Department of Computer Science
Series of Publications A
Report A-2021-5

Performance Benchmarking and Query Optimization for Multi-Model Databases

Chao Zhang

Doctoral thesis, to be presented for public examination with the permission of the Faculty of Science of the University of Helsinki, in Auditorium B123, Exactum, Pietari Kalmin katu 5, on the 19th of April, 2021 at 13 o'clock.

University of Helsinki
Finland

Supervisor
Jiaheng Lu, University of Helsinki, Finland

Pre-examiners
Yongluan Zhou, University of Copenhagen, Denmark
Meike Klettke, University of Rostock, Germany

Opponent
Konstantinos Stefanidis, Tampere University, Finland

Custos
Jiaheng Lu, University of Helsinki, Finland

Contact information

Department of Computer Science
P.O. Box 68 (Pietari Kalmin katu 5)
FI-00014 University of Helsinki
Finland

Email address: info@cs.helsinki.fi
URL: http://cs.helsinki.fi/
Telephone: +358 2941 911

ISSN 1238-8645
ISBN 978-951-51-7197-9 (paperback)
ISBN 978-951-51-7198-6 (PDF)
Helsinki 2021
Unigrafia

Performance Benchmarking and Query Optimization for Multi-Model Databases

Chao Zhang

Department of Computer Science
P.O. Box 68, FI-00014 University of Helsinki, Finland
chao.z.zhang@helsinki.fi

PhD Thesis, Series of Publications A, Report A-2021-5
Helsinki, April 2021, 68+90 pages
ISSN 1238-8645
ISBN 978-951-51-7197-9 (paperback)
ISBN 978-951-51-7198-6 (PDF)

Abstract

Multi-Model DataBases (MMDBs) are database management systems that utilize a single platform to store, manipulate, and query data in various data models, e.g., relational, tree, and graph models. Numerous MMDBs have been developed to facilitate multi-model data management, but they differ fundamentally in data storage, query language, and query processing. Existing tools are not suitable for benchmarking MMDBs because they do not consider the multi-model data and workloads. Therefore, it is of paramount importance to provide a new benchmark to evaluate the performance of MMDBs. Furthermore, MMDBs bring new issues for query optimization as traditional techniques cannot properly optimize the queries due to the lack of consideration of multi-model operators and storage. Thus, it calls for new approaches to optimize multi-model queries.

This thesis is divided into two parts to accomplish the above two goals, respectively. In the first part, we developed a new benchmark system for MMDBs in a scenario of social commerce with five data models, i.e., relational, JSON, XML, graph, and key-value. This benchmark system is an end-to-end tool that consists of various components, including synthetic data generation, workload specification, parameter curation, and execution client. Moreover, we leveraged the developed system to conduct a holistic experimental evaluation of the state-of-the-art MMDBs to compare their performance and identify their performance bottlenecks in handling multi-model workloads. In the second part, we proposed two query optimization

techniques for MMDBs. Firstly, we proposed a kernel density estimation (KDE)-based model to estimate the selectivity of multi-model joins that involve predicates for relational and tree data models. The estimation method can serve as a building block for selecting an optimal joining order in a cross-model query execution plan. We also proposed two approaches, the max-min approximation (MMA) and grid-based approximation (GBA) models, to approximate the KDE contribution while improving the estimation efficiency for large data samples. Secondly, we studied the problem of view selection in the relational-based graph databases to avoid the costly joins in the relational engine. Particularly, we proposed an end-to-end system that can automatically create, evaluate, and select views for accelerating the query processing. We proposed an extended graph view that can answer the subgraph and supergraph queries simultaneously. We devised a filtering-and-verification framework that enables the verification of the query containment by graph views. We formalized the view selection problem as a 0-1 Knapsack problem, then we developed a view selection algorithm, named graph gene algorithm (GGA), which explores the graph view transformations to reduce the view space and optimize the view benefit. Overall, the present thesis advances MMDBs in three aspects, i.e., performance benchmarking, join selectivity estimation, and automatic view selection.

Computing Reviews (2012) Categories and Subject Descriptors:

Information systems → Data management systems → Database administration → Database performance evaluation
Information systems → Data management systems → Database management system engines → Database query processing → Query optimization
Information systems → Data management systems → Database management system engines → Database query processing → Database views

General Terms:
Multi-Model Databases (MMDBs), Benchmarking, Query Optimization

Additional Key Words and Phrases:
Multi-Model Query, Data Generation, Parameter Curation, Relation-Tree Join, Join Selectivity, Kernel Density Estimation, Relational-Based Graph Databases, View Selection

Acknowledgements

There is no royal road to science, and only those who do not dread the fatiguing climb of its steep paths have a chance of gaining its luminous summits.
— *Karl Marx*

The finance support from China Scholarship Council (CSC) fellowship (201606360137), the Academy of Finland (310321) and CIMO fellowship have made it possible for me to complete this dissertation. Special thanks go to Kela, the Social Insurance Institution of Finland, which has supported my family in Finland with a number of benefits ensuring a high quality of life for us during my Ph.D. studies.

I would like to sincerely thank my dear supervisor, Jiaheng Lu. Your countless pieces of advice, inspiration, and support have been invaluable throughout my Ph.D. journey. Your spirit of science will surely be helping my academic career in the future. It is absolutely impossible for me to complete this dissertation without your supervision and help. No matter how busy and how tired you were, you can always spend time discussing with me for every detail of my idea and paper. I really appreciate that you always gave me timely feedback when I was not on the right track. I am thankful for your encouragement and inspiration when I got frustrated, particularly with paper rejections. After all, acceptances build CVs while rejections build men. It is my great honor to be your Ph.D. student.

I wish to extend my gratitude to the friends and the members of the UDBMS group in the department. Thank you for all the help in my Ph.D. studies and in daily life. It has been a joy to work and hang out with you during these years. Particularly, I want to thank Pengfei Xu, Chen He, Lumin Wei, Jun Chen, Jianjun Wang, Yuxing Chen, Gongsheng Yuan, Qingsong Guo, Dawei Wang, Lizhen Fu, and Valter Uotila.

I would like to express my appreciation to our fantastic colleagues at the University for your abundant help. Particularly, I want to thank Pirjo Moen, Ritva Karttunen, Päivi Lindeman, Minna Lauri, Roosa Sillanpää, Marina Kurten, and Ville Hautakangas.

Special thanks go to Prof. Meike Klettke and Prof. Yongluan Zhou for their precious time and rigorous pre-examination, which helped me improve the quality of this thesis.

Finally, and most importantly, I want to thank my family for the boundless love and support you gave to me throughout my Ph.D. journey. First of all, I would like to dedicate this thesis to my dear father, who lives in my mind forever. Furthermore, I want to express my deep gratitude to my mother, father-in-law, and mother-in-law for their support and understanding during these years. In the end, I am thankful for my beloved wife, Qiutong Liu, who has been accompanying me and taking care of our two-year-old boy, Muzhi Zhang, in Finland throughout my Ph.D. journey.

Helsinki, March, 2021
Chao Zhang

List of Original Publications

This dissertation is based on the following peer-reviewed original articles, which are referred to Articles I-V. The contributions of these articles and the author's contributions are described in Section 1.3.

Article (I) Chao Zhang. Parameter Curation and Data Generation for Benchmarking Multi-Model Queries. In *PhD workshop of the 44th International Conference on Very Large Data Bases (2018): 1-4.*

Article (II) Chao Zhang, Jiaheng Lu, Pengfei Xu, and Yuxing Chen. UniBench: A Benchmark for Multi-Model Database Management Systems. In *Proceedings of the 10th Technology Conference on Performance Evaluation and Benchmarking, pp. 7-23. Springer, Cham, 2018.*

Article (III) Chao Zhang, Jiaheng Lu. Holistic Evaluation in Multi-Model Databases Benchmarking. In *Distributed and Parallel Databases (2019): 1-33.*

Article (IV) Chao Zhang, Jiaheng Lu. Selectivity Estimation for Relation-Tree Joins. In *Proceedings of the 32nd International Conference on Scientific and Statistical Database Management, pp. 37-48. 2020.*

Article (V) Chao Zhang, Jiaheng Lu, Qingsong Guo, Xinyong Zhang, Xiaochun Han, Minqi Zhou. G-View: Automatic View Selection in Graph Databases. *Submitted to the 33th International Conference on Scientific and Statistical Database Management.*

Besides the above papers, which contributed to this thesis, the author of the present dissertation also participated in the following paper:

Article (VI) Lu, Jiaheng, Zhen Hua Liu, Pengfei Xu, and Chao Zhang. UDBMS: road to unification for multi-model data management. In *workshops of the 37th International Conference on Conceptual Modeling (2018): 285-294.*

The author of the present thesis participated in envisioning and designing the principles, technologies, and architecture of a unified database system. He also wrote a significant part of the paper.

List of Acronyms and Glossaries

AD	Ancestor-Descendant
ADR	Ancestor-Descendant Region
AI	Artificial Intelligence
AQL	ArangoDB Query Language
CDF	Cumulative Distribution Function
CG	Coding Graph
Choke Point	Technical points of the query design
CLVSC	Customer Lifetime Value in Social Commerce
Cross Model	A joint operation of multiple data models
CRUD	Create, Read, Update, Delete
DAG	Directed Acyclic Graph
DBA	DataBase Administrator
DBMS	DataBase Management System
DML	Data Manipulation Language
DPS	Dynamic Programming Selection
End-to-End System	A complete system from beginning to end
ETL	Extract, Transform, Load
FISSION	Break a view pattern to multiple graph genes
FUSION	Merge a set of patterns to a view pattern
GBA	Grid-Based Approximation
GGA	Graph Gene Algorithm
Graph Gene	A subgraph pattern of a view pattern
HTAP	Hybrid Transaction Analytical Processing
Join Selectivity	A ratio of the join size divided by the size of Cartesian products

JSE	Join Selectivity Estimation
KDE	Kernel Density Estimation
LHS	Latin Hypercube Sampling
LVE	Location-Value Estimation
MMA	Max-Min Approximation
MMDB	Multi-Model DataBase
MV	Materialized View
Native Store	A data store without the model conversion
OG	Original Graph
OLAP	On-Line Analytical Processing
OLTP	On-Line Transaction Processing
Parameter Curation	A method of configuring benchmark queries
Pattern Containment	Query pattern is contained
PC	Parent-Child
PH	Position Histogram
Polyglot Persistence	An approach of using polystores
Query Containment	Query results are contained
RDD	Resilient Distributed Dataset
RFM	Recency, Frequency, Monetary
RGDB	Relational-based Graph DataBase
RJ	Relational Join
RTJSE	Relational-Tree Join
SI	Subgraph Isomorphism
SQL	Structured Query Language
SRS	Simple Random Sampling
ST	Spanning Tree

Contents

Chapter 1

Introduction

One of the most challenging issues in the big data era is the "Variety" of data. The data may be produced in diverse formats - structured, semi-structured, and unstructured, hence can be represented as different data models in the databases. Examples of data models are relational, tree (e.g., XML, JSON), and graph (e.g., property graph, RDF). Recently, a number of native multi-model databases (MMDBs), e.g., ArangoDB [1], OrientDB [2], AgensGraph [3], have emerged. They particularly aim to manage the heterogeneous datasets and handle the hybrid workloads for modern Web-based applications such as customer-360-view and social commerce [4]. A fundamental difference between MMDBs and single-model databases is that MMDBs utilize a single database management system to store, manipulate, and query data in multiple models while single-model databases are meant to manage the data with one data model. That is, all the input data for single-model databases will be extracted, transformed, loaded to the data stored in one model, e.g., relational model. Then users query the data with a model-specific query language, e.g., SQL. With multi-model databases, users can import and query multi-model data *as is*, i.e., no schema is required when loading the data and queries can be specified with native query operators to access different types of data. Moreover, compared to the polyglot persistence [5] that employs separate databases in different types, e.g., SQL and NoSQL, to satisfy various use cases, the approaches of MMDBs reduce integration, migration, development, maintenance, and operational issues [6].

With the proliferation of MMDBs, the diversity of them also poses new challenges on their usability in multi-model data management [7]. On the one hand, a number of leading databases including SQL and NoSQL databases, have claimed that they can support the functionality of multi-model data management in a unified platform [6]. However, they dif-

fer fundamentally in data storage, query language, and query processing. This situation obstructs a direct comparison of them using existing tools. Therefore, users cannot appropriately choose the right type of MMDBs while respecting their applications. For instance, consider three MMDBs: ArangoDB, OrientDB, and AgensGraph, all of them are capable of managing graph, JSON, and tabular data, but their storage strategies and query processing techniques are quite different. Therefore, there is a need for a new benchmark tool to identify the pros and cons of their multi-model approaches. On the other hand, MMDBs feature a core multi-model query language, which allows users to quickly analyze their multi-model data with a single statement. However, this brings new issues for query optimization as traditional techniques are unsuitable for optimizing the queries due to the lack of considerations of multi-model operators and storage. The reason is two-fold. Firstly, unlike the traditional queries that only consider the operators from a single model, MMDBs can perform a multi-model query that involves operators from multiple data models, including the joining operations between the data of multi-model, e.g., the joins between relational tables and JSON documents. Thus new techniques are needed to optimize the multi-model query in a holistic way. Secondly, since MMDBs adopt the "single-store, multi-model" strategy, the single-model query has to be processed by model conversion and query translation when the query and data store are incompatible. This could lead to a significant performance overhead as the query is not performed using the native processing engine and data store. For instance, if a SQL query is embedded with a graph traversal, such a graph operator has to be translated to the relational operators with joins. The transformation could be expensive when the graph traversal involves multiple entities and relations.

This dissertation is divided into two parts for addressing the aforementioned three challenges accordingly. Firstly, we propose UniBench [8, 9], a generic benchmark that aims to evaluate the performance of multi-model databases based on a social commerce application. Its mixed data model contains five data models, i.e., JSON, XML, key-value, relational, and graph. It is an end-to-end tool consisting of three key components including synthetic data generation, workload specification, and parameter curation. We have leveraged UniBench to make a comprehensive evaluation of four representatives. We have identified the pros and cons of their multi-model approaches, as well as the common optimization issues in processing multi-model queries. Secondly, we studied the problem of join selectivity estimation of multi-model joins that involve predicates from relational and tree data models. We proposed three kernel density estima-

tion models to estimate the selectivity of relation-tree joins. The results have shown that our method can improve 80% accuracy over the state-of-the-art methods. Thirdly, we proposed an automated graph view selection tool, G-View, which utilizes views to speed up the graph queries for relational-based graph databases, thereby avoiding the query translation, model conversion, and costly joins for the relational engine. We proposed a filtering-and-verification framework to check the query containment by views. We formalized the view selection problem as an 0-1 Knapsack problem and developed a graph genetic algorithm that explores graph view set transformations to automatically select the views with a smaller view space and a higher view benefit. Extensive experiments demonstrated the constructed views by G-View can achieve two orders of magnitude query speedup in a relational-graph database, and the selection algorithm, GGA, outperforms other selection methods in both effectiveness and efficiency.

1.1 Motivation

Benchmarking is a common practice for evaluating and comparing the database management systems with a suite of software programs [8, 9, 10]. The database benchmark has a history of nearly four decades, and new benchmark systems have been emerging as DBMS technologies advance. The earliest database benchmarking tool can be traced back to the Wisconsin benchmark [11] in the early 1980s. Since then, various database benchmarks have been proposed. To name a few, TPC-C [12], TPC-DI [13], OO7 [14], XMark [15], YCSB [16], LDBC [17], and BigBench [18, 19]. However, previous benchmarks are not designed for MMDBs, thus they are unsuitable for benchmarking the MMDBs. For example, they can not generate the multi-model data and provide meaningful cross-model workloads, hence they are unable to simulate the realistic multi-model scenarios for evaluating MMDBs. This calls for a new benchmark for evaluating the performance of MMDBs and identifying the performance bottlenecks in the context of multi-model workloads. To address this problem, we develop the first benchmark, UniBench, for multi-model databases.

In addition to the database benchmark, MMDBs also lack the fundamental techniques aiming for multi-model query optimization [6, 8, 9]. The foremost is the join selectivity estimation [20] that can be used to optimize the joining order of a query. Accurate estimates of the join selectivity are critical as a query optimizer can utilize them to select the optimal plan [21]. However, since a multi-model query may involve joins across data of diverse models, the existing techniques, such as sampling-based methods

[22, 23, 24, 25, 26], model-based methods [20, 27, 28, 29], and histogram-based methods [30, 31, 32], fail to provide a satisfactory estimation quality because they mainly work for a single model case, e.g., relational or tree model. Therefore, it calls for a unified and effective approach to address the problem of selectivity estimation for multi-model join. Kernel Density Estimation (KDE for short) is a widely used method in the statistic and database field [20, 27, 33, 34]. In this dissertation, we investigate how to apply the KDE-based model to derive a selectivity estimation for relation-tree joins. Furthermore, we will discuss how to extend our approach to estimate the relation-graph joins.

As MMDBs adopt the "single-store, multi-model" storage strategy, they could have the performance overhead for answering the single-model query. This is because the query is not processed natively when the storage is implemented by an extension of another model. Such a situation is most notable for the Relational-based Graph DataBases (RGDBs), e.g., Oracle Graph [35], DB2 Graph [36], and SQLG [37], where the graph model is implemented over the relational store, thus a graph query is performed by using the costly joins over the relational store. What is worse, such cases can incur a significant performance overhead when a graph query involves numerous joins. In this dissertation, we study how to construct and select views to speed up the graph queries in RGDBs. Materializing view selection is a well-studied topic in DBMS [38, 39, 40, 41, 42, 43, 44, 45, 46, 47]. However, existing methods did not consider the setting of multi-model databases, which is quite different from previous settings, from cost evaluation to processing techniques. In addition, existing methods cannot fully exploit the graph properties of views, e.g., supergraph views and common subgraph views, which leads to a low view utility and the duplicate view content. To address these problems, we developed an end-to-end graph view selection tool, G-View, to automatically create, evaluate, and select graph views in the relational-based graph databases. We formalized the view selection problem as a 0-1 Knapsack problem and proposed a search-based algorithm, GGA, to judiciously select a view set based on a query workload.

1.2 Research questions and challenges

The scope of the present dissertation covers two aspects of multi-model databases: database benchmarking and query optimization. In the following, we present three specific research questions (RQ for short) and discuss their main challenges accordingly.

1.2.1 Research questions and challenges

RQ1: How do we benchmark multi-model databases?

The first research question focuses on the multi-model database benchmarking. There are four main challenges in benchmarking multi-model databases. Firstly, a database benchmark normally produces the synthetic and scalable data around a data model for testing the databases. When it comes to the multi-model cases, the challenge is to provide the data of variety and scalability simultaneously. Thus, we need to design a new mix model and a new data generator for providing multi-model data. Secondly, in the scenario with multi-model data, our goal is to design multi-model workloads concerning both technique and business dimensions to simulate the workloads in real cases. It is challenging due to the huge design space of the workloads. Thus, there is a need for new design principles to guide the workload design. Thirdly, given a query workload, it is a common practice to configure the query parameters, a.k.a., substitution parameters, to perform the query. However, for a multi-model query, different parameters result in the disparate size of intermediate results. Thus it calls for an effective way to control the parameter effect in order to give a fair comparison. Lastly, to verify the feasibility and usability of the proposed benchmark, it is crucial to implement our benchmark in the state-of-the-art multi-model databases. The challenge here is to investigate various query implementations and definitions due to the lack of standard and theoretical foundations of multi-model query processing.

RQ2: How do we tackle the selectivity estimation problem for relation-tree joins?

The second research question is about the selectivity estimation of multi-model joins, which is related to the multi-model query optimization. In this thesis, we focus on the relation-tree joins, e.g., joins across relational tables and JSON documents, but we also discuss its extension to the relation-graph join. There are four challenges to this problem. Firstly, unlike the traditional joins, e.g., relational joins, that are connected by the same unit of data, e.g., tuples, relation-tree joins are joining operations between tuples and trees. Thus, new definitions of the relation-tree join and join results are required. Secondly, since relational tables consist of tuples and trees contain values and structures, existing methods such as sampling-based and histogram-based methods, can hardly capture both

the values and structures. Thus it calls for a new estimation method that can take into consideration the value join and structural join simultaneously. Thirdly, the relation-tree join may have complex join structures, e.g., multiple value joins and twig joins. Therefore, it requires an effective algorithm to handle such complex cases. Lastly, when the data is huge, it is challenging to guarantee both the efficiency and effectiveness of the estimator. Hence, it calls for a new method to make the trade-off between efficiency and effectiveness.

RQ3: How do we construct and select the graph views in a multi-model database under a space constraint?

The third research question lies in the view construction and selection problem for multi-model databases. Given a graph query workload and a space budget, we aim to construct and select the most beneficial views in the relational-based graph databases within the budget. The view selection problem can be modeled as an optimization problem of maximizing the view benefit. We particularly formalize the selection problem as an 0-1 Knapsack problem, which is NP-hard. We consider the dynamic case in which the views can be changed, e.g., by merging, breaking, and removing views. There are four main challenges to this problem. Firstly, given a query workload, the challenge is how to construct the minimum number of views to cover the workload. Secondly, given a query and a view, the challenge is how to verify if the query can be answered by the view. Thirdly, given a graph query and a view set, when the query can be answered by multiple views, how do we select a view set for the query, and how do we evaluate the benefit for the view set? Fourthly, as the search space of view combinations is huge, and exploring the structural properties among views involves an NP-hard problem of subgraph isomorphism [48], it is impossible to enumerate all the options. Therefore, it calls for an efficient and effective algorithm to find an optimal view set.

1.2.2 The interrelations of the research questions

All the above three research questions are centered around multi-model databases, and their interrelations are as follows:

RQ1 is the foundation of RQ2 and RQ3 because (1) it provides the experimental basis, e.g., data and workloads, for the latter two; and (2) it identifies the performance bottlenecks of MMDBs, thereby motivating RQ2 and RQ3. Both RQ2 and RQ3 aim to optimize the multi-model queries, but they take different routes. RQ2 is to optimize the query execution plan

by addressing the selectivity estimation problem, while RQ3 is to leverage views to avoid the model conversion by tackling the view selection problem.

1.3 Thesis contributions

The present article-based dissertation has made both practical and theoretical contributions to multi-model databases. It consists of five peer-reviewed articles with four published (Article I-IV) and one submitted (Article V). In particular, Articles I, II, and III are devoted to addressing the question RQ1 by developing a multi-model database benchmark. Article IV answers RQ2 by proposing three KDE-based models. Article V addresses RQ3 by developing an end-to-end graph view selection tool with a new selection algorithm. The contributions of each article are summarized as follows:

Article I: *Parameter Curation and Data Generation for Benchmarking Multi-Model Queries*

This article [49] gave a general proposal for benchmarking multi-model queries. We discussed the motivations, challenges, and solutions to the problems of data generation and parameter curation in benchmarking multi-model queries. We also presented the preliminary results of the proposed solutions for these two problems.

Article II: *UniBench: A Benchmark for Multi-model Database Management Systems*

This article [8] proposed the first benchmark for multi-model databases based on a social commerce application, which includes five data models, i.e., graph, JSON, XML, key-value, and the relational model. We developed a new data generator on top of Spark SQL, which can provide correlated multi-model data with a given scale factor. Particularly, we proposed a three-phase framework to model the customers' behaviors in real cases for synthesizing realistic data. We designed a set of multi-model workloads that contain ten queries and two transactions by considering technical and business dimensions. Extensive experiments were conducted on two MMDBs: ArangoDB [1] and OrientDB [2].

Article III: *Holistic evaluation in multi-model databases benchmarking*

This journal article [9] was an extension of Article II. We addressed a problem of parameter curation that aims to configure the benchmark queries for performance evaluation. We proposed a sampling-based algorithm, MJFast, to efficiently configure the queries. We have proved

that MJFast is better than random sampling in terms of query diversity. Moreover, we conducted extensive experiments over four representatives: ArangoDB [1], OrientDB [2], AgensGraph [3], and Spark SQL [50]. Specifically, we investigated the distributed architectures of them and studied the multi-model query processing of four query languages, AQL, Orient SQL, SQL/Cypher, and Spark SQL. We leveraged UniBench to compare the performance of four chosen MMDBs in data importing, query processing, transaction processing, and distributed query processing. We have identified two common issues of multi-model query optimization for current MMDBs.

Article IV: *Selectivity Estimation for Relation-Tree Joins*

This article [51] studied a new join selectivity estimation problem for relation-tree joins, named the RTJSE problem, which can serve as a building block towards relation-tree query processing in modern databases. We proposed a new kernel-based estimation model called the location-value estimation (LVE) to tackle the RTJSE problem. We started with a simple case of relation-tree joins. Then, we extended it to more complicated cases involving twig joins. We proposed the max-min approximation (MMA) and grid-based approximation (GBA) to improve the estimation efficiency with guaranteed error bounds. We have conducted extensive experiments to compare the LVE model and its variants to the state-of-the-art methods. The results have shown that our KDE model significantly outperformed them in terms of accuracy, efficiency, and scalability.

Article V: *G-View: Automatic View Selection In Graph Databases*

This article [52] proposed an end-to-end graph view selection tool to automatically select views to accelerate the graph query processing in RGDBs. We proposed a query-driven method that automatically translates the graph queries to query patterns and creates the graph views by persisting all the edge-induced subgraphs. We proposed a filtering-and-verification framework, which checks the query containment by graph views and ensures the view content includes all the query results for contained queries. We proposed a view selection algorithm, named GGA, to select the views into the memory under a space budget, which explores various graph view transformations to search an optimal view selection. We conducted extensive experiments on various query workloads and datasets. The results have shown that the views generated by our tool achieved significant speedups compared to relational and graph store, respectively. Moreover, GGA outperformed other selection methods in both effectiveness and efficiency.

The contributions of the author. The present author is the first author and main contributor to all of the above articles. For Articles I, II, and III, the author collected the metadata, designed and developed the benchmark system, proposed and implemented the key techniques and algorithms, surveyed the state-of-the-art MMDBs, designed the experiments, and carried out the evaluation in collaboration with the rest of the authors. For Article IV, the author proposed the initial idea and carried out the main parts to all phases, from algorithm design, theoretical analysis to experimental evaluation. For Article V, the research problem came from a real industrial scenario. The author developed the original idea and was the main contributor of all parts including problem formulation, system design, and algorithm implementation, as well as the experimental evaluation. The articles are reprinted in the appendix of this thesis, and none of the articles have been used in previous dissertations.

1.4 Dissertation structure

The rest of the dissertation is structured as follows: in Chapter 2, we introduced all the components of the proposed benchmark for MMDBs in detail, including the components of data generation, workload specification, and parameter curation, as well as the benchmarking results and gained insights with UniBench. In Chapter 3, we focus on addressing the relation-tree join selectivity problem. We first introduced the preliminaries and problem definition. We then presented the KDE-based estimation models and algorithms for the RTJSE problem. We also discussed the extension to the relation-graph joins. In Chapter 4, we presented the view-based methods for accelerating graph queries in relation-based graph databases. We first introduced the preliminaries with a motivating example. We then presented the methods of view construction, view evaluation, and view selection. Finally, in Chapter 5, we concluded this dissertation with an extensive discussion on the scientific contributions and research findings. Furthermore, we also discussed open challenges and future directions for advancing multi-model databases.

Chapter 2

A benchmark system for MMDBs

This chapter presents a benchmark system for multi-model databases. Particularly, Section 2.1 introduces the background of multi-model databases and related work in database benchmarking; Section 2.2 gives an overview of the proposed end-to-end benchmark, including data generation, workload specification, and parameter curation; Sections 2.3 presents the process of data generation; Section 2.4 introduces the workload specification and the principles for workload design; Section 2.5 introduces the problem of parameter curation for configuring the benchmark queries, and then presents a sampling-based algorithm to address the problem. Finally, Section 2.6 introduces four systems under test and the benchmarking results with UniBench.

2.1 Background and related work

In this section, we introduce the background and related work on benchmarking multi-model databases. We first briefly introduce the multi-model databases in Subsection 2.1.1, then present the related work on database benchmarking in Subsection 2.1.2.

2.1.1 The background of MMDBs

Multi-model databases (MMDBs) are proposed to address the "*Variety*" challenge of data with a single data management platform and a core declarative query language. The idea of MMDBs can be traced back to the concept of ORDBMS (i.e., Object-Relational DataBase Management Systems) [6] in the early 1980s, which can store and process various formats of data such as relational, JSON, XML, and object in the relational databases by leveraging domain-specific functions and indexing. The query languages

are mostly developed based on SQL query languages such as SQL/XML [53] and SQL/JSON [54]. In the early 2010s, the proliferation of NoSQL MMDBs, e.g., OrientDB, ArangoDB, Redis, push forward this approach and have shown that the non-relational models like graph, JSON, and key-value models can also be the first-class citizen of MMDBs. Without loss of generality, we define the scope of multi-model databases in this thesis as those database management systems supporting the multi-model storage and query processing with a single platform.

Generally, there are three classes of multi-model databases. The first class is the extended multi-model database, which provides the multi-model functionalities based on the original single-model database. For instance, Oracle [54] is originally a relational database. It extends to a multi-model database supporting JSON by storing the JSON data into a column, then exploits SQL/JSON path language as a cross-model query language. Redis [55] is a well-known key-value database which has become a multi-model database after adding JSON and graph modules on top of its key-value store. The second class is the native multi-model database, e.g., ArangoDB [1], OrientDB [2], AgensGraph [3]. The main differences between them and the extended multi-model databases are two-folds. Firstly, they are designed for multi-model data management at the very beginning, and they treat data of each supported model as the "first-class citizen", so they have no explicitly primary model when managing the data. Secondly, they have developed new query languages, e.g., AQL, SQL/Cypher, and Orient SQL, to query the multi-model data with more flexible and expressive power. Nevertheless, they also have the primary model in terms of the implementation. For example, ArangoDB's primary model is the JSON model, and it implements the index-based adjacency in JSON documents to support the graph data management. OrientDB is based on the graph model, which can embed a JSON document as a graph node. The third class is the Big data system. A representative is Spark SQL [50], which stores the multi-model data in distributed files, e.g., HDFS, and provides a uniform data access to deal with multi-model data based on a unified *DataFrame* representation, along with many domain-specific functions and libraries for handling JSON, graph, and tabular data.

It is worth noting that many cloud platforms such as CosmosDB [56], Datastax [57], DynamoDB [58], employ multiple databases with specialized APIs, e.g., MongoDB API, Gremlin API, and etcd API [56], to support diverse models. However, these kinds of systems manage the databases independently and cannot support the cross-model query processing. Therefore, they are currently excluded from the scope of MMDBs in the thesis.

Table 2.1: Classification of database benchmarks.

Data model	Representatives
Relational	TPC-C, TPC-E, TPC-H, SSB, TPC-DS
Object	HyperModel, OO1, OO7, Bucky
XML	XOO7, XMach, XBench, XMark
KV, Column, Document	YCSB, YCSB++
Spatial	DynaMark, COST, BerlinMod
Big data	MRBench, HiBench, BigBench
Graph	LinkBench, RBench, LDBC
Other	CH-Benchmark, JOB, Sysbench, TPC-DI

2.1.2 Related work

Database benchmarking plays an important role in comparing the performance of relevant database systems. Moreover, benchmarking the databases can accelerate their development progress and make database technologies viable. Numerous database benchmarks have been proposed for various purposes. They vary greatly in the application domain, target system type, data model, workload specification, and performance metric.

Table 2.1 shows the representatives of database benchmark categorized by data models. The relational model is the most important data model in the field of databases because of the success of commercial RDBMSs. The mainstream of benchmarks for the relational model is the TPC-backed specification, such as TPC-C [12] and TPC-E [59] with On-Line Transaction Processing (OLTP) workloads, TPC-H [60] and TPC-DS [61] with On-Line Analytics Processing (OLAP) workloads for data warehouses. There also emerged a host of benchmarking work for object databases because of the prevalence of the object model between the 1980s and 2000s. To name a few, HyperModel [62], OO1 [63], OO7 [64], and Bucky [65]. During the 2000s-2010s, a number of benchmarks, such as XOO7 [66], XMach-1 [67], XBench [68], XMark [15] were proposed to evaluate the XML query processing. Meanwhile, several spatial-based benchmarks were developed for evaluating spatio-temporal databases, particularly for spatial indexing, like DynaMark [69], COST [70], and BerlinMod [71]. Since the 2010s, the YCSB [16] benchmark has been widely used to measure the read-write performance of various distributed cloud database systems with NoSQL models such as KV (Key-Value), column, and document models. One of its extensions, YCSB++ [16], was developed later to evaluate some advanced

features in BigTable-style stores. At the same time, a series of benchmarks have emerged for evaluating the distributed systems using the Map-Reduce paradigm, such as MRBench [72], HiBench [73], and BigBench [18]. In the year of 2013, graph-based benchmarks such as LinkBench [74], LDBC [17] and RBench [75] were proposed to evaluate graph traversals and complex graph analysis. Particularly, LDBC has currently developed three types of benchmarks for graph data systems, namely, the graphalytics benchmark [76] for graph analysis platforms, the Social Network Benchmark (SNB) [17] for graph databases, and the Semantic Publishing Benchmark (SPB) [77] for RDF-based semantic databases. There are other benchmarks proposed for particular purposes. For instance, CH-Benchmark [78] combined TPC-C and TPC-H into a hybrid benchmark aiming for testing the Hybrid Transaction Analytics Processing (HTAP). JOB [21] is a well-known benchmark for evaluating join ordering of a relational query plan. SysBench [79] is a micro-benchmark for evaluating the CPU, I/O and memory performance when performing concurrent database operations. TPC-DI [13] is a benchmark that features an input dataset with structured data, i.e., CSV files, semi-structured, i.e., XML documents, and unstructured data, i.e., texts, and its aim is to evaluate the transformation cost in the process of data integration.

Unfortunately, existing benchmarks are not suitable for benchmarking multi-model databases due to the lack of (i) a holistic data model bridging SQL and NoSQL models, (ii) a multi-model and scalable data generator, and (iii) the cross-model workloads. Several evaluation efforts [80, 81] have been done on multi-model databases recently. Nevertheless, they only focus on simple workloads, such as CRUD operation, aggregation, graph depth traversal, which are inadequate since they do not account for complex workloads concerning multi-model characteristics. This motivates us to develop a new benchmark for multi-model databases.

2.2 The key components of UniBench

In this section, we briefly introduce three key components of UniBench, namely, *data generation*, *workload specification*, and *parameter curation*.

The first component is the *data generation*. Specifically, our generation is based on a new mix model in a scenario of social commerce [4]. Such a scenario includes five data models, i.e., graph, relational, JSON, key-value, and XML. We model the data in an empirical way for the chosen scenario. For example, the social network is modeled as a graph, the customer information is modeled as a relational table, and the order information is

modeled as a JSON file. Please find the detailed schema description in Article III [9]. The workflow of data generation is as follows: (1) we first collect the metadata from Amazon review [82], DBpedia data [83], and the LDBC graph data [17], we then correlate the metadata from these three data sources. (2) we leverage a three-phase framework to generate the multi-model data by simulating the customer's behaviors in the scenario of social commerce [4].

The *workload specification* component includes multi-model queries and transactions. Specifically, it consists of ten queries and two transactions for evaluating the multi-model query processing. The design principles of the workloads consider both technique and business dimensions. For the business dimension, we designed four business levels, namely, individual, conversation, community, and commerce to cover various business cases. From a technical perspective, the queries are designed according to the *choke point* [17] methodology which has taken into consideration three choke points for multi-model query processing, including multi-model join ordering, aggregation, and transaction.

The *parameter curation* component is used to configure the binding parameters for the query. The cornerstone of this component is the MJ-Fast algorithm. Specifically, it first characterizes the multi-model query by identifying the parameters and corresponding intermediate results. Then it generates the size vectors associated with each parameter group. Finally, it selects the k curated parameters for the multi-model query based on the Latin hypercube sampling (LHS) [84] method.

2.3 Data generation

In this section, we introduce the process of data correlation and data generation with more details. Particularly, the data correlation aims to combine the collected metadata. The data generation is based on a three-phase framework including purchase, propagation purchase, and re-purchase for simulating the users' behaviors in the social commerce.

2.3.1 Metadata correlation

We collect the metadata by combining data from Amazon review [82], DBpedia dataset [83], and LDBC graph data [17], we then correlate the metadata. Specifically, we correlate the person table in the LDBC dataset and the country table in the DBpedia dataset by using the same country seed, and correlate the country table in the DBpedia dataset and the product table in the Amazon review data by using the same brand seed. Moreover,

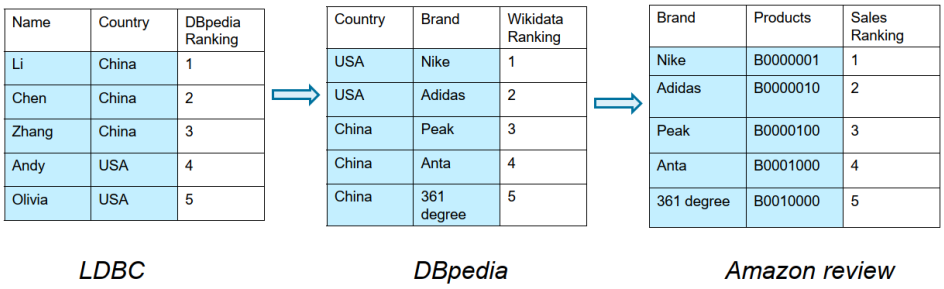


Figure 2.1: Data Correlation.

we leverage the real data distribution to rank the values, that is, DBpedia ranking for the countries, Wikidata ranking for the brands, and sales rankings for items. With such correlations, we generate the CDF distribution using Zipf distribution, and then randomly generate the multi-model data. As shown in Figure 2.1, the values in metadata are correlated: China correlates with the Chinese names, brands, and products, while the USA correlates with the American names, brands, and products. As a result, the generated data gives the realistic data content and distributions.

2.3.2 Purchase

In this phase, we generate the data based on the persons' interests. These interests are generated based on the person's location and gender, and they are a set of tags associated with products with the one-to-one relationship. Furthermore, we obtain the number of transactions for each person based on the number of their interests, we then assign the interested items to each transaction and randomly choose the feedback of the items, we finally generate the transaction data using Spark SQL. Consequently, the generated data in this phase consist of three parts: (1) a simplified social network graph from LDBC, (2) the correlated vendor, brand, and product data from the metadata, and (3) the synthesized transaction data including customers, orders, invoices, and feedback.

2.3.3 Propagation-purchase

In this phase, we propagate customer purchase behaviors to their friends. We select the top-k items for the target persons and generate the new data based on a scoring function. This is motivated by the observation that people with same attributes more likely have the same behaviors, and

people also trust the product recommendations from friends. In particular, the scoring function is defined as follows:

$$S_{ui} = \sum_{k} k \times Pr(R_{ui} = k | A = a_u) + E(R_{vi} : \forall v \in N(u)) \tag{2.1}$$

where the former part is the expectation of the target user u's rating on the target item i, and the latter part is the expectation of u's friends $N(u)$'s rating on the target item i; $A = \{a_1, a_2, ..., a_m\}$ is u's attribute set and k is the scoring scale from 1 to 5.

2.3.4 Re-purchase

In this phase, we propose a CLVSC model to model the customers' re-purchase behaviors, which is defined as follows:

$$\begin{aligned} S_{ib}(CLVSC) = {} & E(X^* \mid n^*, x', n, m, \alpha, \beta, \gamma, \delta) \\ & \times (E(M \mid p, q, \upsilon, m_x, x) + E(S \mid \bar{s}, \theta, \tau)) \end{aligned} \tag{2.2}$$

where i and b are the customer and brand index, respectively. In the CLVSC model, $E(X^* \mid n^*, x', n, m, \alpha, \beta, \gamma, \delta)$ is the expected number of the behaviors, $E(M \mid p, q, \upsilon, m_x, x)$ is the expected monetary value, and $E(S \mid \bar{s}, \theta, \tau)$ is the expected social engagement of the customer. The detailed description of the model and the variables can be found in the journal paper [9].

2.3.5 Implementations of the data generator

To offer the scalability of the data generation, we implemented the data generator using Spark SQL, which not only can generate the data in parallel with RDD partitions, but also can serialize the data with a number of IO APIs for generating JSON, XML, and CSV data. Moreover, we leveraged the scikit-learn library for training the Bayes model in the propogation-phase, and used BTYD package [85] for training the CLVSV model.

2.4 Workload specification

The workload of the proposed benchmark comprises ten multi-model queries and two transactions. In particular, the ten queries are categorized to four main levers involving data of multi-model: *individual*: {Q1}, *conversation*: {Q2, Q3}, *community*: {Q4, Q5, Q6}, and *commerce*: {Q7, Q8, Q9, Q10}. A key feature of these queries is that each query involves at least two data

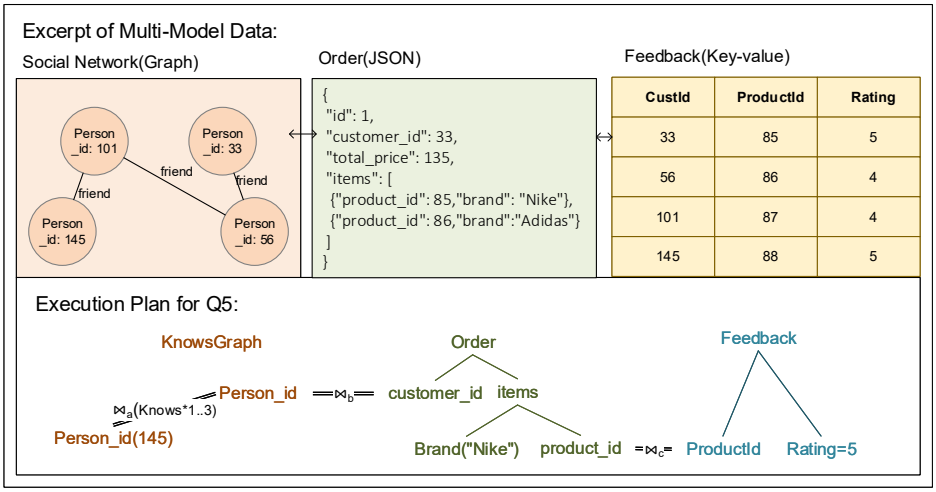


Figure 2.2: Join ordering of multi-model query.

models, along with different operators. For example, Q2 will join data from a relational table, graphs, and JSON files. Q5 joins data from relational, graphs, and key-value, with recursive graph traversal, embedded document filtering, and relational joins. Moreover, in these four levers, a host of business cases in different granularity are included, such as customer 360 view in Q1, product recommendation in Q2, Q4, Q5, Q6, sentiment analysis in Q3, Q7, performance transparency in Q8, Q9, and return analysis in Q10. The detailed descriptions of the workloads are presented in Article II [8].

We consider three *choke-points* from the technical perspective: (i) join ordering, (ii) complex aggregation, and (iii) cross-model consistency. As for (i), it requires the query optimizer to find an optimal join order for the given multi-model query. Let us take Q5 as an example: *Given a customer and a product brand, return this customer's friends within 3-hop friendship who have bought products in the given brand, and these products have the 5-rating reviews by the friends.*. As shown in Figure 2.2, the logic plan below the data contains three types of joins, i.e., 3-hop graph joins, graph-JSON join, and JSON-table join. This requires the query optimizer to accurately estimate the selectivity of cross-model joins and judiciously determine the optimal join order. Concerning (ii), it demands the processing engine to handle the complex aggregation of multi-model data. For instance, Q10 first performs an aggregating operation on the graph data, then it conducts an RFM aggregation over the JSON data. For (iii), it tests the ability of the execution engine to handle the read-write multi-model transactions

while ensuring consistency and efficiency. For instance, both T1 and T2 transactions involve read and write operations on JSON files, tables, and XML data.

2.5 Parameter curation

In this section, we introduce the component of parameter curation. We first discuss the problem of parameter curation for configuring the multi-model queries in Subsection 2.5.1, we then present an efficient solution, MJFast algorithm, in Subsection 2.5.2.

2.5.1 The problem of parameter curation

Once we have finalized the multi-model queries, we need to configure the *substitution parameters* of them for query evaluation, e.g., the customer id and brand name for Q5. This procedure is called *parameter curation* [86] in benchmarking. A naive solution is to randomly sample the parameter values to feed the queries. However, different *substitution parameters* would lead to contradictory benchmarking results due to the skewed data distribution. This observation motivates us to develop a new method to address such a problem. Intuitively, we aim to configure a query with diverse parameter values for introducing the query diversity. With such diversity, we can conduct a holistic evaluation of the multi-model query processing. However, unlike the parameter curation in the single-model query such as SQL, where the intermediate results involve tuples, multi-model data has different types of intermediate results. To deal with such a challenge, we formally defined the intermediate results associated with each group of parameter values as size vector, and each element of a size vector is a dimension of the intermediate results, e.g., graph size, JSON size, or their join size. We then defined the query diversity as the sum of the size vectors' minimum pair distances. Finally, we formalized the problem as multi-model parameter curation, with the goal of maximizing the parameter diversity. Please find the detailed definition in Article III [9].

2.5.2 MJFast algorithm

The problem of multi-model parameter curation is computationally costly if all the size vectors need to be calculated. Moreover, finding the k size vectors with maximum parameter diversity is NP-hard, which can be reduced from the K-center problem. To address these problems, we propose a new algorithm, called MJFast, based on the Latin Hypercube Sampling

(LHS) [84] which is a multi-dimensional stratified sampling method. The detailed description of MJFast is given in Article III [9]. We summarize the algorithm in the following three steps:

1. Identify the base dimension for each data model and compute the intermediate result size regarding each base dimension.

2. Employ the LHS method to select the size vectors, which divides the size range of each base dimension into k segments, then sample k size vectors from the d-dimensional buckets.

3. Map the size vectors to the corresponding parameter groups for configuring the queries.

The major advantage of the MJFast algorithm is two-fold: (1) we only compute the base dimension of the size vectors to measure the intermediate results, which can greatly reduce the computation cost of calculating all the dimensions, and (2) we can evenly choose values of each dimension by using the LHS method, which guarantees the diversity of the query parameters. As a result, the complexity of MJFast is O(nlogn+n*c), where n is the number of size vectors and c is the average cost of computing a partial size vector. We have also theoretically proved that MJFast has a higher query diversity than random sampling with the probability $1 - \frac{(k-1)^{k-1}}{k^3}$, where k is the configured size of parameters.

2.6 Benchmarking MMDBs with UniBench

We have conducted a holistic evaluation of MMDBs. After investigating a variety of MMDBs and their underlying techniques, we choose four representatives, namely, OrientDB, ArangoDB, AgensGraph, and Spark SQL. In the following, we introduce them with respect to the internal data representation, system architecture and query processing, we then summarize the benchmarking results. The examples of query implementation and the detailed evaluation results are presented in Article III [9].

2.6.1 OrientDB

OrientDB [2] is a native multi-model database that supports multi-model query processing, transactions, sharding, and replication. Its primary model is the graph model implemented by the index-free adjacency structure in

which every vertex has the direct pointers to its neighbors. OrientDB utilizes the object record to represent the data that can be a document, a bytes record (BLOB) and a vertex, or even an edge. In such a way, it can support the data models that include document, graph, object, and key-value models simultaneously. For the query processing, it develops a SQL-like language called *OrientDB SQL* with extensions to enable the navigational path and graph functionality. Regarding the distributed architecture, OrientDB has a multi-master distributed architecture that is built on the Hazelcast framework [87].

2.6.2 ArangoDB

ArangoDB [1] is a native multi-model database that supports multi-model queries, transactions, sharding, and replication. ArangoDB is built centering around a JSON model in which JSON documents are organized into typed collections. Furthermore, the graph model is implemented by storing the vertex collections and edge collections separately, with indices built on the associated keys. ArangoDB has developed a brand-new query language, called AQL, which is a pure data manipulation language (DML). Along with a variety of functions, operators, and patterns, AQL supports querying and modifying the graph, JSON, key-value data in one statement. Regarding the distributed architecture, ArangoDB has a multi-role distributed architecture including the agent, coordinator, and database server. In particular, the agent is an instance that stores the cluster configuration. A coordinator is an instance that connects the clients and the cluster, as well as schedules the tasks. A DB server stores the data and executes the AQL queries locally.

2.6.3 AgensGraph

AgensGraph [3] is another multi-model database that is built on top of a widely used relational database, PostgreSQL, with a focus on the graph extension. AgensGraph implements the graph model using the index-based adjacency structure where graphs are represented in vertex tables and edge tables with indexed primary and foreign keys. Since it is highly compatible with PostgreSQL that already supports the relational, XML, and JSON, AgensGraph can blend the SQL-extension language and the graph query language Cypher to query multi-model data at the same time. Unfortunately, AgensGraph has no particular distributed architecture, meaning that users have to shard their data manually when the dataset becomes large.

2.6.4 SparkSQL

Apache Spark [88] has become the de facto standard of distributed computing framework for large-scale data analytics. Interestingly, one of its components, Spark SQL, [50] has been growing to be a multi-model data processing engine. Particularly, Spark SQL now can process multi-model data, including relational, graph, key-value, and JSON data based on a wealth of libraries. A unified data abstraction is the *DataFrame* structure which relies on the immutable, in-memory, distributed, and parallel capabilities of RDD. Regarding the distributed architecture, Spark has a master/worker architecture including a driver program, a cluster manager, and multiple workers. A driver program is used to configure the cluster properties. A cluster manager is responsible for allocating the resources and scheduling the jobs for the workers. A worker node performs the actual tasks using one or more executors.

2.6.5 Summary of the benchmarking results

By benchmarking the chosen four MMDBs with UniBench, our main findings are as follows:

1. For data importing, AgensGraph is 1.5x and 9x faster than ArangoDB and OrientDB, respectively.

2. Regarding query processing, OrientDB is best at two point multi-model queries (Q1 and Q2), and two path-oriented queries (Q5 and Q6), ArangoDB excels at multi-model join queries (Q3 and Q4), AgensGraph is the winner of the multi-model aggregation queries (Q7, Q8, Q9, Q10).

3. For transaction processing, AgensGraph is 3.4x and 10x faster than ArangoDB and OrientDB because of its advanced relational engine.

4. Concerning the distributed mode, Spark SQL excels at fast data ingestion and parallel aggregation, but is an order of magnitude slower in the join tasks due to the shuffle joins and the lack of local indices.

2.7 Discussions

Through benchmarking MMDBs with UniBench, we have verified the applicability and feasibility of UniBench. With its rich data formats and multi-model query workloads, we have not only found the pros and cons of chosen MMDBs as shown in the benchmarking results, but also identified the common performance bottlenecks of them in querying the multi-model data. First, MMDBs cannot select the optimal join order due to the lack of cross-model selectivity estimation. Since MMDBs are in their infancy, their query optimizers are not as mature as the relational optimizers. Most of their query optimizers rely on the one-dimensional histogram to optimize the query, leading to inaccurate cardinality estimation and poor join plans. Second, MMDBs can incur a significant overhead because of the model conversion. As MMDBs use their primary models to implement all the supported models, when the query becomes complicated, the cost of model conversion is substantial. We will present our solutions to tackle these two problems in Chapter 3 and 4, respectively.

2.8 Chapter summary

In this chapter, we introduced the background of multi-model data management and related work on database benchmarking. Then we presented our benchmarking system, UniBench, for multi-model databases. Specifically, we first presented an overview of the UniBench system. Following that, we described the key techniques of three components including data generation, workload specification, and parameter curation. In particular, we introduced a three-phase framework for data generation, the design principles for the workload, and an efficient algorithm for addressing the problem of parameter curation. Finally, we introduced four chosen MMDBs and summarized the benchmarking results.

Chapter 3

Selectivity estimation in MMDBs

Throughout the benchmarking process, we have a key observation: although the state-of-the-art MMDBs can support multi-model joins, such as graph-JSON, JSON-relational, and graph-relational joins, they lack specific methods and algorithms to optimize the execution plan. The reason is that they simply utilize one-dimensional histograms or random samples to estimate the query selectivity and generate the execution plan. However, they do not take into consideration the case that a query may involve operators of various data models, hence unable to holistically optimize the query. This issue motivates us to study the problem of selectivity estimation in MMDBs for a holistic query optimization.

This chapter studies the problem of selectivity estimation for relation-tree joins, which is crucial for the query optimization with both relational and tree data. It first introduces the background on relation-tree joins, then presents a KDE-based solution, and finally summarizes the experimental results. It also contains a discussion of how to extend our approach to the case of relation-graph joins by the end of the chapter.

3.1 Introduction

Storing and accessing both relational and tree data without the ETL process can benefit many applications, ranging from integrated health-care services [89], online recommender systems [90] to smart traffic management [91]. Nowadays, many MMDBs can leverage a mix query such as AQL [1], SQL/JSON [54] and SQL/XML [53], to perform the online analysis of the relational and tree data simultaneously. One of the salient features of them is the relation-tree join that includes both value and structural joins. We use the following example to illustrate the relation-tree join.

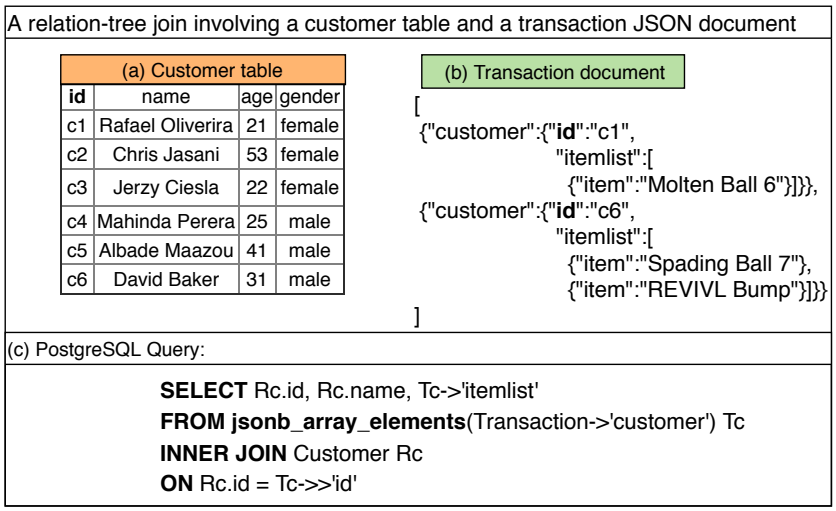


Figure 3.1: The motivating example of a relation-tree join involving a customer table and a transaction JSON document. Below the datasets is the join query written in SQL query with JSON path expressions of the PostgreSQL database.

Example 3.1 *Given the customer table and transaction document shown in Figure 3.1. A meaningful query here is to find the customers' profile information in the table with their purchased items from the document for the business analysis on customers' purchase behaviors. Particularly, we implement the query using SQL with path-oriented expressions for JSON (see Figure 3.1c). Note that the relation-tree join is achieved by joining the tuples in R_c with an array of subtrees T_c on the common key id.*

The problem of join selectivity estimation has been extensively studied in the relational databases [20, 22, 24, 28, 29, 92, 93] and in the XML databases [25, 26, 31, 94, 95]. In the relational databases, Vengerov et al. [22] proposed the correlated sampling to estimate the join size subject to filter conditions. Kostas et al. [28] leveraged a graphical model to estimate a joint distribution of all attribute values. Kipf et al. [29] recently adopted a deep learning model, i.e., convolutional neural network, to learn the join correlations among data. In the XML databases, Luo. et al. [25] developed a subtree sampling to estimate the twig query selectivity, which samples the subtree via a predefined sampling ratio, then derives approximate answers from the sampled tree. Wu. et al. [95] proposed a histogram-based approach named PH-histogram, which constructs two-dimensional histograms over the tree data to estimate the structural join

size. Zhang. et al. [26] proposed the multi-layer tree synopses to summarize the structural information for twig query selectivity estimation. However, straightforwardly combining the relational and structural estimators fails to capture the join correlations among data, hence unable to obtain high-quality estimates. For example, naively computing the join contribution from the relational and tree samples often leads to a biased and error-prone estimation due to the missing of join partners in the samples. Thus it calls for a unified and effective method to address the selectivity estimation problem for relation-tree joins.

KDE is a widely used method in statistic and database field [20, 27, 33, 34]. Kiefer et al. utilized the KDE models to estimate the selectivity of range query [27] and join query [20] in a relational database. Particularly, they implemented the KDE estimators using graphics cards to speed up the KDE computation. In [33], a KDE-based estimator was proposed to estimate the cardinality of skyline query. Zhang et al. [34] studied the approximations for KDE in large data. This paper differentiates from previous KDE-based works in that we apply the KDE model to the RTJSE problem, which essentially results in a new KDE estimator.

3.2 Problem definition

In this section, we formally define the relation-tree join and the selectivity estimation problem. We give the definitions of value join and structural join separately, then we define the relation-tree join based on them. Furthermore, we define the problem of selectivity estimation.

3.2.1 Relation-Tree Join (RTJ)

A relation R has a set of attributes $A = \{A_1, ., A_n\}$. Each tuple $t \in R$ is a *n-ary* tuple denoted by $t =< v_1, ., v_n >$ where v_i is the value corresponding to attribute A_i from the domain $\mathrm{dom}(A_i)$.

A tree is a *rooted labeled tree* $T = (r, N, K, V, E, L)$, where r is the unique root node; N is a finite set of labeled nodes having at most one incoming edge; K is a set of keys, V is a set of values, and both are from an alphabet Σ; $E \subseteq N \times N$ is a set of edges, in which each edge (n, n') points from a parent node n to a child node n' that models a parent-child (PC) relationship; L is a function $f : N \to K \times V$ such that for each node $n \in N$, and $L(n)$ is a key-value pair $(k, v) \in K \times V$.

Definition 3.1 (Value Join) *A cross-model value join operation is denoted as $R \bowtie_\theta T$ where R is a relation with a set of attributes A; T is*

a tree that includes a set of keys K; θ is the value join predicate on the intersection of A and K, requiring that the attribute's name and value of tuples in R must be equal to the key and value of tree nodes in T. The join result is a table, which consists of a set of unified tuples combining the involved attributes in R and keys in T.

Definition 3.2 (Structural Join) *The structural join is a twig pattern matching $\gamma = (N, E, f)$ where N is a set of labeled query nodes; $E \subseteq N \times N$ is a set of edges; f is a binary function modeling the edge type between two nodes, which includes the single-child-axis (/), the branch-child-axis ([]), and the descendant-axis (//). The join result is a set of distinct twig pattern matches, requiring any binary functions in γ is satisfied simultaneously.*

Definition 3.3 (Relation-Tree Join) *Formally, given a relation R and a tree $T = (r, N, K, V, E, L)$, a relation-tree join is defined as $R \bowtie_\eta T$, where the relation-tree join predicate is $\eta = \theta \wedge \gamma$ that combines the cross-model value join predicate θ and structural join predicate γ.*

Example 3.2 *According to the above definition, the relational-tree join in the motivating example can be represented as follows:*

$$R_c \bowtie_{(R_c.id=T_c.id)\wedge(\exists x.k=customer, y.k=id, z.k=item, x[y]//z)} T_c \tag{3.1}$$

As shown in Figure 3.2, the above join is a two-way relation-tree join where the relation R_c has a set of attributes $A = \{id, name, age, gender\}$, and the tree T_c contains a set of keys $K = \{customer, id, itemlist, item\}$. Specifically, the relation-tree join consists of a cross-model value join and a structural join. The cross-model value join is on the key id that corresponds to the attribute id of relation R_c and key id of nodes in tree T_c, respectively. The structural join is a twig pattern matching (see Figure 3.2b).

3.2.2 The problem of selectivity estimation for RTJ

Given a relation-tree join Q over a relation R and a tree T. Our goal is to predict Q's join selectivity denoted by $\Phi(Q)$, which has the following formula:

$$\Phi(Q) = \frac{|R'(Q) \bowtie T'(Q)|}{||R| \times |T||} \tag{3.2}$$

where the numerator is the relation-tree join size, and the denominator is the size of the relation-tree Cartesian product.

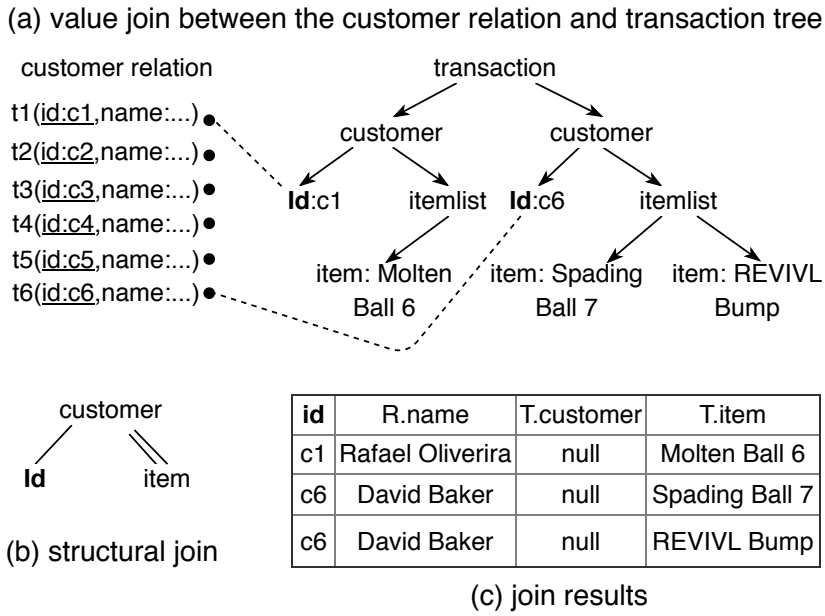
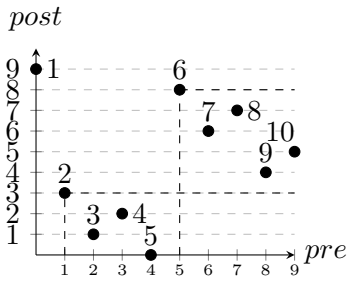


Figure 3.2: Illustration for a relation-tree join and join results.

3.3 The KDE-based models

Kernel Density Estimation (KDE) is a data-driven, non-parametric method to estimate probability densities based on a data sample [96]. KDE model is inherently a data sample and has no assumptions on the underlying data. Thus it is generally more robust than parametric techniques. In particular, KDE has been widely used to estimate the selectivity in the database research literature [20, 27, 33, 34]. These research efforts have shown that KDE converges faster to the underlying distribution than histogram approaches do and can consistently offer superior estimation quality to the purely sampling approach. Given a random sample $S = \{\overrightarrow{u}^{(1)}, \ldots, \overrightarrow{u}^{(s)}\}$ of size s from a d-dimensional dataset $D \in \mathbb{R}^d$, the kernel estimator assigns a probability density to each point $u^{(i)} \in S$ such that $\hat{p(u)} : \mathbb{R}^d \to \mathbb{R}$. Finally, it averages the KDE contributions $\hat{p(S)}$ of sample S to derive an estimate $\Phi(S)$. Particularly, the KDE contribution $\hat{p(x)}$ of a point x is defined as follows:

$$\hat{p(x)} = \frac{1}{S} \sum_{u \in S} K(\frac{x-u}{h}) = \frac{1}{S \cdot h} \sum_{u \in S} K_h(x-u) \tag{3.3}$$



	pre	*post*	*level*	*(key:value)*
1	0	9	0	Transaction:null
2	1	3	1	customer:null
3	2	1	2	id:c1
4	3	2	2	itemlist:null
5	4	0	3	item:Molten Ball 6
6	5	8	1	customer:null
7	6	6	2	id:c6
8	7	7	2	itemlist:null
9	8	4	3	item:Splading Ball 7
10	9	5	3	item:REVIVL Bump

Figure 3.3: Pre/post plane and the corresponding node encoding table for the tree data of Figure 3.1b.

where the function K is the kernel function, which defines the shape of the local probability distributions; h is the smoothing factor called bandwidth which controls the scope or width of the kernel distributions.

3.3.1 Location-Value Estimation (LVE) model

We propose a new kernel-based estimation model for the RTJSE problem. Our basic idea is to build the KDE model on the relational and tree samples, then obtain an estimator combining the value joins and structural joins. More specifically, we first build an index over the tree data to encode the structural and value information. Then, we sample the tuples from the base tables and sample the tree nodes from the tree index. Finally, we apply the location-value estimation (LVE) model to the data sample to derive a joint KDE estimator including the structural join and value join.

In order to estimate the structural join and value join simultaneously, we extend a previous tree encoding technique [97] to build an index upon the tree data. Figure 3.3 shows the pre/post encoding which makes an AD relationship between node a and d satisfying the formula $a.pre < d.pre \wedge a.post > d.post$. Our key idea is to integrate and sum up the kernel density over the descendant region of each ancestor, then average the KDE contribution as the join contribution. Consider a two-way relation-tree join as follow:

$$Q = R \bowtie_{(R.A=T.A)\wedge(\exists x.k=A,y.k=D,A//D)} T \tag{3.4}$$

where A is the join key for a value join between a relation R and a tree T, and nodes with key A should be the ancestors of those with key D in T.

For an individual join value v and a pre/post numbering, m and n, we can express its true join size as follow:

$$\begin{aligned} &|R| \cdot p_R(A.v = v) \cdot |T_A| \cdot p_{T_A}(A.v = v \wedge A.pre < m \wedge A.post > n) \\ &\cdot |T_D| \cdot p_{T_D}(D.pre > m \wedge D.post < n) \end{aligned} \tag{3.5}$$

In this equation, we express the AD pattern nodes T_A and T_D separately and use function $p(c)$ denotes the exact selectivity for a predicate c.

As illustrated in Figure 3.4a, for a sample ancestor node, only the region of its lower right includes the descendant nodes according to the numbering formula. We define such regions as ADR (Ancestor-Descendant Region). Motivated by this observation, we propose a point-wise method to approximate the contribution by integrating the kernel density over the ADR.

By integrating the Gauss distribution of each sample over the $ADR_p^{(j)}$ and grouping by the common factor of the sum of ancestor samples $\sum_{j=1}^{s_2}$, the estimator of Location-Value Estimation (LVE) model, arrives at:

$$\begin{aligned} \hat{\Phi}(Q) = \frac{1}{s_1 \cdot s_2 \cdot s_3} \sum_{j=1}^{s_2} \Bigg(\sum_{i=1}^{s_1} \Big(\mathcal{N}_{r^{(i)},(\delta_r^2+\delta_t^2)}(t^{(j)}) \Big) \\ \cdot \sum_{k=1}^{s_3} \Bigg(\frac{1}{2} \Big[erf(\frac{u_{pre} - q_{pre}^{(k)}}{\sqrt{2} \cdot h}) - erf(\frac{l_{pre} - q_{pre}^{(k)}}{\sqrt{2} \cdot h}) \Big] \\ \cdot \frac{1}{2} \Big[erf(\frac{u_{post} - q_{post}^{(k)}}{\sqrt{2} \cdot h}) - erf(\frac{l_{post} - q_{post}^{(k)}}{\sqrt{2} \cdot h}) \Big] \Bigg) \Bigg) \end{aligned} \tag{3.6}$$

where erf : $\mathbb{R} \to \mathbb{R}$ denotes the *error function*, which is defined as $erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x \exp -t^2 dt$; u_{pre} and l_{pre} (resp. u_{post} and l_{post}) denote the upper and lower bounds of the ADR region of j-th sample point $p^{(j)}$. In particular, u_{pre} is the maximum preorder numbering; l_{pre} is equal to $p_{pre}^{(j)}$; u_{post} is equal to $p_{post}^{(j)}$ and l_e is 0. Please refer to Article IV [51] to see the detailed derivation of the above equation.

3.3.2 Max-Min Approximation (MMA) Model

To improve the estimation efficiency of the LVE model, we propose the max-min approximation (MMA) model to approximate the KDE contribution with less computation. As shown in Figure 3.4b, the *max* and *min* points are two vertices (left upper and right lower) on the left diagonal of a grid. Suppose a grid contains a set of ancestor samples A, and the join

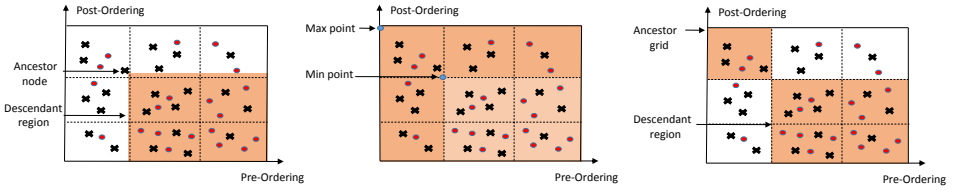


Figure 3.4: An illustration of the proposed KDE models.

contribution of *max* and *min* points are J_u and J_l, respectively. Then we derive Lemma 3.1 and the following estimator:

$$\hat{\Phi}_{MMA} = J_\theta \cdot \frac{1}{|g|} \cdot \sum_{i=1}^{g} \sum_{j=1}^{g} \frac{(J_u - J_l)}{2} \tag{3.7}$$

Lemma 3.1 *Given a set of points P, and a grid of size $g \times g$, the method MMA gives an approximate kernel density estimate KDE_G of P, such that, for each binary pattern, we have $|KDE_P - KDE_G| \leq \epsilon$, where ϵ is equal to $\sum_{i=1}^{g} \sum_{j=1}^{g} \frac{|n_g| \cdot (J_u^{g_{i\times j}} - J_l^{g_{i\times j}})}{2}$.*

3.3.3 Grid-Based Approximation (GBA) Model

To consider the overlap property of data, e.g., customer nodes can have the descendant customer nodes, we also propose a grid-based approximation (GBA). Figure 3.4c demonstrates the main idea of GBA. The intuition is that since ancestor nodes can have many overlap descendant nodes, we can substantially reduce the number of estimations by grouping the points to grids and then estimating the join contribution for an ancestor grid and its corresponding descendant grids. The KDE contribution consists of four cases: (i) the diagonal AD case between two regions, i.e., an ancestor grid with its lower-right region; (ii) an ancestor grid with its lower region; (iii) an ancestor grid with its right region; (iv) an overlapping grid with both ancestor and descendant nodes. With a uniform assumption, we derive Lemma 3.2 and the following estimator:

$$\hat{\Phi}_{GBA} = J_\theta \cdot \frac{1}{|g|} \sum_{i=1}^{g} \sum_{j=1}^{g} J_g^A * \left(J_{G^{g^{lr}}}^D + \frac{1}{16} J_g^D + \frac{1}{4} \cdot (J_{G_g^r}^D + J_{G_g^l}^D) \right) \tag{3.8}$$

Lemma 3.2 *Given a set of points P, and a grid of size $g \times g$, for an arbitrary grid $g_{i\times j}$ ($i < |g|$ and $j < |g|$), the upper gap bound of GBA's KDE ϵ_g is $J_{g^{lu}}^A \cdot (J_{g^{rl}}^D + J_{G_g^r}^D + J_{G_g^l}^D)$.*

Due to the limited space, the readers are recommended to see the detailed proofs of the lemmas, the details of the estimators of the KDE models, as well as the estimation algorithms in Article IV [51].

3.3.4 The estimation procedure

Summarizing our approach, we can construct a KDE-based estimator and apply it to estimate the join selectivity by the following four steps:

1. Encode the tree data using the pre/post numbering method.

2. Choose a set of representative relation-tree joins. Initialize the bandwidth using Scott's rule [96] and train the value join bandwidth h_v and structural join bandwidth h_l separately.

3. Collect a random sample of size s for the tree nodes and relational tuples involved in the query.

4. Apply the KDE models, e.g., LVE, MMA, GBA, to the samples to obtain a KDE estimate. Depending on the requirement, when users would like to have an accurate estimate, the LVE is recommended. If users need a fast estimate, MMA is preferable. Or, if users want to make a trade-off between efficiency and accuracy, GBA is a good choice.

3.4 Summary of the experimental results

We compared our models with three estimators: (1) Simple Random Sampling (SRS), (2) Position Histogram (PH) [95], and (3) Join KDE (RJ-KDE) [20]. We used both synthetic and real-life datasets including the `DBLP`, `XMARK` , `LDBC` and `UniBench` datasets. We have designed 12 relation-tree joins with structural and equality predicates. For the synthetic relations, each relation has the form: `relation name (join key)`. For the tree pattern, we consider the different types of structural joins, including the AD relationship, path-based query, and twig queries.

We summarize the experimental results as follows:

1. Our LVE-KDE join estimator significantly outperformed other estimators in accuracy and efficiency. Particularly, LVE-KDE improves 80%,

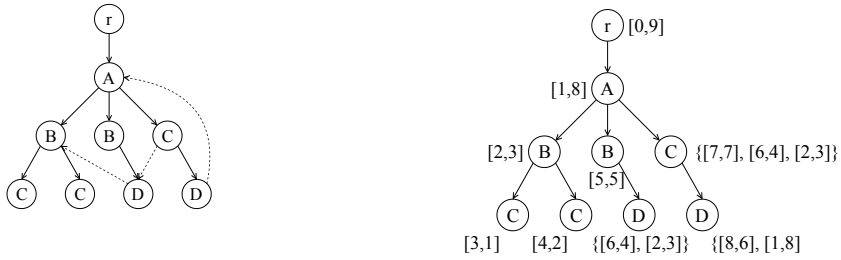
82%, 96% of estimation accuracy compared with RJ-KDE, SRS, and PH, respectively. Moreover, LVE-KDE is 5.1×, 1.2× faster than PH and SRS respectively, and faster than RJ-KDE by an order of magnitude.

2. Our two KDE approximators can speed up the LVE-KDE 4× and 2×, respectively. Regarding the estimation accuracy, MMA-KDE improves 65%, 67%, 93% of accuracy over RJ-KDE, SRS, and PH, respectively. GBA-KDE improves 70%, 71%, 94% of accuracy over RJ-KDE, SRS, and PH, respectively.

3.5 An extension to relation-graph joins

In this section, we briefly discuss how to extend our approach to the graph case. Similar to structural join in the tree data, the structural join in the graphs has two binary relationships: *direct connected* and *indirect connected*. Intuitively, if we manage to represent the graph as a tree, we then can readily extend the tree coding scheme to work on graphs. After that, we can apply our method to the relation-graph case as well. However, two problems hinder this extension. First, a graph may contain a cycle, thus a node can be reachable to its parents. Second, even for a directed, acyclic graph (DAG), a node may have more than one parent after representing the DAG as a tree. Thus, the tree-coding schema does not suffice to infer the full transitive closure in a graph due to these two problems. To overcome this problem, we propose a polymorphic labeling approach to encode the graph which has two characteristics: (i) one node could have multiple numbering labels, and (ii) one numbering label could represent multiple nodes. To encode the graph data, we utilize the tree-coding scheme to label the nodes in a spanning tree ST that is derived from the original graph OG, and then duplicate the label for nodes that lead to the above-mentioned two problems. As a result, the nodes' labels can be used to determine if two nodes are connected. Finally, applying the location-based KDE model to the problem of relation-graph JSE is straightforward.

Example 3.3 *Figure 3.5(a) shows an OG that (i) contains a cycle, i.e., $(A \rightarrow C \rightarrow D \rightarrow A)$, and (ii) has two nodes B and D owning multiple parents, i.e., $A \rightarrow (B) \leftarrow D$ and $B \rightarrow (D) \leftarrow C$. Figure 3.5(b) depicts a Coding Graph (CG) that is labeled by the polymorphic approach. In specific, it first labels the ST that excludes the dotted edges in the OG. Then it replicates the label of a node B to D, D's label to C, and A's label to D in succession. Consequently, we can infer the reachability of two nodes by Definition 3.4. For instance, three nodes: D with labels {[8,6],[1,8]}, D with*



(a) Original Graph (OG) that includes a root r, labeled vertices, and edges

(b) Coding Graph (CG)

Figure 3.5: Encoding for the graph data

labels {[6,4],[2,3]}, C with labels {[3,1]}, are reachable from node C with label {[7,7],[6,4],[2,3]} according to conditions (i),(ii),and (iii), respectively. Note that the first element in the set is the tree label.

Definition 3.4 (Reachability of the Graph coding) *Given a directed graph $\mathcal{G}$ labeled by the graph coding in which each vertex v associates a tree label t_v and a set of extended non-tree labels S_v. Two nodes $s, e \in \mathcal{G}$ are reachable from s to e iff they hold one of the following conditions:*

(a) $t_s.pre < t_e.pre \wedge t_s.post > t_e.post$,

(b) $S_s \supset (t_e \cup S_e)$, *or*

(c) $\exists l_s \in S_s$: $l_s.pre < t_e.pre \wedge l_s.post > t_e.post$.

3.6 Chapter summary

This chapter studied a new problem of estimating the join selectivity of relation-tree joins (RTJ). We formally defined the problem of selectivity estimation for RTJ. We then proposed the LVE model, a kernel-based approach to deal with the estimation problem by considering both value joins and structural joins. Furthermore, we proposed two approaches, MMA and GBA models, to approximate the KDE contribution of the LVE model while improving the estimation efficiency for large data samples. We also discussed how to apply our method to estimate the selectivity of relation-graph joins.

Chapter 4

View selection in MMDBs

As MMDBs adopt the "single-store, multi-model" storage strategy, they may have performance overhead for answering the single-model query. This is because the query is not processed natively when the storage is implemented by an extension of another model. This issue motivates us to study the problem of automatic view selection in MMDBs.

This chapter studies the problem of view selection in MMDBs, with an emphasis on the graph query optimization over relational and graph data. It first introduces the background and limitations on current relational-based graph databases, then presents the related work on materialize view selection. Furthermore, it presents our methods for view creation, evaluation, and selection in detail. Finally, it summarizes the experimental results and gained insights.

4.1 Introduction

As the property graph model and graph query languages are rapidly gaining popularity in graph data management, a number of leading relational databases have enabled both graph technologies over a relational store, including Oracle Graph [35], DB2 Graph [36], and SQLG [37], etc. We refer to them as Relational-based Graph DataBases (RGDBs for short). Particularly, they can support two kinds of storage back-ends: the relational store and in-memory graph store. Unfortunately, both approaches have limitations in answering the graph queries. For the former one, RGDBs implement the property graph model using underlying relational tables, translate the graph queries to SQL dialects, and execute the SQL in a relational engine. For instance, the SQLG [37] system takes a Gremlin query and translates its Gremlin pipelines into a combined SQL query with

pre-defined SQL templates. However, such approaches have performance overheads in answering the ad-hoc graph queries due to computing the graphs from the relational store at runtime. For the latter one, they rely on DBA's or experts' knowledge to manually create the in-memory graphs, thus are inefficient for large workloads. For instance, Oracle graph uses the In-Memory Analyst (PGX) [35] to read graphs into memory from the underlying tables. What is worse, it is unclear whether or not the extracted graphs have included all the results for the queries since they are built empirically.

To bridge this gap between the availability of graph analytics in a relational engine and the need for efficient analytics using native graph query languages, we propose a view selection tool, G-View, which can judiciously generate a view set from a query workload by automatically transforming the candidate graph views and taking into consideration their efficacy. Specifically, given a graph query set and a space budget, G-View translates each query to a candidate view pattern and checks the query containment via a filtering-and-verification framework. G-View then selects the views using a graph gene algorithm (GGA), which relies on a three-phase framework that explores graph view set transformations to reduce the view space and optimize the view benefit. Finally, G-View generates the extended graph views that store all the edge-induced subgraphs to answer the subgraph and supergraph queries simultaneously. Extensive experiments on real-life and synthetic datasets demonstrate the constructed views by G-View can significantly speed up the queries for graph databases, and its core selection algorithm, GGA, outperforms other selection methods in both effectiveness and efficiency.

This chapter is based on Article V [52]. The main objective is to optimize the graph queries in RGDBs using extended graph views as such a situation is most notable for RGDBs. Nevertheless, our method is also applicable to optimize the queries in other graph databases, e.g., Neo4j. The reason is that our method is based on a general framework for Gremlin queries. As long as the graph databases support the Gremlin query processing, our system can be easily incorporated into them without modifying their underlying implementations.

4.2 Related work

Materializing view selection is a well-studied topic in relational [38, 39, 40, 41, 42, 98, 99, 100, 101], XML [43, 44, 102], semantic [103, 104], and streaming databases [47]. Various methods are proposed to select the ma-

terialized views for different target queries, e.g., SQL and XQuery [105]. Particularly, Microsoft AutoAdmin [40] is a view selection wizard for SQL Server databases. IBM DB2 Advisor [101] is another tool that recommends the views, indices, and partitionings for a given workload. Chaves et al. [39] encoded the relational views as genes and applied the gene algorithm to the view selection problem in the setting of distributed databases. In [41], the authors modeled the view selection problem as an ILP (Integer Linear Programming) problem. They developed a reinforcement learning method to select subqueries to materialize. There has been a host of work on processing XML queries using views [43, 44, 102]. In [102], the authors studied the XML view selection in relational tables, they formalized the problem as a set-cover problem and developed a greedy-based algorithm to address the problem. In [44], the authors studied the view selection problem for XPath workloads, they proposed a greedy-based solution that makes the space/time trade-off. Katsifodimos et al. [43] studied the view selection for XQuery workloads. They first developed a greedy-based algorithm for a Knapsack selection problem, then proposed a heuristic algorithm to search for an optimal view set based on multi-view rewriting. There has also been work for RDF view selection [104, 106]. Goasdoué et al. [106] solved the view selection problem as a search process. They proposed heuristic strategies to search for a set of reformulated RDF views to minimize the defined cost model. Unfortunately, none of these works considered the structural properties of graph queries in view selection, thus they cannot be applied to the graph view selection problem directly.

There has been work on graph view selection. Particularly, Fan et al. [45] studied the minimal and minimum containment problem in view selection. Kaskade [46] modeled the view selection problem as an 0-1 Knapsack problem, then used a branch-and-bound solver to select the graph views. However, there are two major limitations to existing methods. Firstly, they only select views with the subgraph patterns to answer the queries while they do not consider using a view with a supergraph pattern to answer the contained queries. This leads to a low utility of the materialized views. Secondly, they cannot effectively explore the possible candidate view combinations to reduce the view space and improve the view benefit. Our method is different from existing methods because we take into account the subgraph/supergraph views, view transformations, and view combinations, yielding a view set with a smaller view size and a higher view benefit. Moreover, we study the graph view selection in RGDBs, which are quite different from previous settings, from cost evaluation to processing techniques.

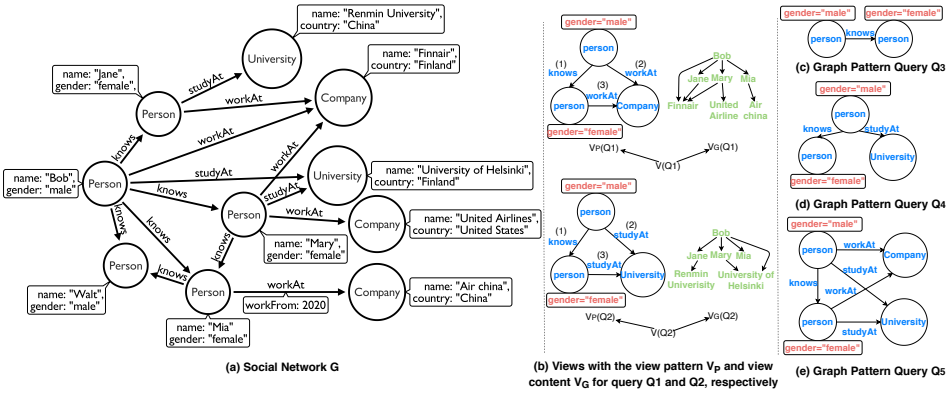


Figure 4.1: A property graph, extended graph views, and pattern queries.

4.3 Motivation and problem definition

In this section, we use a motivating example to introduce the preliminaries. We then define the problem of view selection. Please refer to Section 2 of Article V to see the formal definitions of the preliminaries.

4.3.1 A motivating example

Figure 4.1a shows a social network graph G from LDBC [107], which is a property graph that consists of three labels of vertices, i.e., person, university, and company, and three labels of edges, i.e., knows, studyAt, and workAt. Each vertex or edge has empty, one, or two properties. Figure 4.1b illustrates two extended graph views $V(Q1)$ and $V(Q2)$ with view patterns and view content. Figure 4.1c depicts three graph pattern queries $\{Q_3, Q_4, Q_5\}$. It can be seen that (1) $V_P(Q_2)$ contains Q_3 and Q_4, and (2) Q_5 is contained by $V_P(Q_1) \cup V_P(Q_2)$. Existing methods fail to answer queries Q_3 and Q_4 because they do not consider using the view $V_P(Q_2)$ to answer Q_3 and Q_4 despite $V_P(Q_2)$ being a supergraph pattern of them. We propose an *extended graph view*, which is able to answer the subgraph and supergraph queries simultaneously. Therefore, our methods ensure all the three pattern queries can be answered by $V(Q1)$ and $V(Q2)$ without accessing the graph G. Furthermore, we propose a view selection algorithm, named GGA, to select the views under a space budget, which explores various options of graph view transformations to find an optimal view set. For instance, we can merge two views $V(Q1)$ and $V(Q2)$ to reduce the common traversal of *knows* edges.

4.3.2 View selection problem

Given a query workload Q and a space budget S, we aim to automatically select an optimal view set $\mathcal{V}$ to materialize under the budget S. Therefore, the view selection problem can be modeled as a Knapsack problem of maximizing the view benefit under the space budget. We adopted a setting where the materialization cost is approximated by the view size. Such a setting assumes a cost model of view materialization that is proportional to the view size. Formally, the problem is defined as follows:

> Given a query workload Q and a space budget S, the problem is to select a set of views $\mathcal{V}_s$ derived from a candidate view set $\mathcal{V}$ that fully covers the query results of Q. The objective is to maximize the total benefit of $b(\mathcal{V}_s, Q)$, under the constraint that the total space occupied by $\mathcal{V}_s$ is less than S.

The view-selection problem is NP-hard [108] even for the special case in which $\mathcal{V}_s$ is a subset of $\mathcal{V}$, and each view $V \in \mathcal{V}_s$ is independent so that each query $q \in Q$ is answered by a single view $V \in \mathcal{V}_s$. For such static cases, there is a straightforward reduction from the Knapsack problem: find a set of k items having the space occupancy $s_1, \ldots, s_k$ and the benefits $b_1, \ldots, b_k$ so as to maximize the sum of the benefits of the selected items that fill the space budget S. Moreover, there could be the dynamic cases in which the views in $\mathcal{V}_s$ can be changed, e.g., by merging, breaking, and removing views. The problem in such cases becomes harder since the space of the potential view sets is extremely huge and it is unfeasible to explore all possible combinations. In the work, we propose a graph gene algorithm to address the problem in the dynamic setting.

4.4 View construction, evaluation, and selection

In this section, we present the methods of view construction, view evaluation, and view selection. Particularly, we introduce the view construction in Subsection 4.4.1, including the construction methods of view pattern and view content. We then present the methods of view evaluation in Subsection 4.4.2. Finally, we propose the Graph-Genetic Algorithm (GGA) for view selection in Subsection 4.4.3.

Example 4.1 *Consider a query in a social network of LDBC [107], which finds the male persons' female friends, and the companies the friends worked at, as well as the universities the friends studied at. The corresponding Gremlin query is expressed as follows:*

```
g.V().has('gender','male').as('p').match(
__.as('p').out('knows').as('f').has('gender','female'),
__.as('f').out('workAt').as('c'),
__.as('f').out('studyAt').as('u'))
.select('p','f','c','u')
```

where the above Gremlin query is a pattern query $Q_G = (V_p, E_p, L, f)$ which defines a set of nodes V_p and edges E_p in the `match` *step. Particularly, each* `as` *step refers to a query node v with a unique alias that has the mapping label $L(v)$; each* `has` *step defines a Boolean predicate $f(v)$ with a key-value pair; each* `out` *step declares an outgoing labeled edge e; the* `select` *step returns all the matched vertices.*

4.4.1 View construction

In the following, we introduce the edge-induced view construction, including the construction of view pattern and view content. The view patterns are derived from the given queries, and the view content is constructed based on the view patterns.

View pattern construction

Given a candidate query set Q, we translate the queries to a pattern query set, then leverage an edge-induced method to construct a candidate view for each pattern query. Particularly, for a pattern query $Q_G \in Q$, we parse it to Gremlin traversals and derive the traversal patterns E_p; we then add each query edge $e \in E_p$ with the predicates to its view pattern $V_P(Q_G)$ in succession. Since the query node v and edge e are labeled with a given alias, the procedure will also map the alias label to the label $L(v)$ and $L(e)$ in the schema graph. Consider Example 4.1, we derive the *knows* edge as the first traversal and add it to the view pattern $V_P(Q_G)$, we then sequentially add the *workAt* and *studyAt* edges. Finally, we will map the aliases { *p, f, c, u* } to labels { *person, person, company, university* } that are inferred from the schema graph.

View content construction

To construct the view content $V_G(Q_G)$, we create an edge-induced graph by the following steps: (i) we traverse each edge $e \in E_p$ in the traversal order as the view pattern $V_P(Q_G)$'s. (ii) for each visited query edge e, we add all the matched results of edges $E(e)$ in the property graph G with their endpoints $V(e)$ to the view content $V_G(Q_G)$. (iii) the procedure terminates when all the patterns have been visited. For Example 4.1, our

method will add all the matched edges and vertices to the view content according to the traversal order of $[knows, workAt, studyAt]$. Regarding the implementation in practice, since the matched results of an edge depend on its previous traversals, we clone the previous traversals and cache the visited endpoints as the intermediate results and use them to compute the matches of the subsequent traversals. As a result, the selected graph views will be materialized in the format of GraphML [109], which is an XML-based representation of a graph.

4.4.2 View evaluation

In this subsection, we propose a filtering-and-verification framework to check the query containment by views for the evaluation of view benefit. Furthermore, we introduce how to evaluate the view benefit for a Gremlin query workload. We also propose a new method for the multi-view evaluation as well.

A Filtering-and-Verification Framework

The filtering-and-verification framework consists of two stages. The first stage will check if a pattern query Q_G is contained by a view pattern $V_P(Q'_G)$. Otherwise, the query will not be evaluated on view V. The second stage will further verify if the view content $V_G(Q_G)$ contains all the matched results of the given query. Intuitively, the first stage checks the containment between a query pattern and a view pattern, and the second stage verifies the containment between query results and the view content.

1) The filtering stage. This stage checks if a query Q_G is a subgraph pattern of a view pattern $V_P(Q'_G)$. It is known that finding all the subgraph isomorphism mappings [48] is NP-hard, but there exist several practical algorithms to decide the answers in polynomial time. In this work, we employ the VF2 algorithm [110] that runs in quadratic time for checking a pattern containment between Q_G and $V_P(Q'_G)$. Note that the original VF2 algorithm does not consider the edge labels and predicates in the graph, thus we will check if the labels and predicates are the same after a subgraph isomorphism M is returned.

2) The verification stage. This stage verifies if a query Q_G can be answered by the view content $V_G(Q'_G)$. It checks if the following conditions hold: (i) there exists a mapping M from each edge pattern $e \in E_p$ to the edge pattern $e' \in E'_p$. (ii) for the edge $e' \in E'_p$ that has no mapping from $e \in E_p$, if the vertex $v'_e \in e'$ has the mapping from $v_e \in e$, the node $v'_e = M(v)$ must have occurred in prefix traversal patterns of E'_p.

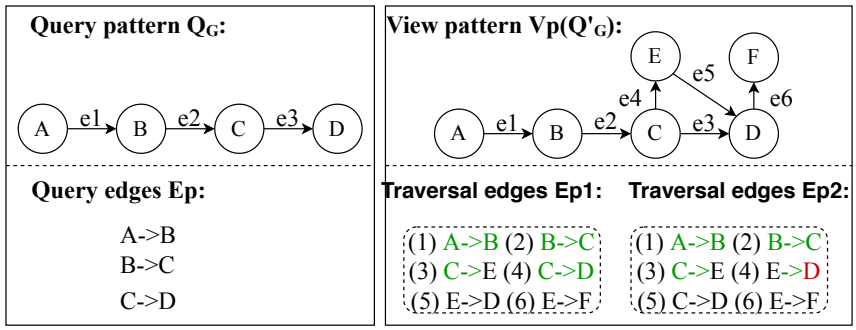


Figure 4.2: Containment between a query pattern and views.

We have proved the filtering-and-verification framework gives a sufficient condition to determine the query containment between the query Q_G and the view V. In addition, we design a view-based verification algorithm (VVA) to efficiently decide whether or not a pattern query Q_G can be answered by the view $V(Q'_G)$. Please refer to Article V to see the detailed proof and the detailed description of the algorithm.

Example 4.2 *Consider a pattern query Q_G and a view pattern $V_P(Q'_G)$ given in Figure 4.2. It is clearly visible that $Q_G \subset V_P(Q'_G)$, thus the framework returns true in the filtering stage. One can verify that the E_p and E_{p1} satisfy the verification conditions since all the mappings of edges and vertices of E_p have been visited in the prefix traversal order of E_{p1}. However, E_p and E_{p2} fail to satisfy condition (ii) in the second stage because the edge $e_5(E \rightarrow D)$ is not an edge mapping from E_p, and the query node D, which is a vertex mapping $M(v_{e3})$, has not been visited in E_{p2}. Therefore, it cannot guarantee that all the matched vertices of node D are included.*

Benefit evaluation

There are two cases when evaluating the view benefit, namely, single-view evaluation and multi-view evaluation. The former one involves only a single view, and the latter one considers multiple views. Before introducing the evaluation methods, we first give the definitions of the view overhead and benefit as follows.

Definition 4.1 (Overhead of a view) *The overhead of a materialized view includes the space overhead $s(v)$ and the computation overhead $o(v)$ of generating the view. In particular, we define $s(v)$ as the byte size by a view v, and $o(v)$ as the CPU time and I/O cost for constructing a view v.*

Definition 4.2 ***(Benefit of a view)****: Given a query workload Q, the benefit b of a view V is defined as the total cost savings by processing the queries using view V compared to using the relational-based graph store G_R:*

$$b(V, Q) = \sum_{q \in Q} (w_i \times (cost(q|G_R) - cost(q|V))) \tag{4.1}$$

where w_i is the weight or frequency of query q_i in Q; $cost(q|G_R)$ and $cost(q|V)$, denote the cost of query evaluation over G_R and view V, respectively. The $cost(q|G_R)$ is calculated over the underlying relational store by translating query q to a SQL query and evaluating it over the relational engine.

Definition 4.3 ***(Benefit of multiple views)****: Given a query q, the benefit b of a multi-view set $\mathcal{V} = \{v_1, v_2, \ldots, v_m\}$ is defined as the cost savings by using the view set $\mathcal{V}$ compared to using G_R:*

$$b(\mathcal{V}, q) = cost(q|G) - (\sum_{V \in \mathcal{V}} cost(q|V) + cost(V_1 \bowtie \ldots \bowtie V_n)) \tag{4.2}$$

where $cost(q|G_R)$ denotes the query cost over the relational-based graph store G_R; cost $\sum_{V \in \mathcal{V}} cost(q|V)$ is the sum of their partial evaluation cost and $cost(V_1 \bowtie \ldots \bowtie V_n)$ is the combining cost;

1) Single-view evaluation. For the single-view case, we evaluate the benefit $b(q, V)$ according to Equation 4.1. Particularly, we use the PROFILE feature [109] of Gremlin to obtain the cost of query evaluation. The PROFILE step returns various metrics about the given Gremlin queries including the result size, count of traversals, and total execution time in each pipeline. We perform the PROFILE step over the view V and over the G_R, respectively, we then take the total execution time as the cost and compute the benefit according to Equation 4.1.

2) Multi-view evaluation. For the multi-view evaluation, we first propose a two-level search algorithm to find the minimal view set $\mathcal{V}'$ that can answer Q_G, we then evaluate the benefit according to Equation 4.2. The two-level search algorithm checks if Q_G is contained by the candidate view $V \in \mathcal{V}$ in the first level, then explores the graph genes that assemble a supergraph pattern of Q_G in the second level. In the worst case that the last view is returned in the second level, the algorithm runs in $O(|\mathcal{V}|(|E'_p| + |E'_p| * |g|^2))$, where $|\mathcal{V}|$ denotes the number of views in the view set $\mathcal{V}$, $|E'_p|$ is the maximum edge size of the view $V_i \in \mathcal{V}$, and $|g|$ denotes the number of genes of the given query Q_G. The details of the algorithm is given in Algorithm 2 of Article V.

After a view set $\mathcal{V}'$ that minimally contains a query Q_G is returned, we evaluate the total benefit $b(\mathcal{V}', Q_G)$ and assign the benefit to the view set $\mathcal{V}'$ as follows:

1. Compute the view cost by summing the total partial evaluation cost $\sum_{V \in \mathcal{V}} cost(Q_G|V)$ and the cost $cost(V_1 \bowtie \ldots \bowtie V_n)$ for combining the intermediate results.

2. Calculate the benefit by subtracting the view cost from the query cost, $cost(Q_G|G_R)$.

3. Assign the benefit to each $V \in \mathcal{V}$ in proportion to its cost: $cost(Q_G|V)$. That is, we derive the corresponding gene $g_{Q_G} \in Q_G$, then evaluate g_{Q_G} over view V. In addition, we measure the combining cost by joining the views on the articulation points.

4.4.3 View selection

Since a view may be contained by another view, and views may have the common parts, the duplication of selected views leads to a relatively larger space occupancy and a lower coverage of the whole workload space. Based on this observation, we propose the GGA algorithm to achieve a higher usage of space and a higher benefit for the workload as a whole.

Graph-Genetic Algorithm (GGA)

The GGA algorithm solves the view selection problem as a state search problem. Every state consists of a set of selected views and a benefit. The initial state corresponds to the input candidate view set $\mathcal{V}$ with a zero benefit $b(\mathcal{V}_0)$. By merging, breaking, and removing views from the initial state, we obtain another state from view set $\mathcal{V}'$ with a new benefit $b(\mathcal{V}')$.

GGA encodes a view pattern, a.k.a., *individual*, by a set of subgraph view patterns, a.k.a., *graph genes*. Particularly, GGA has three atomic behaviors for view pattern transformations, namely, FISSION, FUSION, and REMOVE. A new *generation*, a.k.a., candidate view set, is generated by a process of probabilistic view transformations and a solution is produced based on their *fitness* value, a.k.a., view benefit. In the following, we introduce the view transformations in detail.

FISSION transformation splits a view pattern to multiple genes. The main goal of this transformation is to enable the identification of common

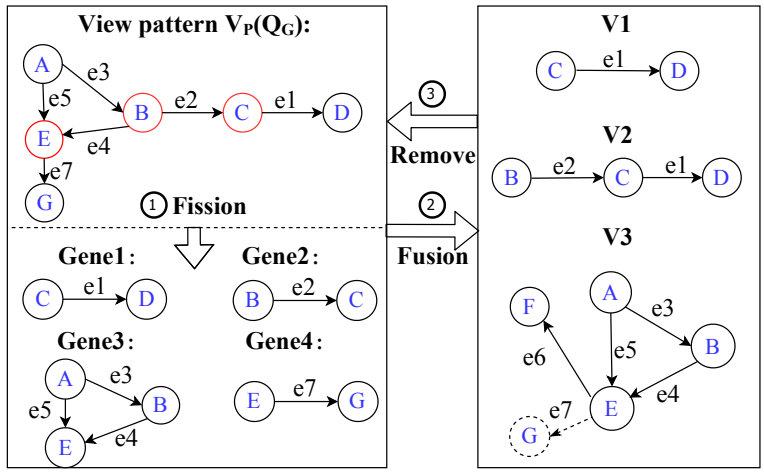


Figure 4.3: An illustration of GGA algorithm.

parts across views heuristically as finding the common subgraphs for the graphs is an NP-hard problem. In particular, we find the articulation points of a view pattern by using the Tarjan Algorithm [111], then obtain multiple graph genes by breaking down the view pattern according to its articulation points. The articulation points are vertices whose removal increases the number of connected components of the graph, and Tarjan Algorithm [111] is a (Depth-First-Search) DFS-based approach that can run in $O(V + E)$ time to compute the articulation points in a directed graph. Note that if the articulation point does not exist, the original view pattern becomes the graph gene itself.

FUSION is opposite to FISSION, namely, this transformation merges or joins a view $V_i \in \mathcal{V}$ to other views $V_{j\neq i} \in \mathcal{V}$. Particularly, FUSION has two variants: (i) merge the sub-views $V_i \subset V_j$: Fusion merges the view V_i to $V_{j\neq i} \in \mathcal{V}$ if V_i is contained by $V_{j\neq i}$. Therefore, V_i is not only a subgraph of $V_{j\neq i}$, but also is a prefix traversal pattern of $V_{j\neq i}$; (ii) merge-join the genes $g_i \subset g_j$: Fusion merges the genes $g_i \in V_i$ if $g_j \in V_{j\neq i}$ contains g_i, and the remaining genes $g_{k\neq i} \in V_i$ are joined to $V_{j\neq i}$ if they are not contained by other views $V_{k\neq i,j}$.

REMOVE transformation eliminates the empty-gene candidate views after a sequence of view transformations. Such candidate views can be removed as they have been contained by other views.

Example 4.3 *Figure 4.3 illustrates the view transformations of the GGA algorithm. Given a view pattern $V_P(Q_G)$ and a view set $\mathcal{V} = \{V_1, V_2, V_3\}$, GGA applies a set of transformations on the view patterns. In the FISSION*

phase, $V_P(Q_G)$ is broken down to four genes based on the articulation points $\{B, C, E\}$. Then the genes are merged to the view set in the FUSION phase. Specifically, genes 1,2,3 are merged to V_1, V_2, V_3, respectively, and the remaining gene 4 is joined to V_3 on node E. Finally, the $V_P(Q_G)$ is removed from the candidate view set and we have reduced the common parts of three graph genes of it, i.e., genes 1, 2, and 3.

Algorithm Description

The details of the GGA algorithm are given in Algorithm 3 of Article V. Given a workload Q and a candidate view set $\mathcal{V}$, it returns a subset $\mathcal{V}_s$ that is transformed from $\mathcal{V}$. In addition, two probabilities p_f and p_f are provided to perform the random fission and fusion transformations, respectively. To summarize, it goes with the following three steps:

1. When the termination condition, e.g., a timeout threshold or an iteration number, is not satisfied, GGA consecutively applies the FISSION, FUSION and, REMOVE transformations to the candidate view set to derive a new state of the view selection in each iteration.

2. Call the helper procedure *SearchAndEvaluate* to evaluate the view benefit and invoke the *DPS* function to select the views based on the dynamic programming strategy.

3. Jump to a new state with a higher benefit, otherwise, ignore the target state and continue applying other transformations starting from the previously attained state

We have proved that GGA has a property of *transformation completeness* in Lemma 2 of Article V, which guarantees that the query workload can be fully covered by any state of the candidate views.

4.5 Summary of the experimental results

We evaluated our methods in two aspects. Firstly, we studied the performance using G-View against (1) a relational-based approach, P-View [45], and (2) a native graph database, SubGraph [109]. Secondly, we compared the GGA algorithm with three competitors: Dynamic Programming Selection (DPS), Greedy-Based Selection (Greedy), and Kascade [46]. We have

used three datasets: LDBC [107], DBLP-citation [112], and Amazon [113] datasets, and designed 12 pattern queries for each datasets.

We summarize the main findings of experiments as follows:

(1) G-View achieved averagely 21x and 2x query performance speedup over P-View and Subgraph, respectively. Moreover, G-View had 2x and 5x smaller space overhead. The space-optimized approach, G-View*, further improved the overhead of G-View by 11%.

(2) GGA improved 63%, 45%, 27% of view benefit compared with Greedy, DPS, Kascade, respectively. Meanwhile, GGA had a size reduction of 59%, 59%, 50% to their view size, respectively.

(3) The GGA algorithm can converge to a steady state within a few iterations, e.g., 10, and can reach states with higher benefit.

4.6 Chapter summary

In this chapter, we proposed to utilize graph views to speed up the graph queries in the relational-based graph databases (RGDBs). We presented new methods for view creation, view evaluation and view selection. We formalized the view selection problem as a Knapsack-style problem and proposed a search-based algorithm, GGA, which explores graph view transformations to reduce the view size and optimize the overall query performance. Our results manifested that the graph views constructed by G-View can significantly improve the Gremlin query performance for RGDBs. Moreover, GGA outperformed other methods in graph view selection concerning effectiveness and efficiency.

Chapter 5

Conclusions and future work

Now this is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning.

— *Winston Churchill*

5.1 Conclusions

Multi-model data management has been an indispensable part of the advanced database management systems because of the data variety of modern applications. A multi-model database is proposed to manage the multi-model data in a unified, timely, and flexible way. Multi-model databases nowadays play important roles in data management. In the DB-engines rankings[1] as of today, there are 8 multi-model databases in the top-10, and 85 multi-model databases among 359 in total. However, despite its popularity, existing multi-model databases lack a specific benchmark and the key techniques for their query optimizers, these problems hinder the advancement of multi-model databases. This dissertation has laid down both practical and theoretical foundations to facilitate the development of multi-model databases in three aspects. In the following, we summarize the main contributions of each aspect.

5.1.1 Multi-model database benchmarking

We proposed the first benchmark system for MMDBs. It features a mixed data model, a multi-model data generator, and a set of workloads including multi-model aggregation, join, and transaction. Specifically, the mixed data model mimics a social commerce scenario [4] that includes relational,

[1] https://db-engines.com/en/ranking

graph, XML, JSON, and key-value data. The data generator is developed to provide realistic, scalable, and multi-model data for testing the performance of databases, which is implemented using Spark SQL. We particularly proposed a three-phase framework to model the customer's purchase behaviors. Furthermore, we have designed ten multi-model queries and two multi-model transactions with the consideration of both business factors, e.g., customer 360's view, and the choke point design, e.g., multi-model join ordering. To configure the queries, we formalized the problem of multi-model parameter curation and have shown the problem is computationally costly and includes an NP-hard problem. We proposed the the MJFast algorithm to efficiently choose the parameter values. We have theoretically proved that MJFast algorithm has a higher query diversity than random sampling. We evaluated the data generator and the method of parameter curation. The results have confirmed the superiority of our approaches in terms of effectiveness, efficiency, and scalability. We surveyed a variety of MMDBs and chose four representatives: ArangoDB [1], OrientDB [2], AgensGraph [3], and Spark SQL [50]. We leveraged UniBench to evaluate and compare their performance. We discussed their internal data representation, query implementation, and distributed execution strategy. We conducted the extensive experimental evaluation, and analytically reported the performance comparison and our learned lessons. Through detailed experiments, we also verified the feasibility and applicability of UniBench.

5.1.2 Multi-model join selectivity estimation

We studied a new join selectivity estimation problem for relation-tree joins. This problem has many applications including multi-model join ordering, query benchmarking, and query refinement. We formalized the estimation problem as the RTJSE problem that aims to estimate the selectivity of structural and value joins. We proposed a kernel-based estimation model called the Location-value estimation (LVE) to tackle the RTJSE problem. To the best of our knowledge, this is the first work that utilizes the KDE model to estimate the relational-tree join. We illustrated how to apply the LVE model to the simple two-way joins in detail, then applied it to more complicated cases including the twig joins. Particularly, we proposed an estimation algorithm that can efficiently obtain an estimate by avoiding duplicate estimation. Furthermore, we proposed the max-min approximation (MMA) and grid-based approximation (GBA) to improve the estimation efficiency of the LVE model. We also provided the error bounds for both approximators. We have conducted extensive experiments to compare the LVE model and its variants to the state-of-the-art methods. The results

show that our KDE model significantly outperforms them in terms of accuracy, efficiency, and scalability.

5.1.3 Multi-model view selection

We studied the problem of graph view selection in relational-based graph databases (RGDBs). We proposed an edge-induced method that automatically creates the views based on the given graph queries. We characterized the query containment as pattern containment and proposed a filtering-and-verification framework to evaluate the view benefit. In addition, we considered the multi-view evaluation and proposed a two-level minimal algorithm to search the minimal view set for an evaluated query. We formalized the view selection problem and showed that it is an NP-hard problem. We proposed the GGA algorithm to select the views into the memory under a space budget by dynamically transforming the candidate views and taking into account their efficacy in answering the queries. We conducted extensive experiments on various query workloads and datasets. The results demonstrated that G-View achieved the highest performance compared to the state of the arts, and GGA outperformed other selection methods.

5.1.4 Summary

To summarize, the present dissertation studied three key research problems for multi-model databases described in Section 1.2, namely, RQ1 for multi-model database benchmarking, RQ2 for multi-model join selectivity estimation, and RQ3 for multi-model view selection. We have made a significant amount of contributions to address these problems from both theoretical and practical aspects. We believe the methods proposed in this dissertation can also find their applications in the areas of polyglot persistence [5], multi-model databases [6], and data lakes [114, 115].

5.2 Future work

The research in this dissertation is a step forward towards providing a more advanced multi-model database. Thus, it is definitely not the end but just the start towards the ultimate goal. Exciting follow-up researches are as follows:

(1) The next generation of UniBench: it would be interesting to enhance the usability of UniBench in the following aspects: (i) to incorporate more data models to the mixed data models for covering more meaningful use cases. For instance, we may add the RDF, time-series, and

spatial-temporal data to the existing scenario. In such a case, we could integrate a new application of smart city to the scenario of social commerce; (ii) to introduce the schema evolution into UniBench. Schema evolution [116] is a common practice in database applications such as online data migration and data integration. Thus it is crucial to enable the schema evolution in UniBench to simulate more multi-model cases [117]. In the case of multi-model data, we will simulate the inter- and intra-model schema evolution where intra-model schema evolution only affects the records in a single model while inter-model schema evolution has an impact on records in multiple models [118]; (iii) to provide a micro multi-model benchmark to complement the present macro multi-model benchmark. This allows the benchmark to evaluate the performance of the single-model query individually in a unified scenario, and ultimately being able to give a holistic evaluation; (iv) to automate the process of workload generation. As the mixed data model becomes more complicated, manually designing the queries is cumbersome and infeasible. In such a case, developing a workload generator with controllable properties can greatly reduce the human effort. For instance, users may give the distribution of multi-model data, e.g., graph: 30%, JSON: 50%, table: 20, then leverage a workload generator to automatically produce a workload targeting the given distribution; (v) to introduce the metrics that can quantify the multi-model capacity of current approaches, from multi-model storage, multi-model data conversion, to multi-model query processing.

(2) A unified method for plan-level query optimization and beyond: the second promising direction is to provide a unified method for optimizing the multi-model queries. In Chapter 3, we have studied the relation-tree join and have discussed the extension of the relation-graph case. These techniques can serve as a building block towards multi-model query optimization in modern databases. The next critical step is to consider a more complicated case that involves relational, tree, and graph data simultaneously. Finally, we aim to develop a unified method to make an accurate cost estimation for a multi-model query plan. With these cost estimates, we can choose an optimal query plan from the candidate query plans, thereby optimizing the query performance. In addition to current plan-level query optimization such as join ordering, we will seek to optimize the queries in the operator level. An example of this is to reduce the operators from two data models to operators in one model. Doing so could reduce the data transmission and the logic complexity of multi-model query processing. Last but not least, we will investigate the possibility of developing a unified index for multi-model query processing [6].

(3) A hybrid method of view selection: In Chapter 4, we have studied the graph view selection in MMDBs. The next key step is to investigate a hybrid method of view selection because a multi-model query may involve operators from various data models. For instance, when a query includes both graph pattern matching and group by clause, we can combine the relational views and graph views to accelerate the query. In the hybrid cases, multi-model views may contain relational views, graph views, and JSON views simultaneously. Such hybrid cases are more complicated than the previous cases as they poses new challenges on view selection. The main challenge is that when a query can be answered by different combinations of multi-model views, finding an optimal solution is non-trivial due to the huge search space. In addition, we need to (1) investigate how to define the query containment between a multi-model query and a set of hybrid views; (2) how to estimate the cost and space of hybrid views; and (3) how to properly select the hybrid views under a space constraint, especially for the cases that hybrid views share redundant data.

(4) Autonomous multi-model databases: Recently, autonomous databases have attracted considerable attention from both academia and industry. They particularly leverage the methods of artificial intelligence (AI) to deal with numerous conventional tasks in databases (AI4DB), including knob tuning [119], cost estimation [120, 121], join order selection [122, 123], index selection [124], and view selection [41]. The results so far are optimistic: they have achieved significant improvement in many aspects of the system performance. Motivated by this trend, we plan to apply AI-enabled methods to solve the relevant problems in multi-model databases. This part can be interleaved with part (2) and (3). For example, for part (2), we can employ a deep learning model [120] to learn a cost estimator for the multi-model queries. For part (3), we may leverage a deep reinforcement learning model [41] to select the multi-model views. We believe an autonomous, self-driving, and model-agnostic multi-model database is the future to deal with the great challenges of multi-model data management, such as automatic data modeling, automatic query processing, and automatic data migration.

References

[1] ArangoDB. The Multi-Model Database for Graph and Beyond. https://www.arangodb.com/, 2020.

[2] OrientDB. The First Multi-Model Open Source NoSQL DBMS. https://www.orientdb.org/, 2020.

[3] Agensgraph. Online Transactional Multi-Model Graph Database. http://www.agensgraph.com/, 2020.

[4] Sunil Gupta, Dominique Hanssens, Bruce Hardie, Wiliam Kahn, V Kumar, Nathaniel Lin, Nalini Ravishanker, and S Sriram. Modeling Customer Lifetime Value. *Journal of service research*, 9(2):139–155, 2006.

[5] Michael Stonebraker. The Case for Polystores. SIGMOD Blog, 2015.

[6] Jiaheng Lu and Irena Holubová. Multi-Model Databases: A New Journey to Handle the Variety of Data. *ACM Comput. Surv.*, 52(3):55:1–55:38, 2019.

[7] Jiaheng Lu and Irena Holubová. Multi-Model Data Management: What's New and What's Next? In *EDBT*, 2017.

[8] Chao Zhang, Jiaheng Lu, Pengfei Xu, and Yuxing Chen. UniBench: A Benchmark for Multi-Model Database Management Systems. In *TPCTC*, pages 7–23. Springer, 2018.

[9] Chao Zhang and Jiaheng Lu. Holistic Evaluation in Multi-Model Databases Benchmarking. *Distributed and Parallel Databases*, pages 1–33, 2019.

[10] Jiaheng Lu. Towards Benchmarking Multi-Model Databases. In *CIDR*, 2017.

[11] David J. DeWitt. The Wisconsin Benchmark: Past, Present, and Future. In *The Benchmark Handbook*, pages 119–165. 1991.

[12] Transaction Processing Performance Council. TPC Benchmark C (Revision 5.11), 2010.

[13] Meikel Poess, Tilmann Rabl, Hans-Arno Jacobsen, and Brian Caufield. TPC-DI: The First Industry Benchmark for Data Integration. *PVLDB*, 7(13):1367–1378, 2014.

[14] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 Benchmark. In *ACM SIGMOD*, pages 12–21, 1993.

[15] Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: A Benchmark for XML Data Management. In *VLDB*, pages 974–985, 2002.

[16] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*, pages 143–154. ACM, 2010.

[17] Orri Erling, Alex Averbuch, Josep-Lluís Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat-Pérez, Minh-Duc Pham, and Peter A. Boncz. The LDBC Social Network Benchmark: Interactive Workload. In *SIGMOD 2015*.

[18] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. BigBench: Towards an Industry Standard Benchmark for Big Data Analytics. In *ACM SIGMOD*, 2013.

[19] Yueguo Chen, Xiongpai Qin, Haoqiong Bian, Jun Chen, Zhaoan Dong, Xiaoyong Du, Yanjie Gao, Dehai Liu, Jiaheng Lu, and Huijie Zhang. A Study of SQL-on-Hadoop Systems. In *Big Data Benchmarks, Performance Optimization, and Emerging Hardware*, pages 154–166, 2014.

[20] Martin Kiefer, Max Heimel, Sebastian Breß, and Volker Markl. Estimating Join Selectivities using Bandwidth-Optimized Kernel Density Models. *PVLDB*, 10(13):2085–2096, 2017.

[21] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How Good Are Query Optimizers, Really? *Proceedings of the VLDB Endowment*, 9(3):204–215, 2015.

[22] David Vengerov, Andre Cavalheiro Menck, Mohamed Zaït, and Sunil Chakkappen. Join Size Estimation Subject to Filter Conditions. *PVLDB*, 8(12):1530–1541, 2015.

[23] Yu Chen and Ke Yi. Two-Level Sampling for Join Size Estimation. In *SIGMOD*, pages 759–774, 2017.

[24] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. Cardinality Estimation Done Right: Index-Based Join Sampling. In *CIDR*, 2017.

[25] Cheng Luo, Zhewei Jiang, Wen-Chi Hou, Feng Yu, and Qiang Zhu. A Sampling Approach for XML Query Selectivity Estimation. In *EDBT*, pages 335–344, 2009.

[26] Ning Zhang, M Tamer Ozsu, Ashraf Aboulnaga, and Ihab F Ilyas. XSEED: Accurate and Fast Cardinality Estimation for XPATH Queries. In *ICDE*, pages 61–61, 2006.

[27] Max Heimel, Martin Kiefer, and Volker Markl. Self-Tuning, GPU-Accelerated Kernel Density Models for Multidimensional Selectivity Estimation. In *SIGMOD*, pages 1477–1492, 2015.

[28] Kostas Tzoumas, Amol Deshpande, and Christian S Jensen. Lightweight Graphical Models for Selectivity Estimation without Independence Assumptions. *PVLDB*, 4(11):852–863, 2011.

[29] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. 2019.

[30] Filippo Furfaro, Giuseppe M Mazzeo, Domenico Saccà, and Cristina Sirangelo. Compressed Hierarchical Binary Histograms for Summarizing Multi-dimensional Data. *Knowledge and information systems*, 15(3):335–380, 2008.

[31] Wei Wang, Haifeng Jiang, Hongjun Lu, and Jeffrey Xu Yu. Containment Join Size Estimation: Models and Methods. In *SIGMOD*, pages 145–156, 2003.

[32] Yuqing Wu, Jignesh M Patel, and HV Jagadish. Using Histograms to Estimate Answer Sizes for XML Queries. *Information Systems*, 28(1-2):33–59, 2003.

[33] Zhenjie Zhang, Yin Yang, Ruichu Cai, Dimitris Papadias, and Anthony K. H. Tung. Kernel-based skyline cardinality estimation. In *SIGMOD*, pages 509–522, 2009.

[34] Yan Zheng, Jeffrey Jestes, Jeff M. Phillips, and Feifei Li. Quality and Efficiency for Kernel Density Estimates in Large Data. In *SIGMOD*, pages 433–444, 2013.

[35] Oracle. Property Graph Developerâs Guide in Oracle 19c. https://www.oracle.com/a/tech/docs/property-graph-developer-guide.pdf, 2020.

[36] Yuanyuan Tian, Wen Sun, Sui Jun Tong, En Liang Xu, Mir Hamid Pirahesh, and Wei Zhao. Synergistic Graph and SQL Analytics inside IBM DB2. *PVLDB*, 12(12):1782–1785, 2019.

[37] Pieter Martin. SQLG: An Implementation of Apache TinkerPop on a RDBMS. http://sqlg.org/docs/2.0.0-SNAPSHOT/, 2020.

[38] Himanshu Gupta. Selection of Views to Materialize in A Data Warehouse. In *ICDT*, pages 98–112. Springer, 1997.

[39] Leonardo Weiss F Chaves, Erik Buchmann, Fabian Hueske, and Klemens Böhm. Towards Materialized View Selection for Distributed Databases. In *EDBT*, pages 1088–1099. ACM, 2009.

[40] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB*, volume 2000, pages 496–505, 2000.

[41] Haitao Yuan, Guoliang Li, Ling Feng, Ji Sun, and Yue Han. Automatic View Generation with Deep Learning and Reinforcement Learning. ICDE, 2020.

[42] Yasin N Silva, Paul-Ake Larson, and Jingren Zhou. Exploiting Common Subexpressions for Cloud Query Processing. In *ICDE*, pages 1337–1348, 2012.

[43] Asterios Katsifodimos, Ioana Manolescu, and Vasilis Vassalos. Materialized View Selection for XQuery Workloads. In *SIGMOD*, pages 565–576, 2012.

[44] Nan Tang, Jeffrey Xu Yu, Hao Tang, M Tamer Özsu, and Peter Boncz. Materialized View Selection in XML Databases. In *DASFAA*, pages 616–630, 2009.

[45] Wenfei Fan, Xin Wang, and Yinghui Wu. Answering Graph Pattern Queries using Views. In *ICDE*, pages 184–195. IEEE, 2014.

[46] Joana MF da Trindade, Konstantinos Karanasos, Carlo Curino, Samuel Madden, and Julian Shun. Kaskade: Graph Views for Efficient Graph Analytics. In *ICDE*, 2020.

[47] Kaiji Chen and Yongluan Zhou. Materialized View Selection in Feed Following Systems. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 442–451. IEEE, 2016.

[48] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. An In-depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases. *Proceedings of the VLDB Endowment*, 6(2):133–144, 2012.

[49] Chao Zhang et al. Parameter Curation and Data Generation for Benchmarking Multi-model Queries. In *Proceedings of the VLDB 2018 PhD Workshop*. CEUR-WS. org, 2018.

[50] Spark SQL: a Spark module for structured data processing. https://spark.apache.org/sql/, 2019.

[51] Chao Zhang and Jiaheng Lu. Selectivity estimation for relation-tree joins. In *32nd International Conference on Scientific and Statistical Database Management*, 2020.

[52] Chao Zhang, Jiaheng Lu, Qingsong Guo, Xinyong Zhang, Xiaochun Han, and Minqi Zhou. G-View: Automatic View Selection in Graph Databases. Manuscript submitted for publication. In *SSDBM 2021: 33th International Conference on Scientfic and Statistical Database Management.*, pages 1–12, March 2021.

[53] J Melton. Information technology–database languages–SQL–Part 14: XML-related specifications (SQL/XML). *ISO/IEC*, pages 9075–14, 2003.

[54] Oracle. Database JSON Developer's Guide - Part II. https://docs.oracle.com/database/122/ADJSN/store-and-manage-json-data.htm, 2018.

[55] Redis: An Open Source, In-Memory Key-Value Database. https://redis.io, 2019.

[56] CosmosDB: A Globally Distributed, Multi-Model Database. https://docs.microsoft.com/en-us/azure/cosmos-db/, 2019.

[57] DataStax: A Scalable, Cloud-Native NoSQL Database Built on Apache Cassandra. https://www.datastax.com/, 2019.

[58] DynamoDB: Fast and flexible NoSQL database service for any scale. https://aws.amazon.com/dynamodb/, 2019.

[59] Transaction Processing Performance Council. TPC Benchmark E (Revision 1.14.0), 2010.

[60] Transaction Processing Performance Council. TPC-H (Revision 2.17.3), 2017.

[61] Meikel Poess, Tilmann Rabl, and Hans-Arno Jacobsen. Analysis of TPC-DS: the First Standard Benchmark for SQL-based Big Data Systems. In *SoCC*, pages 573–585, 2017.

[62] T Lougenia Anderson, Arne J Berre, Moira Mallison, Harry H Porter, and Bruce Schneider. The Hypermodel Benchmark. In *International Conference on Extending Database Technology*, pages 317–331. Springer, 1990.

[63] Rick GG Cattell and J Skeen. Object Operations Benchmark. *ACM Transactions on Database Systems (TODS)*, 17(1):1–31, 1992.

[64] Michael J Carey, David J DeWitt, and Jeffrey F Naughton. The OO7 Benchmark. *ACM SIGMOD Record*, 22(2):12–21, 1993.

[65] Michael J Carey, David J DeWitt, Jeffrey F Naughton, Mohammad Asgarian, Paul Brown, Johannes E Gehrke, and Dhaval N Shah. The BUCKY Object-Relational Benchmark. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 135–146, 1997.

[66] Ying Guang Li, Stéphane Bressan, Gillian Dobbie, Zoé Lacroix, Mong Li Lee, Ullas Nambiar, and Bimlesh Wadhwa. XOO7: Applying OO7 Benchmark to XML Query Processing Tool. In *Proceedings of the tenth international conference on Information and knowledge management*, pages 167–174, 2001.

[67] Timo Böhme and Erhard Rahm. Multi-user Evaluation of XML Data Management Systems with XMach-1. In *Efficiency and Effectiveness of XML Tools and Techniques and Data Integration over the Web*, pages 148–159. Springer, 2002.

[68] Benjamin Bin Yao, M Tamer Ozsu, and Nitin Khandelwal. XBench Benchmark and Performance Testing of XML DBMSs. In *Proceedings. 20th International Conference on Data Engineering*, pages 621–632. IEEE, 2004.

[69] Jussi Myllymaki and James Kaufman. Dynamark: A Benchmark for Dynamic Spatial Indexing. In *International Conference on Mobile Data Management*, pages 92–105. Springer, 2003.

[70] Christian S Jensen, Dalia Tiešytė, and Nerius Tradišauskas. The COST Benchmark-Comparison and Evaluation of Spatio-Temporal Indexes. In *International Conference on Database Systems for Advanced Applications*, pages 125–140. Springer, 2006.

[71] Christian Düntgen, Thomas Behr, and Ralf Hartmut Güting. BerlinMOD: A Benchmark for Moving Object Databases. *The VLDB Journal*, 18(6):1335, 2009.

[72] Kiyoung Kim, Kyungho Jeon, Hyuck Han, Shin-gyu Kim, Hyungsoo Jung, and Heon Y Yeom. MRBench: A Benchmark for MapReduce Framework. In *2008 14th IEEE International Conference on Parallel and Distributed Systems*, pages 11–18. IEEE, 2008.

[73] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The HiBench Benchmark Suite: Characterization of the MapReduce-based Data Analysis. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, pages 41–51. IEEE, 2010.

[74] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. LinkBench: A Database Benchmark based on the Facebook Social Graph. In *ACM SIGMOD*, pages 1185–1196, 2013.

[75] Shi Qiao and Z. Meral Özsoyoglu. RBench: Application-Specific RDF Benchmarking. In *ACM SIGMOD*, 2015.

[76] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardto, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, et al. LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms. *Proceedings of the VLDB Endowment*, 9(13):1317–1328, 2016.

[77] Venelin Kotsev, Nikos Minadakis, Vassilis Papakonstantinou, Orri Erling, Irini Fundulaki, and Atanas Kiryakov. Benchmarking RDF Query Engines: The LDBC Semantic Publishing Benchmark. In Irini Fundulaki, Anastasia Krithara, Axel-Cyrille Ngonga Ngomo, and Vassiliki Rentoumi, editors, *ISWC*, volume 1700 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2016.

[78] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, et al. The Mixed Workload CH-benchmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems*, pages 1–6, 2011.

[79] Alexey Kopytov. SysBench: A System Performance Benchmark. *http://sysbench. sourceforge. net/*, 2004.

[80] Fábio Roberto Oliveira and Luis M. del Val Cura. Performance Evaluation of NoSQL Multi-Model Data Stores in Polyglot Persistence Applications. In *IDEAS*, pages 230–235, 2016.

[81] Ewa Pluciennik and Kamil Zgorzalek. The Multi-Model Databases - A Review. In *BDAS*, pages 141–152, 2017.

[82] Jure Leskovec, Lada A. Adamic, and Bernardo A. Huberman. The Dynamics of Viral Marketing. *TWEB*, 1(1):5, 2007.

[83] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Christian Bizer. DBpedia - A Large-Scale, Multilingual Knowledge Base Extracted from Wikipedia. *Semantic Web*, 6(2):167–195, 2015.

[84] Michael D. McKay, Richard J. Beckman, and William J. Conover. A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output From a Computer Code. *Technometrics*, 42(1):55–61, 2000.

[85] Edward Wadsworth. Buy Til You Die- A Walkthrough. Citeseer, 2012.

[86] Andrey Gubichev and Peter Boncz. Parameter Curation for Benchmark Queries. In *TPCTC*, pages 113–129, 2014.

[87] Hazelcast Open Source Project. https://hazelcast.com, 2019.

[88] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. *HotCloud*, 10(10-10):95, 2010.

[89] Edgar Tello-Leal, Pablo David Villarreal, Omar Chiotti, Ana Bertha Rios-Alvarado, and Ivan López-Arévalo. A Technological Solution to Provide Integrated and Process-Oriented Care Services in Healthcare Organizations. *IEEE Trans. Industrial Informatics*, 12(4):1508–1518, 2016.

[90] Mohamed Sarwat, Raha Moraffah, Mohamed F. Mokbel, and James L. Avery. Database System Support for Personalized Recommendation Applications. In *ICDE*, pages 1320–1331, 2017.

[91] Zeenat Rehena and Marijn Janssen. Towards a Framework for Context-Aware Intelligent Traffic Management System in Smart Cities. In *WWW*, pages 893–898, 2018.

[92] Noga Alon, Phillip B Gibbons, Yossi Matias, and Mario Szegedy. Tracking join and self-join sizes in limited storage. *J. Comput. Syst. Sci.*, 64(3):719–747, 2002.

[93] Cristian Estan and Jeffrey F Naughton. End-biased Samples for Join Cardinality Estimation. In *ICDE*, pages 20–20, 2006.

[94] Neoklis Polyzotis, Minos Garofalakis, and Yannis Ioannidis. Selectivity estimation for XML twigs. In *ICDE*, pages 264–275, 2004.

[95] Yuqing Wu, Jignesh M Patel, and HV Jagadish. Using Histograms to Estimate Answer Sizes for XML Queries. *Information Systems*, 28(1-2):33–59, 2003.

[96] David W Scott. *Multivariate Density Estimation: Theory, Practice, and Visualization*. John Wiley & Sons, 2015.

[97] Torsten Grust, Maurice Van Keulen, and Jens Teubner. Staircase Join: Teach a Relational DBMS to Watch Its (axis) Steps. In *VLDB*, pages 524–535. Elsevier, 2003.

[98] Venky Harinarayan, Anand Rajaraman, and Jeffrey D Ullman. Implementing Data Cubes Efficiently. In *SIGMOD Record*, volume 25, pages 205–216, 1996.

[99] Rada Chirkova and Chen Li. Materializing Views with Minimal Size to Answer Queries. In *PODS*, pages 38–48. ACM, 2003.

[100] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. Selecting Subexpressions to Materialize at Datacenter Scale. *PVLDB*, 11(7):800–812, 2018.

[101] Daniel C Zilio, Jun Rao, Sam Lightstone, Guy Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *VLDB*, pages 1087–1097, 2004.

[102] Bhushan Mandhani and Dan Suciu. Query Caching and View Selection for XML Databases. In *VLDB*, pages 469–480. VLDB Endowment, 2005.

[103] The World Wide Web Consortium. SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/, 2008.

[104] Roger Castillo and Ulf Leser. Selecting Materialized Views for RDF Data. In *International Conference on Web Engineering*, pages 126–137. Springer, 2010.

[105] The World Wide Web Consortium. XQuery 1.0: An XML Query Language (Second Edition). http://www.w3.org/TR/xquery/, 2015.

[106] François Goasdoué, Konstantinos Karanasos, Julien Leblay, and Ioana Manolescu. View Selection in Semantic Web Databases. *PVLDB*, 5(2):97–108, 2011.

[107] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. The LDBC Social Network Benchmark: Interactive Workload. In *SIGMOD*, pages 619–630. ACM, 2015.

[108] Rada Chirkova, Jun Yang, et al. Materialized Views. *Foundations and Trends® in Databases*, 4(4):295–405, 2012.

[109] Apache Tinkerpop. https://tinkerpop.apache.org/docs/3.4.4/, 2020.

[110] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (Sub) Graph Isomorphism Algorithm for Matching Large Graphs. *TPAMI*, 26(10):1367–1372, 2004.

[111] Robert Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[112] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. Arnetminer: Extraction and Mining of Academic Social Networks. In *SIGKDD*, pages 990–998, 2008.

[113] Jure Leskovec, Lada A Adamic, and Bernardo A Huberman. The Dynamics of Viral Marketing. *TWEB*, 1(1):5–es, 2007.

[114] Fatemeh Nargesian, Erkang Zhu, Renée J. Miller, Ken Q. Pu, and Patricia C. Arocena. Data Lake Management: Challenges and Opportunities. *PVLDB*, 12(12):1986–1989, 2019.

[115] Meike Klettke, Hannes Awolin, Uta Störl, Daniel Müller, and Stefanie Scherzinger. Uncovering the Evolution History of Data Lakes. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 2462–2471. IEEE, 2017.

[116] Stefanie Scherzinger, Meike Klettke, and Uta Störl. Managing Schema Evolution in NoSQL Data Stores. In Todd J. Green and Alan Schmitt, editors, *Proceedings of the 14th International Symposium on Database Programming Languages (DBPL 2013), August 30, 2013, Riva del Garda, Trento, Italy*, 2013.

[117] Mark Lukas Möller, Stefanie Scherzinger, Meike Klettke, and Uta Störl. Why It Is Time for Yet Another Schema Evolution Benchmark. In *International Conference on Advanced Information Systems Engineering*, pages 113–125. Springer, 2020.

[118] Irena Holubová, Meike Klettke, and Uta Störl. Evolution Management of Multi-model Data. In *Heterogeneous Data Management, Polystores, and Analytics for Healthcare*, pages 139–153. Springer, 2019.

[119] 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 *Proceedings of the 2019 International Conference on Management of Data*, pages 415–432, 2019.

[120] Ji Sun and Guoliang Li. An End-to-End Learning-based Cost Estimator. *Proc. VLDB Endow.*, 13(3):307–319, 2019.

[121] Ryan C. Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.*, 12(11):1705–1718, 2019.

[122] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. Reinforcement Learning with Tree-LSTM for Join Order Selection. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1297–1308. IEEE, 2020.

[123] Ji Zhang. AlphaJoin: Join Order Selection à la AlphaGo. In Ziawasch Abedjan and Katja Hose, editors, *Proceedings of the VLDB 2020 PhD Workshop co-located with the 46th International Conference on Very Large Databases (VLDB 2020), August 31 - September 4, 2020*, volume 2652 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2020.

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