.* [66], cf. Chapter 12. The source code and data [65] are publicly accessible for reproducibility and future research.

CONTRIBUTIONS OF THE AUTHOR The contributions to the above publications by Benjamin Hilprecht, the author of this dissertation, are as follows. Benjamin Hilprecht is the leading author of both publications and was thus responsible for the proposed approach for zero-shot learning, the experimental evaluation, and the manuscript for both publications. The co-author Carsten Binnig contributed in-

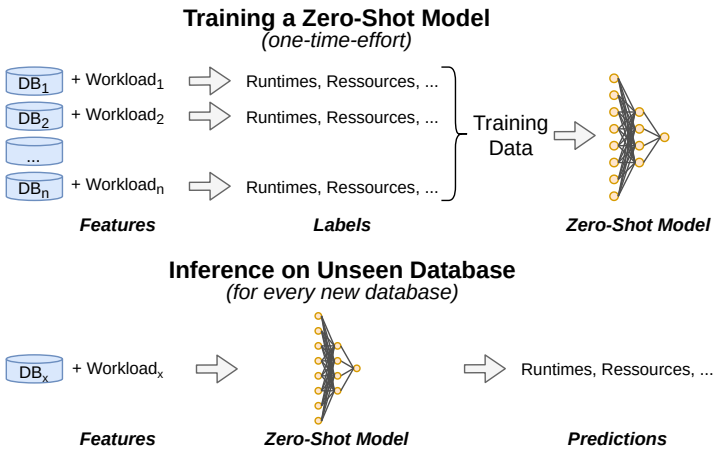


Figure 5.1: Overview of zero-shot learning for databases. In line with other zero-shot approaches such as GPT-3 which enables zero-shot learning for NLP, a zero-shot model for databases can generalize to a completely new database and workload without the need to be trained on that particular database.

valuable feedback and agrees with the use of the publications for this dissertation.

5.1 ZERO-SHOT LEARNED DATABASE COMPONENTS

In this section, we introduce the general idea behind zero-shot learned database components before we discuss the key challenges associated with it.

5.1.1 *Overview*

Figure 5.1 shows the high-level idea that is behind the general paradigm of zero-shot learning for databases. During the learning phase, similar to workload-driven learning, for zero-shot learning, we have to execute a representative workload and collect training data.

The main difference to workload-driven learning though, which makes our approach attractive, is that zero-shot models *generalize to unseen databases out-of-the-box*. To allow a zero-shot model to make predictions about unseen databases without the need to retrain the model for this particular database, we require a new method of representing queries as we discuss below (cf. Key Challenges). A transferable representation is at the core of learning zero-shot models in a generalizable way and thus enables them to make predictions for queries on a new database (e.g., for physical cost estimation) that the model has never seen before.

Moreover, for being able to generalize to new databases, a zero-shot model is trained on different databases. While this might seem to cause high upfront costs before a zero-shot model can be used, it is

important to note that the *training data collection is a one-time-effort* which is very different from workload-driven learning that needs to collect training data for every new database a model should support. Moreover, cloud database providers such as AWS, Microsoft, or Google, typically already have significant amounts of such information available since they keep logs of their customer workloads and could thus apply zero-shot learning right away without the need to collect training data in the first place.

Finally, a last important aspect is that zero-shot learning is not only generalizable across databases but is a new learning approach that can be applied to a *variety of database tasks* that range from physical cost estimation, design tuning or knob tuning to query optimization and scheduling. To enable zero-shot models to generalize to different tasks though, the models need to be capable of capturing not only information about query plans and their runtimes but also information about other aspects (e.g., how indexes or changes in the database configuration influence the query runtime) as we discuss later.

5.1.2 *Key Challenges*

In the following, we discuss the key challenges that we think are at the core to make zero-shot learning for databases efficient and accurate.

TRANSFERABLE REPRESENTATION OF DATABASE AND QUERIES. State-of-the-art workload-driven models [79, 161] can only leverage training data from a single database and thus they cannot simply be trained on a variety of databases to obtain zero-shot models. The reason is that the query representation is not *transferable* to an unseen database. For instance, attributes names (e.g., those used in filter predicates) are typically encoded using a one-hot encoding assigning each column present in the database a specific position in a feature vector. Hence, the column `production_year` of the IMDB dataset might be encoded using the vector $(0, 1, 0)$ (assuming that there are only three columns in total). If the same model is now used to predict query costs for the SSB dataset, the second column in the database might be `region`, which has very different semantics (i.e., very different data distributions or even a different data type). As such, cost models based on non-transferable representations will produce estimates that are most likely way off. In fact, such non-transferable feature encodings are used in various places of query representations such as table names as part of plan operators or literals in filter predicates. Hence, for zero-shot models, we require a novel representation of queries that is transferable across databases while still being expressive enough to enable accurate estimations. The specific representation depends on the concrete task at hand. In the subsequent Section 5.2 we will derive such an architecture for the task of cost estimation.

TRAINING DATA COLLECTION AND ROBUSTNESS. A second key challenge to enable zero-shot learning is clearly the training data collection for learning a zero-shot cost model. An important question is how many and which databases and workloads a zero-shot model needs to observe during training to make robust decisions on unseen databases. As we show in our initial experiments for zero-shot cost estimation in Section 5.3, for example, already a relatively small number of databases along with the respective workload information (e.g., the featurized query plans and runtimes) is sufficient to generalize robustly and even outperform existing workload-driven approaches.

In Chapter 12, we also provide theoretical arguments to recognize cases when a zero-shot model has seen sufficiently many databases to generalize robustly. Intuitively, we evaluate the model on test databases that have not been used during training similar to the common practice in ML to evaluate a model on a holdout set.

SEPARATION OF CONCERNS. Finally, a last important aspect of zero-shot models is to decide what should be learned by the model to fulfill its core promise and when to separate concerns. For example, workload-driven approaches often prefer end-to-end learning, i.e., to make predictions for a query plan (e.g., the runtime), they internalize both the data characteristics (e.g., the data size and distributions) as well as the system characteristics (e.g., the runtime complexity of database operators) in one model.

However, since the data characteristics can be entirely different for a new database, such an end-to-end approach will not work for zero-shot learning. Hence, we suggest that data characteristics for zero-shot learning should be captured by separate data-driven models (such as [72, 185]). For example, a feature that can be captured by a data-driven model are the input- and output-cardinalities of operators in a query plan. That way, when using cardinalities as input features for the zero-shot models, these models learn to predict the runtime behavior of operators based on input/output sizes that can be derived for any database which again enables a transferable representation of queries that does not depend on the concrete data distribution of a single database. In particular, for the task of cost estimation (cf. Section 5.2) we will demonstrate that leveraging data-driven models to predict intermediate cardinalities as features yields more precise zero-shot cost estimation models.

One could now argue that this violates the core promise of zero-shot learning since data-driven models need to be learned for each new database. However, data-driven models can be derived from a database without running any training query and typically a sample of the database is enough to train these models. Moreover, for cardinality estimation, we could even use simple non-learned estimators (e.g., histograms) as input for the zero-shot models. As we show in our initial results in Chapter 12 and the subsequent Section 5.2, even those

simple estimators often provide sufficient evidence for our zero-shot cost models to produce accurate estimates.

To summarize, a key question in this context is to decide what a zero-shot model should learn and which aspects should be treated separately. Clearly, a guide for this question is to think about what is tied to a particular data distribution and which aspects hold in general which should then be included in the zero-shot model. Moreover, as we discuss in more detail in Chapter 11, for design tuning or query optimization another question is how to combine zero-shot models with optimization procedures or other learning approaches (e.g., value networks) to implement efficient search strategies.

5.2 ZERO-SHOT LEARNED COST ESTIMATION

In this section, we specifically focus on the problem of cost estimation, which we tackle using zero-shot learning. We first introduce the problem of zero-shot cost estimation and then present an overview of our approach before we discuss the underlying assumptions and limitations.

5.2.1 *Problem Statement*

The overall goal of zero-shot cost estimation is to predict query latencies (i.e., runtimes) on an unseen database without having observed any query on this unseen database. As before we use the term database to refer to a particular dataset (i.e., a set of tables with a given data distribution). Note that the problem of zero-shot cost estimation is thus in sharp contrast to the problem addressed by state-of-the-art workload-driven cost models, which train a model per database. Finally, while we believe that zero-shot learning for DBMSs is more generally applicable, we restrict ourselves to cost estimations for relational DBMSs (both single-node and distributed). In particular, zero-shot cost models for other types of systems such as graph-databases or streaming systems are interesting avenues for future work.

5.2.2 *Our Approach*

A key challenge for developing zero-shot cost models is the question how to design a model that generalizes across databases. Here, we draw inspiration from the way classical cost models in DBMSs are designed. Typically, these consist of two models: a database-agnostic model to estimate the runtime cost and a database-dependent model (e.g., histograms) to capture data characteristics. When predicting the cost of a query, the estimated cardinalities and other characteristics (i.e., outputs of the database-dependent models) serve as input to the general database-agnostic cost model, which captures the general

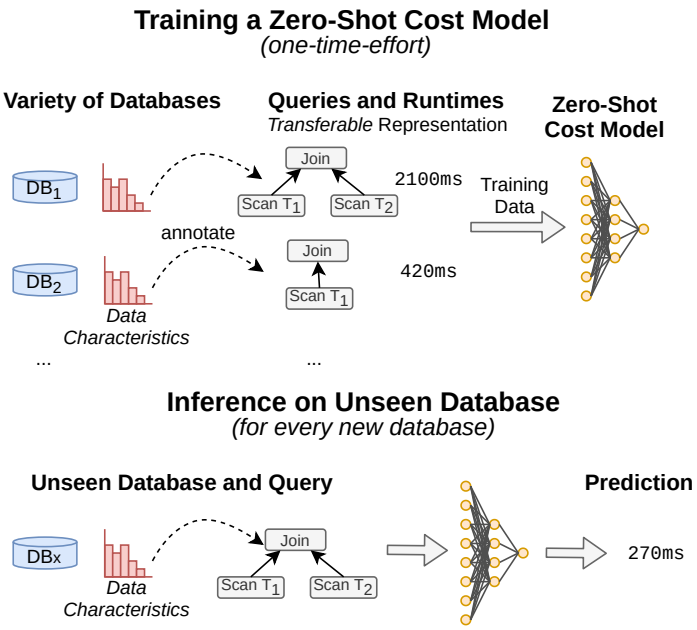


Figure 5.2: Overview of Zero-Shot Cost Estimation. The zero-shot cost model is trained *once* using a variety of queries and databases. At inference time, the model can then provide cost estimates for an unseen database and queries without requiring additional training queries. Enabling zero-shot cost estimation is based on two key ideas: (1) a new *transferable* query representation and model architecture is used to enable cost predictions on unseen databases and (2) we separate concerns, i.e., a zero-shot model learns a general database-agnostic cost model, which takes database-specific characteristics as input.

system behavior (e.g., the costs of a sequential scan grows linearly w.r.t. the number of rows). While the classical models are lightweight, they often largely under- or overestimate the true costs of a query since models are too simple to capture complex interactions in the query plan and data.

Hence, in our approach, we also separate concerns but use a much richer learned model, which similarly takes data characteristics of the unseen database as input to predict query runtimes in a database-agnostic manner. As depicted in Figure 5.2 (upper part), for training such a zero-shot cost model we provide different query plans along with the runtime as well as the data characteristics of the plan (such as tuple width as well as intermediate cardinalities) to the zero-shot cost model. Once trained, the model can be used on unseen databases to predict the query runtime as shown in Figure 5.2 (lower part).

As mentioned before, to predict the runtime of a query plan on a new (unseen) database, we feed the query plan together with its data characteristics into a zero-shot model. While data characteristics such as the tuple width can be derived from the database catalogs, other characteristics such as intermediate cardinalities require more complex techniques. To derive intermediate cardinalities of a query plan in

our approach we thus make use of previously proposed data-driven learning [72, 185] that can provide exact estimates on a given database. Note that this does not contradict our main promise of zero-shot learning since data-driven models to capture data characteristics can be learned without queries as training data.

Another core challenge of enabling zero-shot cost models that can estimate the runtime of a plan given its data characteristics is how to represent query plans, which serve as input to the model. While along with workload-driven cost models, particular representation methods for query plans have already been proposed, those are not applicable for zero-shot learning. The reason is that the representations are not *transferable* across databases as discussed in Section 5.1.2. For instance, literals in filter predicates are provided as input to the model (e.g., `2021` for the predicate `movie.production_year=2021`). However, the selectivity of literals will vary largely per database since the data distribution will likely be different (e.g., there might not even exist movies produced in 2021 in the test database).

Hence, as a second technique, we propose a new representation for queries that completely relies on features that can be derived from any database to allow the model to generalize to unseen databases. For example, predicates for filter operations in a query are encoded by the general predicate structure (e.g., which data types and comparison operators are used in a predicate) instead of encoding the literals. In addition, data characteristics of a filter operator (e.g., input and output cardinality to express the selectivity) are provided as additional input to a zero-shot model. That way, a zero-shot model can learn the runtime overhead of a filter operation based on database-agnostic characteristics.

Finally, a last important aspect of zero-shot cost models is that they can easily be extended to *few-shot learning*. Hence, instead of using the zero-shot model out-of-the box (which already can provide good performance), one can fine-tune the model with only a few training queries on an unseen database.

5.2.3 *Assumptions and Limitations*

While we expect zero-shot cost models to support a variety of different databases and workloads out-of-the-box, we next discuss the assumptions for a successful generalization.

Our main assumption is that we only focus on the transfer of learned cost models across databases for a *single database system* on a *fixed hardware*. We think that this is already challenging and allows for many interesting use cases. For instance, with zero-shot cost models cloud DBMSs (such as Redshift or Snowflake) can use learned cost models for new customer databases and workloads with significantly lower training overhead compared to the existing workload-driven

models that require that a model is trained per new database. While we believe that zero-shot cost models can be extended to support also the transfer of cost models between different hardware setups and DBMSs by adding additional transferable features, we leave this to future work and assume a fixed hardware and DBMSs in this dissertation.

Furthermore, while zero-shot cost models can generalize to unseen query patterns as we show in our experiments, it is clearly required that the training queries have a certain *coverage*, i.e., come with a diverse set of workloads and databases. For instance, it is a minimum requirement that every physical operator is observed in the training data s.t. the model can internalize the overall characteristics. Moreover, if there are extreme differences between training and test workloads, we expect the zero-shot model accuracy to degrade. We discuss how to detect and mitigate such cases by fine-tuning a zero-shot model in Chapter 12.

5.3 KEY FINDINGS

We now present the key findings of the evaluation of zero-shot cost models. In particular, we will demonstrate that our approach provides accurate cost estimates on entirely unseen databases and state-of-the-art workload-driven approaches require ten thousands of query executions for a similar accuracy on an unseen database. Note that an extensive evaluation including generalization experiments and ablation studies can be found along with more details in Chapter 12.

GENERALIZATION TO UNSEEN DATABASES. To evaluate zero-shot cost models, we first require a benchmark that includes a variety of databases and workloads to pretrain the zero-shot models. We provide more details on the benchmark in Chapter 12. Note that the benchmark also encompasses established datasets and workloads such as JOB on IMDB, TPC-H and SSB, which are commonly used to evaluate learned cost estimation approaches. To validate that zero-shot cost models generalize to unseen databases, we trained a zero-shot model using workloads on 19 out of these 20 datasets of the benchmark as training data and evaluated the model on the workload of the unseen (remaining) database. In particular, we use the trained model to predict the runtimes of the queries on the unseen database and report the median q-error [79, 116].

As described in Section 5.2, zero-shot cost models require intermediate cardinalities as features as a result of the separation of concerns. In this experiment, we either used predictions of data-driven cardinality estimators or the actual cardinalities, which are not available in practice prior to execution but serve as an interesting upper baseline for zero-shot learning (i.e., how accurate the predictions become with

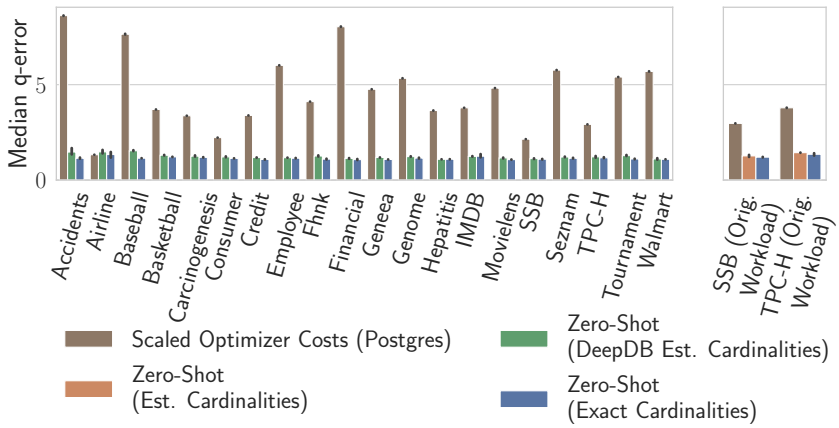


Figure 5.3: Zero-Shot Generalization across Databases. The zero-shot models are trained using workloads on 19/20 databases and tested on the remaining unseen database. Overall, zero-shot models are significantly more accurate than the scaled estimates of the optimizer cost model. In addition to using workloads as defined by our benchmark (left), we repeated this experiment with standard benchmark workloads (SSB and TPC-H on the right) to further show the generalization potentials of zero-shot cost models.

perfect cardinality estimates). For the data-driven cardinality estimator, we trained DeepDB [72] models, which worked best in preliminary experiments. To the best of our knowledge, we are the first to propose zero-shot cost estimation and thus no other learned approaches are included as a direct baseline in this first experiment where we aim to analyze the accuracy on unseen databases.

The results can be seen in Figure 5.3. In general, the zero-shot models offer robust performances for all of the databases despite the varying complexity. In fact, all median q-errors are below 1.54 for the version using DeepDB cardinality estimates (vs. 8.62 in the worst case for the *Scaled Optimizer* cost). Finally, we can see that zero-shot cost models using DeepDB cardinalities are almost matching the performance with perfect cardinalities. This suggests that the models can cope with partially inaccurate cardinalities.

Overall, we can see that the zero-shot cost models are significantly more accurate than the scaled optimizer estimates outperforming these on 18 out of 19 datasets and being on par for the last remaining dataset (Airline). The reason is that zero-shot cost models capture subtleties in operator performance and interactions of operators in the plan more accurately than simplistic cost models. The results are just on par for the remaining database since the optimizer costs are relatively accurate because it is merely a star schema, i.e., a relatively simple schema structure.

COMPARISON WITH WORKLOAD-DRIVEN LEARNING. In the following, we contrast the performance of zero-shot cost models with workload-driven approaches. An interesting question is how many

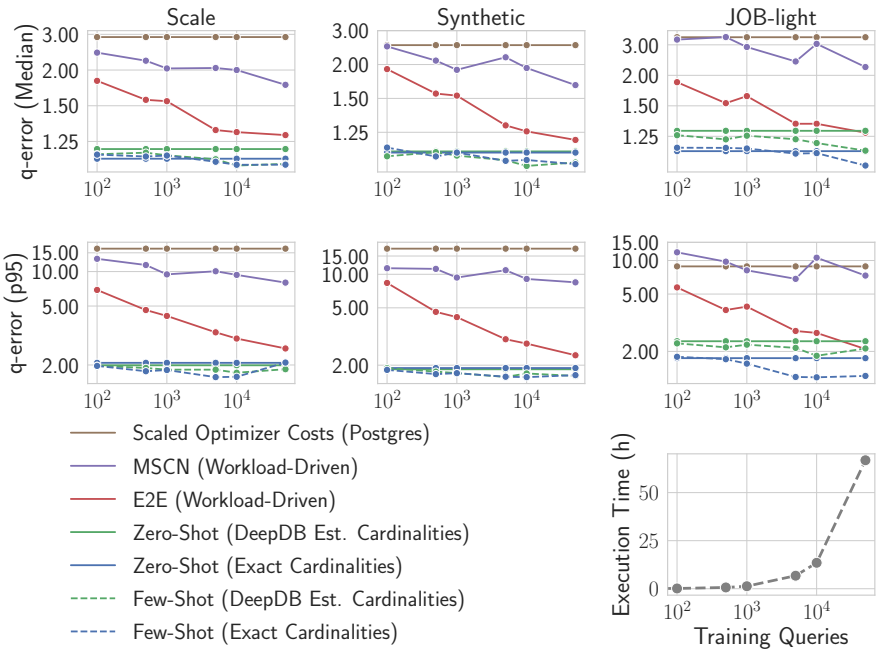


Figure 5.4: Estimation Errors of Workload-Driven Models for a varying Number of Training Queries compared with Zero-Shot Cost Models. Even the most accurate workload-driven model (E2E) requires approximately 50k query executions on an unseen database for a comparable performance with zero-shot models, which is roughly equivalent to 66 hours of executed workload. Since zero-shot models do not require any additional queries it is significantly cheaper to deploy them for a new database.

training queries are required for workload-driven learning on an unseen database to match the performance of zero-shot learning, which we will study next. In particular, in this experiment, we evaluate the q-errors for the scale, synthetic, and JOB-light workloads (IMDB). As before zero-shot models are not trained on IMDB at all (but on the other 19 databases) while workload-driven models are trained on a varying number of training queries on IMDB.

As baselines, we use the state-of-the-art models for workload-driven cost estimation, namely the E2E model proposed by Sun and Li [161] as well as the MSCN model by Kipf et al. [79]. Furthermore, as the last baseline, we again employ the scaled costs of the Postgres query optimizer.

In Figure 5.4, we depict the median q-error of comparing our zero-shot performance to the baselines as discussed before for the IMDB benchmark workloads for a varying number of training queries. As we can see, the zero-shot cost models can estimate the runtimes accurately even though queries on the IMDB dataset were not observed in the training data. In particular, E2E requires about 50k training queries on the IMDB database to be on par with zero-shot cost models. As we can see in the lower right plot in Figure 5.4, this amount of queries takes approximately 66 hours to run, which is a significant effort given

that it has to be repeated for every new database. Another interesting comparison is to use the training queries also to fine-tune the zero-shot models on the IMDB database; i.e., we use zero-shot models in the few-shot mode discussed in the paper. As we can see, few-shot cost models that are fine-tuned on the IMDB database can further improve the cost estimation accuracy of zero-shot models. It is thus beneficial to also leverage fine-tuning in case training queries for the unseen database are available.

Finally, we can see that the MSCN models are not equally accurate, which is likely due to the fact that they do not consider the physical plan that was run to execute a given query. Still, all learned approaches are more accurate than the scaled optimizer in the median after only a few queries. Furthermore, we can observe that zero-shot and few-shot cost models not only outperform workload-driven models in the median but also in the tail performance, i.e., on the 95th percentile q-error. We can observe similar effects for the maximum q-error.

5.4 DISCUSSION

Our above results indicate that zero-shot learned database components can indeed generalize to unseen databases out-of-the-box without requiring additional training data. In particular, for cost estimation, comparable workload-driven approaches require 50k query executions on the unseen database for a comparable performance with zero-shot models, which is roughly equivalent to 66 hours of executed workload. Hence, when deployed for cloud vendors, zero-shot models have the potential to significantly reduce the costs for training data collection - potentially no training data has to be collected at all since query logs are available and sufficient for training.

As mentioned above, we believe that data-driven learning and zero-shot learning can be combined effectively. In fact, we have demonstrated that the intermediate cardinalities predicted by data-driven models can serve as informative features for zero-shot cost models. However, this requires that a data-driven model is available for an unseen database. In cases where this additional cost of training a data-driven model per database is not acceptable, we can fall back to using optimizer cardinality estimates as features, which we have also shown to yield competitive accuracies.

While we believe that our initial results on zero-shot cost estimation are promising, there are many open research questions to derive zero-shot models for other database tasks. We will provide a discussion in Section 6.2.1.

6 CONCLUSION AND OUTLOOK

We now conclude with the contributions of this dissertation towards data-efficient learned database components in Section 6.1. Afterwards, we will provide an outlook on the open challenges for learned database components in Section 6.2, which further motivates future research in this field.

6.1 DATA-EFFICIENT LEARNED DATABASE COMPONENTS

This dissertation addresses the problem of the high costs of training data collection for state-of-the-art learned database components. In particular, for each unseen database that should be supported ten thousands of queries have to be executed to gather sufficient training data for the underlying models. This dissertation thus introduced three directions to alleviate this problem: *(i)* data-efficient workload-driven learning, *(ii)* data-driven learning and *(iii)* zero-shot learned database components.

First, in Chapter 3, we introduced two techniques to reduce the number of required training queries for workload-driven models - simulation models to bootstrap the components without query executions and differentiable programming, which allows incorporating domain knowledge in the design. While these techniques reduced the cost of workload-driven learning, the need for training queries for unseen databases is still not completely eliminated. In addition, both the simulation model and the differentiable programming require an explicit modeling of the underlying task, which can be hard to achieve.

Hence, we proposed two alternative paradigms of learned components throughout this dissertation. First, in Chapter 4, we introduced data-driven learning where the learned components leverage the data distribution instead of a representative workload. This in fact completely eliminates the need for training queries and can achieve state-of-the-art performance in cardinality estimation, AQP and completion of relational datasets. However, data-driven approaches are not as broadly applicable as workload-driven models since they only leverage the data distribution as a signal and cannot solve tasks that require knowledge about workload executions.

We thus proposed zero-shot learned database components in Chapter 5. The idea is to pretrain the models on a variety of databases and workloads to enable those to generalize to unseen databases out-of-the-box, i.e., without observing an additional training workload. We have demonstrated that the key challenge of such models

is to derive a transferable representation and have thus proposed such a novel architecture for the task of cost estimation. In fact, we have demonstrated that zero-shot cost models can predict costs on unseen databases out-of-the-box and comparable workload-driven models require ten thousands of query executions as training data for a comparable performance.

Overall, we can thus conclude that the proposed techniques substantially improve the data-efficiency of learned database components. In particular, for data-driven learning training, a model on a sample of the database is sufficient to support an unseen database whereas for zero-shot learning the same model can be used for a variety of databases, and thus no additional effort is incurred. Hence, the high costs of supporting unseen databases, which are required for workload-driven models, are completely eliminated, which we believe could be crucial for cloud DBMS vendors since this significantly reduces the effort to support a large number of customers.

We have validated the directions by solving individual database tasks such as cardinality estimation. Importantly, we were not only able to achieve a comparable performance but outperformed the state-of-the-art workload-driven models even if a large set of training queries is provided for training such models. While the individual tasks are providing evidence that the proposed directions are viable, we believe that our ideas are general and could be used for many learned database components for a variety of tasks. For instance, we expect that the applicability of zero-shot learning is not limited to the cost estimation task but is a broader concept that could also be used for tasks such as query optimization, scheduling or physical design advisors.

6.2 OUTLOOK

While the contributions of this dissertation significantly improve the data-efficiency of learned database components, there are many more challenges and interesting research directions that have to be solved before learned components can be broadly adopted in database systems. We see several research challenges in the areas of *(i)* a broader coverage of tasks for zero-shot learning eventually resulting in full zero-shot databases (cf. Section 6.2.1), *(ii)* improvements for data-driven models to support a broader set of query classes and an increased efficiency (cf. Section 6.2.2) and *(iii)* robustness and debugability, which is especially important for the use in production systems (cf. Section 6.2.3).

6.2.1 *End-to-End Zero-Shot Databases*

While we have demonstrated that cost estimation can be solved with zero-shot learning, we believe that many more tasks can be supported in the future finally yielding full zero-shot database systems.

The first step towards this vision should be to support tasks, which are commonly required in cloud databases, where zero-shot learning is particularly interesting since training data in the form of query logs is already available. Important tasks include physical design decisions such as materialized view or index selection, which have a significant impact on the workload runtime but are non-trivial to automate due to the complex interactions of physical design and query executions. In addition, zero-shot query optimizers are particularly interesting since workload-driven learned optimizers have already been shown to yield more efficient query plans [109, 112, 184] while reducing the immense engineering efforts required to maintain and improve query optimizers.

An important question is how the search space is represented for such zero-shot tasks and how the data distribution can be encoded. For instance, for query optimization, there is an immense design space of physical plans for a particular logical plan. Analogously, there are many possible materialized views that can potentially speed up a workload. While there have already been initial works to represent this search space for workload-driven models as an RL problem [59, 99, 112], it is not straightforward to implement this as zero-shot models since a key requirement is that the underlying representation is transferable. We believe that the transferable graph encoding suggested for cost estimation can already be a good starting point for additional tasks.

Another interesting question is how the data distribution should be encoded. For the cost estimation problem, we have used intermediate cardinalities as features and were thus able to logically separate the general cost estimation model from a particular data distribution of a single database. However, this technique is not generally applicable. For instance, for query optimization, selecting the right join ordering is an important subproblem, which depends on complex correlations that determine the result sizes of subjoins. It is not viable to provide the expected cardinalities for all subjoins as features to the model. However, again workload-driven solutions to this problem cannot directly be leveraged for zero-shot models since these implicitly also learn the data distribution of the training database and would thus not be transferable. One solution could be generally to provide the data distribution and correlations in the dataset as features for the zero-shot optimization model. Both traditional histogram-based approaches that capture the distribution of a single attribute or a set of attributes

or modern representations that have been suggested for pretrained cardinality estimation models [102] could be useful for this problem.

Finally, we believe that the zero-shot components could be combined to obtain an entire database system that automatically adapts to a particular database and workload with none or only a few observed queries. This could potentially allow significant speedups since state-of-the-art systems are typically one-size-fits-it-all solutions based on heuristics that can be specialized heavily for a particular database. However, this not only requires that more components can be expressed as a zero-shot component but also architectural approaches that ensure that the components jointly yield a robust system that offers a sufficient performance.

6.2.2 *Practical Data-Driven Learning*

As demonstrated for cost estimation, we believe that zero-shot learned components can often make use of data-driven models that provide informative features based on the data distribution. However, in order to be practically applicable, we see two main research challenges regarding training efficiency and query coverage.

First, while data-driven models do not require observing any training queries, we still have to train a single model for every database. This means that we do not incur the cost of running a representative workload for every customer as for workload-driven learning but the cost of training a data-driven model. Since especially for cloud database vendors this cost has to be incurred for every customer, this motivates further research on efficient data-driven models. In particular, generating statistics in the form of histograms in traditional systems is usually very efficient. In contrast, data-driven learning requires training times in the order of minutes as demonstrated for the DeepDB system and thus leaves a potential for optimizations. Of particular interest could again be initial work on pretrained models for cardinality estimation [102] that allows to efficiently obtain a representation of the data distribution of a single table that can afterwards be used to predict selectivities. An additional direction could be to combine more traditional approaches (e.g., synopsis such as histograms) with ML to obtain both efficient to construct and accurate estimators. For instance, such classical statistics could serve as a feature for ML-based techniques. In these cases, the distinction between data-driven learning and zero-shot learning is less clear since pretraining might also play an important role.

Second, as of today data-driven models only support a subset of all possible queries and are thus inferior to workload-driven models in this regard. For instance, predicates including wildcards on strings, nested queries or acyclic joins are typically not directly supported. While we could fall back to more traditional techniques for queries that

come with such predicates, broader coverage of data-driven models would enable a broader set of workloads to benefit and also justify the cost of creating the models in the first place.

6.2.3 *Robustness and Debuggability*

Finally, it is crucial for database developers and vendors to have a certain robustness and debuggability in case the learned component does not behave as expected. Specifically, the derived components should also provide a sufficient and predictable performance for previously unseen data and workload characteristics. Moreover, in cases where problems occur, it should be possible for developers to reason about the learned component to eventually stabilize the system.

This is challenging to achieve with state-of-the-art learned components since these rely on ML models, in particular in many cases DNNs, which are black boxes and often do not enable a closer inspection of the inner workings of the models. While explainable ML [19] has seen some progress in recent years, it is still hard to reason why a particular model output came to be. We believe that there are two main strategies to alleviate this problem.

First, a major design question is what parts of the learned component have to be learned. In general, while an end-to-end formulation is appealing since the model has more degrees of freedom to solve the problem, it will be harder to reason about individual decisions. For instance, for materialized view selection, we could formulate the entire problem as a single RL problem solved by a single DNN. This would make it very hard to explain why a particular materialized view was preferred over another one. In contrast, we could also formulate the problem as an optimization problem and only estimate the benefit of a particular materialized view (i.e., how much the performance of a single query can be improved) using a learned model. If an unexpected materialized view is selected in the end, we would be able to inspect whether the benefit was overestimated or the optimization problem was the root cause - for instance, due to a budget that did not allow other materialized view candidates to be chosen.

Second, the class of ML models has a significant impact on the explainability and predictability of the learned component. For instance, for linear models, it is straightforward to examine the impact of a single feature on the model output and the behavior is less brittle w.r.t. minor perturbations of the input. This robustness could be very desirable for learned components. In contrast, while a DNN can potentially achieve superior results, it is harder to reason about it and more prone to unpredictable behavior. Hence, in the design of learned database components, explainability, robustness and performance are a tradeoff, where the choice of models can have a significant impact.

Overall, we believe that the robustness and debuggability of learned components is a particularly challenging research direction and at the same time of high importance for practitioners.

Part II

PEER-REVIEWED PUBLICATIONS

7

LEARNING A PARTITIONING ADVISOR FOR CLOUD DATABASES

ABSTRACT

Cloud vendors provide ready-to-use distributed DBMS solutions as a service. While the provisioning of a DBMS is usually fully automated, customers typically still have to make important design decisions which were traditionally made by the database administrator such as finding an optimal partitioning scheme for a given database schema and workload. In this paper, we introduce a new learned partitioning advisor based on Deep Reinforcement Learning (DRL) for OLAP-style workloads. The main idea is that a DRL agent learns the cost tradeoffs of different partitioning schemes and can thus automate the partitioning decision. In the evaluation, we show that our advisor is able to find non-trivial partitionings for a wide range of workloads and outperforms more classical approaches for automated partitioning design.

BIBLIOGRAPHIC INFORMATION

The content of this chapter was previously published in the peer-reviewed work Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. "Learning a Partitioning Advisor for Cloud Databases." In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. Ed. by David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo. ACM, 2020, pp. 143–157. DOI: 10.1145/3318464.3389704. URL: https://doi.org/10.1145/3318464.3389704. The contributions of the author of this dissertation are summarized in Section 3.1.

7.1 INTRODUCTION

MOTIVATION: Providing data solutions as a service is a growing field in the cloud industry. Cloud platforms such as Amazon Web Services or Microsoft Azure provide multiple ready-to-use scale-out DBMS solutions for OLAP-style workloads as a service. Using these services, customers can easily deploy a database, define their database schema, upload their data and then query the database using a cluster of machines. While the provisioning is usually fully automated, many design decisions which were traditionally made by the database administrator remain a manual effort. For example, in Azure's Data Warehouse but also in Amazon Redshift customers have to choose a partitioning attribute of a table to split large tables horizontally across multiple machines. Partitioning the database can dramatically improve the performance of analytical workloads since data-intensive SQL queries can be farmed out to multiple machines.

While partitioning a database in an optimal manner is a non-trivial task it has a significant impact on the overall performance. For example, analytical queries typically involve multiple joins over potentially large tables. If two tables are co-partitioned on the join attributes they can be joined locally on each node avoiding costly network transfers. Deciding for complex schemata with many tables and possible join paths which tables should be co-partitioned is a non-trivial task since this not only depends on the schema but also other factors such as table sizes, the query workload (i.e., which joins are actually important and how often tables are joined), or hardware characteristics such as network speed and of course the database implementation itself.

There exists already a larger body of work to automate the physical design of distributed DBMSs including the data partitioning [3, 127, 145]. These advisors formalize the problem as an optimization problem and thus rely on cost models to estimate the runtime of queries for different partitionings. However, this approach is unsuitable for a cloud providers: First, cloud providers typically allow customers to deploy their DBMS solutions on various hardware platforms which renders the problem of acquiring exact cost models a challenge on its own. Secondly, even if the cost model is tuned for a given hardware platform, optimizer cost estimates are still often notoriously inaccurate [91] resulting in non-optimal partitioning designs if existing automated design approaches are used as we show in our experiments.

CONTRIBUTIONS: In this paper, we propose a different route and make the case to use Deep Reinforcement Learning (DRL) to realize a cloud partitioning advisor as a service that can be used for internal and external DBMS solutions. The advantage for DRL is that a DRL agent learns by trial and error, and thus they do not rely on the fact that an accurate cost model is available. Instead, by deploying

different partitionings and observing query runtimes, they learn the tradeoffs for varying workloads. Once trained, our learned advisor can be queried to obtain a partitioning for the observed workload.

One could now argue that instead of learning a DRL agent, we could simply learn a neural cost model that predicts the runtime for different partitionings and then use an optimization procedure similar to [127] to select a suitable partitioning. Recently, learned cost models have also been used for query optimization [112] or cardinality estimation [79]. The main reason why we use DRL for the partitioning problem is that it inherently addresses the exploitation vs. exploration tradeoff (i.e., it efficiently navigates the space of possible partitionings instead of relying, for example, on naïve random sampling from the space of possible solutions to collect training data). This is especially important for learned cost models, where collecting training data can be immensely expensive since a representative set of queries has to be executed over a potentially large database. In our case, the high training costs are amplified since we need to run the same set of queries over a representative set of different partitionings of the database where repartitioning itself is a costly operation. The exploration/exploitation behavior of DRL thus helps us concentrating on "promising" partitionings by exploitation and repartitioning intelligently if we explore (i.e., we try out promising partitions in the neighborhood first). This general advantage of RL was exploited in the data management community for similar problems as well [110, 192].

To summarize, we make the following contributions:

1. We first formalize a framework that translates the partitioning problem to a DRL problem.

2. We present a two-step learning procedure to efficiently reduce the training time of our DRL agent that first bootstraps a DRL agent offline (with a simple network-centric cost model) and then refines the agent online by actually running real workloads.

3. Moreover, we propose an extension that makes use of a committee of DRL agents to improve the adaptivity of our approach for dynamic workloads. This also allows us to extend the advisor using an incremental approach if new queries are added to the workload or the database schema changes.

4. In our evaluation, we show that our approach can handle a variety of different database schemata and workloads. We also compare our approach to classical optimization-based approaches and show that our approach is able to find non-obvious solutions that outperform classical optimization-based approaches even if accurate cost estimates would be available.

OUTLINE: The remainder of this paper is organized as follows: First, in Section 7.2 we provide an overview of our approach to use

DRL to learn a partitioning advisor. In Section 7.3, we formalize the partitioning problem as a DRL problem before we introduce our training procedures in Section 7.4. We then explain how to obtain partitionings at inference time in Section 7.6, explain optimizations for workload changes in Section 7.5 and present the results of our experimental evaluation in Section 7.7. Finally, we conclude with related work in Section 7.8 and a summary in Section 7.9.

7.2 OVERVIEW

The basic idea of this paper is to train a Reinforcement Learning (RL) agent for each cloud customer that learns the tradeoffs of using different partitionings for a given database schema for different workloads. Learning these tradeoffs is appealing since cost models are known to be notoriously inaccurate [91] and would thus over- or underestimate the benefits of certain partitionings. To this end, we propose to train a DRL agent that learns the tradeoffs of using different partitionings and thus can be used to suggest a partitioning for a given customer workload. An overview of our approach is depicted in Figure 7.1.

In order to make use of our learned partitioning approach, the customer only needs to provide the DBMS (schema and data) and a sample workload that reflects the set of typical queries in a production workload. Based on this information, we train a DRL agent in an offline- and online phase (step 1 and 2). After training, the DRL agent can then be used in the production DBMS to decide which partitionings to deploy by monitoring the actual workload (observed workload). Changes in the workload might then trigger the DRL agent to suggest new partitionings that are more suited for a given workload (without retraining the agent).

In the following, we describe the high-level design of our training procedure for the DRL agent which is the core contribution of this paper. A detailed discussion about the training procedure can be found in Section 7.4. In general, DRL agents learn by interacting with an environment by choosing actions and observing rewards which they seek to maximize. In our setup, the environment is the DBMS which the agent manipulates with actions that change the partitioning of individual tables. During the training phase, the agent learns to minimize the runtime of a given workload consisting of a mix of representative queries. In the training phase, the agents thus learns the effects of different partitionings on individual query latencies.

Naïvely, we could train the agent on the customer database directly but this would require a high effort to collect the training data. For instance, repartitioning a large database table can take several minutes to complete. During the training phase the agent requires several of these actions to learn the effects. We therefore separate the training process into two phases: (1) *offline* and (2) *online* training. In the offline training phase, the agent solely interacts with a "simulation" of the customer

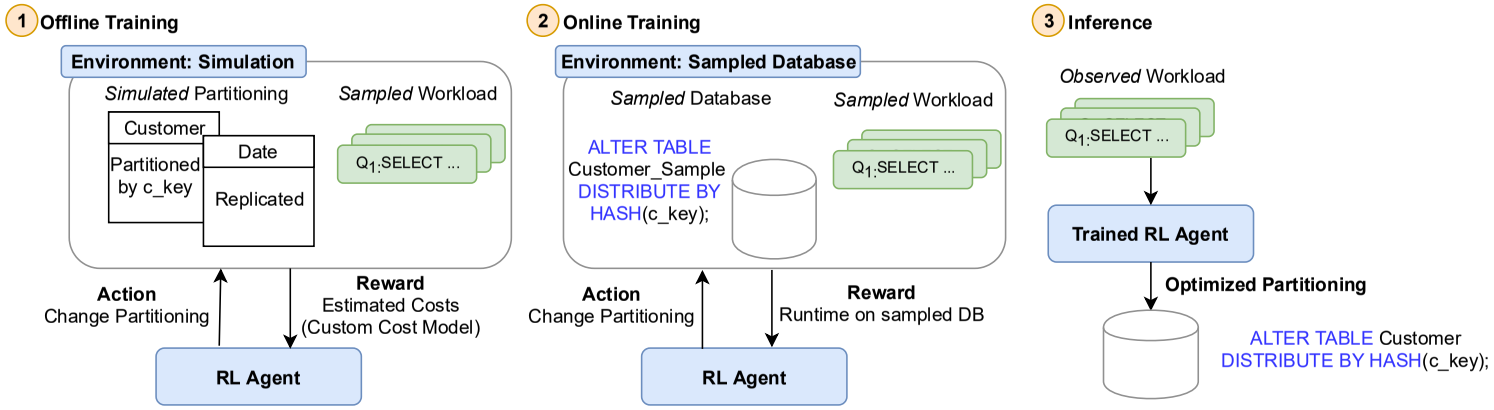


Figure 7.1: Overview of DRL-based approach to Learn a Cloud Partitioning Advisor.

database. Since the network is typically the bottleneck of distributed joins, we developed a simple yet generic cost model focused on the network overhead required to answer a query given a certain partitioning. In combination with the metadata (schema and table sizes) about the customer database, we can estimate the query costs given a partitioning in our simulation. These estimates are used as rewards for the agent. Though not precise, this bootstraps the agent and enables it to already find a reasonable partitioning given a production workload (i.e., a mix of SQL queries). In our experiment, we show that a DRL agent using this approach is already able to find partitionings that are on par with traditional optimization-based partitioning advisors that rely on DBMS internal cost models.

In an optional online training phase, the agent then does not just interact with a simulation but with a real database. However, instead of using the complete database we only use a sample of the data to speed-up this step of the training phase. The benefit of this phase is that it does not depend on the accuracy of our simple network-centric cost model anymore. Instead, we can simply measure the runtimes of queries on the sampled database to compute the rewards of the agent. Consequently, the agent learns the effects of partitionings more accurately.

Once the training is completed, we finally use the agent to make actual partitioning decisions. As input, it requires a workload, i.e. which queries were submitted in a certain time window. Based on this workload, the agent suggests partitionings which we deploy on the actual customer database. In many cases, the agent can be used directly to suggest an optimized partitioning without any further training. However, in case the database schema changes or completely new classes of queries occur in a workload our advisor needs to be retrained. In order to optimize for this case, we provide an incremental training procedure that we discuss in Section 7.5.

7.3 PARTITIONING AS A DRL PROBLEM

As discussed before, in this paper we use DRL to tackle the partitioning problem of databases. While DRL is typically used for sequential decision making, it has successfully been applied to solve classical combinatorial optimization problems [12, 78, 132] as well. The intuition of this paper is similar since the partitioning problem is indeed a combinatorial optimization problem. The main reason why RL has proven to be beneficial when applied to optimization problems is that it efficiently tackles the exploitation vs. exploration tradeoff (i.e., it more efficiently navigates the space of possible solutions instead of relying, for example, on naïve greedy search). This is especially important in our domain where collecting training data can be extremely expensive since a representative set of queries has to be executed over a potentially large database. We now discuss the required background

on Deep Reinforcement Learning (DRL) before we show how the partitioning problem can be formulated as a DRL problem including how we featurize the DBMS schema and a SQL workload.

7.3.1 *Background on DRL*

In Reinforcement Learning (RL), an agent interacts with an environment by choosing actions. Specifically, at each discrete time step t, the agent observes a state s_t. By choosing an action $a \in A$, it transitions to a new state s_{t+1} and obtains a reward r_t. Mathematically, this can be modeled as Markov decision process. The way the agent picks the actions depending on the state is called policy π. The goal of the agent is to maximize the rewards over time. However, the greedy policy, i.e. selecting the action with the highest immediate reward, might not be the best strategy. Instead, the agent might select an action that enables higher rewards in the future. Consequently, when selecting actions the agent should always keep the long-term rewards in mind [164].

One approach to solving this problem is Q-learning. With the Q-function, the expected discounted future rewards are approximated as follows if we pick action a at state s:

$$Q(s,a) = \mathbb{E}\left(\sum_{t=0}^{\infty} r_t(s_t,a_t)\gamma^t | s_0 = s, a_0 = a\right).$$

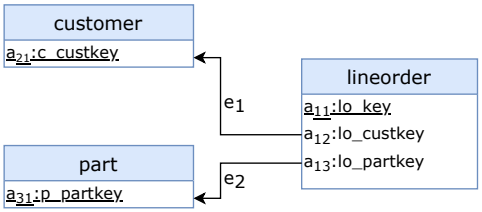
In Q-learning, the rewards are discounted with a factor $\gamma < 1$ to account for a higher degree of uncertainty for future states. The Q-function is learned during training. Note that if the approximation is good enough we can choose an optimal action for a state s as $\text{argmax}_{a \in A} Q(s,a)$.

During training we also have to select random actions such that there is a tradeoff between exploration and exploitation what we have learned so far. Usually, exploration is realized by picking a random action with probability ε. This probability is decreased over time [164] by multiplication with a factor called *epsilon decay*.

There are different ways of realizing the Q-function. For Deep Q-learning [115] (or Deep Reinforcement Learning), a neural network $Q_\theta(s,a)$ with weights θ is used for the approximation. Having observed a state s_t and an action a_t, the corresponding immediate reward r_t and the future state s_{t+1} the network is updated via Stochastic Gradient Descent (SGD) and the squared error loss

$$(r_t + \gamma \,\text{argmax}_{a \in A} Q(s_{t+1}, a_t) - Q(s_t, a_t))^2.$$

The intuition is that the expected discounted future rewards when selecting action a_t in step t should be the immediate reward r_t together with the maximum expected discounted future rewards when selecting the best action a in the next step $t+1$ discounted by γ, i.e. $\text{argmax}_{a \in A} Q(s_{t+1}, a_t)$.



q_1: SELECT * FROM customer c, lineorder l
WHERE l.lo_custkey=c.c_custkey;
q_2: SELECT * FROM part p, lineorder l
WHERE l.lo_partkey=p.p_partkey;

(a) Database and Workload

Foreign-Key Edges:
Edge e_1 for `lo_custkey`→ `c_custkey`: active
Edge e_2 for `lo_partkey`→ `c_partkey`: inactive
$s(E) = \left(e_1, e_2\right) = \left(1, 0\right)$

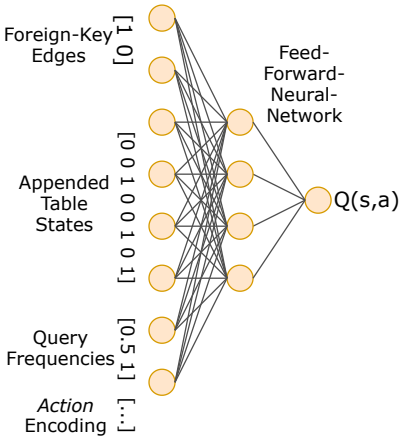
Table States:
`lineorder` partitioned by `lo_custkey`
$s(lineorder) = \left(r_1, a_{11}, a_{12}, a_{13}\right) = \left(0, 0, 1, 0\right)$

`customer` partitioned by `c_custkey`
$s(customer) = \left(r_2, a_{21}\right) = \left(0, 1\right)$

`part` replicated
$s(part) = \left(r_3, a_{31}\right) = \left(1, 0\right)$

Query Frequencies:
q_2 occurs twice as frequently as q_1
$s(Q) = \left(f_1, f_2\right) = \left(0.5, 1\right)$

(b) State Representation



(c) Q-Network with Encoded State

Figure 7.2: State Representation of Simplified SSB Schema and Workload.

7.3.2 *Problem Modeling*

In order to formulate the partitioning problem as a DRL problem we model the database and the workload as state and possible changes in the partitioning as actions. Rewards correspond to the gain in performance for a given workload. During training, the agent thus learns the impact of different partitionings on the workload. Figure 7.2 shows an example of our encoding for a simple database with three tables and a workload with two queries.

PARTITIONING STATE: The most important part of the state is to model a partitioning for a given database. For simplicity, we assume that only one partitioning scheme is used (e.g., hash-partitioning) that horizontally splits a table into a fixed number of shards (which is equal to the number of nodes in the database cluster). Moreover, replicated tables are also copied to all nodes in the cluster. In fact, these are the partitioning / replication options supported by the two DBMSs we used in our evaluation. However, in general our approach can easily be extended to more complex partitioning schemes as well. Following the assumptions that a table T_i can either be replicated or alternatively partitioned by one of its attributes $a_{i1}, a_{i2}, \ldots, a_{in}$, we can encode the state as a binary vector using an one-hot encoding $s(T_i) = \left(r_i, a_{i1}, a_{i2}, \ldots, a_{in}\right)$, where r_i encodes whether a table is replicated and the remaining bits indicate whether an attribute is used for partitioning. For instance, if the `part` table in Figure 7.2a is replicated, its state vector is $(r_3, a_{31}) = (1, 0)$ whereas the `customer` table is partitioned by the attribute a_{21} and the resulting vector is $(r_2, a_{21}) = (0, 1)$ (as shown in in Figure 7.2b).

To reduce the exploration of sub-optimal partitionings, we further extend the state representation making it explicit which tables are co-partitioned, i.e., the partitioning attributes of the tables match their join attributes. For instance, if the `customer` and `lineorder` table in Figure 7.2 are partitioned by the attributes `lo_custkey` and `c_custkey` respectively, we can join them locally on each node without shuffling. To explicitly encode co-partitioning we introduce the concept of edges; i.e., if an edge between a pair of join attributes a_{ir} and a_{js} of the corresponding tables T_i and T_j is activated, it guarantees co-partitioning. For instance, since the edge e_1 in Figure 7.2b is active the `customer` and `lineorder` tables are co-partitioned. The fixed set of possible edges E can easily be extracted from the given schema and workload (i.e., all possible join paths). Since every edge can either be active or inactive, the edge states can be represented as a fixed-size binary vector. To represent the features for the partitioning of a database with multiple tables as input for our Q-Network, we append the state vectors of all tables. For instance, the edge vectors and individual table vectors of Figure 7.2b are appended in Figure 7.2c and fed into the Q-network.

Since this input is of fixed length, we are able to use a feed-forward neural network to predict the Q-value.

WORKLOAD STATE: Moreover, we need to model the workload as part of the state since for the same database schema, different workloads result in different partitioning strategies that should be selected. Formally, a workload is a set of SQL queries $Q_1, Q_2, \ldots, Q_n$. One way to model the workload is to encode each query using different one hot encoded vectors, i.e., one vector for the set of tables, join predicates, where conditions etc., similar to [79, 161]. However, this modeling approach assumes that only queries of a typical pattern occur (e.g., queries without nesting) and thus this approach is not suited for our approach since a partitioning advisor should be trained on arbitrary workloads where the query patterns are not known in advance and complex queries involving nested queries and complex predicate conditions appear.

Encoding nested queries with the featurization as proposed in [79, 161] would be in general possible but result in an overly complex encoding with many more input vectors and a neural network structure which requires an extensive training. However, a more complex encoding is still only able to represent a fixed class of queries. Moreover, more complex encodings typically require orders of magnitude more training data.

We thus take a different route to featurize the workload based on the observation that OLAP workloads are typically composed of complex but *recurring* queries. We assume that a representative set of possible queries q_i in a workload of queries Q is known in advance which is not uncommon in OLAP workloads. To encode a specific workload, we use a vector where an entry encodes the current normalized frequency f_i of a query q_i: $s(Q) = \left(f_1, \ldots, f_m\right)$. That way, the input state can represent different query mixes. For example, since the query q_2 occurs twice as often as the query q_1 the frequency vector becomes $(0.5, 1)$ in Figure 7.2b.

Moreover, completely new queries can be supported in our state encoding without the need to train a new DRL agent from scratch. One case that we typically see in analytical workloads is that the same query is used with different parameter values resulting in different selectivities. In order to support this case, we bucketize queries into classes with different selectivity ranges and use different entries in $s(Q)$; i.e., one for each bucket. That way, if a query is used with a new set of parameter values, it is supported by finding the corresponding entry in $s(Q)$ and increasing the query frequency f_i. For supporting completely new SQL queries and not just new parameter values of existing queries in the workload, we provide entries in $s(Q)$ that are initially set to 0 (i.e., no query of this type occurs in the workload) and use those entries for new queries if they occur. We support this case in

our approach by using a committee of DRL agents that we can extend incrementally. As we show in our experiment in Section 7.7, the time required for this is only a small fraction of the original training time.

ACTIONS: A small state space is essential to apply Q-learning because we have to compute the Q-values for all possible actions to decide which action to execute in a state. We designed the actions to affect at most the partitioning of a single table. More precisely, we support two types of actions: (1) partitioning a table by an attribute or (2) replicate a table. During training, the RL agent can only select one of these actions at each step. This reduces the repartitioning costs during training since similar partitionings are observed successively.

In addition, we provide an action for (de-)activating edges as a short-cut to change the partitioning. Intuitively, activating an edge co-partitions two tables while the de-activation of edges allows follow-up actions to choose a new strategy (e.g., replication discussed above). It is important that the set of edges to be activated is conflict-free. For this, we solely allow to activate an edge if there are no two edges which require a table T_i to be partitioned by different attributes a_{ir} and $a_{ir'}$. For example, edge e_2 cannot be activated in Figure 7.2 because e_1 is already active. First, the conflicting edge e_1 would have to be deactivated.

An action a is encoded similarly to the partitioning and workload state: we use appended one-hot encoded vectors to capture the information required for an action, i.e., the kind of action (replicate, partition, (de-) activate an edge etc.), the affected table and attribute as well as the (de-)activated edge. Both the state s and an action a are then used as input for the neural network to predict the Q-value $Q(s, a)$.

REWARDS: The overall goal of the learned advisor is to find a partitioning that minimizes the runtime for the workload mix (queries and their frequencies) modeled as part of the input state. This objective has to be minimized by the DRL agent and can be used as a reward. Estimates of the simple network-centric cost model $c_m(P, q_i)$ for the queries q_i given a partitioning P are used for the offline training and actual runtimes $c_r(P, q_i)$ for the online training. Since the DRL agents seeks to maximize the reward, we use negative costs in the reward definition resulting in $r = -\sum_{j=1}^{m} f_j c(P, q_j)$.

We decided to exclude the costs of repartitioning the database as rewards into our learning procedure since we aim for setups where we expect that repartitioning does not happen that often and can be executed in the background especially for OLAP workloads and thus does not have a negative effect on the actual workload execution. In case repartitionings should be used more frequently, these cost should

be included into the rewards to prefer repartitionings that can be applied with less cost.

7.4 TRAINING PROCEDURE

In the following, we discuss the details of the offline and the online phase of our DRL-based training procedure. At the end of this section, we further discuss optimizations of our approach that allow us to provide a higher accuracy for changing workloads (i.e., if the frequency of queries change) and to incrementally add new unseen queries (and tables).

7.4.1 *Phase 1: Offline Training*

For training a partitioning advisor, the DRL agent interacts with the state reflecting the current partitioning by selecting different actions and observing rewards as described before in Section 7.3. During the offline training phase, the database partitioning is simulated and the runtimes are estimated using our network-centric cost model $c_m(P, q_i)$ approximating computation and network transfer costs of a given query q_i for a partitioning strategy P. In particular, similar to an optimizer the cost model enumerates different join orderings. For each individual join in the plan, it estimates the optimal join strategy (symmetric repartitioning join, symmetric repartitioning join, broadcast single table or co-located join) and the resulting network and computation costs. The sum of the costs is finally returned as cost estimate for the query. In our experiments, we show that based on this simple network-centric cost model, we can already train an DRL agent that is able to suggest reasonable partitionings.

Using our simple network-centric cost model as well as the state/action representation introduced before, we can train the DRL agent as described in Algorithm 1. The training is divided into sequences of states and actions of length t_{max} called episodes. The selected actions change the partitioning used for the cost estimation $c_m(P, q_i)$. Similar to typical RL implementations, the DRL agent returns to the state s_0 at the end of every episode. To guarantee that the DRL agent can find a suitable partitioning we have to make sure that it can reach any other partitioning within t_{max} steps starting from the initial partitioning s_0. Since for every table we need just one action to partition it by any attribute or to replicate it, any state can be reached within at most $|T|$ actions (where $|T|$ denotes the number of tables in the schema). Hence, we need to set $t_{\text{max}} \geq |T|$. However, as t_{max} influences the training time it is also a hyperparameter and can similarly be tuned.

Algorithm 1 Offline Training

1: Randomly initialize Q-network Q_θ
2: Randomly initialize target network $Q_{\theta'}$
3: **for** e in $0, 1, \ldots, e_{\max}$ **do** ▷ Episodes
4: Reset to state s_0
5: **for** t in $0, 1, \ldots, t_{\max}$ **do** ▷ Steps in Episode
6: Choose $a_t = \text{argmax}_a\, Q_\theta(s_{t+1}, a)$ with probability $1 - \varepsilon$, otherwise random action
7: Execute action a_t (i.e., simulate what the next state s_{t+1} and partitioning P_{t+1} would be)
8: Compute reward with cost model c_m: $r_t = \sum_{j=1}^{m} f_j c_m(P_{t+1}, q_j)$
9: Store transition (s_t, a_t, r_t, s_{t+1}) in B
10: Sample minibatch (s_i, a_i, r_i, s_{i+1}) from B
11: Train Q-network with SGD and loss $\sum_{i=1}^{b}(r_i + \gamma\, \text{argmax}_{a \in A}\, Q_{\theta'}(s_{i+1}, a) - Q_\theta(s_i, a_i))^2$
12: Decrease ε
13: Update weights of target model: $\theta' = (1 - \tau)\theta' + \tau\theta$

7.4.2 *Phase 2: Online Training*

In contrast to offline training, the idea of online training is to deploy the partitionings P_i on a database cluster and measure the true runtimes $c_r(P_i, q_i)$ to compute the reward. However, the naïve approach is way too expensive to be used in practice. Imagine for example that we need 1200 episodes for training each having $t_{\max} = 100$ steps. Assume we have just a few queries and a small schema such that the total workload takes around 20 minutes and the repartitioning takes another 20 minutes on average. If we simply executed every action, i.e., we repartition the tables and measure the workload runtimes on a cluster, we would end up with a runtime of $(20mins + 20mins) * 1200 * 100 \approx 9years$.

Therefore, online training is intended to (only) serve as refinement in addition to offline training. This has no effect if we use the same degree of exploration, i.e. if we choose random actions with the same probabilities $1 - \varepsilon$. Note that ε is multiplied with a certain factor called *epsilon decay* after every episode to decrease it over time. For online training, we start with the ε value that we would reach after 600 episodes (i.e. half of the usual amount of episodes) in the offline phase. This already significantly reduces the training costs as we will show in our experiments. However, this does not suffice to effectively reduce the time of the online phase in practice. We therefore use further optimizations which aim to minimize the online training time of the DRL agent as discussed next.

SAMPLING: Instead of using all tuples of a database, we just use a sample for every table. This speeds up both the runtime of the

queries and the time needed to repartition or replicate any table. In addition, we found it useful not to use the runtimes of a query $c_r(P, q_i)$ directly but to multiply this with a certain factor for every query. The intuition is that some queries scale better than others on the full dataset. Hence, runtime improvements of queries that scale better and thus also run fast on the full dataset should weigh lower than improvements of queries which are very slow on the full dataset. To this end, we measure the runtimes of each query q_i for the partitioning $P_{offline}$ found in the offline phase once for the full dataset $c_{full}(P_{offline}, q_i)$ and once for the sample $c_{sample}(P_{offline}, q_i)$. Afterwards, we scale the costs for each query q_i with the corresponding factor $\mathcal{S}_i = \frac{c_{full}(P_{offline}, q_i)}{c_{sample}(P_{offline}, q_i)}$.

One question is how many tuples have to be sampled per table, i.e. how the sampling rate is chosen. Higher sampling rates result in a longer runtime of the online phase since both the query runtimes as well the repartitioning times will increase. In contrast, smaller sampling rates might lead to suboptimal partitionings. This can happen if partitionings P' have shorter weighted runtimes $\mathcal{S}_i c_{sample}(P', q_i)$ on the sample than a superior partitioning P^* for the full dataset. We can account for these cases by selecting several partitionings $P_1, \ldots, P_n$ and measure their runtime both for the sample and for the full dataset. If partitionings with shorter weighted runtimes on the sample also lead to shorter runtimes on the full dataset size the sampling rate is sufficient. If not, the sample size has to be increased. As a simple heuristic one can empirically determine a threshold below which table sizes should not fall below after sampling. This guarantees that tables have a certain minimum size. If this threshold is large enough, optimal partitionings on the samples will also be optimal on the full dataset with high probability. A cloud provider could empirically determine this threshold for every database and hardware setup.

QUERY RUNTIME CACHING: If the DRL agent visits two states s_i and s_j during training which have the same corresponding partitioning P, the runtimes do not have to be measured twice. Hence, we can cache query runtimes to faster compute recurring reward values. Additionally, if the partitionings of the states s_i and s_j differ only for a certain set of tables $\{T_{i1}, T_{i2}, \ldots, T_{in}\}$ we only have to measure the runtimes of queries q_i that contain at least one of these tables. In particular, the runtime of every query q_i containing the tables $\{T_{i1}, T_{i2}, \ldots, T_{in}\}$ depends only on the states of these tables, i.e. $s(T_{i1}), s(T_{i2}), \ldots, s(T_{in})$. Hence, for every query we can maintain a table containing the different state combinations $s(T_{i1}), s(T_{i2}), \ldots, s(T_{in})$ and the runtime of the query on the sample dataset. In summary, when visiting a new state we examine the state of every table $s(T_i)$; i.e. whether it is replicated or hash-partitioned by a certain attribute, and run only the queries q_i for which we do not have a runtime entry for the state combination of relevant tables $s(T_{i1}), s(T_{i2}), \ldots, s(T_{in})$.

LAZY REPARTITIONING: The approach of lazy repartitioning is to keep track of the partitioning deployed on the database P_{actual} and the partitioning P_t of the state s_t the agent is currently at. Every time the agent chooses an action and we reach a new state we first check which queries $\{q_{j1}, \ldots, q_{jn}\}$ have to be executed on the database. Especially in later phases of training this will be significantly fewer queries than the full set Q since many runtimes will be in the Query Runtime Cache. For this set we determine the set of tables $\{T_{i1}, \ldots, T_{im}\}$ which are contained in these queries. Only if P_{actual} and P_t do not match for one of the tables, we actually repartition the table.

TIMEOUTS: The idea of this optimization is that a partitioning where a single query exceeds a certain time limit cannot be optimal. Hence, we can safely abort the query execution and move on with training. Recall that the reward for a partitioning P for online training is defined to be $r = -\sum_{j=1}^{m} f_j \mathcal{S}_j c_{sample}(P, q_j)$. We can similarly compute the (online) reward $r_{offline}$ of the partitioning $P_{offline}$ found in the offline phase. If a query q_i takes longer than $-r_{offline}/(\mathcal{S}_i \cdot f_i)$ we can safely abort it since the corresponding partitioning will definitely result in a lower reward. If we are aware of a partitioning with an even higher reward r', the timeout can further be reduced to $-r'/(\mathcal{S}_i \cdot f_i)$.

7.5 OPTIMIZATIONS FOR WORKLOAD CHANGES

In the following, we discuss two enhancements for training the partitioning advisor: (1) using a committee of experts rather than a single agent to further increase the capacity for workloads with many tables and queries, (2) using incremental training to adjust a learned advisor if new queries occur in the workload.

COMMITTEE OF EXPERTS: The main goal of our approach is to train a DRL agent just once such that it generalizes over different workload mixes (i.e., different query frequencies). If the workload mix changes, we want to use inference of the trained DRL agent and obtain a new partitioning that works better for the new workload mix.

A more advanced approach enabling more accurate results for a wide variety of workloads (i.e., large query sets) is not to train only a single RL agent to suggest partitionings for all possible frequency vectors but to use several expert models for subsets of all possible workload mixes. Using more models allows experts to specialize on certain aspects of the problem and moreover increases the overall capacity of the model. The related ensemble approach is a common optimization in machine learning to optimize the model performance. The question is how the workload space can be partitioned efficiently into different expert models. In the following, we explain our approach called *DRL subspace experts*.

The main idea of DRL subspace experts is to first obtain so called reference partitionings $\tilde{P}_1, \cdots \tilde{P}_n$ which are optimized for certain workloads. To find these, we use the inference procedure of the naïve model (i.e., the RL agent which was trained for the whole workload space) and ask this agent for the optimal partitioning using m frequency vectors where one query q_i is over-represented: $f_1, \ldots, f_{i-1}, f_i, f_{i+1}, \ldots, f_m$ with $f_j = f_{low}$ for $j \in \{0, 1, \ldots, i-1, i+1, \ldots, m\}$ and $f_i = f_{high}$. The main intuition is that individual queries might favor opposing partitioning strategies that we aim to simulate by "extreme" frequency vectors. Since many queries share the same reference partitioning, the number of distinct partitionings n is much smaller than the number of queries m (i.e., $n << m$). The distinct partitionings resulting from this step is the set of reference partitionings $\tilde{P}_1, \cdots \tilde{P}_n$.

For example, for a given workload with 10 queries we would sample 10 frequency vectors each representing a workload were one query is over-represented. We then use these to obtain the reference partitionings from a naïve model with only one agent. Based on these 10 frequency vectors, we might end up having just three different reference partitions $\tilde{P}_1, \tilde{P}_2$ and $\tilde{P}_3$.

Once we determined the reference partitionings, we can separate the workload space, i.e. the set of different frequency vectors. We say that a frequency vector $(f_1, \ldots, f_m)$ belongs to the frequency subspace of one of these reference partitionings $\tilde{P}_i$ if

$$\tilde{P}_i = \text{argmax}_{\tilde{P} \in \{\tilde{P}_1, \cdots \tilde{P}_n\}} - \sum_{j=1}^{m} f_j \mathcal{S}_j c_{sample}(\tilde{P}, q_j),$$

i.e., if the reward of the naïve RL agent is the maximum among $\tilde{P}_1, \cdots \tilde{P}_n$ for this frequency vector. Afterwards, we then train one DRL agent for each of these subspaces. The resulting DRL agents can be considered experts for their frequency subspace. For each of the frequency subspaces the training is similar to training the DRL agent for the naïve approach. The only difference is that the DRL agents are only trained for frequencies of their dedicated subspace. One problem is how to sample more frequency vectors from the same subspace. To obtain frequency vectors for different subspaces, we sample frequencies uniformly and assign each frequency vector to the DRL agent for the respective reference partitioning $\tilde{P}_i$.

An important aspect is that the training of these subspace expert models does typically not require any actual execution of queries on the database cluster since we can reuse query runtimes in the Query Runtime Cache of the naïve approach. When training the subspace expert models, however, we might encounter partitionings that were not seen when training the naïve model. For these cases, we have no entries in the Query Runtime Cache and the queries need to be actually executed. However, these cases are rare since the naïve agent visits all optimal or near-optimal partitionings with high probabilities already.

INCREMENTAL TRAINING: A final interesting aspect of our online approach is that we can easily support new queries by incremental training. The main idea is that if new queries are added to a workload, we do not have to train a new model from scratch. Instead, we add new inputs representing the query frequencies to the input state of the naïve model and retrain it only with frequency vectors that include the new queries. Again, the Query Runtime Cache can be reused and we only require actual runtimes for the new queries. Afterwards the naïve model can be used again to obtain the new reference partitionings. Only if a new reference partitioning is found, we have to train a new expert agent for that subspace. Otherwise, it is sufficient to refine the existing subspace experts with the cached query runtimes.

7.6 MODEL INFERENCE

Having trained the learned partitioning advisor, we now describe how it can be used to suggest a partitioning. This can either be the case if an initial partitioning of the database should be suggested or the workload changes. We first assume that only one DRL agent is trained before we explain how the inference works if a committee of experts is used.

INFERENCE WITH ONE DRL AGENT: We assume that a frequency vector is given that represents the current workload mix. The intuition is to fully exploit the knowledge of the trained agent by always selecting the partitioning action with maximum expected future rewards, i.e. the highest Q-value.

When applying the inference procedure, we always start with the same initial state s_0 also used during training. From the initial state s_0, we iteratively choose the action that maximizes the Q-function, i.e., $a_t = \text{argmax}_a Q_\theta(s_{t+1}, a)$. For this, we enumerate all possible actions in that state and evaluate the neural network for each action. Since we designed the action space to be small, this is very efficient. Every time we choose an action, this changes the state s and thus the partitioning. Note that we do not have to deploy every state in this sequence. Instead, we use the same simulation that is also used in the offline phase. Consequently, we execute $t_{\max}$ actions and thus obtain a sequence of actions $(s_0, a_0, r_0, s_1, \ldots, s_{t_{\max}}, a_{t_{\max}}, r_{t_{\max}})$. Afterwards, we do not simply suggest the partitioning represented by the last state $s_{t_{\max}}$, since the DRL agent tends to oscillate around the best partitioning P^* (i.e, the partitioning with the highest reward is not necessarily represented by the last state). Instead, we identify the state s_t in the sequence above with a maximum reward and return the corresponding partitioning P^*.

INFERENCE WITH A COMMITTEE OF DRL AGENTS: If we want to obtain a new partitioning when a committee of experts was trained,

we first determine which subspace $\tilde{P}_i$ of the frequency space the vector belongs to: $\tilde{P}_i = \text{argmax}_{\tilde{p} \in \{\tilde{p}_1, \cdots \tilde{p}_n\}} - \sum_{j=1}^{m} f_j \mathcal{S}_j c_{sample}(\tilde{P}, q_j)$. The DRL agent is selected by choosing the DRL agent for the reference partitioning with the lowest estimated runtime (which is the same procedure we use when training the expert models). Afterwards, we use the inference procedure discussed before with the corresponding expert model for $\tilde{P}_i$.

7.7 EXPERIMENTAL EVALUATION

In the following, we evaluate the benefits of using learned partitioning advisors for databases with schemas of varying complexity. We study the following aspects of our approach:

1. **Performance after Offline Training**. In the first experiment (Section 7.7.2), we validate that DRL agents that are trained purely offline find partitionings outperforming typical heuristics and are competitive with those found by state-of-the-art partitioning advisors.

2. **Improvement due to Online Training.** Furthermore, if additionally trained online, the DRL agent clearly outperforms state-of-the-art systems and finds non-obvious partitionings with superior runtime as we demonstrate in our second experiment (Section 7.7.3). We moreover study the isolated runtime savings of our suggested optimizations of the online phase.

3. **Adaptivity to Data and Workload.** Another benefit of our approach is the flexibility w.r.t. changes in the workload (Section 7.7.4). Hence, in the third experiment we first show that the committee of experts can suggest partitionings that improve over the naïve model for changing workloads. Furthermore, we examine the additional training time required if new queries are added to a workload and the effect of database updates.

4. **Other Learned Approaches.** We empirically validate that using DRL for the partitioning problem is superior to learning a neural cost model (Section 7.7.5) which is minimized for a given workload to find suitable partitionings.

5. **Adaptivity to Hardware Characteristics.** Finally, in the last experiment (Section 7.7.6) we show that our agent can also adapt to changes in the deployment (i.e., if hardware characteristics change) which is not trivial with existing approaches.

7.7.1 *Workloads, Setup and Baselines*

For the experiments, we used different databases and workloads that we explain in the following. Moreover, we also discuss the learning

Parameter	Value
Learning Rate	$5 \cdot 10^{-4}$
τ (Target network update)	10^{-3}
Optimizer	Adam
Experience Replay Buffer Size	10000
Batch Size for Experience Replay	32
Epsilon Decay	0.997
t_{max} (Max Stepsize)	100
Episodes	600/1200
Network Layout	128-64
γ (Reward Discount)	0.99

Table 7.1: Hyperparameters used for DRL training.

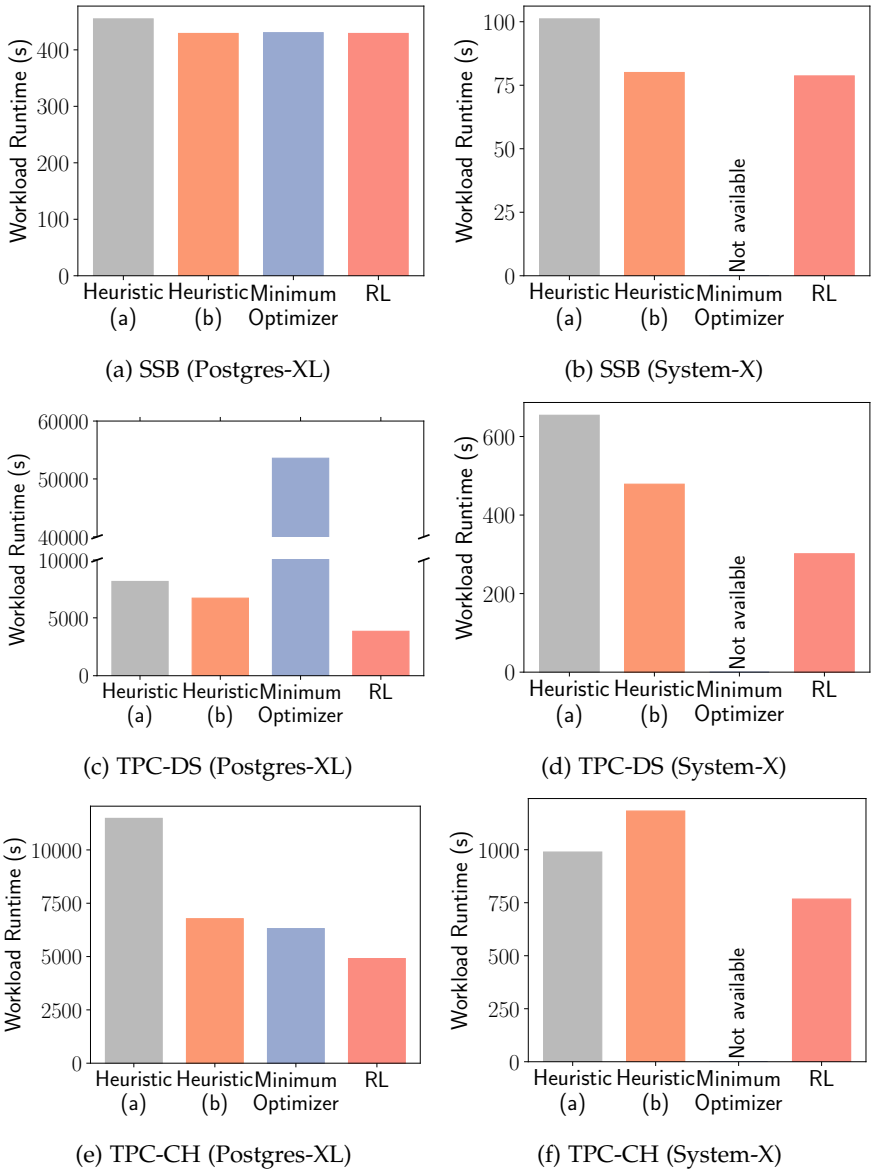
setup that we used for training the partitioning advisors as well as the baselines.

DATA AND WORKLOADS: We evaluated the partitioning advisor on three different database schemas and workloads varying in complexity: (1) As the simplest case, we used the Star Schema Benchmark (SSB) and its workload [129]. SSB is based on TPC-H and re-organizes the database in a pure star schema with 5 tables (1 fact and 4 dimension tables) and 13 queries. (2) The second database and workload we used was TPC-DS [168]. TPC-DS comes with a much more complex schema of 24 tables (7 fact and 17 dimensions tables) and 99 queries (including complex nested queries). For Postgres-XL, which is one of the systems used in the evaluation, only 60 of the 99 queries could be executed due to restrictions in which queries it supports. (3) In cloud data warehouses such as Amazon Redshift, customers are not required to use a star schema but can design an arbitrary schema for their database. To test how well our learned advisor can cope with more complex schemata which are not based on a star-schema, we additionally used the TPC-CH benchmark [47], which is the combination of the schema of the TPC-C benchmark with analytical queries of the TPC-H schema (adopted for the TPC-C schema). Originally, the TPC-CH benchmark combines analytical queries and transactions in a mixed workload. For the purpose of this paper, we only used the analytical queries to represent the workload in our evaluation. Furthermore, in the standard version of TPC-CH all tables can be co-partitioned by the `warehouse-id`. While our DRL agents also propose this solution when using the original TPC-CH schema, we do not think that such a trivial solution is realistic for many real-world schemata. Hence, we further added complexity and decided to restrict possible partitionings such that tables cannot be partitioned by `warehouse-id` only. For all benchmarks (SSB, TPC-DS, and TPC-CH), we used the scaling factor SF=100.

SETUP: The partitionings for different analytical schemas were evaluated on two database systems. To show that our learned approach is in general applicable to both disk-based and memory-based distributed databases, we used Postgres-XL 10R1.1 (a popular open-source distributed disk-based database) [142] and System-X (a commercial distributed in-memory database). For running the databases in a distributed setup, we used CloudLab [28], a scientific infrastructure for cloud computing research. For our experiments, we provisioned clusters of different sizes ranging from 4 to 6 nodes. Each node was configured to use 128GB of DDR4 main memory, two Intel Xeon Silver 4114 10-core CPUs and a 10Gbps interconnect. The partitioning advisor is built using neural networks implemented in Keras. In particular, the neural network to approximate the Q-functions used 2 hidden layers with 128 and 64 neurons, respectively. We used the standard ReLU activation function in every layer and a linear function for the output (to represent the Q-value) which is a common combination for DRL. An overview of all hyperparameters which we found to work best for training can be seen in Table 7.1. The only hyperparameter we changed for the different databases was the amount of episodes we used to train the model. Since SSB has a significantly lower amount of tables and queries we only trained the DRL agents for 600 episodes instead of 1200 episodes for TPC-DS and TPC-CH.

BASELINES: Previous approaches typically use the optimizer cost estimates or heuristics to optimize the partitioning design [3, 127, 145]. We additionally compared the partitionings found by our approaches to heuristics that are typically used by a database administrator [191]. For both simple and more complex star schemata (SSB and TPC-DS) this means that usually fact tables are co-partitioned with either the most frequently joined dimension table (Heuristic (a)) or the largest dimension table (Heuristic (b)). For the more complex schema TPC-CH, we either naïvely replicated small tables and partitioned larger tables by primary key (Heuristic (a)) or greedily co-partitioned the largest pairs of tables while still replicating smaller tables (Heuristic (b)).

Automated partition designers [3, 127, 145] usually make use of the optimizer cost estimates, i.e. they enumerate different physical designs, let the optimizer estimate the costs for all queries in the workload and choose the partitioning candidate with minimal costs. While several optimizations exist that make the integration of the optimization and the database cost estimation tighter (e.g., Nehme et al. [127] make use of the MEMO data structure in Microsoft SQL server), they still suggest the partitioning with minimal query optimizer cost estimates. As a second baseline, we thus implemented a similar optimization algorithm enumerating candidate solutions and minimizing the optimizer cost estimates for Postgres-XL. However, for System-X this was not possible because the optimizer cost estimates are not accessible.



(a) SSB (Postgres-XL)

(b) SSB (System-X)

(c) TPC-DS (Postgres-XL)

(d) TPC-DS (System-X)

(e) TPC-CH (Postgres-XL)

(f) TPC-CH (System-X)

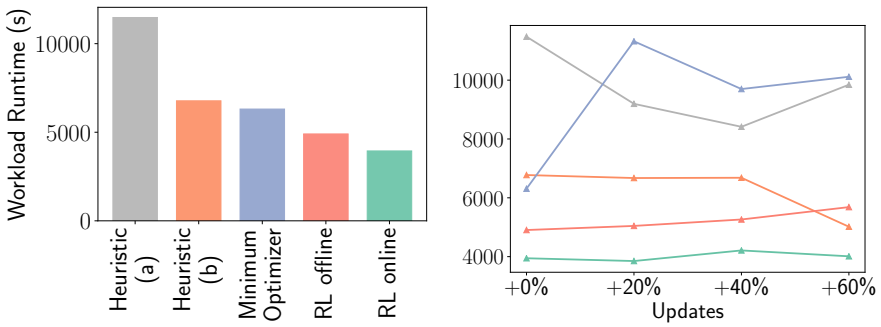
Figure 7.3: Offline RL vs. Baselines.

Cloud providers offering multiple commercial DBMS systems face similar problems and thus this approach is not available for System-X.

7.7.2 *Exp. 1: Offline Training*

For each database mentioned before in the setup, we trained a dedicated DRL agent with offline training, i.e. using our simple network-centric cost model. We report the averaged total runtime of all queries for five runs for the partitionings suggested by our advisor and the baselines in Figure 7.3.

RESULTS FOR SSB: For the SSB benchmark, the two heuristics co-partition the fact table with either the most frequently joined dimen-



(a) TPC-CH (b) TPC-CH with Updates

Figure 7.4: Online RL vs. Baselines.

sion table (`Date`) or the largest dimension table (`Customer`). The optimizer predicts minimal costs when partitioning the lineorder table by primary key and replicating all dimension tables. Our learned advisor also suggests to co-partition the fact table with the largest dimension table for Postgres-XL (same as Heuristic (b)). For System-X our learned advisor additionally suggests to partition the `Part` dimension table by its primary key leading to a minimal runtime improvement.

RESULTS FOR TPC-DS: For TPC-DS, which is a more complex schema composed of several fact tables with shared dimensions, the DRL agents finds superior solutions that are non-obvious. Here, the improvements are more significant reducing the runtime over Heuristic (a) by approximately 50%. For both Postgres-XL and System-X, the DRL agents propose to co-partition the fact tables with a medium-sized dimension table, i.e. `Item`. This has the advantage that local joins are possible if two fact tables are joined, e.g. the fact tables for `StoreSales` and `StoreReturns`. Moreover, for System-X the `Customer` table is co-partitioned with the `CustomerAddress` table allowing local joins. In contrast, the partitioning with the minimal optimizer costs for Postgres-XL leads to a suboptimal partitioning. This is due to the high query complexity resulting in erroneous cost estimates.

RESULTS FOR TPC-CH: As discussed before, TPC-CH uses a significantly more complex schema than SSB and TPC-DS since it is not similar to a star schema. While Heuristic (b) has better runtimes than Heuristic (a) on Postgres-XL, Heuristic (a) outperforms Heuristic (b) on System-X. This counter-intuitive result is due to the fact that partitioning a table by `district-id` (as Heuristic (b) does) results in skewed partition sizes in System-X. Compared to the two heuristics, the DRL-agent proposes improved partitionings. For Postgres-XL it proposes to co-partition the `Customer`, `Order`, `NewOrder` and additionally the `Orderline` table by `district-id` but to replicate the `Stock` table. This avoids that `Orderline` has to be shuffled over the network for a join. For System-X, the DRL agent additionally partitioned the

Optimizations	Training Time	Speedup
None	4621h	-
+ Runtime Cache	1160.4h	4.0
+ Lazy Repartitioning	60h	19.3
+ Timeouts	33.4h	1.8
+ Offline Phase	13.3h	2.5

Table 7.2: Training Time Reduction of Optimizations.

`Stock` table but also used a compound key combining `warehouse-id` and `district-id` to mitigate the skew (which was reflected in the simple network-centric cost model).

7.7.3 *Exp. 2: Online Training*

In this experiment we evaluate whether DRL agents trained online are superior over purely offline-trained agents. We focus on the most complex schema, i.e. TPC-CH, and Postgres-XL to analyze the additional online phase that leverages actual runtimes instead of cost estimates. For the online training we refine the DRL agent that was already bootstrapped with our simple network-centric cost model offline.

The runtime of the benchmark queries using the suggested partitionings on the full TPC-CH database are shown in Figure 7.4a. The partitioning suggested by the online-trained agent is 20% superior to the partitioning of the offline-trained agent. The online-trained DRL agent suggests a new partitioning where the `NewOrder`, `Order` and `Orderline` table are co-partitioned by `Order-Id` and the `Customer` table is replicated in addition. Interestingly, this partitioning has higher costs according to our simple network-centric cost model. However, the online phase is not affected by the inaccuracy of our simple network-centric cost model and was thus able to improve over the offline-trained agent.

If executed naïvely, the online training phase is time-consuming. We thus want to examine the effect of different optimizations. For this experiment, we were only running the training with all optimizations (except timeouts) activated. By keeping track of the queries that would be executed twice without *Runtime Caching*, as well as how often a table would be repartitioned without *Lazy Repartitioning* and how much time could be saved with a particular *Timeout*, we could determine the savings of the optimizations. As we can see in Table 7.2 every optimization significantly reduces the runtime and the largest improvement can be obtained with *Lazy Repartitioning*. The last optimization compares the training time of an agent that was bootstrapped in an offline phase with a randomly initialized agent.

The online-phase with all optimizations and for a model that was bootstrapped offline took 13.3 hours. We believe that a training time of several hours is acceptable since the model has to be trained only

once for different workload mixes and can afterwards be used as a partitioning advisor if the workload changes (as we show in the next experiment). Moreover, especially in cloud setups, we can easily clone the instances. Hence, setting up a similar cluster to retrain the agent for several hours to obtain a refined model should be feasible considering that customers usually have one cluster provisioned all the time to do analytics. The cloning is especially efficient for our setup since we do not have to clone the entire data but only a sample of each table.

7.7.4 *Exp. 3: Adaptivity to Data & Workload*

The following experiments validate the adaptivity of a DRL agent to changing data and workloads. We first demonstrate that our approach can still find optimal partitionings without additional training even if the data and the mix of queries changes. Moreover, in a last experiment we examine the additional training time required if completely new queries are added to the workload.

EXP. 3A: CHANGING DATA: First, we evaluated how robust the trained RL agent is if the data changes. In this experiment, we use the TPC-CH schema as before and train the RL advisor on the full database (100%). Afterwards, we update the DBMS and bulk load up to 60 % of new data into the TPC-CH schema. We use the bulk update procedure of TPC-H and transform the data to the TPC-CH schema since our main focus is on warehousing and the TPC-CH benchmark does not support bulk updates. Figure 7.4b shows the results of using our online-trained RL advisor (without any retraining) compared to all other baselines. The large deterioration of the "minimal optimizer" baseline in the measurement is due to different query plans chosen by the PGXL optimizer after updates are applied. As we can see, the partitioning found by the RL advisor constantly performs best even for relatively large update rates of up to 60%. However, if the database significantly changes, we need to retrain our advisor (which is not needed in this experiment though). A helpful indicator to decide when retraining is needed might be a change of the query plan. Moreover, there exists a huge body of work in ML to detect drifts in training data (which is related to this problem). Developing techniques to robustly detect when to retrain is an interesting avenue for future work though.

EXP. 3B: CHANGING WORKLOAD MIX: In this experiment we show that our learned advisor finds optimal partitionings for different query mixes. To this end, we trained an DRL agent with the naïve approach for different workload frequencies for the TPC-CH schema. Moreover, we additionally trained a committee of experts for the subspace experts approach as described in Section 7.5. In any case, the advisor only has to be trained once and generalizes to different workloads as we will

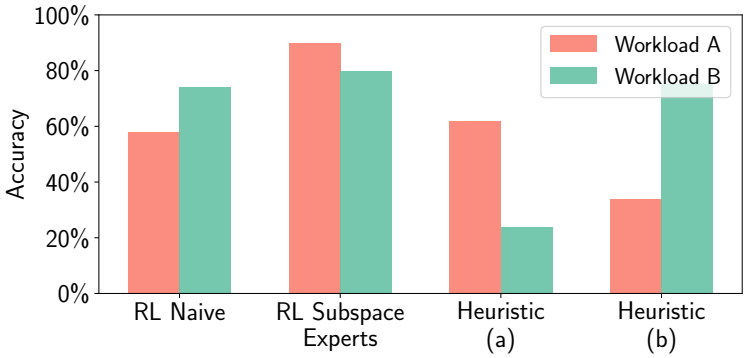


Figure 7.5: Best Partitioning found by Different Approaches for Varying Workloads (higher is better).

show in this experiments. For the naïve model and the committee of experts, we both used an online training phase on Postgres-XL. Note that we can apply all optimizations for the online training as well. Moreover, we can reuse the Query Runtime Cache of the previous experiments if we train multiple experts.

After training both approaches, we report the percentage of correct partitionings for two different workloads clusters in Figure 7.5. Each cluster is a set of different frequency vectors (i.e., workload mixes): for cluster A the frequencies were sampled uniformly and for cluster B queries joining the `Stock` and the `Item` tables are more likely to occur. If the partitioning found by either approach is best for the respective cluster, we say that the approach has found the optimal partitioning for this workload mix. We compare the naïve approach and the subspace experts approach with two heuristics. Heuristic (a) always chooses the optimal partitioning found after online training in the previous Section. Heuristic (b) always chooses a partitioning where the `Stock` and `Item` tables are co-partitioned. The results are given in Figure 7.5. As we can see, the accuracy can significantly be improved when using subspace experts outperforming all other approaches. We conclude that is beneficial to divide the problem of finding an optimal partitioning for a given workload into subproblems which are then solved by the dedicated expert model. This is due to the well known technique of using ensembles of ML models to improve the performance.

EXP. 3C: NEW QUERIES: In our formulation of the problem we decided not to encode the complex nested queries typically occurring in OLAP-workloads to avoid an overly complex neural network architecture requiring too much training data. Instead, we represent the workload as frequencies of a representative set of queries. Once trained, our learned partitioning advisor can suggest partitionings for any of those workloads. However, if completely new queries occur that have a significant impact on the workload runtime and do not have a similar query in the set of representative queries we need incre-

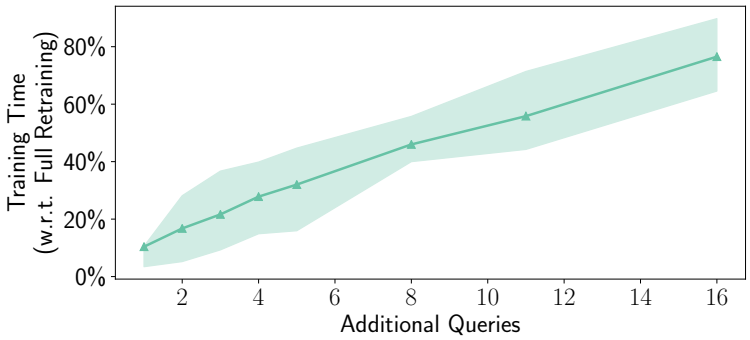


Figure 7.6: Training Time of Additional Training (relative to Full Retraining) with 25% and 75% Quantiles.

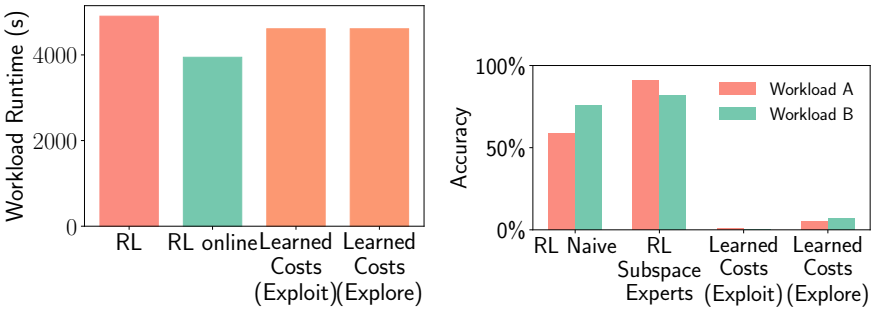
mental training, i.e. we train the agent for workloads where these new queries occur which is significantly faster than training a new agent from scratch. In this experiment, we evaluate the additional training overhead if such new queries are introduced. In particular, it proves that additional training is much cheaper than training the agent from scratch if new queries occur.

We again trained a committee of experts for TPC-CH on top of Postgres-XL as the underlying database. However, in contrast to the previous experiment, we first randomly removed a fraction of the queries of the TPC-CH benchmark. We then retrained the advisor for the additional queries and calculated, with the help of already measured runtimes, how long such an additional training takes on average if part of the workload is not known initially.

Figure 7.6 shows the time for incremental training relative to the time required to train an DRL agent from scratch, depending on how many additional TPC-CH queries were added after the initial training. As we can see, the overhead of incremental training is much lower than training a partitioning advisor from scratch. This is because, similar to exploiting a bootstrapped DRL agent using the offline phase, we can start with a lower ε-value in the incremental training of the new naïve model resulting in fewer explorations. In addition, incremental training can also make use of the Query Runtime Cache, which keeps actual query execution to a minimum as many queries are already known.

7.7.5 *Exp. 4: Other Learned Approaches*

An alternative to using RL is to learn an ML model to predict the costs of a partitioning and use a classical optimization procedure to select the best partitioning for a given workload. Recently, learned cost models have been used for query optimization [112] or cardinality estimation [79]. For instance, [112] iteratively choose optimal query plans according to their neural cost model. In the following, we explain how we implemented the neural cost model for partitioning.



(a) TPC-CH Schema (b) Workload Adaptivity

Figure 7.7: RL vs. Neural Baselines.

Similar to our offline phase, we first use an offline bootstrapping step where the neural cost model is trained using runtime estimates based on our simple network-centric cost model. We use 100k workload/partitioning pairs for the offline phase since this is equivalent to the number of workload/partitioning pairs that our RL agent sees in its offline phase. Afterwards, analogous to our online training, we then run a workload mix on a real DBMS to improve the neural cost model using actual runtimes.

For the online training, we use multiple iterations, where we repartition the database to minimize the current cost model in every iteration. We then retrain the neural cost model with the runtimes collected on the partitionings observed during the iteration and then start the next iteration (i.e., sample a new workload and again minimize the cost model). To be fair, we allow the same overall training time for both approaches in the online phase (RL and the neural cost models) and also enable all optimizations we also use for our RL agent (e.g., runtime caching etc.). To simulate a more exploration-driven variant in contrast to the exploitation-driven variant above which selects the best partitioning in each iteration, we also implemented a variant that starts with a random partitioning in every iteration.

To show the efficiency we compare the runtime of the partitioning schemes suggested by the online-trained neural cost models, our online-trained RL agent and the RL agent that was only trained offline. In this experiment, we use the same workload (i.e., TPC-CH) as in Exp. 2. As a result, we can see in Figure 7.7a that the online-trained neural cost models (i.e., both the exploitation- and the exploration-driven variant) improve the offline-trained RL agent by only 6% while our online-trained RL shows an improvement of 20% compared to the offline-trained RL agent. Moreover, we also tested how well the neural cost model generalizes to new (unseen) workloads by using the same setup as in Exp. 3 where we sample new workloads uniformly. As we can see in Figure 7.7b the online-trained neural cost model only finds the optimal partitioning in 5% of the cases (vs. 91% for online-RL) for

workload A and 7% of the cases (vs. 82% for online-RL) for workload B.

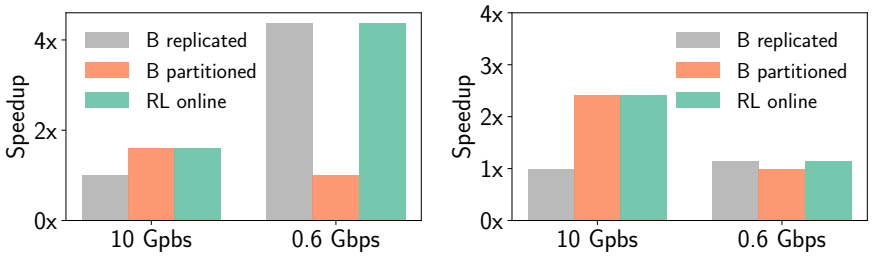
We investigated why the neural cost model approach does not perform as well as our RL agent in both experiments above. The reason is that our RL agent observed three times as many different partitionings as the learned cost model in the same training time. This effectively means that our RL agent explores partitionings with a lower average runtime and shows that the exploration/exploitation strategy of our RL agent actually leads to a more efficient navigation through the solution space.

7.7.6 *Exp. 5: Adaptivity to Deployment*

Another advantage of using an DRL agent as partitioning advisor is that it can adapt the partitioning for different deployments which is an important scenario for cloud providers that allow customers to migrate their cluster to a new set of virtual machines with different characteristics. For showing the adaptivity of our learned advisor, we created a simple microbenchmark to empirically validate this. It consists of three relations `A`, `B` and `C` where `A` is a fact table and `B` and `C` are dimension tables. The relation sizes are inspired by the relation sizes of the `Lineorder`, `Order` and `Partsupp` table of the TPC-H benchmark. The workload consists of just two queries joining the fact table `A` with one of the dimension tables `B` or `C` with selectivities between 2% and 5%.

In the optimal partitioning, table `A` and `C` have to be co-partitioned because `C` is significantly larger than `B`. Depending on the network bandwidth, however, it might be optimal to either partition or replicate table `B`. For example, for a high-bandwidth network it might be beneficial to partition `B`, say, on its primary key. When joined with table `A` the scan of table `B` can be distributed among all cluster nodes (if the table is partitioned) and the remaining tuples have to be shuffled over the network. If table `B`, however, is replicated we do not have to send tuples over the network for the join but the scan is also not distributed across nodes. Hence, the question whether or not partitioning is beneficial depends on the speed of the network compared to the scan speed of the table. As network costs are more significant if one does not need costly disk accesses we decided to use System-X for the evaluation which is an in-memory database.

To show the effect, we used two different hardware deployments for System-X. One time, we used the usual 10 Gbps interconnect, one time we only used 0.6 Gbps interconnects. This is also the bandwidth offered for the basic deployment of Amazon Redshift. We trained one DRL agent on the full dataset (approx. 100 GB) for the two hardware deployments. In Figure 7.8 the effects of partitioning or replicating table `B` can be seen for both the slow and the fast network. In the



(a) Standard HW (b) Slower Compute

Figure 7.8: Adaptivity to Deployment.

Figure, we use the slowest approach of both as reference and show the speed-up of the others (i.e., higher is better). As we can see, for the slow network it is optimal to replicate table B, while for the fast network it is better to partition it.

We repeated the experiment on less powerful hardware (nodes with a 32-core AMD 7452 CPU and 128GB ECC Memory (8x 16 GB 3200MT/s RDIMMs)) in Figure 7.8b. In this case, the benefit of replicating table B is less significant for the slow network since the scan costs are more dominant. However, in all cases the DRL agent (after retraining the model on the hardware setup) suggests the optimal solution.

7.8 RELATED WORK

PARTITIONING FOR OLTP AND OLAP: Many approaches focus on transactional workloads [25, 29, 45, 139, 143]. In general, these approaches partition the data such that distributed transactions across nodes occur less frequently. For example, SCHISM [29] defines a graph consisting of tuples as nodes and transactions as edges and uses a min-cut to partition the tuples. Pavlo et al. [139] developed an alternative approach that is also capable of stored procedure routing and replicated secondary indexes. Fetai et al. focus especially on cloud environments [45].

For OLAP-workloads Eadon et al. [40] proposed REF-partitioning, i.e., to co-partition chains of tables linked via foreign key relationships. Since this technique can be exploited if a system supports hash-partitioning by any attribute most partitioning advisors and also our technique indirectly make use of REF-partitioning. Zamanian et al. [191] extend this approach such that even more locality can be obtained but at the cost of higher replication. For this, the database has to support predicate-based reference partitioning. In contrast, [103] iteratively improves the partitioning and relies on hyper-partitioning and hyperjoins. However, these features are currently not supported by Postgres-XL or System-X and could thus not be evaluated.

AUTOMATED DATABASE DESIGN: Automatic design advisors are an active area of research [3, 127, 144, 145, 196]. However, many of these approaches [145, 196] focus only on single-node systems while only a few advisors for distributed databases are specialized on partitioning design [3, 127, 145]. These approaches, however, rely only on the cost model of the optimizer which is often inaccurate [91]. As in query optimization, this can result in wrong decisions [93] since the benefit of some query plans (partitionings) is over- or underestimated. Different from these approaches we developed a simple network-centric cost model and a dedicated online phase that is able to cope with inaccuracies of the cost model.

Even worse, some databases do not provide access to the cost estimates of the query optimizer at all such as System-X in our experiments. However, even databases offering cost estimates for query plans might not be suited for automated cost-based partitioning design since they do not provide a what-if mode for partitioning, i.e. the partitioning has to be actually deployed to obtain an estimate. This was the motivation for us to develop a simple network-centric cost model for partitionings to be used in the offline phase.

Another approach [144] optimizes both analytical and transactional workloads by partially allocating already partitioned tables in an optimal manner to minimize runtime or maximize throughput. Different from this approach, which is only focusing on the allocation, in this paper we provide a new solution to find a partitioning scheme which is an orthogonal problem to data allocation. Furthermore, the paper relies on an allocation heuristic which cannot take the actual execution cost into account. Marcus et al. [111] fragment tables and decide on replication and placement based on the how often they are queried. This strategy results in a custom partitioning scheme that is not supported by many databases and hence inapplicable for an automatic cloud partitioning advisor like ours. Moreover, it does not minimize network costs by leveraging local joins.

Recently, many approaches suggest to use machine learning to automate database administration and tuning [83, 138] and improve internal database components like join ordering [85] or cardinality estimation [79]. In particular, DRL [115, 156] was often used to tackle data management problems. For example Li et al. [97] focus on the scheduling problem for distributed stream data processing systems, Durand et al. [38] optimize the physical table layout or Zhang et al. [192] automate database configuration tuning. Different from those papers, we focus on data partitioning in distributed databases which was not yet considered.

7.9 CONCLUSION AND FUTURE WORK

In this paper, we introduced a new approach for learning a cloud partitioning advisor based on DRL. The main idea is that a DRL agent

learns its decisions based on experience by monitoring the rewards for different workloads and partitioning schemes. The agent is first bootstrapped using a simple network-centric cost model to make the training phase more efficient and afterwards refined with actual runtimes. In the evaluation, we showed that our approach is not only able to find partitionings that outperform existing approaches for automated partitioning design but that it can also adjust to different workloads and new queries. In the future, we plan to combine our approach with systems that predict future workloads to pro-actively re-partition the database as well as to decide whether the costs for repartitioning pay off in the long run.

7.10 ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their helpful feedback. This research is supported by gifts from Huawei, Oracle, and SAP as well as the Collaborative Research Center 1053 (MAKI) of the German Research Foundation (DFG).

8

DBMS FITTING: WHY SHOULD WE LEARN WHAT WE ALREADY KNOW?

ABSTRACT

Deep Neural Networks (DNNs) have successfully been used to replace classical DBMS components such as indexes or query optimizers with learned counterparts. However, commercial vendors are still hesitating to put DNNs into their DBMS stack since these models not only lack explainability but also have other significant downsides such as the requirement for high amounts of training data resulting from the need to learn all behavior from data.

In this paper, we propose an alternative approach to learn DBMS components. Instead of relying on DNNs, we propose to leverage the idea of differentiable programming to fit DBMS components instead of learning their behavior from scratch. Differentiable programming is a recent shift in machine learning away from the direction taken by DNNs towards simpler models that take advantage of the problem structure. In a case study we analyze and discuss how to fit a model to estimate the cost of a query plan and present initial experimental results that show the potential of our approach.

BIBLIOGRAPHIC INFORMATION

The content of this chapter was previously published in the peer-reviewed work 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: *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020. URL: http://cidrdb.org/cidr2020/papers/p34-hilprecht-cidr20.pdf. The contributions of the author of this dissertation are summarized in Section 3.2.

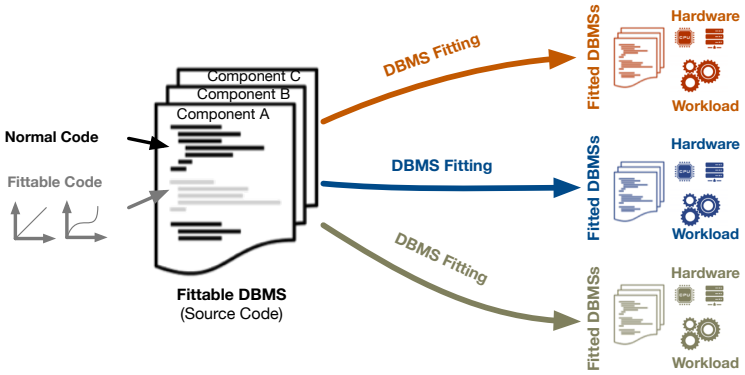


Figure 8.1: A FITable DBMS - The idea is that the code base of a DBMS consists of fittable code that allows a DBMS to adjust its behavior to hardware and workload characteristics.

8.1 INTRODUCTION

MOTIVATION Deep Neural Networks (DNNs) have not only shown to solve many complex problems such as image classification or machine translation, but are applied in many other domains, too. This is also the case for DBMSs, where DNNs have been successfully used not only for automatic database tuning [6, 192], but also to replace existing components with learned counterparts such as learned cost models [79, 161] as well as learned query optimizers [110, 112], learned indexes [48, 84], and learned scheduling or query processing schemes [105, 153].

The power of using DNNs results from the fact that DNNs represent heavily parameterized models that can approximate arbitrary functions. However, the black-box nature makes *DNNs hard to explain*; i.e., decisions of DNNs cannot really be inspected to understand how the learned algorithm is accomplishing its goals. For example, in the case of a learned cost model such as [161] that predicts the execution costs for a given query plan using black-box DNNs, a database administrator would not be able understand why the model produced a certain cost estimate. This is very different from classical cost models that estimate the costs of a plan by combining different factors such as cost of data accesses as well as processing costs. While these models are explainable and allow a database administrator to understand the decisions of the model they are hard to tune and often provide inaccurate estimates [91].

Moreover, explainability is not the only reason why commercial vendors are hesitating to put DNNs into their DBMS stack[34]:

- First, *DNNs are data-hungry* since they have to learn even basic system behavior (that might be well known by a DBMS developer) purely from training data. For example, when learning a cost model, large training corpora are required which need significant time and resources to be constructed since each query

in the training corpora needs to be executed and the execution time needs to be collected. Even worse, this is not a one-time effort, since the same procedure needs to be repeated for every new database that needs to be supported by the optimizer or if the current database is not static (i.e., the schema or data is changing).

- Second, it has been shown that *DNNs are susceptible to small changes* in the input; i.e., already a small change for one input feature can cause the DNN to produce erroneous predictions with high confidence. This is an effect that has also been shown by adversarial attacks. For DBMS, this problem cannot be ignored since it shows the general lack of DNNs to generalize to unseen input in a stable manner and to provide a robustness for DBMS components.

- Third, *DNNs are expensive to update*. In DBMSs, a learned component might need to be updated if the database or the workload changes. However, updating a learned component often requires collecting new training data and an expensive retraining of the DNN. While this might be acceptable in some cases (e.g., the retraining of a learned cost model might eventually be acceptable since it can be done offline), retraining might be too expensive for other components such as learned indexes that require online updates [84] if the database is dynamically changing as for OLTP workloads.

CONTRIBUTIONS In this paper, we propose a different route for learned DBMSs to tackle the aforementioned issues of black-box based approaches. Instead of relying on DNNs to replace classical DBMS components, we propose to leverage the idea of differentiable programming[1] to implement *FITable DBMSs*. In a nutshell, differentiable programming is a recent shift in machine learning, away from the direction taken by DNNs that increasingly use heavily parameterized models and towards simpler white-box models that take more advantage of the problem structure [176]. Recently, differential programming has been used successfully to fit models in domains such as computer vision [98] to encode knowledge on how basic image processing primitives (e.g., edge detectors) work or to learn a physics engine [9] where differentiable functions encode the basic laws of physics that are then fitted.

The main idea of FITable DBMSs goes in the same direction where DBMS components (or parts) are implemented using differentiable functions as shown in Figure 8.1: Similar to normal code, differentiable functions implement logic inside a DBMS that already encodes the basic behavior of a component, but unlike normal code these functions

1 https://www.facebook.com/yann.lecun/posts/10155003011462143

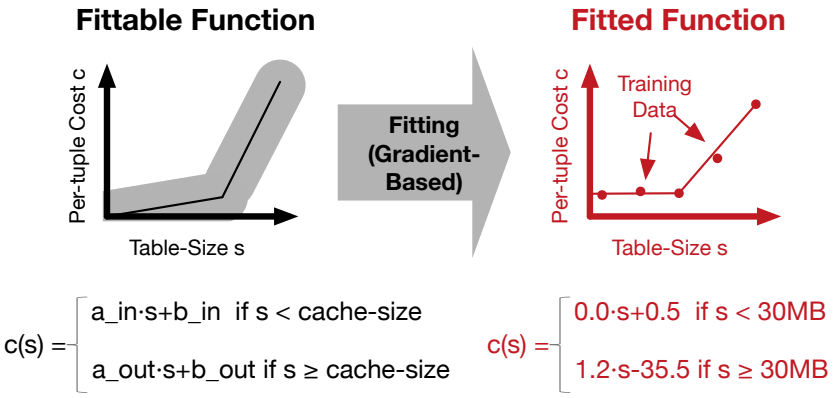


Figure 8.2: Fitting a simple cost model for a scan operator to predict the per-tuple access cost - The example shows how the piecewise linear function can be fitted based on training data by learning the slope and intercept of each segment. For fitting we can use a gradient-based optimization method such as gradient descent.

in addition contain learnable parameters that allow to fit their behavior to a concrete workload and hardware. For example, query optimizers need to be able to estimate the physical cost (i.e., the total execution time) of query plans. Here, fittable functions could be used to describe the basic shape of cost functions for operators that could then be fitted to the behavior of the underlying hardware. While cost functions are a natural candidate for fitting, we believe that other components inside a DBMS such as data structures or execution strategies can benefit from fitting as we will discuss later.

FITable DBMSs are thus different from current approaches for learned DBMS components that purely rely on DNNs [83], since the behavior of a fittable component does not need to be learned from scratch. As a result, fittable functions not only need much less training data compared to training a DNN (which captures the same behavior) but also provide other benefits regarding the explainability and generalization as we show later in this paper. Another direction that is related to FITable DBMSs is the idea of synthesizing data structures [73] from known building blocks such as lists, dense arrays, zone maps, etc. to optimally support a given workload and hardware. Similar to fitting DBMS components, synthesizing data structures also generates explainable DBMS code. However, a major difference to fitting is that while synthesizing only targets the design of data structures, fitting is applicable to a broader set of DBMS components.

OUTLINE The remainder of the paper is organized as follows. In Section 8.2, we present our vision towards so called FITable DBMSs. Afterwards, in Section 8.3 we show by a concrete use case how the idea of fittable DBMS components could be used for cost estimation

and present initial experimental results that show the benefits of our approach. Finally, we conclude in Section 8.4.

8.2 VISION: A FITABLE DBMS

8.2.1 *Basic Idea of Fitting*

The vision of a FITable DBMSs is that *DBMS components (or parts of them) are implemented as differentiable functions* that allow us to adapt the behavior of the component to optimally support a concrete workload and hardware. For instance, a simplified cost model to estimate the execution time of a scan operator in a main-memory DBMS can be modelled as a differentiable function `cost_scan_op` as shown in Listing 8.1. The main idea of this function is that the costs for reading a tuple depend on the table size which can be represented by a piece-wise linear function using two segments for tables that fit into the cache and for those which spill out of the cache.

Listing 8.1: Fittable Function for Simple Cost Model

```
# table-size = size in Byte / no-tuples = number of tuple in table
def cost_scan_op(params, table_size, no_tuples):
        # piecewise linear model
        if table_size< params['cache-size']:
                slope = params['a_in']
                intercept = params['b_in']
                cost_per_tuple = slope * table_size + intercept
        else:
                slope = params['a_out']
                intercept = params['b_out']
                cost_per_tuple = slope * table_size + intercept

        return no_tuples * cost_per_tuple
```

The main benefit of fittable code is that it not only leverages the domain knowledge of the developer (e.g., that the tuple-access cost can be modelled as a piece-wise linear function in our example) but more importantly that the concrete behavior can be fitted automatically to the actual behavior.

The fittable part of the code is captured by parameters that can be learned from concrete behavior. In our example, the learnable parameters are the slope (i.e., `params['a_in']` and `params['a_out']`) and intercept (i.e., `params['b_in']` and `params['b_out']`) of both segments. For fitting the cost model, the actual costs of running the scan operator on different table sizes need to be collected. Since functions are differentiable, normal gradient-based optimization can be used to fit the parameters (i.e., minimize the error of the cost function) as shown in Figure 8.2. Once the parameters are fitted they can be used at runtime of a DBMS, just like fully specified source code.

The power of differentiable programming stems from the fact that the database developer does not have to come up with the gradients

herself. Instead, frameworks such as Autograd[2] support automatic differentiation [176] of ordinary code, which may contain all the usual control structures, including loops, if statements, recursion, and closures. In our example, the code for the cost function in Listing 8.1 is implemented using a normal if-else control flow that can be differentiated automatically.

Overall, fittable code in contrast to black-box DNN models thus provides many advantages: First, fittable code is more *data-efficient*, i.e. we require much less training data since the differentiable function already defines the basic shape of a function that needs to be learned. Furthermore, fitting a differentiable function does not always need to rely on gradient-based methods that typically require multiple passes over the training data. Instead, it can often be implemented by computationally much simpler approaches that only require a single pass [48]. Second, fittable functions typically *generalize* better and are less susceptible to small changes in the input, since they already define a reasonable behavior based on their shape. Finally, fitted code is *explainable and debuggable*. If the behavior is unexpected, the developer can debug the DBMS code (as usual) since the general code structure reflecting the domain knowledge is still interpretable and remains unchanged.

8.2.2 *The Bigger Picture*

There are many different directions the research community can investigate how fitting can be used to adapt DBMS components to a given hardware and workload. In the following, we discuss several DBMS components that are candidates for fitting but also propose directions regarding the general learning setup.

ANALYTICAL MODELS In general, ideal candidates for fitting are DBMS components where (parametrizable) analytical models already exist. For example, transaction scheduling relies on models for conflict probability. While recent papers aim to learn the conflict probability using end-to-end ML models [153] there already exist analytical models [13] that define the conflict probability based on number of concurrent transactions, the database size, etc. However, these models typically rely on parameters that reflect the latency of lock requests as well as reads/writes. Fitting could be used to learn these parameters from the actual behavior of running these operations on a concrete hardware. Other ideas for which analytical models exist that can be fitted are software-prefetching or caching strategies that all rely on similar parameters to predict access costs.

2 https://github.com/HIPS/autogradr

DATA STRUCTURES AND ALGORITHMS While all the before-mentioned applications target the fitting of functions that model different notions of analytical models used in DBMSs for query optimization and scheduling (e.g., execution cost operations, conflict probability), we believe that fitting can be used also for other data structures and algorithms of a DBMS. For example, indexes such as B-trees can be seen as a function that predict the position of a key in a sorted array. While existing papers [84] use black-box DNNs to approximate this mapping function, in [48] we already showed that the idea of fittable white-box functions can be used to learn the mapping. Similar ideas for fitting that learn the data distribution can be used for learning algorithms of database operators such as sorting which is again in contrast to [83] which proposes to use black-box DNNs also for learning algorithms.

END-TO-END LEARNING Another interesting route that needs to be explored is how more complex models can be fitted end-to-end when using white-box fittable functions. End-to-end learning is typically seen as a major advantage of black-box DNNs which can combine several layers (e.g., fully-connected vs. convolutional) to capture complex behaviors. Differentiable programming makes the composition of complex models and end-to-end training also applicable for white-box models. The main idea is that similar to DNNS, white-box functions can also be combined into more complex models and auto-differentiation can then be applied to fit the composed models directly. For example, we will show later in this paper how a cost model to estimate the query execution cost of a complete plan is composed of cost models for individual operators that can be fitted end-to-end based on the monitored execution time of complete query plans.

GREY-BOX LEARNING Finally, while fittable (i.e. white-box) functions provide many advantages over black-box DNNs, we still think that there is a need to combine both. The combination enables a DBMS to learn parts of components where the behavior can not easily be modeled as a fittable function or where the behavior is not known in advance. For example, it is hard to define fittable functions for cost models of operations that are allowed to call user-defined functions, since the complexity of the user-defined code can vary significantly. In this case, a normal DNN can be used to estimate the cost of the user-defined operation and still be combined with the fitted parts of the optimizer. Since fittable functions as well as DNNs are both differentiable, the composed model is still differentiable (due to the chain rule) and can be trained end-to-end.

8.3 CASE STUDY: A FITTABLE COST MODEL

In this section, we discuss the potentials of fitting by presenting a case study with a fittable cost model. In the following, we first dis-

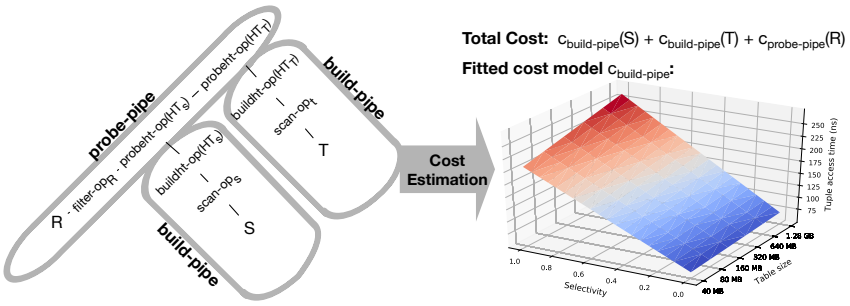


Figure 8.3: Basic idea of our fittable cost model - The total cost of a query plan is estimated based on fitted cost models for each pipeline type. In this example, the build-pipeline type is used in two instantiations over tables S and T and the probe-pipeline type is used in one instantiation over table R, which probes into the hash tables HT_S and HT_T, created by the other two pipelines. The cost models for each pipeline type are based on general features of a pipeline, such as the size of the input table, tuple-width, selectivity of operators etc.

cuss pitfalls of today's approaches for cost models, before we discuss how fitting can be applied to cost models to mitigate these issues. Afterwards, we show initial experimental results of our fitted cost model.

8.3.1 *The Need for better Cost Models*

Models that predict the execution cost of SQL queries are essential components of DBMSs. Query optimizers are the most well-known component that rely on cost models to choose between different alternative query plans based on cost estimations. However, this is not the only component in a DBMS that relies on cost models. More recently, papers have suggested to use cost models for self-driving databases [104] that automate physical design choices.

Traditionally, cost models are handcrafted in a DBMS and thus rely on detailed knowledge about the complexity of the underlying algorithm and data structures. However, these models are typically non-trivial to tune and often provide inaccurate estimates even when using automatic calibration tools [91]. And this is not the only obstacle of existing cost models. Other issues are that these models are also hard to extend since a new model needs to be handcrafted for every new operator implementation. Moreover, today's models do not cover complex operations that allow users to call user-defined functions.

Recent approaches thus suggest to learn cost models by using DNNs instead of handcrafting them [161]. While these approaches can estimate the execution costs more accurately even for complex operations, they suffer from the general problems of using DNNs

not only regarding high training cost but also explainability and robustness of DNNs plus missing update capabilities, as discussed before.

8.3.2 *Fitting a Cost Model*

In the following, we present a fittable cost model that combines (1) the ability of differentiable programming to encode knowledge about the general shape of cost functions for individual operators with (2) the capabilities to capture important effects of the underlying hardware by learning important parameters of the model by fitting. Figure 8.3 shows the basic idea of our fittable cost model.

The model is targeted towards DBMSs that execute SQL queries in a pipelined manner, which is the case for most commercial DBMSs that either implement a classical iterator model (for individual tuples or blocks of tuples) or DBMSs that rely on pipeline-based code generation for query execution, such as Hyper. In order to estimate the execution time of complete query plans, the model estimates the costs of each pipeline and then aggregates the cost to compute the total cost of that query plan. The core components of our model are thus fitted cost models that we use to estimate the costs of individual pipelines.

An important aspect is that a fitted cost model can be used to estimate the execution costs for a wide variety of different instantiations of the same type of pipeline; i.e., we learn the general behavior of a pipeline type that can be applied to different tables, rather than learning a cost model for each particular instantiation of a pipeline over a given table. For example, the cost model shown in Figure 8.3 (right-hand-side) can be used to estimate the costs for both build-pipelines that are executed over two different tables S and T by providing the features of the pipeline as input to the model. In order to enable that the same cost model can be used for different instantiations of the same pipeline type, our cost models take general features of a pipeline (such as the base table size, tuple-width, etc.) as input to estimate the execution time.

The currently supported pipeline types in our cost model are shown in Table 8.1 (as the first three rows). The cost model for each of these pipeline types is composed of one or multiple differentiable functions that capture the cost for each operator used in that pipeline. For example, the cost model $c_{build-pipe}$ is composed of two differentiable functions: one function $c_{scan-op}$ that captures the cost for a filter operator and one function $c_{buildht-op}$ that captures the cost of building a hash table. The fittable cost models for the operators that we use for the different pipeline types are shown in Table 8.1 (as the last four rows).

Another aspect of our fittable cost model is that the cost models for pipeline types, such as $c_{build-pipe}$, define weights (e.g., w_f and w_b) that

Name	Type	Cost function	Learned Parameters and Comments
scan-pipe	pipeline	$c_{scan-pipe} = w_f \cdot c_{scan-op}(T) + w_m \cdot c_{mat-op}(apply_{pipe}(T))$	w_f and w_m.
build-pipe	pipeline	$c_{build-pipe} = w_f \cdot c_{scan-op}(T) + w_b \cdot c_{buildht-op}(apply_{pipe}(T)$	w_f and w_b.
probe-pipe	pipeline	$c_{probe-pipe} = w_f \cdot c_{scan-op}(T) + w_p \cdot \sum_{i=1}^{n} c_{probeht-op}(HT^i, apply_{pipe}(T) + w_m \cdot c_{mat-op}(apply_{pipe}(T))$	w_f, w_p and w_m. The pipeline can probe into multiple hash-tables denoted as HT^i for the i-th join to implement multi-way joins.
scan-op	operator	$c_{scan-op}(T) = \begin{cases} rows(T) \cdot (a_1 \cdot \lvert T\rvert + b_1) & \lvert T\rvert < L1\text{-}cache \\ rows(T) \cdot (a_2 \cdot \lvert T\rvert + b_2) & L1\text{-}cache \leq \lvert T\rvert < L2\text{-}cache \\ rows(T) \cdot (a_3 \cdot \lvert T\rvert + b_3) & L2\text{-}cache \leq \lvert T\rvert < L3\text{-}cache \\ rows(T) \cdot (a_4 \cdot \lvert T\rvert + b_4) & L3\text{-}cache \leq \lvert T\rvert \end{cases}$	a_i and b_i where a_i and b_i are linear combinations of tuple-width and number of attributes in the filter predicate each having its own fittable parameter. Moreover, we not only use different parameters a_i and b_i (i.e., segments) for different table sizes but also for selectivities < 0.5 and ≥ 0.5 as well as for number of attributes in selection predicates to model effects such as branch-mispredictions and effects of cache-line sizes. However, showing the parameters for all cases in this table would decrease the readability and thus we omit them.
buildht-op	operator	$c_{buildht-op}(T) = w_b \cdot tw \cdot rows(T)$	w_b. Cost for inserting a tuple linearly depending on tuple-width (tw).
probeht-op	operator	$c_{probeht-op}(HT, T) = \begin{cases} w_{p1} \cdot tw \cdot rows(T) & ht - size < L1\text{-}cache \\ w_{p2} \cdot tw \cdot rows(T) & L1\text{-}cache \leq \lvert HT\rvert < L2\text{-}cache \\ w_{p3} \cdot tw \cdot rows(T) & L2\text{-}cache \leq \lvert HT\rvert < L3\text{-}cache \\ w_{p4} \cdot tw \cdot rows(T) & L3\text{-}cache \leq \lvert HT\rvert \end{cases}$	w_{p1}, w_{p2}, and w_{p3}. We use different parameters to reflect the different cost depending on the fact whether the HT fits into one level of the caches. Moreover, the cost of probing a single tuple into a HT linearly depends on the tuple-width (tw) of the probed tuple.
mat-op	operator	$c_{mat-op}(T) = w_m \cdot \lvert T\rvert$	w_m. Cost for materializing a single tuple are constant.

Table 8.1: Fittable cost models for pipeline types and operators - T is the input table of a pipeline, $|T|$ is the size of the input table in Byte and rows(T) the number of rows in T, $apply_{pipe}(T)$ is the resulting table T after applying all downstream operators on T, HT is a hash-table that is either build or probed and $|HT|$ is the size of the hash-table in Byte.

reflect the influence of a particular operator on the overall cost, when executed in that pipeline. These parameters are fitted individually for each pipeline type, since the same operator (when used in different pipeline types) can have a different influence on the overall cost. For example, the cost of materializing the output might be more dominant in a scan-pipeline than in a probe-pipeline, where the overall cost is dominated by random memory accesses resulting from probing into the hash table(s).

Finally, for the actual fitting of the cost models of the different pipeline types, we collect the actual runtime for a variety of pipeline instances for a given hardware platform. We use this collected training data for gradient-based optimization to fit the cost model and learn the parameters of pipelines end-to-end, as indicated in Table 8.1. The pipeline types we currently support already allow us to estimate the execution time for a wide variety of query plans ranging from simple query plans over a single table to complex query plans with multi-way hash joins over multiple tables consisting of multiple build- and probe-pipelines. In the future, we plan to extend the pipeline types to cover also other operations such as aggregations.

8.3.3 *Initial Results*

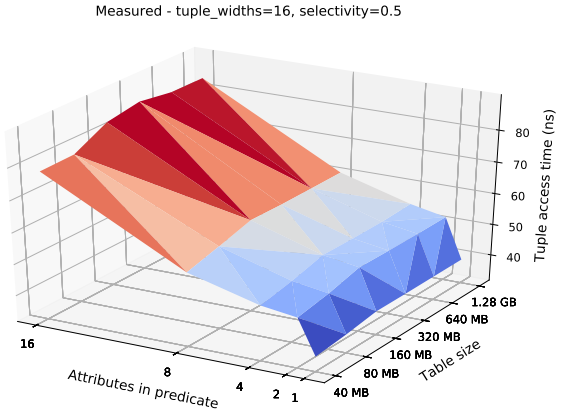
In the following, we show the initial results of fitting our cost model and compare the results also to recent learned cost models that purely rely on DNNs [161]. The aim of our experiments is to show that (1) white-box model can provide high accuracy for cost estimates, (2) white-box models need less training data than black-box models and (3) white-box models can generalize.

For the experiments, we implemented our fittable cost model based on the Autograd[3] framework in Python and a prototypical main-memory based execution engine in C++ to run SQL queries to collect training data. The code of our implementation is available open-source[4].

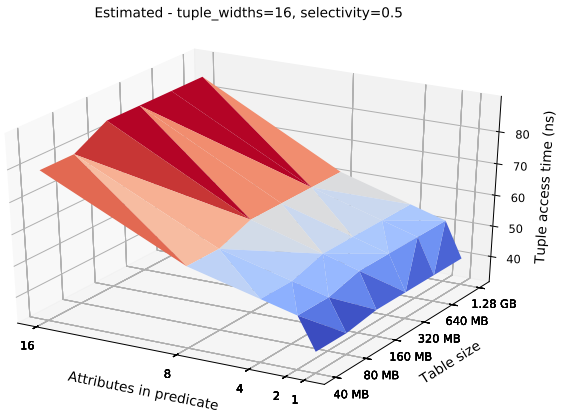
For running all experiments, we used a server with two Intel Gold 5120 Skylake CPUs (2.2 GHz, 19.25 MiB L3 cache) and 384GB of DDR4 RAM. For collecting training data, all SQL queries were executed single-threaded inside our execution engine. We make use of the Adam optimizer inside the Autograd framework for fitting our cost model.

EXP. 1 - ACCURACY OF MODEL In this experiment, we report the accuracy of our fittable cost model to show their potential to provide high quality estimates. To measure the quality of cost estimates in this experiment, we use the q-error, which is the factor by which an

3 https://github.com/HIPS/autograd
4 https://github.com/DataManagementLab/cidr-cost-model/



(a) Real execution time



(b) Estimated execution time

Figure 8.4: Exp. 1 - Real and estimated execution time for the scan-pipeline type for table sizes larger than L3 cache. The plots show the real and estimated execution time for the different tables sizes and number of attributes used in the selection predicate. We see that for one attribute in the selection predicate the tuple-access time is much lower since the attribute to be evaluated fits into one L1 cache line. Our cost model for the scan-pipeline captures this by using different segments in a piecewise-linear function for the filter-op as discussed in Table 8.1 (last column).

Name	Median q-error	90th-percentile
scan-pipe	1.0148	1.0287
build-pipe	1.0355	1.0663
probe-pipe	1.0403	1.0735

Table 8.2: Q-error of different pipelines when trained on 100% of the training data on all table sizes. We see that the median q-error for the different pipelines is maximum 1.0403.

estimate differs from the real execution time. For example, if the real execution time of a pipeline is 100ms, the estimates of 10ms or 1000ms both have a q-error of 10. Using the ratio instead of an absolute or quadratic error captures the intuition that for making optimization decisions only relative differences matter.

For collecting training and testing data, we created tables of different sizes (from 32 kB to 1.28 GB) with a varying tuple-width from 1 attribute (4 Byte) up to 16 attributes (64 Byte). For these tables we then executed query plans (single table and join queries) composed of the pipeline types supported by our cost model. In total, we thus collected the execution time for 56,730 pipeline instances, evenly spread across the different pipeline types. Afterwards, we randomly split the data into 90% for training and 10% for testing.

The q-error (median and 90th percentile) for all table sizes are shown in Table 8.2. We can see that our fittable cost models can provide accurate estimates for the different pipeline types with a median q-error of less than 1.0403. While in this experiment, we used the full training data, in the next experiment (Exp. 2) we see that already 5% of the training data is enough to achieve a similarly low q-error for our model. Additionally, compared to a black-box DNN, which we also show in the next experiment, the q-error of our fitted cost model is lower and requires less training data.

We also visualize the results of the estimated costs of our model and the real execution times in Figure 8.4 to see that our model precisely captures the tuple access cost.

EXP. 2 - DATA EFFICIENCY OF MODEL In this experiment, we show the data efficiency of our fittable cost model. For this experiment, we use the same testing set as before but vary the size of the training data used for fitting our model. Moreover, in order to show that our model is more data efficient than a black-box model for cost estimation, we implemented the approach suggested in [161] that uses a tree-based DNN to estimate the cost of a query plan. A tree-based DNN uses a separate DNN for each operator in a query plan that can be stacked together and trained end-to-end. Our implementation of their approach based on the Autograd framework is also available in our open-source repository.

The results for learning the cost model for simple query plans on a single table are shown in Figure 8.5. We can see that our white-box

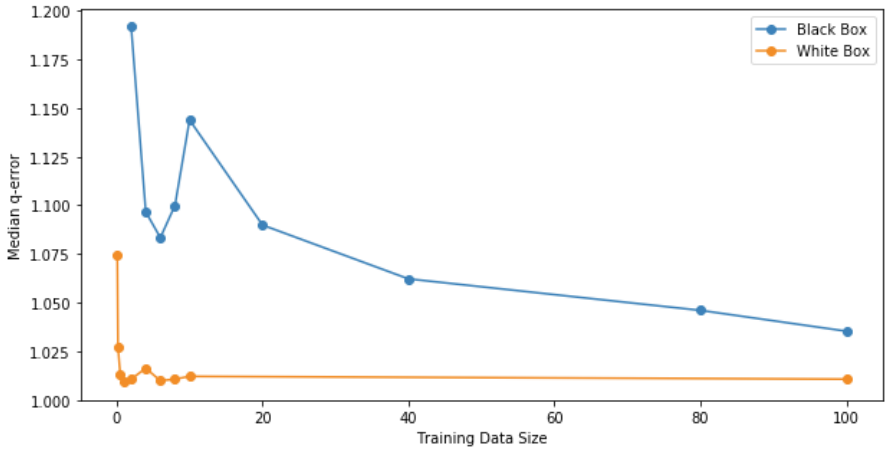


Figure 8.5: Exp. 2 - Data efficiency of our fittable cost model. This plot shows the result for the scan-pipeline comparing the median q-error of our model based (white-box) to a DNN-based model (black-box) based on [161], when using only x% of the original training data.

model can already achieve a low q-error with only 5% of the training data. In contrast, the black-box model requires much more training data to achieve a low q-error even for these simple queries. More interestingly, if we provide the full training data to the black-box model, it is not able to reach the same accuracy that our white-box model achieves with only 5% of the training data.

We also executed the same experiment for more complex query plans that include joins over two tables. The results (which we do not plot due to space restrictions in this paper) show a similar trend as for the scan-pipeline only.

EXP. 3 - GENERALIZABILITY OF MODEL Finally, in the last experiment we show the capability of our cost model to generalize queries over new tables. In order to show that our fitted cost model can generalize to new unseen tables, we excluded tables sizes larger than 320MB from the training data. For testing, we used tables of sizes that the model had not seen before including table sizes that are in the range of those the model had seen before (e.g., 256MB) as well as table sizes larger than the model had seen (e.g. 512 and 1024MB). The results for estimating the cost of the scan-pipeline is depicted in Figure 8.6 showing the real execution time as well as the estimated execution time for these unseen tables.

As we can see, the model generalizes to these new tables. The median q-error and 90*th* percentile are similar to the results of Exp. 2. We do not show results of the black-box model that we used in Exp. 2 since this model cannot generalize to new unseen table sizes. The reason is that tables in this model are encoded using one-hot vectors; i.e., the model learns the cost estimation individually for a particular

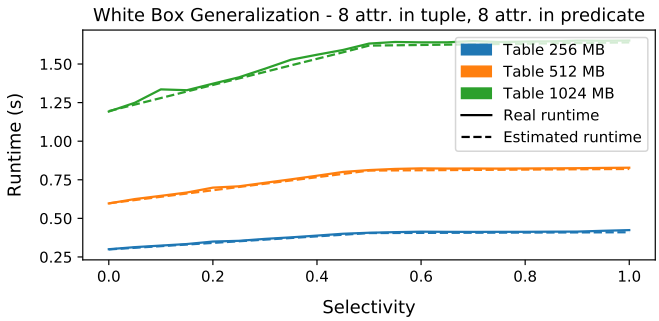


Figure 8.6: Exp. 3 - Generalizability of our fittable model to unseen tables. Results show the real and estimated execution time for table sizes of 256 MB, 512 MB, and 1 GB that have not been used in the training set.

table rather than learning a cost model that is based on general features such as table-sizes as we do.

8.4 CONCLUSION

In this paper, we have presented our vision towards FITable DBMSs. Based on our initial case study with a fitted cost model, we have shown that fitting not only needs much less training data but also generalizes, since the model itself captures the general shape of how the cost of operators in a DBMS typically behave.

While cost modeling is a natural candidate for fitting, we believe that fitting can be used for many other DBMS components. Furthermore, since differential programming enables end-to-end learning by composing white-box and black box models, we believe that this allows us to build holistic models that span across different DBMS components; e.g., to combine a fittable model for caching with a fittable cost model for query optimization to enable better decisions in a DBMS system.

9

DEEPDB: LEARN FROM DATA, NOT FROM QUERIES!

ABSTRACT

The typical approach for learned DBMS components is to capture the behavior by running a representative set of queries and use the observations to train a machine learning model. This workload-driven approach, however, has two major downsides. First, collecting the training data can be very expensive, since all queries need to be executed on potentially large databases. Second, training data has to be recollected when the workload or the database changes. To overcome these limitations, we take a different route and propose a new data-driven approach for learned DBMS components which directly supports changes of the workload and data without the need of retraining. Indeed, one may now expect that this comes at a price of lower accuracy since workload-driven approaches can make use of more information. However, this is not the case. The results of our empirical evaluation demonstrate that our data-driven approach not only provides better accuracy than state-of-the-art learned components but also generalizes better to unseen queries.

BIBLIOGRAPHIC INFORMATION

The content of this chapter was previously published in the peer-reviewed work Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. "DeepDB: Learn from Data, not from Queries!" In: *Proc. VLDB Endow.* 13.7 (2020), pp. 992–1005. DOI: 10.14778/3384345.3384349. URL: http://www.vldb.org/pvldb/vol13/p992-hilprecht.pdf. The contributions of the author of this dissertation are summarized in Section 4.1.

9.1 INTRODUCTION

MOTIVATION Deep Neural Networks (DNNs) have not only been shown to solve many complex problems such as image classification or machine translation, but are applied in many other domains, too. This is also the case for DBMSs, where DNNs have successfully been used to replace existing DBMS components with learned counterparts such as learned cost models [79, 161] as well as learned query optimizers [112], or even learned indexes [84] or query scheduling and query processing schemes [105, 153].

The predominant approach for learned DBMS components is that they capture the behavior of a component by running a representative set of queries over a given database and use the observations to train the model. For example, for learned cost models such as [79, 161] different query plans need to be executed to collect the training data, which captures the runtime (or cardinalities), to then learn a model that can estimate costs for new query plans. This observation also holds for the other approaches such as learned query optimizers or the learned query processing schemes, which are also based on collected training data that requires the execution of a representative workload.

A major obstacle of this workload-driven approach is that collecting the training data is typically very expensive since many queries need to be executed to gather enough training data. For example, approaches like [79, 161] have shown that the runtime of hundreds of thousands of query plans is needed for the model to provide a high accuracy. Still, the training corpora often only cover a limited set of query patterns to avoid even higher training costs. For example, in [79] the training data covers only queries up to two joins (three tables) and filter predicates with a limited number of attributes.

Moreover, the training data collection is not a one-time effort since the same procedure needs to be repeated over and over if the workload changes or if the current database is not static and the data is constantly being updated as it is typical for OLTP. Otherwise, without collecting new training data and retraining the models for the characteristics of the changing workload or data, the accuracies of these models degrade with time.

CONTRIBUTIONS In this paper, we take a different route. Instead of learning a model over the workload, we propose to learn a purely data-driven model that captures the joint probability distribution of the data and reflects important characteristics such as correlations across attributes but also the data distribution of single attributes. Another important difference to existing approaches is that our data-driven approach supports direct updates; i.e., inserts, updates, and deletes on the underlying database can be absorbed by the model without the need to retrain the model.

As a result, since our model captures information of the data it can not only be used for one particular task but supports many different tasks ranging from query answering, over cardinality estimation to potential other more sophisticated tasks such as in-DBMS machine learning inference. One could now think that this all comes at a price and that the accuracy of our approach must be lower since the workload-driven approaches get more information than a pure data-driven approach. However, as we demonstrate in our experiments, this is not the case. Our approach actually outperforms many state-of-the-art workload-driven approaches and even generalizes better.

However, we do not argue that data-driven models are a silver bullet to solve all possible tasks in a DBMS. Instead, we think that data-driven models should be combined with workload-driven models when it makes sense. For example, a workload-driven model for a learned query optimizer might use the cardinally estimates of our model as input features. This combination of data-driven and workload-driven models provides an interesting avenue for future work but is beyond the scope of this paper.

To summarize, the main contributions of this paper are:(1) We developed a new class of deep probabilistic models over databases: Relational Sum Product Networks (RSPNs), that can capture important characteristics of a database. (2) To support different tasks, we devise a probabilistic query compilation approach that translates incoming database queries into probabilities and expectations for RSPNs. (3) We implemented our data-driven approach in a prototypical DBMS architecture, called DeepDB, and evaluated it against state-of-the-art learned and non-learned approaches.

OUTLINE The remainder of the paper is organized as follows. In Section 9.2 we first present an overview of DeepDB and then discuss details of our models and the query compilation in Sections 9.3 and 9.4. Afterwards, we explain further extensions of DeepDB in Section 9.5 before we show an extensive evaluation comparing DeepDB against state-of-the art approaches for various tasks. Finally, we iterate over related work in Section 9.7 before concluding in Section 9.8.

9.2 OVERVIEW AND APPLICATIONS

OVERVIEW As shown in Figure 9.1, the main idea of DeepDB is to learn a representation of the data offline. An important aspect of DeepDB is that we do not aim to replace the original data with a model. Instead, a model in DeepDB augments a database similar to indexes to speed-up queries and to provide additional query capabilities while we can still run standard SQL queries over the original database.

To optimally capture relevant characteristics of relational data in DeepDB, we developed a new class of models called *Relational Sum Product Networks* (RSPNs). In a nutshell, RSPNs are a class of deep

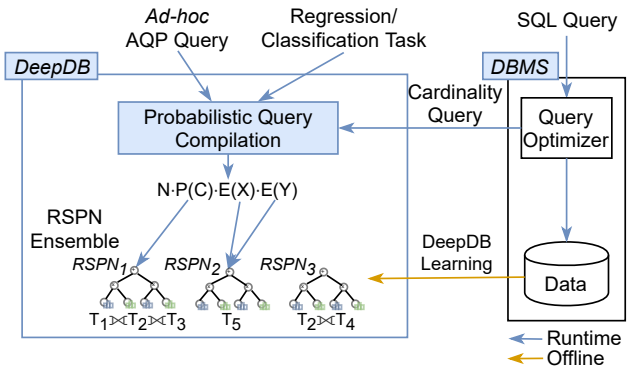


Figure 9.1: Overview of DeepDB.

probabilistic models that capture the joint probability distribution over all attributes in a database that can then be used at runtime to provide the answer for different user tasks.

While RSPNs are based on Sum Product Networks (SPNs) [117, 141], there are significant differences: (1) While SPNs support only single tables and simple queries (i.e., no joins and no aggregation functions), RSPNs can be built on arbitrary schemata and support complex queries with multi-way joins and different aggregations (COUNT, SUM, AVG). Moreover, RSPNs also go beyond the idea of other recent learned data models that need to know join paths a priori such as [105, 186] since RSPNs allow true ad-hoc joins by combining RSPN models. (2) Another major difference is that RSPNs support direct updates, i.e., if the underlying database changes the RSPN can directly ingest the updates without the need to retrain the model. (3) RSPNs also include a set of database-specific extensions such as NULL-value handling and support for functional dependencies.

Once the RSPNs are created offline, they can be leveraged at runtime for a wide variety of different applications, ranging from user-facing tasks (e.g., to provide fast approximate answers for SQL queries) to system-internal tasks (e.g., to provide estimates for cardinalities). In order to support these tasks, DeepDB provides a new so called *probabilistic query compilation* procedure that translates a given task into evaluations of expectations and probabilities on RSPNs. We now give a brief overview of the applications currently supported by the query compilation engine of DeepDB.

CARDINALITY ESTIMATION The first task DeepDB supports is cardinality estimation for a query optimizer. Cardinality estimation is needed to provide cost estimates but also to find the correct join order during query optimization. A particular advantage of DeepDB over existing learned approaches for cardinality estimation [79, 161] is that we do not have to create dedicated training data, i.e. pairs of queries and cardinalities. Instead, since RSPNs capture the characteristics of the data independent of a workload, we can support arbitrary join queries without the need to train a model for a particular workload.

Moreover, RSPNs can be kept up to date at low costs similar to traditional histogram-based approaches, which is different from other workload-driven learned approaches for cardinality estimation such as [79, 161] which require retraining.

APPROXIMATE QUERY PROCESSING (AQP) The second task we currently support in DeepDB is AQP. AQP aims to provide approximate answers to support faster query response times on large datasets. The basic idea of how a query on a single table is executed inside DeepDB is simple: for example, an aggregate query $\texttt{AVG}(X)$ with a where condition C is equal to the conditional expectation $\mathbb{E}(X \mid C)$ which can be approximated with RSPNs. In DeepDB, we implement a more general AQP procedure that leverages the fact that RSPNs can support joins of multiple tables. A major difference to other learned approaches for AQP such as [105, 167] is again that DeepDB supports ad-hoc queries and is thus not limited to the query types covered by the training set.

OTHER APPLICATIONS While the applications above show the potential of DeepDB, we believe DeepDB is not limited to those applications. For example, machine learning inference tasks such as regression and classification can be answered by RSPNs. However, discussing these opportunities in detail is beyond the scope of this paper.

9.3 LEARNING A DEEP DATA MODEL

In this section, we introduce Relational Sum Product Networks (RSPNs), which we use to learn a representation of a database and, in turn, to answer queries using our query engine explained in the next section. We first review Sum Product Networks (SPNs) and then introduce RSPNs. Afterwards, we describe how an ensemble of RSPNs can be created to encode a given database multiple tables.

9.3.1 *Sum Product Networks*

Sum-Product Networks (SPNs) [141] learn the joint probability distribution $P(X_1, X_2, \ldots, X_n)$ of the variables X_1, X_2, ..., X_n in the dataset. They are an appealing choice because probabilities for arbitrary conditions can be computed very *efficiently*. We will later make use of these probabilities for our applications like AQP and cardinality estimation.

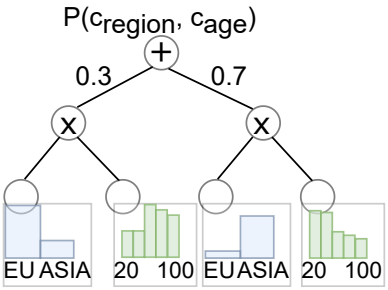
For the sake of simplicity, we restrict our attention to Tree-SPNs, i.e., trees with sum and product nodes as internal nodes and leaves. Intuitively, sum nodes split the population (i.e., the rows of dataset) into clusters and product nodes split independent variables of a population (i.e., the columns of a dataset). Leaf nodes represent a single attribute and approximate in the present paper the distribution of that attribute

c_id	c_age	c_region
1	80	EU
2	70	EU
3	60	ASIA
4	20	EU
...	...	...
998	20	ASIA
998	25	EU
999	30	ASIA
1000	70	ASIA

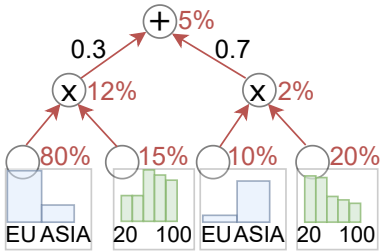
(a) Example Table

c_age	c_region
80	EU
70	EU
60	ASIA
20	EU
...	...
...	...
20	ASIA
25	EU
30	ASIA
70	ASIA

(b) Learning with Row/ Column Clustering



(c) Resulting SPN



(d) Probability of European Customers younger than 30

Figure 9.2: Customer Table and corresponding SPN.

either using histograms for discrete domains or piecewise linear functions for continuous domains [118]. For instance, in Figure 9.2c, an SPN was learned over the variables *region* and *age* of the corresponding *customer* table in Figure 9.2a. The top sum node splits the data into two groups: The left group contains 30% of the population, which is dominated by older European customers (corresponding to the first rows of the table), and the right group contains 70% of the population with younger Asian customers (corresponding to the last rows of the table). In both groups, region and age are independent and thus split by a product node each. The leaf nodes determine the probability distributions of the variables *region* and *age* for every group.

Learning SPNs [50, 118] works by recursively splitting the data in different clusters of rows (introducing a sum node) or clusters of independent columns (introducing a product node). For the clustering of rows, a standard algorithm such as *KMeans* can be used or the data can be split according to a random hyperplane. To make no strong assumptions about the underlying distribution, Randomized Dependency Coefficients (RDC) are used for testing independence of different columns [101]. Moreover, independence between all columns is assumed as soon as the number of rows in a cluster falls below a threshold n_{min}. As stated in [117, 141], SPNs in general have polynomial size and allow inference in linear time w.r.t. the number of nodes. However, for the configurations we use in our experiments, we can even bound the size of the SPNs to linear complexity w.r.t. the number of columns in a dataset since we set $n_{min} = n_s/100$ (i.e. relative to the sample size), which turned out to be a robust configuration.

With an SPN at hand, one can compute probabilities for conditions on arbitrary columns. Intuitively, the conditions are first evaluated on every relevant leaf. Afterwards, the SPN is evaluated bottom up. For instance in Figure 9.2d, to estimate how many customers are from Europe and younger than 30, we compute the probability of European customers in the corresponding blue *region* leaf nodes (80% and 10%) and the probability of a customer being younger than 30 (15% and 20%) in the green *age* leaf nodes. These probabilities are then multiplied at the product node level above, resulting in probabilities of 12% and 2%, respectively. Finally, at the root level (sum node), we have to consider the weights of the clusters, which leads to $12\% \cdot 0.3 + 2\% \cdot 0.7 = 5\%$. Multiplied by the number of rows in the table, we get an approximation of 50 European customers who are younger than 30.

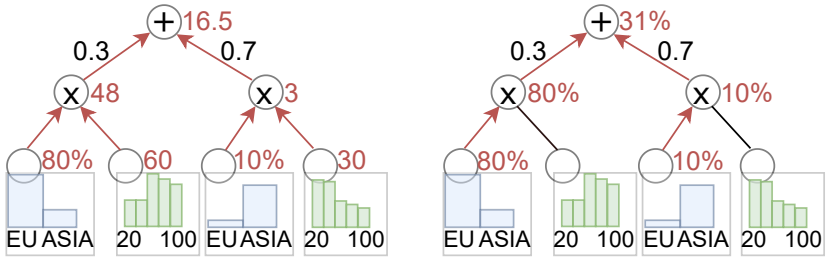
9.3.2 *Relational Sum-Product Networks*

One important issue with SPNs is that they can only capture the data of single tables but they also lack other important features needed for DeepDB. To alleviate these problems, we now introduce RSPNs.

EXTENDED INFERENCE ALGORITHMS The first and most important extension is that for many queries such as AVG and SUM expectations are required (e.g., to answer a SQL aggregate query which computes an average over a column). In order to answer these queries, RSPNs allows computing expectations over the variables on the leaves to answer those aggregates. To additionally apply a filter predicate, we still compute probabilities on the leaves for the filter attribute and propagate both values up in the tree. At product nodes, we multiply the expectations and probabilities coming from child nodes whereas on sum nodes the weighted average is computed. In Figure 9.3, we show an example how the average age of European customers is computed. The ratio of both terms yields the correct conditional expectation. A related problem is that SPNs do not provide confidence intervals. We also developed corresponding extensions on SPNs in Section 9.5.1.

DATABASE-SPECIFICS Finally, SPNs lack support for important database specifics: (1) First, SPNs do not provide mechanisms for handling NULL values. Hence, we developed an extension where NULL values are represented as a dedicated value for both discrete and continuous columns at the leaves during learning. Furthermore, when computing conditional probabilities and expectations, NULL values must be handled according to the three-valued logic of SQL. (2) Second, SPNs aim to generalize the data distribution and thus approximate the leaf distribution, abstracting away specifics of the dataset to generalize. For instance, in the leaf nodes for the age in Figure 9.2c, a piecewise linear function would be used to approximate the distribution [118]. Instead, we want to represent the data as accurate as possible. Hence, for continuous values, we store each individual value and its frequency. If the number of distinct values exceeds a given limit, we also use binning for continuous domains. (3) Third, functional dependencies between non-key attributes $A \rightarrow B$ are not well captured by SPNs. We could simply ignore these and learn the RSPN with both attributes A and B, but this often leads to large SPNs since the data would be split into many small clusters (to achieve independence of A and B). Hence, we allow users to define functional dependencies along with a table schema. If a functional dependency $A \rightarrow B$ is defined, we store the mapping from values of A to values of B in a separate dictionary of the RSPN and omit the column B when learning the RSPN. At runtime, queries with filter predicates for B are translated to queries with filter predicates for A.

UPDATABILITY Finally, a last important extensions of RSPNs over SPNs is the direct updatability of the model. If the underlying database tables are updated, the model might become inaccurate. For instance, if we insert more young European customers in the table in Figure 9.2a, the probability computed in Figure 9.2d is too low and thus the RSPN needs to be updated. As described before, an RSPN consists of product



(a) $\mathbb{E}(\texttt{c\_age} \cdot 1_{\texttt{c\_region='EU'}})$ (b) $P(\texttt{c\_region='EU'})$

Figure 9.3: Process of computing $\mathbb{E}(\texttt{c\_age} \mid \texttt{c\_region='EU'})$.

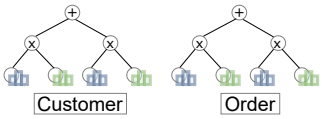
and sum nodes, as well as leaf nodes, which represent probability distributions for individual variables. The key-idea to support direct updates of an existing RSPN is to traverse the RSPN tree top-down and update the value distribution of the weights of the sum-nodes during this traversal. For instance, the weight of a sum node for a subtree of younger European customers could be increased to account for updates. Finally, the distributions in the leaf-nodes are adjusted. The detailed algorithm of how to directly update RSPNs is discussed in Section 9.5.2.

9.3.3 *Learning Ensembles of RSPNs*

In order to support ad-hoc join queries one could naively learn a single RSPN per table as we discuss in Section 9.4. However, in this case potential correlations between tables might be lost and lead to inaccurate approximations. For learning an ensemble of RSPNs for a given database with multiple tables, we thus take into account if tables of a schema are correlated.

In the following, we describe our procedure that constructs a so called *base ensemble* for a given database scheme. In this procedure, for every *foreign key→primary key* relationship we learn an RSPN over the corresponding full outer join of two tables if there is a correlation between attributes of these two tables. Otherwise, RSPNs for the single tables will be learned. For instance, if the schema consists of a `Customer` and an `Order` table as shown in Figure 9.4, we could either learn two independent RSPNs (one for each table) or a joint RSPN (over the full outer join).

In order to test independence of two tables and thus to decide if one or two RSPNs are more appropriate, we check for every pair of attributes from these tables if they can be considered independent or not. In order to enable an efficient computation, this test can be done on a small random sample. As a correlation measure that does not make major distributional assumptions, we compute RDC values [101] between two attributes, which are also used in the SPN learning algorithm [118]. If the maximum pairwise RDC value between all attributes of two tables exceeds a threshold (where we use the standard



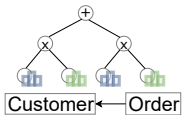
Customer

c_id	c_age	c_region	$\mathcal{F}_{C \leftarrow O}$
1	20	EU	2
2	50	EU	0
3	80	ASIA	2

Order

o_id	c_id	o_channel
1	1	ONLINE
2	1	STORE
3	3	ONLINE
4	3	STORE

(a) Ensemble with Single Tables



Customer ⟗ Order

$\mathcal{N}_C$	c_id	c_age	c_region	$\mathcal{F}'_{C \leftarrow O}$	$\mathcal{N}_O$	o_id	o_channel
1	1	20	EU	2	1	1	ONLINE
1	1	20	EU	2	1	2	STORE
1	2	50	EU	1	0	NULL	NULL
1	3	80	ASIA	2	1	3	ONLINE
1	3	80	ASIA	2	1	4	STORE

(b) Ensemble with Full Outer Join

Figure 9.4: Two RSPN Ensembles for the same Schema. Additional (blue) columns are also learned by the RSPNs.

thresholds of SPNs), we assume that two tables are correlated and learn an RSPN over the join.

In the base ensemble only correlations between two tables are captured. While in our experiments, we see that this already leads to highly accurate answers, there might also be correlations not only between directly neighboring tables. Learning these correlations helps to further improve the accuracy of queries that span more than two tables. For instance, if there was an additional `Product` table that can be joined with the `Orders` table and the product prize is correlated with the customers region, this would not be taken into account in the *base ensemble*. In Section 9.5.3, we thus extend our basic procedure for ensemble creation to take dependencies among multiple tables into account.

9.4 QUERY COMPILATION

The main challenge of probabilistic query compilation is to translate an incoming query into an inference procedure against an ensemble of RSPNs. The class of SQL queries that DeepDB currently supports are of the form:

$\mathbf{Q}_D$: `SELECT AGG`

```
FROM T_1 JOIN T_2 ON ... JOIN T_n ON ...
WHERE T_i.a OP LITERAL AND/OR ...
(GROUP BY ...);
```

where AGG is one of the aggregations COUNT, SUM, or AVG over a numerical attribute, the joins are acyclic equi-joins and the filter in the WHERE clause are either a conjunction of filters or a disjunction. While conjunctions are supported natively by RSPNs, disjunctions are realized using the principle of inclusion and exclusion. In the filters, OP is one of the operators <,>,=, <=,>=,IN. Finally, there is an optional GROUP BY clause on one or several attributes.

Most importantly, in DeepDB the queries are supported ad-hoc, i.e. an RSPN ensemble is learned once and then arbitrary queries of the above form can be answered using our probabilistic query compilation procedure. In the following, we first describe how this procedure works for COUNT queries without grouping which is sufficient for cardinality estimation. We then show the extensions to support a broader set of aggregate queries for AQP including other aggregates (AVG and SUM) as well as grouping.

9.4.1 *Simple COUNT Queries*

In this section, we explain how we can translate COUNT queries with and without filter predicates over single tables or over joins of multiple tables using inner joins (equi-joins). These types of queries can be used already for cardinality estimation but also cover some cases of aggregate queries for AQP. For answering the simple COUNT queries, we distinguish three cases of how queries can be mapped to RSPNs: (1) an RSPN exists that exactly matches the tables of the query, (2) the RSPN is larger and covers more tables, and (3) we need to combine multiple RSPNs since there is no single RSPN that contains all tables of the query.

CASE 1: EXACT MATCHING RSPN AVAILABLE The simplest case is a single table COUNT query with (or without) a filter predicate. If an RSPN is available for this table and N denotes the number of rows in the table, the result is simply $N \cdot P(C)$. For instance, the query

```
Q_1: SELECT COUNT(*) FROM CUSTOMER C
WHERE c_region='EU';
```

can be answered with the CUSTOMER RSPN in Figure 9.4a. The result is $|\texttt{C}| \cdot \mathbb{E}(\mathbf{1}_{\texttt{c\_region='EU'}}) = 3 \cdot \frac{2}{3} = 2$. Note that 1_C denotes the random variable being one if the condition C is fulfilled and thus $\mathbb{E}(\mathbf{1}_C) = P(C)$. While a conjunction in a filter predicates is directly supported, a disjunction could be realized using the inclusion-exclusion principle.

A natural extension for COUNT queries over joins could be to learn an RSPN for the underlying join and use the formula $|J| \cdot P(C)$ where

the size of the joined tables without applying a filter predicate is $|J|$. For instance, the query

```
Q2: SELECT COUNT(*) FROM CUSTOMER C
NATURAL JOIN ORDER O
WHERE c_region='EU' AND
o_channel='ONLINE';
```

could be represented as

$$|\texttt{C} \bowtie \texttt{O}| \cdot P(\texttt{o\_channel='ONLINE'} \cap \texttt{c\_region='EU'})$$

which is $4 \cdot \frac{1}{4} = 1$.

However, joint RSPNs over multiple tables are learned over the full outer join. By using full outer joins we preserve all tuples of the original tables and not only those that have one or more join partner in the corresponding table(s). This way we are able for example to answer also single table queries from a joint RSPN, as we will see in Case 2. The additional NULL tuples that result from a full outer join must be taken into account when answering an inner join query. For instance, the second customer in Figure 9.4b does not have any orders and thus should not be counted for query Q_2. To make it explicit which tuples have no join partner and thus would not be in the result of an inner join, we add an additional column $\mathcal{N}_T$ for every table such as in the ensemble in Figure 9.4b. This column is also learned by the RSPN and can be used as an additional filter column to eliminate tuples that do not have a join partner for the join query given. Hence, the complete translation of query Q_2 for the RSPN learned over the full outer join in Figure 9.4b is $|\texttt{C} ⟗ \texttt{O}| \cdot P(\texttt{o\_channel='ONLINE'} \cap \texttt{c\_region='EU'} \cap \mathcal{N}_O = 1 \cap \mathcal{N}_C = 1) = 5 \cdot \frac{1}{5} = 1$.

CASE 2: LARGER RSPN AVAILABLE The second case is that we have to use an RSPN that was created on a set of joined tables, however, the query only needs a subset of those tables. For example, let us assume that the query Q_1 asking for European customers is approximated using an RSPN learned over a full outer join of customers and orders such as the one in Figure 9.4b. The problem here is that customers with multiple orders would appear several times in the join and thus be counted multiple times. For instance, the ratio of European customers in the full outer join is 3/5 though two out of three customers in the dataset are European.

To address this issue, for each foreign key relationship $S \leftarrow P$ between tables P and S we add a column $\mathcal{F}_{S \leftarrow P}$ to table S denoting how many corresponding join partners a tuple has. We call these *tuple factors* and later use them as correction factor. For instance, in the customer table in Figure 9.4a for the first customer the tuple factor is two since there are two tuples in the order table for this customer. It is important to note that tuple factors have to be computed only once per pair of tables that can be joined via a foreign key. In DeepDB, we do

this when the RSPNs for a given database are created and our update procedure changes those values as well. Tuple factors are included as additional column and learned by the RSPNs just as usual columns. When used in a join, we denote them as $\mathcal{F}'_{S \leftarrow P}$. Since we are working with outer joins, the value of $\mathcal{F}'$ is at least 1.

We can now express the query counting European customers as $|\texttt{C} \mathbin{⟗} \texttt{O}| \cdot \mathbb{E}(1/\mathcal{F}'_{C \leftarrow O} \cdot \mathbf{1}_{\texttt{c\_region='EU'}} \cdot \mathcal{N}_C)$ which results in $5 \cdot \frac{1/2+1/2+1}{5} = 2$. First, this query both includes the first customer (who has no orders) because the RSPN was learned on the full outer join. Second, the query also takes into account that the second and third customer have two orders each by normalizing them with their tuple factor $\mathcal{F}'_{C \leftarrow O}$. In general, we can define the procedure to compile a query requiring only a part of an RSPN as follows:

Theorem 1 *Let Q be a* `COUNT` *query with a filter predicate C which only queries a subset of the tables of a full outer join J. Let $\mathcal{F}'(Q, J)$ denote the product of all tuple factors that cause result tuples of Q to appear multiple times in J. The result of the query is equal to:*

$$|J| \cdot \mathbb{E}\left(\frac{1}{\mathcal{F}'(Q,J)} \cdot 1_C \cdot \prod_{T \in Q} \mathcal{N}_T\right)$$

For an easier notation, we write the required factors of query Q as $\mathbf{F}(Q)$. The expectation $\mathbb{E}(\mathbf{F}(Q))$ of theorem 1 can be computed with an RSPN because all columns are learned.

CASE 3: COMBINATION OF MULTIPLE RSPNS As the last case, we handle a `COUNT` query that needs to span over multiple RSPNs. We first handle the case of two RSPNs and extend the procedure to n RSPNs later. In this case, the query can be split into two subqueries Q_L and Q_R, one for each RSPN. There can also be an overlap between Q_L and Q_R which we denote as Q_O (i.e., a join over the shared common tables). The idea is first to estimate the result of Q_L using the first RSPN. We then multiply this result by the ratio of tuples in Q_R vs. tuples in the overlap Q_O. Intuitively, this expresses how much the missing tables not in Q_L increase the COUNT value of the query result.

For instance, there is a separate RSPN available for the `Customer` and the `Order` table in Figure 9.4a. The query Q_2, as shown before, would be split into two queries Q_L and Q_R, one against the RSPN built over the `Customer` table and the other one over the RSPN for the `Order` table. Q_O is empty in this case. The query result of Q_2 can thus be expressed using all these sub-queries as:

$$|\texttt{C}| \cdot \underbrace{\mathbb{E}(\mathbf{1}_{\texttt{c\_region='EU'}} \cdot \mathcal{F}_{C \leftarrow O})}_{Q_L} \cdot \underbrace{\mathbb{E}(\mathbf{1}_{\texttt{o\_channel='ONLINE'}})}_{Q_R}$$

which results in $3 \cdot \frac{2+0}{3} \cdot \frac{2}{4} = 1$. The intuition of this query is that the left-hand side that uses Q_L computes the orders of European

customers while the right-hand side computes the fraction of orders that are ordered online out of all orders.

We now handle the more general case that the overlap is not empty and that there is a foreign key relationship $S \leftarrow T$ between a table S in Q_O (and Q_L) and a table T in Q_R (but not in Q_L). In this case, we exploit the tuple factor $\mathcal{F}_{S \leftarrow T}$ in the left RSPN. We now do not just estimate the result of Q_L but of Q_L joined with the table T. Of course this increases the overlap which we now denote as Q'_O. As a general formula for this case, we obtain Theorem 2:

Theorem 2 *Let the filter predicates and tuple factors of $Q_L \setminus Q_O$ and $Q_R \setminus Q_O$ be conditionally independent given the filter predicates of Q_O. Let $S \leftarrow T$ be the foreign key relationship between a table S in Q_L and a table T in Q_R that we want to join. The result of Q is equal to*

$$|J_L| \cdot \mathbb{E}\left(\mathbf{F}(Q_L) \cdot \mathcal{F}_{S \leftarrow T}\right) \cdot \frac{\mathbb{E}\left(\mathbf{F}(Q_R)\right)}{\mathbb{E}\left(\mathbf{F}(Q'_O)\right)}.$$

Independence across RSPNs is often given since our ensemble creation procedure preferably learns RSPNs over correlated tables as discussed in Section 9.3.

Alternatively, we can start the execution with Q_R. In our example query Q_2 where Q_R is the query over the orders table, we can remove the corresponding tuple factor $\mathcal{F}_{C \leftarrow O}$ from the left expectation. However, we then need to normalize Q_L by the tuple factors to correctly compute the fraction of customers who come from Europe. To that end, the query Q_2 can alternatively be computed using:

$$|\mathtt{O}| \cdot \mathbb{E}(\mathbf{1}_{\mathtt{o\_channel='ONLINE'}}) \cdot \frac{\mathbb{E}\left(\mathbf{1}_{\mathtt{c\_region='EU'}} \cdot \mathcal{F}_{C \leftarrow O} \mid \mathcal{F}_{C \leftarrow O}\right)}{\mathbb{E}\left(\mathcal{F}_{C \leftarrow O} \mid \mathcal{F}_{C \leftarrow O} > 0\right)}$$

EXECUTION STRATEGY If multiple RSPNs are required to answer a query, we have several possible execution strategies. Our goal should be to handle as many correlations between filter predicates as possible because predicates across RSPNs are considered independent. For instance, assume we have both the `Customer`, `Order` and `Customer-Order` RSPNs of Figure 9.4 in our ensemble, and a join of customers and orders would have filter predicates on `c_region`, `c_age` and `o_channel`. In this case, we would prefer the `Customer-Order` RSPN because it can handle all pairwise correlations between filter columns (`c_region`-`c_age`, `c_region`-`o_channel`, `c_age`-`c_channel`). Hence, at runtime we greedily use the RSPN that currently handles the filter predicates with the highest sum of pairwise RDC values. We also experimented with strategies enumerating several probabilistic query compilations and using the median of their predictions. However, this was not superior to our RDC-based strategy. Moreover, the RDC values have already been computed to decide which RSPNs to learn. Hence, at runtime this strategy is very compute-efficient.

The final aspect is how to handle joins spanning over more than two RSPNs. To support this, we can apply Theorem 2 several times.

9.4.2 *Other Aggregate Queries*

So far, we only looked into COUNT queries without group-by statements. In the following, we first discuss how we extend our query compilation to also support AVG and SUM queries before we finally explain group-by statements as well as outer joins.

AVG QUERIES We again start with the case that we have an RSPN that exactly matches the tables of a query and later discuss the other cases. For this case, queries with `AVG` aggregates can be expressed as conditional expectations. For instance, the query

```
Q3: SELECT AVG(c_age) FROM CUSTOMER C
WHERE c_region='EU';
```

can be formulated as $|\texttt{C}| \cdot \mathbb{E}(\texttt{c\_age} \mid \texttt{c\_region='EU'})$ with the ensemble in Figure 9.4a.

However, for the case that an RSPNs spans more tables than required, we cannot directly use this conditional expectation because otherwise customers with several orders would be weighted higher. Again, normalization by the tuple factors is required. For instance, if the RSPN spans customers and orders as in Figure 9.4b for query Q_3 we use

$$\frac{\mathbb{E}\left(\frac{\texttt{c\_age}}{\mathcal{F}'_{C\leftarrow O}} \mid \texttt{c\_region='EU'}\right)}{\mathbb{E}\left(\frac{1}{\mathcal{F}'_{C\leftarrow O}} \mid \texttt{c\_region='EU'}\right)} = \frac{20/2 + 20/2 + 50}{1/2 + 1/2 + 1} = 35.$$

In general, if an average query for the attribute A should be computed for a join query Q with filter predicates C on an RSPN on a full outer join J, we use the following expectation to answer the average query:

$$\mathbb{E}\left(\frac{A}{\mathcal{F}'(Q,J)} \mid C\right) / \mathbb{E}\left(\frac{1}{\mathcal{F}'(Q,J)} \mid C\right).$$

The last case is where the query needs more than one RSPN to answer the query. In this case, we only use one RSPN that contains A and ignore some of the filter predicates that are not in the RSPN. As long as A is independent of these attributes, the result is correct. Otherwise, this is just an approximation. For selecting which RSPN should be used, we again prefer RSPNs handling stronger correlations between A and P quantified by the RDC values. The RCDs can also be used to detect cases where the approximation would ignore strong correlations with the missing attributes in P.

SUM QUERIES For handling `SUM` queries we run two queries: one for the `COUNT` and `AVG` queries. Multiplying them yields the correct result for the `SUM` query.

GROUP-BY QUERIES Finally, a `group by` query can be handled also by several individual queries with additional filter predicates for every group. This means that for n groups we have to compute n times more expectations than for the corresponding query without grouping. In our experimental evaluation, we show that this does not cause performance issues in practice if we compute the query on the model.

OUTER JOINS Query compilation can be easily extended to support outer joins as well (left/right/full). The idea is that we only filter out tuples that have no join partner for all inner joins (case 1 and 2 in Section 9.4.1) but not for outer joins (depending on the semantics of the outer join). Moreover, in case 3, the tuple factors $\mathcal{F}$ with value zero have to be handled as value one to support the semantics of the corresponding outer join.

9.5 DEEPDB EXTENSIONS

We now describe important extensions of our basic framework presented before.

9.5.1 *Support for Confidence Intervals*

Especially for AQP confidence intervals are important. However, SPNs do not provide those. After the probabilistic query compilation the query is expressed as a product of expectations. We first describe how to estimate the uncertainty for each of those factors and eventually how a confidence interval for the final estimate can be derived.

First, we split up expectations as a product of probabilities and conditional expectations. For instance, the expectation $\mathbb{E}(X \cdot 1_C)$ would be turned into $\mathbb{E}(X \mid C) \cdot P(C)$. This allows us to treat all probabilities for filter predicates C as a single binomial variable with probability $p = \prod P(C_i)$ and the amount of training data of the RSPN as $n_{samples}$. Hence, the variance is $\sqrt{n_{samples} p(1-p)}$. For the conditional expectations, we use the Koenig-Huygens formula $\mathbb{V}(X \mid C) = \mathbb{E}(X^2 \mid C) - \mathbb{E}(X \mid C)^2$. Note that also squared factors can be computed with RSPNs since the square can be pushed down to the leaf nodes. We now have a variance for each factor in the result.

For the combination we need two simplifying assumptions: (i) the estimates for the expectations and probabilities are independent, and (ii) the resulting estimate is normally distributed. In our experimental evaluation, we show that despite these assumptions our confidence intervals match those of typical sample-based approaches.

We can now approximate the variance of the product using the independence assumption by recursively applying the standard equation for the product of independent random variables: $\mathbb{V}(XY) = \mathbb{V}(X)\mathbb{V}(Y) + \mathbb{V}(X)\mathbb{E}(Y)^2 + \mathbb{V}(Y)\mathbb{E}(X)^2$. Since we know the variance

of the entire probabilistic query compilation and we assume that this estimate is normally distributed we can provide confidence intervals.

9.5.2 *Support for Updates*

The intuition of our update algorithm is to regard RSPNs as indexes. Similar to these, insertions and deletions only affect subtrees and can be performed recursively. Hence, the updated tuples recursively traverse the tree and passed weights of sum nodes and the leaf distributions are adapted. Our approach supports *insert* and *delete* operations, where an *update*-operation is mapped to a pair of *delete* and *insert* operations.

Algorithm 2 Incremental Update

```
1:  procedure UPDATE_TUPLE(node, tuple)
2:      if leaf-node then
3:          update_leaf_distribution(node, tuple)
4:      else if sum-node then
5:          nearest_child ← get_nearest_cluster(node, tuple)
6:          adapt_weights(node, nearest_child)
7:          update_tuple(nearest_child, tuple)
8:      else if product-node then
9:          for child in child_nodes do
10:             tuple_proj ← project_to_child_scope(tuple)
11:             update_tuple(child, tuple_proj)
```

The update algorithm is depicted in Algorithm 2. Since it is recursive, we have to handle sum, product and leaf nodes. At sum nodes (line 4) we have to identify to which child node the inserted (deleted) tuple belongs to determine which weight has to be increased (decreased). Since children of sum nodes represent row clusters found by *KMeans* during learning [118], we can compute the closest cluster center (line 5), increase (decrease) its weight (line 6) and propagate the tuple to this subtree (line 7). In contrast, product nodes (line 8) split the set of columns. Hence, we do not propagate the tuple to one of the children but split it and propagate each tuple fragment to the corresponding child node (lines 9-11). Arriving at a leaf node, only a single column of the tuple is remaining. We now update the leaf distribution according to the column value (line 2).

This approach does not change the structure of the RSPN, but only adapts the weights and the histogram values. If there are new dependencies as a result of inserts they are not represented in the RSPN. As we show in Section 9.6.1 on a real-word dataset, this typically does not happen, even for high incremental learning rates of 40%. Nevertheless, in case of new dependencies the RSPNs have to be rebuilt. This is solved by checking the database cyclically for changed dependencies by calculating the pairwise RDC values as explained in Section 9.5.3 on column splits of product nodes. If changes are detected in the dependencies, the affected RSPNs are regenerated. As for traditional indexes, this can be done in the background.

9.5.3 *Ensemble Optimization*

As mentioned before, we create an ensemble of RSPNs for a given database. The base ensemble contains either RSPNs for single tables or they span over two tables connected by a foreign key relationship if they are correlated. Correlations occurring over more than two tables are ignored so far since they lead to larger models and higher training times. In the following, we thus discuss an extension of our ensemble creation procedure that allows a user to specify a training budget (in terms of time or space) and DeepDB selects the additional larger RSPNs that should be created.

To quantify the correlations between tables, as mentioned already before, we compute the pairwise RDC values for every pair of attributes in the schema. For every pair of tables, we define the maximum RDC value between two columns $\max_{c \in T_i, c' \in T_j} rdc(c, c')$ as the dependency value. The dependency value indicates which tables should appear in the same RSPN and which not. For every RSPN the goal is to achieve a high mean of these pairwise maximal RDC values. This ensures that only tables with high pairwise correlation are merged in an RSPN.

The limiting factor (i.e., the constraint) for the additional RSPN ensemble selection should be the budget (i.e., extra time compared to the base ensemble) we allow for the learning of additional RSPNs. For the optimization procedure, we define the maximum learning costs as a factor B relative to the learning costs of the base ensemble C_{Base}. Hence, a budget factor $B = 0$ means that only the base ensemble would be created. For higher budget factors $B > 0$, additional RSPNs over more tables are learned in addition. If we assume that an RSPN r among the set of all possible unique RSPNs R has a cost $C(r)$, then we could formulate the optimization problem as a minimization of $\sum_{r \in \mathcal{E}} \overline{\{\max_{c \in T_i, c' \in T_j} rdc(c, c') \mid T_i, T_j \in r\}}$ subject to $\sum_{r \in \mathcal{E}} C(r) \leq B \cdot C_{Base}$.

However, estimating the real cost $C(r)$ (i.e., time) to build an RSPN r is hard and thus we can not directly solve the optimization procedure. Instead, we estimate the relative cost to select the RSPN r that has the highest mean RDC value and the lowest relative creation cost. To model the relative creation cost, we assume that the costs grow quadratic with the number of columns $cols(r)$ since the RDC values are created pairwise and linear in the number of rows $rows(r)$. Consequently, we pick the RSPN r with highest mean RDC and lowest cost which is $cols(r)^2 \cdot rows(r)$ as long as the maximum training time is not exceeded.

9.6 EXPERIMENTAL EVALUATION

In this Section, we show that DeepDB outperforms state-of-the-art systems for both cardinality estimation and AQP. The RSPNs we used

in all experiment were implemented in Python as extensions of SPFlow [119]. As hyperparameters, we used an RDC threshold of 0.3 and a minimum instance slice of 1% of the input data, which determines the granularity of clustering. Moreover, we used a budget factor of 0.5, i.e. the training of the larger RSPNs takes approximately 50% more training time than the base ensemble. We determined these hyperparameters using a grid-search, which gave us the best results across different datasets.

9.6.1 *Experiment 1: Cardinality Estimation*

WORKLOAD AND SETUP As in [79, 91], we use the JOB-light benchmark as workload for all approaches (DeepDB and baselines). The benchmark uses the real-world IMDb database and defines 70 queries. Furthermore, we additionally defined a synthetic query set of 200 queries were joins from three to six tables and one to five filter predicates appear uniformly on the IMDb dataset. We use this query set to compare the generalization capabilities of the learned approaches.

As baselines, we used the following learned and traditional approaches: First we trained a Multi-Set Convolutional Network (MCSN) [79] as a learned baseline. MCSNs are specialized deep neural networks using the join paths, tables and filter predicates as inputs. As representative of a synopsis-based technique, we implemented an approach based on wavelets [21]. The main idea of [21] is that one wavelet is built per table. Moreover, query operators (e.g., joins) can be executed directly on the wavelet representation. We have chosen this approach because it is similar to DeepDB since the tables that are joined by queries do not have to be known beforehand. We also implemented an approach called *Perfect Selectivities*. In this approach, we use an oracle that returns the true cardinalities for single tables. This approach can be seen as the best case for any synopsis-based approach that supports ad-hoc queries by combining "perfect" synopsis on single tables. Finally, we use the standard cardinality estimation of Postgres 11.5 as well as online random sampling and Index-Based Join Sampling (IBJS) [92] as a non-learned baselines. Similar to DeepDB, IBJS considers potential correlations across tables when sampling. For DeepDB, we use the hyper-parameters discussed before and a sample size of 10M samples for constructing RSPNs if not noted otherwise.

TRAINING TIME AND STORAGE OVERHEAD In contrast to other learned approaches for cardinality estimation [79, 161], no dedicated training data is required for DeepDB. Instead, we just learn a representation of the data. The training of the base ensemble takes 48 minutes. The creation time includes the data preparation time to sample and compute the tuple factors as introduced in Section 9.4.1. In contrast, for the MCSN [79] approach, 100k queries need to be executed to collect cardinalities resulting in 34 hours of training data preparation

Table 9.1: Estimation Errors for the JOB-light Benchmark

	median	90th	95th	max
DeepDB	1.34	2.50	3.16	39.63
DeepDB (Storage Opt.)	1.32	4.14	5.74	72.00
Perfect Selectivities	2.08	9	11	33
MCSN	3.22	65	143	717
Wavelets	7.64	9839	15332	564549
Postgres	6.84	162	817	3477
IBJS	1.67	72	333	6949
Random Sampling	5.05	73	10371	49187

time (when using Postgres). Moreover, the training of the neural network takes only about 15 minutes on a Nvidia V100 GPU. As we can see, our training time is much lower since we do not need to collect any training data for the workload. Another advantage is that we do not have to re-run the queries once the database is modified. Instead, we provide an efficient algorithm to update RSPNs in DeepDB as discussed in Section 9.3.2.

Another dimension is the storage footprint needed for the different approaches. While the sampling-based approaches, i.e., IBJS and random sampling, do not incur a storage overhead, their limiting factor is the number of samples which is´ determined by the latency. All other approaches require only a few KB to MB of storage for the IMDb database of the JOB-light benchmark (which uses 3.7 GB disk space). The storage overhead of DeepDB is 28.9MB vs. 2.6 MB for MSCN and just 60kB for Postgres that uses histograms with just 100 buckets by default (however with the lowest accuracy as we show next). For the wavelet approach we used $20k$ wavelet coefficients to allow as much storage as the standard version of DeepDB requires. In addition, we also created a storage-optimized version of DeepDB, which has a similar storage footprint as MCSNs by reducing the number of samples. In contrast to DeepDB, allowing a larger storage overhead for MSCNs by for instance adding hidden layers does not improve the performance since we already use the optimized hyperparameters of [79]. As we show next, the storage-optimized version of DeepDB can provide accuracies that are still significantly better than all other baselines including MCSN. Furthermore, while there has been a line of research optimizing the storage footprint of DNNs there are no comparable approaches for SPNs. We believe that future research will reduce the storage requirements for DeepDB even further. However, we think that even a few MB of storage for an entire database of several GB is still acceptable for more accurate cardinality estimates.

ESTIMATION QUALITY The prediction quality of cardinality estimators is usually evaluated using the q-error, which is the factor by which an estimate differs from the real execution join size. For example, if

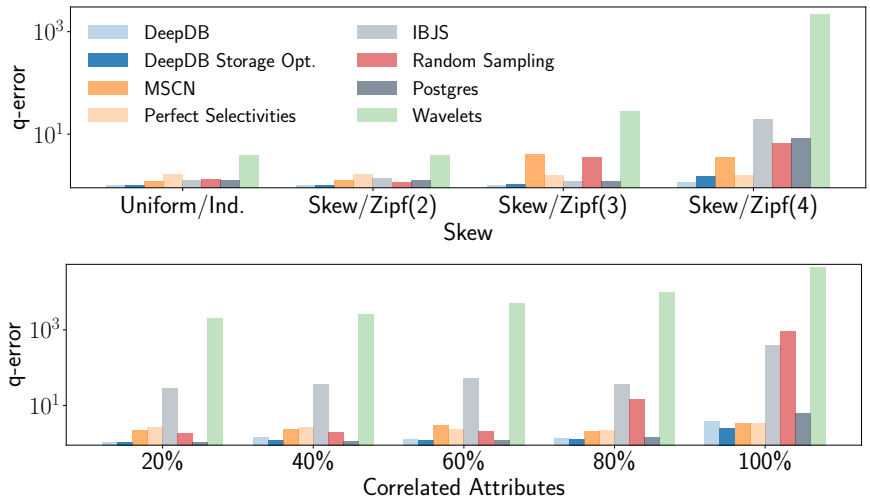


Figure 9.5: Mean Estimation Errors for Synthetic Data.

the real result size of a join is 100, the estimates of 10 or 1k tuples both have a q-error of 10. Using the ratio instead of an absolute or quadratic error captures the intuition that for making optimization decisions only relative differences matter. In Table 9.1, we depict the median, 90-th and 95-th percentile and max q-errors for the JOB-light benchmark of our approach compared to the other baselines. We additionally provide the q-errors for a storage-optimized version of DeepDB, which relies only on a base ensemble and 100k samples per RSPN. As we can see, both DeepDB and the storage-optimized version outperform the best competitors often by orders of magnitude. While IBJS provides a low q-error in the median, the advantage of learned MCSNs is that they outperform traditional approaches by orders of magnitude for the higher percentiles and are thus more robust. DeepDB not only outperforms IBJS in the median, but provides additional robustness having a 95-th percentile for the q-errors of 3.16 vs. 143 (MCSN). The q-errors of both Postgres and random sampling are significantly larger both for the medians and the higher percentiles. Finally, wavelets have the highest error since they suffer from the curse of dimensionality (as we show later in Figure 9.12). While *Perfect Selectivities* which is based on an oracle provides errors better than wavelets it is still worse than DeepDB since it does not take correlations across tables into account.

SYNTHETIC DATA In order to further investigate the tradeoffs of the different approaches, we implemented a synthetic data generator for the IMDb schema (such that we can then run the JOB-light benchmark). First, we generated data with uniform distributions without any correlations. Second, we varied the characteristics that make cardinality estimation hard in reality; i.e., we used skewed distributions and correlations between different columns. We then used the same approaches as before to provide cardinality estimates for the original 70 JOB-light queries and report the mean q-errors of queries not having a cardinality of zero because otherwise the q-error is not defined. Figure 9.5

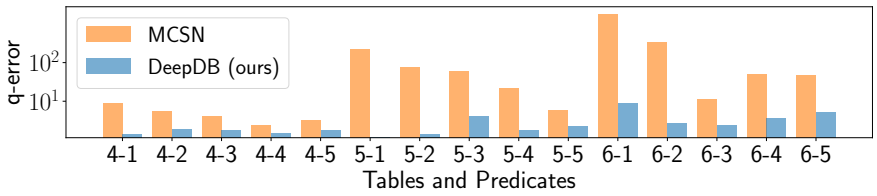


Figure 9.6: Median q-errors (logarithmic Scale) for different Join Sizes (4,5,6) and Number of Filter Predicates (1-5).

shows the mean q-errors (log-scale) for varying degrees of skew (upper plot) and varying degrees of correlation (lower plot). We can see that DeepDB and the storage optimized version can both outperform all other baselines. While on uniform/independent data, DeepDB provides no significant advantage even over simple techniques such as random sampling or Postgres (as expected), DeepDB outperforms the other baselines for higher degrees of skew/correlation. For higher degrees of skew/correlation, the approaches based on sampling (random sampling, IBJS) as well as Postgres all degrade significantly. Compared to those approaches, MSCN can handle skew/correlation much better but still degrades which we attribute again to the coverage of the training queries. Finally, wavelets again provide the lowest accuracy on all configurations since they suffer from the curse of dimensionality similar to the real-world data in Figure 9.1.

GENERALIZATION CAPABILITIES Especially for learned approaches, the question of generalization is important, i.e., how well the models perform on previously unseen queries. For instance, by default the MCSN approach is only trained with queries up to three joins because otherwise the training data generation would be too expensive [79]. Similarly in our approach, in the ensemble only few RSPNs with large joins occur because otherwise the training would also be too expensive. However, both approaches support cardinality estimates for unseen queries.

To compare both learned approaches, we randomly generated queries for joins with four to six tables and one to five selection predicates for the IMDb dataset. In Figure 9.6, we plot the resulting median q-errors for both learned approaches: DeepDB and MCSN [79]. The median q-errors of DeepDB are orders of magnitude lower for larger joins. Additionally, we can observe that, for the MCSN approach, the estimates tend to become less accurate for queries with fewer selection predicates. One possible explanation is that more tuples qualify for such queries and thus higher cardinalities have to be estimated. However, since there are at most three tables joined in the training data such higher cardinality values are most likely not predicted. Thus, using RSPNs leads to superior generalization capabilities.

UPDATES In this experiment, we show the update capabilities of RSPNs. The easy and efficient updateability is a clear advantage of

Table 9.2: Estimation Errors for JOB-light after Updates.

Temporal Split	< 2019 (0%)	< 2011 (4.7%)	< 2009 (9.3%)	< 2004 (19.7%)	< 1991 (40.1%)
Median	1.22	1.28	1.31	1.34	1.41
90th	3.45	3.17	3.23	3.60	4.06
95th	4.77	4.30	3.83	4.07	4.35

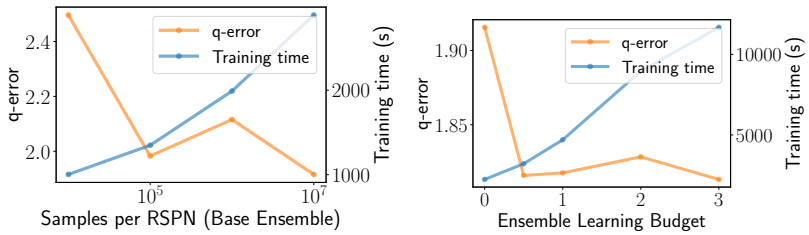


Figure 9.7: Q-errors and Training Time (in s) for varying Budget Factors and Sample Sizes.

DeepDB compared to deep-learning based approaches for cardinality estimation [79, 161]. To show the effects of updates on the accuracy, we first learn the base RSPN ensemble on a certain share of the full IMDb dataset and then use the remaining tuples to update the database.

To ensure a realistic setup, we split the IMDb dataset based on the production year (i.e., newer movies are inserted later). As depicted in Table 9.2 the q-errors do not change significantly for updated RSPNs even if the update fraction increases; i.e., if we split on earlier production years. For building the RSPNs, we use zero as the budget factor to demonstrate that even a base RSPN ensemble provides good estimates after updates. This is also the reason why the estimation errors slightly deviate from Table 9.1. Our results in Table 9.2 show that with a higher fraction of updates, the accuracy drops only slightly. The reason is that the structure of the RSPN tree is not changed by updates, but only the parameters of the RSPNs which might not be the optimal structure anymore if the data distributions/correlations change due to the updates. In case the accuracy drops beyond a threshold, DeepDB can still decide to recreate the RSPN offline based on the new data.

PARAMETER EXPLORATION Finally, in the last experiment, we explore the tradeoff between ensemble training time and prediction quality of DeepDB. We first vary the budget factor used in the ensemble selection between zero (i.e. learning only the base ensemble with one RSPN per join of two tables) and B=3 (i.e. the training of the larger RSPNs takes approximately three times longer than the base ensemble) while using 10^7 samples per RSPN. We then use the resulting ensemble to evaluate 200 queries with three to six tables and one to five selection predicates. The resulting median q-errors are shown in Figure 9.7. For higher budget factors the means are improving but already saturate at $B = 0.5$. This is because there are no

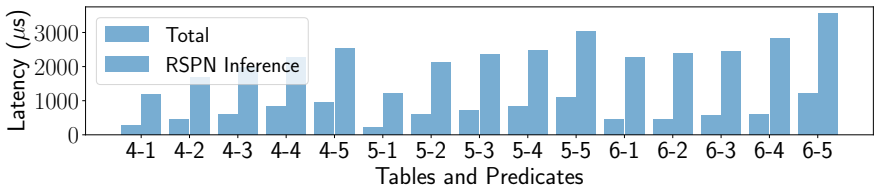


Figure 9.8: Latencies of DeepDB for different Join Sizes (4,5,6) and Number of Filter Attributes (1-5).

strong correlations in larger joins that have not already been captured in the base ensemble.

Moreover, we evaluate the effect of the sampling to reduce the training time. In this experiment we vary the sample size from 1000 to 10 million. We observe that while the training time increases, the higher we choose this parameter, the prediction quality improves (from 2.5 to 1.9 in the median). In summary, the training time can be significantly reduced if slight compromises in prediction quality are acceptable. When minimization of training time is the more important objective we can also fall back and only learn RSPNs for all single tables and no joins at all. This reduces the ensemble training time to just five minutes. However, even this cheap strategy is still competitive. For JOB-light this ensemble has a median q-error of 1.98, a 90-th percentile of 5.32, a 95-th percentile of 8.54 and a maximum q-error of 186.53. Setting this in perspective to the baselines, this ensemble still outperforms state of the art for the higher percentiles and only Index Based Join Sampling is slightly superior in the median. This again proves the robustness of RSPNs.

LATENCIES The estimation latencies for cardinalities using DeepDB are currently in the order of μs to *ms* which suffices for complex join queries that often run for multiple seconds on larger datasets. If more complex predicates spanning over several columns are used or more tables are involved in the join the latencies increase. In Figure 9.8 we investigate this effect in more detail. We report both the latency required for the RSPN inference and the total time including the overhead of translating the queries to expectations and probabilities using our probabilistic query compilation procedure. The RSPN inference is efficient because C++ code is compiled automatically for the trained RSPNs similar to [158]. As we see, while the latencies increase for more complex predicates and joins, they are still around 3ms in the worst case and in the range of μs for easier queries. In future, we plan to optimize not just RSPN inference but also the overhead of query translation to bring the total latency even closer to only the RSPN inference.

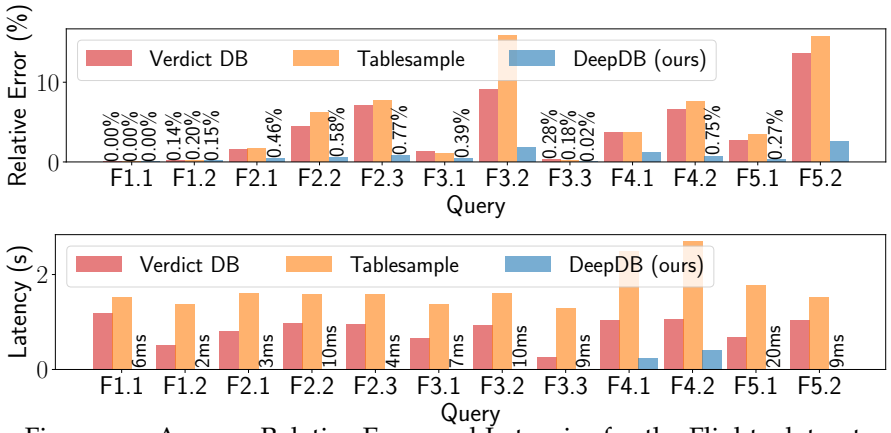


Figure 9.9: Average Relative Error and Latencies for the Flights dataset.

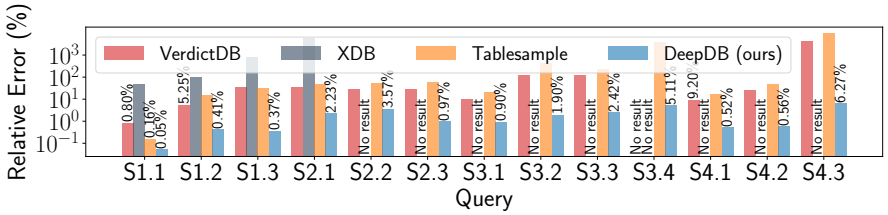


Figure 9.10: Average Relative Error for SSB dataset. Note the logarithmic Scale for the Errors.

9.6.2 *Experiment 2: AQP*

WORKLOAD AND SETUP We evaluated the approaches on both a synthetic dataset and a real-world dataset. As synthetic dataset, we used the Star Schema Benchmark (SSB) [128] with a scale factor of 500 with the standard queries (denoted by S1.1-S4.3). As the real-world dataset, we used the Flights dataset [46] with queries ranging from selectivities between 5% an 0.01% covering a variety of group by attributes, `AVG`, `SUM` and `COUNT` queries (denoted by F1.1-F5.2). To scale the dataset up to 1 billion records we used IDEBench [41].

As baselines we used VerdictDB [135], Wander Join/XDB [95] and the Postgres `TABLESAMPLE` command (using random samples). VerdictDB is a middleware that can be used with any database system. It creates a stratified and a uniform sample for the fact tables to provide approximate queries. For VerdictDB, we used the default sample size (1% of the full dataset) for the Flights dataset. For the SSB benchmark, this led to high query latencies and we thus decided to choose a sample size such that the query processing time was two seconds on average. Wander Join is a join sampling algorithm leveraging secondary indexes to generate join samples quickly. We set the time bound also to two seconds for a fair comparison and only evaluated this algorithm for datasets with joins. To this end, we created all secondary indexes for joins and predicates. For `TABLESAMPLE` we chose a sample size such that the queries take two seconds on average. For DeepDB, we use a sample size of 10M samples for the Flights dataset and 1M samples for the SSB dataset to construct RSPNs.

TRAINING TIME AND STORAGE OVERHEAD The training took just 17 minutes for the SSB dataset and 3 minutes for the Flights dataset. The shorter training times compared to the IMDb dataset are due to fewer cross-table correlations and hence fewer large RSPN models in the ensemble. For VerdictDB, uniform and stratified samples have to be created from the dataset. This took 10 hours for the flights dataset and 6 days for the SSB benchmark using the standard setup of VerdictDB.For wander join, secondary indexes had to be created also requiring several hours for the SSB dataset.

For the Flights dataset the model size of DeepDB is 2.2 MB (vs. 11.4 MB for VerdictDB) and for the SSB dataset DeepDB requires 34.4 MB (vs. 30.7 MB for VerdictDB). In contrast to DeepDB and VerdictDB, XDB and Postgres `TABLESAMPLE` compute samples online and thus do not have any additional (offline) storage overhead.

ACCURACY AND LATENCY For AQP two dimensions are of interest: the quality of the approximation and the runtime of queries. For reporting the quality of the approximation we use the relative error which is defined as $\frac{|a_{true} - a_{predicted}|}{a_{true}}$ where a_{true} and $a_{predicted}$ are the true and predicted aggregate function, respectively. If the query is a group by query, several aggregates have to be computed. In this case, the relative error is averaged over all groups.

For the Flights dataset, as shown in Figure 9.9 we can observe that DeepDB always has the lowest average relative error. This is often the case for queries with lower selectivities where sample-based approaches have few tuples that satisfy the selection predicates and thus the approximations are very inaccurate. In contrast, DeepDB does not rely on samples but models the data distribution and leverages the learned representation to provide estimates. For instance, for query 11 with a selectivity of 0.5% VerdictDB and the `TABLESAMPLE` strategy have an average relative error of 15.6% and 13.6%, respectively. In contrast, the average relative error of DeepDB is just 2.6%.

Moreover, the latencies for both `TABLESAMPLE` and VerdictDB are between one and two seconds on average. In contrast, DeepDB does not rely on sampling but on evaluating the RSPNs. This is significantly faster resulting in a maximum latency of 31ms. This even holds true for queries with several groups where more expectations have to be computed (at least one additional per different group).

The higher accuracies of DeepDB are even more severe for the SSB benchmark. The queries have even lower selectivities between 3.42% and 0.0075% for queries 1 to 12 and 0.00007% for the very last query. This results in very inaccurate predictions of the sample-based approaches. Here, the average relative errors are orders of magnitude lower for DeepDB always being less than 6%. In contrast, VerdictDB, Wander Join and the `TABLESAMPLE` approach often have average relative errors larger than 100%. Moreover, for some queries no estimate can be given at all because no samples are drawn that satisfy the filter

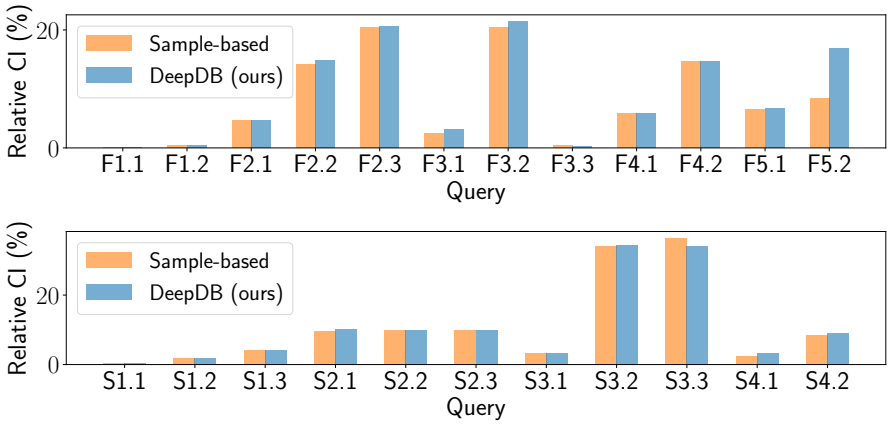


Figure 9.11: True and predicted relative length of the Confidence Intervals.

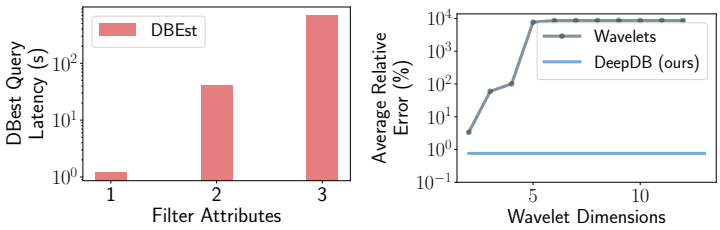


Figure 9.12: Microbenchmarks for non ad-hoc Approaches.

predicates. However, while the other approaches take two seconds to provide an estimate, DeepDB requires no more than 293ms in the worst case. In general latencies are lower for queries with fewer groups because less expectations have to be computed.

CONFIDENCE INTERVALS In this experiment, we evaluate how accurate the confidence intervals predicted by DeepDB are. To this end, we measure the relative confidence interval length defined as: $\frac{a_{predicted} - a_{lower}}{a_{predicted}}$, where $a_{predicted}$ is the prediction and a_{lower} is the lower bound of the confidence interval. This relative confidence interval length is compared to the confidence interval of a sample-based approach. For this we draw 10 million samples (as many samples as our models use for learning in this experiment) and compute estimates for the average, count and sum aggregates. We then compute the confidence intervals of these estimates using standard statistical methods. The resulting confidence interval lengths can be seen as ground truth and are compared to the confidence intervals of our system in Figure 9.11. Note that we excluded queries where less than 10 samples fulfilled the filter predicates. In these cases the estimation of a standard deviation has itself a too high variance.

In all cases, the confidence intervals of DeepDB are very good approximations of the true confidence intervals. The only exception is query F5.2 for the Flights dataset which is a difference of two `SUM` aggregates. In this case, assumption (i) of Section 9.5.1 does not hold: the probabilities and expectation estimates cannot be considered

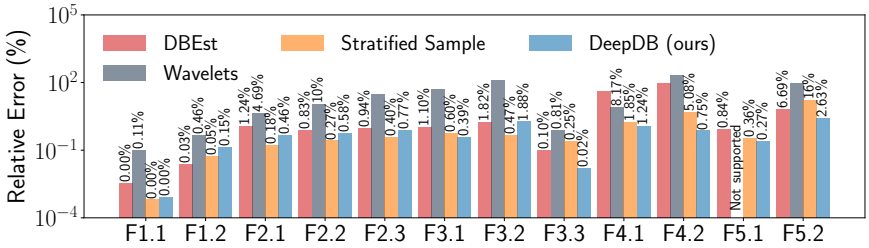


Figure 9.13: Relative Error for DeepDB vs. Approaches with a priori Knowledge.

independent. This is the case because both SUM aggregates contain correlated attributes and thus the confidence intervals are overestimated. However, note that in the special case of the difference of two sum aggregates the AQP estimates are still very precise as shown in Figure 9.9 for the same query F5.2. Such cases can easily be identified and only occur when arithmetic expressions of several aggregates should be estimated.

NON AD-HOC APPROACHES We now compare the accuracy of DeepDB against approaches that either require a priori information about the workload or can make use of it to provide better accuracies. The results of DeepDB and the other approaches on the Flights dataset are shown in Figure 9.13.

First, we compared DeepDB to the wavelet approach [21] that we used before. As discussed, this approach does not require a priori information but a priori knowledge can be used to optimize wavelets since they do not scale to a large number of dimensions. Thus, if the required column combinations of all queries are known beforehand, we can construct optimal (minimal-sized) wavelets on a per-query basis. However, as shown in Figure 9.13, even when using a priori information for wavelets, DeepDB still outperforms them. We investigated this effect further and found that the accuracy of wavelets significantly drops even for a small number of dimensions as shown in Figure 9.12 (right).

Second, we also compared DeepDB to other approaches that require a priori information: (1) stratified sampling as used in BlinkDB [2] which mitigates effects of skew and (2) a recent learned AQP approach called DBEst [105]. It is important to note that different from DeepDB these approaches cannot answer ad-hoc queries with column combinations not covered in the a priori information which is possible in DeepDB by combining multiple RSPNs using our probabilistic query compilation approach as discussed in Section 9.4. In the following, we discuss the results if a priori information for all queries on the Flights dataset is available. As shown in Figure 9.13 for these queries, stratified sampling can provide accuracies comparable to DeepDB. However, interestingly if a query has highly selective filter conditions on one of the stratification columns (as in query F5.2 or F4.2), DeepDB

is still superior. Moreover, while for DBEst the accuracies are also comparable to DeepDB we noticed that for some queries it takes up to 20s to provide an answer. We investigated this effect closer in Figure 9.12 (left) finding that the latency is exponential w.r.t. the number of filter conditions on numeric columns. This is inevitable since the approach relies on an integration over the domains of numeric columns.

9.7 RELATED WORK

LEARNED CARDINALITY ESTIMATION The problem of selectivity estimation for single tables is a special case of cardinality estimation. There is a large body of work applying different ML approaches including probabilistic graphical models [52, 53, 170], neural networks [87, 100] and deep learning density models [61] to this problem. Recently, Dutt et al. [39] suggested using lightweight tree-based models in combination with log-transformed labels. The first works applying ML to cardinality estimation including joins used simple regression models [5, 106]. More recently, Deep Learning was specifically proposed to solve cardinality estimation end-to-end [79, 161]. Woltmann et al. [179] also separate the problem of cardinality estimation on a large schema by learning models similar to [79] for certain schema subparts. However, two models for schema sub-parts cannot be combined to provide estimates for a larger join. Other techniques exploit learned models for overlapping subgraph templates for recurring cloud workloads [180]. All these models need a workload to be executed and used as training data which is different from our data-driven approach.

LEARNED AQP Early work [152] suggests to approximate OLAP cubes by mixture models based on clusters in the data. Though greatly reducing the required storage, the approximation errors are relatively high. *FunctionDB* [166] constructs piecewise linear functions as approximation. In contrast to DeepDB, only continuous variables are supported. *DBEst* [105] builds models for popular queries. However, in contrast to DeepDB slight variations of those popular queries and no ad-hoc queries are supported. Park et al. suggested *Database Learning* [136] which builds a model from query results that is leveraged to provide approximate results for future queries. In contrast, DeepDB is data-driven and does not require past query results. Moreover, specialized generative models were suggested to draw samples for AQP [167]. However, this technique does not work for joins.

PROBABILISTIC DATABASES Similar to our work, probabilistic databases have used graphical models to represent joint probability distributions [75, 147, 151, 175] to overcome the tuple-independence assumption that early approaches relied on [31, 130, 160]. For instance, Markov Logic Networks (MLNs) were used to explicitly specify correlations in probabilistic databases [58, 75]. In contrast to DeepDB, correlations need to be manually specified and are not learned. Moreover,

Rekatsinas et al. [147] instead use factor graphs to model correlations in probabilistic databases and construct annotated arithmetic circuits (AACs) which also encode a probability distribution with sum and product nodes similar to SPNs. Different from DeepDB, additional representations (lineage-AACs) have to be constructed on a per-query basis whereas RSPNs are data-driven and thus workload-independent. Finally, related to the idea of computing a data-driven model is generally the field of knowledge compilation where an expensive offline phase creates a representation for instance for evaluating Boolean formulas [14, 15, 32]. However, non of these approaches targets the complexity of SQL queries supported in DeepDB.

9.8 CONCLUSION AND FUTURE WORK

In this work we have proposed DeepDB which is a data-driven approach for learned database components. We have shown that our approach is general and can be used to support various tasks including cardinality estimation and approximate query processing. We believe our data-driven learning approach can also be used for other DBMS components. For instance, it has already been shown that column correlations can be exploited to improve indexing [182]. In addition, SPNs naturally provide a notion of correlated clusters that can also be used for suggesting using interesting patterns in data exploration.

9.9 ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their helpful feedback. The work partly grew out of discussions within and support of the missions “Federated Machine Learning” and “ML for Vulnerability Detection” of the National Research Center ATHENE, the AI lighthouse BMWi project SPAICER (01MK20015E), and the DFG Training Group “Adaptive Preparation of Information from Heterogeneous Sources” (AIPHES, GRK 1994/1).

10

RESTORE- NEURAL DATA COMPLETION FOR RELATIONAL DATABASES

ABSTRACT

Classical approaches for OLAP assume that the data of all tables is complete. However, in case of incomplete tables with missing tuples, classical approaches fail since the result of a SQL aggregate query might significantly differ from the results computed on the full dataset. Today, the only way to deal with missing data is to manually complete the dataset which causes not only high efforts but also requires good statistical skills to determine when a dataset is actually complete. In this paper, we propose an automated approach for relational data completion called ReStore using a new class of (neural) schema-structured completion models that are able to synthesize data which resembles the missing tuples. As we show in our evaluation, this efficiently helps to reduce the relative error of aggregate queries by up to 390% on real-world data compared to using the incomplete data directly for query answering.

BIBLIOGRAPHIC INFORMATION

The content of this chapter was previously published in the peer-reviewed work Benjamin Hilprecht and Carsten Binnig. "ReStore - Neural Data Completion for Relational Databases." In: *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. Ed. by Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava. ACM, 2021, pp. 710–722. DOI: 10.1145/3448016.3457264. URL: https://doi.org/10.1145/3448016.3457264. The contributions of the author of this dissertation are summarized in Section 4.2.

10.1 INTRODUCTION

MOTIVATION. OLAP and data warehousing play a significant role today for many organizations and enterprises for decision making. This is evident since many new scalable OLAP services are becoming available in the cloud such as AWS Redshift [10], Snowflake [157], or Azure data warehousing [11] that allow customers to analyze large datasets using aggregate queries. A critical assumption for OLAP, however, is that the data itself has to be complete before it can be used for decision making, i.e., data in tables is complete and no tuples are missing. Traditionally, this was achieved by loading data only from well curated (internal) data sources into a data warehouse. In enterprises, these are typically OLTP systems that store data about customers, products, orders, etc. However, while data in this context might still require data integration and cleaning [26, 36, 56, 177] since it comes from multiple sources, the data is typically considered complete and all relevant tuples were expected to be present in the data warehouse.

However, this assumption does not hold anymore for many of the more modern analytics scenarios. Instead of using only well curated (internal) data sources in a data warehouse, more and more external data sources are being used in OLAP scenarios. A problem of these external data sources is that the data might be incomplete. For example, to extend our warehouse we might want to use a CSV file from an open data platform containing information about cities where customers come from — however, data for some cities is missing. Moreover, in addition to external data sources there are many more applications where tables can be incomplete such as scenarios where data needs to be collected manually and thus collecting a complete dataset is too expensive or even impossible.

In case of incomplete tables, classical databases fail since the result of a SQL aggregate query might significantly differ from the results computed on the full dataset which in turn leads to erroneous conclusions in data analysis and decision making. Moreover, existing techniques that can produce approximate aggregate query answers [2, 22] on samples might also fail since data is often missing systematically (e.g., samples for some groups in the data are missing completely). Even worse, the missing data might introduce a bias and hence the data can not be seen as a uniform (random) sample.

For example, suppose we want to create a housing database of rental apartments and their neighborhoods (covering all cities in the US). While we have a complete neighborhoods table, the apartments data is incomplete since not all states provide this information (i.e., apartments of individual states might be missing completely). However, this missing data might introduce a bias in the available data, e.g., since most data comes from states with high population densities

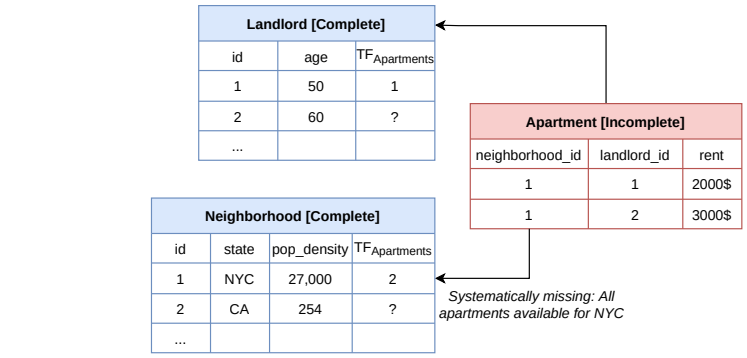
where rents are higher. If we now use a SQL aggregate query on the incomplete apartment table to determine the average rental price of apartments across all states, we could obtain (highly) inaccurate results due to the missing apartment tuples.

CONTRIBUTIONS. The only way to deal with missing tuples in databases for OLAP today is to manually add the missing tuples before using the database for decision making. The manual completion of an incomplete database, however, causes an enormous effort in data acquisition and in checking the completeness of the acquired data. Moreover, in many situations it might not even be possible to complete a database manually at all. In this paper, we thus propose a new learned approach called ReStore for automatic data completion for incomplete relational databases. While there has been already significant work to impute missing values (e.g., replace a missing attribute) including learned approaches [146, 181, 187], to the best of our knowledge there is no work to synthesize data for incomplete tables in a relational schema where tuples are missing and might introduce a bias.

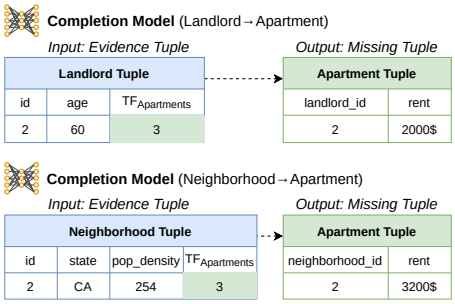
The main idea of our approach is that we use the complete tables in a database as evidence to synthesize the missing data even if the missing data introduces a bias in the incomplete table. For instance, in the example above, we could use the complete neighborhoods table to synthesize the apartment tuples for the missing states. One might now wonder how the bias from the missing data can be removed. The intuition is that our neural completion models learn from the available data how typical apartments look like based on information from the neighborhood table (e.g., rents will be higher in neighborhoods with higher population density). During completion, we take this information from neighborhoods into account to synthesize the missing tuples.

To enable data completion our approach works in two steps (cf. Figure 10.1): (1) in a first step, the user has to annotate the schema and provide minimal information about the relational dataset once for all queries (such as if a table is complete or incomplete). (2) Once annotated, we learn so called completion models over the incomplete dataset to capture the complex correlations and dependencies across complete and incomplete tables. Using these models, we are then able to synthesize data to complete the missing data for executing aggregate queries. As we show in our evaluation, this efficiently helps to reduce the relative error of aggregate queries by up to 390% on real-world data compared to using the incomplete data directly for decision making.

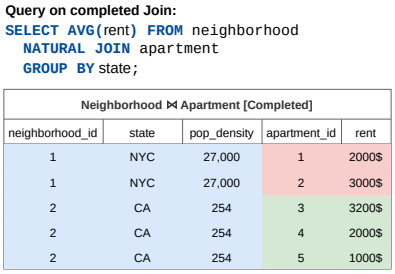
OUTLINE. The remainder of the paper is structured as follows. In Section 10.2, we provide a more formal definition of the problem and



(a) Annotated Example Schema.



(b) Models Synthesize Missing Tuples.



(c) Incompleteness Join.

Figure 10.1: Overview of ReStore to synthesize missing data (green) from existing data (blue and red). (a) Based on the annotated schema and the available data, the completion models are learned. (b) The learned schema-structured model can be used to synthesize a missing `apartment` tuple using a complete `neighborhood` tuple as input. (c) The model generates missing data for a given user query at runtime to answer queries over incomplete tables. The generated tuple factors (TFs) allow us to estimate the number of missing tuples.

present an overview of ReStore to tackle this problem. Afterwards, in Section 10.3 we present the details of the neural completion models before we then discuss in Section 10.4 how these models can be used to generate the missing data for answering aggregate queries. Furthermore, Section 10.5 provides further important details on automatic selection of completion models given a user query before we discuss a technique to estimate the confidence of a completion in Section 10.6. In Section 10.7, we discuss the results of our evaluation using synthetic and real-world datasets. Finally, we present related work in Section 10.8 and then conclude in Section 10.9.

10.2 OVERVIEW

In this section, we introduce the problem statement before we give an overview of our approach and discuss the general assumptions.

10.2.1 *Problem Statement*

In brief, the problem that we solve in this paper can be described as follows. We are given an incomplete database D^i that consists of complete tables $T_1, T_2, \ldots$ and incomplete tables $T_j, T_{j+1}, \ldots$. The goal is to generate data for the incomplete tables $T_j, T_{j+1}, \ldots$ based on the available data that allows us to answer a query workload $Q_1, Q_2, \ldots, Q_n$ of aggregate queries such that query results on the completed database $Q_i(D^c)$ are close to the query results on the true (complete) database $Q_i(D)$. Note that this formulation allows us to generate missing data individually for each query to answer the given query as accurately as possible. However, we can still cache generated data such that we do not need to generate new data for every query individually as we discuss later.

An important question for this problem is how to measure success. Based on our problem definition, a natural metric is how much the relative error of a query result on the incomplete database can be reduced by completing the data; i.e., how much more accurate the query results are after the completion. The relative error reduction for a given query Q_i can thus be defined as follows:

$$\textit{Rel. Error Reduction} = E_r(Q_i(D^i), Q_i(D)) - E_r(Q_i(D^c), Q_i(D)) \quad (10.1)$$

where the relative error E_r is the difference of the two query results normalized by the true query result. While for aggregate queries without a group-by, the relative error is trivial, for group-by queries we use the average relative error over all result tuples [72].

A limitation of the relative error reduction metric is that it does not show how well the bias of the incomplete database can be reduced independent of a given workload. We thus use a second metric called *bias reduction* to measure the success of data completion. This metric shows how well the true data distribution of a given attribute could be restored. For continuous attributes X, the bias reduction is defined as follows:

$$\textit{Bias Reduction} = 1 - \frac{|AVG^c(X) - AVG(X)|}{|AVG(X) - AVG^i(X)|} \quad (10.2)$$

where $AVG^c(X), AVG^i(X), AVG(X)$ are the averages of attribute X on D^c, D^i and D, respectively. Hence, the bias reduction is normalized in the interval $[0, 1]$ where larger values are preferable. For categorical attributes, we use the fraction of the biased attribute value since an average cannot be computed.

10.2.2 *Our Approach*

As mentioned before, our approach called ReStore to tackle this problem consists of the two steps depicted in Figure 10.1: First, a user has

to (once) annotate a database schema before we train neural completion models that can be used to generate the missing data required to execute aggregate queries over the completed database.

SCHEMA ANNOTATION. In the annotation step, a user must indicate for a given incomplete database which tables are complete and which ones are incomplete. An example for an annotated schema is depicted in Figure 10.1a which consists of three tables of a housing database where two tables are marked as complete (`landlord` and `neighborhood`) and one table (`apartment`) is marked as incomplete.

In addition, information about the relationships between tables needs to be annotated. Here, the user has to provide information whether there are any *complete* foreign-key relationships between tuples from a complete table and an incomplete table. For example, in Figure 10.1a all apartments of neighborhoods in NYC are available but not those for CA. In many of the application scenarios, the information which relationships are complete is known a priori and thus does not cause additional manual annotation overhead. For example, often a complete subset of data (e.g., apartments of a certain state) is available.

Based on the annotation, so called tuple factors (TF) [72] can now be automatically computed step to capture information about the relationships across complete and incomplete tables as shown in in Figure 10.1a (e.g., how many apartments a complete neighborhood has). Based on the available data and the computed tuple factors, we then learn our completion models as discussed next.

The user might also have other additional information, which can help to further enhance the quality of the synthesized data. Among these are table sizes for incomplete tables or aggregate statistics (e.g., average rental prices in certain states). Using techniques like iterative proportional fitting [131], this information can be used to improve our generated data. These techniques are orthogonal to our approach and we thus exclude them in the remainder.

MODEL TRAINING AND DATA COMPLETION. Given an annotated schema, we can now learn the completion models. As depicted in Figures 10.1b, two completion models have been learned that can either take data from the complete `neighborhood` table or the complete `landlord` table to synthesize missing `apartment` tuples. By taking complete tables as evidence our models synthesize missing tuples even if there is a bias in the missing data since we capture correlations across tables (e.g., which types of apartments are expected based on the characteristics of the neighborhoods).

These completion models can now be used at runtime to complete the missing data for a given user query. For instance, if a user wants to know the average rent per state, we first compute the completed join `neighborhood` $\bowtie$ `apartment`. More precisely, we introduce a new operator called *incompleteness join* to join complete and incomplete

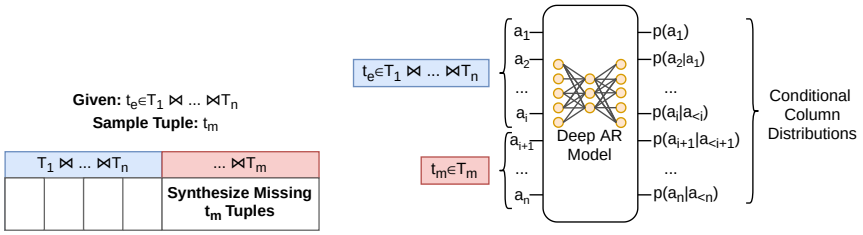
tables that generates the missing tuples needed to make the join complete. In our example, the incompleteness join would generate apartments for neighborhoods where the data is missing using the appropriate completion model of Figure 10.1b. Once the missing tuples for the join are generated (i.e., the incompleteness join produced its output), we can compute the aggregated result using a normal aggregation operator.

We decided to complete data on a per-query basis at runtime since completing the full database might be too expensive (and actually not needed) for large datasets. However, it is important to note that the models are not query-dependent and only have to be learned once for an incomplete schema and can be reused across queries. Moreover, the generated data can still be materialized or even generated a priori as we will discuss in Section 10.4.

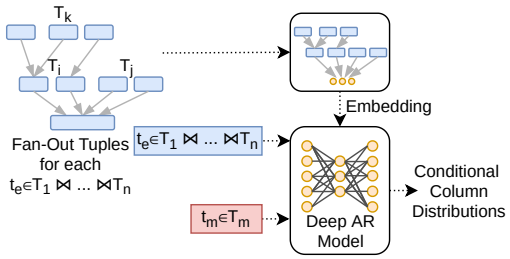
SUPPORTED SCHEMA AND QUERIES. In general, our approach supports any relational schema where tables are connected via foreign-key relationships. For the workload, we currently limit ourselves to acyclic Select-Project-Aggregate-Join (SPJA) queries where joins are equi-joins along foreign-key relationships which are typical queries for decision making. An important aspect is that we can support arbitrary filter predicates or aggregate functions as well as any number of group-by attributes. The reason is that once data is completed for a join, we use normal query operators (e.g., filter or aggregate operators) to compute the query results. Supporting other types of queries, however, is indeed possible. For example, other join types (e.g., non-equi joins) could be added by deriving tuple factors that represent these join conditions.

10.2.3 *Discussion*

The central assumption of our approach is that both missing and available tuples have consistent correlations; i.e., while there can be a bias in the available tuples, it is required that the missing tuples have the same correlations between attributes as the remaining tuples. This is not a requirement specifically for ReStore but for any system that uses machine learning to complete a dataset since otherwise the available tuples cannot be used as evidence to predict the missing tuples. More technically, the conditional distributions of missing tuples t_m given an evidence tuple t_e should be equivalent for remaining and missing tuple distributions, i.e., $P_m(t_m \mid t_e) \approx P_r(t_m \mid t_e)$. If this assumption holds, the main factor determining how accurately the original query result can be restored is the *predictability* of the query attributes as we will later show in our experimental evaluation. If the attributes are not predictable given the evidence tuples, our models will complete the data with lower confidence (cf. Section 10.6).



(a) Completion Task. (b) Simple Completion Model.



(c) Schema-Structured Completion Model.

Figure 10.2: Learned Completion Models in ReStore. (a) The goal is to complete a table T_m using the join $T_1 \bowtie \ldots \bowtie T_n$ of complete tables $T_1, \ldots, T_n$ as evidence. (b) Simple completion models are based on autoregressive models and learn conditional distributions $P(a_i|a_{<i})$ over all attributes in $T_1 \bowtie \ldots \bowtie T_n \bowtie T_m$ (including the incomplete table T_m). After learning, we can use conditional sampling to synthesize missing tuples t_m given an evidence tuple $t_e \in T_1 \bowtie \ldots \bowtie T_n$. (c) Schema−structured models incorporate additional (so called fan-out) evidence of a tuple t_e using tree embeddings.

10.3 LEARNED COMPLETION MODELS

A natural fit for the completion task of ReStore are so called deep autoregressive (AR) models [51, 125, 133]. In the following, we first discuss the relevant background on AR models and then present a first class of simple completion models based on AR models. Afterwards, we present schema-structured autoregressive (SSAR) models which are more expressive than the simple completion models since they can capture the structural information in complex relational schemas that can be used as evidence for generating missing tuples.

10.3.1 *Background on Autoregressive Models*

Autoregressive models learn a probability distribution by approximating the density of observed variables $p(x_1, \ldots, x_n)$. These models exploit that any density can be decomposed into a product of conditional densities $p(x) = \prod_{i=1}^{n} p(x_i \mid x_{<i})$. The factors express the conditional density of the i-th variable given its predecessors.

The popular MADE [51] models realize an autoregressive architecture using deep learning techniques. The network obtains a vector $(x_1, \ldots, x_n)$ as input and is trained to output the conditional densities $(p(x_1), p(x_2|x_1), \ldots, p(x_n|x_{<n}))$. It is ensured that the i-th output $p(x_i \mid x_{<i})$ only depends on inputs with an index $< i$ using masked layers that prevent the flow of information from subsequent inputs.

Conditional sampling (and hence generating new data) can now easily be implemented using iterative forward sampling. Assume that we are given a partial vector $(x_1, \ldots, x_i)$ and want to sample the remaining entries $(x_{i+1}, \ldots, x_n)$ of the vector, i.e., sample from the conditional distribution $p(x_{\geq i}|x_{<i})$. By making use of the autoregressive model, we can first predict the distribution $p(x_{i+1}|x_{\leq i})$ and sample the next variable x_{i+1}. We can now repeat the procedure by feeding the vector $(x_1, \ldots, x_i, x_{i+1})$ into the network to predict $p(x_{i+2}|x_{\leq i+1})$ and so forth until we have finally computed a conditional sample for all missing variables of the input vector.

10.3.2 *Simple Completion Models*

As a first contribution, we present a simple class of completion models based on AR models. The general idea of these models is to use tuples of a complete table $t_e \in T_1$ (or of a join of complete tables $t_e \in T_1 \bowtie \ldots \bowtie T_n$) as *evidence* to synthesize a missing tuple of one incomplete table T_m. In other words, the completion models take a tuple t_e as input and synthesize a missing tuple t_m for the incomplete table T_m.

To capture distributions and correlations present in the dataset and eventually generate the missing T_m tuples, a completion model for one

incomplete table T_m is learned over the join of $T_1 \bowtie \ldots \bowtie T_n \bowtie T_m$ (based on the available data). Clearly, for complex schemata with potentially multiple incomplete tables we need to learn multiple completion models. An efficient learning procedure for complex schemata is presented at the end of this Section. In the following, we focus on the question how a single completion model for one incomplete table is derived. We first consider the case of using a single complete table as evidence to generate tuples of an incomplete table and later show how joins of tables can be used as evidence.

SINGLE EVIDENCE TABLE. Let us first consider the case of a single complete table T_1 which is connected via a foreign-key relationship to the incomplete table T_m. Our goal is to synthesize the missing tuples in T_m. To this end, a deep AR model is trained over the join $T_1 \bowtie T_m$ (more precisely, all join attributes $a_1, \ldots, a_n$ including tuple factors as depicted in Figure 10.2b) using the available data. Afterwards, for every tuple t_1 of the complete table T_1 we can synthesize an appropriate tuple for the incomplete table T_m by sampling from the conditional distribution $t_m \sim P(t_m|t_1)$. For instance, in our example in Figure 10.1 we can synthesize an `apartment` tuple, given a `neighborhood` tuple.

Intuitively, a given `neighborhood` tuple tells us what a typical apartment in that neighborhood looks like. Moreover, we can synthesize tuple factors for a given `neighborhood` tuple if it is not already available. This tells us in addition how many apartments a neighborhood (given its characteristics) has. Using the tuple factor, we can now synthesize as many tuples as are missing; e.g., for a neighborhood that should have three apartments but the dataset contains only one, two new `apartment` tuples need to be synthesized. This also allows to debias a dataset. For instance, the model might predict more missing tuples for neighborhoods in areas with higher population density and since population density and rental prices could be correlated, it will synthesize more expensive apartments resulting in an overall higher average rent. More details on how more complex queries can be handled an debiased is given in Section 10.4. For now, we simply focus on the data generation process for generating one missing tuple t_m from a given evidence tuple t_e.

ADDITIONAL EVIDENCE TABLES. Instead of using only a tuple of table T_1 as evidence, we can also use information from additional tables $T_2, \ldots, T_n$ as evidence. The condition for more than one complete table to be used as evidence is that they are connected via foreign-key relationships directly or indirectly to T_1. This is necessary because otherwise it is not clear which tuples of complete tables should be combined to generate a tuple in the incomplete table.

For instance, in Figure 10.1, we cannot use the `landlord` and the `neighborhood` tables as evidence in one model to synthesize an `apartment`

tuple. While this is technically possible, we do not know a priori in which neighborhoods a landlord has apartments. Trying out all possible combinations is computationally infeasible. Hence, in this particular case, we have to decide whether to use a completion model that uses neighborhoods as evidence or one that uses landlords as input for the completion. This decision is discussed in Section 10.5 where we present an algorithm for automatically selecting which data to use as evidence. However, as mentioned before, we can still use additional evidence tables $T_2, \ldots, T_n$ as long as they are connected via foreign-key relationships to T_1 (which itself is connected to the incomplete table T_m). The idea is that we can use a tuple from the join $t_e \in T_1 \bowtie \ldots \bowtie T_n$ as evidence to generate a tuple for T_m by sampling from $P(t_m|t_e)$. For instance, if the state information of a neighborhood would be represented in a separate table that was connected to the `neighborhood` table via a foreign-key reference, we could use the joined tuple $t_s \bowtie t_n$ of the `neighborhood` and `state` tables as input to more accurately predict a missing `apartment` tuple t_a. The attributes of the state table in this case serve as additional features for the deep AR model.

FAN-OUT EVIDENCE. However, even in the case that all additional evidence tables $T_2, \ldots, T_n$ are connected to T_1 there are limits to which evidence tables can be used. In case one of the tables in $T_2, \ldots, T_n$ introduces a fan-out (i.e., the evidence tuple t_1 is connected to more than one tuple directly or indirectly in the additional table) the table cannot be used as additional evidence. We call this *fan-out evidence*. The reason is that if a tuple t_1 in T_1 has several matching tuples (say in T_2), it is not clear which of these tuples should be provided as additional input for the AR model to synthesize a tuple $t_m \in T_m$. For instance, if we had an additional `school` table in our example which is connected to the `neighborhood` table, one tuple could have multiple `school` tuples. To address this issue, we introduce Schema-Structured Completion Models.

10.3.3 *Schema-Structured Completion Models*

As mentioned before, simple AR completion models cannot leverage evidence of an additional complete table if it introduces a fan-out. This motivates Schema-Structured Autoregressive (SSAR) models which are capable of incorporating this information in the completion process.

SUPPORTING FAN-OUT EVIDENCE. Similar to AR models, SSAR models are learned over the join of evidence tables $T_1 \bowtie \ldots \bowtie T_n$ (which do not introduce any fan-out evidence) and the incomplete table T_m as shown in Figure 10.2c. In order to take the additional tables which introduce a fan-out evidence into account, we perform

an acyclic walk on the schema graph. That means for a given evidence tuple $t_e \in T_1 \bowtie \ldots \bowtie T_n$, for which we want to generate the missing tuple t_m, we first additionally join tuples from fan-out tables (e.g., T_i and T_j in Figure 10.2c). This can be done recursively for tables which have an additional fan-out relationship to tables that are not directly connected to t_e (such as T_k in Figure 10.2c). This results in a tree structure of tuples representing the fan-out evidence, which is then encoded and fed into the neural network in addition to the evidence tuple t_e to predict appropriate tuples t_m of the incomplete table T_m. For instance, for a given `neighborhood` tuple t_n we would feed the tree with t_n as root and all schools in this neighborhood as children into the model. To use this tree structure as input to our SSAR models, we encode the tree using a tree embedding architecture. In particular, we use sum-pooling for the child embeddings which are fed into an additional feed-forward network. This architecture was shown to be a universal function approximator for permutation invariant functions [190]. We additionally use weight sharing for tuples of the same table to reduce the number of parameters.

SELF-EVIDENT DATA COMPLETION. In addition to using tree-structured models to incorporate evidence from additional fan-out tables, we can use tree models also for incorporating the already available data of the incomplete table itself. For instance, let us again consider the case that the `apartment` table is incomplete and we wish to complete the join of `neighborhood` $\bowtie$ `apartment` using a complete `neighborhood` table. Given a `neighborhood` tuple, the SSAR model has to predict an appropriate missing `apartment` tuple. As mentioned before, some apartments of a given neighborhood might already be available (but not all). Using tree embeddings, these `apartment` tuples could also be fed into the SSAR model as additional (self-)evidence. The intuition is that, if there are typical constellations of apartments in a neighborhood (e.g., typically they have comparable prices), this will be learned by the SSAR model and taken into account during the completion further refining the synthesized data.

10.3.4 *Learning on Complex Schemata*

So far, we have focused on the question how one individual completion model works. However, given an annotated schema of a complex database, we have to learn multiple models to potentially synthesize the data for arbitrary joins containing incomplete tables. More precisely, unless otherwise specified by the user, we want to be able generate tuples for any table T_x using any connected table T_y as evidence. Naively, we would have to learn a single SSAR (or AR) model for every every pair of tables T_x, T_y that are connected via a foreign-key to complete T_x using T_y and potentially all other (non fan-out

and fan-out) tables connected to T_y that can be used as additional evidence. However, this would lead to a high number of models and consequently high training times. Instead, as we show next models can be merged (before learning them) to reduce the number of models and overall training time significantly.

MERGING EXAMPLE. For instance, if we want to complete T_2 using T_3 and T_1 using $T_2 \bowtie T_3$ both completions can be done using the same model. We only have to make sure that attributes from T_3 are first and that the ones of T_2 and T_1 are second and third, respectively. This is possible since the model provides both $p(T_1|T_2, T_3)$ and $p(T_2|T_3)$. However, because AR models require a fixed ordering of variables, merging is not always possible. For instance, a model that has to learn $p(T_2|T_1)$ cannot be merged because we cannot find an ordering of variables that allows to predict both $p(T_2|T_1)$ and $p(T_1|T_2, T_3)$.

MODEL MERGING. In our approach, we first require for two models M_1 and M_2 to be merged that the set of tables of M_1 is a subset of the tables of M_2 or vice versa. In addition, we have to check whether there exists a consistent variable ordering. To this end, we construct a directed graph that contains a node for every involved table. For every table that should be completed, we add an arc from every evidence table to this table. Only if the resulting graph is cycle-free a valid ordering of tables can be derived and we merge the models. In particular, we use the topological sorting as ordering. We merge models until no more non-conflicting merges are available.

10.4 QUERY-DRIVEN DATA COMPLETION

In this Section, we show how the completion models (AR and SSAR) can be used to complete data for a given user query that might contain joins over complete and incomplete tables.

10.4.1 *Overview of Query Processing*

Data completion using ReStore happens on a per-query basis at runtime during query processing. We decided to do the completion on a per-query basis because an offline completion of the full database especially for larger databases is costly and might actually not be required. As queries, we support SPJA-queries such as the one shown in Figure 10.1c) that are typical for OLAP with acyclic equi-joins along foreign-keys and arbitrary filters and aggregations (with and without group-by).

In order to answer such a query, we first compute the join $J = T_{u1} \bowtie \ldots \bowtie T_{un}$ over all tables (complete and incomplete) contained in the user query. During the join computation, we complete the join using our completion models such that J contains all data as if the

join would be executed on a complete database. Afterwards, we then apply filter predicates, aggregations and groupings to answer the user query over the completed join.

For efficiency, we push down filter predicates and generate only missing data for the requested subset of tuples in the join of the query. However, for simplicity of explanation, we assume in the following that filters are executed after the join. Moreover, as we describe Section 10.4.5, this also enables optimizations to reuse the generated data for subsequent queries.

10.4.2 *Single Incomplete Table in a Query*

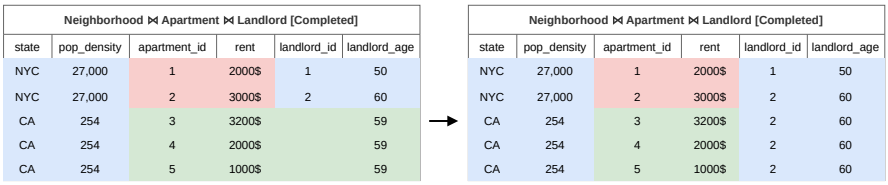
We now first consider the case where a single table T_m in the join $T_{u1} \bowtie \ldots \bowtie T_{un}$ of the user query is incomplete (as it is the case in Figure 10.1c) and discuss the case where multiple tables in the user query are incomplete later. In principle, different models could be available to synthesize data for the incomplete table T_m. We now discuss how a completion works if a model M is already selected and discuss in Section 10.5 how to select a completion model.

Moreover, we initially assume that the tables that are used as evidence for generating missing data for T_m are among the complete tables in the join $T_{u1} \bowtie \ldots \bowtie T_{un-1} \bowtie T_{un}$. To differentiate in the sequel between the tables $T_1 \bowtie \ldots \bowtie T_n$ needed as evidence for the completion model M and the user join $T_{u1} \bowtie \ldots \bowtie T_{un}$, we use the terms *completion path* and *query path*, respectively.

In the following, without loss of generality, we assume that $T_m = T_{un}$ is the incomplete table and T_m is connected to the complete (evidence) table T_{u1} via a foreign key or vice versa. The step of extending a join of complete tables $T_{u1} \bowtie \ldots \bowtie T_{un-1}$ with an incomplete table T_{un} to $T_{u1} \bowtie \ldots \bowtie T_{un-1} \bowtie T_{un}$ while generating the missing tuples is called *incompleteness join*.

COMPLETION PATH EQUALS QUERY PATH. The simplest case for an incompleteness join is where the query path is equal to the completion path. Imagine, there is one more `state` table in our example of Figure 10.1 which has a reference to the `neighborhood` table and a user requests a join of the complete `state` and `neighborhood` tables with the incomplete `apartment` table. In this case, we could use a completion model that uses states and neighborhoods as evidence to synthesize apartments, the query path and completion path would be both `state` $\bowtie$ `neighborhood` $\bowtie$ `apartment`.

For executing an incompleteness join in this case, we first join the complete evidence tables $T_e = T_{u1} \bowtie \ldots \bowtie T_{un-1}$ (`state` and `neighborhood` in our example). Afterwards, we iterate over all evidence tuples $t_e \in T_e$ and synthesize the missing data for the user join. In case of SSAR models additional fan-out evidence tables need to be



Neighborhood ⋈ Apartment ⋈ Landlord [Completed]

state	pop_density	apartment_id	rent	landlord_id	landlord_age
NYC	27,000	1	2000$	1	50
NYC	27,000	2	3000$	2	60
CA	254	3	3200$		59
CA	254	4	2000$		59
CA	254	5	1000$		59

Neighborhood ⋈ Apartment ⋈ Landlord [Completed]

state	pop_density	apartment_id	rent	landlord_id	landlord_age
NYC	27,000	1	2000$	1	50
NYC	27,000	2	3000$	2	60
CA	254	3	3200$	2	60
CA	254	4	2000$	2	60
CA	254	5	1000$	2	60

Figure 10.3: Nearest Neighbor Replacement. Foreign-keys are not synthesized for the `apartment` table and thus the tuples cannot be joined with the complete `landlord` table. Hence, `landlord` tuples are first synthesized and afterwards replaced with "similar" `landlord` tuples

joined separately to construct the query tree for each evidence tuple t_e which is fed into the SSAR model. For generating the missing data using the completion model, we have to differentiate whether the relationship of t_e and tuples of the incomplete table T_m is a `1:n` or `n:1` relationship (i.e., if one evidence tuple t_e has multiple join partners or one join partner in the incomplete table).

In case of a `1:n` relationship, we first have to determine how many t_m tuples have to be generated per t_e tuple which can be estimated using the tuple factors which are learned by the corresponding AR or SSAR completion model. Moreover, we have to determine how many t_m-tuples already exist (since some might already be available but not all) and synthesize only the missing number of tuples. This can be done efficiently during joining by first creating a hash-map on the incomplete table (which is needed for joining anyway) that additionally counts the occurrences of tuples with the same foreign-key in the T_m table. For instance, if we want to synthesize apartments given the join `neighborhood` ⋈ `states`, we first have to predict how many apartments we expect to see per neighborhood, i.e., the tuple factor per neighborhood. Afterwards, we synthesize the appropriate number of apartments using the join of `state` and `neighborhood` as evidence. For the output of the incompleteness join, we then join t_e with all the existing and synthesized tuples.

In case of a `n:1` relationship, we can disregard tuple factors and only need to generate one missing tuple t_m per evidence tuple t_e if needed. For example, for a join of the `landlord` table with the incomplete `apartment` table, we synthesize a landlord only for apartments where the `landlord` tuple is missing.

COMPLETION PATH CONTAINED IN QUERY PATH. We now consider the case that the completion path is contained in the query path (i.e., the query path contains more tables than the completion path).

In this case, we use a similar approach as before and use the completion path tables as evidence for the model to generate the missing tuples in T_m but then need to join the remaining complete tables of the user query (not in the completion path). For instance, assume a user wants to join all three tables in the example in Fig-

ure 10.1 (neighborhood, apartment, and landlord). To process such as query, we could use a completion model which allows us to generate apartment tuples from neighborhood tuples to produce a "completed" join for those two tables. Afterwards, we then need to join this output with the complete landlord table. However, our completion models do not generate foreign keys (to the landlords) for the synthesized apartment tuples since AR and SSAR models are not suited for generating such type of information.

Hence, we cannot use a normal join operator for joining the output of an incompleteness join with the next complete table (e.g., with landlord in our example) but have to process this join in a different manner. In this case, we again use a completion model that allows us to generate a new landlord tuple using apartments and a neighborhood as evidence as depicted in Figure 10.3 (left). Since the landlord table, however, is a complete table we then replace the synthesized tuple with an existing tuple that has the highest similarity (i.e., lowest euclidean distance) with the synthesized tuple. For instance in Figure 10.3, the last three synthesized landlord tuples are replaced with the second landlord from the complete table since they are very similar.

However, an *exact* nearest neighbor replacement of the generated landlord tuple would come at a high cost of computing the pairwise distances of all synthesized tuples and tuples of the complete table during query processing. Hence, we employ approximate nearest neighbor approaches and batching for the replacement. This is crucial to achieve a competitive performance. In general, this join procedure has to be used if foreign keys in an intermediate result are missing but required for a join with a complete table. Otherwise, normal joins can be used. Although the synthesized data is of high-quality the replacement is required to fully comply with the user annotations - it is unexpected to see new synthesized tuples for complete tables.

10.4.3 *Multiple Incomplete Tables in a Query*

We have now discussed all techniques required to complete a user query where the query path includes only a single incomplete table. The case of several incomplete tables can now easily be derived. In particular, we again assume that the completion path is given and repeatedly apply incompleteness joins as before and use the nearest neighbor replacement where appropriate. The order which table to complete first is determined using the techniques in Section 10.5 to automatically select the best completion model.

There is only one difference compared to the single incomplete table case since we have to apply the nearest neighbor replacement also for incomplete tables. In particular, if we synthesize tuples for an incomplete table, we might still synthesize too many tuples since foreign-keys of previous tables might not be generated and thus even

Algorithm 3 Single Table Completion

Input: Requested Join Tables $\mathbf{J}_{req} = T_{u1}, \ldots, T_{un}$
Input: $T_1, \ldots, T_n$ (Path from complete Table T_1 to $\mathbf{J}_{req}$)
Output: Approximated Complete Join $T_{u1} \bowtie \ldots \bowtie T_{un}$

$\mathbf{J} \leftarrow T_1$
for T_i **in** $T_1, \ldots, T_{n-1}$ **do**
  // Incompleteness Join
  $\mathbf{J}_{incomplete} \leftarrow J \bowtie T_i$
  if $T_i \bowtie T_{i+1}$ is Fan-Out **then**
    Predict Tuple Factor $\mathcal{F}_{T_{i+1} \leftarrow T_i}$ for every $t \in \mathbf{J}$
    $\mathcal{F}_{T_{i+1} \leftarrow T_i} \leftarrow \mathcal{F}_{T_{i+1} \leftarrow T_i}$ - Current No of Join Partners in T_{i+1}
    $\mathbf{J}_{syn} \leftarrow$ Duplicate each $t \in J$ $\mathcal{F}_{T_{i+1} \leftarrow T_i}$ times
  else
    $\mathbf{J}_{syn}$ Tuples in $\mathbf{J}$ without Join Partner in T_{i+1}

  // AR or SSAR Tuple Synthesis
  $\mathbf{M} \leftarrow$ Completion Model for $T_i \rightarrow T_{i+1}$
  $\mathbf{J}_{syn} \leftarrow$ Synthesize Columns of T_{n+1} in $\mathbf{J}_{syn}$ using $\mathbf{M}$

  // Euclidean Replacement
  if Last Join or Next Join Fan-Out **then**
    $\mathbf{J}_{syn} \leftarrow$ `euclidean_replace`($\mathbf{J}_{syn}$, T_{i+1})
  $\mathbf{J} \leftarrow \mathbf{J}_{syn} \cup \mathbf{J}_{incomplete}$
return $\mathbf{J}$

though the tuples are still in the database, they would still not appear in the resulting join. Hence, we have to estimate how often a tuple of the incomplete table should appear in the full join and complete accordingly.

The pseudocode for the general case which summarizes the discussions in Sections 10.4.2 and 10.4.3 is shown in Algorithm 3.

10.4.4 *Additional Cases for Data Completion*

COMPLETION WITH ADDITIONAL TABLES. We have now considered the case of incomplete tables in a user query under the condition that the completion path is a subset of the requested query path. However, this is not necessarily the case since the completion path can also contain additional tables: for instance, if the user queries the `landlord` and the `apartment` table but for the completion of the `apartment` table the lower model in Figure 10.1b is chosen which uses neighborhoods as evidence. The high-level idea for query processing in such a case is that we first use the join over all tables in the completion path to synthesize the missing data for the incomplete table (e.g., the `apartment` table is completed using the `neighborhood` table) and afterwards potentially have to reweight tuples according to the introduced fan-out similar to [72].

MULTI-PATH COMPLETION. Another interesting case is that using only a single path for the completion of one incomplete table can be insufficient. For instance, let us consider a slightly modified schema of a complete `apartment` table, an incomplete `neighborhood` table and an

additional complete `school` table which has a foreign-key relationship to the neighborhoods. If a user now simply queries the `neighborhood` table and we complete the neighborhoods via the `school` table, neighborhoods that do not have any schools will be missing (since we never generate them if we use a completion path from `school` to `neighborhood`). In these cases, we use all paths to synthesize data and combine data based on tuple factors.

10.4.5 *Further Optimizations*

While our data completion process synthesizes data at query runtime, data which is synthesized for one query can be reused for related queries. This allows for (i) caching of data synthesized at runtime or (ii) an offline completion independently of the workload.

We first discuss how data for completed joins can be reused. In particular, since aggregations and filters are applied after completing a join to approximate a query Q in ReStore, the completed data of Q can be reused for a query Q' if they use the same join path J. Moreover, if a query Q' requires additional tables not covered in J, we can start from J and generate additional data incrementally for further incomplete tables. Finally, if Q' only requires a sub-path of J, we can reuse the data by projecting J to the tables required by Q.

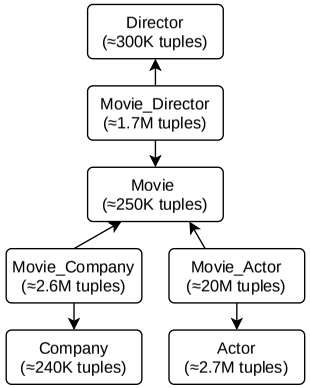
Second, as mentioned before we can also generate missing data prior to the query runtime. One way is to predict which queries will occur at runtime and thus optimize which incompleteness joins to create. However, if there is no knowledge about potential queries, simple heuristics-driven strategies can be used. In particular, we can create data for every pair of a joinable incomplete and complete table. This would allow us to answer any query on a single incomplete table or a join of a complete and incomplete table without the need to generate data.

10.5 MODEL AND PATH SELECTION

In the approach discussed so far there are some degrees of freedom. In particular, whether we should rather learn AR or SSAR models and which complete tables (i.e., which completion path) should be used for the completion. Both decisions can have a significant impact on the quality of the completion. Intuitively, while the first aspect determines whether we learn a model that is fitting the data well, the second aspect is important because different completion tables have a varying significance for the join we want to complete.

BASIC SELECTION. To decide whether a model should be used for completion of an incomplete table (or not), it is important to check the accuracy (i.e., test loss) of the models prior to using the model for completion. If the accuracy is too low this means that the true attribute

(a) Housing Schema.



(b) Movie Schema.

Setup	Biased Attribute	Tuple Factor Keep Rate	Keep Rates `landlord`	`apartment`	`neighborhood`	
H_1	`apartment.price`	30%	100%	20-80%	100%	
H_2	`apartment.room_type`	30%	100%	20-80%	100%	
H_3	`apartment.property_type`	30%	100%	20-80%	100%	
H_4	`landlord.landlord_since`	30%	20-80%	100%	100%	
H_5	`landlord.response_rate`	30%	20-80%	100%	100%	
Setup	Biased Attribute		`movie`	`director`	`actor`	`company`
M_1	`movie.production_year`	20%	20-80%	100%	100%	100%
M_2	`movie.genre`	20%	20-80%	100%	100%	100%
M_3	`movie.country`	20%	20-80%	100%	100%	100%
M_4	`director.birth_year`	20%	80%	20-80%	100%	100%
M_5	`company.country_code`	20%	80%	100%	100%	20-80%

(c) Completion Setups.

Figure 10.4: Datasets and Completion Setups.

values can hardly be reconstructed since they are not predictable and the bias is likely not reduced significantly.

ADVANCED SELECTION. For the remaining models we have to estimate the quality of each completion model. To this end, we derive additional incomplete scenarios with the given incomplete dataset as ground truth to assess model and path quality. The underlying assumption is that if the models and paths are able to reconstruct our incomplete dataset they are also able to perform the actual completion with high accuracy.

In practice, the user often suspects a bias in the data but the extent of it is unclear. This information can additionally be provided by the user and used for the model selection. For instance, an incomplete table might cover more high-population neighborhoods and thus the user expects an overestimation of the average rent. As we will show in our experiments, this additional information can significantly improve the quality of the synthesized data.

10.6 COMPLETION CONFIDENCE

It is crucial for practitioners to be aware of the confidence of query results after the data completion. For this, we provide confidence interval estimations of the query results for how certain our models are when synthesizing missing data. In the following, we start with the simple case that involves only a single incomplete table and then explain the more general case.

10.6.1 *Simple Case*

For the simple case, we assume that we have a similar housing database as before but with only two tables: an incomplete apartments table where apartments can have two types (large and small) and a complete neighborhoods table. Furthermore, assume that a user issues a count-query that joins these two tables to compute the frequency of the two apartment types for which we want to compute confidence intervals. Intuitively, if the neighborhood tuples do not provide strong evidence about the types of missing apartments (i.e., if there is a low correlation), the completion models will predict both apartment types with equal probabilities for each missing tuple. In this case, we should have a low confidence and predict wide confidence intervals. In contrast, if the model predicts the apartment type with high certainty, the confidence interval should be more tight.

In order to compute confidence intervals for a query over an incomplete table, we use the following two-step procedure: (1) We first compute the certainty $C(t_e)$ of a prediction for an attribute of a missing tuple given an evidence tuple t_e in ReStore. For this, we compare the

probability distribution of the predicted attribute value P_{model} for one synthesized tuple with the distribution of the attribute values in the training data $P_{incomplete}$. If the model is uncertain when synthesizing an attribute value for one missing tuple t_m, given the evidence tuple t_e, it will simply predict the distribution of values in the training data (i.e., $P_{model} \approx P_{incomplete}$). However, if the model is certain given an evidence tuple, it will predict a particular attribute value (e.g., a large or a small apartment type) with higher probability. Hence, for computing the certainty of a prediction, we compute the similarity of the distribution P_{model} with $P_{incomplete}$ using the KL-divergence and normalize it to $[0, 1]$ by $1 - exp(-D_{KL})$. (2) Second, we compute confidence intervals for each synthesized tuple as follows. For this, we introduce a lower and upper bound distribution (P_{lower} and P_{upper}). In our example, we use a distribution for the upper bound P_{upper} where one particular apartment type (e.g., the small apartments) occurs in 95% of the cases (for a 95% confidence). The upper bound of our confidence intervals can then be computed using $C(t_e)P_{model}(t_e) + (1 - C(t_e))P_{upper}$. For the lower confidence interval, we simply replace P_{upper} by P_{lower} where P_{lower} represents the distribution where apartments only occur in 5%.

10.6.2 *General Case*

The procedure above can be generalized to queries that (1) involve multiple incomplete tables and (2) other aggregate functions. In order to support (1), we generate the missing tuples using the completion models similar to the the normal completion process. However, for every query attribute that has to be synthesized we define an individual distribution P_{lower} and P_{upper} (based on the given confidence) and compute the model confidence intervals as described before. Again, instead of using P_{model} directly, we use $C(t_e)P_{model}(t_e) + (1 - C(t_e))P_{lower}$ for the synthesized attributes when computing the lower bound and similarly P_{upper} for the upper bound. For this process, we assume that attribute values of different tables are correlated to generate conservative (i.e., worst case) confidence bounds. (2) As mentioned before we can also support other aggregate functions. For example, to support average in addition to count aggregates we define P_{lower} and P_{upper} for continuous attributes. Moreover, sum aggregates can be treated as a combination of average and count. Note that we currently only support completion confidence intervals for query attributes used in an aggregation (i.e., count, avg, sum). For other query types, we can resort to per-query statistics that we show a user such as the ratio of synthesized vs. existing tuples.

10.7 EXPERIMENTAL EVALUATION

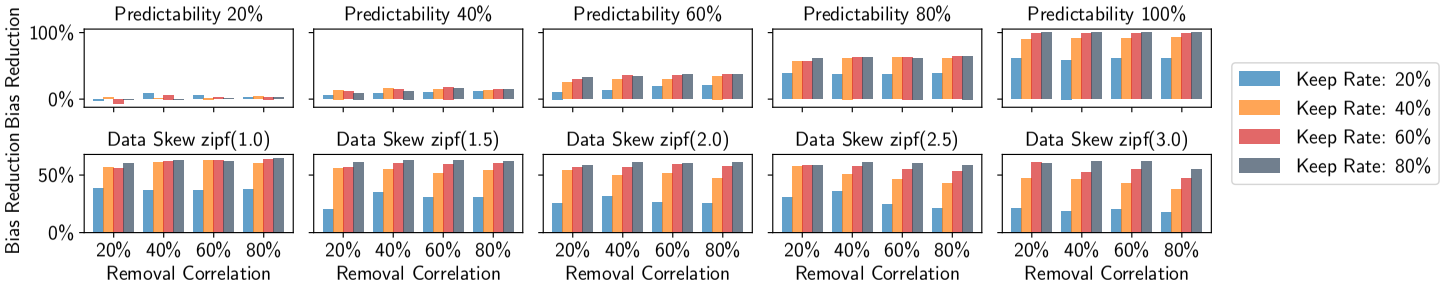
In this Section, we evaluate both the quality of the completed relational datasets as well as several performance aspects of ReStore[1]: *(Exp. 1 & 2) Data Completion:* We first evaluate how well our models can correct incomplete datasets given certain data characteristics. *(Exp. 3) Query Processing:* In addition, we demonstrate the end-to-end accuracy of our approach using aggregate queries on real-world datasets. *(Exp. 4) Accuracy and Performance:* We finally discuss the accuracies of the different models and the model selection as well as the time required for model training and data completion. For further details on the experiments, we plan to publish an extended technical report.

10.7.1 *Datasets and Implementation*

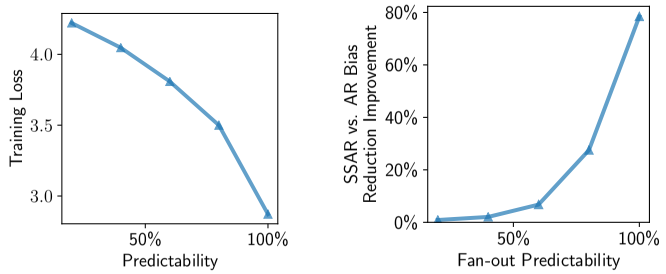
DATASETS. We first evaluate our approach on a synthetic dataset to investigate which factors determine the quality of our completion in isolation. However, restricting ourselves to synthetic datasets is insufficient since they do not exhibit as complex distributions and correlations as real-world datasets. We thus also evaluate our approach on two real-world relational datasets with different complexity. The first schema is a housing dataset derived from the Airbnb data[2] which we normalized to obtain different relations for landlords, neighborhoods and apartments (Figure 10.4a). The movies schema is derived from the popular IMDB[3] dataset but with two important differentiations. We first merged the `movie_info` table information into the `movie` table to obtain more interesting attributes, i.e., genre and rating. Moreover, we explicitly divided the `person` relation into actors and directors exhibiting a more interesting relational structure as depicted in Figure 10.4b. For both datasets we create incomplete versions by removing a varying ratio of tuples to simulate different degrees of incompleteness. Details on how we removed data will be given in our experiments.

IMPLEMENTATION. All models were implemented with PyTorch [137]. For the AR models, we used the model in [186] as a starting point.[4] Similar to [186], we use learned embeddings to represent attribute values in the AR and SSAR completion models. In particular, we use the MADE [51] architecture for AR models with residual connections and ReLU activation functions. For the neural tree architectures in the SSAR models we use a deep sets architecture [190].

1 Code is available online: https://github.com/DataManagementLab/restore
2 https://public.opendatasoft.com/explore/dataset/airbnb-listings
3 http://homepages.cwi.nl/~boncz/job/imdb.tgz
4 https://github.com/naru-project/naru



(a) Bias Reductions.



(b) Training Loss. (c) Fan-out Predictability.

Figure 10.5: Bias Reductions for the Synthetic Datasets. (a) Predictability affects reconstructability. Skewness has no effect on data completion. (b) The test loss is an effective criterion for model selection as discussed in Section 10.5. (c) SSAR models are superior over AR models since they can capture fan-out evidence (called fan-out predictability in the Figure).

10.7.2 *Exp. 1: Data Completion on Synthetic Data*

In this experiment, we first study the factors that determine the quality of our completions. For this, we generate different synthetic datasets where we vary different data characteristics that might influence how well the data is reconstructable. As an additional sanity check, we want to investigate if our automatic model and path selection strategies are able to identify cases that prevent the data completion. We first introduce the metrics and setup before we discuss our results on synthetic data.

COMPLETION SETUPS. For this experiment we use a simple synthetic dataset with only two tables: a complete table T_A with a single attribute A and an incomplete table T_B with a single attribute B where T_B has a foreign-key relationship to T_A. As main parameters which might have an influence on how well a dataset is reconstructable we vary the *predictability* (i.e., how well an attribute can be estimated) and the *skew*. In particular, the categorical attribute B is generated such that B can be perfectly predicted given A (i.e., B is functional dependent on A) and we then incrementally add more noise to reduce the predictability. Moreover, the attribute A is generated either using a uniform or skewed distribution where the Zipf factor is varied (for a fixed predictability of 80%). In addition, we not only vary the predictability of B given A but also the *fan-out predictability* (i.e., how well a missing tuple in T_A can be predicted given other T_A tuples).

In order to derive an incomplete dataset from the synthetic dataset, we systematically remove tuples using two parameters: *removal correlation* and *keep rate*. The keep rate determines the percentage of tuples which are not removed from table T_B. In order to introduce a bias, we correlate the probability of a tuple being removed with the value of the attribute B. The corresponding parameter controls the strength of this correlation. In particular, we correlate the removal probability with the appearance of one attribute value of $b \in B$.

METRICS AND BASELINES. We use the metrics defined in Section 10.2.1. For evaluating the quality of data completion (Exp. 1 and Exp 2), we show the *bias reduction* since it is independent of a given workload. For experiments (Exp. 3) which involve a workload, we additionally show the *relative error*. Unless otherwise stated, we report the metrics for an optimal model and path selection. We provide a dedicated analysis of the model and path selection in Exp. 4. Both metrics show how well we can reconstruct the complete (true) dataset compared to using the incomplete dataset. We do not compare to other baselines, since to the best of our knowledge no approach exists that is capable of completing relational datasets across tables.

RESULTS. As we see from Figure 10.5a (upper row) the predictability is the key factor determining the success of the debiasing. Intuitively, a high predictability allows our model to accurately estimate the missing values of the attribute B for the missing tuples. However, in cases where the attribute B cannot accurately be predicted given attribute A, the test loss of the model is also higher as shown in Figure 10.5b. This confirms that checking the model accuracy is an effective criterion for model selection as discussed in Section 10.5. In those cases, no automated approach could successfully debias the dataset. As we will see in the subsequent experiments, while predictability is a prerequisite for an accurate completion we can largely reduce the bias for a wide set of real-world datasets. This is the case since real-world data is often largely correlated which can be exploited when predicting missing tuples.

Moreover, attribute skew as shown in Figure 10.5a (lower row) does not seem to have a large influence on the performance of our approach. The reason is that the model can still accurately predict the value of attribute B as long as there is a sufficient amount of training data. Finally, as we can see in Figure 10.5c SSAR models are superior over AR models since they can capture fan-out evidence. For showing this, we feed the tuples in T_B that share the same tuple in T_A as self-evidence into the SSAR models (which is a type of fan-out evidence as described in Section 10.3.3). As we see, if the coherence within the group of tuples in T_B that share a reference to the same tuple in T_A is higher (which we call fan-out predictability) the bias reduction of SSAR compared to AR models improves.

CONFIDENCE INTERVALS. In addition to bias reduction, we next evaluate the quality of our confidence intervals using synthetic data. Similar as before, we use a setup with two tables: a complete table T_A with a single attribute A and an incomplete table T_B with a single attribute B where T_B has a foreign-key relationship to T_A. Moreover, we vary the predictability as noted in the setup of this experiment. Note that due to a bias, a certain attribute value b of B can appear less/more frequently in the incomplete table compared to the complete table.

We now compute the confidence intervals for a count-query over B that reports how often a particular attribute value b occurs. We have chosen the attribute value b with the highest deviation between incomplete and complete data which is a challenging task for ReStore. Hence, confidence intervals are particularly of interest. In Figure 10.6, we report the fraction of the attribute value b in the true (i.e., original) and the completed database using 95% confidence intervals for the setup described before.

As we can see, the true fraction of the selected attribute value b on the complete dataset is always within the predicted confidence bounds

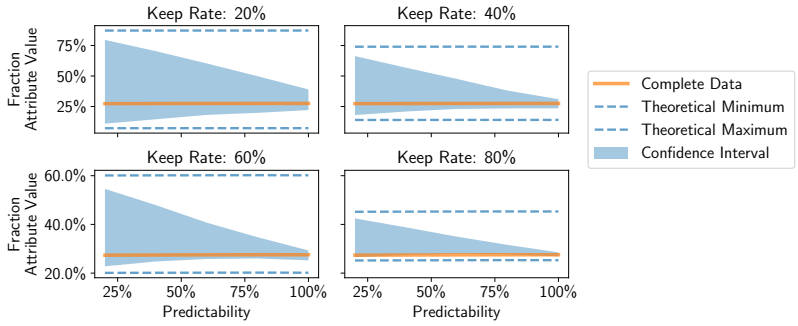


Figure 10.6: Predicted Confidence Intervals on the Synthetic Data for a Removal Correlation of 40%. The bounds always capture the true fraction of the attribute value and an increased predictability results in tighter confidence intervals.

and a larger keep rate results in tighter confidence bounds. Moreover, as expected an increased predictability (x-axis) results in more confident completions and thus tighter confidence bounds. In addition to the predicted confidence bounds, in Figure 10.6 we also plot the theoretical minimum and maximum of the bounds. The theoretical minimum and maximum of the bounds can be computed by replacing all respectively none of the missing values with the given attribute value b. As a sanity check, we see that our confidence bounds also fall into the theoretical bounds.

10.7.3 *Exp. 2: Data Completion on Real Data*

In this experiment, we analyze how well our approach can complete the two real-world datasets. This is more challenging since the underlying schemas are significantly more complex as depicted in Figures 10.4a and 10.4b. Additionally, the data distributions exhibit more interesting correlations.

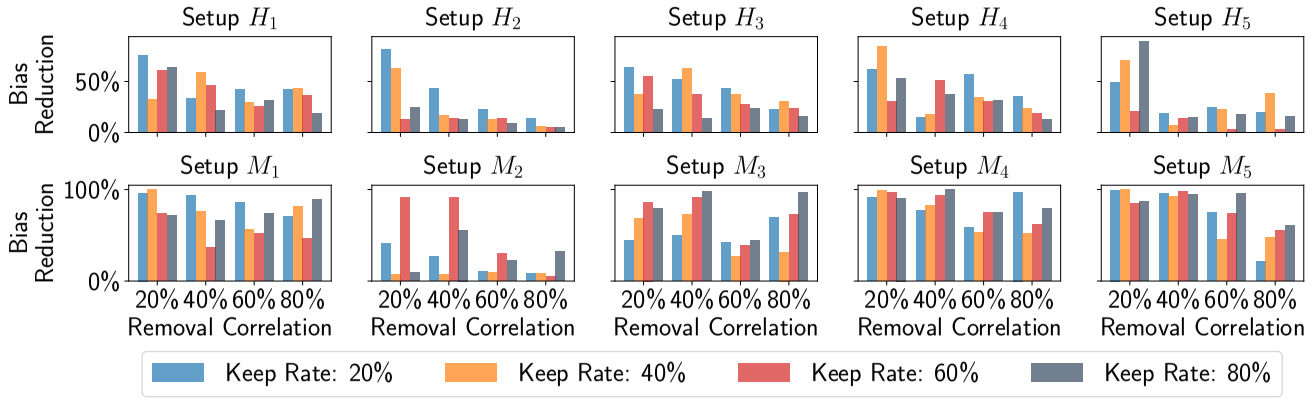
COMPLETION SETUPS. Per dataset we have defined five setups as depicted in Table 10.4c (denoted as H_i and M_i for the housing and movie data, respectively). In each setup, we create an incomplete relational dataset by systematically removing tuples using a particular attribute resembling different data types (categorical and continuous) and data distributions. Similar to the synthetic dataset, we vary the following parameters: *keep rate* and *removal correlation* which are varied from 20% to 80% for all setups. For categorical attributes, we again correlate the removal with the appearance of an attribute value whereas for continuous attributes we correlate it with the normalized attribute value (i.e., to obtain a specific Pearson correlation coefficient). Moreover, we only keep a small share of all tuple factors - 20% for the movie dataset and 30% for the housing dataset to compensate for an overall smaller dataset. In addition, to include some even more challenging setups for the movies dataset we additionally remove all

tuples in the $m:n$ relationship tables (i.e., `movie_company` etc.) which do not have a matching tuple after the removal. For the setups M_4 and M_5 we additionally remove 20% of the `movie` tuples.

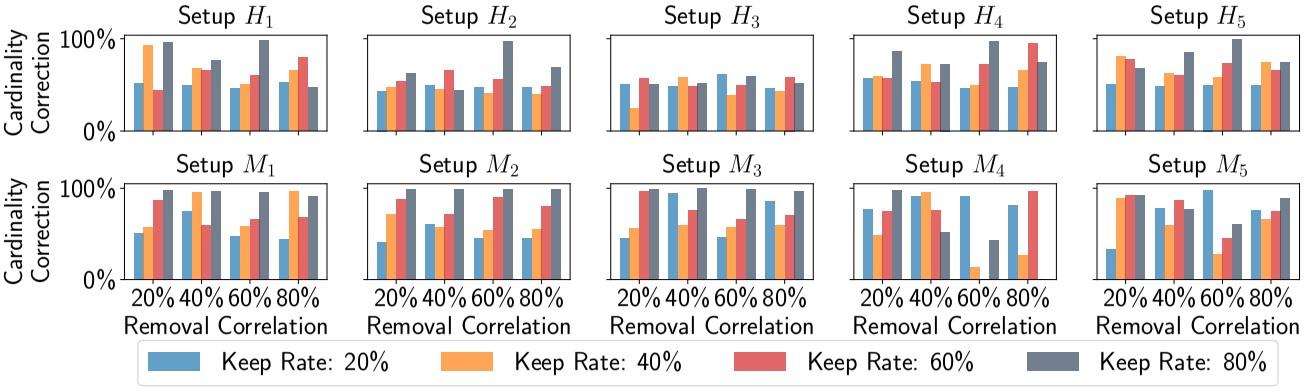
RESULTS. As discussed before, an interesting metric is how well we could debias the incomplete data using our completion models under the different setups. The results are shown Figure 10.7a for all five setups given a variety of keep rates (between 20% and 80%) and removal correlations. As we see, the bias can significantly be reduced for all setups indicating the high quality of our completion models. This especially holds for the setups of the movies dataset where up to 100% of the bias can be removed. In general, a lower removal correlation is beneficial for our approach. The reason is that the lower the correlation, the more examples of high attribute values (for continuous attributes) remain in the training set and thus the model can learn more precisely what leads to those higher values. During the completion it can then predict more accurately whether larger values are likely to occur. The keep rates do not seem to have a significant impact. The reason is that there are two opposing effects. On the one hand, a larger keep rate leads to a larger training dataset and the model can thus learn the distribution more accurately. On the other hand, the absolute error $|AVG_{complete}(X) - AVG_{incomplete}(X)|$ becomes smaller and the model has to predict more extreme values to correct the bias. Consequently, we do not see more accurate completions for larger keep rates.

However, the quality of the completion varies for the different setups. The reason is that the remaining evidence, i.e., the complete tables in the schema are not equally useful. Some attributes of available data are in general less predictable and if those are used for a biased removal, it becomes harder to correct the bias. Interestingly, we do not see the general trend that the completions become less accurate for longer completion paths. Recall that for setups M_4 and M_5, all single completion paths span at least five tables. However, the completions are significantly more accurate than those of M_2. This highlights that the predictability of the biased attribute has the most significant impact on the bias reduction. In general, for setups such as H_2 and M_2 where the evidence of the complete tables does not allow an accurate prediction of the biased attribute, the models cannot correct the bias. This is consistent with our findings on synthetic data.

COUNT CORRECTION. We are also interested in how accurately the table sizes are estimated using different ratios of available tuple factors. Similarly to the bias reduction we define the *cardinality correction* as $1 - \frac{|Completed\ Tuples| - |Complete\ Tuples|}{|Incomplete\ Tuples| - |Complete\ Tuples|}$. As we can see in Figure 10.7b, the cardinalities of the complete tables can relatively accurately be



(a) Bias Reductions.



(b) Cardinality Corrections.

Figure 10.7: Completion Results for Real-World Data using the Setups of the Housing (H_i) and Movies (M_i) Datasets.

predicted even though only $20 - 30\%$ of all tuple factors are kept in the incomplete datasets.

10.7.4 *Exp. 3: Query Processing*

COMPLETION SETUPS. We now investigate the end-to-end performance of our approach for query processing. To this end, we use a workload of both single table and join queries with aggregates and various filter predicates.[5] We then derive incomplete datasets similar to Exp. 1. and compare the relative error of the queries computed on the incomplete dataset and our completed dataset (using the original complete datasets as ground truth). We show the absolute improvement for the relative error for the queries.

RESULTS. As we can see in Figure 10.8, we can achieve significant improvements motivating the use of our approach for practical applications. We can see that `COUNT` and `SUM` queries are in general largely improved while the improvements for the `AVG` queries are smaller. The reason is that for `AVG` queries the improvement depends on the scaling and translation of the attribute as well as the absolute error introduced by the biased removal. This varies largely for the different attributes in our datasets. This emphasizes the importance of the bias reduction metric in the first experiment.

In addition, we noticed that for join queries on the smaller housing dataset and low keep rates the predictions of our models tend to be inferior to the incomplete dataset. In this case, the AR and SSAR models cannot observe sufficient training data to make accurate predictions. We thus recommend not to use our approach if the number of available tuples is very low. However, for larger datasets it is also more time-consuming to complete them manually and in these cases our approach achieves significantly more accurate query results as we can see from Figure 10.8.

10.7.5 *Exp. 4: Accuracy and Performance Aspects*

MODEL AND PATH SELECTION. We next investigate how reliable our model and path selection works. To this end, we plot all bias reductions and the performance of the model selection strategies in Figure 10.9. If we provide the information which bias is suspected in the data (red dots in Figure 10.9), we often pick the optimal path and model. However, even if this information is not available (orange dots in Figure 10.9), we select models that can effectively reduce bias.

TRAINING TIME. In Figure 10.10 we depict the average training time of the AR and SSAR models for the different completion setups.

5 https://github.com/DataManagementLab/restore

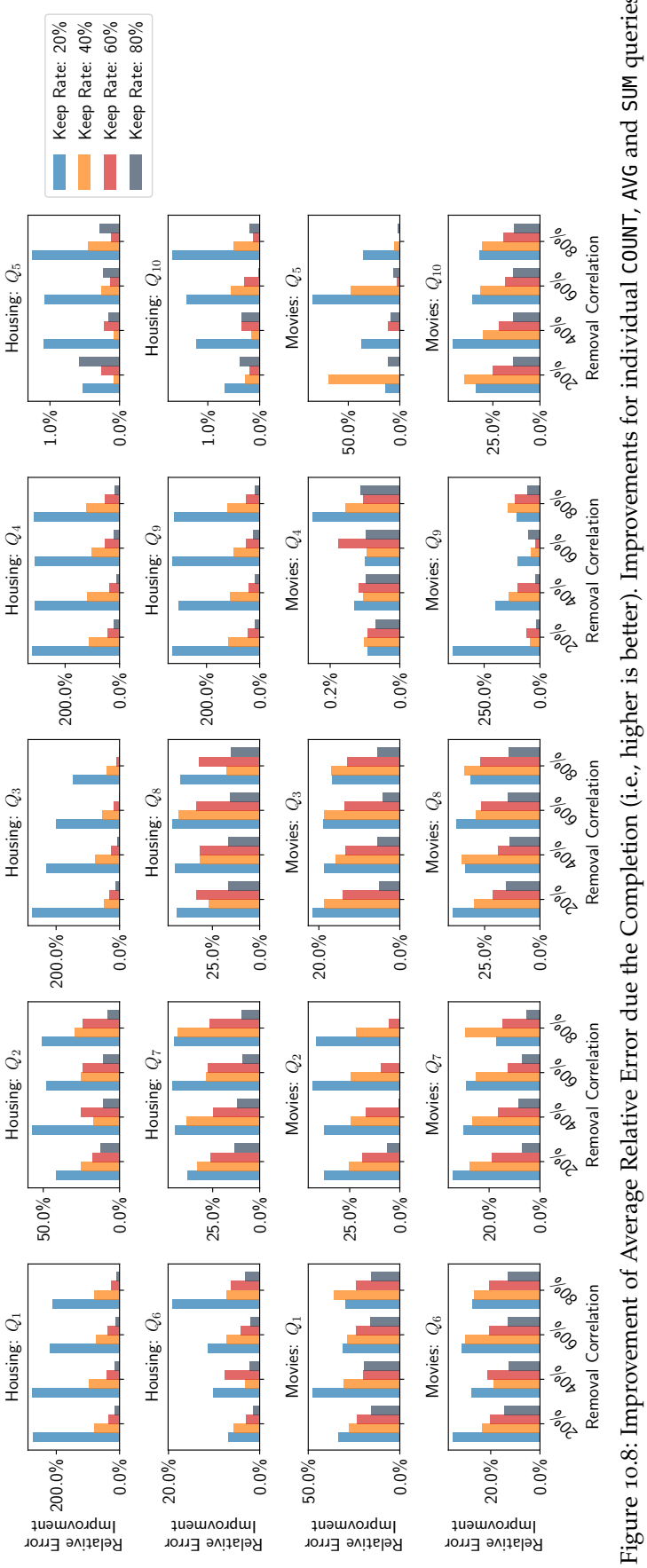


Figure 10.8: Improvement of Average Relative Error due the Completion (i.e., higher is better). Improvements for individual COUNT, AVG and SUM queries are shown as separate plots.

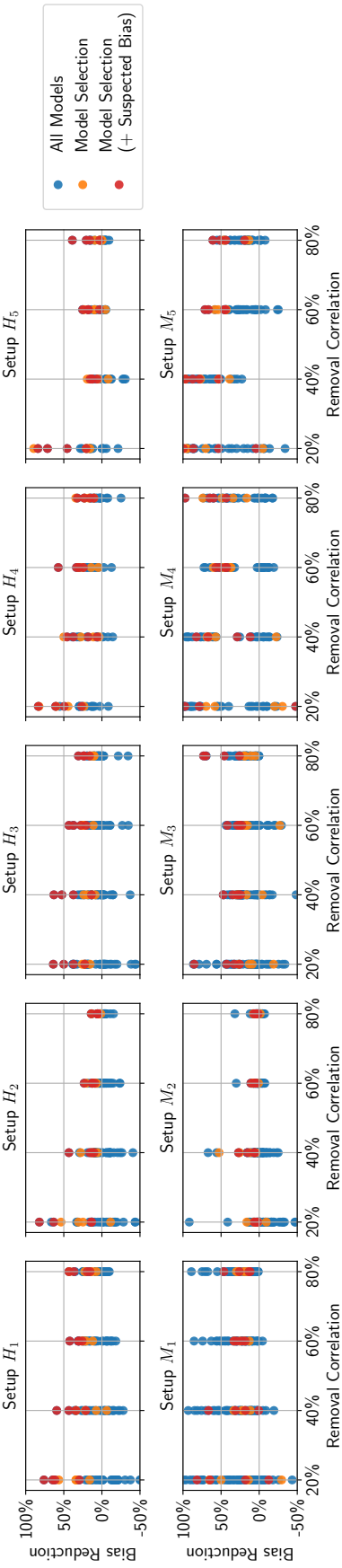


Figure 10.9: Quality of Models selected by our Model Selection Strategy vs. all Models for the different Housing Setups (H_1 to H_5) and Movies Setups (M_1 to M_5). We can select the best performing model in almost all cases if a bias is suspected. If this information is not available, the selected model is usually among the most suitable models for the completion.

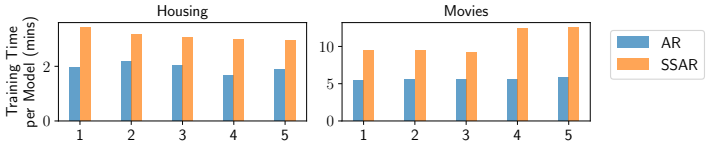


Figure 10.10: Time required for Training.

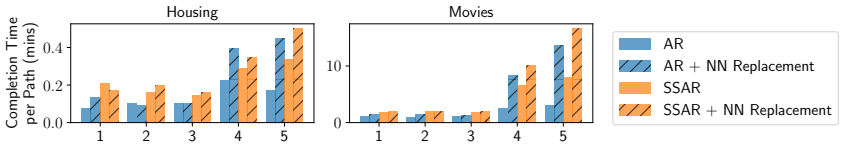


Figure 10.11: Time required for completing one Path.

As we can see in general, AR models require less training time (< 2 minutes for housing and < 6 minutes for the movies dataset). The reason is twofold. First, the models do not require acyclic walks on the schema which have to be performed to gather training data for the SSAR models. Moreover, the models are not as complex since the tree models for the schema walks are not required. While SSAR models require a longer training time, this can be justified with a better performance for some completion setups.

COMPLETION TIME. Finally, we discuss the time needed for data completion. As we can see in Figure 10.11 the completion via one path takes less than 30 seconds for all setups of the housing dataset. For the larger movies dataset, however, the completion took less than two minutes for the completion setups $M_1 - M_3$. For the more challenging setups with long-distance completion paths (distance of four) the completion takes around 16 minutes. However, here millions of tuples have to be synthesized. Moreover, we can see that the nearest neighbor replacement increases the runtime of the completion. As mentioned before, for those scenarios we can alternatively generate the data offline.

10.8 RELATED WORK

MISSING DATA IN OLAP. Closest to our approach is probably the recent Themis [131] system. Different from ReStore, Themis is restricted to work for a single table and requires aggregate information. Themis either reweights existing tuples or learns probabilistic models for missing groups. The techniques for leveraging aggregate knowledge such as iterative proportional fitting could seamlessly be integrated in our approach. Chung et al. [27] estimate the impact of missing tuples on aggregate queries when several but overlapping data sources are integrated. Moreover, only the single table case is discussed here. There has also been work on determining when incomplete data still leads to complete query results [94, 121] or which parts of the result are complete [90] which is orthogonal to our work.

DATA GENERATION. In order to compensate missing tuples, we synthesize missing data using AR and SSAR models. This is related to approaches that synthesize tuples [24, 42, 162, 183] using deep models such as GANs [57, 149]. A main motivation is to synthesize data satisfying data privacy. In contrast to ReStore, the models typically only support individual tables instead of complex schemas.

UNCERTAIN AND PROBABILISTIC DATABASES. Another line of work [43, 163] uses the possible world semantics [1] to handle uncertain data, i.e., either tuple values or the inclusion of tuples in the dataset are uncertain. The goal is to estimate possible results for queries. Alternatively, uncertainty can be modeled using probabilistic databases [31, 75, 130, 147, 151, 160, 175] where tuples or sets of tuples are annotated with probabilities. In contrast to our work, missing tuples cannot be handled directly. Possibly missing tuples would have to be manually inserted in the database and annotated with a probability which is challenging since the user often does not have an understanding of what data is missing.

DATA CLEANING. Our approach is also related to data cleaning. A major direction in data cleaning are approaches for value imputation [26]. For value imputation, there exist many techniques that leverage probabilistic graphical models [146], relational dependency networks [114] or neural approaches [181, 187]. All these approaches, however, cannot synthesize completely missing tuples as we do. Another interesting direction is [177] which estimates the result of aggregate queries by cleaning a sample of dirty data. However, again missing tuples are not being compensated for.

10.9 CONCLUSION AND FUTURE WORK

In this paper, we have introduced ReStore— an approach that approximates queries over a relational database in cases where only incomplete data is available (i.e., tuples in individual tables are missing). In our experimental evaluation, we have demonstrated that our approach can synthesize missing data with high accuracy and thus enables improved decision making on top of incomplete relational databases. In future work, we also want to investigate how the models devised can be used for tasks like missing data imputation or other downstream tasks (e.g., learning a classification model) that can now use the completed dataset as input. In addition, we believe that combining our approach with probabilistic databases is also a promising direction.

ACKNOWLEDGMENTS

This research was partly funded by the BMBF Project *KompAKI*, the Hochtief project *AICO* (AI in Construction) as well as the HMWK cluster project *3AI* (The Third Wave of AI).

11

ONE MODEL TO RULE THEM ALL: TOWARDS ZERO-SHOT LEARNING FOR DATABASES

ABSTRACT

In this paper, we present our vision of so called *zero-shot learning for databases* which is a new learning approach for database components. Zero-shot learning for databases is inspired by recent advances in transfer learning of models such as GPT-3 and can support a new database out-of-the box without the need to train a new model. Furthermore, it can easily be extended to few-shot learning by further retraining the model on the unseen database. As a first concrete contribution in this paper, we show the feasibility of zero-shot learning for the task of physical cost estimation and present very promising initial results. Moreover, as a second contribution we discuss the core challenges related to zero-shot learning for databases and present a roadmap to extend zero-shot learning towards many other tasks beyond cost estimation or even beyond classical database systems and workloads.

BIBLIOGRAPHIC INFORMATION

The content of this chapter was previously published in the peer-reviewed work Benjamin Hilprecht and Carsten Binnig. "One Model to Rule them All: Towards Zero-Shot Learning for Databases." In: *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org, 2022. URL: https://www.cidrdb.org/cidr2022/papers/p16-hilprecht.pdf. The contributions of the author of this dissertation are summarized in Chapter 5.

11.1 INTRODUCTION

MOTIVATION. Building computer systems often involves solving complex problems in all layers of those systems. To reduce complexity when building those computer systems and solve the problems in a tractable manner, these systems have heavily relied on heuristics or simplified analytical models in the past. Very recent work in the systems community, however, has outlined a broad scope where machine learning vastly outperforms these traditional heuristics. This is also the case for databases, where existing DBMS components have been replaced with learned counterparts such as learned cost and cardinality estimation models [67, 72, 79, 113, 161, 185] as well as learned query optimizers [85, 108, 110, 112] or even learned indexes [34, 35, 48, 84] and learned query scheduling strategies [107, 153].

The predominant approach that has been used in the past for learned database components is workload-driven learning. The idea of workload-driven learning is to capture the behavior of a DBMS component by running a representative set of queries over a given database and then use the observations to train the underlying model. For example, for learned cardinality estimation models such as [79, 161] a set of queries must be executed to collect query plans and their cardinalities, which serve as training data for learning a model that can be used to estimate the cardinalities for new queries. The very same procedure is applied if workload-driven learning is used for the other DBMS components such as learned physical design advisors (e.g., an advisor for index selection) [99, 192] or other components.

However, a major obstacle to these workload-driven approaches is the collection of training data. For example, in [79, 161] it was shown that thousands of query plans and their true cardinalities are needed for training the model to achieve a high accuracy. Running such a set of training queries on potentially very large databases to collect the training data can take hours or even days while the actual training of the underlying models often only takes a few minutes. And unfortunately, the training data collection needs to be repeated for every new database that needs to be supported.

To reduce the high cost of training data collection, reinforcement learning (RL) has been used to execute training queries [70, 97, 99, 192] in a more targeted manner (i.e., letting the RL agent decide which queries to execute next). However, even with reinforcement learning still a large amount of training queries needs to be executed for learning a model. Moreover, training the model is not a one-time effort since similar to workload-driven approaches the learning procedure needs to be repeated for every new database at hand.

A different direction that has thus been proposed to avoid the expensive training data collection by running queries on a new database are so called data-driven approaches [72, 185, 186] that learn a model

purely from the underlying data. A prime example where data-driven learning is a perfect fit is cardinality estimation. However, data-driven learning is no silver bullet either since for some DBMS components the information about the runtime behavior of queries is required. One such example is learned physical cost estimation where the runtime behavior of queries needs to be captured by a model to make predictions. A similar observation holds for many other database tasks such as physical design tuning or knob tuning where the effects of a certain decision on the runtime of a workload need to be learned.

VISION AND CONTRIBUTIONS. In this paper, we thus present our vision of so called *zero-shot learning for databases* which is a new learning approach for database components that can support a broad set of tasks on the one hand but does not require to collect training data for supporting a new database on the other hand. In that regard, zero-shot learning for databases combines the benefits of data-driven learning and workload-driven learning. The general idea behind zero-shot learning for databases is motivated by recent advances in transfer learning of models. Similar to other approaches such as GPT-3 [16] which enables zero-shot learning for NLP, a zero-shot model for databases is trained on a wide collection of different databases and workloads and can thus generalize to a completely new database and workload without the need to be trained particularly on that database.

As a core contribution in this paper, we discuss how such an approach of zero-shot learning for databases could work and we also show the feasibility of this approach for the task of physical cost estimation. In our initial results for physical cost estimation, we show that zero-shot models can significantly outperform workload-driven approaches even when providing workload-driven models with a large number of training queries for a particular database at hand whereas zero-shot models can support them out-of-the-box. Moreover, we believe that the real power of zero-shot learning stems from the fact that it is a general principle that can be used for various learned database tasks. For example, we already have initial promising results suggesting that zero-shot learning can not only be used for physical cost estimation on a new database but also for physical design tuning and, in particular, index selection on a database the model has not seen before.

Finally, an important aspect of zero-shot models is that zero-shot learning can easily be extended to *few-shot learning*. Hence, instead of using the zero-shot model out-of-the box (which already can provide good performance), one can fine-tune the model with only a few training queries on an unseen database or task. Compared to workload-driven learning, few-shot learning will require way fewer training queries for adaptation since the general system behavior is already internalized by the zero-shot model.

OUTLINE. The remainder of this paper is structured as follows: Section 11.2 first gives an overview of zero-shot learning for databases and discusses the core challenges related to zero-shot learning. Section 11.3 then discusses the case study of using our approach for learning a zero-shot physical cost model. Moreover, in this section we also present our initial experimental results. Afterwards, Section 11.4 discusses a research roadmap for zero-shot learning for databases beyond cost estimation. Finally, we conclude with a peak into the future in Section 11.5.

11.2 ZERO-SHOT LEARNING FOR DATABASES

In this section, we first give a brief overview of how zero-shot learning for databases works in general and then discuss the core challenges related to enable this approach in an efficient manner.

11.2.1 *Overview of the Approach*

Figure 11.1 shows the high-level idea that is behind our vision of zero-shot learning for databases. During the learning phase, similar to workload-driven learning, for zero-shot learning we have to execute a representative workload and collect training data.

The main difference to workload-driven learning though, which makes our approach attractive, is that zero-shot models *generalize to unseen databases out-of-the-box*. To allow a zero-shot model to make predictions about unseen databases without the need to retrain the model for this particular database, we provide a new method of representing queries as we discuss below (cf. Key Challenges). This transferable representation is at the core of learning zero-shot models in a generalizable way and thus enables them to make predictions for queries on a new database (e.g., for physical cost estimation) that the model has never seen before.

Moreover, for being able to generalize to new databases, a zero-shot model is trained on different databases. While this might seem to cause high upfront costs before a zero-shot model can be used, it is important to note that the *training data collection is a one-time-effort* which is very different from workload-driven learning that needs to collect training data for every new database a model should support. Moreover, cloud database providers such as AWS, Microsoft, or Google, typically already have significant amounts of such information available since they keep logs of their customer workloads and could thus apply zero-shot learning right away without the need to collect training data in the first place.

Finally, a last important aspect is that zero-shot learning is not only generalizable across databases but is a new learning approach that can be applied to a *variety of database tasks* that range from physical cost

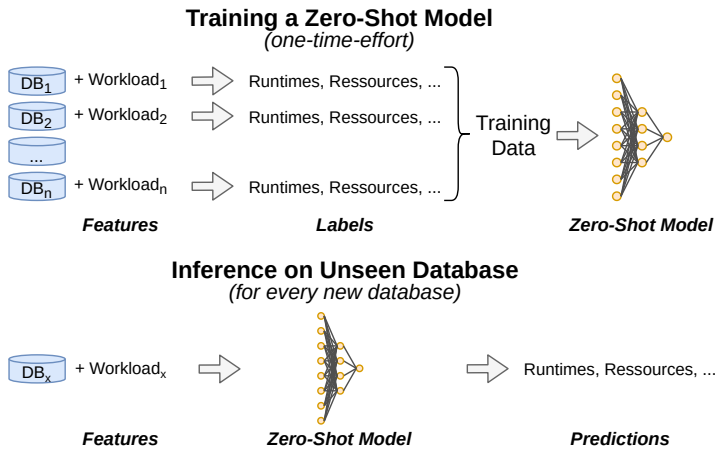


Figure 11.1: Overview of zero-shot learning for databases. In line with other zero-shot approaches such as GPT-3 which enables zero-shot learning for NLP, a zero-shot model for databases can generalize to a completely new database and workload without the need to be trained on that particular database.

estimation, design tuning or knob tuning to query optimization and scheduling as we discuss in Section 11.3 and Section 11.4. To enable zero-shot models to generalize to different tasks though, the models need to be capable of capturing not only information about query plans and their runtimes but also information about other aspects (e.g., how indexes or changes in the database configuration influence the query runtime) as we discuss later.

11.2.2 *Key Challenges*

Enabling zero-shot learning for databases comes with various research challenges. In the following, we discuss the key challenges that we think are at the core to make zero-shot learning for databases efficient and accurate.

TRANSFERABLE REPRESENTATION OF DATABASE AND QUERIES. State-of-the-art workload-driven models [79, 161] can only leverage training data from a single database and thus they cannot simply be trained on a variety of databases to obtain zero-shot models. The reason is that the query representation is not *transferable* to an unseen database. For instance, attributes names (e.g., those used in filter predicates) are typically encoded using a one-hot encoding assigning each column present in the database a specific position in a feature vector. Hence, the column `production_year` of the IMDB dataset might be encoded using the vector $(0, 1, 0)$ (assuming that there are only three columns in total). If the same model is now used to predict query costs for the SSB dataset, the second column in the database might be `region`, which has very different semantics (i.e., very different data distributions or even a different data type). As such, cost models based

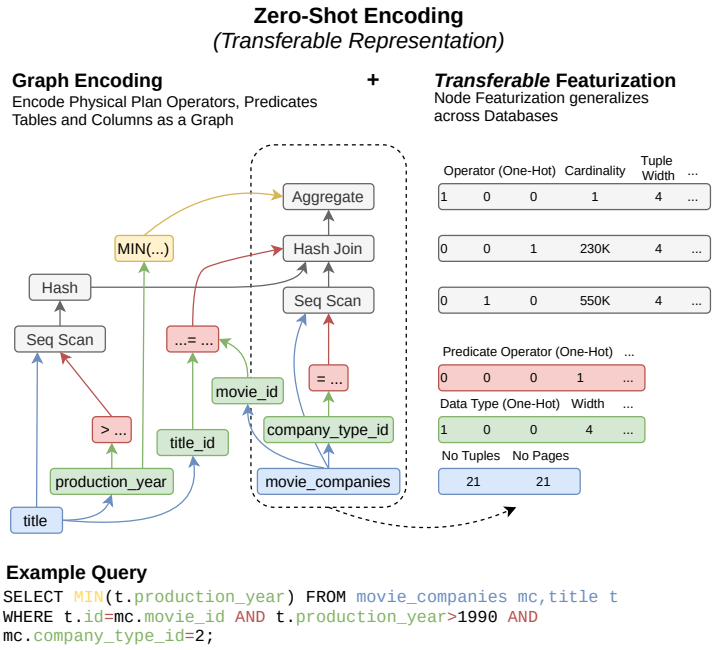


Figure 11.2: Graph encodings with transferable features as a zero-shot encoding. The query is represented as a graph with different node types (for plan operators, predicates, tables, columns etc.) and nodes are annotated with *transferable* features. The representation allows the model to generalize across databases since features remain consistent.

on non-transferable representations will produce estimates that are most likely way off. In fact, such non-transferable feature encodings are used in various places of query representations such as table names as part of plan operators or literals in filter predicates. Hence, for zero-shot models we require a novel representation of queries that is transferable across databases while still being expressive enough to enable accurate estimations.

We will now introduce *query graph encodings with transferable features* (cf. Figure 11.2) which generalize across databases and have the potential to be applied in various zero-shot learned database components. While graph-based encodings have already been used to represent query plan operators and predicates [161], we in contrast encode the entire query as a graph (including tables, columns etc.) and use transferable features per node allowing models using this representation to generalize across databases. For instance, an involved column `production_year` would now be represented using a graph node with transferable features (e.g., the data type). If the same model is now deployed for a different database such as SSB, the corresponding columns would be represented using different graph nodes with their corresponding data characteristics and thus the representation is not inconsistent for different databases (in contrast to one-hot encodings of columns).

While the previously presented representation already allows to represent queries in a very expressive way, extensions might be necessary depending on the specific task that should be solved. When designing a representation for a specific task, it is important that the representation (i) captures all important aspects such that the models are able to solve the task at hand and (ii) that the representation is consistent for different databases (i.e., transferable).

TRAINING DATA COLLECTION AND ROBUSTNESS. A second key challenge to enable zero-shot learning is clearly the training data collection for learning a zero-shot cost model. Here an important question is how many and which databases and workloads a zero-shot model needs to observe during training to make robust decisions on unseen databases. As we show in our initial experiments for zero-shot cost estimation in Section 11.3, for example, already a relatively small number of databases along with the respective workload information (e.g., the featurized query plans and runtimes) is sufficient to generalize robustly and even outperform existing workload-driven approaches. However, clearly we need to develop more theoretical foundations to help guide us as to whether or not a zero-shot model has seen sufficiently many databases and queries.

One way to address this could be to evaluate the model on test databases that have not been used during training similar to the common practice in machine learning to evaluate a model on a holdout set. While this could provide an initial model validation, clearly more theoretical foundations are needed depending on the concrete task. A related question in this respect is how to create the training databases and workloads if they are not yet available (as for cloud providers). An interesting direction here is to use a synthetic approach and generate databases / workloads with different characteristics.

SEPARATION OF CONCERNS. Finally, a last important aspect of zero-shot models is to decide what should be learned by the model to fulfill its core promise and when to separate concerns. For example, workload-driven approaches often prefer end-to-end learning, i.e., to make predictions for a query plan (e.g., the runtime), they internalize both the data characteristics (e.g., the data size and distributions) as well as the system characteristics (e.g., the runtime complexity of database operators) in one model.

However, since the data characteristics can be entirely different for a new database, such an end-to-end approach will not work for zero-shot learning. Hence, we suggest that data characteristics for zero-shot learning should be captured by separate data-driven models (such as [72, 185]). For example, a feature that can be captured by a data-driven model are the input- and output-cardinalities of operators in a query plan. That way, when using cardinalities as input features for the zero-shot models, these models learn to predict the runtime

behavior of operators based on input/output sizes that can be derived for any database which again enables a transferable representation of queries that does not depend on the concrete data distribution of a single database.

One could now argue that this violates the core promise of zero-shot learning since data-driven models need to be learned for each new database. However, data-driven models can be derived from a database without running any training query and typically a sample of the database is enough to train these models. Moreover, for cardinality estimation we could even use simple non-learned estimators (e.g., histograms) as input for the zero-shot models. As we show in our initial results in Section 11.3, even those simple estimators often provide sufficient evidence for our zero-shot cost models to produce accurate estimates.

To summarize, a key question in this context is to decide what a zero-shot model should learn and which aspects should be treated separately. Clearly a guide for this question is to think about what is tied to a particular data distribution and which aspects hold in general which should then be included in the zero-shot model. Moreover, as we discuss later, for design tuning or query optimization another question is how to combine zero-shot models with optimization procedures or other learning approaches (e.g., value networks) to implement efficient search strategies.

11.3 CASE STUDY: COST ESTIMATION

In this case study, we demonstrate how zero-shot learning can be used for physical cost estimation. We envision this to be a potential core building block for zero-shot models for many other DBMS tasks as we discuss in the next section.

11.3.1 *Zero-Shot Cost Estimation*

The main promise of zero-shot cost estimation is that a trained model can predict the query runtime on a new database out-of-the-box. In the following, we give a brief overview of how we implemented our initial prototype for zero-shot cost estimation for Postgres and contrast it to recent workload-driven approaches for cost estimation using the example query plan.

At the core, we use a *transferable* query representation as introduced in Figure 11.2 and propose a new architecture to capture the graph structure. As depicted in the figure, each node in this graph represents a physical operator (as opposed to a logical operator) to capture the differences in runtime complexity during learning. In addition, we use nodes to represent involved tables, columns, aggregations and predicates whereas for each node we use transferable features. For

instance, the `movie_companies` table uses generalizable features such as the number of tuples and pages. Note that for state-of-the-art workload-driven learning, the table would instead be represented as a one-hot encoded vector which does not enable generalization across databases as discussed before.

Moreover, as mentioned before, for zero-shot models we want the model to learn the general runtime behavior of operators in a DBMS (i.e., system characteristics). Hence, the zero-shot model should learn system characteristics separate from data characteristics. This is very different from workload-driven models which learn both aspects end-to-end in one model as mentioned before. For instance, workload-driven models include the values involved in filter predicates (e.g., `1990`) in the featurization of a query and thus learn the selectivity of the filter operation implicitly. In contrast, for zero-shot models we only encode the predicate structure (to represent the general computational complexity) and explicitly use estimated cardinalities from a data-driven model or the query optimizer as input.

Finally, a last important aspect is our proposed model architecture as well as the learning and inference procedure for a featurized plan. Here, we exploit that the graph encoding results in DAGs where the root node of the plan are also the root nodes of the DAGs. The learning overall happens in three steps: we first encode the features of the individual nodes in a fixed-size hidden vector (the initial hidden states). The hidden states of the individual plan nodes are afterwards combined using a bottom-up message passing phase in the DAG. Finally, the hidden state of the root node is fed into a multilayer perceptron (MLP) which predicts the final runtime. In particular in the message passing phase, the DAG is traversed bottom up. The hidden states of the children are summed up (similar to the DeepSets [190] architecture) and combined with the hidden state of the parent node using an MLP to update the hidden state. This process is repeated until the root node is reached and the information of the entire tree is combined. For inference to make a prediction for a query plan on a new database, a new DAG is constructed using the node-specific MLPs and the features are propagated through the tree up to the root node which then predicts the runtime for the new query.

11.3.2 *Initial Evaluation*

SETUP AND BASELINES In an initial experiment, we trained a zero-shot cost estimation model on workloads we executed on a set of publicly available datasets that cover a range of databases with a different number of tables and database sizes. We then predicted the runtimes for a workload on an entirely unseen database to validate that zero-shot cost estimation can provide high accuracies.

As baselines, we used state-of-the-art workload-driven approaches for cost estimation. In particular, we used the end-to-end model of

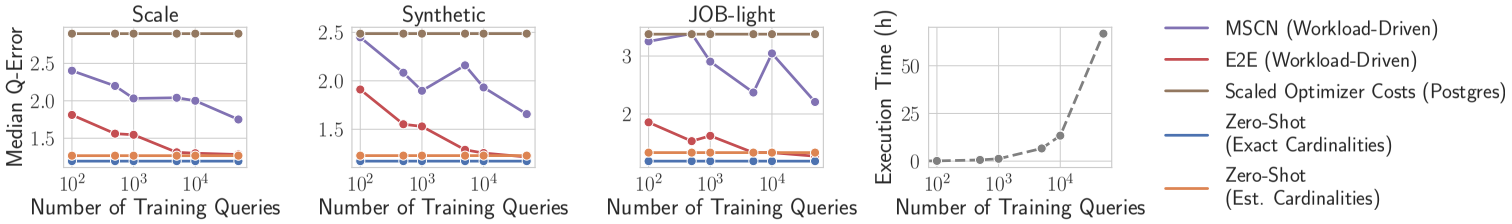


Figure 11.3: Estimation Errors of Workload-Driven Models for a varying Number of Training Queries compared with Zero-Shot Cost Models. The zero-shot models were trained using query executions on entirely different databases and thus do not require any queries on the IMDB database. In contrast, even the most accurate workload-driven model (E2E) requires approximately 10^4 query executions on an unseen database for a comparable performance with zero-shot models which is roughly equivalent to 13 hours of executed workload. Since zero-shot models do not require any additional queries it is significantly cheaper to deploy them for a new database.

[161] called E2E and the MSCN architecture [79] that was initially proposed for cardinality estimation. In addition, we use a simple linear model that obtains actual runtimes from the internal cost metric of the Postgres optimizer (called Scaled Optimizer Cost).

TRAINING OF BASELINES AND ZERO-SHOT MODELS For the workload-driven models, we collected training data of different sizes ranging from a small training set size of 100 queries up to very large training sets with 50,000 queries (similar to [161]). Note, that for every new database such training queries need to be executed and the runtimes need to be collected before a workload-driven model can be trained. Moreover, if the database is updated and data characteristics change, the training data collection needs to be repeated.

For training the zero-shot model, we also need to collect training data. Overall, this gathering of the training data is clearly a significant effort. However, as mentioned before this is a one-time effort and the resulting model can be reused across different databases.

To decide which number of training databases and workloads is sufficient, we evaluated the performance on a holdout test database as we added additional training databases. For each of the training databases we randomly generated 5,000 training queries which we used as training data. After 19 databases, the performance stagnated and we can thus conclude that already a moderate number of training databases is sufficient for zero-shot models to generalize. In total the workload size for zero-shot learning was thus in a similar ballpark compared to the maximum size we used for training the workload-driven baselines. Moreover, the queries for zero-shot learning and workload-driven learning were similar and covered up to five-way joins with up to five numerical and categorical predicates and up to three aggregates. However, different from a workload-driven model, the zero-shot model was not trained on the database it should make a prediction on.

INITIAL RESULTS To evaluate the trained models, we used our zero-shot model as well as the other baselines to predict the runtimes of the commonly used *scale*, *synthetic* and *job-light* benchmarks [79] on the IMDB database. For zero-shot models, we show two versions: one that that uses the Postgres cardinalities as input as well as one version, which uses exact cardinalities (as an upper baseline to show how accurate zero-shot models can become).

As a result, we report the commonly used Q-error which is the factor the predicted runtime deviates from the true runtime. The advantage of zero-shot learning over workload-driven approaches is that no queries on the test database are required for training. To demonstrate this tradeoff, we vary the number of training queries that can be used for the workload-driven baselines and compare the accuracy with zero-shot learning in Figure 11.3. Overall, zero-shot learning can predict

the runtimes very accurately. Since the IMDB dataset was never used for one of the training queries this shows, that zero-shot learning can generalize to unseen databases. Interestingly, even the zero-shot model using only cardinality estimates of the Postgres optimizer is still very accurate.

In contrast, the workload-driven E2E models [161] are less accurate than zero-shot models even for a large set of training queries on the IMDB database for the scale and synthetic benchmarks. For the simpler job-light queries that rarely contain range predicates, the E2E model is on par with the zero-shot model for the larger training sets. However, still the E2E models cannot match the performance of zero-shot learning with exact cardinalities. In addition, we note that the MSCN models are significantly less accurate since they use a much simpler featurization based on one-hot encodings (and not a tree-based featurization). Moreover, note that even though we repeated all measurements multiple times and report the median there are still some peaks in the reported Q-errors of MSCN due to a particularly high variance.

While our initial results are promising, in future we plan to conduct more extensive experiments that show the robustness of zero-shot learning in several dimensions (e.g., more complex queries but also other databases).

11.4 BEYOND COST ESTIMATION

In the following, we will discuss how zero-shot models can be extended beyond cost estimation.

11.4.1 *Physical Design and Knob Tuning*

A first clear extension of zero-shot cost models as described in Section 11.3 is to allow them to support a so called *"What-If"* mode. This enables zero-cost models to predict the runtime of a query given a certain physical design or a database configuration (also called knob configuration). For example, one could then ask the model how the runtime of a query changes if a certain index would exist or how the runtime changes if the buffer size would be increased.

Both these tasks — physical design and knob tuning — are classical problems of DBMSs that have been addressed in the past already by using optimization approaches [4, 18, 23, 124, 127]. However, the main problem was that these approaches relied on inexact cost estimates coming from classical optimizer cost models that were extended to support a *"What-If"* mode. For that reason, recent approaches have suggested to use workload-driven learning — in many cases reinforcement learning [7, 70, 89, 99, 150, 178, 192]. While these approaches have been shown to be more accurate than the more classical approaches,

they again need to first run training queries under different physical layouts or knob configurations for every new database.

Hence, to avoid these high-training costs for every new database one could use a zero-shot cost model in a *"What-If"* mode. To show the feasibility of this direction, we extended our zero-shot cost model to support also decisions towards index tuning. At the core, the zero-shot model for index tuning should be able to support predictions of the runtime as if a certain index would exist in a database. For training a zero-shot cost model that can answer such questions, we again used the 19 databases as training data that we also used in Section 11.3. However, we additionally created a random but fixed set of indexes per database before running the training queries. The zero-shot cost model could recognize for which training query an index was used since the physical operators in a query plan change (e.g., an index scan instead of a table scan was used). The zero-shot model could thus learn how the runtime changes for query plans, which use an index scan compared to query plans which do not use an index scan.

For showing that the learned model could estimate the runtime of queries correctly given a certain index, we again use the IMDB database for the evaluation that the zero-shot model had not seen before. For testing, we asked the model to estimate the runtime of queries if an index would exist again for randomly selected attributes of queries. The estimation errors for zero-shot learning for this workload are given in Table 11.1 (last line). As we can see, the estimations are still very accurate but clearly the maximum Q-error increases compared to the results for zero-shot cost models in Section 11.3 before (upper lines).

To further improve the quality of zero-shot cost models for index tuning one might need to think about a more sophisticated workload sampling or provide additional characteristics for indexes (e.g., expected index height) as input features to a zero-shot model that could be derived with additional data-driven models. Furthermore, we think that we could use zero-shot models to build other design advisors (e.g., for materialized views) or support zero-shot knob tuning to select an optimal database configuration for a given workload without having seen the database for training. Finally, for knob tuning one could also think about using zero-shot models to only guide the search initially to a good start configuration (i.e., to narrow down the search space) and then use online approaches for fine-tuning the knobs since knobs can be changed easily compared to the high cost of changing a physical layout.

11.4.2 *Query Optimization*

Another direction for zero-shot learning are end-to-end learned optimizers and not just the learning of cost models. Recently, it was also

	Zero-Shot (Exact Card.)			Zero-Shot (Estimated Card.)		
Workload	median	95th	max	median	95th	max
Scale	1.19	1.93	3.93	1.26	2.46	4.70
Synthetic	1.17	1.90	4.40	1.21	2.17	6.88
JOB-light	1.18	1.85	2.47	1.33	2.56	4.00
Index	1.21	2.51	10.73	1.33	3.59	24.62

Table 11.1: Estimation errors (Q-errors) of zero-shot models for index tuning (last line) compared to zero-shot cost models without *What-if* support (upper lines).

proposed to replace query optimizers that typically rely on heuristics (i.e., simple cost models) and manual engineering by machine learning models [85, 108, 110, 112]. While initial results are promising and even commercial optimizers can be outperformed by learned ones, current approaches are also dominated by reinforcement leaning or workload-driven learning in general. Again, all these approaches are database-dependent and cannot generalize to unseen databases. Moreover, for learning an optimizer a huge number of queries has to be executed to learn what is a good plan for a given query. We envision that this overhead for unseen databases can be eliminated completely using zero-shot learning.

An initial naïve approach for this could be to use the devised zero-shot cost estimation model to evaluate candidate plans and thus better guide the query optimizer to plans with low costs. For instance, zero-shot cost estimation models could be used in conjunction with classical dynamic programming or even approaches like Bao [108]. However, with more sophisticated approaches, we think that zero-shot learning could potentially replace classical heuristics like dynamic programming entirely by devising zero-shot value networks to learn search strategies for query optimization. Value networks [165] have shown to learn policies that involve planning-based reasoning. This way, zero-shot query optimizers could come up with plans that classical optimizers would not have considered while avoiding the burden to run thousands of queries to train the learned optimizer for every new database.

11.4.3 *Discussion*

In addition to design advisors, knob tuning, or database optimizers there are many more DBMS components that could benefit from zero-shot learning. For example, zero-shot cost models could be used to predict not only the runtime but also other aspects such as resource consumption and thus be used also for runtime decisions (e.g., query scheduling). Moreover, by extending the features of the *"What-If"* mode, we could also support hardware aspects and predict the

runtime of queries on an unseen hardware, e.g., to select an optimal cloud instance for a given workload.

Another interesting question is how zero-shot learning should be integrated into the overall DBMS architecture. Here we envision a route where *zero-shot cost models* as presented in Section 11.3 form a "kind-of" *central brain* in a DBMS that can be leveraged by various DBMS components that complement such a central component with more targeted models. These additional models could for example be zero-shot models that focus on learning particular search strategies or specific data-driven models to capture interesting data characteristics as we discussed before.

Finally, as mentioned before it can be beneficial to fine-tune a zero-shot model also on the unseen database. The resulting few-shot models leverage the observed workload on the database similar to workload-driven models and thus likely offer more accurate predictions. However, the main difference to workload-driven models is that our approach also offers accurate out-of-the-box predictions for unseen databases by using zero-shot models and also requires fewer queries for adaptation on an unseen database since the general system behavior is already internalized by the zero-shot model. As such, it is significantly more efficient to fine-tune a model for unseen databases than to train one from scratch every time as it is necessary for workload-driven models.

11.5 LOOKING INTO THE FUTURE

In this paper, we have shown a new approach for learned database components that can support new databases without running any training query on that database. Moreover, zero-shot models can be fine-tuned on the unseen database for more accurate predictions resulting in few-shot models. While we have focused on single-node databases and classical database workloads in the first place, we believe that zero-shot models can be applied more broadly. One direction are distributed DBMSs where zero-shot models can be extended to support tasks such as to optimize a distributed data layout. Another direction is to extend zero-shot models for other types of data-intensive workloads (e.g., data streaming).

Moreover, when thinking more broadly, zero-shot models seem to also be an attractive model for any system builder and can be also used at various levels of granularity to predict the performance of individual components (e.g., very fine-grained on the data structure and algorithm level) or very coarse-grained (at the system level). For example, when being used for data structures and algorithms, zero-shot models would be an efficient vehicle for self-designing data structures [74]. To conclude, we think that zero-shot learning opens up many avenues of research since it provides not only a more *sustainable*

way to build learnable system components but it also seems to be a *general paradigm* that can be applied more broadly and at different levels.

11.6 ACKNOWLEDGMENTS

We thank the reviewers for their feedback and comments. This research and development project is funded by the German Federal Ministry of Education and Research (BMBF) within the "The Future of Value Creation – Research on Production, Services and Work" program and managed by the Project Management Agency Karlsruhe (PTKA). The author is responsible for the content of this publication. In addition, the research was partly funded by the Hochtief project *AICO* (AI in Construction), the HMWK cluster project *3AI* (The Third Wave of AI), as well as the DFG Collaborative Research Center 1053 (MAKI). Finally, we want to thank the Amazon Redshift team for valuable discussions.

12

ZERO-SHOT COST MODELS FOR OUT-OF-THE-BOX LEARNED COST PREDICTION

ABSTRACT

In this paper, we introduce zero-shot cost models, which enable learned cost estimation that generalizes to unseen databases. In contrast to state-of-the-art workload-driven approaches, which require to execute a large set of training queries on every new database, zero-shot cost models thus allow to instantiate a learned cost model out-of-the-box without expensive training data collection. To enable such zero-shot cost models, we suggest a new learning paradigm based on pre-trained cost models. As core contributions to support the transfer of such a pre-trained cost model to unseen databases, we introduce a new model architecture and representation technique for encoding query workloads as input to those models. As we will show in our evaluation, zero-shot cost estimation can provide more accurate cost estimates than state-of-the-art models for a wide range of (real-world) databases without requiring any query executions on unseen databases. Furthermore, we show that zero-shot cost models can be used in a few-shot mode that further improves their quality by retraining them just with a small number of additional training queries on the unseen database.

BIBLIOGRAPHIC INFORMATION

The content of this chapter was previously published in the peer-reviewed work Benjamin Hilprecht and Carsten Binnig. "Zero-Shot Cost Models for Out-of-the-box Learned Cost Prediction." In: *Proc. VLDB Endow.* 15.11 (2022), pp. 2361–2374. DOI: 10.14778/3551793.3551799. URL: http://www.vldb.org/pvldb/vol15/p2483-hilprecht.pdf. The contributions of the author of this dissertation are summarized in Chapter 5.

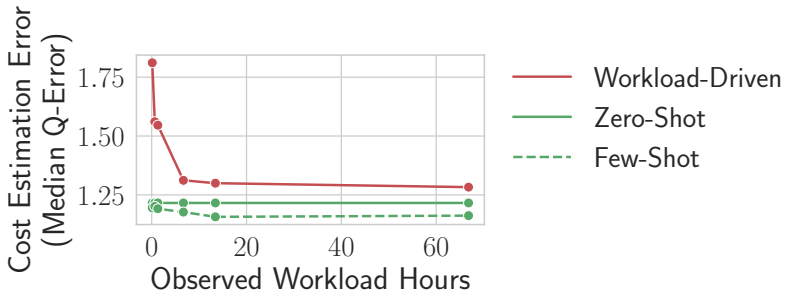


Figure 12.1: Cost Estimation Errors on the IMDB database. While workload-driven approaches [161] require many hours of workload executions as training data, our zero-shot cost model supports the unseen IMDB database *out-of-the-box* and provides highly accurate cost estimates. If a workload is observed however, the zero-shot model can be fine-tuned, which further improves the performance.

12.1 INTRODUCTION

MOTIVATION. Accurate physical cost estimation (i.e., estimating query latencies) is crucial for query optimization in DBMSs. Classically, cost estimation is performed using models that make several simplifying assumptions. As a result, such models often over- or underestimate runtimes, leading to suboptimal planning decisions that degrade the overall query performance [93]. Recently, machine learning (ML) has thus been used for learned cost models that do not need to make such simplifying assumptions [161].

While it was shown that the cost estimates of such learned cost models are significantly more accurate than those of the traditional cost models, the existing approaches rely on *workload-driven* learning where models have to observe thousands of queries on the same database[1] for which the cost prediction should be performed. This workload execution is required to gather the training data, which can take hours (or days) since tens of thousands of queries need to be executed on potentially large databases.

In Figure 12.1, we show the cost estimation accuracy depending on how many hours we allow for gathering the training data for a workload-driven model. As we can see, even for a medium-sized database such as IMDB, it takes more than 5 hours of running queries on this database to gather enough training data such that the cost estimation model can provide a decent accuracy.

Unfortunately, collecting training data by running queries is not a one-time effort. In fact, the training data collection has to be repeated for every new database a learned model should be deployed for. This is due to the fact that current model architectures for workload-driven learning tie a trained model to a particular database instance.

1 Throughout this paper, we use the term database to refer to a particular dataset with certain data characteristics.

Consequently, for every (new) unseen database we not only have to train a model from scratch but also gather training data in the form of queries. And even for the same database, in case of changed data characteristics due to updates, training data collection needs to be repeated. Overall, these repeated high costs for obtaining training data for unseen databases render workload-driven learning unattractive for many practical deployments.

CONTRIBUTIONS. In this paper, we thus suggest a new learning paradigm for cost estimation called zero-shot cost models that reduces these high efforts. The general idea behind zero-shot cost models is motivated by recent advances in transfer learning of machine learning models. While a wide spectrum of methods have been proposed already to tackle zero-shot learning in domains such as NLP [16] or computer vision [88], no approaches for zero-shot learning exist for learned DBMS components and in particular also for cost models. To enable this, as a core contribution in this paper, we propose a new query and data representation that allows zero-shot cost models to be pre-trained across databases and thus be used out-of-the-box (or with minimal fine-tuning only) on unseen databases.

In fact, as depicted in Figure 12.1 zero-shot cost models can thus provide a high accuracy and even outperform existing workload-driven approaches that have been trained on large sets of training queries. One could now argue that it might be a significant effort to collect sufficient training data across databases for pre-training a zero-shot model. However, in contrast to workload-driven models, which require training data for every unseen database, training data collection is a one-time effort; i.e., once trained the zero shot model can be used for any new unseen database. In fact, in our evaluation we show that zero-shot models can provide high accuracies for a wide variety of real-world databases. Moreover, historical traces can be used, which eliminates the need to collect any training data. For example, cloud providers such as AWS, Microsoft, or Google, typically anyway keep logs of their customer workloads, which could directly be used as training data for zero-shot learning without collecting any further training data.

A key aspect to enable zero-shot learning is that a cost model can be transferred to new (unseen) databases, i.e., the models leverage observed query executions on a variety of different databases to predict runtimes on new (unseen) databases. However, state-of-the-art model architectures used for workload-driven learning do not support this training and inference mode since they are tied to a particular database. As a core novel contribution for zero-shot cost models we thus devise a new model architecture based on a representation of queries that generalizes across databases using a *transferable* representation with features such as the tuple width that can be derived from any database. Moreover, zero-shot models *separate concerns*; i.e., data characteristics

of a new database (e.g., rows of tables) are not implicitly learned as in classical workload-driven learning (which hinders generalization), but are provided as input to the model.

Another core question for zero-shot models is at which point a sufficient amount of different training databases and workloads was observed to generalize robustly to unseen databases. To answer this question, as a second contribution in this paper we derive a method to estimate how accurate the runtime estimations of zero-shot models will be for unseen databases. We also discuss how to address cases of workload drifts where the zero-shot models are expected to generalize less robustly. Furthermore, we also show that zero-shot models are widely applicable beyond cost models for query optimizers for single-node DBMSs, which is the main focus of this paper. For instance, we have initial results that zero-shot cost models can be naturally extended to distributed DBMS or even other use cases such as providing cost estimates for design advisors where the goal is to automatically find a suitable database design (e.g., a set of indexes) for a given workload. Due to space constraints, we defer these results to an extended technical report.

Finally, in our extensive experimental evaluation, we verify that zero-shot cost models generalize robustly to unseen databases and workloads while providing cost estimates which are more accurate than those of workload-driven models. As part of this evaluation, we also provide a new benchmark (beyond JOB), which is necessary to evaluate cost estimation models more broadly on a variety of (real-world) databases. We will make this benchmark including query executions for training cost models publicly available and hope that it will benefit future research in learned cost estimation and potentially beyond.

OUTLINE In Section 12.2, we give an overview of our approach and describe the model architecture in more detail in Section 12.3. We then derive formal methods to recognize when sufficient training data is available for the model to generalize in Section 12.4. Before discussing the evaluation in Section 12.6, we describe the design decisions for our proposed benchmark to evaluate cost models. Finally, we present related work (Section 12.7) and conclude in Section 12.8.

12.2 OVERVIEW

In this section, we introduce the problem of zero-shot cost estimation and then present an overview of our approach.

12.2.1 *Problem Statement*

The overall goal of zero-shot cost estimation is to predict query latencies (i.e., runtimes) on an unseen database without having observed any query on this unseen database. Throughout this paper we use the term database to refer to a particular dataset (i.e., a set of tables with a given data distribution). Note that the problem of zero-shot cost estimation is thus in sharp contrast to the problem addressed by state-of-the-art workload-driven cost models, which train a model per database. Finally, while we believe that zero-shot learning for DBMSs is more generally applicable, we restrict ourselves in this paper to cost estimations for relational DBMSs. In particular, zero-shot cost models for other types of systems such as graph-databases or streaming systems are interesting avenues of future work.

12.2.2 *Our Approach*

A key challenge for developing zero-shot cost models is the question how to design a model that allows to generalize across databases. Here, we draw inspiration from the way classical cost models in DBMSs are designed. Typically, these consist of two models: a database-agnostic model to estimate the runtime cost and a database-dependent model (e.g., histograms) to capture data characteristics. When predicting the cost of a query, the estimated cardinalities and other characteristics (i.e., outputs of the database-dependent models) serve as input to the general database-agnostic cost model, which captures the general system behavior (e.g., the costs of a sequential scan grows linearly w.r.t. the number of rows). While the classical models are lightweight, they often largely under- or overestimates the true costs of a query since models are too simple to capture complex interactions in the query plan and data.

Hence, in our approach, we also separate concerns but use a much richer learned model, which similarly takes data characteristics of the unseen database as input to predict query runtimes in a database-agnostic manner. As depicted in Figure 12.2 (upper part), for training such a zero-shot cost model we provide different query plans along with the runtime as well as the data characteristics of the plan (such as tuple width as well as intermediate cardinalities) to the zero-shot cost model. Once trained, the model can be used on unseen databases to predict the query runtime as shown in Figure 12.2 (lower part).

As mentioned before, to predict the runtime of a query plan on a new (unseen) database, we feed the query plan together with its data characteristics into a zero-shot model. While data characteristics such as the tuple width can be derived from the database catalogs, other characteristics such as intermediate cardinalities require more complex techniques. To derive intermediate cardinalities of a query plan in our

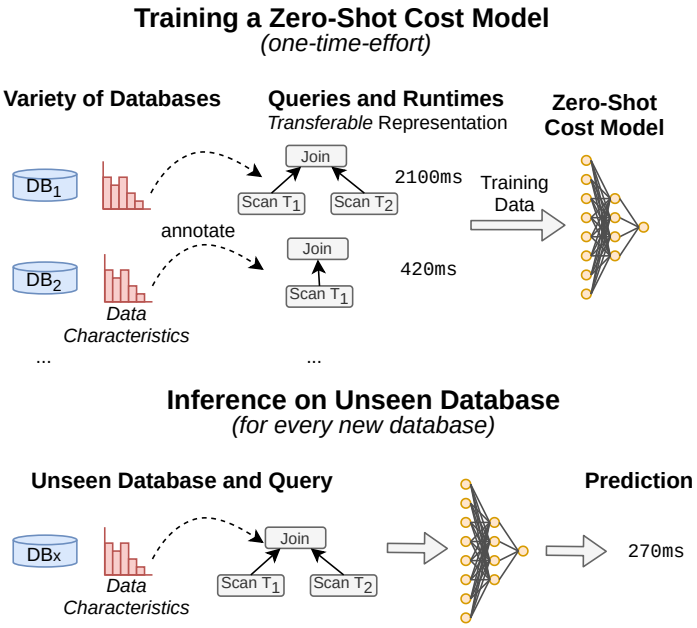


Figure 12.2: Overview of Zero-Shot Cost Estimation. The zero-shot cost model is trained *once* using a variety of queries and databases. At inference time, the model can then provide cost estimates for an unseen database and queries without requiring additional training queries. Enabling zero-shot cost estimation is based on two key ideas: (1) a new *transferable* query representation and model architecture is used to enable cost predictions on unseen databases and (2) we separate concerns, i.e., a zero-shot model learns a general database-agnostic cost model, which takes database-specific characteristics as input.

approach we thus make use of data-driven learning [72, 185] that can provide exact estimates on a given database. Note that this does not contradict our main promise of zero-shot learning since data-driven models to capture data characteristics can be learned without queries as training data.

Another core challenge of enabling zero-shot cost models that can estimate the runtime of a plan given its data characteristics is how to represent query plans, which serve as input to the model. While along with workload-driven cost models, particular representation methods for query plans have already been proposed, those are not applicable for zero-shot learning. The reason is that the representations are not *transferable* across databases. For instance, literals in filter predicates are provided as input to the model (e.g., `2021` for the predicate `movie.production_year=2021`). However, the selectivity of literals will vary largely per database since the data distribution will likely be different (e.g., there might not even exist movies produced in 2021 in the test database).

Hence, as a second technique in this paper, we propose a new representation for queries that completely relies on features that can be derived from any database to allow the model to generalize to unseen databases. For example, predicates for filter operations in a query are encoded by the general predicate structure (e.g., which data types and comparison operators are used in a predicate) instead of encoding the literals. In addition, data characteristics of a filter operator (e.g., input and output cardinality to express the selectivity) are provided as additional input to a zero-shot model. That way, a zero-shot model can learn the runtime overhead of a filter operation based on database-agnostic characteristics. We present details of our query representation in Section 12.3.

Finally, a last important aspect of zero-shot cost models is that they can easily be extended to *few-shot learning*. Hence, instead of using the zero-shot model out-of-the box (which already can provide good performance), one can fine-tune the model with only a few training queries on an unseen database.

12.2.3 *Assumptions and Limitations*

While we expect zero-shot cost models to support a variety of different databases and workloads out-of-the-box, we next discuss the assumptions for a successful generalization.

In this paper, the main assumption is that we only focus on the transfer of learned cost models across databases for a *single database system* on a *fixed hardware*. We think that this is already challenging and allows for many interesting use cases. For instance, with zero-shot cost models cloud DBMSs (such as Redshift or Snowflake) can use learned cost models for new customer databases and workloads

with significantly lower training overhead compared to the existing workload-driven models that require that a model is trained per new database. While we believe that zero-shot cost models can be extended to support also the transfer of cost models between different hardware setups and DBMSs by adding additional transferable features, we leave this to future work and assume a fixed hardware and DBMS in this paper.

Furthermore, while zero-shot cost models can generalize to unseen query patterns as we show in our experiments, it is clearly required that the training queries have a certain *coverage*, i.e., come with a diverse set of workloads and databases. For instance, it is a minimum requirement that every physical operator is observed in the training data s.t. the model can internalize the overall characteristics. Moreover, if there are extreme differences between training and test workloads, we expect the zero-shot model accuracy to degrade. We discuss how to detect and mitigate such cases by fine-tuning a zero-shot model in Section 12.4.

12.3 ZERO-SHOT COST MODELS

As mentioned in Section 12.2, a zero-shot cost model (once trained) is able to predict the runtime of a query on an entirely new database without retraining. A core building block needed to enable a zero-shot model is a new representation of queries that can generalize across databases. In the following, we thus first explain how we devised such a transferable query representation and then discuss how inference and training of a zero-shot model that uses this representation works.

12.3.1 *Query Representation*

State-of-the-art workload-driven models [79, 161] for cost estimation do not use a transferable query representations and can thus only be used on the database they were trained on. To better understand why current query representations are not transferable, we first explain how they typically encode queries.

12.3.1.1 *Query Representation for Workload-Driven Models.*

At the core, query representations used for workload-driven approaches hard-code the model against a single database. For example, column names (e.g., those used in filter predicates) are typically encoded using a one-hot encoding assigning each column present in the database a specific position in a feature vector. For instance, the column `production_year` of the IMDB dataset might be encoded using the vector $(0, 1, 0)$ (assuming that there are only three columns in total). If the same model should now be used to predict query costs for the SSB dataset, some columns might not even exist or even worse they

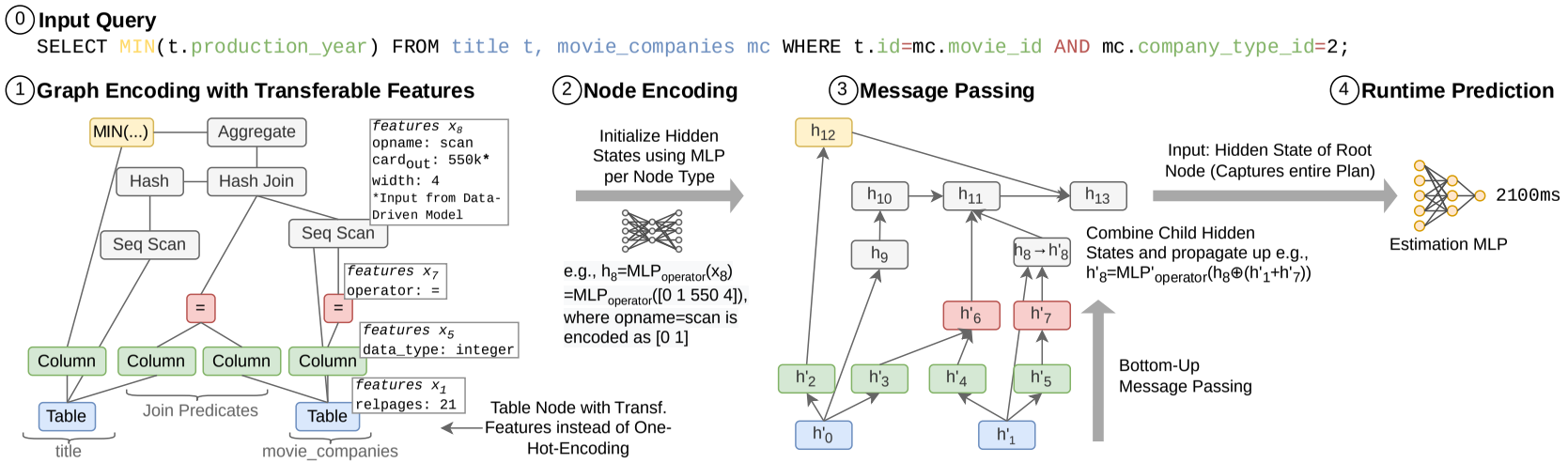


Figure 12.3: Using Zero-Shot Models for Cost Estimation (i.e., for Inference) on an unseen Database. (1) A query is represented as a graph with different node types (to represent plan operators, predicates, tables, columns etc.) and nodes are annotated with *transferable* features, which generalize across databases. (2) Afterwards, the resulting feature vectors of the nodes are fed into node-type-specific Multi-Layer-Perceptrons (MLPs) to obtain a hidden state, which is (3) propagated through the query-tree using bottom-up message passing to account for interactions among connected nodes. (4) Finally, the hidden state of the root node (encoding the entire graph) is fed into a final model — the estimation MLP — which predicts the query runtime.

might exist but have very different data distributions or even a different data type. In fact, non-transferable feature encodings are not only used for columns but in various places of the query representation such as encoding table names or literals in filter predicates.

12.3.1.2 *Query Representation for Zero-Shot Cost Models.*

Hence, for zero-shot cost models we require a new query representation that is transferable across databases. The main idea of the transferable representation we suggest in this paper is shown in Figure 12.3. At the core, a query plan and the involved tables and columns are represented using a graph where graph nodes use transferable features (1) (i.e., features that provide meaningful information to predict runtime on different databases). This representation then serves as input for the training and inference process of zero-shot cost models (2)-(4) that we explain in the subsequent sections. In the following, we discuss the graph encoding of the transferable featurization in detail.

GRAPH ENCODING OF QUERY PLANS. While graph-based representations have been already used to represent query operators of a query plan [161], our representation has significant differences. First, as shown in Figure 12.3 (1), our representation not only encodes physical plan operators as nodes (gray) in the graph as in previous work [161], but it also covers all query plan information more holistically using different nodes types for input columns (green), tables (blue) as well as predicate information (red). Second, as discussed before, previous approaches also covered such information, however, they used one-hot-encodings (which are non-transferable) while our representation captures the query complexity in a transferable way.

For instance, to encode filter predicates, different from previous approaches we encode the predicate structure as nodes (red) without literals. In particular, we encode information such as data types of the columns and operators used for comparisons. For example, the filter predicate `company_type_id=2` for the query (0) in Figure 12.3, is encoded using a *column* node (x_5) with the comparison node = (x_7). As such, a zero-shot cost model provided with the transferable features (e.g., intermediate cardinalities, which are given by the data-driven models) can infer the complexity of the predicates to estimate the query runtime.

TRANSFERABLE FEATURIZATION. While our graph representation allows to flexibly encode query plans across databases, we similarly have to make sure that the features used to represent nodes in the graph (1) (e.g., plan operators as shown in gray) are transferable. In particular, when used on different databases, features should not encode any implicit information that hinder the transfer of the model to a new unseen database.

The concrete set of such features used for the different node types in our graph representation is depicted in Table 12.1 specifically for zero-shot cost models on Postgres, which we use as DBMS in all our experiments. For instance, input column nodes (green) use features such as the data type or the width in bytes. Similarly, for tables (blue nodes), we use other transferable features (e.g., the number of rows as well as the number of pages on disk). However, note that the general class of features allows cost predictions also for other single-node DBMSs such as MySQL. The rationale for selecting these features is to include transferable features that cover very different aspects aspects regarding the query (e.g., involved operators, column types in predicates) as well as the data (e.g., queried tables, data distribution). As we will show in an ablation study in the evaluation, each such aspect with the corresponding individual features improves the cost estimation accuracy.

Importantly, transferable features can either characterize the query plan (e.g., operator types) or represent the data characteristics (e.g., intermediate cardinalities) and together allow a zero-shot cost model to generalize to an unseen database. For transferable features that represent data characteristics many can be derived from the metadata of a database (such as the the number of rows of a table node). However, some other features that represent data characteristics — e.g., the estimated output cardinality of an operator node — require more involved techniques. In Section 12.3.4, we discuss alternatives of how we provide estimated output cardinalities to zero-shot cost models.

12.3.2 *Inference on Zero-Shot Models*

Once a query graph with the transferable features on an unseen database is constructed for a query plan, it can be used as input for a (trained) zero-shot cost model to predict the runtime. Predicting the runtime of a new query plan with a zero-shot cost model is executed in three steps, which we depict as pseudocode in Algorithm 4: First, we compute a hidden state for every node of the query graph (2) given the node-wise input features. Second, the information of different graph nodes is combined using message passing (3) before a Multi-Layer-Perceptron (MLP) predicts the runtime of the query plan (4). The same steps are also reflected in Figure 12.3 (2) to (4).

In particular, in step (2), the feature vectors x_v of each graph node v are encoded using a node-type specific MLP, i.e., nodes of the same type (e.g., all plan operators) use the same MLP to initialize their hidden state h_v (line 5). For instance, in Figure 12.3, the hidden state h_8 of the node representing the sequential scan on the `movie_companies` table is obtained by feeding the feature vector x_8 (containing transferable features) into an MLP, which is shared among all plan operators (gray nodes).

Node Type/Category	Feature	Description
Operators/	`workers`	Number of parallel workers
Data Distribution	`opname`	Name of physical operator
	$\texttt{card}_{\texttt{out}}$	Estimated output cardinality of operator
	`width`	Tuple width
	`card_prod`	Estimated product of children output cardinalities
Predicate	`operator`	Operator type (e.g., `=`)
	`literal_feat`	Feature capturing literal complexity, e.g., number of values for `IN` operator or regex complexity
Table	`relpages`	Number of pages
	`reltuples`	Number of rows
Input Column	`width`	Avg. number of bytes to represent a value
	`correlation`	Attribute correlation with row number
	`data_type`	Data type of column
	`ndistinct`	Number of distinct attribute values
	`null_frac`	Fraction of `NULL` values
Output Column	`aggregation`	Which aggregation type is used

Table 12.1: Zero-Shot Features. All features are transferable and have the same semantics for different databases.

Afterwards, in step (3), a message passing scheme is applied, which is prominently used in graph neural networks (GNNs) [54] to model the interactions between nodes in graphs (i.e., to capture interactions of query operators in the plan such as effects of a pipelined query execution). Different from message passing schemes for general graph encodings, for the message passing in zero-shot models we can exploit the fact that queries can be represented as directed acyclic graphs (DAGs) since query-plans are tree-structured. We thus use a novel bottom-up message passing scheme through the graph (i.e., in topological ordering) to obtain an updated hidden state h'_v of a node v that contains all information of the child nodes. During this pass, the updated hidden states h'_u of the children u are combined by summation [190] and concatenated with the initial hidden state h_v of a node and fed into a node-type-specific MLP (line 7). For instance, in Figure 12.3, the updated hidden state h'_8 of the scan node is obtained by summing up the updated hidden states of the child nodes (representing the table and predicate operator of the scan) concatenated with the initial hidden state (capturing properties of the scan), which is then fed into an MLP, which is again shared among all plan operators.

Finally, as a result of step (3) the updated hidden state h'_r of the root node r of a query plan captures the properties of the entire query. For the cost prediction in step (4), we thus feed this hidden state into a final estimation MLP to obtain the cost estimate $\hat{c} = MLP_{est}(h_r)$

Algorithm 4 Inference on Zero-Shot Models

1: **Input:** Query graph encoding with nodes v and input features x_v
2: **Output:** Cost estimate $\hat{c}$
3:
4: **for** $v \in$ graph encoding **do** ▷ Compute hidden state per node (2)
5: $h_v \leftarrow MLP_{T(v)}(x_v)$
6: **for** $v \in$ in topological ordering **do** ▷ Bottom-up pass in graph (3)
7: $h'_v \leftarrow MLP'_{T(v)}\left(\sum_{u \in children(v)} h'_u \oplus h_v\right)$
8: $\hat{c} \leftarrow MLP_{est}(h'_r)$ ▷ Estimate costs using root node state (4)
9: **return** $\hat{c}$

(line 8). Hence, in Figure 12.3 (4), the updated hidden state h'_{13} is fed into the final estimation MLP to obtain the cost estimate since it captures information of the entire plan.

12.3.3 *Training Zero-Shot Models*

As mentioned before, a zero-shot cost model is trained on several databases and queries to learn the runtime complexity of query plans given the input features. To be more precise, a zero-shot cost model is trained in a supervised fashion using pairs (P, c) that consists of a plan P with the respective features and the actual runtime cost c. Importantly, all steps described in the inference procedure (node encoding, message passing and finally runtime estimation) are differentiable, which allows us to train the model parameters of the MLPs used for all zero-shot model components jointly in an end-to-end fashion. As loss function to compare the actual costs c of a featurized query plan P with the estimated costs $\hat{c}$, we use the Q-error loss $\max(\frac{c}{\hat{c}}, \frac{\hat{c}}{c})$ [79, 161] since this worked best for zero-shot models compared to other alternatives.

12.3.4 *Deriving Data Characteristics*

As discussed before, an important aspect of a zero-shot model is that the model is not tied to a particular data distribution of a single database. For enabling this, we provide data characteristics such as column widths in bytes, number of pages and tuples of tables but also output cardinalities of operators as input to those models. To be more precise, given a particular query plan for which the runtime should be estimated, those features have to be annotated for each graph node in the query encoding.

While the majority of those features can simply be derived from the database catalog, intermediate cardinalities in a query plan are notoriously hard to predict and simple statistics are known to be often imprecise [93]. Hence, learned approaches to tackle cardinality

estimation have been proposed to derive accurate intermediate cardinalities. While in principle such learned approaches can be used to predict intermediate cardinalities, which are then used as input for the zero-shot models, there are important trade-offs when choosing which techniques are suitable for zero-shot learning. In the following, we discuss these aspects.

First, a zero-shot cost model should be able to predict query runtimes on databases that were not seen before without relying on an observed workload on that database. Since workload-driven models for cardinality estimation require such queries as training data, they are not suited for our purpose of predicting cardinalities for zero-shot models. Second, traditional histogram-based approaches have the advantage that no additional efforts are required since the query optimizers anyway have built-in techniques. However, they are often imprecise. Third, data-driven models are more precise but also need to be trained. However, the training does not rely on query executions and is thus usually just in the order of minutes. Unfortunately, state-of-the-art data-driven cardinality estimators do not yet support the same variety of different queries as traditional approaches.

Hence, we have two options to supply zero-shot cost models with intermediate cardinalities. First, we can train a data-driven model, which results in more accurate cardinality as input to a zero-shot model and thus also cost estimates. However, if the effort of training a data-driven model for a new database is not acceptable or the workload is not supported by data-driven learning, we can fall back to the cardinality estimates of the query optimizer. In our evaluation, we will demonstrate that zero-shot models can still produce reasonable estimates even if only cardinalities estimates from traditional models are available.

12.4 ROBUSTNESS OF ZERO-SHOT MODELS

An important question for zero-shot models is at which point a sufficient amount of different training databases (and workloads) was observed to generalize robustly to unseen databases. As discussed in Section 12.2.3, a minimum requirement is to have sufficient coverage of the training data for the expected queries in the evaluation workload. For instance, all operators should be observed at least once and the number of joins, group by attributes as well as databases used for pre-training etc. should be representative as well. However, it is still interesting to provide an estimate of how precise the zero-shot cost models will be for unseen databases and what can be done in cases of more severe workload drifts which we will discuss below.

12.4.1 *Estimating the Generalization Performance*

We first formalize the problem, before we derive a method to estimate the generalization error. For training a zero-shot model, we have observed n databases and workloads. In particular, for each of the databases D_i we have access to training data T_i in the form of query plans and their runtimes $T_i = \{(P_1, c_1), (P_2, c_2), \dots (P_m, c_m)\}$. We are now interested in how accurately the zero-shot cost model Z will predict the runtimes for plans T^* on some unseen database D^*. In particular, if the expected error is acceptable, we have observed a sufficient amount of databases and workloads. More formally, we will define some error metric $E(T_i)$ with which we can compare the true runtimes and model predictions for some database D_i. An example for such a metric could be the prominently used median Q-error. We are now interested in estimating this error metric for an unseen database $E(T^*)$, i.e., the expected generalization error.

We now make use of statistical techniques to estimate the generalization error. For instance, in ML it is standard practice to train the model on a subset of the data and then use the remaining samples to estimate the error for future unseen datasets. Analogously, we can train the zero-shot model on a subset of the training databases $T_1, T_2, \dots, T_i$ (i.e., for a subset of databases) and evaluate the trained model on the remaining databases $T_{i+1}, \dots, T_n$. Similar to cross validation, we can repeat this procedure with different splits and average the test errors to estimate the generalization error $E(T^*)$, i.e., how accurate the model is expected to be on an unseen database. Interestingly, this is an unbiased estimator of the test error $E(T^*)$ under the independent identically distributed (i.i.d.) assumption, which we will discuss shortly. Hence, using only the observed databases and queries, we can estimate how accurate the model predictions for unseen databases will be.

In order to now evaluate whether the model has observed a sufficient amount of databases and workloads, we can use two techniques. First, we can simply estimate the generalization error as described above and stop the training if it is sufficient. However, in this case we have to decide which generalization error is acceptable. A second technique (which we actually use in this paper) is to estimate if additional training databases will improve the generalization performance. For this, we train the model on subsets of all training databases. If the estimated generalization error $E(T^*)$ does not improve significantly for a larger number of training databases, we can conclude that additional databases will not improve the generalization capabilities of the zero-shot cost model and thus stop the training data collection.

12.4.2 *Tackling Workload and Data Drifts*

The performance of zero-shot cost models will deteriorate if the new database and workload is significantly different from the training data. While data drifts can be handled by providing up-to-date cardinality estimates (from simple or data-driven models) as we show in our experiments, workload drifts need a more careful handling. For instance, if there are significantly larger joins in the unseen database than for the training databases, the zero-shot model might not be able to predict the runtime with the same high accuracy. As we will show in our experimental evaluation, however, zero-shot cost models can often still generalize robustly in practice and can provide more accurate estimates than other baselines in case of workload drifts. In addition, we suggest a strategy to detect cases of workload drifts by monitoring the test error and propose to tackle workload-drifts using few-shot learning.

Note that in cases of workload drifts the i.i.d. assumption does not hold and the Q-error on the unseen database is larger than implied by the generalization error. More technically, the i.i.d. assumption is a common assumption in ML that requires that the training datasets and test datasets are independent samples of some distribution $\mathcal{D}$. Due to a workload drift, the samples are no longer independent and thus the generalization error $E(T^*)$ might be increased. A simple yet effective strategy to recognize those cases is thus to monitor the error for unseen databases during inference. In cases where the error exceeds a certain threshold, one could decide to fine-tune the zero-shot model using the additional observed queries as training data (resulting in few-shot models). We will demonstrate in the experimental evaluation that zero-shot cost models fine-tuned on a small number of additional queries can significantly improve the accuracy on the unseen database in such cases.

12.5 A NEW BENCHMARK

In order to properly train and evaluate cost models, we require both a diverse set of databases and executed workloads on these databases. Since currently there is no suitable benchmark with such properties, we created a new benchmark (that includes existing benchmarks such as JOB), which we discuss in this section. Furthermore, we will make this benchmark publicly available to foster future research in this area.

12.5.1 *Design Decisions*

For many years, DBMS systems were evaluated using synthetic benchmarks such as TPC-H [169], TPC-DS [123] or SSB [128]. While such benchmarks allow to evaluate the general system performance and

scalability, they are in isolation insufficient to evaluate cost prediction models since the predicted cardinalities of the query optimizer are significantly more accurate than in practice. The reason is that the data is synthetic and thus no interesting correlations have to be captured making cardinality estimation challenging in practice. Hence, Leis et al. [93] suggested the JOB-workload on the IMDB dataset that comes with challenging correlations and has become the standard method (along with the simplified JOB-light workload [79]) to evaluate learned cost and cardinality models.

While the IMDB benchmark is useful to evaluate workload-driven cost estimators that need to work on a single database only, it cannot be used for the evaluation of zero-shot cost models since these have to be trained on a variety of different databases. Moreover, even for workload-driven cost estimators a benchmark that spans a more diverse set of databases would definitively be helpful to evaluate the prediction quality. Hence, we decided to create a new benchmark that covers established datasets such as IMDB but also additional datasets that have other characteristics.

12.5.2 *Datasets*

As discussed before, it is insufficient to just add synthetic datasets since correlations hardly resemble data distributions found in the real-world. We thus decided to leverage *publicly available real-world datasets* [120] together with the *datasets used in established benchmarks such as JOB*. Since certain databases were very small in size, we additionally scaled them to larger sizes to be interesting for cost estimation (s.t. a sample of queries takes a predefined threshold of time). In addition to the datasets mentioned before, we also include data and workloads of existing benchmarks such as SSB and TPC-H. As these benchmarks rely on synthetic data, this further increases the variety of data characteristics our benchmark covers for testing learned cardinality estimators. Overall, the benchmark comprises of 20 databases that vary largely in the number of tables, columns and foreign-key relationships.

12.5.3 *Workloads and Traces*

Furthermore, for benchmarking learned cost models, workloads are required for training and testing. To simplify the comparison with prior work we first include predefined benchmark queries for databases that come with such workloads (e.g., JOB for IMDB). However, since for the majority of the databases mentioned before no workloads are available, we implemented a workload generator that generates different types of queries. For creating the workload, the generator supports three modes: a *standard* mode where Select-Project-Aggregate-Join (SPAJ) queries with conjunctive predicates on numeric and categorical

columns similar to the ones used by Kipf et al. [79] are generated, a more *complex* mode, which includes predicates involving disjunctions, string comparisons with regex predicates, `IS (NOT) NULL` comparisons and `IN` operators (resembling the complexity of the JOB-workload) and finally an *index* workload where random indexes (both foreign key and for predicate columns) are created throughout the execution of the standard workload, which is challenging due to the varying physical designs. Since the benchmark will be publicly available it can be easily extended in the future.

In addition to the datasets and the workload generator, the benchmark comes with workload traces (e.g., executions of the queries and their runtime) for all 20 databases that can be used directly by other researchers as training / testing data (which we also used in our evaluation). To be more precise, we generated 15,000 queries per database and executed those queries on a Postgres DBMS (v12) on `c8220` nodes on the cloudlab platform. Overall, this also allows for a better reproducibility since this platform can be used by other researchers as well. To limit the already excessive resource consumption required to produce this trace, we excluded queries running longer than 30 seconds from the benchmark for all workloads. In total, the execution of these more than 300k queries takes 10 days if executed on a single node. As part of the traces, we not only provide the runtime of the query but also the physical plan used to run the query along with actual cardinalities.

12.6 EXPERIMENTAL EVALUATION

In this Section, we evaluate zero-shot cost estimation with a set of different experiments:

- **Exp 1. Zero-Shot Accuracy on Unseen Databases.** We evaluate how accurately zero-shot cost models can predict costs for unseen databases.
- **Exp 2. Zero-Shot vs. Workload-Driven.** In addition, we compare the training overhead and accuracy with state-of-the-art workload driven approaches.
- **Exp 3. Generalization.** In this experiment, we study how our models generalize under workload drifts (i.e., under database updates and larger unseen joins).
- **Exp 4. Training and Inference Performance.** Furthermore, we evaluate the training and inference performance of zero-shot cost models and compare training efforts to workload-driven models.
- **Exp 5. Ablation Studies.** Finally, we show the effects of different design alternatives of zero-shot models as well as a study where

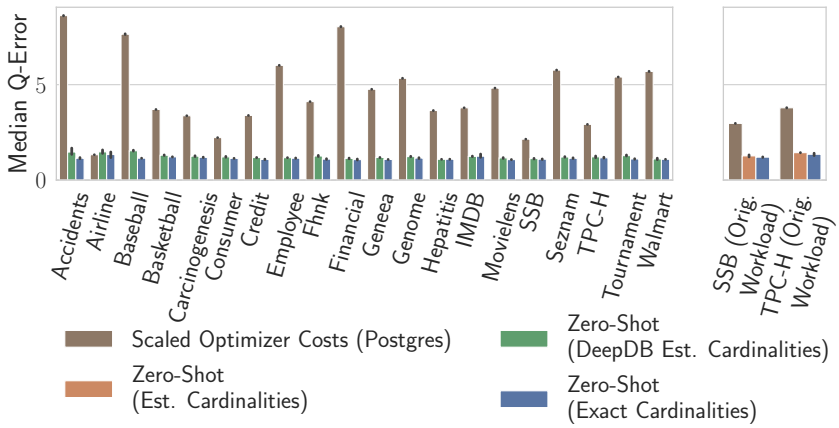


Figure 12.4: Zero-Shot Generalization across Databases. The zero-shot models are trained using workloads on 19/20 databases and tested on the remaining unseen database. Overall, zero-shot models are significantly more accurate than the scaled estimates of the optimizer cost model. In addition to using workloads as defined by our benchmark (left), we repeated this experiment with standard benchmark workloads (SSB and TPC-H on the right) to further show the generalization potentials of zero-shot cost models.

we determine how many database are sufficient for zero-shot cost models to generalize.

For all experiments, we use the traces of the benchmark discussed before (for training and testing).

12.6.1 *Exp 1: Zero-Shot Accuracy on Unseen Databases*

First, in order to evaluate the accuracy of zero-shot cost models, we trained a zero-shot model using workloads on 19 out of the 20 datasets of the benchmark as training data and evaluated the model on the workload of the unseen (remaining) database. In particular, we use the trained model to predict the runtimes of the queries on the unseen database and report the median Q-error. In the first experiment, we focus on the *standard* workloads and defer the results of the *complex* and *index* workloads of our benchmark to follow-up experiments. For this experiment, we ran each setup three times using different seeds for the cost estimation for every unseen database.

For showing the performance of zero-shot cost models on unseen databases, we used two variants of providing intermediate cardinalities - we either used predictions of learned cardinality estimators or the actual cardinalities, which are not available in practice prior to execution but serve as an interesting upper baseline for zero-shot learning (i.e., how accurate the predictions become with perfect cardinality estimates). For the data-driven cardinality estimator, we trained DeepDB [72] models, which worked best in preliminary experiments. To the best of our knowledge, we are the first to propose zero-shot cost estimation and thus no other learned approaches are included

as a direct baseline in this first experiment where we aim to analyze the accuracy on unseen databases. For instance, workload-driven approaches would need query executions on the unseen database, which we do not provide in the zero-shot setting. However, we compare our approach with workload-driven models in Section 12.6.2.

As a sanity check that zero-shot models provide better performance than classical cost estimation models that rely on simple (non-learned) techniques (and as such could also count as zero-shot cost models), we use cost estimates coming from the Postgres query optimizer as a baseline similar to previous work [161]. Moreover, for the distributed setup we later on also employ the cost estimates of a commercial cloud DBMS. Since Postgres cost estimates are provided as abstract cost units, we use a simple linear model on top of Postgres estimates (and hence the results are called *Scaled Optimizer*), which provides actual query runtimes. Different from [161], which directly take the cost units as runtime (in ms), using a linear model on top results in a much lower Q-error for Postgres. For training the simple linear model we are using the same training data from the other 19 databases as for zero-shot models to be fair.

The results can be seen in Figure 12.4. In general, the zero-shot models offer robust performances for all of the databases despite the varying complexity. In fact, all median Q-errors are below 1.54 for the version using DeepDB cardinality estimates (vs. 8.62 in the worst case for the *Scaled Optimizer* cost). Finally, we can see that zero-shot cost models using DeepDB cardinalities are almost matching the performance with perfect cardinalities. This suggests that the models can cope with partially inaccurate cardinalities. Indeed, as we will see in a follow-up experiment, this even holds when we use potentially inaccurate cardinality estimates coming from a classical optimizer instead.

Overall, we can see that the zero-shot cost models are significantly more accurate than the scaled optimizer estimates outperforming these on 18 out of 19 datasets and being on par for the last remaining dataset (Airline). The reason is that zero-shot cost models capture subtleties in operator performance and interactions of operators in the plan more accurately than simplistic cost models. The results are just on par for the remaining database since the optimizer costs are relatively accurate because it is merely a star schema, i.e., a relatively simple schema structure.

To demonstrate that zero-shot cost models also improve the estimates for workloads of traditional benchmarks, we repeat the previous experiment with the original benchmark queries of SSB and TPC-H.[2] Again we train on 19 out of 20 datasets (excluding either SSB or

2 For SSB, we used all queries as-is. For TPC-H, since our current implementation does not support the `subplan` operator of Postgres, we rewrote subqueries using joins. However, we believe that our approach can also be extended to support subqueries.

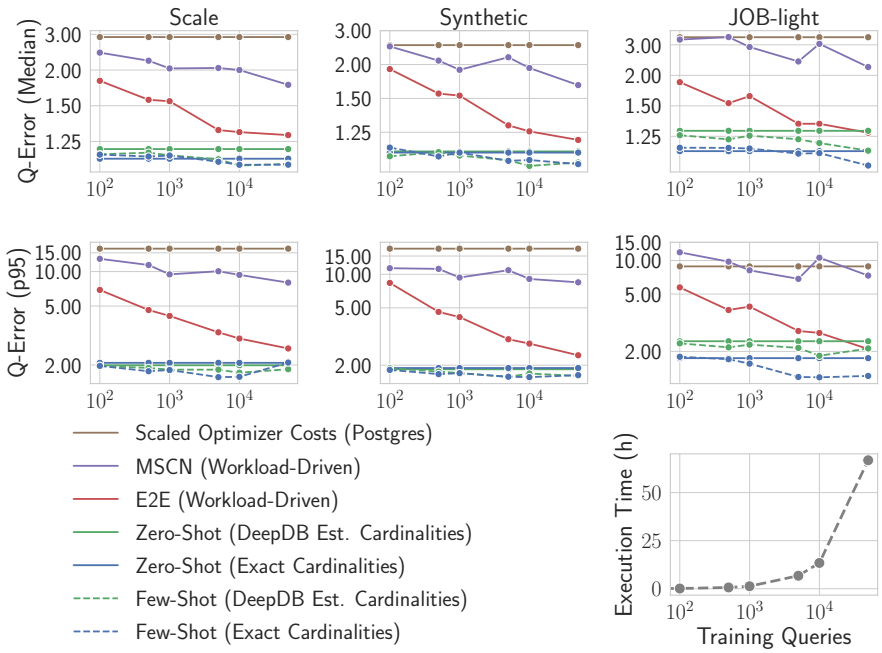


Figure 12.5: Estimation Errors of Workload-Driven Models for a varying Number of Training Queries compared with Zero-Shot Cost Models. Even the most accurate workload-driven model (E2E) requires approximately 50k query executions on an unseen database for a comparable performance with zero-shot models, which is roughly equivalent to 66 hours of executed workload. Since zero-shot models do not require any additional queries it is significantly cheaper to deploy them for a new database. However, zero-shot models can be fine-tuned to obtain few-shot models, which further improve the accuracy.

TPC-H) and show the median Q-errors of both the baseline and our approaches. Note that DeepDB does not support all operators in TPC-H and thus we use Postgres cardinality estimates (orange bar in Figure 12.4) instead. As we can see, the results are very similar to the results using our new benchmark for zero-shot cost estimation providing additional evidence that zero-shot cost estimation can improve the cost estimation accuracy on queries from standard benchmarks as well.

12.6.2 *Exp 2: Zero-Shot vs. Workload-Driven*

In the following, we contrast the performance of zero-shot cost models with workload-driven approaches.

TRAINING OVERHEAD. An interesting question is how many training queries are required for workload-driven learning on an unseen database to match the performance of zero-shot learning, which we will study next. In particular, in this experiment we evaluate the Q-errors for the scale, synthetic, and JOB-light workloads (IMDB). As

before zero-shot models are not trained on IMDB at all (but on the other 19 databases) while workload-driven models are trained on a varying number of training queries on IMDB.

For the workload-driven approaches we use the E2E model proposed by Sun and Li [161] as well as the MSCN model by Kipf et al. [79]. The idea of the E2E models is to featurize the physical query plans and feed them into a neural model to predict the runtime. However, in contrast to zero-shot cost models the query plan representation is not transferable and thus the train and test databases have to be identical. The MSCN model, which was initially developed for cardinality estimation uses a more high level representation and encodes the sets of joins, predicates and tables of a query, which are then fed into a neural architecture, which is thus oblivious of the physical plans used. Both models are trained on a varying number of training queries, which are generated for the IMDB dataset similar to the original training setup used by Sun and Li [161]. Furthermore, as a last baseline, we again employ the scaled costs of the Postgres query optimizer.

In Figure 12.5, we depict the median Q-error of comparing our zero-shot performance to the baselines as discussed before for the IMDB benchmark workloads for a varying number of training queries. As we can see the zero-shot cost models can estimate the runtimes accurately even though queries on the IMDB dataset were not observed in the training data. In particular, E2E requires about 50k training queries on the IMDB database to be on-par with zero-shot cost models. As we can see in the lower right plot in Figure 12.5 this amount of queries takes approximately 66 hours to run, which is a significant effort given that it has to be repeated for every new database. Another interesting comparison is to use the training queries also to fine-tune the zero-shot models on the IMDB database; i.e., we use zero-shot models in the few-shot mode discussed in the paper. As we can see, few-shot cost models that are fine-tuned on the IMDB database can further improve the cost estimation accuracy of zero-shot models. It is thus beneficial to also leverage fine-tuning in case training queries for the unseen database are available.

Finally, we can see that the MSCN models are not equally accurate, which is likely due to the fact that they do not consider the physical plan that was run to execute a given query. Still, all learned approaches are more accurate than the scaled optimizer in the median after only a few queries. Furthermore, we can observe that zero-shot and few-shot cost models not only outperform workload-driven models in the median but also in the tail performance, i.e., on the 95th percentile Q-error. We can observe similar effects for the maximum Q-error.

COMPLEX QUERIES. In this experiment, we next focus on the performance for complex queries. For this, we again train on 19 datasets and test on the IMDB database (this time using the *complex* benchmark queries) using the JOB-Full benchmark, which (different from the

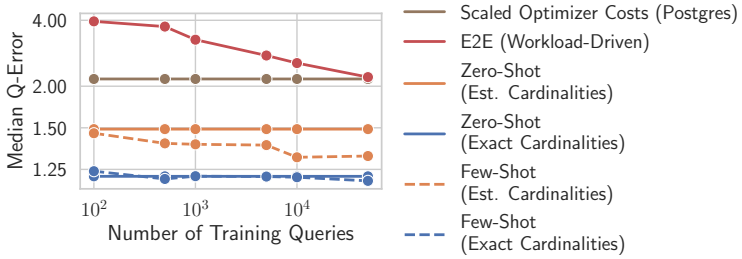


Figure 12.6: JOB-Full Workload. Zero-shot models are significantly more accurate than the workload-driven model (E2E) and the scaled optimizer estimates even for the complex JOB benchmark. Again few-shot learning can further improve the performance of zero-shot models.

other workloads on IMDB) contains also queries with a higher number of joins and more complex predicates including pattern-matching queries on strings. Note that data-driven models do not support complex predicates and we thus resort to the cardinality estimates of the query optimizer (Postgres) to inform the zero-shot model. As baselines, we again compare to the scaled optimizer costs and E2E, which in contrast to MSCN supports complex predicates. To be fair, we use training queries with complex predicates on IMDB for the workload-driven models. In addition, we also report the accuracy of zero-shot models fine-tuned on the IMDB database using the few-shot learning.

As we can see in Figure 12.6, again zero-shot models outperform the other approaches. In particular, even the version using just optimizer cardinality estimates is more accurate than E2E using 50k queries, which emphasizes that zero-shot cost models are robust w.r.t. imprecise cardinality estimates. The E2E models in this case need 50k queries just to match the performance of the scaled optimizer costs, which is inferior to the previous experiment with a lower query complexity. The reason is that the E2E model has to learn the data distribution of strings as well and support complex predicates including wildcards while only observing queries. We hope that in the future, data-driven models support string predicates and disjunctions as well to be used in conjunction with zero-shot cost models also for complex queries. Similar to the previous experiment, few-shot learning can further improve the accuracy.

12.6.3 *Exp 3: Generalization*

In this experiment, we investigate how robustly zero-shot cost models react to changes in the data characteristics and workload.

GENERALIZATION TO UPDATES. For the first aspect, we analyze the effects of updates on the accuracy of cost estimation. For this, we only train on a fraction of the full data and then update the database (without retraining the prediction models). After the update of the

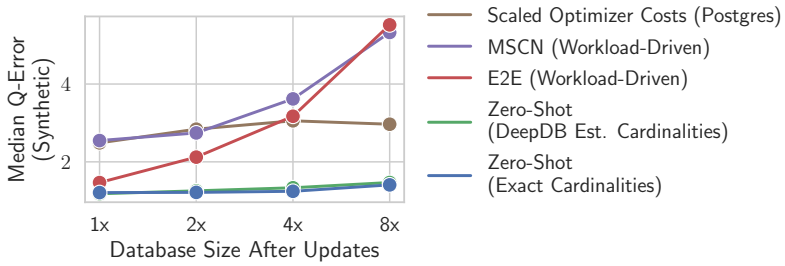


Figure 12.7: Zero-Shot Models are robust w.r.t. Updates. Without any retraining we do not see regressions in cost estimation accuracy even for massive update rates (up to 8 times the size of the original database using rescaling). In contrast, workload-driven models require additional training queries.

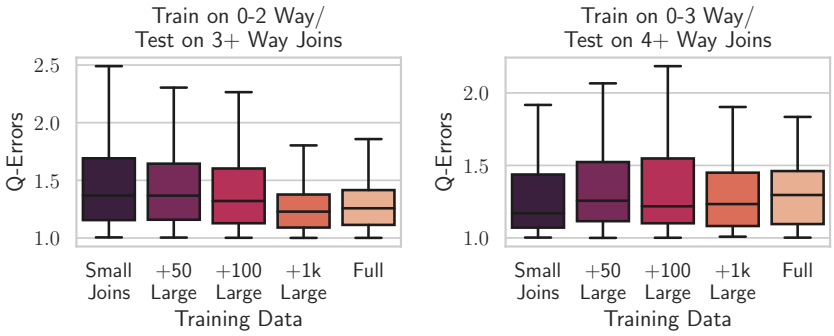
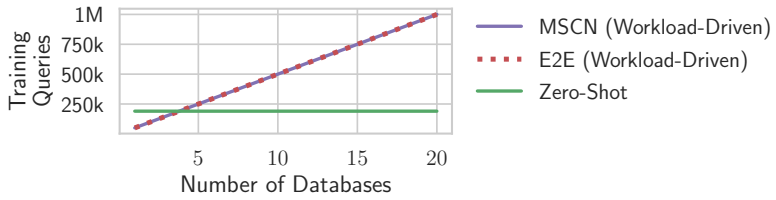


Figure 12.8: Zero-shot cost models generalize robustly to larger Joins. Compared to zero-shot models trained also on larger joins (Full), the zero-shot models trained only on smaller joins (Small Joins) have only minor regressions in accuracy. In addition, fine-tuning the zero-shot cost models on a low number of additional queries with larger joins (resulting in few-shot models) further improves the performance.

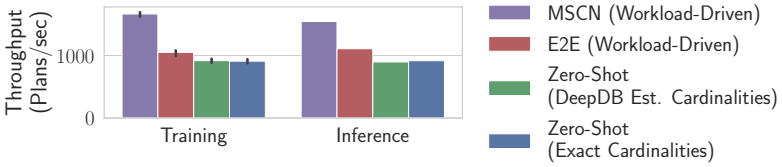
database, we then predicted the query runtimes using zero-shot cost models as well as the other baselines (workload-driven models and the scaled optimizer). Note, that workload-driven models are expected to result in inferior performance for a higher fraction of updates since they cannot capture database updates without collecting new training data. This is very different from zero-shot models that get informed by data-driven models that can thus adjust to data updates without the need to retrain. In particular, the data-driven models from DeepDB [72] as well as classical statistics such as histograms that are compatible with zero-shot cost models are directly updateable with low overhead and hence can provide also accurate estimates after the update.

We depicted the results in Figure 12.7. As we can see, there is almost no performance degradation for the zero-shot cost models with a higher update fraction. Note that we did not retrain the zero-shot cost models at all to achieve the performance but simply relied on the ability to generalize to different data characteristics. In contrast, for workload-driven models we observe a performance degradation since those models would require additional training queries on the updated database to be adapted. The reason is that the models also internalize the data distribution (i.e., table sizes and correlations) implicitly during the training and can only be informed about changes by observing additional query runtimes. This is especially problematic for more update-heavy workloads were frequently additional training queries have to be run to update the models. Note that the scaled optimizer costs do not experience such a degradation but are again less accurate than zero-shot models.

GENERALIZATION TO WORKLOAD DRIFTS. In this experiment, we investigate how zero-shot models react to workload drifts, in particular to larger joins that appear after training a cost prediction model. To this end, we trained the zero-shot models using only queries with up to 2 or 3-way joins on the 19 training datasets and evaluate the model using 3-way or 4-way joins (or larger) on the IMDB dataset, respectively. Since we suggest to address workload-drifts using few-shot learning, we also introduce variants that are fine-tuned on a small amount of large joins on the IMDB database. As we can see in Figure 12.8, the performance of the model with a training set constrained to small joins does not degrade heavily compared to the model that was also trained on larger joins on the remaining 19 datasets (Full) indicating a robust generalization to larger joins. In addition, few-shot models fine-tuned on a small amount of larger joins (≈ 50) observed on the IMDB dataset is sufficient to achieve the same median Q-error. An even larger amount of retraining queries allows to outperform the original zero-shot model, which is consistent with previous experiments showing that few-shot learning further improves the accuracy.



(a) Required Training Queries.



(b) Train and Test Throughput.

Figure 12.9: Training and Inference and Performance. Even though zero-shot models generalize across databases they almost match the inference and training throughput of the most accurate workload-driven alternative (E2E) and quickly amortize in terms of required training queries.

12.6.4 *Exp 4: Efficiency of Training and Inference*

In this experiment, we evaluate the efficiency of training and inference of zero-shot models compared to workload-driven models.

TRAINING OVERHEAD. In a first experiment, we compare the number of training queries required for zero-shot models as well as for workload-driven models. Importantly, workload-driven models need to be trained on every single database while zero-shot models can (once trained) be applied to many different databases out-of-the-box. For showing this effect we analyze how many training queries would be required for supporting a varying number of unseen databases for which new cost estimates are required The results are shown in Figure 12.9a. As we can see since workload execution is a one-time effort for zero-shot models (since they generalize across databases) this quickly amortizes compared to workload-driven learning since for workload-driven models, we need to collect training data for every new database.

TRAINING AND INFERENCE THROUGHPUT. In a second experiment, we compare the training and inference throughput of zero-shot cost models with state-of-the-art workload-driven approaches. In this experiment, we aim to show that zero-shot models are not imposing higher overhead for training and inference and thus can be used efficiently in real DBMSs. As we can see in Figure 12.9b, zero-shot models achieve a comparable throughput and thus do not impose higher overhead compared to workload-driven models. As we can see, the MSCN models achieve higher throughput compared to all other models (zero-shot and E2E). The reason is that these models featurize

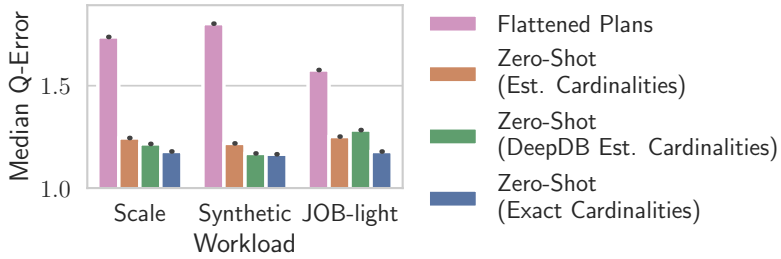


Figure 12.10: Ablation Study. Using a flattened representation of the plans instead of our graph-based encoding yields less accurate models. Zero-shot models using the cardinality estimates of the query optimizer are still reasonably accurate.

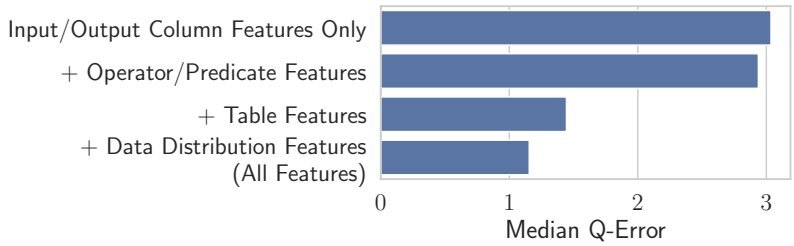


Figure 12.11: Feature Ablation Study. All groups of query graph features used in zero-shot cost models individually improve the accuracy. Table and data distribution features have the largest impact since they determine the scan cost and size of intermediate joins, respectively.

the physical query plan resulting in larger graphs compared to MSCN models, which only encode the joins, tables and predicates in a query. However, this comes at the cost of an inferior predictive performance as shown before.

12.6.5 *Exp. 5: Ablation Study*

In this experiment, we present the results of our ablation study showing the effects of the different design choices as well as the efficiency of

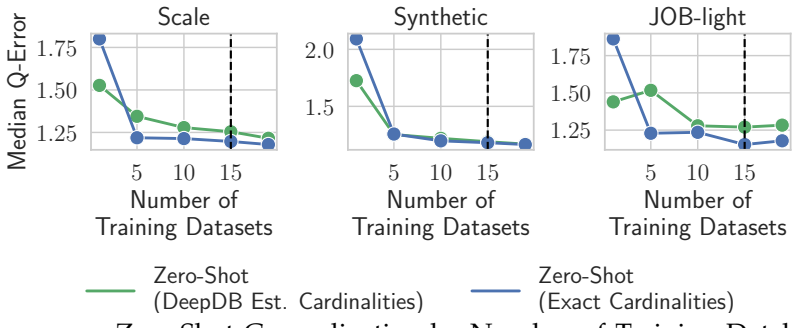


Figure 12.12: Zero-Shot Generalization by Number of Training Databases. If we use more than 15 training databases we start to see diminishing returns in accuracy suggesting that the variety of databases in the benchmark is sufficient.

our estimator to determine how many different databases are needed for training a zero-shot model.

ZERO-SHOT DESIGN SPACE. We first explore the different design space options of zero-shot cost estimation. In particular, we focus on the questions how different cardinality estimation techniques impact the model accuracy and whether our new model architecture using graph encodings is actually required or a simpler architecture suffices.

To address the latter question, we implemented a different version of zero-shot cost estimation that represents a single query plan as a flat vector (instead of using a graph). In particular, the chosen representation is similar to Ganapathi et al. [49] that represents a query plan using a vector where each physical operator corresponds to two entries in the vector: one that counts how often the operator appears in the plan and one that sums up the cardinality estimates for that operator. For instance, if we only had sequential scans and nested loop joins in the query plans and one plan would scan two relations of 1M tuples each and join them resulting in 1M tuples, the vector representing the query plan could be $(2, 2M, 1, 1M)$. Given this representation, we train a state-of-the-art regression model [77] to predict the runtime given a vector. Similar to the zero-shot models, we train on the remaining 19 datasets and evaluate the performance on the IMDB benchmarks.

As we can see in Figure 12.10, the flattened version of zero-shot cost models is significantly less accurate than our proposed transferable graph-based representation. The reason is that the interactions of physical operators in the plan can only be modeled approximately if represented as a vector while our graph-based encoding allows the neural model to capture such interactions more accurately. Second, regarding cardinality estimates, we can see that data-driven cardinality estimates improve the accuracy of zero-shot cost models compared to models using optimizer cardinality estimates. However, the estimates are still very accurate even if cardinality estimates are annotated from simple cardinality estimation models that are used in DBMSs today. This is especially useful for query types that data-driven models do not support as of today and where the optimizer cardinality estimates hence serve as a fallback.

QUERY GRAPH FEATURIZATION. In addition, we also study the impact of different featurizations of the query graph representations used by zero-shot cost models. In particular, instead of training zero-shot cost models using all the features introduced in Table 12.1, we will gradually include more groups of features (e.g., all features related to scanned tables). We then report the median Q-error achievable with this set of features. As we can see in Figure 12.11, each group of features individually improves the performance of the models. The most significant improvement is due to features characterizing the

tables as well as the data distribution (e.g., cardinalities) as these features influence the runtime overhead of a query most significantly. However, we can conclude that all features are worth incorporating in zero-shot models as long as they are transferable as described in Section 12.3. The rationale is to include as many aspects that could impact the query runtime as possible in the query representation.

NUMBER OF TRAINING DATABASES. As described in Section 12.4.1, in order to assess whether a zero-shot cost model has seen a sufficient number of training databases and workloads, we estimate the expected generalization error for a varying number of training databases. The generalization error is estimated by computing the test error on an unseen holdout database. If the model performance plateaus for a certain number of training databases, we can conclude that the number of training databases is sufficient.

In this experiment, we show how the generalization error develops for a growing number of training databases (i.e., from just using one up to all 19 databases). For estimating the generalization error, we use the standard benchmark queries as defined on the IMDB dataset (i.e., we use the synthetic, scale and JOB-light [79] workloads). As we can observe in Figure 12.12, as expected the generalization errors reduce with a growing number of databases. This is the case because with an increased number of databases the model can observe a larger variety of different data characteristics and can thus more robustly predict the runtimes for an unseen database, i.e., IMDB in this case. Interestingly, we can already achieve a reasonably small generalization error after just five different databases indicating that a moderate number of databases can be sufficient for zero-shot learning. Moreover, we clearly observe diminishing returns between 15 to 19 databases.

We can thus conclude that the number of training datasets from the benchmark is indeed sufficient to allow a zero-shot model to generalize robustly to unseen databases from the benchmark and that further datasets will likely not improve the model performance.

WORKLOAD & DATA DRIFTS. While zero-shot cost models generalize to some extent under workload and data drifts as demonstrated in Section 12.6.3, there is clearly a point where the evaluation workload differs too severely from any workload observed at training time by the model and where the performance will degrade as discussed Section 12.2.3. In a final ablation study, we thus want to investigate further how quickly the performance degrades in such cases and thus intentionally create evaluation workloads that differ largely from the training workload (due to significantly more joins, group by attributes, number of predicates, aggregations or larger dataset sizes). As we can see in Figure 12.13, the more different the test database and workload is from the training workload and data, the more the performance degrades. However, while the zero-shot model can still predict costs relatively accurately for cases close to training data examples (e.g., five

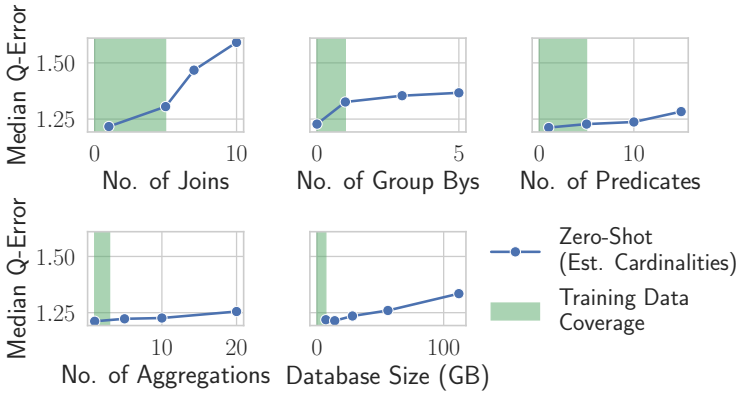


Figure 12.13: Accuracy under Workload and Data Drifts. Training data coverage is shown in green (e.g., the training data contains 0-5-way joins and we generalize up to 10-way joins). While zero-shot cost models generalize reasonably well to unseen workload and data characteristics, we see an increase in estimation errors for more severe drifts as expected.

aggregates instead of one) the performance degrades as the characteristics are further changed. Moreover, not all aspects have an equal impact on the accuracy: for example, the number of joins seems to be the most severe factor, which is expected, since more joins can result in large intermediate join sizes the model has not experienced before. Note, that in this experiment we report the results of zero-shot models using cardinality estimates of the Postgres optimizer but we could observe similar effects for the other methods as well.

12.7 RELATED WORK

LEARNED COST ESTIMATION. Closest to our work are workload-driven approaches for cost estimation. Neural predictions models [113, 161] have been proposed for cost estimation by featurizing the physical query plan as a tree. However, the models are workload-driven and thus require thousands of query executions for an unseen database. Recently, a framework has been proposed to efficiently gather this training data [173]. In contrast, zero-shot learning completely alleviates the need to run a representative workload for new databases. Moreover, workload-driven models were extended by improving inference and training performance [76] and to concurrent query latency prediction [194]. These ideas are orthogonal and could potentially be applied to zero-shot learning as well. An alternative to reduce the required training queries for cost estimation is DBMS fitting [67] where the idea is to model the operator complexity and adjust this basic model by fitting the parameters using differentiable programming. However, the operator complexity has to be modeled explicitly, which can be impossible for complex queries.

Earlier work proposes to use statistical methods to predict the costs of queries. For instance, it was proposed to learn models at a per-

operator level [5, 96] to predict the overall query runtime. However, since interactions of operators cannot be learned and the models are thus too simplistic, the performance is inferior to workload-driven approaches [113]. An alternative idea is to represent query plans as flat vectors [49] to treat cost estimation using supervised regression, which we have shown to be less accurate than zero-shot cost estimation (cf. Section 12.6.1). In addition, it was suggested to leverage query executions on smaller data samples [44, 172] or queries sharing common subexpressions [155, 180] for cost estimation. In both cases, the test workload needs to closely resemble the train workload for the models to be effective.

LEARNED DBMS COMPONENTS AND DESIGN ADVISORS Machine learning has been applied more broadly to optimize DBMS systems by replacing traditional approaches for tasks such as query optimization [85, 109, 110, 112] or query scheduling [107, 153]. In addition, it was applied to knob tuning [192], materialized view selection [59, 99], index selection [89] or partitioning [70]. Note that all these approaches are workload-driven since query executions on the test database are required to train the models.

12.8 CONCLUSION AND FUTURE WORK

In this paper, we have demonstrated that it is possible to accurately and robustly predict query runtimes on entirely unseen databases, i.e., in a zero-shot setting. In addition, fine-tuning the zero-shot models to obtain few-shot models can further improve the performance if training queries are available on the new database. We enabled this by deriving a transferable representation of queries that generalizes across databases and a specialized model architecture.

As a future direction, we argue that zero-shot learning even has a much broader applicability and could be applied to a large set of learned DBMS components including design advisors etc. Furthermore, we believe that the underlying principles can be applied to an even broader set of data systems (e.g., data streaming systems).

12.9 ACKNOWLEDGMENTS

We thank the reviewers for their feedback. This research is funded by the BMBF project within the "The Future of Value Creation – Research on Production, Services and Work" program, the Hochtief project *AICO* (AI in Construction), the HMWK cluster project *3AI* (The Third Wave of AI), as well as the DFG Collaborative Research Center 1053 (MAKI). Finally, we want to thank hessian.AI at TU Darmstadt as well as DFKI Darmstadt.

BIBLIOGRAPHY

[1] Serge Abiteboul, Paris C. Kanellakis, and Gösta Grahne. "On the Representation and Querying of Sets of Possible Worlds." In: *Theor. Comput. Sci.* 78.1 (1991), pp. 158–187. DOI: 10.1016/0304-3975(51)90007-2. URL: https://doi.org/10.1016/0304-3975(51)90007-2.

[2] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. "BlinkDB: queries with bounded errors and bounded response times on very large data." In: *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*. Ed. by Zdenek Hanzálek, Hermann Härtig, Miguel Castro, and M. Frans Kaashoek. ACM, 2013, pp. 29–42. DOI: 10.1145/2465351.2465355. URL: https://doi.org/10.1145/2465351.2465355.

[3] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollár, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. "Database Tuning Advisor for Microsoft SQL Server 2005." In: *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*. Ed. by Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer. Morgan Kaufmann, 2004, pp. 1110–1121. DOI: 10.1016/B978-012088469-8.50097-8. URL: https://doi.org/10.1016/B978-012088469-8.50097-8.

[4] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. "Automated Selection of Materialized Views and Indexes in SQL Databases." In: *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*. Morgan Kaufmann, 2000, pp. 496–505. URL: http://www.vldb.org/conf/2000/P496.pdf.

[5] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B. Zdonik. "Learning-based Query Performance Modeling and Prediction." In: *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*. Ed. by Anastasios Kementsietsidis and Marcos Antonio Vaz Salles. IEEE Computer Society, 2012, pp. 390–401. DOI: 10.1109/ICDE.2012.64. URL: https://doi.org/10.1109/ICDE.2012.64.

[6] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. "Automatic Database Management System Tuning Through Large-scale Machine Learning." In: *Proceedings of the*

2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017. Ed. by Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu. ACM, 2017, pp. 1009–1024. DOI: 10.1145/3035918.3064029. URL: https://doi.org/10.1145/3035918.3064029.

[7] Sami Alabed and Eiko Yoneki. "High-Dimensional Bayesian Optimization with Multi-Task Learning for RocksDB." In: *EuroMLSys@EuroSys 2021, Proceedings of the 1st Workshop on Machine Learning and Systemsg Virtual Event, Edinburgh, Scotland, UK, 26 April, 2021*. Ed. by Eiko Yoneki and Paul Patras. ACM, 2021, pp. 111–119. DOI: 10.1145/3437984.3458841. URL: https://doi.org/10.1145/3437984.3458841.

[8] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, TJ Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. "Amazon Redshift re-invented." In: *SIGMOD/PODS 2022*. 2022. URL: https://www.amazon.science/publications/amazon-redshift-re-invented.

[9] Filipe de Avila Belbute-Peres, Kevin A. Smith, Kelsey R. Allen, Josh Tenenbaum, and J. Zico Kolter. "End-to-End Differentiable Physics for Learning and Control." In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. Ed. by Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett. 2018, pp. 7178–7189. URL: https://proceedings.neurips.cc/paper/2018/hash/842424a1d0595b76ec4fa03c46e8d755-Abstract.html.

[10] *AWS Redshift*. https://aws.amazon.com/redshift. Accessed: 2020-09-12.

[11] *Azure SQL Data Warehouse*. https://azure.microsoft.com/services/synapse-analytics/. Accessed: 2020-09-12.

[12] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. "Neural Combinatorial Optimization with Reinforcement Learning." In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Workshop Track Proceedings*. OpenReview.net, 2017. URL: https://openreview.net/forum?id=Bk9mxlSFx.

[13] Philip A. Bernstein and Eric Newcomer. *Principles of Transaction Processing for Systems Professionals*. Morgan Kaufmann, 1996. ISBN: 1-55860-415-4.

[14] Simone Bova. "SDDs Are Exponentially More Succinct than OBDDs." In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*. Ed. by Dale Schuurmans and Michael P. Wellman. AAAI Press, 2016, pp. 929–935. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12270.

[15] Guy Van den Broeck and Adnan Darwiche. "On the Role of Canonicity in Knowledge Compilation." In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*. Ed. by Blai Bonet and Sven Koenig. AAAI Press, 2015, pp. 1641–1648. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9961.

[16] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. "Language Models are Few-Shot Learners." In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. Ed. by Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin. 2020. URL: https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

[17] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. "Language Models are Few-Shot Learners." In: *CoRR* abs/2005.14165 (2020). arXiv: 2005.14165. URL: https://arxiv.org/abs/2005.14165.

[18] Nicolas Bruno and Surajit Chaudhuri. "Automatic Physical Database Tuning: A Relaxation-based Approach." In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*. Ed. by Fatma Özcan. ACM, 2005, pp. 227–238. DOI: 10.1145/1066157.1066184. URL: https://doi.org/10.1145/1066157.1066184.

[19] Nadia Burkart and Marco F. Huber. "A Survey on the Explainability of Supervised Machine Learning." In: *J. Artif. Intell. Res.* 70 (2021), pp. 245–317. DOI: 10.1613/jair.1.12228. URL: https://doi.org/10.1613/jair.1.12228.

[20] Kaushik Chakrabarti, Minos N. Garofalakis, Rajeev Rastogi, and Kyuseok Shim. "Approximate Query Processing Using Wavelets." In: *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*. Ed. by Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang. Morgan Kaufmann, 2000, pp. 111–122. URL: http://www.vldb.org/conf/2000/P111.pdf.

[21] Kaushik Chakrabarti, Minos N. Garofalakis, Rajeev Rastogi, and Kyuseok Shim. "Approximate query processing using wavelets." In: *VLDB J.* 10.2-3 (2001), pp. 199–223. DOI: 10.1007/s007780100049. URL: https://doi.org/10.1007/s007780100049.

[22] Surajit Chaudhuri, Bolin Ding, and Srikanth Kandula. "Approximate Query Processing: No Silver Bullet." In: *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. Ed. by Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu. ACM, 2017, pp. 511–519. DOI: 10.1145/3035918.3056097. URL: https://doi.org/10.1145/3035918.3056097.

[23] Surajit Chaudhuri and Vivek R. Narasayya. "An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server." In: *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*. Morgan Kaufmann, 1997, pp. 146–155. URL: http://www.vldb.org/conf/1997/P146.PDF.

[24] Haipeng Chen, Sushil Jajodia, Jing Liu, Noseong Park, Vadim Sokolov, and V. S. Subrahmanian. "FakeTables: Using GANs to Generate Functional Dependency Preserving Tables with Bounded Real Data." In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*. Ed. by Sarit Kraus. ijcai.org, 2019, pp. 2074–2080. DOI: 10.24963/ijcai.2019/287. URL: https://doi.org/10.24963/ijcai.2019/287.

[25] Kaiji Chen, Yongluan Zhou, and Yu Cao. "Online Data Partitioning in Distributed Database Systems." In: *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015*. Ed. by Gustavo Alonso, Floris Geerts, Lucian Popa, Pablo Barceló, Jens Teubner, Martin Ugarte, Jan Van den Bussche, and Jan Paredaens.

OpenProceedings.org, 2015, pp. 1–12. DOI: 10.5441/002/edbt.2015.02. URL: https://doi.org/10.5441/002/edbt.2015.02.

[26] Xu Chu, Ihab F. Ilyas, Sanjay Krishnan, and Jiannan Wang. "Data Cleaning: Overview and Emerging Challenges." In: *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. Ed. by Fatma Özcan, Georgia Koutrika, and Sam Madden. ACM, 2016, pp. 2201–2206. DOI: 10.1145/2882903.2912574. URL: https://doi.org/10.1145/2882903.2912574.

[27] Yeounoh Chung, Michael Lind Mortensen, Carsten Binnig, and Tim Kraska. "Estimating the Impact of Unknown Unknowns on Aggregate Query Results." In: *ACM Trans. Database Syst.* 43.1 (2018), 3:1–3:37. DOI: 10.1145/3167970. URL: https://doi.org/10.1145/3167970.

[28] *CloudLab*. https://www.cloudlab.us/.

[29] Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. "Schism: a Workload-Driven Approach to Database Replication and Partitioning." In: *Proc. VLDB Endow.* 3.1 (2010), pp. 48–57. DOI: 10.14778/1920841.1920853. URL: http://www.vldb.org/pvldb/vldb2010/pvldb%5C_vol3/R04.pdf.

[30] Benoit 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 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. Ed. by Fatma Özcan, Georgia Koutrika, and Sam Madden. ACM, 2016, pp. 215–226. DOI: 10.1145/2882903.2903741. URL: https://doi.org/10.1145/2882903.2903741.

[31] Nilesh N. Dalvi and Dan Suciu. "Efficient query evaluation on probabilistic databases." In: *VLDB J.* 16.4 (2007), pp. 523–544. DOI: 10.1007/s00778-006-0004-3. URL: https://doi.org/10.1007/s00778-006-0004-3.

[32] Adnan Darwiche and Pierre Marquis. "A Knowledge Compilation Map." In: *J. Artif. Intell. Res.* 17 (2002), pp. 229–264. DOI: 10.1613/jair.989. URL: https://doi.org/10.1613/jair.989.

[33] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. "AI Meets AI: Leveraging Query Executions to Improve Index Recommendations." In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. ACM, 2019, pp. 1241–1258. DOI: 10.1145/

3299869.3324957. URL: https://doi.org/10.1145/3299869.3324957.

[34] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. "ALEX: An Updatable Adaptive Learned Index." In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. Ed. by David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo. ACM, 2020, pp. 969–984. DOI: 10.1145/3318464.3389711. URL: https://doi.org/10.1145/3318464.3389711.

[35] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. "Tsunami: A Learned Multi-dimensional Index for Correlated Data and Skewed Workloads." In: *Proc. VLDB Endow.* 14.2 (2020), pp. 74–86. DOI: 10.14778/3425879.3425880. URL: http://www.vldb.org/pvldb/vol14/p74-ding.pdf.

[36] Xin Luna Dong and Theodoros Rekatsinas. "Data Integration and Machine Learning: A Natural Synergy." In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*. Ed. by Ankur Teredesai, Vipin Kumar, Ying Li, Rómer Rosales, Evimaria Terzi, and George Karypis. ACM, 2019, pp. 3193–3194. DOI: 10.1145/3292500.3332296. URL: https://doi.org/10.1145/3292500.3332296.

[37] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. "Tuning database configuration parameters with ituned." In: *VLDB* 2.1 (2009), pp. 1246–1257.

[38] Gabriel Campero Durand, Rufat Piriyev, Marcus Pinnecke, David Broneske, Balasubramanian Gurumurthy, and Gunter Saake. "Automated Vertical Partitioning with Deep Reinforcement Learning." In: *New Trends in Databases and Information Systems, ADBIS 2019 Short Papers, Workshops BBIGAP, QAUCA, SemBDM, SIMPDA, M2P, MADEISD, and Doctoral Consortium, Bled, Slovenia, September 8-11, 2019, Proceedings*. Ed. by Tatjana Welzer, Johann Eder, Vili Podgorelec, Robert Wrembel, Mirjana Ivanovic, Johann Gamper, Mikolaj Morzy, Theodoros Tzouramanis, Jérôme Darmont, and Aida Kamisalic Latific. Vol. 1064. Communications in Computer and Information Science. Springer, 2019, pp. 126–134. DOI: 10.1007/978-3-030-30278-8_16. URL: https://doi.org/10.1007/978-3-030-30278-8%5C%5C_16.

[39] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek R. Narasayya, and Surajit Chaudhuri. "Selectivity Estimation for Range Predicates using Lightweight Models." In:

Proc. VLDB Endow. 12.9 (2019), pp. 1044–1057. DOI: 10.14778/3329772.3329780. URL: http://www.vldb.org/pvldb/vol12/p1044-dutt.pdf.

[40] George Eadon, Eugene Inseok Chong, Shrikanth Shankar, Ananth Raghavan, Jagannathan Srinivasan, and Souripriya Das. "Supporting table partitioning by reference in oracle." In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*. Ed. by Jason Tsong-Li Wang. ACM, 2008, pp. 1111–1122. DOI: 10.1145/1376616.1376727. URL: https://doi.org/10.1145/1376616.1376727.

[41] Philipp Eichmann, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. "IDEBench: A Benchmark for Interactive Data Exploration." In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. Ed. by David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo. ACM, 2020, pp. 1555–1569. DOI: 10.1145/3318464.3380574. URL: https://doi.org/10.1145/3318464.3380574.

[42] Ju Fan, Tongyu Liu, Guoliang Li, Junyou Chen, Yuwei Shen, and Xiaoyong Du. "Relational Data Synthesis using Generative Adversarial Networks: A Design Space Exploration." In: *Proc. VLDB Endow.* 13.11 (2020), pp. 1962–1975. URL: http://www.vldb.org/pvldb/vol13/p1962-fan.pdf.

[43] Su Feng, Aaron Huber, Boris Glavic, and Oliver Kennedy. "Uncertainty Annotated Databases - A Lightweight Approach for Approximating Certain Answers." In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. Ed. by Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska. ACM, 2019, pp. 1313–1330. DOI: 10.1145/3299869.3319887. URL: https://doi.org/10.1145/3299869.3319887.

[44] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. "Jockey: guaranteed job latency in data parallel clusters." In: *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*. Ed. by Pascal Felber, Frank Bellosa, and Herbert Bos. ACM, 2012, pp. 99–112. DOI: 10.1145/2168836.2168847. URL: https://doi.org/10.1145/2168836.2168847.

[45] Ilir Fetai, Damian Murezzan, and Heiko Schuldt. "Workload-driven adaptive data partitioning and distribution - The Cumulus approach." In: *2015 IEEE International Conference on Big*

Data (IEEE BigData 2015), Santa Clara, CA, USA, October 29 - November 1, 2015. IEEE Computer Society, 2015, pp. 1688–1697. DOI: 10.1109/BigData.2015.7363940. URL: https://doi.org/10.1109/BigData.2015.7363940.

[46] *Flights Dataset*. https://www.kaggle.com/usdot/flight-delays. Accessed: 2019-06-30.

[47] Florian Funke, Alfons Kemper, and Thomas Neumann. "Benchmarking Hybrid OLTP and OLAP Database Systems." In: *Datenbanksysteme für Business, Technologie und Web (BTW), 14. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 2.-4.3.2011 in Kaiserslautern, Germany*. Ed. by Theo Härder, Wolfgang Lehner, Bernhard Mitschang, Harald Schöning, and Holger Schwarz. Vol. P-180. LNI. GI, 2011, pp. 390–409. URL: https://dl.gi.de/20.500.12116/19591.

[48] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. "FITing-Tree: A Data-aware Index Structure." In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. Ed. by Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska. ACM, 2019, pp. 1189–1206. DOI: 10.1145/3299869.3319860. URL: https://doi.org/10.1145/3299869.3319860.

[49] Archana Ganapathi, Harumi A. Kuno, Umeshwar Dayal, Janet L. Wiener, Armando Fox, Michael I. Jordan, and David A. Patterson. "Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning." In: *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*. Ed. by Yannis E. Ioannidis, Dik Lun Lee, and Raymond T. Ng. IEEE Computer Society, 2009, pp. 592–603. DOI: 10.1109/ICDE.2009.130. URL: https://doi.org/10.1109/ICDE.2009.130.

[50] Robert Gens and Pedro M. Domingos. "Learning the Structure of Sum-Product Networks." In: *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*. Vol. 28. JMLR Workshop and Conference Proceedings. JMLR.org, 2013, pp. 873–880. URL: http://proceedings.mlr.press/v28/gens13.html.

[51] Mathieu Germain, Karol Gregor, Iain Murray, and Hugo Larochelle. "MADE: Masked Autoencoder for Distribution Estimation." In: *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*. Ed. by Francis R. Bach and David M. Blei. Vol. 37. JMLR Workshop and Conference Proceedings. JMLR.org, 2015, pp. 881–889. URL: http://proceedings.mlr.press/v37/germain15.html.

[52] Lise Getoor and Lilyana Mihalkova. "Learning statistical models from relational data." In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*. Ed. by Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis. ACM, 2011, pp. 1195–1198. DOI: 10.1145/1989323.1989451. URL: https://doi.org/10.1145/1989323.1989451.

[53] Lise Getoor, Benjamin Taskar, and Daphne Koller. "Selectivity Estimation using Probabilistic Models." In: *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*. Ed. by Sharad Mehrotra and Timos K. Sellis. ACM, 2001, pp. 461–472. DOI: 10.1145/375663.375727. URL: https://doi.org/10.1145/375663.375727.

[54] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. "Neural Message Passing for Quantum Chemistry." In: *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, 2017, pp. 1263–1272. URL: http://proceedings.mlr.press/v70/gilmer17a.html.

[55] Jonathan Goldstein and Per-Åke Larson. "Optimizing Queries Using Materialized Views: A practical, scalable solution." In: *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*. Ed. by Sharad Mehrotra and Timos K. Sellis. ACM, 2001, pp. 331–342. DOI: 10.1145/375663.375706. URL: https://doi.org/10.1145/375663.375706.

[56] Behzad Golshan, Alon Y. Halevy, George A. Mihaila, and Wang-Chiew Tan. "Data Integration: After the Teenage Years." In: *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*. Ed. by Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts. ACM, 2017, pp. 101–106. DOI: 10.1145/3034786.3056124. URL: https://doi.org/10.1145/3034786.3056124.

[57] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron C. Courville, and Yoshua Bengio. "Generative Adversarial Nets." In: *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*. Ed. by Zoubin Ghahramani, Max Welling, Corinna Cortes, Neil D. Lawrence, and Kilian Q. Weinberger. 2014, pp. 2672–2680. URL: https://proceedings.

neurips.cc/paper/2014/hash/5ca3e9b122f61f8f06494c97b1afccf3-Abstract.html.

[58] Eric Gribkoff and Dan Suciu. "SlimShot: In-Database Probabilistic Inference for Knowledge Bases." In: *Proc. VLDB Endow.* 9.7 (2016), pp. 552–563. DOI: 10.14778/2904483.2904487. URL: http://www.vldb.org/pvldb/vol9/p552-gribkoff.pdf.

[59] Yue Han, Guoliang Li, Haitao Yuan, and Ji Sun. "An Autonomous Materialized View Management System with Deep Reinforcement Learning." In: *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 2021, pp. 2159–2164. DOI: 10.1109/ICDE51399.2021.00217. URL: https://doi.org/10.1109/ICDE51399.2021.00217.

[60] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. "Deep Learning Models for Selectivity Estimation of Multi-Attribute Queries." In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. Ed. by David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo. ACM, 2020, pp. 1035–1050. DOI: 10.1145/3318464.3389741. URL: https://doi.org/10.1145/3318464.3389741.

[61] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. "Multi-Attribute Selectivity Estimation Using Deep Learning." In: *CoRR* abs/1903.09999 (2019). arXiv: 1903.09999. URL: http://arxiv.org/abs/1903.09999.

[62] Benjamin Hilprecht and Carsten Binnig. "One Model to Rule them All: Towards Zero-Shot Learning for Databases." In: *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org, 2022. URL: https://www.cidrdb.org/cidr2022/papers/p16-hilprecht.pdf.

[63] Benjamin Hilprecht and Carsten Binnig. "ReStore - Neural Data Completion for Relational Databases." In: *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. Ed. by Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava. ACM, 2021, pp. 710–722. DOI: 10.1145/3448016.3457264. URL: https://doi.org/10.1145/3448016.3457264.

[64] Benjamin Hilprecht and Carsten Binnig. *ReStore - Neural Data Completion for Relational Databases*. https://github.com/DataManagementLab/restore. 2021.

[65] Benjamin Hilprecht and Carsten Binnig. *Zero-Shot Cost Estimation Models*. https://github.com/DataManagementLab/zero-shot-cost-estimation. 2022.

[66] Benjamin Hilprecht and Carsten Binnig. "Zero-Shot Cost Models for Out-of-the-box Learned Cost Prediction." In: *Proc. VLDB Endow.* 15.11 (2022), pp. 2361–2374. DOI: 10.14778/3551793.3551799. URL: http://www.vldb.org/pvldb/vol15/p2483-hilprecht.pdf.

[67] 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: *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020. URL: http://cidrdb.org/cidr2020/papers/p34-hilprecht-cidr20.pdf.

[68] 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?* https://github.com/DataManagementLab/cidr-cost-model. 2020.

[69] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. "Learning a Partitioning Advisor for Cloud Databases." In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. Ed. by David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo. ACM, 2020, pp. 143–157. DOI: 10.1145/3318464.3389704. URL: https://doi.org/10.1145/3318464.3389704.

[70] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. "Learning a Partitioning Advisor for Cloud Databases." In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. Ed. by David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo. ACM, 2020, pp. 143–157. DOI: 10.1145/3318464.3389704. URL: https://doi.org/10.1145/3318464.3389704.

[71] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. *DeepDB: Learn from Data, not from Queries!* https://github.com/DataManagementLab/deepdb-public. 2020.

[72] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. "DeepDB: Learn from Data, not from Queries!" In: *Proc. VLDB Endow.* 13.7 (2020), pp. 992–1005. DOI: 10.14778/3384345.3384349. URL: http://www.vldb.org/pvldb/vol13/p992-hilprecht.pdf.

[73] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, and Zichen Zhu. "Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn." In: *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019. URL: http://cidrdb.org/cidr2019/papers/p143-idreos-cidr19.pdf.

[74] Stratos Idreos, Kostas Zoumpatianos, Subarna Chatterjee, Wilson Qin, Abdul Wasay, Brian Hentschel, Mike S. Kester, Niv Dayan, Demi Guo, Minseo Kang, and Yiyou Sun. "Learning Data Structure Alchemy." In: *IEEE Data Eng. Bull.* 42.2 (2019), pp. 47–58. URL: http://sites.computer.org/debull/A19june/p47.pdf.

[75] Abhay Kumar Jha and Dan Suciu. "Probabilistic Databases with MarkoViews." In: *Proc. VLDB Endow.* 5.11 (2012), pp. 1160–1171. DOI: 10.14778/2350229.2350236. URL: http://vldb.org/pvldb/vol5/p1160%5C%5C_abhayjha%5C%5C_vldb2012.pdf.

[76] Johan Kok Zhi Kang, Gaurav, Sien Yi Tan, Feng Cheng, Shixuan Sun, and Bingsheng He. "Efficient Deep Learning Pipelines for Accurate Cost Estimations Over Large Scale Query Workload." In: *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. Ed. by Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava. ACM, 2021, pp. 1014–1022. DOI: 10.1145/3448016.3457546. URL: https://doi.org/10.1145/3448016.3457546.

[77] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. "LightGBM: A Highly Efficient Gradient Boosting Decision Tree." In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. Ed. by Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett. 2017, pp. 3146–3154. URL: https://proceedings.neurips.cc/paper/2017/hash/6449f44a102fde848669bdd9eb6b76fa-Abstract.html.

[78] Elias B. Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. "Learning Combinatorial Optimization Algorithms over Graphs." In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems*

2017, December 4-9, 2017, Long Beach, CA, USA. Ed. by Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett. 2017, pp. 6348–6358. URL: https://proceedings.neurips.cc/paper/2017/hash/d9896106ca98d3d05b8cbdf4fd8b13a1-Abstract.html.

[79] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. "Learned Cardinalities: Estimating Correlated Joins with Deep Learning." In: *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019. URL: http://cidrdb.org/cidr2019/papers/p101-kipf-cidr19.pdf.

[80] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. "SOSD: A Benchmark for Learned Indexes." In: *CoRR* abs/1911.13014 (2019). arXiv: 1911.13014. URL: http://arxiv.org/abs/1911.13014.

[81] 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: *Proc. VLDB Endow.* 13.11 (2020), pp. 2382–2395. URL: http://www.vldb.org/pvldb/vol13/p2382-kossmann.pdf.

[82] Jan Kossmann, Alexander Kastius, and Rainer Schlosser. "SWIRL: Selection of Workload-aware Indexes using Reinforcement Learning." In: *Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022, Edinburgh, UK, March 29 - April 1, 2022*. Ed. by Julia Stoyanovich, Jens Teubner, Paolo Guagliardo, Milos Nikolic, Andreas Pieris, Jan Mühlig, Fatma Özcan, Sebastian Schelter, H. V. Jagadish, and Meihui Zhang. OpenProceedings.org, 2022, 2:155–2:168. DOI: 10.48786/edbt.2022.06. URL: https://doi.org/10.48786/edbt.2022.06.

[83] 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: *CIDR*. www.cidrdb.org, 2019. URL: http://cidrdb.org/cidr2019/papers/p117-kraska-cidr19.pdf.

[84] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. "The Case for Learned Index Structures." In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. Ed. by Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein. ACM, 2018, pp. 489–504. DOI: 10.1145/3183713.3196909. URL: https://doi.org/10.1145/3183713.3196909.

[85] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. "Learning to Optimize Join Queries With Deep Reinforcement Learning." In: *CoRR* abs/1808.03196 (2018). arXiv: 1808.03196. URL: http://arxiv.org/abs/1808.03196.

[86] Moritz Kulessa, Benjamin Hilprecht, Alejandro Molina, Kristian Kersting, and Carsten Binnig. "Towards Model-based Approximate Query Processing." In: *AIDB@VLDB 2019, 1st International Workshop on Applied AI for Database Systems and Applications, Held with VLDB 2019*. Ed. by Berthold Reinwald and Bingsheng He. 2019.

[87] M. Seetha Lakshmi and Shaoyu Zhou. "Selectivity Estimation in Extensible Databases - A Neural Network Approach." In: *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*. Ed. by Ashish Gupta, Oded Shmueli, and Jennifer Widom. Morgan Kaufmann, 1998, pp. 623–627. URL: http://www.vldb.org/conf/1998/p623.pdf.

[88] Christoph H. Lampert, Hannes Nickisch, and Stefan Harmeling. "Learning to detect unseen object classes by between-class attribute transfer." In: *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009), 20-25 June 2009, Miami, Florida, USA*. IEEE Computer Society, 2009, pp. 951–958. DOI: 10.1109/CVPR.2009.5206594. URL: https://doi.org/10.1109/CVPR.2009.5206594.

[89] Hai Lan, Zhifeng Bao, and Yuwei Peng. "An Index Advisor Using Deep Reinforcement Learning." In: *CIKM '20: The 29th ACM International Conference on Information and Knowledge Management, Virtual Event, Ireland, October 19-23, 2020*. ACM, 2020, pp. 2105–2108. DOI: 10.1145/3340531.3412106. URL: https://doi.org/10.1145/3340531.3412106.

[90] Willis Lang, Rimma V. Nehme, Eric Robinson, and Jeffrey F. Naughton. "Partial results in database systems." In: *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*. Ed. by Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu. ACM, 2014, pp. 1275–1286. DOI: 10.1145/2588555.2612176. URL: https://doi.org/10.1145/2588555.2612176.

[91] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. "How Good Are Query Optimizers, Really?" In: *Proc. VLDB Endow.* 9.3 (2015), pp. 204–215. DOI: 10.14778/2850583.2850594. URL: http://www.vldb.org/pvldb/vol9/p204-leis.pdf.

[92] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. "Cardinality Estimation Done Right: Index-Based Join Sampling." In: *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org, 2017. URL: http://cidrdb.org/cidr2017/papers/p9-leis-cidr17.pdf.

[93] 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 J.* 27.5 (2018), pp. 643–668. DOI: 10.1007/s00778-017-0480-7. URL: https://doi.org/10.1007/s00778-017-0480-7.

[94] Alon Y. Levy. "Obtaining Complete Answers from Incomplete Databases." In: *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*. Ed. by T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda. Morgan Kaufmann, 1996, pp. 402–412. URL: http://www.vldb.org/conf/1996/P402.PDF.

[95] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. "Wander Join: Online Aggregation via Random Walks." In: *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. 2016, pp. 615–629. DOI: 10.1145/2882903.2915235. URL: https://doi.org/10.1145/2882903.2915235.

[96] Jiexing Li, Arnd Christian König, Vivek R. Narasayya, and Surajit Chaudhuri. "Robust Estimation of Resource Consumption for SQL Queries using Statistical Techniques." In: *Proc. VLDB Endow.* 5.11 (2012), pp. 1555–1566. DOI: 10.14778/2350229.2350269. URL: http://vldb.org/pvldb/vol5/p1555%5C_jiexingli%5C_vldb2012.pdf.

[97] Teng Li, Zhiyuan Xu, Jian Tang, and Yanzhi Wang. "Model-free Control for Distributed Stream Data Processing using Deep Reinforcement Learning." In: *Proc. VLDB Endow.* 11.6 (2018), pp. 705–718. DOI: 10.14778/3184470.3184474. URL: http://www.vldb.org/pvldb/vol11/p705-li.pdf.

[98] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. "Differentiable programming for image processing and deep learning in halide." In: *ACM Trans. Graph.* 37.4 (2018), 139:1–139:13. DOI: 10.1145/3197517.3201383. URL: https://doi.org/10.1145/3197517.3201383.

[99] Xi Liang, Aaron J. Elmore, and Sanjay Krishnan. "Opportunistic View Materialization with Deep Reinforcement Learning." In: *CoRR* abs/1903.01363 (2019). arXiv: 1903.01363. URL: http://arxiv.org/abs/1903.01363.

[100] Henry Liu, Mingbin Xu, Ziting Yu, Vincent Corvinelli, and Calisto Zuzarte. "Cardinality estimation using neural networks." In: *Proceedings of 25th Annual International Conference on Computer Science and Software Engineering, CASCON 2015, Markham, Ontario, Canada, 2-4 November, 2015*. Ed. by Jordan Gould, Marin Litoiu, and Hanan Lutfiyya. IBM, 2015, pp. 53–59. URL: http://dl.acm.org/citation.cfm?id=2886453.

[101] David López-Paz, Philipp Hennig, and Bernhard Schölkopf. "The Randomized Dependence Coefficient." In: *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*. Ed. by Christopher J. C. Burges, Léon Bottou, Zoubin Ghahramani, and Kilian Q. Weinberger. 2013, pp. 1–9. URL: https://proceedings.neurips.cc/paper/2013/hash/aab3238922bcc25a6f606eb525ffdc56-Abstract.html.

[102] Yao Lu, Srikanth Kandula, Arnd Christian König, and Surajit Chaudhuri. "Pre-training Summarization Models of Structured Datasets for Cardinality Estimation." In: *Proc. VLDB Endow.* 15.3 (2021), pp. 414–426. DOI: 10.14778/3494124.3494127. URL: http://www.vldb.org/pvldb/vol15/p414-lu.pdf.

[103] Yi Lu, Anil Shanbhag, Alekh Jindal, and Samuel Madden. "AdaptDB: Adaptive Partitioning for Distributed Joins." In: *Proc. VLDB Endow.* 10.5 (2017), pp. 589–600. DOI: 10.14778/3055540.3055551. URL: http://www.vldb.org/pvldb/vol10/p589-lu.pdf.

[104] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. "Query-based Workload Forecasting for Self-Driving Database Management Systems." In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. Ed. by Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein. ACM, 2018, pp. 631–645. DOI: 10.1145/3183713.3196908. URL: https://doi.org/10.1145/3183713.3196908.

[105] Qingzhi Ma and Peter Triantafillou. "DBEst: Revisiting Approximate Query Processing Engines with Machine Learning Models." In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. Ed. by Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska. ACM, 2019, pp. 1553–1570. DOI: 10.1145/3299869.3324958. URL: https://doi.org/10.1145/3299869.3324958.

[106] Tanu Malik, Randal C. Burns, and Nitesh V. Chawla. "A Black-Box Approach to Query Cardinality Estimation." In: *Third Biennial Conference on Innovative Data Systems Research, CIDR 2007, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*. www.cidrdb.org, 2007, pp. 56–67. URL: http://cidrdb.org/cidr2007/papers/cidr07p06.pdf.

[107] 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, Beijing, China, August 19-23, 2019*. Ed. by Jianping Wu and Wendy Hall. ACM, 2019, pp. 270–288. DOI: 10.1145/3341302.3342080. URL: https://doi.org/10.1145/3341302.3342080.

[108] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. "Bao: Learning to Steer Query Optimizers." In: *CoRR* abs/2004.03814 (2020). arXiv: 2004.03814. URL: https://arxiv.org/abs/2004.03814.

[109] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. "Bao: Making Learned Query Optimization Practical." In: *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. Ed. by Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava. ACM, 2021, pp. 1275–1288. DOI: 10.1145/3448016.3452838. URL: https://doi.org/10.1145/3448016.3452838.

[110] Ryan Marcus and Olga Papaemmanouil. "Deep Reinforcement Learning for Join Order Enumeration." In: *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2018, Houston, TX, USA, June 10, 2018*. Ed. by Rajesh Bordawekar and Oded Shmueli. ACM, 2018, 3:1–3:4. DOI: 10.1145/3211954.3211957. URL: https://doi.org/10.1145/3211954.3211957.

[111] Ryan Marcus, Olga Papaemmanouil, Sofiya Semenova, and Solomon Garber. "NashDB: An End-to-End Economic Method for Elastic Database Fragmentation, Replication, and Provisioning." In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. Ed. by Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein. ACM, 2018, pp. 1253–1267. DOI: 10.1145/3183713.3196935. URL: https://doi.org/10.1145/3183713.3196935.

[112] Ryan C. Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. "Neo: A Learned Query Optimizer." In: *Proc.*

VLDB Endow. 12.11 (2019), pp. 1705–1718. DOI: 10.14778/3342263.3342644. URL: http://www.vldb.org/pvldb/vol12/p1705-marcus.pdf.

[113] Ryan C. Marcus and Olga Papaemmanouil. "Plan-Structured Deep Neural Network Models for Query Performance Prediction." In: *Proc. VLDB Endow.* 12.11 (2019), pp. 1733–1746. DOI: 10.14778/3342263.3342646. URL: http://www.vldb.org/pvldb/vol12/p1733-marcus.pdf.

[114] Chris Mayfield, Jennifer Neville, and Sunil Prabhakar. "ERACER: a database approach for statistical inference and data cleaning." In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*. Ed. by Ahmed K. Elmagarmid and Divyakant Agrawal. ACM, 2010, pp. 75–86. DOI: 10.1145/1807167.1807178. URL: https://doi.org/10.1145/1807167.1807178.

[115] 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: *Nat.* 518.7540 (2015), pp. 529–533. DOI: 10.1038/nature14236. URL: https://doi.org/10.1038/nature14236.

[116] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. "Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors." In: *Proc. VLDB Endow.* 2.1 (2009), pp. 982–993. DOI: 10.14778/1687627.1687738. URL: http://www.vldb.org/pvldb/vol2/vldb09-657.pdf.

[117] Alejandro Molina, Sriraam Natarajan, and Kristian Kersting. "Poisson Sum-Product Networks: A Deep Architecture for Tractable Multivariate Poisson Distributions." In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*. Ed. by Satinder Singh and Shaul Markovitch. AAAI Press, 2017, pp. 2357–2363. URL: http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14530.

[118] Alejandro Molina, Antonio Vergari, Nicola Di Mauro, Sriraam Natarajan, Floriana Esposito, and Kristian Kersting. "Mixed Sum-Product Networks: A Deep Architecture for Hybrid Domains." In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*. Ed. by Sheila A. McIl-

raith and Kilian Q. Weinberger. AAAI Press, 2018, pp. 3828–3835. URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16865.

[119] Alejandro Molina, Antonio Vergari, Karl Stelzner, Robert Peharz, Pranav Subramani, Nicola Di Mauro, Pascal Poupart, and Kristian Kersting. "SPFlow: An Easy and Extensible Library for Deep Probabilistic Learning using Sum-Product Networks." In: *CoRR* abs/1901.03704 (2019). arXiv: 1901.03704. URL: http://arxiv.org/abs/1901.03704.

[120] Jan Motl and Oliver Schulte. "The CTU Prague Relational Learning Repository." In: *CoRR* abs/1511.03086 (2015). arXiv: 1511.03086. URL: http://arxiv.org/abs/1511.03086.

[121] Amihai Motro. "Integrity = Validity + Completeness." In: *ACM Trans. Database Syst.* 14.4 (1989), pp. 480–502. DOI: 10.1145/76902.76904. URL: https://doi.org/10.1145/76902.76904.

[122] Kunal Mukerjee, Tomas Talius, Ajay Kalhan, Nigel Ellis, and Conor Cunningham. "SQL Azure as a Self-Managing Database Service: Lessons Learned and Challenges Ahead." In: *IEEE Data Eng. Bull.* 34.4 (2011), pp. 61–70. URL: http://sites.computer.org/debull/A11dec/azure2.pdf.

[123] Raghunath Othayoth Nambiar and Meikel Poess. "The Making of TPC-DS." In: *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*. Ed. by Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim. ACM, 2006, pp. 1049–1058. URL: http://dl.acm.org/citation.cfm?id=1164217.

[124] Dushyanth Narayanan, Eno Thereska, and Anastassia Ailamaki. "Continuous resource monitoring for self-predicting DBMS." In: *13th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2005), 27-29 September 2005, Atlanta, GA, USA*. IEEE Computer Society, 2005, pp. 239–248. DOI: 10.1109/MASCOTS.2005.21. URL: https://doi.org/10.1109/MASCOTS.2005.21.

[125] Charlie Nash and Conor Durkan. "Autoregressive Energy Machines." In: *CoRR* abs/1904.05626 (2019). arXiv: 1904.05626. URL: http://arxiv.org/abs/1904.05626.

[126] Parimarjan Negi, Ryan C. Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. "Flow-Loss: Learning Cardinality Estimates That Matter." In: *Proc. VLDB Endow.* 14.11 (2021), pp. 2019–2032. DOI: 10.14778/3476249.3476259. URL: http://www.vldb.org/pvldb/vol14/p2019-negi.pdf.

[127] Rimma V. Nehme and Nicolas Bruno. "Automated partitioning design in parallel database systems." In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*. Ed. by Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis. ACM, 2011, pp. 1137–1148. DOI: 10.1145/1989323.1989444. URL: https://doi.org/10.1145/1989323.1989444.

[128] Patrick E. O'Neil, Elizabeth J. O'Neil, Xuedong Chen, and Stephen Revilak. "The Star Schema Benchmark and Augmented Fact Table Indexing." In: *Performance Evaluation and Benchmarking, First TPC Technology Conference, TPCTC 2009, Lyon, France, August 24-28, 2009, Revised Selected Papers*. Ed. by Raghunath Othayoth Nambiar and Meikel Poess. Vol. 5895. Lecture Notes in Computer Science. Springer, 2009, pp. 237–252. DOI: 10.1007/978-3-642-10424-4_17. URL: https://doi.org/10.1007/978-3-642-10424-4%5C%5C_17.

[129] Patrick E O'Neil, Elizabeth J O'Neil, and Xuedong Chen. "The star schema benchmark (SSB)." In: *Pat* 200.0 (2007), p. 50.

[130] Dan Olteanu, Jiewen Huang, and Christoph Koch. "SPROUT: Lazy vs. Eager Query Plans for Tuple-Independent Probabilistic Databases." In: *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*. Ed. by Yannis E. Ioannidis, Dik Lun Lee, and Raymond T. Ng. IEEE Computer Society, 2009, pp. 640–651. DOI: 10.1109/ICDE.2009.123. URL: https://doi.org/10.1109/ICDE.2009.123.

[131] Laurel J. Orr, Magdalena Balazinska, and Dan Suciu. "Sample Debiasing in the Themis Open World Database System." In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. Ed. by David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo. ACM, 2020, pp. 257–268. DOI: 10.1145/3318464.3380606. URL: https://doi.org/10.1145/3318464.3380606.

[132] Aditya Paliwal, Felix Gimeno, Vinod Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, and Oriol Vinyals. "Reinforced Genetic Algorithm Learning for Optimizing Computation Graphs." In: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL: https://openreview.net/forum?id=rkxDoJBYPB.

[133] George Papamakarios, Iain Murray, and Theo Pavlakou. "Masked Autoregressive Flow for Density Estimation." In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. Ed. by Isabelle Guyon, Ulrike

von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett. 2017, pp. 2338–2347. URL: https://proceedings.neurips.cc/paper/2017/hash/6c1da886822c67822bcf3679d04369fa-Abstract.html.

[134] Panos Parchas, Yonatan Naamad, Peter Van Bouwel, Christos Faloutsos, and Michalis Petropoulos. "Fast and Effective Distribution-Key Recommendation for Amazon Redshift." In: *Proc. VLDB Endow.* 13.11 (2020), pp. 2411–2423. URL: http://www.vldb.org/pvldb/vol13/p2411-parchas.pdf.

[135] Yongjoo Park, Barzan Mozafari, Joseph Sorenson, and Junhao Wang. "VerdictDB: Universalizing Approximate Query Processing." In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. Ed. by Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein. ACM, 2018, pp. 1461–1476. DOI: 10.1145/3183713.3196905. URL: https://doi.org/10.1145/3183713.3196905.

[136] Yongjoo Park, Ahmad Shahab Tajik, Michael J. Cafarella, and Barzan Mozafari. "Database Learning: Toward a Database that Becomes Smarter Every Time." In: *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. Ed. by Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu. ACM, 2017, pp. 587–602. DOI: 10.1145/3035918.3064013. URL: https://doi.org/10.1145/3035918.3064013.

[137] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. Ed. by Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett. 2019, pp. 8024–8035. URL: https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

[138] 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: *8th Biennial Conference on Innovative Data Systems Research,*

CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings. www.cidrdb.org, 2017. URL: http://cidrdb.org/cidr2017/papers/p42-pavlo-cidr17.pdf.

[139] Andrew Pavlo, Carlo Curino, and Stanley B. Zdonik. "Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems." In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*. Ed. by K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman. ACM, 2012, pp. 61–72. DOI: 10.1145/2213836.2213844. URL: https://doi.org/10.1145/2213836.2213844.

[140] 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: *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 2021, pp. 600–611. DOI: 10.1109/ICDE51399.2021.00058. URL: https://doi.org/10.1109/ICDE51399.2021.00058.

[141] Hoifung Poon and Pedro M. Domingos. "Sum-product networks: A new deep architecture." In: *IEEE International Conference on Computer Vision Workshops, ICCV 2011 Workshops, Barcelona, Spain, November 6-13, 2011*. IEEE Computer Society, 2011, pp. 689–690. DOI: 10.1109/ICCVW.2011.6130310. URL: https://doi.org/10.1109/ICCVW.2011.6130310.

[142] *Postgres-XL database*. https://www.postgres-xl.org/.

[143] Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. "SWORD: scalable workload-aware data placement for transactional workloads." In: *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013*. Ed. by Giovanna Guerrini and Norman W. Paton. ACM, 2013, pp. 430–441. DOI: 10.1145/2452376.2452427. URL: https://doi.org/10.1145/2452376.2452427.

[144] Tilmann Rabl and Hans-Arno Jacobsen. "Query Centric Partitioning and Allocation for Partially Replicated Database Systems." In: *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. Ed. by Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu. ACM, 2017, pp. 315–330. DOI: 10.1145/3035918.3064052. URL: https://doi.org/10.1145/3035918.3064052.

[145] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy M. Lohman. "Automating physical database design in a parallel database." In: *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002*.

Ed. by Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki. ACM, 2002, pp. 558–569. DOI: 10.1145/564691.564757. URL: https://doi.org/10.1145/564691.564757.

[146] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. "HoloClean: Holistic Data Repairs with Probabilistic Inference." In: *Proc. VLDB Endow.* 10.11 (2017), pp. 1190–1201. DOI: 10.14778/3137628.3137631. URL: http://www.vldb.org/pvldb/vol10/p1190-rekatsinas.pdf.

[147] Theodoros Rekatsinas, Amol Deshpande, and Lise Getoor. "Local structure and determinism in probabilistic databases." In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*. Ed. by K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman. ACM, 2012, pp. 373–384. DOI: 10.1145/2213836.2213879. URL: https://doi.org/10.1145/2213836.2213879.

[148] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson, 2020. ISBN: 9780134610993. URL: http://aima.cs.berkeley.edu/.

[149] Tim Salimans, Ian J. Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. "Improved Techniques for Training GANs." In: *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*. Ed. by Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett. 2016, pp. 2226–2234. URL: https://proceedings.neurips.cc/paper/2016/hash/8a3363abe792db2d8761d6403605aeb7-Abstract.html.

[150] Thomas Schmied, Diego Didona, Andreas C. Döring, Thomas P. Parnell, and Nikolas Ioannou. "Towards a General Framework for ML-based Self-tuning Databases." In: *EuroMLSys@EuroSys 2021, Proceedings of the 1st Workshop on Machine Learning and Systemsg Virtual Event, Edinburgh, Scotland, UK, 26 April, 2021*. Ed. by Eiko Yoneki and Paul Patras. ACM, 2021, pp. 24–30. DOI: 10.1145/3437984.3458830. URL: https://doi.org/10.1145/3437984.3458830.

[151] Prithviraj Sen, Amol Deshpande, and Lise Getoor. "PrDB: managing and exploiting rich correlations in probabilistic databases." In: *VLDB J.* 18.5 (2009), pp. 1065–1090. DOI: 10.1007/s00778-009-0153-2. URL: https://doi.org/10.1007/s00778-009-0153-2.

[152] Jayavel Shanmugasundaram, Usama M. Fayyad, and Paul S. Bradley. "Compressed Data Cubes for OLAP Aggregate Query Approximation on Continuous Dimensions." In: *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge*

Discovery and Data Mining, San Diego, CA, USA, August 15-18, 1999. Ed. by Usama M. Fayyad, Surajit Chaudhuri, and David Madigan. ACM, 1999, pp. 223–232. DOI: 10.1145/312129.312231. URL: https://doi.org/10.1145/312129.312231.

[153] Yangjun Sheng, Anthony Tomasic, Tieying Zhang, and Andrew Pavlo. "Scheduling OLTP transactions via learned abort prediction." In: *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM at SIGMOD 2019, Amsterdam, The Netherlands, July 5, 2019*. Ed. by Rajesh Bordawekar and Oded Shmueli. ACM, 2019, 1:1–1:8. DOI: 10.1145/3329859.3329871. URL: https://doi.org/10.1145/3329859.3329871.

[154] Suraj Shetiya, Saravanan Thirumuruganathan, Nick Koudas, and Gautam Das. "Astrid: Accurate Selectivity Estimation for String Predicates using Deep Learning." In: *Proc. VLDB Endow.* 14.4 (2020), pp. 471–484. DOI: 10.14778/3436905.3436907. URL: http://www.vldb.org/pvldb/vol14/p471-shetiya.pdf.

[155] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. "Cost Models for Big Data Query Processing: Learning, Retrofitting, and Our Findings." In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. Ed. by David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo. ACM, 2020, pp. 99–113. DOI: 10.1145/3318464.3380584. URL: https://doi.org/10.1145/3318464.3380584.

[156] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. "Mastering the game of Go with deep neural networks and tree search." In: *Nat.* 529.7587 (2016), pp. 484–489. DOI: 10.1038/nature16961. URL: https://doi.org/10.1038/nature16961.

[157] *Snowflake Cloud Data Warehouse*. https://www.snowflake.com/. Accessed: 2020-09-12.

[158] Lukas Sommer, Julian Oppermann, Alejandro Molina, Carsten Binnig, Kristian Kersting, and Andreas Koch. "Automatic Mapping of the Sum-Product Network Inference Problem to FPGA-Based Accelerators." In: *36th IEEE International Conference on Computer Design, ICCD 2018, Orlando, FL, USA, October 7-10, 2018*. IEEE Computer Society, 2018, pp. 350–357. DOI: 10.1109/ICCD.2018.00060. URL: https://doi.org/10.1109/ICCD.2018.00060.

[159] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. "C-Store: A Column-oriented DBMS." In: *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*. Ed. by Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi. ACM, 2005, pp. 553–564. URL: http://www.vldb.org/archives/website/2005/program/paper/thu/p553-stonebraker.pdf.

[160] Dan Suciu and Christopher Re. *Efficient top-K query evaluation on probabilistic data*. US Patent 7,814,113. Oct. 2010.

[161] Ji Sun and Guoliang Li. "An End-to-End Learning-based Cost Estimator." In: *Proc. VLDB Endow.* 13.3 (2019), pp. 307–319. DOI: 10.14778/3368289.3368296. URL: http://www.vldb.org/pvldb/vol13/p307-sun.pdf.

[162] Lijun Sun and Alexander Erath. "A Bayesian network approach for population synthesis." In: *Transportation Research Part C: Emerging Technologies* 61 (2015), pp. 49–62.

[163] Bruhathi Sundarmurthy, Paraschos Koutris, Willis Lang, Jeffrey F. Naughton, and Val Tannen. "m-tables: Representing Missing Data." In: *20th International Conference on Database Theory, ICDT 2017, March 21-24, 2017, Venice, Italy*. Ed. by Michael Benedikt and Giorgio Orsi. Vol. 68. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 21:1–21:20. DOI: 10.4230/LIPIcs.ICDT.2017.21. URL: https://doi.org/10.4230/LIPIcs.ICDT.2017.21.

[164] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning - an introduction*. Adaptive computation and machine learning. MIT Press, 1998. ISBN: 978-0-262-19398-6. URL: https://www.worldcat.org/oclc/37293240.

[165] Aviv Tamar, Yi Wu, Garrett Thomas, Sergey Levine, and Pieter Abbeel. "Value Iteration Networks." In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*. Ed. by Carles Sierra. ijcai.org, 2017, pp. 4949–4953. DOI: 10.24963/ijcai.2017/700. URL: https://doi.org/10.24963/ijcai.2017/700.

[166] Arvind Thiagarajan and Samuel Madden. "Querying continuous functions in a database system." In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*. Ed. by Jason Tsong-Li Wang. ACM, 2008, pp. 791–804. DOI: 10.1145/1376616.1376696. URL: https://doi.org/10.1145/1376616.1376696.

[167] Saravanan Thirumuruganathan, Shohedul Hasan, Nick Koudas, and Gautam Das. "Approximate Query Processing using Deep Generative Models." In: *CoRR* abs/1903.10000 (2019). arXiv: 1903.10000. URL: http://arxiv.org/abs/1903.10000.

[168] *TPC-DS benchmark*. http://www.tpc.org/tpcds/.

[169] *TPC-H benchmark*. http://www.tpc.org/tpch/.

[170] Kostas Tzoumas, Amol Deshpande, and Christian S. Jensen. "Efficiently adapting graphical models for selectivity estimation." In: *VLDB J.* 22.1 (2013), pp. 3–27. DOI: 10.1007/s00778-012-0293-7. URL: https://doi.org/10.1007/s00778-012-0293-7.

[171] 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 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*. IEEE Computer Society, 2000, pp. 101–110. DOI: 10.1109/ICDE.2000.839397. URL: https://doi.org/10.1109/ICDE.2000.839397.

[172] Shivaram Venkataraman, Zongheng Yang, Michael J. Franklin, Benjamin Recht, and Ion Stoica. "Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics." In: *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*. Ed. by Katerina J. Argyraki and Rebecca Isaacs. USENIX Association, 2016, pp. 363–378. URL: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/venkataraman.

[173] Francesco Ventura, Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. "Expand your Training Limits! Generating Training Data for ML-based Data Management." In: *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. 2021, pp. 1865–1878. DOI: 10.1145/3448016.3457286. URL: https://doi.org/10.1145/3448016.3457286.

[174] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. "Building An Elastic Query Engine on Disaggregated Storage." In: *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*. Ed. by Ranjita Bhagwan and George Porter. USENIX Association, 2020, pp. 449–462. URL: https://www.usenix.org/conference/nsdi20/presentation/vuppalapati.

[175] Daisy Zhe Wang, Eirinaios Michelakis, Minos N. Garofalakis, and Joseph M. Hellerstein. "BayesStore: managing large, uncertain data repositories with probabilistic graphical models."

In: *Proc. VLDB Endow.* 1.1 (2008), pp. 340–351. DOI: 10.14778/1453856.1453896. URL: http://www.vldb.org/pvldb/vol1/1453896.pdf.

[176] Fei Wang, James M. Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. "Backpropagation with Callbacks: Foundations for Efficient and Expressive Differentiable Programming." In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. Ed. by Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett. 2018, pp. 10201–10212. URL: https://proceedings.neurips.cc/paper/2018/hash/34e157766f31db3d2099831d348a7933-Abstract.html.

[177] Jiannan Wang, Sanjay Krishnan, Michael J. Franklin, Ken Goldberg, Tim Kraska, and Tova Milo. "A sample-and-clean framework for fast and accurate query processing on dirty data." In: *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*. Ed. by Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu. ACM, 2014, pp. 469–480. DOI: 10.1145/2588555.2610505. URL: https://doi.org/10.1145/2588555.2610505.

[178] Thomas Wang, Simone Ferlin, and Marco Chiesa. "Predicting CPU usage for proactive autoscaling." In: *EuroMLSys@EuroSys 2021, Proceedings of the 1st Workshop on Machine Learning and Systemsg Virtual Event, Edinburgh, Scotland, UK, 26 April, 2021*. Ed. by Eiko Yoneki and Paul Patras. ACM, 2021, pp. 31–38. DOI: 10.1145/3437984.3458831. URL: https://doi.org/10.1145/3437984.3458831.

[179] Lucas Woltmann, Claudio Hartmann, Maik Thiele, Dirk Habich, and Wolfgang Lehner. "Cardinality estimation with local deep learning models." In: *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2019, Amsterdam, The Netherlands, July 5, 2019*. Ed. by Rajesh Bordawekar and Oded Shmueli. ACM, 2019, 5:1–5:8. DOI: 10.1145/3329859.3329875. URL: https://doi.org/10.1145/3329859.3329875.

[180] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. "Towards a Learning Optimizer for Shared Clouds." In: *Proc. VLDB Endow.* 12.3 (2018), pp. 210–222. DOI: 10.14778/3291264.3291267. URL: http://www.vldb.org/pvldb/vol12/p210-wu.pdf.

[181] Richard Wu, Aoqian Zhang, Ihab F. Ilyas, and Theodoros Rekatsinas. "Attention-based Learning for Missing Data Imputation in HoloClean." In: *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*.

Ed. by Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze. mlsys.org, 2020. URL: https://proceedings.mlsys.org/book/307.pdf.

[182] Yingjun Wu, Jia Yu, Yuanyuan Tian, Richard Sidle, and Ronald Barber. "Designing Succinct Secondary Indexing Mechanism by Exploiting Column Correlations." In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. Ed. by Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska. ACM, 2019, pp. 1223–1240. DOI: 10.1145/3299869.3319861. URL: https://doi.org/10.1145/3299869.3319861.

[183] Lei Xu and Kalyan Veeramachaneni. "Synthesizing Tabular Data using Generative Adversarial Networks." In: *CoRR* abs/1811.11264 (2018). arXiv: 1811.11264. URL: http://arxiv.org/abs/1811.11264.

[184] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. "Balsa: Learning a Query Optimizer Without Expert Demonstrations." In: *CoRR* abs/2201.01441 (2022). arXiv: 2201.01441. URL: https://arxiv.org/abs/2201.01441.

[185] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. "NeuroCard: One Cardinality Estimator for All Tables." In: *Proc. VLDB Endow.* 14.1 (2020), pp. 61–73. DOI: 10.14778/3421424.3421432. URL: http://www.vldb.org/pvldb/vol14/p61-yang.pdf.

[186] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. "Deep Unsupervised Cardinality Estimation." In: *Proc. VLDB Endow.* 13.3 (2019), pp. 279–292. DOI: 10.14778/3368289.3368294. URL: http://www.vldb.org/pvldb/vol13/p279-yang.pdf.

[187] Jinsung Yoon, James Jordon, and Mihaela van der Schaar. "GAIN: Missing Data Imputation using Generative Adversarial Nets." In: *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. Ed. by Jennifer G. Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, 2018, pp. 5675–5684. URL: http://proceedings.mlr.press/v80/yoon18a.html.

[188] Haitao Yuan, Guoliang Li, Ling Feng, Ji Sun, and Yue Han. "Automatic View Generation with Deep Learning and Reinforcement Learning." In: *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE,

2020, pp. 1501–1512. DOI: 10.1109/ICDE48307.2020.00133. URL: https://doi.org/10.1109/ICDE48307.2020.00133.

[189] Matei Zaharia, Ali Ghodsi, Reynold Xin, and Michael Armbrust. "Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics." In: *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org, 2021. URL: http://cidrdb.org/cidr2021/papers/cidr2021%5C_paper17.pdf.

[190] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabás Póczos, Ruslan Salakhutdinov, and Alexander J. Smola. "Deep Sets." In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. Ed. by Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett. 2017, pp. 3391–3401. URL: https://proceedings.neurips.cc/paper/2017/hash/f22e4747da1aa27e363d86d40ff442fe-Abstract.html.

[191] Erfan Zamanian, Carsten Binnig, and Abdallah Salama. "Locality-aware Partitioning in Parallel Database Systems." In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. Ed. by Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives. ACM, 2015, pp. 17–30. DOI: 10.1145/2723372.2723718. URL: https://doi.org/10.1145/2723372.2723718.

[192] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, et al. "An end-to-end automatic cloud database tuning system using deep reinforcement learning." In: *SIGMOD*. 2019, pp. 415–432.

[193] Xuanhe Zhou, Luyang Liu, Wenbo Li, and et al. "AutoIndex: An Incremental Index Management System for Dynamic Workloads." In: *ICDE*. 2022.

[194] Xuanhe Zhou, Ji Sun, Guoliang Li, and Jianhua Feng. "Query Performance Prediction for Concurrent Queries using Graph Embedding." In: *Proc. VLDB Endow.* 13.9 (2020), pp. 1416–1428. DOI: 10.14778/3397230.3397238. URL: http://www.vldb.org/pvldb/vol13/p1416-zhou.pdf.

[195] Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. "FLAT: Fast, Lightweight and Accurate Method for Cardinality Estimation." In: *Proc. VLDB Endow.* 14.9 (2021), pp. 1489–1502. DOI: 10.14778/3461535.3461539. URL: http://www.vldb.org/pvldb/vol14/p1489-zhu.pdf.

[196] 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: *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*. Ed. by Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer. Morgan Kaufmann, 2004, pp. 1087–1097. DOI: 10.1016/B978-012088469-8.50095-4. URL: http://www.vldb.org/conf/2004/IND4P1.PDF.